Lex\_Spring Boot

About Spring Boot:

Spring is a very popular Java-based framework for constructing web and enterprise applications. However, you are aware that configuring a Spring application is a tedious job. Even though it provides flexibility in bean configuration with multiple ways such as XML, annotation, and Java-based configurations, there is no escape from the configuration. Configuring the Spring features may need more time of developers and may distract the developers from solving the business problems.

Thanks to Spring Team for releasing Spring Boot, one of the topmost innovations in all existing Spring Framework. Spring Boot provides a new prototype for developing Spring applications with nominal effort. Using Spring Boot, you will be able to develop Spring applications with extra agility and capacity to focus on solving business needs with nominal (or possibly no) thought of configuring Spring itself.

In this course, we will discuss all the cool features of Spring Boot that make it easy to create stand-alone, production-grade applications that we can just run.

Target Audience: Developers

**InfyGo Reservation System**

InfyGo is a leading online travel company in India. It owns an online application named IRS (InfyGo Reservation System), built using Spring Core + Spring Data JPA + Spring MVC.

InfyGo customers can book their air tickets at an affordable price, following simple and easy steps through the IRS website. If the customers enter their travel details, the website will fetch the best-suited flights for them.

Let us experience the IRS application.

**Demo: Executing InfyGo Reservation System developed using SpringMVC .**

Highlights:

* Import the SpringMVC project into eclipse.
* Execute the project to understand the InfyGo Reservation System problem domain.

Demo Steps:

Step 1: Download the [InfyGoMVCMaven](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_1189424319468846600/web-hosted/assets/InfyGoMaven.zip) zip file, extract and save it to a local drive.

Step 2: Extract the zip file.

Step 3: Open Eclipse and create a new workspace.

Create a new workspace or select the existing workspace.

Click on OK and then workbench.

Step 4: Import the extracted InfyGoMaven application.

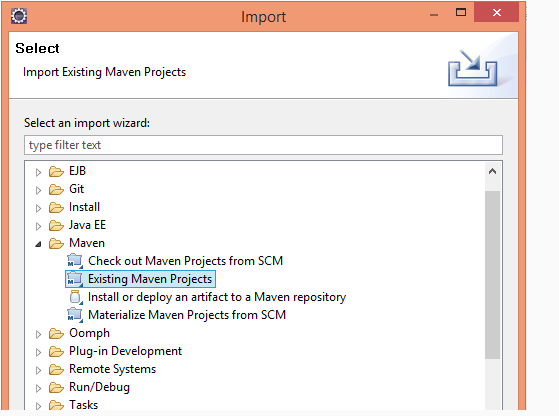
Go to File menu -> Import -> Maven -> Existing Maven Projects -> Browse and select the extracted folder and click on Next then Finish.



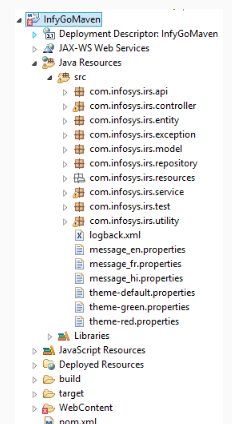
Click on OK and then workbench.

Step 4: Import the extracted InfyGoMaven application.

Go to File menu -> Import -> Maven -> Existing Maven Projects -> Browse and select the extracted folder and click on Next then Finish.

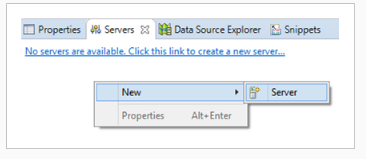


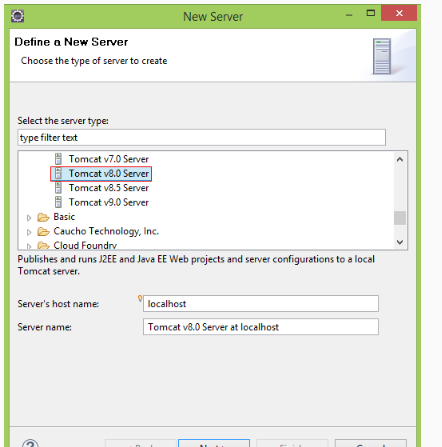
You can see the project structure below.



Once imported, you can see an error mark in the project as shown above. This error will vanish when you add the tomcat server.

Step 5: Add a new tomcat server like below.





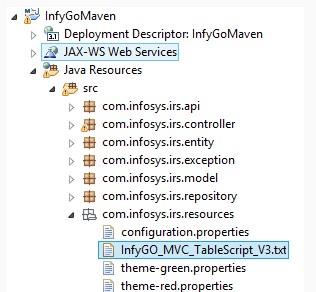
Select the Tomcat 8 server folder from your local drive.



Click on Next and Finish.

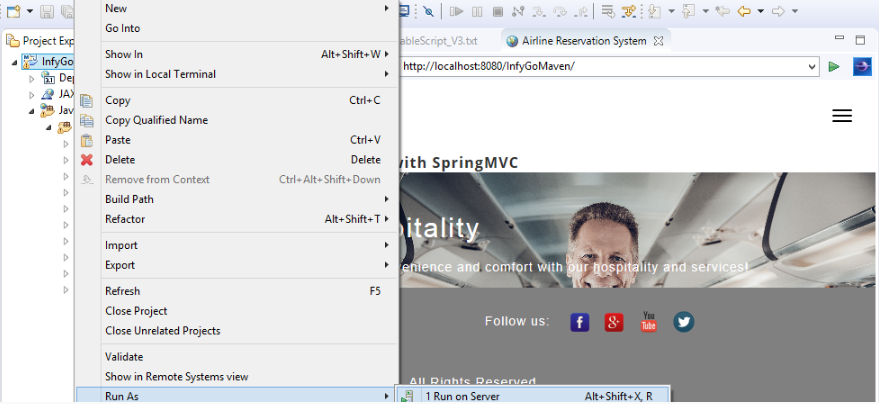
Step 6: Start MySQL server and create db\_irs\_dev database.

Then go to com.infosys.irs.resources -> open InfyGoMvc table script copy and paste into MySQL, execute the table scripts.



Step 7: Run the project.

Right-click on the project -> Run as - > Run On Server.

