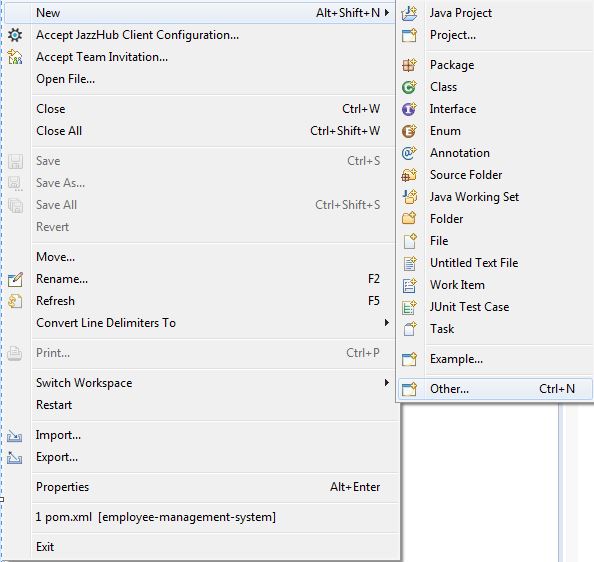
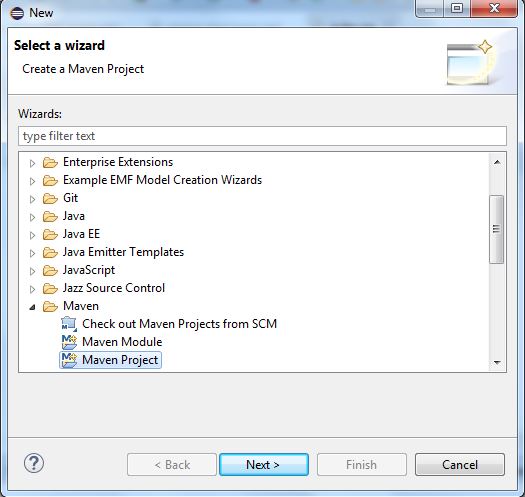
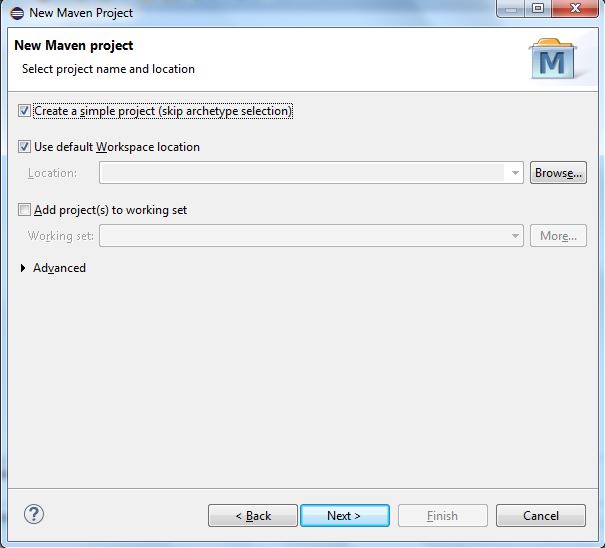
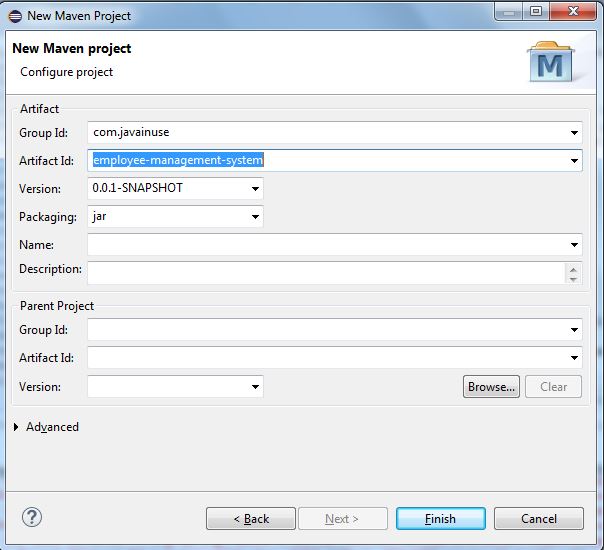
**Spring**

# Configuring the Development Environment

In this chapter we will do the project setup which will be required throughout this series. We will be creating a Maven project with minimum dependencies added to the **pom.xml** for getting Spring up and running. In Future chapters further dependencies will be added to the pom.xml as and when required.

Lets Begin

For this series we will be using  
1. Eclipse IDE(Luna)  
  
We will be creating a Maven Project.Open the Eclipse IDE.  
1. Go to File->new->other->Maven Project.Click Next Button.  
  
 

If not already selected, select Use default Workspace Location, select Create a Simple Project  
Click Next Button.  
  
 

4. Enter the values for Group Id as **com.javainuse** and Artifact Id as **employee-management-system**.  
  
5. Click Finish.  
Our Maven Web Project is now created. We will now modify the pom.xml file to add dependencies required for Spring. Currently we have only added the dependency **spring-context**, other dependencies will be added later as required. After modification our pom.xml file is as follows-

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>employee-management-system</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>4.0.5.RELEASE</version>**

**</dependency>**

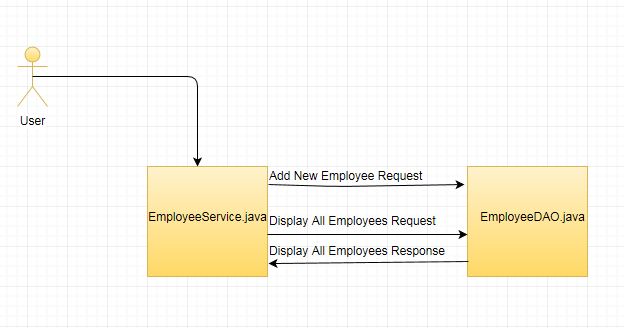
</dependencies>

</project>

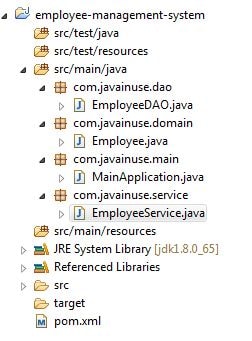
# Now run the following Maven command.

# 1. clean:install - This will download the dependencies required and build the war file. So our Eclipse is now configured. Eclipse project structure at the end of this chapter is as follows- final_basic1-5

# Traditional approach without using Dependency Injection

we saw how to create a Maven project with required dependencies which will be used throughout this series. Before we start with Spring lets understand the Traditional approach of Development. In traditional programming approach the dependencies of a class are determined and instantiated at runtime as and when they are required. Thus when need arises we instantiate the dependency class using the new keyword. Using this approach has many disadvantages which we will see later.  
Here we will implement an Employee Management system. The design will be as follows.  
  
The user can perform the functions of adding new employee, displaying all employees. We will implement this system using Traditional approach.

Lets Begin

We will create Eclipse Maven project as follows-  
  
We will first create our Domain Class Employee.java as follows

package com.javainuse.domain;

public class Employee {

private String empId;

private String name;

private String designation;

private double salary;

public Employee() {

}

public Employee(String empId, String name, String designation, double salary) {

super();

this.setEmpId(empId);

this.name = name;

this.designation = designation;

this.salary = salary;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDesignation() {

return designation;

}

public void setDesignation(String designation) {

this.designation = designation;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

}

Next we will write EmployeeService.java. The EmployeeService.java has a dependency on the EmployeeDAO.java. So we create an instance of the EmployeeDAO class in the EmployeeService.java using the new keyword. The function of the EmployeeDAO class is to store employee details in DB and fetch these details when required from DB.  
  
The EmployeeService.java has two methods.

* addNewEmployee(Employee emp)- This method calls the EmployeeDAO class method to save the new Employee details.
* getEmployees()- This method calls the EmployeeDAO class method to get the List of all employees.

package com.javainuse.service;

import java.util.List;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

public class EmployeeService {

//insert dependency using new keyword

**EmployeeDAO empDAO = new EmployeeDAO();**

public void addNewEmployee(Employee emp) {

empDAO.addNewEmployee(emp);

}

public List<Employee> getEmployees() {

return empDAO.getAllEmployees();

}

}

Currently in this chapter we will not use any DB. We will mock the EmployeeDAO.java as follows to save the employee details in an Employee ArrayList and return the same. So our

EmployeeDAO.java will be as follows-

package com.javainuse.dao;

import java.util.ArrayList;

import java.util.List;

import com.javainuse.domain.Employee;

public class EmployeeDAO {

private List<Employee> testEmployees = new ArrayList<Employee>();

public void addNewEmployee(Employee employee) {

testEmployees.add(employee);

}

public List<Employee> getAllEmployees() {

return new ArrayList<Employee>(testEmployees);

}

}

Finally we will write MainApplication.java to run our program. We create new instances of Employee and add it to the new instance of EmployeeService.

package com.javainuse.main;

import java.util.List;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

public class MainApplication {

public static void main(String[] args) {

EmployeeService empService = new EmployeeService();

Employee emp1 = new Employee("1", "Test1", "Manager", 1000);

Employee emp2 = new Employee("1", "Test2", "Manager", 1000);

Employee emp3 = new Employee("1", "Test3", "Manager", 1000);

Employee emp4 = new Employee("1", "Test4", "Manager", 1000);

Employee emp5 = new Employee("1", "Test5", "Manager", 1000);

empService.addNewEmployee(emp1);

empService.addNewEmployee(emp2);

empService.addNewEmployee(emp3);

empService.addNewEmployee(emp4);

empService.addNewEmployee(emp5);

List<Employee>employees = empService.getEmployees();

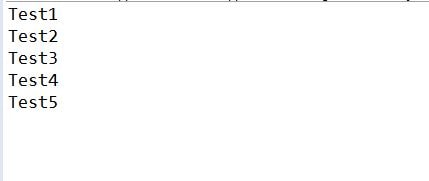
for (Employee employee : employees) {

System.out.println(employee.getName());

}

}

}

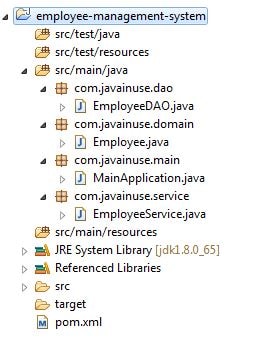
On Running this program, we get the output as  


**Disadvantages of this traditional approach-**

* The main disadvantage of this approach is the tight coupling between the classes. The EmployeeService and the  
  EmployeeDAO are tightly coupled. If the EmployeeDAO is suppose replaced by another class in future then there will be code changes required in the EmployeeService class
* The code becomes difficult to analyze as new user will have to check where all the instances are created for the dependencies.
* Testing of the code is cumbersome since we cannot mock the classes for testing purpose.
* Spring provides support for different frameworks like Hibernate which makes it easier to implement these frameworks. With traditional approach this is not the case.

# Dependency Injection using Core Java

we saw the traditional approach of instantiating the dependencies when needed using the new keyword. Inversion of Control (IOC) is a design pattern which suggests to determine, instantiate and provide the dependencies of a class beforehand. Thus, Inversion of control pattern inverts the responsibility of managing life cycle of object. IOC can be implemented using Dependency injection. In this chapter we will modify the Employee Management System implemented in previous chapter to use Dependency Injection. As we saw in the previous chapter EmployeeService class has a dependency of EmployeeDAO class, which is implemented using the new keyword. Here we will insert this dependency using Dependency Injection. This chapter makes use of Core Java to explain Dependency Injection Principles. Dependency Injection using Spring is implemented in [next chapter](https://www.javainuse.com/spring/sprbasic4).  
  
Lets Begin

We will create Eclipse Maven project as follows-  
  
Dependency injection can be implemented in java in 3 ways-  
1. Setter Injection.  
2. Constructor Injection  
3. Interface Injection  
  
Spring uses only Setter and Constructor Injection so we will only look at these. We will first have a look at setter injection.

**Dependency Injection using Setter Injection.**

For Setter Injection we will have to create the setters for all dependencies required.  
EmployeeService.java has dependency of EmployeeDAO.java.  
Unlike in [chapter2](https://www.javainuse.com/spring/sprbasic2), we will modify the EmployeeService.java to remove instantiation of EmployeeDAO.java using the new keyword.

