**MVC Design Pattern**

With MVC design pattern, we have following components on which our design depends:

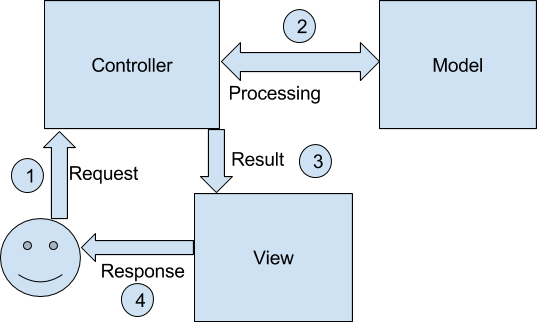
* The model which is transferred from one layer to the other.
* The View which is responsible to show the data present in the application.
* The controller is responsible to accept a request from a user, modify a model and send it to the view which is shown to the user.

The idea behind MVC pattern is a very clear separation between domain objects which represents real-world entities and the presentation layer we see on the screen. Domain objects should be completely independent and should work without a View layer as well.

In this lesson, we will try to establish a strong MVC example with a Spring Boot application which uses [Thymeleaf](https://www.thymeleaf.org/" \t "_blank) in the View layer.

**MVC Application Architecture**

Before we start building our application, it is always a good idea to structure it. Based on the MVC pattern, we will have following layers in our application:



**MVC Pattern Example**

As we will build a Spring Boot application based on MVC pattern, we start by making a Maven based project and added Spring Boot related dependencies to it.

<parent>

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

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

<version>1.5.7.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</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-data-jpa</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</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>

Now, let’s start by defining our model.

**MVC Model**

Let’s start defining the pieces and move with our model first. The first step for a sample app is defining our **M**odel class:

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

@Entity

public class Person {

@Id

@GeneratedValue

private Long id;

private String name;

private int age = 0;

//standard getters and setters

}

This is a simple model class which can contain a Person’s information. Let’s use the defined model in our Controller and Service.

**MVC Controller**

Now that we know what data to transmit in Model, let us define a **C**ontroller to allow user to request this data:

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

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

@Controller

public class PersonController {

@Autowired

PersonService personService;

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

public String greetingForm(Model model) {

model.addAttribute("person", new Person());

return "greeting";

}

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

public String addPerson(Model model, @ModelAttribute Person person) {

personService.createPerson(person);

model.addAttribute("people", personService.getAllPersons());

return "result";

}

}

In the controller above, we defined two APIs:

1. The base URL of the application presents a simple View we design next.
2. The /person URL of the application presents a simple View which tabulates the person data.

We will see the Views in a minute. Before that, we need to provide the Service and Repository layer as well.

import java.util.List;

public interface PersonService {

Person createPerson(Person person);

List<Person> getAllPersons();

}

The interface implementation looks like:

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

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class PersonServiceImpl implements PersonService {

@Autowired

private PersonRepository personRepository;

@Override

public Person createPerson(Person person) {

return personRepository.save(person);

}

@Override

public List<Person> getAllPersons() {

return personRepository.findAll();

}

}

Finally, The DAL layer, the repository interface looks like:

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

import org.springframework.stereotype.Repository;

@Repository

public interface PersonRepository extends JpaRepository<Person, Long> {

}

Excellent! Now that we’re done with the all the service and DB access logic, we can finally try our View.

**MVC View**

Let’s finally put our **V**iew in place. We first define our greetings.html page in src/main/resources/templates directory.

<!DOCTYPE HTML>

<html xmlns:th="https://www.thymeleaf.org">

<head>

<title>Spring Boot MVC</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

</head>

<body>

<h1>Add new person</h1>

<form action="#" th:action="@{/person}" th:object="${person}" method="post">

<p>Name: <input type="text" th:field="\*{name}"/></p>

<p>Age: <input type="number" th:field="\*{age}"/></p>

<p><input type="submit" value="Submit"/> <input type="reset" value="Reset"/></p>

</form>

</body>

</html>

Next, we define our result.html page as well:

<!DOCTYPE HTML>

<html xmlns:th="https://www.thymeleaf.org">

<head>

<title>Spring Boot MVC</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

</head>

<body>

<div th:if="${not #lists.isEmpty(people)}">

<h2>Person List</h2>

<table class="glyphicon glyphicon-calendar">

<tr>

<th>Id</th>

<th>Name</th>

<th>Age</th>

</tr>

<tr th:each="person : ${people}">

<td th:text="${person.id}"></td>

<td th:text="${person.name}"></td>

<td th:text="${person.Age}"></td>

</tr>

</table>

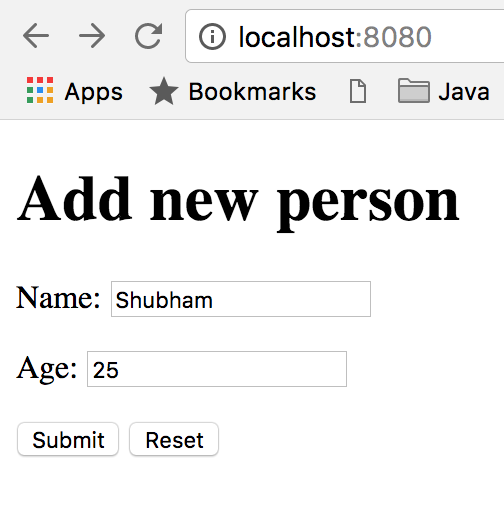
</div>

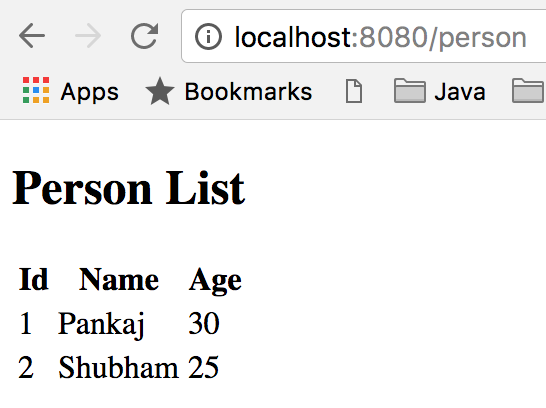
</body>

</html>

**MVC Pattern Test**

Now, we can finally run our application to see it in action. But the application was not the focus here. Focus was to keep the three components completely separate which was excellent to observe. However, below images show the project output.





**Advantages of MVC Design pattern**

There are many advantages for using MVC pattern. Let’s state some of them here:

1. MVC allows rapid application development. If we simply define the Model layer in advance and pass this information to UI development team, they can start making the Views in paralles as Backend developers design the application Controllers and logic behind them resulting in faster development.
2. While changing a View mechanism, Controller layer doesn’t even have to know where the data goes. It only knows a logical view name and not even the HTML extension. So, it is very easy to switch to an Angular based view without affecting the Controller.
3. MVC pattern emphasis on the low coupling between different components of an application. So, the View layer have no dependency on service layer and only Controller depends on it, even that with the interfaces and not from concrete implementation.