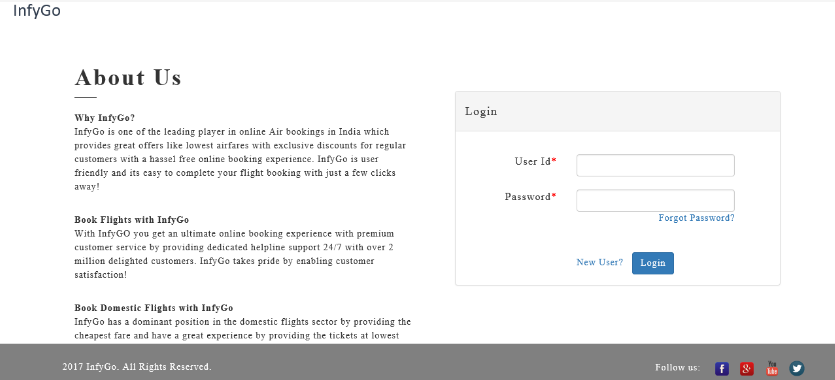


Click on Next and Finish.

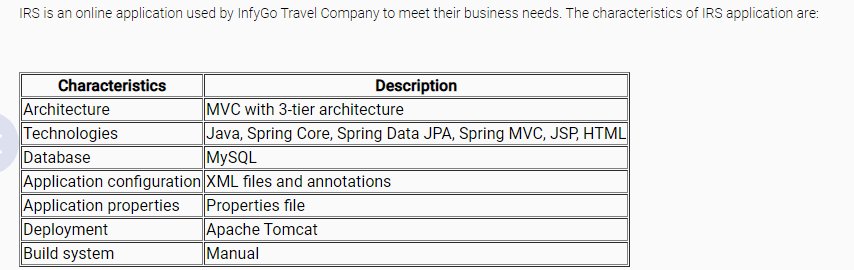
Step 8: Copy the URL from eclipse and paste it into chrome, observe the below home page.



Go to Sign Up and register with InfyGo, then login with username and password.



Register with InfyGo then login and observe the search flights, add passenger details, make a payment, etc.



**Challenges faced with IRS:**

**Application server**

The operations team will deliver and support the servers, the development team will run their code on it. Therefore, the development team has to depend on the operations team for any issues with the server.

The company has to spend money on obtaining a license for the application server.

**Dependencies**

The development team has to pursuit all the well-matched libraries for the Spring version used in their application and manually configure them. The existing IRS application uses around 50 jar files.

The team needs to identify the appropriate library files for every new feature added to the application. Forex, to enable the AOP feature in the existing IRS application, five more jars need to be added.

If the Spring version needs to be updated, then the team has to update all the libraries used. To upgrade the Spring version of the IRS application, all the 50 jar files need to be updated.

**Configurations**

The development team has to write and manage a lot of configurations in the form of .properties files or .xml files to configure Spring beans, Servlet classes, data sources and etc.

The application properties need to overridden for various environments like production, development, testing, etc. For e.g., the database used by the testing team may be different from the one used by the development team.

To overcome these challenges, InfyGo management decided to use Spring Boot. Let us compare the results after using Spring Boot on IRS.

**Spring MVC IRS vs Spring Boot IRS**

The below table compares the existing IRS application with the IRS application built with Spring Boot.

| **Challenges** | **Criteria** | **Existing IRS** | **IRS built with Spring Boot** |
| --- | --- | --- | --- |
| Application Server | Execution | An external application server helps to run the application. | Spring Boot provides an embedded application server |
| Cost | The Licensing cost has to be paid for the application server | As the application server is embedded within Spring Boot, no need to pay for it. |
| Dependencies | Dependent Libraries | Developers have to manually configure all the dependent libraries. | Spring Boot starters take care of configuring dependent libraries. |
| Development Time | More time is taken for pre-setup work like adding libraries, configuring spring features and etc. | Less time is taken for pre-setup work as Spring Boot does auto-configuration |
| Configurations | Upgradation | More time is taken to upgrade the application to the latest Spring version. | Less time is taken as everything gets updated implicitly just by changing the parent section of pom.xml |
| Configuration files | Developers have to manually do a lot of configurations in the form of .xml files or .properties files or .java files to enable Spring features. | Spring Boot required no such configuration files as the required features would be auto-configured. |
| Customization based on the environment | Highly customizable but the properties need to be manually overridden for the existing environment. | Through Profiles and external configurations, the application can be easily customized for any environment. |
| Production-ready features | Coding has to complete manually if application health needs to monitor in production. | With the help of Actuators, web endpoints can be given to monitoring the application health |

Let us learn how to build an IRS application using Spring Boot.

Before proceeding, let us see what Spring Boot is and what its features are.

**FEATURES:**

**What is Spring Boot?**

Spring Boot is one of the Spring projects that use a new development model for quickly building Spring applications with very little configuration.

# **However, how does it work?**

Because it is,

1. Opinionated

Spring Boot forms opinions. It means that Spring Boot has some sensible defaults, which you can use to quickly build your application. For example, Spring Boot uses embedded Tomcat as the default web container.

**2. Customizable**

Though Spring Boot has its defaults, you can easily customize it at any time during your development based on your need. Forex, if you prefer Jetty, then it should be updated easily in the POM file to replace the default Tomcat container.

# **Spring Boot components**

* Starter - Manages the dependencies of the Spring-based projects by combining the group of common or related dependencies. Spring Boot forms the opinions based on starters.
* Auto configures - Detects and automatically configures the Spring applications based on the added dependencies.
* CLI - Command Line Interface to start, test, and stop Spring Boot applications from the command prompt (Not discussed in this course)
* Actuator - Enables enterprise features and gives an insight into the application.

We will be discussing all these components in detail as we progress.

Let us discuss in detail to develop an application using Spring Boot.

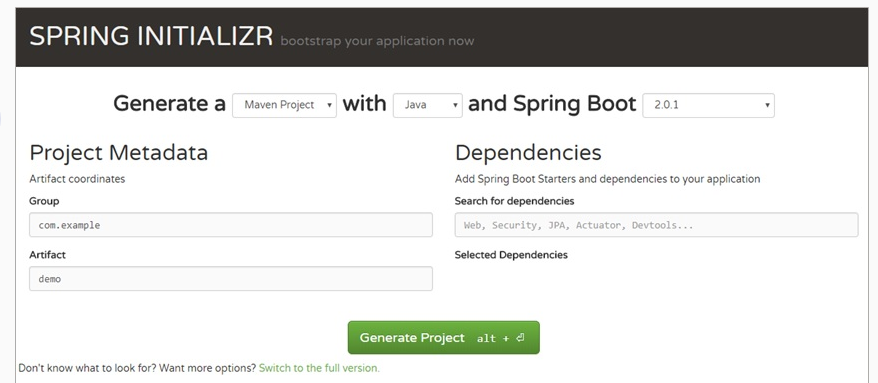
**Boot strapping Spring Boot application**

You are going to use Spring Initializer and Spring Tool Suite IDE to develop Spring Boot applications.