Also, we will add setter for EmployeeDAO instance.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>employee-management-system</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

</dependencies>

</project>

Next we will insert the instance of EmployeeDAO.java in EmployeeService using Setter Injection.The code  
empService.setEmpDAO(empDAO); inserts the dependency in Employee Service using Setter dependency Injection.

package com.javainuse.main;

import java.util.List;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

public class MainApplication {

public static void main(String[] args) {

EmployeeDAO empDAO = new EmployeeDAO();

EmployeeService empService = new EmployeeService();

**//insert dependency using setter**

**empService.setEmpDAO(empDAO);**

Employee emp1 = new Employee("1", "Test1", "Manager", 1000);

Employee emp2 = new Employee("1", "Test2", "Manager", 1000);

Employee emp3 = new Employee("1", "Test3", "Manager", 1000);

Employee emp4 = new Employee("1", "Test4", "Manager", 1000);

Employee emp5 = new Employee("1", "Test5", "Manager", 1000);

empService.addNewEmployee(emp1);

empService.addNewEmployee(emp2);

empService.addNewEmployee(emp3);

empService.addNewEmployee(emp4);

empService.addNewEmployee(emp5);

List<Employee> employees = empService.getEmployees();

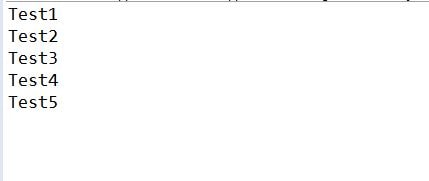
for (Employee employee : employees) {

System.out.println(employee.getName());

}

}

}

When we run the program we get the output as  


**Dependency Injection using Constructor Injection.**

For Constructor Injection we will have to create the constructor for all dependencies required.  
Unlike in [chapter2](https://www.javainuse.com/spring/sprbasic2), we will modify the EmployeeService.java to remove instantiation of EmployeeDAO.java using the new keyword. Also we will add Constructor for EmployeeService which takes the argument EmployeeDAO instance.

package com.javainuse.service;

import java.util.List;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

public class EmployeeService {

EmployeeDAO empDAO;

public EmployeeService() {

}

**//Constructor for EmployeeService for constructor injection**

**public EmployeeService(EmployeeDAO empDAO) {**

**super();**

**this.empDAO = empDAO;**

**}**

public void addNewEmployee(Employee emp) {

empDAO.addNewEmployee(emp);

}

public List<Employee> getEmployees() {

return empDAO.getAllEmployees();

}

}

Next, we will insert the instance of EmployeeDAO.java in EmployeeService using Constructor Injection. The code  
new EmployeeService(empDAO); inserts the dependency in Employee Service using Constructor Injection.

package com.javainuse.main;

import java.util.List;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

public class MainApplication {

public static void main(String[] args) {

EmployeeDAO empDAO = new EmployeeDAO();

**//insert dependency using Constructor injection**

**EmployeeService empService = new EmployeeService(empDAO);**

Employee emp1 = new Employee("1", "Test1", "Manager", 1000);

Employee emp2 = new Employee("1", "Test2", "Manager", 1000);

Employee emp3 = new Employee("1", "Test3", "Manager", 1000);

Employee emp4 = new Employee("1", "Test4", "Manager", 1000);

Employee emp5 = new Employee("1", "Test5", "Manager", 1000);

empService.addNewEmployee(emp1);

empService.addNewEmployee(emp2);

empService.addNewEmployee(emp3);

empService.addNewEmployee(emp4);

empService.addNewEmployee(emp5);

List<Employee> employees = empService.getEmployees();

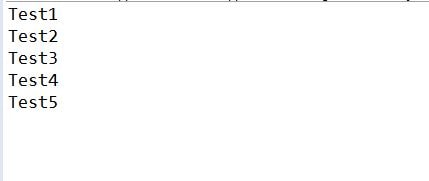
for (Employee employee : employees) {

System.out.println(employee.getName());

}

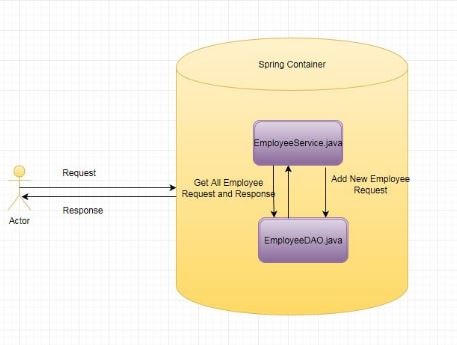
}

}

When we run the program, we get the output as  
  
**Advantages of Dependency Injection using core java-**

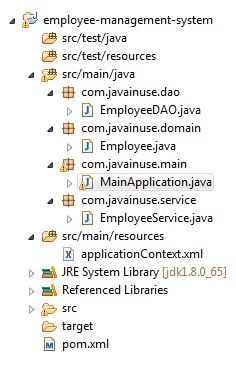
* The main advantage is the loose coupling between the classes. The EmployeeService and the EmployeeDAO are loosely coupled. If the EmployeeDAO is suppose replaced by another class in future then there will be no code changes required in the EmployeeService class
* The code becomes easy to analyse as new user will not have to check where all the instances are created for the dependencies.
* Testing of the code is easy since we can mock the classes for testing purpose.

# Dependency Injection using Spring

we saw the Dependency Injection using Core Java. We will now implement dependency injection using Spring. The approach is similar to previous chapter but now the Spring container is responsible for the life cycle of objects. Unlike in the previous chapter the Dependency Injection functionality is implemented using the Spring Configuration File.  
The design will be as follows.  
  


This chapter makes use of Spring to explain Dependency Injection Principles.

Lets Begin

We will create Eclipse Maven project as follows-  


**Dependency Injection using Spring Setter Injection**

For Setter Injection we will have to create the setters for all dependencies required. Unlike in [chapter2](https://www.javainuse.com/spring/sprbasic2), we will modify the EmployeeService.java to remove instantiation of EmployeeDAO.java using the new keyword. Also we will add setter for EmployeeDAO instance.

package com.javainuse.service;

import java.util.List;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

public class EmployeeService {

EmployeeDAO empDAO;

public void addNewEmployee(Employee emp) {

empDAO.addNewEmployee(emp);

}

public List<Employee> getEmployees() {

return empDAO.getAllEmployees();

}

**//Setter for EmployeeDAO for setter injection**

**public void setEmpDAO(EmployeeDAO empDAO) {**

**this.empDAO = empDAO;**

**}**

}

Next, we will insert the instance of EmployeeDAO.java in EmployeeService using Setter Injection.  
This is done in the configuration file named as ApplicationContext.xml as follows.  
property name="empDAO" ref=employeeDAO inserts the dependency in Employee Service using Setter Injection for above setter.

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

**<!-- Create bean of type EmployeeService class and insert dependency of EmployeeDAO class in it using Setter Dependency Injection -->**

<bean id="employeeService" class="com.javainuse.service.EmployeeService">

**<property name="empDAO" ref="employeeDAO" />**

</bean>

**<!-- Create bean of type EmployeeDAO class -->**

<bean id="employeeDAO" class="com.javainuse.dao.EmployeeDAO" />

</beans>

Thus, the classes are configured as beans when using Spring. These beans are then retrieved from the Spring container as follows.

package com.javainuse.main;

import java.util.List;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

public class MainApplication {

public static void main(String[] args) {

**//Load the Spring container using the Spring Configuration File**

ClassPathXmlApplicationContext container = new ClassPathXmlApplicationContext(

"application.xml");

**//Retrieve the Bean from the Spring Conatiner**

**EmployeeService empService = container.getBean(EmployeeService.class);**

Employee emp1 = new Employee("1", "Test1", "Manager", 1000);

Employee emp2 = new Employee("1", "Test2", "Manager", 1000);

Employee emp3 = new Employee("1", "Test3", "Manager", 1000);

Employee emp4 = new Employee("1", "Test4", "Manager", 1000);

Employee emp5 = new Employee("1", "Test5", "Manager", 1000);

empService.addNewEmployee(emp1);

empService.addNewEmployee(emp2);

empService.addNewEmployee(emp3);

empService.addNewEmployee(emp4);

empService.addNewEmployee(emp5);

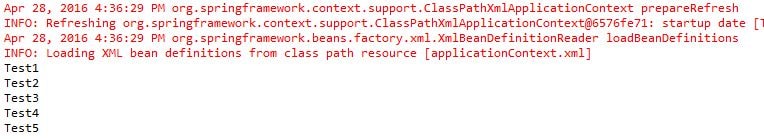
List<Employee> employees = empService.getEmployees();

for (Employee employee : employees) {

System.out.println(employee.getName());

}

}



When we run the program, we get the output as above

**Dependency Injection using Spring Constructor Injection**

For Constructor Injection we will have to create the constructor for all dependencies required.  
Unlike in [chapter2](https://www.javainuse.com/spring/sprbasic2), We will modify the EmployeeService.java to remove instantiation of EmployeeDAO.java using the new keyword. Also we will add Constructor for EmployeeService which takes the argument EmployeeDAO instance.

package com.javainuse.service;

import java.util.List;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

public class EmployeeService {

EmployeeDAO empDAO;

public EmployeeService() {

}

**//Constructor for EmployeeService for constructor injection**

**public EmployeeService(EmployeeDAO empDAO) {**

**super();**

**this.empDAO = empDAO;**

**}**

public void addNewEmployee(Employee emp) {

empDAO.addNewEmployee(emp);

}

public List<Employee> getEmployees() {

return empDAO.getAllEmployees();

}

}

Next we will insert the instance of EmployeeDAO.java in EmployeeService using Constructor Injection. This is done in the configuration file named ApplicationContext.xml as follows.  
constructor-arg ref="employeeDAO" inserts the dependency in Employee Service using Setter Injection for the above constructor.

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

**<!-- Create bean of type EmployeeService class and insert dependency of EmployeeDAO class in it using Constructor Dependency Injection -->**

<bean id="employeeService" class="com.javainuse.service.EmployeeService">

**<constructor-arg ref="employeeDAO" />**

</bean>

<bean id="employeeDAO" class="com.javainuse.dao.EmployeeDAO" />

</beans>

Thus, the classes are configured as beans when using Spring. These beans are then retrieved from the Spring container as follows.

package com.javainuse.main;

import java.util.List;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.javainuse.dao.EmployeeDAO;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

public class MainApplication {

public static void main(String[] args) {

// Load the Spring conatiner using the Spring Configuration File

**ClassPathXmlApplicationContext container = new ClassPathXmlApplicationContext(**

**"applicationContext.xml");**

EmployeeService empService = container.getBean(EmployeeService.class);

Employee emp1 = new Employee("1", "Test1", "Manager", 1000);

Employee emp2 = new Employee("1", "Test2", "Manager", 1000);

Employee emp3 = new Employee("1", "Test3", "Manager", 1000);

Employee emp4 = new Employee("1", "Test4", "Manager", 1000);

Employee emp5 = new Employee("1", "Test5", "Manager", 1000);

empService.addNewEmployee(emp1);

empService.addNewEmployee(emp2);

empService.addNewEmployee(emp3);

empService.addNewEmployee(emp4);

empService.addNewEmployee(emp5);

List<Employee> employees = empService.getEmployees();

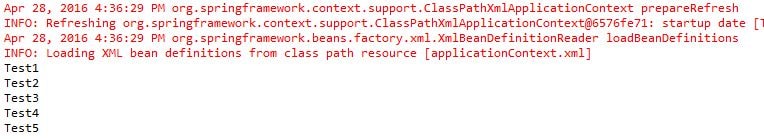
for (Employee employee : employees) {

System.out.println(employee.getName());

}

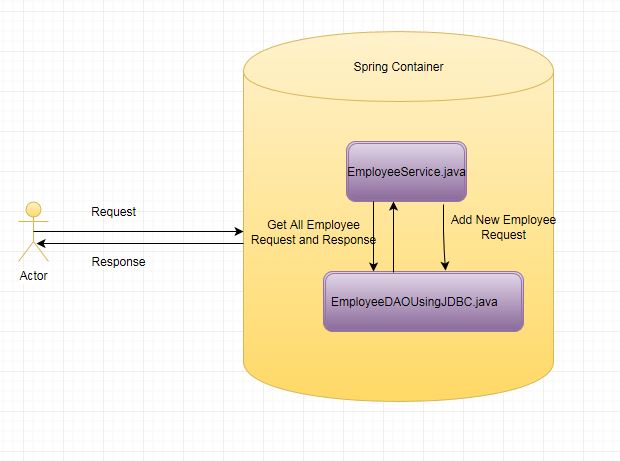
}

}

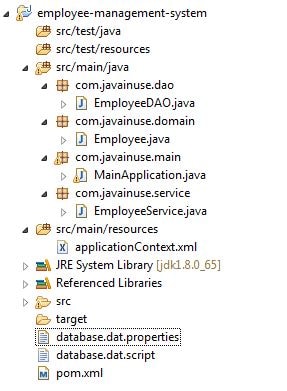
When we run the program, we get the output as  
  
Advantages of Dependency Injection using Spring-

* The main advantage is the loose coupling between the classes. The EmployeeService and the EmployeeDAO are loosely coupled. If the EmployeeDAO is suppose replaced by another class in future then there will be no code changes required in the EmployeeService class
* The code becomes easy to analyze as new user will not have to check where all the instances are created for the dependencies.
* Testing of the code is easy since we can mock the classes for testing purpose.
* Spring controls the lifecycle of the objects.
* Spring has support for many third-party frameworks like Hibernate, Apache Camel etc. Thus, these can be easily integrated using Spring.