# **Spring Initializer**

A web tool by Spring official web site helps to download a Spring Boot project with all required dependencies pre-configured, could be imported into your IDE

It lets you choose between the Maven or Gradle project with the required dependencies Boot version for your project.



You can switch to the Full Version option to add the dependencies by clicking on the checkboxes and to configure the advanced options like Cloud, Social, Template Engines, etc.

You, as the developer, can just focus on the application code.

# **Spring Tool Suite**

Spring Tool Suite is an Eclipse-based IDE dedicated to developing Spring-based applications.

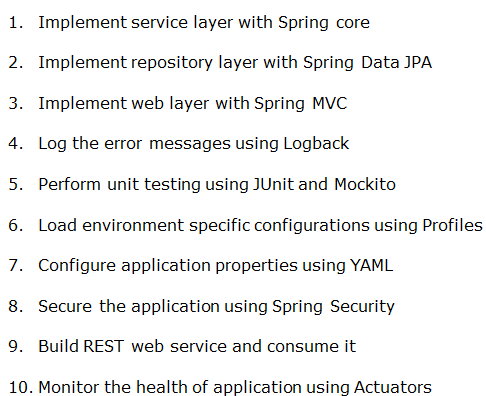
It comes with the pre-installed latest Eclipse release, Spring-based components, server integration for Eclipse, and other things required to generate an Enterprise application using Spring.

Let us see how to develop an IRS application using Spring initializr and Spring Tool Suite.

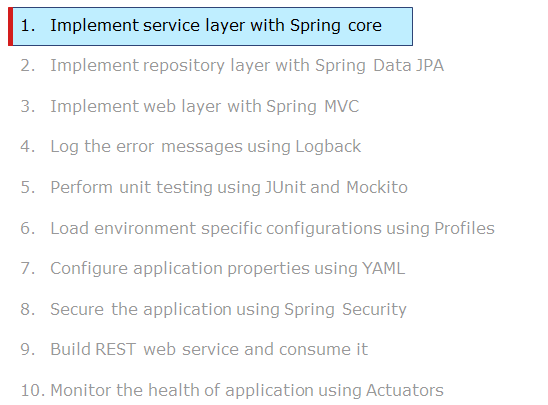
# **Spring Boot 2 Features**

* Spring Boot 2 requires Java 8 as a baseline and could support Java 9 also.
* Spring Boot requires Spring 5 as a baseline.
* Jetty is 9.4 and Tomcat is 8.5
* Starter POM and auto-configuration for reactive Spring Data Cassandra, MongoDB, Redis, and Couchbase.
* Spring Boot 2 enables HTTP/2 for MVC and WebFlux applications.
* Default security configuration is available and customization is also possible.
* Provides a brand new architecture that supports Spring MVC, Jersey, and WebFlux.
* The metrics of Spring Boot have been replaced with dimensional metrics and Micrometer.

Developing IRS with Spring Boot



Implement service layer with Spring core



The service layer uses Spring core libraries to implement the business functionalities. Let us discuss how to integrate Spring Core with Boot.

Highlights:

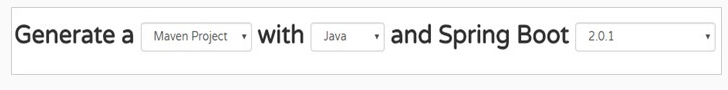
* Spring Boot project creation using Spring Initializr
* How to code and run the Spring Boot application in STS
* How to integrate Spring Boot with Spring Core

Demo steps:

Step 1: Go to the Spring Initializr website and create a new maven based Spring Boot project.

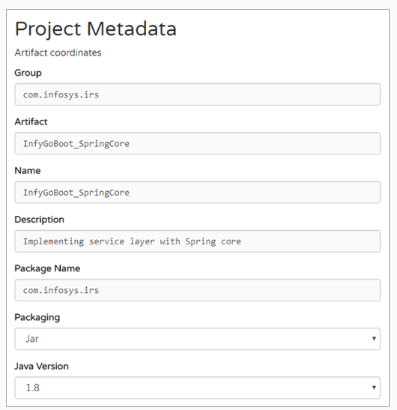
Step 2: Generate a 'Maven' project with the latest Spring Boot version.

Note: The course demos have been created using Spring Boot 2.0.1 version.



Step 3: Click on 'Switch to the full version' to provide package details.

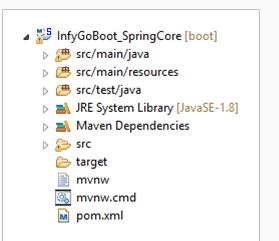
Step 4: Fill in the 'Project Metadata' details as shown below.



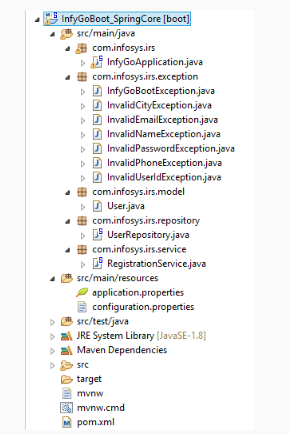
Step 5: Click on 'Generate Project'.

Step 6: Save the zip file in your system in any local location and unzip it.

Step 7: Open STS and import the project created above by selecting the File menu and choosing Import -> Existing Maven Project -> Browse to the application and Finish. You can see the project structure below.



Step 8: Create Java files with the package structure as shown below.



Step 9: Implement 'User.java' as below. It holds the user information and it is used to transfer the user details across the layers of the application

public class User {

private String userId;

private String password;

private String name;

private String city;

private String email;

private String phone;

public String getUserId() {

return userId;

}

public void setUserId(String userId) {

this.userId = userId;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getPhone() {

return phone;

}

public void setPhone(String phone) {

this.phone = phone;

}

}

Step 10: Implement 'UserRepository.java' as below. As you are concentrating more on implementing a service layer, you will have a stub for the repository layer as of now. Later you will be learning the implementation of the repository layer in Spring Boot.

import org.springframework.stereotype.Component;

@Component

public class UserRepository {

private String successMessage = "UserRespository.REGISTRATION\_SUCCESS";

public String registerUser() {

return successMessage;

}

}

Step 11: Implement 'InfyGoBootException.java' as below.

package com.infosys.irs.exception;

/\*\*

\* All other user defined exception classes will inherit this exception class.

\*

\*/

public class InfyGoBootException extends Exception {

private static final long serialVersionUID = 1L;

public InfyGoBootException(String message) {

super(message);

}

}

Step 11: Implement 'InvalidCityException.java' as below.

package com.infosys.irs.exception;

public class InvalidCityException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

/\*\*

\*

\* This Exception is thrown from RegistrationService class with error message

\* RegistrationService.INVALID\_CITY if the given city is not matching the

\* constraints given in the regular expression.

\*

\*

\*

\*/

public InvalidCityException(String message) {

super(message);

}

}

Step 12: Implement 'InvalidEmailException.java' as below.

package com.infosys.irs.exception;

public class InvalidEmailException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

/\*\*

\*

\* This Exception is thrown from RegistrationService class with error message

\* RegistrationService.INVALID\_EMAIL if the given email is not matching the

\* constraints given in the regular expression.

\*

\*

\*

\*/

public InvalidEmailException(String message) {

super(message);

}

}

Step 13: Implement 'InvalidNameException.java' as below.

package com.infosys.irs.exception;

public class InvalidNameException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

/\*\*

\*

\* This Exception is thrown from RegistrationService class with error message

\* RegistrationService.INVALID\_NAME if the given name is not matching the

\* constraints given in the regular expression.

\*

\*

\*

\*/

public InvalidNameException(String message) {

super(message);

}

}

Step 14: Implement 'InvalidPasswordException.java' as below.

package com.infosys.irs.exception;

public class InvalidPasswordException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

/\*\*

\*

\* This Exception is thrown from RegistrationService class with error message

\* RegistrationService.INVALID\_PASSWORD if the given password is not matching

\* the constraints given in the regular expression.

\*

\*

\*

\*/

public InvalidPasswordException(String message) {

super(message);

}

}

Step 15: Implement 'InvalidPhoneException.java' as below.

package com.infosys.irs.exception;

public class InvalidPhoneException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

/\*\*

\*

\* This Exception is thrown from RegistrationService class with error message

\* RegistrationService.INVALID\_PHONE\_NUMBER if the given phone is not matching

\* the constraints given in the regular expression.

\*

\*

\*

\*/

public InvalidPhoneException(String message) {

super(message);

}

}

Step 16: Implement 'InvalidUserIdException.java' as below.

package com.infosys.irs.exception;

public class InvalidUserIdException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

/\*\*

\*

\* This Exception is thrown from RegistrationService class with error message

\* RegistrationService.INVALID\_USER\_ID if the given userId is not matching the

\* constraints given in the form of regular expression.

\*

\*

\*

\*/

public InvalidUserIdException(String message) {

super(message);

}

}

Step 17: Implement 'RegistrationService.java' as below. This class is responsible for taking care of business validations and business operations.

package com.infosys.irs.service;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

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

import org.springframework.stereotype.Service;

import com.infosys.irs.exception.InfyGoBootException;

import com.infosys.irs.exception.InvalidCityException;

import com.infosys.irs.exception.InvalidEmailException;

import com.infosys.irs.exception.InvalidNameException;

import com.infosys.irs.exception.InvalidPasswordException;

import com.infosys.irs.exception.InvalidPhoneException;

import com.infosys.irs.exception.InvalidUserIdException;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

@Service

public class RegistrationService {

@Autowired

private UserRepository userRepository;

String regex1 = "^[a-zA-Z0-9]{4,15}+$";

public String registerUser(User user) throws InfyGoBootException {

String registrationMessage = null;

validateUser(user);

registrationMessage = userRepository.registerUser();

return registrationMessage;

}

public void validateUser(User user) throws InfyGoBootException {

if (!isValidUserId(user.getUserId()))

throw new InvalidUserIdException("RegistrationService.INVALID\_USER\_ID");

if (!isValidPassword(user.getPassword()))

throw new InvalidPasswordException("RegistrationService.INVALID\_PASSWORD");

if (!isValidName(user.getName()))

throw new InvalidNameException("RegistrationService.INVALID\_NAME");

if (!isValidCity(user.getCity()))

throw new InvalidCityException("RegistrationService.INVALID\_CITY");

if (!isValidEmail(user.getEmail()))

throw new InvalidEmailException("RegistrationService.INVALID\_EMAIL");

if (!isValidPhoneNumber(user.getPhone()))

throw new InvalidPhoneException("RegistrationService.INVALID\_PHONE\_NUMBER");

}

public Boolean isValidUserId(String userid) {

Boolean b1 = false;

Pattern pattern1 = Pattern.compile(regex1);

Matcher matcher1 = pattern1.matcher(userid);

if (matcher1.matches())

b1 = true;

return b1;

}

public Boolean isValidPassword(String password) {

Boolean b1 = false;

String regex2 = "^[a-zA-Z0-9]{8,15}+$";

Pattern pattern2 = Pattern.compile(regex2);

Matcher matcher2 = pattern2.matcher(password);

if (matcher2.matches())

b1 = true;

return b1;

}

public Boolean isValidName(String name) {

Boolean b1 = false;

Pattern pattern3 = Pattern.compile(regex1);

Matcher matcher3 = pattern3.matcher(name);

if (matcher3.matches())

b1 = true;

return b1;

}

public Boolean isValidCity(String city) {

Boolean b1 = false;

Pattern pattern4 = Pattern.compile(regex1);

Matcher matcher4 = pattern4.matcher(city);

if (matcher4.matches())

b1 = true;

return b1;

}

public Boolean isValidEmail(String email) {

Boolean b1 = false;

String regex5 = "^[A-Za-z0-9+\_.-]+@(.+)$";

Pattern pattern5 = Pattern.compile(regex5);

Matcher matcher5 = pattern5.matcher(email);

if (matcher5.matches())

b1 = true;

return b1;

}

public Boolean isValidPhoneNumber(String number) {

Boolean b1 = false;

String regex6 = "[0-9]{10}";

Pattern pattern6 = Pattern.compile(regex6);

Matcher matcher6 = pattern6.matcher(number);

if (matcher6.matches())

b1 = true;

return b1;

}

}

Step 18: Add 'configuration.properties' file under the folder 'src/main/resources' as shown below. It defines the success and failure messages of your application.

#-----------SERVICE EXCEPTIONS-------------------

RegistrationService.INVALID\_USER\_ID=Invalid userId!

RegistrationService.INVALID\_PASSWORD=Invalid password!

RegistrationService.INVALID\_CITY=Invalid city!

RegistrationService.INVALID\_EMAIL=Invalid email id!