# Implement Traditional JDBC

we implemented Dependency injection using Spring. Till now we had mocked the DAO class to represent Employee data. We will now fetch this employee information from the Database. In this chapter we implement the traditional JDBC in EmployeeDAO.java to store and fetch Employee information. In future chapters we will implement the Spring JDBC to understand its advantages over the Traditional JDBC approach. We make use of HSQLDB for database. The design will be as follows.  


Lets Begin

We will create Eclipse Maven project as follows-  
  
First modify the pom.xml to include dependency hsqldb required for HSQLDB. So our POM will be as follows

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>employee-management-system</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

**<dependency>**

**<groupId>hsqldb</groupId>**

**<artifactId>hsqldb</artifactId>**

**<version>1.8.0.10</version>**

**</dependency>**

</dependencies>

</project>

Next, we modify the EmployeeDAO.java to implement the traditional JDBC for various employee operations like saving Employee Information, retrieving employee information from DB etc. In the EmployeeDAO.java we have added a constructor. This constructor calls a method CreateEmpoyeeTable. The CreateEmpoyeeTable method creates a table named Employee in the DB. This table is created only the first time EmployeeDAO is loaded. If the table already exists in the DB then no new table is created.For adding and retrieving Employee information we have used traditional JDBC. The remaining classes will be the same as in previous chapter.

We will now modify the EmployeeDAO.java to make DataBase connections to fetch data from the DataBase.

package com.javainuse.dao;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

import java.util.List;

import com.javainuse.domain.Employee;

public class EmployeeDAO {

//Create Employee Table in constructor.

public EmployeeDAO() {

try {

Class.forName("org.hsqldb.jdbcDriver");

CreateEmpoyeeTable();

}

catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

private static void CreateEmpoyeeTable() {

try {

Connection con = null;

try {

con = DriverManager.getConnection("jdbc:hsqldb:file:database.dat;shutdown=true", "sa", "");

Statement stmt = con.createStatement();

stmt.executeUpdate(

"create table Employee(empId VARCHAR(20), name VARCHAR(50), designation VARCHAR(50),salary VARCHAR(50))");

System.out.println("Created new table Employee");

}

finally {

if (con != null)

con.close();

}

}

catch (SQLException e) {

System.out.println("Employee table has already been created...");

}

}

public void addNewEmployee(Employee employee) {

try {

Connection con = null;

PreparedStatement insertEmployee = null;

try {

con = DriverManager.getConnection("jdbc:hsqldb:file:database.dat;shutdown=true", "sa", "");

insertEmployee =

con.prepareStatement("insert into Employee (empid, name, designation) values (?, ?, ?)");

insertEmployee.setString(1, employee.getEmpId());

insertEmployee.setString(2, employee.getName());

insertEmployee.setString(3, employee.getDesignation());

insertEmployee.executeUpdate();

System.out.println("Created new Employee");

}

finally {

if (con != null)

con.close();

}

}

catch (SQLException e) {

System.out.println("Exception Occured");

}

}

public List<Employee> getAllEmployees() {

List<Employee> results = new ArrayList<Employee>();

try {

Connection con = null;

PreparedStatement retrieveBooks = null;

ResultSet rs = null;

try {

con = DriverManager

.getConnection(

"jdbc:hsqldb:file:database.dat;shutdown=true",

"sa", "");

retrieveBooks = con.prepareStatement("select \* from Employee");

rs = retrieveBooks.executeQuery();

while (rs.next()) {

String empId = rs.getString(1);

String name = rs.getString(2);

String designation = rs.getString(3);

double salary = 0;

Employee nextBook = new Employee(empId, name, designation,

salary);

results.add(nextBook);

}

} finally {

if (rs != null)

rs.close();

if (con != null)

con.close();

}

} catch (SQLException e) {

System.out.println("Exception Occured");

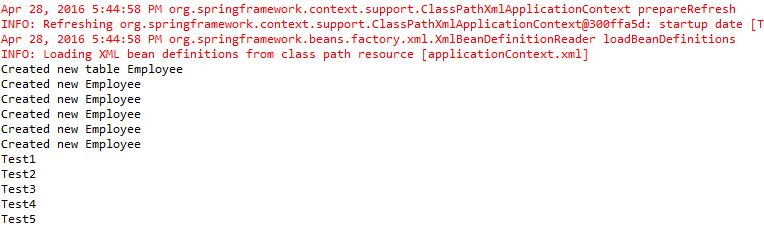
}

return results;

}

}

When we run the program, we get the output as



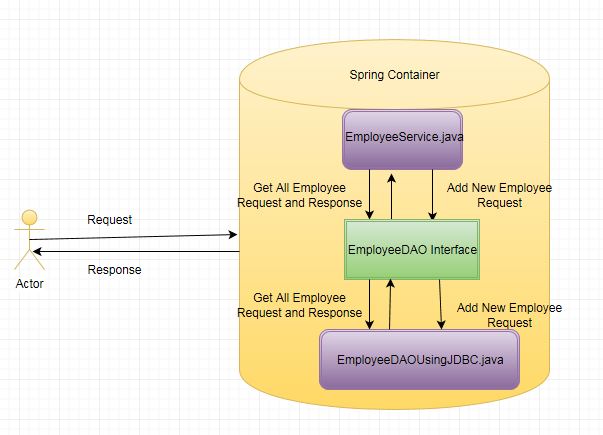
**Disadvantages of Traditional JDBC-**

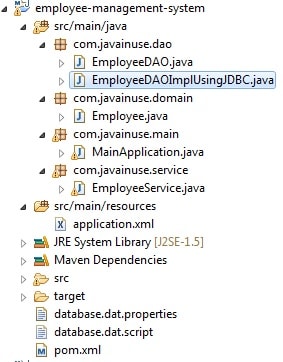
* The main disadvantage of this approach is the amount of repetitive code we have to write to accomplish DB connection. Such repetitive code is known as boilerplate code. For example, for both addEmployee and getAllEmployees method we have to write the boilerplate code to get the connection. If in future another method suppose deleteEmployee is added then this boilerplate code has to be repeated
* The error handling is more repetitive and complex. JDBC uses the rather Uninformative SQLException. JDBC has no exception hierarchy.
* Care has to be taken to release the resources after the DB function has been performed. We handle this code in the final block.

# Programming to Interface concept in Spring

we implemented Traditional JDBC in Spring. We also saw the disadvantages of this approach. We will now implement Spring-JDBC for storing and retrieving employee information from the Database. But before we do that, lets understand the concept of Programming to Interface concept. In this chapter we will implement the Programming to interface using Spring Now suppose if we have to replace the EmployeeDAO.java class which uses Traditional JDBC with another class named EmployeeDAOUsingSpringJDBC.java which uses Spring JDBC.

This will also involve changing our EmployeeService.java class to replace the instance of EmployeeDAO.java with EmployeeDAOUsingSpringJDBC.java. So there is still some tight coupling between the EmployeeService and EmployeeDAO.java class. In future if we implement EmployeeDAOUsingHibernate.java then again EmployeeService.java will have to be changed.

Now this is not good coding practice and may lead to problems especially if the code has many Service classes. This issue is resolved by using Programming to Interface in Spring. Lets see how this is implemented. The design will be as follows.  
  
  
Lets Begin

We will create Eclipse Maven project as follows-  
  
We will implement an additional layer between the Service and the DAO layer. This additional layer will be an interface. The Service class will always reference this interface to interact with the DAO layer which implements this interface. If in future we have a new DAO implementation there will be no change in the Service layer or the DAO interface. We will only have to make some small changes in the configuration file application.xml. So lets begin this implementation. First rename the EmployeeDAO.java having the traditional JDBC code to EmployeeDAOImplUsingJDBC.java. Next create a interface named as EmployeeDAO.java having the add and getAllEmployees methods as follows.

package com.javainuse.dao;

import java.util.List;

import com.javainuse.domain.Employee;

public interface EmployeeDAO {

public void addNewEmployee(Employee employee);

public List<Employee> getAllEmployees();

}

Modify the EmployeeDAOUsingJDBC.java to implement the new EmployeeDAO.java interface. It already has the interface methods implemented.

package com.javainuse.dao;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

import java.util.List;

import com.javainuse.domain.Employee;

public class EmployeeDAOImplUsingJDBC implements EmployeeDAO {

//Create Employee Table in constructor.

public EmployeeDAOImplUsingJDBC() {

try {

Class.forName("org.hsqldb.jdbcDriver");

createEmployeeTable();

}

catch (ClassNotFoundException e) {

System.out.println("Exception occured in DAO constructor");

}

}

private static void createEmployeeTable() {

try {

Connection con = null;

try {

con = DriverManager.getConnection("jdbc:hsqldb:file:database.dat;shutdown=true", "sa", "");

Statement stmt = con.createStatement();

stmt.executeUpdate(

"create table Employee(empId VARCHAR(20), name VARCHAR(50), designation VARCHAR(50),salary VARCHAR(50))");

System.out.println("Created new table Employee");

}

finally {

if (con != null)

con.close();

}

}

catch (SQLException e) {

System.out.println("Employee table has already been created...");

}

}

@Override

public void addNewEmployee(Employee employee) {

try {

Connection con = null;

PreparedStatement insertEmployee = null;

try {

con = DriverManager.getConnection("jdbc:hsqldb:file:database.dat;shutdown=true", "sa", "");

insertEmployee =

con.prepareStatement("insert into Employee (empid, name, designation) values (?, ?, ?)");

insertEmployee.setString(1, employee.getEmpId());

insertEmployee.setString(2, employee.getName());

insertEmployee.setString(3, employee.getDesignation());

insertEmployee.executeUpdate();

System.out.println("Created new Employee");

}

finally {

if (con != null)

con.close();

}

}

catch (SQLException e) {

System.out.println("Exception Occured while adding new Employee");

}

}

@Override

public List<Employee> getAllEmployees() {

List<Employee> results = new ArrayList<Employee>();

try {

Connection con = null;

PreparedStatement retrieveBooks = null;

ResultSet rs = null;

try {

con = DriverManager.getConnection("jdbc:hsqldb:file:database.dat;shutdown=true", "sa", "");

retrieveBooks = con.prepareStatement("select \* from Employee");

rs = retrieveBooks.executeQuery();

while (rs.next()) {

String empId = rs.getString(1);

String name = rs.getString(2);

String designation = rs.getString(3);

double salary = 0;

Employee emp = new Employee(empId, name, designation, salary);

results.add(emp);

}

}

finally {

if (rs != null)

rs.close();

if (con != null)

con.close();

}

}

catch (SQLException e) {

System.out.println("Exception Occured while retrieving employees");

}

return results;

}

}

Our EmployeeService.java will remain unchanged. The only difference will be that EmployeeDAO will now refer to the  
EmployeeDAO.java interface.