RegistrationService.INVALID\_NAME=Invalid name!

RegistrationService.INVALID\_PHONE\_NUMBER=Invalid phone number!

UserRespository.REGISTRATION\_SUCCESS=You are successfully registered with IRS!

Step 19: Rename 'InfyGoBootSpringCoreApplication.java' as 'InfyGoApplication.java' and implement as below. This will be the starter class of your application. It takes care of input and output operations

package com.infosys.irs;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Scanner;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

import com.infosys.irs.model.User;

import com.infosys.irs.service.RegistrationService;

@SpringBootApplication

@PropertySource(value = { "classpath:configuration.properties" })

public class InfyGoApplication implements CommandLineRunner {

@Autowired

private Environment environment;

@Autowired

ApplicationContext context;

public static void main(String[] args) {

SpringApplication.run(InfyGoApplication.class, args);

}

@Override

public void run(String... arg0) throws Exception {

try {

User user = new User();

Scanner sc = new Scanner(System.in);

System.out.println("Enter user id: ");

String uid = sc.next();

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

String pwd = sc.next();

System.out.println("Enter name: ");

String name = sc.next();

System.out.println("Enter city: ");

String city = sc.next();

System.out.println("Enter email: ");

String mail = sc.next();

System.out.println("Enter phone: ");

String phone = sc.next();

user.setCity(city);

user.setUserId(uid);

user.setPassword(pwd);

user.setName(name);

user.setEmail(mail);

user.setPhone(phone);

RegistrationService service = (RegistrationService) context.getBean("registrationService");

String registrationMessage = service.registerUser(user);

System.out.println(environment.getProperty(registrationMessage));

} catch (Exception e) {

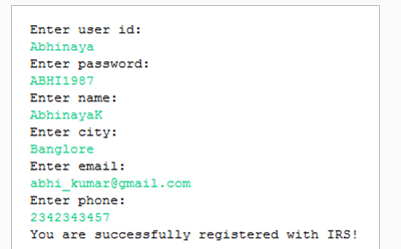
System.out.println(environment.getProperty(e.getMessage()));

}

}

}

Step 20: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App. Observe the input and output on the console as below.



Thus you have implemented the service layer for the User Registration use case of IRS application.

Let us discuss a few important things that you need to observe in this demo.

Things to observe

You created the Spring Boot Maven project structure very easily with the help of Spring initializr. As a developer, you focused only on implementing business functionalities.

Have you realized that you have not done any of the below steps that you typically perform for any other Spring applications?

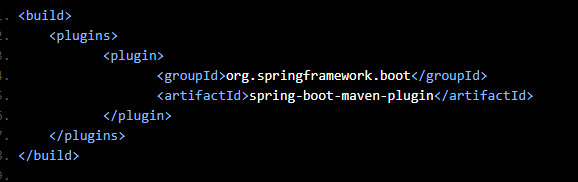
* Adding dependent jar files in the classpath
* Writing XML files for Spring configuration
* Building the application

Then how did it happen?

Adding Spring dependencies and configuring Spring beans taken care of by,

* Spring Boot starters
* @SpringBootApplication

Building the application is taken care of by Maven. Spring Initializr provides a ready-to-use Maven pom.xml to package the application as Jar or War. Open the pom.xml file in your demo and observe the build tag given below.



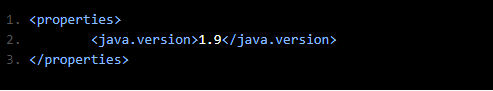
Apart from the build configuration, your pom.xml file is configured with a lot more dependencies as well. You will be seeing this in detail while we discuss starters and @SpringBootApplication annotation

**Spring-boot-starter-parent**

Spring Boot offers various starters to insert jars to the classpath easily. The spring-boot-starter-parent is a main starter defined in the <parent/> section of pom.xml file. Dependencies inherited from here using Maven build.

| Spring-boot-starter-parent | |
| --- | --- |
| How to configure? | <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.1.RELEASE</version>  </parent> |
| Advantages | * It provides the Maven defaults like Java 1.8 as the default compiler level, plugin configuration (Exec, Surefire, etc.), UTF-8 source encoding and etc. * It determines the version number of other Spring dependencies. Changing the parent's version will implicitly change the version of all other dependencies. |

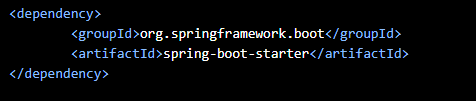
The Maven defaults provided by spring-boot-starter-parent is possible to override. Forex, to change the Java version, you can easily add a java-version property in your pom.xml as below.



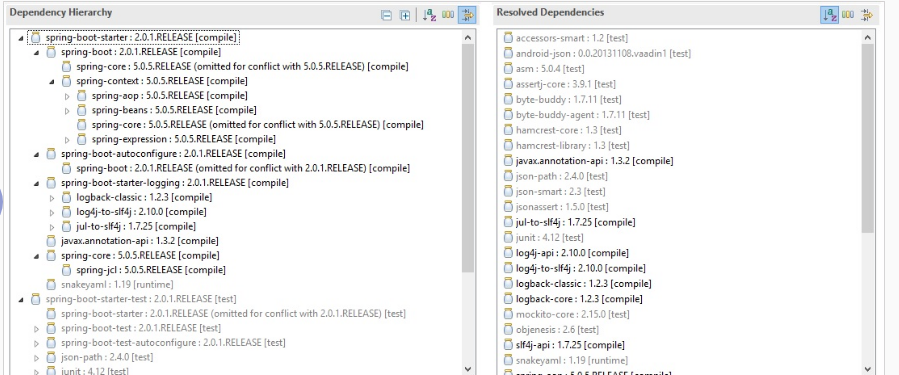
You have already seen the parent starter of Spring Boot. Now let us discuss other starters of Spring Boot.

The starters combine all the related dependencies under a single dependency. It is a one-stop-shop for all required Spring related technologies without browsing through the net and downloading them individually.

The core starter of Spring Boot is spring-boot-starter. It informs Spring Boot that it is a Spring core application.



Note that the version number of spring-boot-starter is not defined here as it can be determined by spring-boot-starter-parent. By opening the Dependency Hierarchy tab of your pom.xml, you can see that this single starter POM combines four more dependencies in it.



All these dependencies together resolve almost 16 jars (like spring-boot-\*.jar, spring-core-\*.jar, spring-context-\*.jar, etc.).

Spring Boot supports many such starters POMs (like spring-boot-starter-test, spring-boot-starter-web, etc.) to resolve much more dependencies. You will learn about a few of them as you progress.

**@SpringBootApplication**

Every Spring Boot project created by Spring initializr will have a main class annotated with @SpringBootApplication. In your case, apply annotation on com.infosys.irs.InfyGoApplication class. This annotation is equivalent to declaring @ComponentScan, @Configuration, and @EnableAutoConfiguration annotations with their defaults.

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

# **@Configuration**

This annotation tags a class as a configuration class. This annotation is equivalent to defining beans in an XML configuration file.

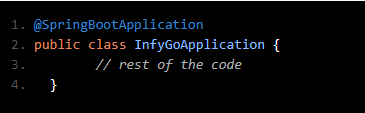
# **@EnableAutoConfiguration**

This annotation auto-configures all the beans based on classpath dependencies. For example, since you have included spring-core dependency in your classpath via spring-boot-starter, the auto-configuration assumes that you are creating a Spring core application and set up Spring accordingly. Refer to Spring documentation on Spring's official site to know more about this annotation.

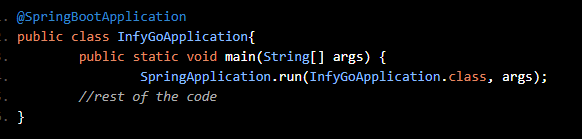
# **@ComponentScan**

This annotation scans the base package (the package under which this class resides) and creates beans of other annotated java components like components, configurations, and services. This is equivalent to Spring's XML configuration <context:component-scan/>.

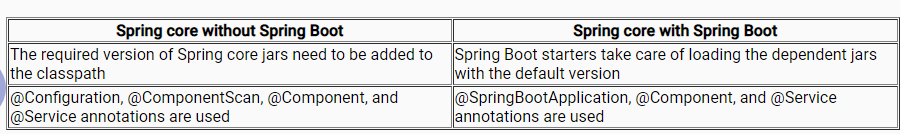
You can override the base package value by providing the scanBasePackages attribute as below:



The class that is annotated with @SpringBootApplication will be considered as the main class, is also a bootstrap class. It kicks starts the application by invoking the SpringApplication.run() method. You need to pass the .class file name of your main class to the run() method as shown below.



The above code informs the Spring framework that SpringBootArsApplication is the primary Spring component. The execution is kick-started from here.

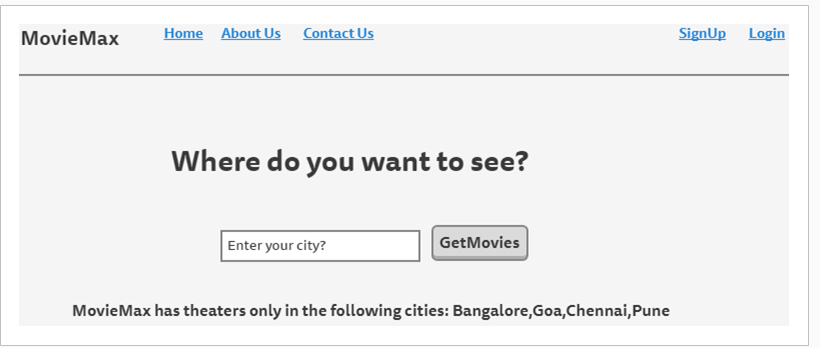


**MovieMax Online movie booking application**

MovieMax is an online movie booking application that provides services to its users to search and book movie tickets.

As part of the Spring Boot exercise, you are going to develop this application in an incremental way i.e. each subsequent exercise built on top of the previous one. By the end of all exercises, you will have an application demonstrating all Spring boot features learned during this course.

Sample UI for MovieMax application given below.

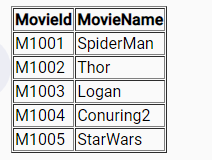


Implementing service layer with Spring core

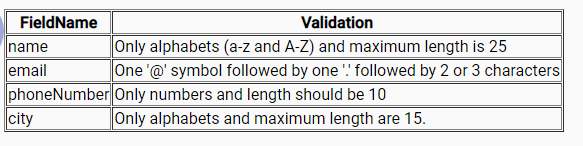
## **Problem Statement:**

Implement the service layer of the Spring Boot application MovieMax.

* The classes for model, service, and main boot application should be in different packages.
* Store the following movie list in a collection.



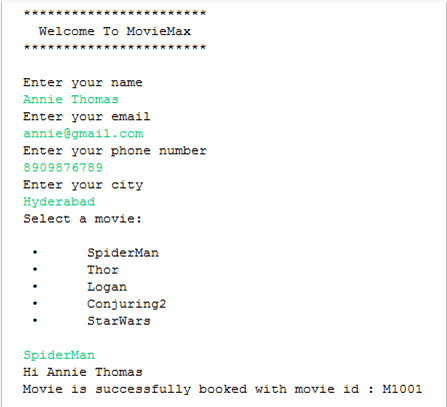
The application should prompt the user to enter the following details and validate as per the given criteria:



* Display available movie names from the collection and prompt the user to select one. Do a case insensitive comparison to identify the movie Id.
* Display a success message if all validations are successful.

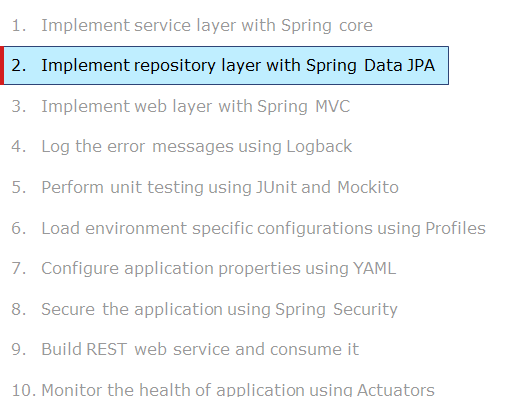
Verification:

* On execution, the application should display the following messages and prompt the user for ticket booking. The appropriate error message should come from invalid data entered by the user.



# Implement repository layer with Spring Data JPA

While developing the service layer of the User Registration use case, you have used a stub for the repository layer. Now, let us discuss how to replace the stub with actual code to perform the database operations using Spring Data JPA API.