We will now modify the application.xml. Now the employeeDAO will refer to

EmployeeDAOImplUsingJDBC.java

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

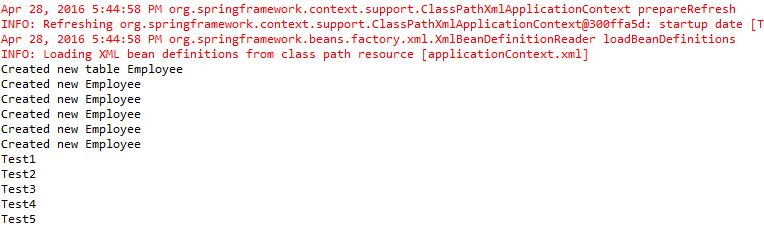
<bean id="employeeService" class="com.javainuse.service.EmployeeService">

<property name="empDAO" ref="employeeDAO" />

</bean>

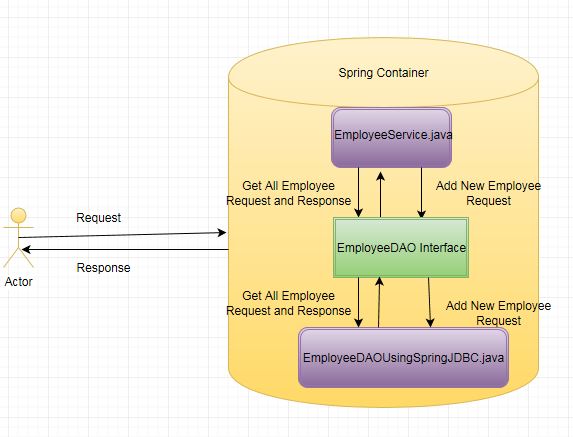
<bean id="employeeDAO" class="com.javainuse.dao.EmployeeDAOImplUsingJDBC" />

</beans>

Now we are done with the coding changes. If we run the MainApplication.java we get the output as.  
  
  
So to implement the Programming to interface in Spring have added the EmployeeDAO.java interface between the EmployeeService.java and the EmployeeDAOUsingJDBC.java. In the next chapter we will add the EmployeeDAOUsingSpringJDBC.java. Now we will only have to make change to the application.xml  
to use this DAO class.

# Implement SpringJDBC

we implemented Traditional JDBC in Spring. We also implemented the Programming to Interface concept in previous chapter. In this chapter we will implement Spring JDBC .The design will be as follows.



Lets Begin

# We will create Eclipse Maven project as follows-

# basic2-7

# First modify the pom.xml to include dependency spring-jdbc So our POM will be as follows

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>employee-management-system</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management-system Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.2.5.RELEASE</version>

</dependency>

<dependency>

<groupId>hsqldb</groupId>

<artifactId>hsqldb</artifactId>

<version>1.8.0.10</version>

</dependency>

</dependencies>

<build>

<finalName>employee-management-system</finalName>

</build>

</project>

Next we create **Spring JDBCTemplate** in the configuration file. This template is inserted in our new DAO class EmployeeDAOImplUsingSpringJDBC using constructor dependency injection. Our Configuration class and the new DAO implementation class will be as follows-

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean id="employeeService" class="com.javainuse.service.EmployeeService">

<property name="empDAO" ref="employeeDAO" />

</bean>

<bean id="employeeDAO" class="com.javainuse.dao.EmployeeDAOImplUsingSpringJDBC">

<constructor-arg ref="jdbcTemplate" />

</bean>

<bean id="dataSource"

class="org.springframework.jdbc.datasource.SimpleDriverDataSource">

<property name="driverClass" value="org.hsqldb.jdbcDriver" />

<property name="url" value="jdbc:hsqldb:file:database.dat;shutdown=true" />

<property name="username" value="sa" />

<property name="password" value="" />

</bean>

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

<constructor-arg ref="dataSource" />

</bean>

</beans>

package com.javainuse.dao;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

import java.util.List;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import com.javainuse.domain.Employee;

public class EmployeeDAOImplUsingSpringJDBC implements EmployeeDAO {

JdbcTemplate jdbcTemplate;

//SpringJdbcTemplate inserted using constructor injection.

public EmployeeDAOImplUsingSpringJDBC(JdbcTemplate jdbcTemplate) {

try {

this.jdbcTemplate = jdbcTemplate;

Class.forName("org.hsqldb.jdbcDriver");

createEmployeeTable();

}

catch (ClassNotFoundException e) {

System.out.println("Exception occured in DAO constructor");

}

}

private void createEmployeeTable() {

try {

jdbcTemplate.update(

"create table Employee(empId VARCHAR(20), name VARCHAR(50), designation VARCHAR(50),salary VARCHAR(50))");

}

catch (Exception e) {

System.out.println("Employee table has already been created...");

}

}

@Override

public void addNewEmployee(Employee employee) {

jdbcTemplate.update("insert into Employee (empid, name, designation) values (?, ?, ?)",

employee.getEmpId(), employee.getName(), employee.getDesignation());

}

@Override

public List getAllEmployees() {

return jdbcTemplate.query("select \* from Employee", new EmployeeMapper());

}

}

//implement Spring RowMapper.

class EmployeeMapper implements RowMapper {

public Employee mapRow(ResultSet rs, int rowNumber) throws SQLException {

String empId = rs.getString(1);

String name = rs.getString(2);

String designation = rs.getString(3);

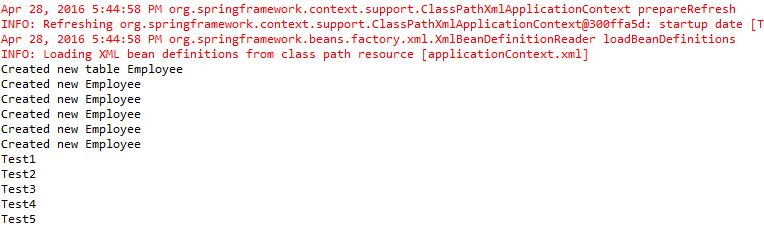
double salary = 0;

Employee emp = new Employee(empId, name, designation, salary);

return emp;

}

}

Now we are done with the coding changes. If we run the MainApplication.java we get the output.  
  


Spring Profiles

Spring Profiles allows users to register beans depending on the profile(dev, test, prod etc).  
So when the application is running in DEV only certain beans can be loaded and when in PROD certain other beans can be loaded.

Lets Begin

For example suppose a Developer wants to use different database configurations for different environments like Dev, Prod etc.  
For this the developer would have created a DAO class and inserted the datasource object as below

public class BusinessDAO {

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

}

Using Traditional Spring features without profile the developer needs to configure the database config as follows

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName">

<value></value>

</property>

<property name="url">

<value></value>

</property>

<property name="username">

<value></value>

</property>

<property name="password">

<value></value>

</property>

</bean>

Then using the PropertyPlaceholderConfigurer the developer could load either the db-dev.properties or db-prod.properties.

This approach is fine if the user wants to load exactly the same database configuration with different values in all environments.

The problem arises when the developer wants to have different DB configurations in different environments.  
So suppose in dev environment the developer wants the following datasource config –

<bean id="dataSource" **class="org.springframework.jdbc.datasource.SimpleDriverDataSource"**>

<property name="driverClassName">

<value></value>

</property>

<property name="url">

<value></value>

</property>

<property name="username">

<value></value>

</property>

<property name="password">

<value></value>

</property>

</bean>

but in prod environment wants the datasource config as follows-

<bean id="dataSource" **class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>

<property name="driverClassName">

<value></value>

</property>

<property name="url">

<value></value>

</property>

<property name="username">

<value></value>

</property>

<property name="password">

<value></value>

</property>

</bean>

**As the developer wants different configurations based on environment only using propertyplaceholder wont suffice and spring profiles should be used.**  
**Spring profiles using Spring XML-**

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd">

**<!--Profile for the Dev Environment-->**

<beans profile="dev">

<bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">

<property name="driverClassName">

<value></value>

</property>

<property name="url">

<value></value>

</property>

<property name="username">

<value></value>

</property>

<property name="password">

<value></value>

</property>

</bean>

<context:property-placeholder location="dev-db.properties" ignore-unresolvable="true"/>

</beans>

**<!--Profile for the Prod Environment-->**

<beans profile="prod">

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName">

<value></value>

</property>

<property name="url">

<value></value>

</property>

<property name="username">

<value></value>

</property>

<property name="password">

<value></value>

</property>

</bean>

<context:property-placeholder location="dev-prod.properties"

ignore-unresolvable="true"/>

</beans>

</beans>

**Spring Profiles using Java Config-**

@Configuration

**@Profile("dev")**

public class DevDBConfig {

@Bean

public DataSource devDataSource() {

{

//Dev based config

return dataSource;

}

}

@Configuration

**@Profile("prod")**

public class ProdDBConfig {

@Bean

public DataSource devDataSource() {

{

//Prod based config

return dataSource;

}

}

**Enabling Profile-**  
Now that multiple profiles have been created, how to select the required profile.  
This can be done in following ways

* Using Spring context environment.  
  **ctx.getEnvironment().setActiveProfiles("prod");**
* Using system property  
  **System.setProperty("spring.profiles.active", "dev");**
* Another way to change the profile is to pass a system parameter at run time  
  **-Dspring.profiles.active="prod"**
* Enabling profile in web.xml

<context-param>

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

<param-value>prod</param-value>

</context-param>

# Design Patterns used in Spring Framework

Following are the design patterns used in Spring Framework

**MVC Pattern**

MVC Design Pattern is a software design that separates the following components of a system or subsystem:

* **Model -** Data about the state of the application or its components. May include routines for modification or access.
* **View -** An interpretation of the data (model). This is only limited to a visual representation, but could be audio, derived information (e.g. statistics piped into another model object), etc. Furthermore, a single model may have multiple views.
* **Control -** Handles external input to the system invoking modifications on the model. The control/view may be closely related (in the case of a UI). However, other external input (such as network commands), may be processed which are completely independent of the view.

**Proxy Pattern**

Spring uses either JDK proxies (preferred wheneven the proxied target implements at least one interface) or CGLIB proxies (if the target object does not implement any interfaces) to create the proxy for a given target bean. Unless configured to do otherwise, Spring AOP performs run-time weaving Suppose we want to log every method entry and exit. This can be achieved by writing log statements in every method at the start and end. But this will require lot of code work. There are various such tasks like Security which need to be applied across all methods or classes. These are known as cross cutting concerns.AOP addresses the problem of cross-cutting concerns, which would be any kind of code that is repeated in different methods and cannot normally be completely refactored into its own module, like with logging or verification.

**Factory Pattern**

This pattern is used by spring to load beans using BeanFactory and Application context.

# Singleton Pattern

# Beans defined in spring config files are singletons by default. A singleton bean in Spring and the singleton pattern are quite different. Singleton pattern says that one and only one instance of a particular class will ever be created per class loader. The scope of a Spring singleton is described as "per container per bean". It is the scope of bean definition to a single object instance per Spring IoC container. The default scope in Spring is Singleton.

# Template method Pattern

# Template method design pattern is to define an algorithm as skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm is preserved by the parent class. These are used extensively to deal with boilerplate repeated code

# FrontController Pattern

# Front Controller is a controller pattern which provides a centralized controller for managing requests. Each client request must go through and be processed by the Front Controller first, no exceptions. All incoming data is delegated to front controller first. Useful for when your application has multiple entry points which you want to centralize through a single point for standardized processing. Spring implements this design pattern using DispatcherServlet, to dispatch incoming requests to the correct controllers.

# View Helper Pattern

# View Helper arranges view components for the user and delegates processing to other business components so the view component doesn't have to contain any processing logic other than logic to present views. Spring makes use of custom JSP tags etc to separate code from presentation in views.

# Prototype Pattern

# The Prototype pattern is known as a creational pattern, as it is used to construct objects such that they can be decoupled from their implementing systems. It creates objects based on a template of an existing object through cloning.

# DI/IOC Pattern

# Dependency Injection/Inversion of Control design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement dependency injection in java to move the dependency resolution from compile-time to runtime.