# Demo: Implementing repository layer with Spring Data JPA

Highlights:

* How to implement Spring Data JPA with Spring Boot project.
* How to connect to MySQL database and insert a row in the existing table

Demo steps:

Step 1: Create the table 'user\_details' under the database 'db\_irs' and insert rows using the script below:

create table user\_details(

userid varchar(20) primary key,

password varchar(20),

name varchar(30),

city varchar(30),

email varchar(30),

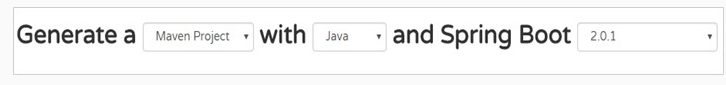
phone varchar(10));

insert into user\_details values("C1002","John1234","John","Chennai","John@gmail.com","9878909876");

insert into user\_details values("C1003","Ruby12345","Ruby","Chennai","Ruby@gmail.com","7890987656");

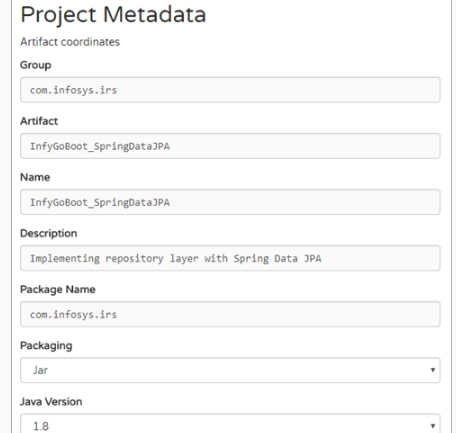
Step 2: Go to the Spring Initializr website.

Step 3: Generate a 'Maven' project with Spring Boot '2.0.1'

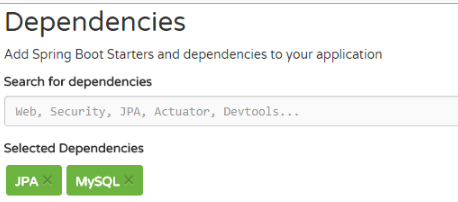


Step 4: Click on 'Switch to the full version' to provide further details.

Step 5: Fill in 'Project Metadata' details as shown below.



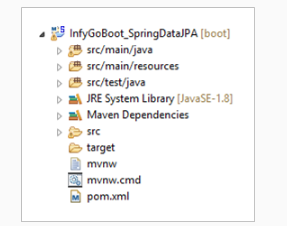
Step 6: Add the 'JPA' and 'MySQL' dependencies as shown below.



Step 7: Click on 'Generate Project'.

Step 8: Save the ZIP file in your system at any location and unzip it.

Step 9: Open STS and import the project created above by selecting the File menu and choosing Import -> Existing Maven Project -> Browse to the application and Finish. You can see the project structure below.



Step 10: Create the Java files with the package structure as shown below. Rename 'com.infosys.irs.InfyGoBootSpringDataJPAApplication.java' as 'com.infosys.irs.InfyGoApplication.java'.

package com.infosys.irs;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Scanner;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

import com.infosys.irs.model.User;

import com.infosys.irs.service.RegistrationService;

@SpringBootApplication

@PropertySource(value = { "classpath:configuration.properties" })

public class InfyGoApplication implements CommandLineRunner {

@Autowired

private Environment environment;

@Autowired

ApplicationContext context;

public static void main(String[] args) {

SpringApplication.run(InfyGoApplication.class, args);

}

@Override

public void run(String... arg0) throws Exception {

try {

User user = new User();

Scanner sc = new Scanner(System.in);

System.out.println("Enter user id: ");

String uid = sc.next();

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

String pwd = sc.next();

System.out.println("Enter name: ");

String name = sc.next();

System.out.println("Enter city: ");

String city = sc.next();

System.out.println("Enter email: ");

String mail = sc.next();

System.out.println("Enter phone: ");

String phone = sc.next();

user.setCity(city);

user.setUserId(uid);

user.setPassword(pwd);

user.setName(name);

user.setEmail(mail);

user.setPhone(phone);

RegistrationService service = (RegistrationService) context.getBean("registrationService");

String registrationMessage = service.registerUser(user);

System.out.println(environment.getProperty(registrationMessage));

} catch (Exception e) {

System.out.println(environment.getProperty(e.getMessage()));

}

}

}

Step 11: Reuse 'User.java', 'InvalidCityException.java', 'InvalidEmailException.java'', 'InvalidNameException.java,' 'InvalidPasswordException.java', 'InvalidPhoneException.java' and ''InvalidUserIdException.java from Spring Core demo.

Step 12: Implement 'UserEntity.java' as below. This class represents the user\_details table in the database.

package com.infosys.irs.entity;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "USER\_DETAILS")

public class UserEntity {

@Id

@Column(name = "userid")

private String userId;

private String password;

private String name;

private String city;

private String email;

private String phone;

public String getPassword() {

return password;

}

public String getName() {

return name;

}

public String getCity() {

return city;

}

public String getEmail() {

return email;

}

public String getPhone() {

return phone;

}

public void setPassword(String password) {

this.password = password;

}

public void setName(String name) {

this.name = name;

}

public void setCity(String city) {

this.city = city;

}

public void setEmail(String email) {

this.email = email;

}

public void setPhone(String phone) {

this.phone = phone;

}

public String getUserId() {

return userId;

}

public void setUserId(String userId) {

this.userId = userId;

}

}

Step 13: Implement 'UserIdAlreadyPresentException.java' as below.

package com.infosys.irs.exception;

/\*\*

\* If the given userId is present then this exception will be thrown.

\*

\*/

public class UserIdAlreadyPresentException extends InfyGoBootException {

private static final long serialVersionUID = 1L;

public UserIdAlreadyPresentException(String message) {

super(message);

}

}

Step 14: Implement 'UserRepository.java' as below. Database operations like create, insert, update, and delete will be performed here.

package com.infosys.irs.repository;

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

import com.infosys.irs.entity.UserEntity;

public interface UserRepository extends JpaRepository<UserEntity, String> {

}

Step 15: Implement 'RegistrationService.java' as below. It validates the user object and passes the validated user to 'UserRepository.java' for insertion.

package com.infosys.irs.service;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

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

import org.springframework.stereotype.Service;

import com.infosys.irs.entity.UserEntity;

import com.infosys.irs.exception.InfyGoBootException;

import com.infosys.irs.exception.InvalidCityException;

import com.infosys.irs.exception.InvalidEmailException;

import com.infosys.irs.exception.InvalidNameException;

import com.infosys.irs.exception.InvalidPasswordException;

import com.infosys.irs.exception.InvalidPhoneException;

import com.infosys.irs.exception.InvalidUserIdException;

import com.infosys.irs.exception.UserIdAlreadyPresentException;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

@Service

public class RegistrationService {

@Autowired

private UserRepository userRepository;