# Spring MVC

# Configuring the Development Environment

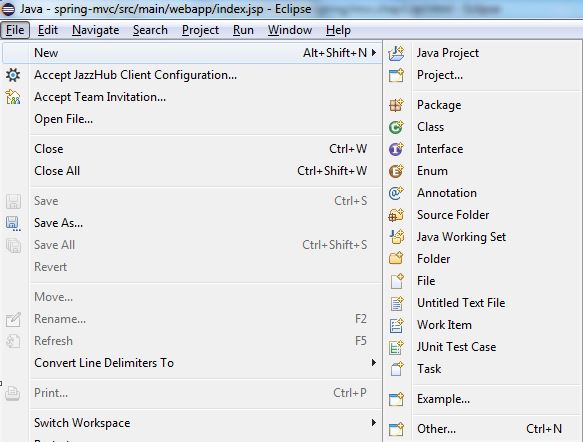
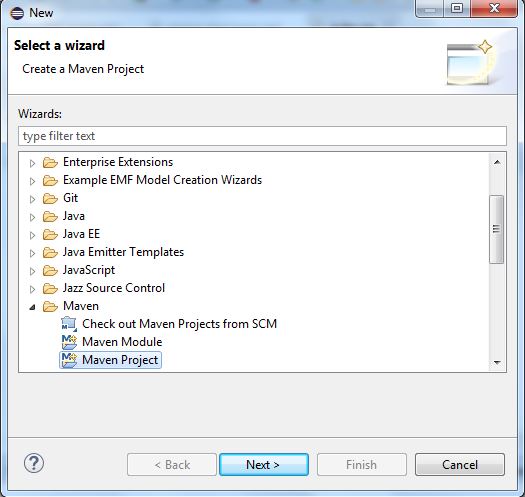
Overview

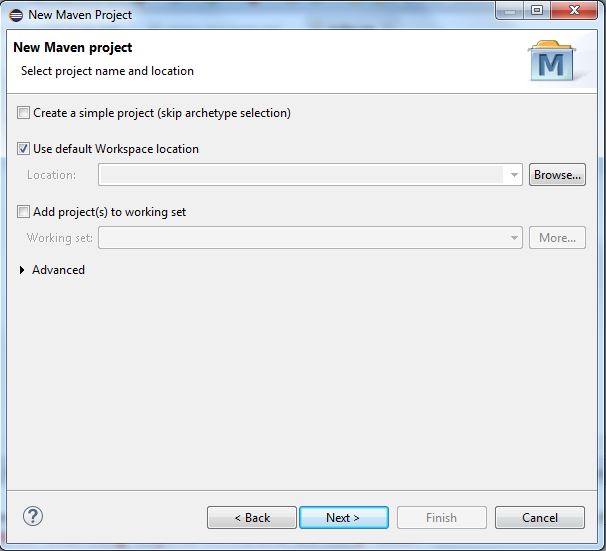
In this chapter we will do the project setup which will be required throughout this series.  
We will be creating a Maven project with minimum dependencies added to POM for getting Spring MVC up and running.  
In Future chapters further dependencies will be added to the POM.xml as and when required.

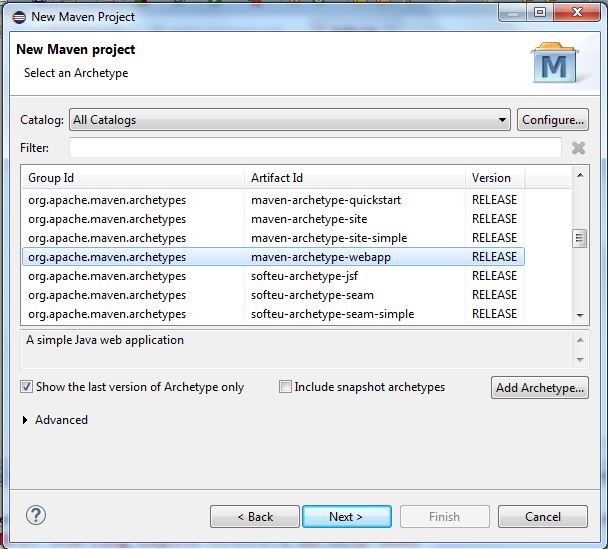
Lets Begin

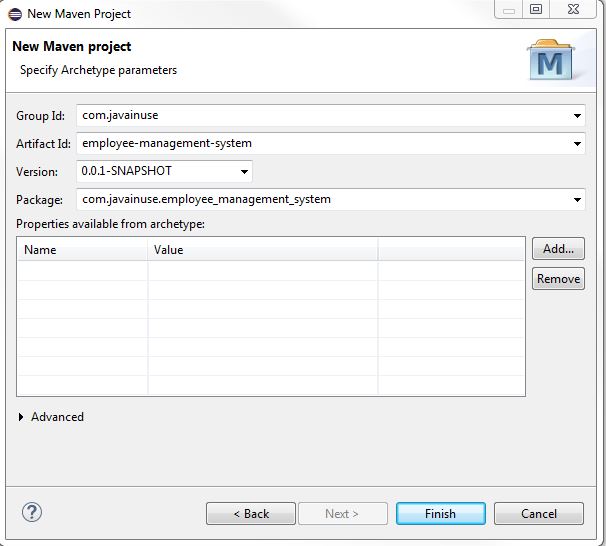
For this series we will be using  
1. Eclipse IDE(Luna)  
2. Apache Tomcat(version 7)  
  
We will be creating a Maven Project and also configuring Tomact in POM for deployment.Open the Eclipse IDE.

1. Go to File->new->other->Maven Project. Click Next Button.

2. If not already selected, select Use default Location. Click Next Button.  


3. Select Archtype as maven-archtype-webapp. Click Next Button.  


4. Enter the values for Group Id as com.javainuse and Artifact Id as employee-management-system.  
  
5. Click Finish  
  
Our Maven Web Project is now created. We will now modify the pom.xml file as follows-  
1. Add dependencies required for Spring MVC.  
2. Add the build plugins required for maven and tomcat configurations.  
  
After modification our pom.xml file is as follows-

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>employee-management-system</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management-system Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

</dependencies>

<build>

<finalName>employee-management-system</finalName>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

</plugin>

</plugins>

</build>

</project>

# Here we have configured the tomcat in the pom itself. Now run the following two Maven commands in sequence 1. clean:install - This will download the dependencies required and build the war file. 2. tomcat:run - This will start the Tomcat and install the war file in Tomcat container. In the console you will see the deployment details as follows mvc1-9

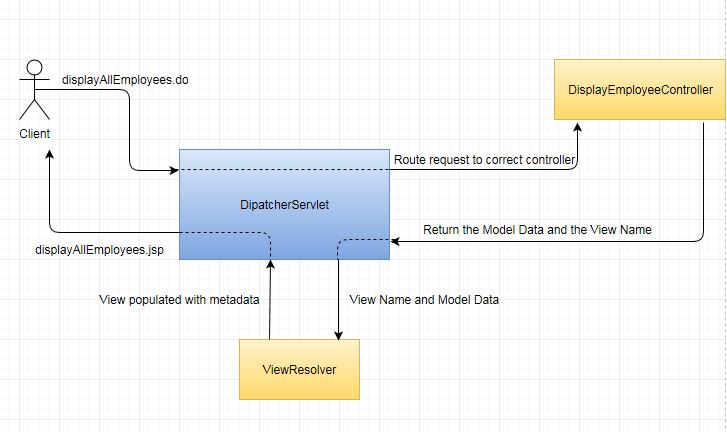
# After running these commands open the browser and go to http://localhost:8080/employee-management-system mvc1-10

# So, our Eclipse and Tomcat are now configured. Eclipse project structure at the end of this chapter is as follows-

# final-mvc1-6

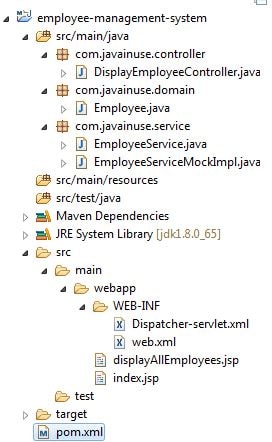
# Creating our first MVC program

Overview

Before we start writing our first Spring MVC program, lets understand how this is implemented in Spring.  
  
Previously when using JSP Servlets, one had to configure a Servlet for each incoming request. In Spring MVC this is not the case, we do not have to configure any servlet. Spring MVC makes use of a pre-configured servlet called DispatcherServlet. We only make use of this DispatcherServlet to route the incoming request to the correct Controller and View and return it to the user.  
  
Lets take an example. Here we will be designing an Employee Management System. Here the user requests to display all Employees information.  


Here the User requests to show all the Employees information in the system. The client browser sends the displayAllEmployees.do request. This request is received by the DispatcherServlet. The DispatcherServlet then routes this request to the correct Controller, in this case the DisplayEmployeeController.java. The DisplayEmployeeController.java does the actual process of retrieving the Employee info. After this it returns to the DispatcherServlet the Model Data which represents all the Employee info retrieved and the view details which will show this model data. The View name returned in our case is displayEmployees.jsp. The ServletDispatcher then routes this data to the ViewResolver. The ViewResolver then selects the correct View, populates it with the Model Data and returns it to DispatcherServlet. The DispatcherServlet then returns this populated View to the client.

Lets Begin

In the [previous chapter](https://www.javainuse.com/spring/mvcchap1) we configured the Development Environment required for writing Spring MVC.  
We will create Eclipse Maven project as follows-  
  
  
We will have to add one more additional dependency to the pom.xml configured in the previous chapter. We add the jstl dependency required for displaying elements in the jsp page. So our pom will be-

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>employee-management-system</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management-system Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

**<dependency>**

**<groupId>javax.servlet</groupId>**

**<artifactId>jstl</artifactId>**

**<version>1.2</version>**

**</dependency>**

</dependencies>

<build>

<finalName>employee-management-system</finalName>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

</plugin>

</plugins>

</build>

</project>

Next we configure DispatcherServlet in web.xml.Here all the incoming request with .do extension will now be routed by the DispatcherServlet to the configured Controllers.-

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

version="2.5">

**<!-- Configure the Disptcher Servlet -->**

<servlet>

<servlet-name>Dispatcher</servlet-name>

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

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

</servlet>

**<!-- Filter the incoming requests for the .do pattern -->**

<servlet-mapping>

<servlet-name>Dispatcher</servlet-name>

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

</servlet-mapping>

</web-app>

Next we will write Application Context to configure the beans. Spring MVC automatically loads a application context with the name- <servlet-name>-servlet.xml. So in our case Spring will automatically load the Application Context named as Dispatcher-servlet.xml.  
In this configuration file we have written code to scan the packages com.javainuse.controller and "com.javainuse.service.

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

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

xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

**<!-- Scan these packages for Annotated classes -->**

<context:component-scan base-package="com.javainuse.controller" />

<context:component-scan base-package="com.javainuse.service" />

<mvc:annotation-driven />

</beans>

Now lets write our Java classes. First we write the Domain class Employee.java-

package com.javainuse.domain;

public class Employee {

private String empId;

private String name;

private String designation;

private double salary;

public Employee() {

}

public Employee(String empId, String name, String designation, double salary) {

super();

this.setEmpId(empId);

this.name = name;

this.designation = designation;

this.salary = salary;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDesignation() {

return designation;

}

public void setDesignation(String designation) {

this.designation = designation;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

}

Next we write the controller. Above in the Dispatcher-context.xml we have context:component-scan base-package="com.javainuse.controller".

So, Spring scans this particular package for a class annotated with @Controller. The DisplayEmployeeController has a dependency of EmployeeService.java. In DisplayEmployeeController we have a method viewAllItems which has been mapped to the URL /viewAllEmployees. In the xml configuration files we have configured the DispatcherServlet such that when the client hits viewAllItems.do the DispatcherServlet scans the package com.javainuse.controller looking for Controller. The DisplayEmployeeController returns ModelAndView containing the populated Model class Employee and the View name. Currently in our configurations we have not specified any ViewResolver. So by default Spring will use the default InternalResourceViewResolver. In the next chapters we will look at the other view resolvers.  
  
We will write a class DisplayEmployeeController as follows-

package com.javainuse.controller;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.servlet.ModelAndView;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

@Controller

public class DisplayEmployeeController {

@Autowired

private EmployeeService employeeService;

@RequestMapping("/viewAllEmployees")

public ModelAndView viewAllItems() {

List<Employee> allEmployees = employeeService.getAllEmployees();

return new ModelAndView("/displayAllEmployees.jsp", "allEmployees",

allEmployees);

}

}

In DisplayEmployeeController we have autowired EmployeeService class. Also in the Dispatcher-context.xml we have context:component-scan base-package="com.javainuse.service" Spring scans this particular package for a class annotated with @Component. The EmployeeService class will actually fetch all the required Employee information. Usually EmployeeService should call the EmployeeDAO which will return all items from the DataBase.But here we will mock the EmployeeService class to just return the list of items as follows. We are using the Programming to Interface concept here. You can read more about this [in this chapter](https://www.javainuse.com/spring/sprbasic6). So we create the EmployeeService.java and EmployeeServiceMockImpl.java as follows.

package com.javainuse.service;

import java.util.List;

import com.javainuse.domain.Employee;

public interface EmployeeService {

public List<Employee> getAllEmployees();

}

package com.javainuse.service;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Component;

import com.javainuse.domain.Employee;

@Component

public class EmployeeServiceMockImpl implements EmployeeService {

private List<Employee> testEmployees = new ArrayList<Employee>();

**// populate the Employee List**

public EmployeeServiceMockImpl() {

testEmployees.add(new Employee("1", "emp1", "M1", 10000));

testEmployees.add(new Employee("2", "emp2", "M2", 20000));

testEmployees.add(new Employee("3", "emp3", "M3", 30000));

testEmployees.add(new Employee("4", "emp4", "M4", 40000));

testEmployees.add(new Employee("5", "emp5", "M5", 50000));

testEmployees.add(new Employee("6", "emp6", "M6", 60000));

}

**// Return the Mocked Employee List**

public List<Employee> getAllEmployees() {

return new ArrayList<Employee>(testEmployees);

}

}

The DisplayEmployeeController returns the list of employees to a JSP page called displayAllEmployees.jsp using the following code statement return new ModelAndView("/displayAllEmployees.jsp", "allEmployees", allEmployees);  
We will write the displayAllEmployees JSP page to display the Employee List allEmployees as follows-

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Show Employees</title>

</head>

<body>

<div id="books">

<ul>

<c:forEach items="" var="nextEmp">

<li>

<h2></h2>

<h2></h2>

<h2></h2>

<h2></h2>

</li>

</c:forEach>

</ul>

</div>

</body>

</html>

# We now run the two eclipse commands- clean install, tomcat:run to deploy the application. Now open a browser and hit the URL http://localhost:8080/employee-management-system/viewAllEmployees.do. We will see the Employee info as follows-

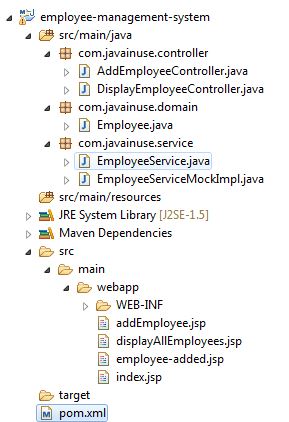
# final-mvc2-2

# Spring MVC: Add Employee Functionality without Spring Form Handling

Overview

In the [previous chapter](https://www.javainuse.com/spring/mvcchap2) we implemented a Simple SpringMVC program to display Employee information. Here we will modify the previous code to add functionality for adding new Employee information to the system.  
Previously we used JSP and JSTL Tag Library in the displayAllEmployees.jsp. SpringMVC also provides its own Form Tag Library ,which has better functionality.  
To show this we will implement the Add Employee Functionality first using normal JSP and JSTL Tag library as before.  
In later chapter we will implement the Add Employee Functionality using Spring Form Library.

Lets Begin

We will create Eclipse Maven project as follows-  


Create a new addEmployee.jsp using JSP and JSTL Tag Library as follows-  
Here the post method addNewEmployee.do is called on pressing the submit button.

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Add Employee</title>

</head>

<body>

<div id="addItem">

<form action="addNewEmployee.do" method="post">

<label>Enter Employee Id</label><input type="text" name="empId" /><br>

<label>Enter Employee Name</label><input type="text" name="name" /><br>

<label>Enter Employee Designation</label><input type="text"

name="designation" /><br> <label>Enter Employee Salary</label><input

type="text" name="salary" /><br> <input type="submit"

value="Add" />

</form>

</div>

</body>

</html>

Now we will write a controller which will map to url addNewEmployee.do.

package com.javainuse.controller;

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

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.servlet.ModelAndView;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

@Controller

public class AddEmployeeController {

@Autowired

private EmployeeService employeeService;

@RequestMapping("/addNewEmployee")

public ModelAndView addEmployee(@RequestParam("empId") String empId,

@RequestParam("name") String name,

@RequestParam("designation") String designation,

@RequestParam("salary") String salary) {

double salaryDouble = new Double(salary);

Employee employee = new Employee(empId, name, designation, salaryDouble);

employeeService.addNewEmployee(employee);

return new ModelAndView("/employee-added.jsp", "name", name);

}

}

Here our EmployeeService is calling addNewEmployee method to add the new Employee info. We will implement this as follows-

package com.javainuse.service;

import java.util.List;

import com.javainuse.domain.Employee;

public interface EmployeeService {

public List getAllEmployees();

public void addNewEmployee(Employee employee);

}

package com.javainuse.service;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Component;

import com.javainuse.domain.Employee;

@Component

public class EmployeeServiceMockImpl implements EmployeeService {

private List testEmployees = new ArrayList();

public DisplayEmployeeServiceMockImpl() {

testEmployees.add(new Employee("1", "emp1", "M1", 10000));

testEmployees.add(new Employee("2", "emp2", "M2", 20000));

testEmployees.add(new Employee("3", "emp3", "M3", 30000));

testEmployees.add(new Employee("4", "emp4", "M4", 40000));

testEmployees.add(new Employee("5", "emp5", "M5", 50000));

testEmployees.add(new Employee("6", "emp6", "M6", 60000));

}

public List getAllEmployees() {

return new ArrayList(testEmployees);

}

public void addNewEmployee(Employee employee) {

testEmployees.add(employee);

}

}

3. Finally we write the JSP page employee-added.jsp to show if the item is successfully added.

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

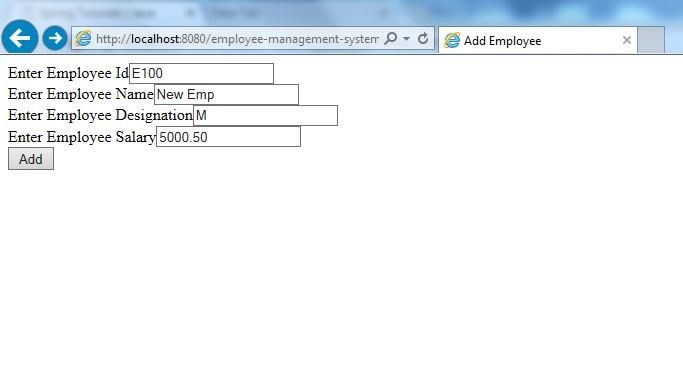
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>

<html>

<h2>The Employee is now added in the System.</h2>

</html>

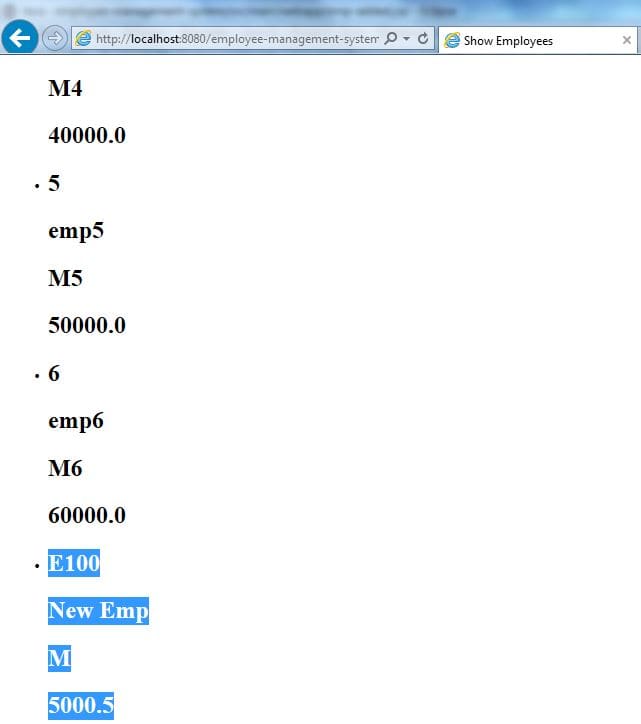
Now open the http://localhost:8080/employee-management-system/addEmployee.jsp and enter the values as below.



click add.



We can confirm if the element has been added by hitting the URl <http://localhost:8080/employee-management-system/viewAllEmployees.do>.

  
  
  
Disadvantages of this approach-

If the salary field is not entered we get a null pointer exception.

No default value like 0.0 for Salary Field.

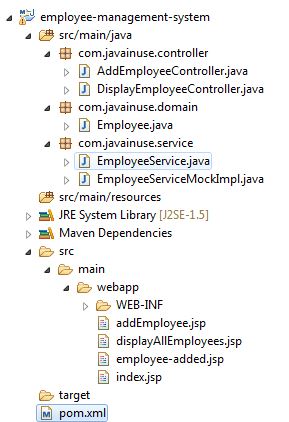
If wrong(non numeric) value is entered for salary, the other field values are also lost.

**Spring MVC: Form handling**

Overview

In the [previous chapter](https://www.javainuse.com/spring/mvcchap3) we implemented a Add new Employee functionality using JSTL. Here we will implement the Add Employee Functionality using Spring Form Library. While using Spring Form Tag Library we will also make use of the Spring Backing Bean

Lets Begin

We will create Eclipse Maven project as follows-  


For this approach we make use of Backing Beans. Here we will split the functionality of addItem method into 2 methods.  
a. The first method show() will create the Backing Bean of Item and return it.  
b. The second method processRequest(Employee emp) will process the addItem Request.  
  
So the AddEmployeeController will be as follows-

package com.javainuse.controller;

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

import org.springframework.stereotype.Controller;

import org.springframework.validation.Errors;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.servlet.ModelAndView;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

@Controller

public class AddEmployeeController {

@Autowired

private EmployeeService employeeService;

@RequestMapping(value = "/addNewEmployee", method = RequestMethod.GET)

public ModelAndView show() {

return new ModelAndView("/addEmployee.jsp", "emp", new Employee());

}

@RequestMapping(value = "/addNewEmployee", method = RequestMethod.POST)

public ModelAndView processRequest(Employee emp, Errors result) {

if (result.hasErrors()) {

return new ModelAndView("/addEmployee.jsp", "emp", emp);

}

employeeService.addNewEmployee(emp);

return new ModelAndView("/employee-added.jsp", "name", emp.getName());

}

}

2. Next we write the addEmployee.jsp which will accept the Backing Bean object created by the controller. Also this jsp makes use of Spring Form Tag Library.

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Add Employee</title>

</head>

<body>

<div id="addEmployee">

<form:form action="addNewEmployee.do" method="post" commandName="emp">

<p>

<label>Enter Employee Id<fmt:message key="emp.empId" /></label>

<form:input path="empId" />

</p>

<p>

<label>Enter Name<fmt:message key="emp.name" /></label>

<form:input path="name" />

</p>

<p>

<label>Enter Type<fmt:message key="emp.designation" /></label>

<form:input path="designation" />

</p>

<p>

<label>Enter Price<fmt:message key="emp.salary" /></label>

<form:input path="salary" />

</p>

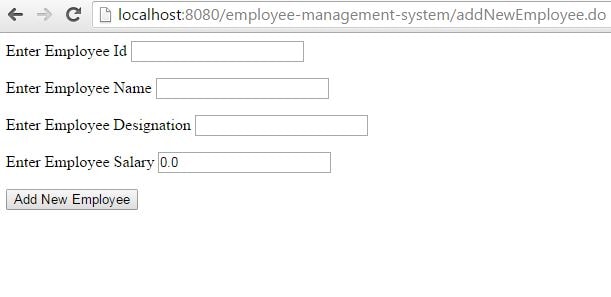
<input type="submit" value="Add New Employee" />

</form:form>

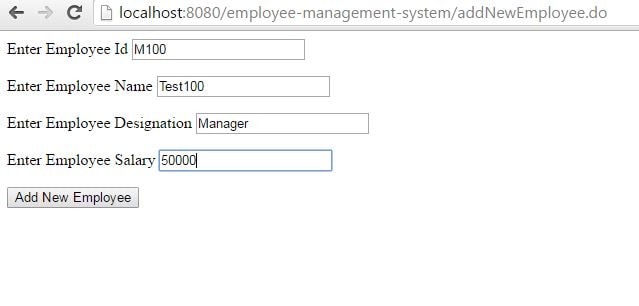
</div>

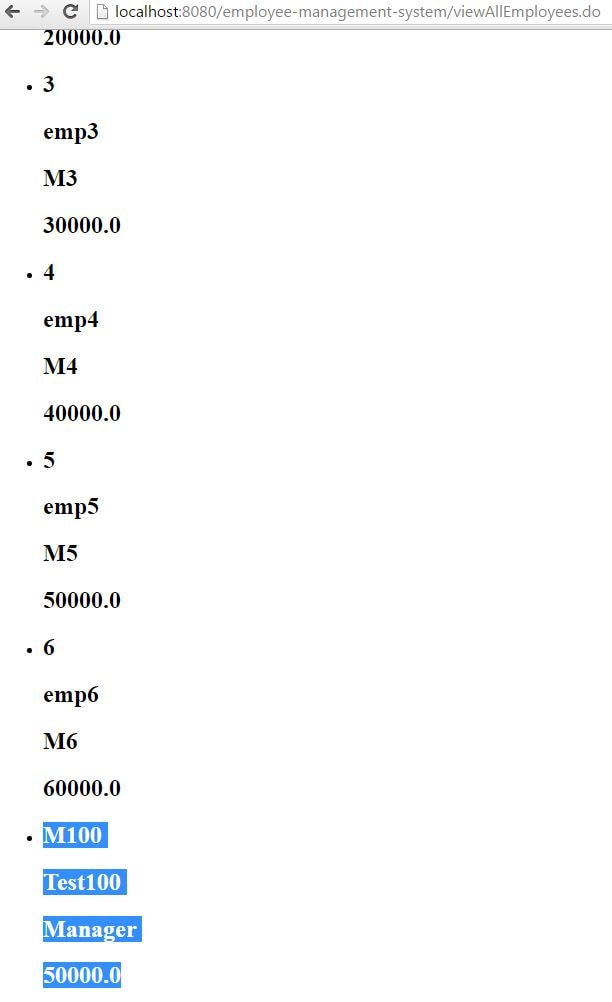
</body>

</html>

Now open the browser and hit the URL http://localhost:8080/employee-management-system/addNewEmployee.do. We will see that the default value of 0.0 for Salary is visible.  


Now enter the values as below and press submit.



  
Verify if the item has been correctly added to the inventory using the URL http://localhost:8080/employee-management-system/viewAllEmployees.do  
  
  
**Advantages of this approach-**

* If the salary field is not entered we do not get a null pointer exception.
* Default value like 0.0 for Salary Field.
* If wrong(non numeric) value is entered for salary, the other field values are not also lost.

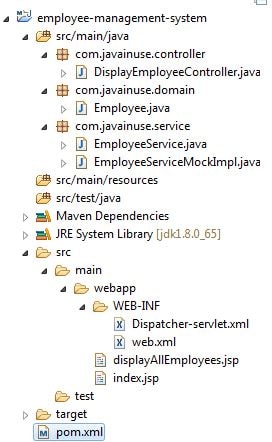
### **REST Webservices**

Spring REST Project to return XML Response

Overview

In this post we will develop REST Spring application to return xml response. This Spring MVC project we expose a REST webservice returning employee information in XML format.Spring REST requires no additional jar files.

Lets Begin

We will create Eclipse Maven project as follows-  
  
So our POM will be as follows

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>employee-management-system</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management-system Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

</dependencies>

<build>

<finalName>employee-management-system</finalName>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

</plugin>

</plugins>

</build>

</project>

The web.xml is as follows. It has a filter to allow all URLs.

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

version="2.5">

<!-- Configure the Disptcher Servlet -->

<servlet>

<servlet-name>Dispatcher</servlet-name>

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

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

</servlet>

<!-- Allow URLs with all extensions -->

<servlet-mapping>

<servlet-name>Dispatcher</servlet-name>

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

</servlet-mapping>

</web-app>

Next we will write Application Context to configure the beans. Spring MVC automatically loads a application context with the name- <servlet-name>-servlet.xml. So in our case Spring will automatically load the Application Context named as Dispatcher-servlet.xml.

In this configuration file we have written code to scan the packages com.javainuse.controller and "com.javainuse.service.

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

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

xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

**<!-- Scan these packages for Annotated classes -->**

<context:component-scan base-package="com.javainuse.controller" />

<context:component-scan base-package="com.javainuse.service" />

<mvc:annotation-driven />

</beans>

Next we define the model class employee. We expect the employee info to be returned in XML format. For this to happen add the @XmlRootElement annotation to the Employee class.

package com.javainuse.domain;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

public class Employee {

private String empId;

private String name;

private String designation;

private double salary;

public Employee() {

}

public Employee(String empId, String name, String designation, double salary) {

super();

this.setEmpId(empId);

this.name = name;

this.designation = designation;

this.salary = salary;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDesignation() {

return designation;

}

public void setDesignation(String designation) {

this.designation = designation;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

}

Also in the Dispatcher-context.xml we have context:component-scan base-package="com.javainuse.service" Spring scans this particular package for a class annotated with @Component. The EmployeeService class will actually fetch all the required Employee information.

Usually EmployeeService should call the EmployeeDAO which will return all items from the DataBase.But here we will mock the EmployeeService class to just return the list of items as follows. We are using the Programming to Interface concept here. You can read more about this [in this chapter](https://www.javainuse.com/spring/sprbasic6). So we create the EmployeeService.java and EmployeeServiceMockImpl.java as follows.

package com.javainuse.service;

import java.util.List;

import com.javainuse.domain.Employee;

public interface EmployeeService {

public List<Employee> getAllEmployees();

}

package com.javainuse.service;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Component;

import com.javainuse.domain.Employee;

@Component

public class EmployeeServiceMockImpl implements EmployeeService {

private List<Employee> testEmployees = new ArrayList<Employee>();

**// populate the Employee List**

public EmployeeServiceMockImpl() {

testEmployees.add(new Employee("1", "emp1", "M1", 10000));

testEmployees.add(new Employee("2", "emp2", "M2", 20000));

testEmployees.add(new Employee("3", "emp3", "M3", 30000));

testEmployees.add(new Employee("4", "emp4", "M4", 40000));

testEmployees.add(new Employee("5", "emp5", "M5", 50000));

testEmployees.add(new Employee("6", "emp6", "M6", 60000));

}

**// Return the Mocked Employee List**

public List<Employee> getAllEmployees() {

return new ArrayList<Employee>(testEmployees);

}

}

Finally we define the DisplayEmployeeController.java where we expose the REST GET service. It will now return the employee with mentioned id.

package com.javainuse.controller;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

@Controller

public class DisplayEmployeeController {

@Autowired

private EmployeeService employeeService;

@RequestMapping("/viewEmployee/{id}")

@ResponseBody

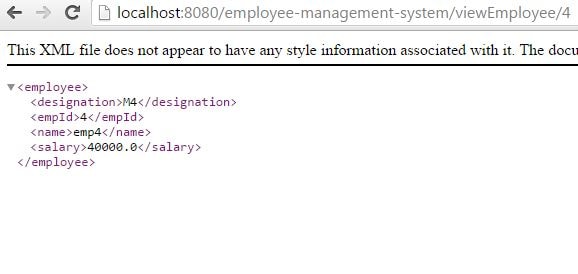
public Employee viewAllItems(@PathVariable String id) {

List<Employee> allEmployees = employeeService.getAllEmployees();

return allEmployees.get(Integer.parseInt(id) - 1);

}

}

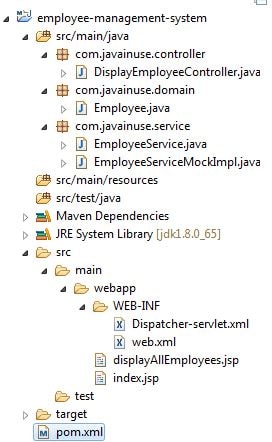
These are the only changes required. We now run the two eclipse commands- clean install, tomcat:run to deploy the application. Hit the browser with the URL http://localhost:8080/employee-management-system/viewEmployee/4  
  


**REST Project to return JSON Response:**

Overview

In this chapter we will modify the previous chapters code to return the output in JSON format other than XML

Lets Begin

We will create Eclipse Maven project as follows-  


In our POM will add the additional dependencies of required for JSON format.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>employee-management-system</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management-system Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.5.4</version>

</dependency>

</dependencies>

<build>

<finalName>employee-management-system</finalName>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

</plugin>

</plugins>

</build>

</project>

The web.xml is as follows. It has a filter to allow all URLs.

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

version="2.5">

<!-- Configure the Disptcher Servlet -->

<servlet>

<servlet-name>Dispatcher</servlet-name>

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

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

</servlet>

<!-- Allow URLs with all extensions -->

<servlet-mapping>

<servlet-name>Dispatcher</servlet-name>

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

</servlet-mapping>

</web-app>

Define the DispatcherServlet.xml to define the configuration for ContentNegotiation parameter.

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd">

<context:component-scan base-package="com.javainuse.controller" />

<context:component-scan base-package="com.javainuse.service" />

<mvc:annotation-driven

content-negotiation-manager="contentNegotiationManager" />

**<bean id="contentNegotiationManager"**

**class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">**

<property name="favorPathExtension" value="false" />

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

<property name="parameterName" value="type" />

**<!-- define the keys which will be specified in the URL to specify the return type -->**

<property name="mediaTypes">

<map>

<entry key="json" value="application/json"></entry>

<entry key="xml" value="application/xml" />

</map>

</property>

</bean>

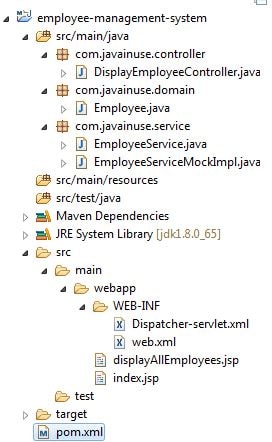
</beans>

**REST Project to return both JSON and XML Response using Spring ContentNegotiationManager**

Overview

In this chapter we will implement the content negotiation for Spring Rest Webservices. Till now we have seen two projects, one [project returned the response as xml](https://www.javainuse.com/spring/rest2) and the [other as JSON](https://www.javainuse.com/spring/rest3). But now suppose we we want to expose a single method as a REST webservice and this should return either xml or JSON depending on some user parameter. This is implemented using ContentNegotiation parameter defined by the Spring Framework. In the request url itself we specify the return type we want using some parameter.

Lets Begin

We will create Eclipse Maven project as follows-  


The pom.xml is as follows-

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>employee-management-system</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management-system Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

**<dependency>**

**<groupId>com.fasterxml.jackson.core</groupId>**

**<artifactId>jackson-annotations</artifactId>**

**<version>2.5.4</version>**

**</dependency>**

**<dependency>**

**<groupId>com.fasterxml.jackson.core</groupId>**

**<artifactId>jackson-databind</artifactId>**

**<version>2.5.4</version>**

**</dependency>**

**<dependency>**

**<groupId>com.fasterxml.jackson.core</groupId>**

**<artifactId>jackson-core</artifactId>**

**<version>2.5.4</version>**

**</dependency>**

</dependencies>

<build>

<finalName>employee-management-system</finalName>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

</plugin>

</plugins>

</build>

</project>

The web.xml is as follows. It has a filter to allow all URLs.

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

version="2.5">

<!-- Configure the Disptcher Servlet -->

<servlet>

<servlet-name>Dispatcher</servlet-name>

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

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

</servlet>

<!-- Allow URLs with all extensions -->

<servlet-mapping>

<servlet-name>Dispatcher</servlet-name>

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

</servlet-mapping>

</web-app>

Define the DispatcherServlet.xml to define the configuration for ContentNegotiation parameter.

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd">

<context:component-scan base-package="com.javainuse.controller" />

<context:component-scan base-package="com.javainuse.service" />

<mvc:annotation-driven

content-negotiation-manager="contentNegotiationManager" />

**<bean id="contentNegotiationManager"**

**class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">**

<property name="favorPathExtension" value="false" />

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

<property name="parameterName" value="type" />

**<!-- define the keys which will be specified in the URL to specify the return type -->**

<property name="mediaTypes">

<map>

<entry key="json" value="application/json"></entry>

<entry key="xml" value="application/xml" />

</map>

</property>

</bean>

</beans>

Define the Employee Model class as follows-

package com.javainuse.domain;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

public class Employee {

private String empId;

private String name;

private String designation;

private double salary;

public Employee() {

}

public Employee(String empId, String name, String designation, double salary) {

super();

this.setEmpId(empId);

this.name = name;

this.designation = designation;

this.salary = salary;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDesignation() {

return designation;

}

public void setDesignation(String designation) {

this.designation = designation;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

}

Define the service layer and its implementation as follows- Also in the Dispatcher-context.xml we have context:component-scan base-package="com.javainuse.service" Spring scans this particular package for a class annotated with @Component. The EmployeeService class will actually fetch all the required Employee information.

Usually EmployeeService should call the EmployeeDAO which will return all items from the DataBase.But here we will mock the EmployeeService class to just return the list of items as follows. We are using the Programming to Interface concept here. You can read more about this [in this chapter](https://www.javainuse.com/spring/sprbasic6). So we create the EmployeeService.java and EmployeeServiceMockImpl.java as follows.

package com.javainuse.service;

import java.util.List;

import com.javainuse.domain.Employee;

public interface EmployeeService {

public List<Employee> getAllEmployees();

}

package com.javainuse.service;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Component;

import com.javainuse.domain.Employee;

@Component

public class EmployeeServiceMockImpl implements EmployeeService {

private List<Employee> testEmployees = new ArrayList<Employee>();

**// populate the Employee List**

public EmployeeServiceMockImpl() {

testEmployees.add(new Employee("1", "emp1", "M1", 10000));

testEmployees.add(new Employee("2", "emp2", "M2", 20000));

testEmployees.add(new Employee("3", "emp3", "M3", 30000));

testEmployees.add(new Employee("4", "emp4", "M4", 40000));

testEmployees.add(new Employee("5", "emp5", "M5", 50000));

testEmployees.add(new Employee("6", "emp6", "M6", 60000));

}

**// Return the Mocked Employee List**

public List<Employee> getAllEmployees() {

return new ArrayList<Employee>(testEmployees);

}

}

No header information like headers="Accept=application/json" or headers="Accept=application/xml" is needed. Since the return type format is being specified by the ContentNegotiationManager in the configuration file.

package com.javainuse.controller;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.servlet.ModelAndView;

import com.javainuse.domain.Employee;

import com.javainuse.service.EmployeeService;

@Controller

public class DisplayEmployeeController {

@Autowired

private EmployeeService employeeService;

@RequestMapping(value = "/viewAllEmployees.do")

@ResponseBody

public Employee viewAllItems() {

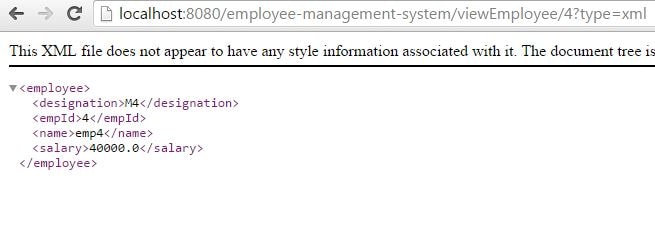
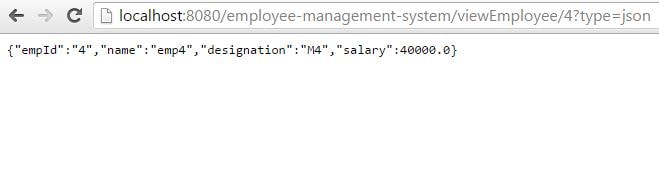
List<Employee> allEmployees = employeeService.getAllEmployees();

return allEmployees.get(0);

}

}

These are the only changes required. We now run the two eclipse commands- clean install, tomcat:run to deploy the application.

Hit the browser with the following URL for json response  
http://localhost:8080/employee-management-system/viewEmployee/4?type=xml  
  
Hit the browser with the following URL for json response  
http://localhost:8080/employee-management-system/viewEmployee/4?type=json  


**Rest-Assured Hello World - Getting started using simple example**

Overview

In [previous chapter](https://www.javainuse.com/spring/rest4) we implemented REST web service to return either xml or JSON depending on some user parameter using ContentNegotiation parameter. In this chapter we test the same webservice using Rest-Assured.

What is Rest-Assured ?

Testing and validating REST services in Java is harder than in dynamic languages such as Ruby and Groovy. REST Assured brings the simplicity of using these languages into the Java domain. It eliminates the requirement of using boiler-plate code to test complex API responses, and supports both XML and JSON.

Lets Begin

The pom.xml will be as follows-

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>RestAssuredTest</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

**<dependency>**

**<groupId>io.rest-assured</groupId>**

**<artifactId>rest-assured</artifactId>**

**<version>3.0.0</version>**

**</dependency>**

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-all</artifactId>

<version>1.3</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

</dependency>

<dependency>

<groupId>pl.pragmatists</groupId>

<artifactId>JUnitParams</artifactId>

<version>1.0.4</version>

</dependency>

</dependencies>

</project>

Below we use the Given/When/Then structure that is borrowed from BDD (Behaviour Driven Development). In Given section we declare things like content type or request body. In When section we provide HTTP method and endpoint. In Then section we declare response verification.

package com.javainuse;

import org.hamcrest.Matchers;

import org.junit.Test;

import io.restassured.response.ValidatableResponse;

import static io.restassured.RestAssured.given;

public class TestRestAssured {

@Test

public void restAssuredTest() {

**ValidatableResponse response =**

**given().queryParam("type", "json").when()**

**.get("http://localhost:8080/employee-management-system/viewEmployee/4").then();**

**System.out.println("Response is - "+response.extract().body().asString());**

response.body(Matchers.containsString("emp4"));

}

}

Now start the [REST web service](https://www.javainuse.com/spring/rest4) we developed in the previous chapter. Run the above class and the test case gets executed successfully. In this test the REST call is made and the response we check if it contains a particular string.

# Spring Boot - Exception Handling

Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications. In this chapter, we will learn how to handle exceptions in Spring Boot.

Before proceeding with exception handling, let us gain an understanding on the following annotations.

## **Controller Advice**

The @ControllerAdvice is an annotation, to handle the exceptions globally.

## **Exception Handler**

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

You can use the following code to create @ControllerAdvice class to handle the exceptions globally −

package com.tutorialspoint.demo.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice

public class ProductExceptionController {

}

Define a class that extends the RuntimeException class.

package com.tutorialspoint.demo.exception;

public class ProductNotfoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

}

You can define the @ExceptionHandler method to handle the exceptions as shown. This method should be used for writing the Controller Advice class file.

@ExceptionHandler(value = ProductNotfoundException.class)

public ResponseEntity<Object> exception(ProductNotfoundException exception) {

}

Now, use the code given below to throw the exception from the API.

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object> updateProduct() {

throw new ProductNotfoundException();

}

The complete code to handle the exception is given below. In this example, we used the PUT API to update the product. Here, while updating the product, if the product is not found, then return the response error message as “Product not found”. Note that the **ProductNotFoundException** exception class should extend the **RuntimeException**.

package com.tutorialspoint.demo.exception;

public class ProductNotfoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

}

The Controller Advice class to handle the exception globally is given below. We can define any Exception Handler methods in this class file.

package com.tutorialspoint.demo.exception;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice

public class ProductExceptionController {

@ExceptionHandler(value = ProductNotfoundException.class)

public ResponseEntity<Object> exception(ProductNotfoundException exception) {

return new ResponseEntity<>("Product not found", HttpStatus.NOT\_FOUND);

}

}

The Product Service API controller file is given below to update the Product. If the Product is not found, then it throws the **ProductNotFoundException** class.

package com.tutorialspoint.demo.controller;

import java.util.HashMap;

import java.util.Map;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.exception.ProductNotfoundException;

import com.tutorialspoint.demo.model.Product;

@RestController

public class ProductServiceController {

private static Map<String, Product> productRepo = new HashMap<>();

static {

Product honey = new Product();

honey.setId("1");

honey.setName("Honey");

productRepo.put(honey.getId(), honey);

Product almond = new Product();

almond.setId("2");

almond.setName("Almond");

productRepo.put(almond.getId(), almond);

}

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object> updateProduct(@PathVariable("id") String id, @RequestBody Product product) {

if(!productRepo.containsKey(id))throw new ProductNotfoundException();

productRepo.remove(id);

product.setId(id);

productRepo.put(id, product);

return new ResponseEntity<>("Product is updated successfully", HttpStatus.OK);

}

}

The code for main Spring Boot application class file is given below −

package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

The code for **POJO class** for Product is given below −

package com.tutorialspoint.demo.model;

public class Product {

private String id;

private String name;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

The code for **Maven build – pom.xml** is shown below −

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

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.tutorialspoint</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.8.RELEASE</version>

<relativePath/>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

The code for **Gradle Build – build.gradle** is given below −

buildscript {

ext {

springBootVersion = '1.5.8.RELEASE'

}

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")

}

}

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = 1.8

repositories {

mavenCentral()

}

dependencies {

compile('org.springframework.boot:spring-boot-starter-web')

testCompile('org.springframework.boot:spring-boot-starter-test')

}

You can create an executable JAR file, and run the Spring Boot application by using the Maven or Gradle commands −

For Maven, you can use the following command −

mvn clean install

After “BUILD SUCCESS”, you can find the JAR file under the target directory.

For Gradle, you can use the following command −

gradle clean build

After “BUILD SUCCESSFUL”, you can find the JAR file under the build/libs directory.

You can run the JAR file by using the following command −

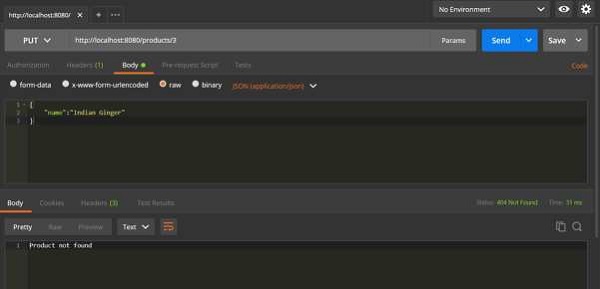
java –jar <JARFILE>

This will start the application on the Tomcat port 8080 as shown below −

Exception Handling Tomcat Application Startded

Now hit the below URL in POSTMAN application and you can see the output as shown below −

Update URL: http://localhost:8080/products/3



# Spring Boot Remove Embedded Tomcat Server, Enable Jetty Server

## **1. Introduction**

In this tutorial, we’ll learn **how to remove the Tomcat server from the Spring Boot application**. Actually, Spring boot by default comes up with the embedded server once we add “**spring-boot-starter-web**” dependency.

But Spring boot gives us the flexibility to use tomcat or not. If we do not want, we can exclude this default server.

Default, Spring boot comes with 3 types of embed servers Tomcat, Jetty and undertow.

First, we’ll see how to **exclude tomcat** and next **add jetty server**.

**2. Tomcat By Default**

Once we add spring-boot-starter-web dependency as part of pom.xml for web application development with spring boot, it gets the tomcat along with all required dependencies. It is always convenient to use directly and auto deployable to tomcat.

|  |  |
| --- | --- |
| 1  2  3  4 | <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-web</artifactId>  </dependency> |

But there are some scenarios, you do not need to use tomcat as part of Spring Boot application when using JMS instead of a web app or want to add Jetty.

## **3. Exclude Tomcat – Maven Pom.xml**

To exclude tomcat from spring boot, just need to add an additional block to the Spring Boot Starter dependency. In the dependency section, We can add  
*<exclusions>* tags that make sure the given artifact is removed at build time.

This is the easiest way to do it.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-web</artifactId>      <exclusions>          <exclusion>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-tomcat</artifactId>          </exclusion>      </exclusions>  </dependency> |

You can use that approach to exclude Tomcat from Spring Boot and also for any other exclusions

## **4. Exclude Tomcat and All Servers – Annotation**

When declaring the @SpringBootApplication annotation, there is a way to exclude all servers and do consider the spring boot application like the web.

To make spring boot as a non-web application, use the following.

|  |  |
| --- | --- |
| 1 | @SpringBootApplication(exclude = {EmbeddedServletContainerAutoConfiguration.class, WebMvcAutoConfiguration.class}) |

And need to add the below property to non-rest applications so that spring boot does not try to start the *WebApplicationContext*. This should go to the application.properties.

|  |  |
| --- | --- |
| 1 | spring.main.web-environment=false |

## **5. Add Jetty Server in Spring Boot**

If you want to use the Jetty server in Spring boot application, first you must need to disable the default tomcat server and then add jetty dependency “

*spring-boot-starter-jetty*“.

|  |  |
| --- | --- |
| 1  2  3  4 | <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-jetty</artifactId>  </dependency> |

After adding jetty in pom.xml then at build time it disables tomcat and maps to the Jetty configurations.

## **6. Gradle – Exclude tomcat and Add Jetty**

This is quite easy then maven. Just add the tomcat at the exclusions section and add jetty in the dependencies section.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | configurations {      compile.exclude module: "spring-boot-starter-tomcat"  }    dependencies {      compile("org.springframework.boot:spring-boot-starter-web:2.0.0.BUILD-SNAPSHOT")      compile("org.springframework.boot:spring-boot-starter-jetty:2.0.0.BUILD-SNAPSHOT")  } |