String regex1 = "^[a-zA-Z0-9]{4,15}+$";

public String registerUser(User user) throws InfyGoBootException {

validateUser(user);

boolean b = userRepository.existsById(user.getUserId());

if (b)

throw new UserIdAlreadyPresentException("RegistrationService.USERID\_PRESENT");

UserEntity userEntity = new UserEntity();

userEntity.setCity(user.getCity());

userEntity.setEmail(user.getEmail());

userEntity.setName(user.getName());

userEntity.setPassword(user.getPassword());

userEntity.setPhone(user.getPhone());

userEntity.setUserId(user.getUserId());

userRepository.saveAndFlush(userEntity);

return "UserRespository.REGISTRATION\_SUCCESS";

}

public void validateUser(User user) throws InfyGoBootException {

if (!isValidUserId(user.getUserId()))

throw new InvalidUserIdException("RegistrationService.INVALID\_USER\_ID");

if (!isValidPassword(user.getPassword()))

throw new InvalidPasswordException("RegistrationService.INVALID\_PASSWORD");

if (!isValidName(user.getName()))

throw new InvalidNameException("RegistrationService.INVALID\_NAME");

if (!isValidCity(user.getCity()))

throw new InvalidCityException("RegistrationService.INVALID\_CITY");

if (!isValidEmail(user.getEmail()))

throw new InvalidEmailException("RegistrationService.INVALID\_EMAIL");

if (!isValidPhoneNumber(user.getPhone()))

throw new InvalidPhoneException("RegistrationService.INVALID\_PHONE\_NUMBER");

}

public Boolean isValidUserId(String userid) {

Boolean b1 = false;

Pattern pattern1 = Pattern.compile(regex1);

Matcher matcher1 = pattern1.matcher(userid);

if (matcher1.matches())

b1 = true;

return b1;

}

public Boolean isValidPassword(String password) {

Boolean b1 = false;

String regex2 = "^[a-zA-Z0-9]{8,15}+$";

Pattern pattern2 = Pattern.compile(regex2);

Matcher matcher2 = pattern2.matcher(password);

if (matcher2.matches())

b1 = true;

return b1;

}

public Boolean isValidName(String name) {

Boolean b1 = false;

Pattern pattern3 = Pattern.compile(regex1);

Matcher matcher3 = pattern3.matcher(name);

if (matcher3.matches())

b1 = true;

return b1;

}

public Boolean isValidCity(String city) {

Boolean b1 = false;

Pattern pattern4 = Pattern.compile(regex1);

Matcher matcher4 = pattern4.matcher(city);

if (matcher4.matches())

b1 = true;

return b1;

}

public Boolean isValidEmail(String email) {

Boolean b1 = false;

String regex5 = "^[A-Za-z0-9+\_.-]+@(.+)$";

Pattern pattern5 = Pattern.compile(regex5);

Matcher matcher5 = pattern5.matcher(email);

if (matcher5.matches())

b1 = true;

return b1;

}

public Boolean isValidPhoneNumber(String number) {

Boolean b1 = false;

String regex6 = "[0-9]{10}";

Pattern pattern6 = Pattern.compile(regex6);

Matcher matcher6 = pattern6.matcher(number);

if (matcher6.matches())

b1 = true;

return b1;

}

}

Step 16: Add properties in the 'src/main/resources/application.properties' file as below to bind the database credentials. (Note: Username and password should be given as per your database account details)

spring.datasource.url = jdbc:mysql://localhost:3306/db\_irs

spring.datasource.username = root

spring.datasource.password = root

Step 17: Add the 'configuration.properties' file under the folder 'src/main/resources' as shown below to configure the success and failure messages of your application.

#-----------SERVICE EXCEPTIONS-------------------

RegistrationService.INVALID\_USER\_ID=Invalid userId!

RegistrationService.INVALID\_PASSWORD=Invalid password!

RegistrationService.INVALID\_CITY=Invalid city!

RegistrationService.INVALID\_EMAIL=Invalid email id!

RegistrationService.INVALID\_NAME=Invalid name!

RegistrationService.INVALID\_PHONE\_NUMBER=Invalid phone number!

UserRespository.REGISTRATION\_SUCCESS=You are successfully registered with IRS!

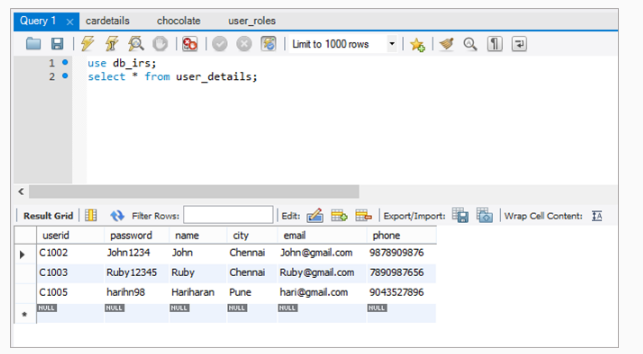
RegistrationService.USERID\_PRESENT=Given user id is already available. Please choose a different one!

Step 18: Reuse 'InfyGoApplication.java' from Demo 2.

Step 19: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App. Observe the input and outcome on the console as displayed below.



Step 20: Go to MySQL database and see the row inserted in the table as shown below.



Thus you have implemented the repository layer with Spring Data JPA for the User Registration use case of IRS application.

Let us discuss a few important things that you need to observe in this demo.

**Things to observe**

Have you realized that you have not performed any of the below activities which you typically perform for Spring Data JPA applications?

* Adding jars related to data persistence in the classpath
* Configuring EntityManager, TransactionManager, JpaVendorAdaptor, and DataSource
* Enabling JPA repositories

Then how were you able to insert a record into the database?

The spring-boot-starter-data-jpa starter POM takes care of it.

**Spring-boot-starter-data-jpa**

# **Adding data persistence related jars**

As you select the JPA dependency in Spring Initializer, the pom.xml is included with spring-boot-starter-data-jpa dependency. This starter informs the Spring Boot that this application enabled with data persistence and it provides the following dependencies.

* Hibernate ORM as JPA implementation
* Spring Data JPA for repositories
* Spring ORM for core support

# **Configuring data persistence beans**

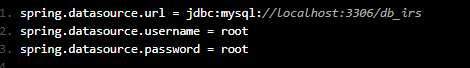
In the presence of spring-boot-starter-data-jpa, by default Spring Boot creates beans for EntityManager, TransactionManager, and JPAVendorAdaptor with its defaults. If you wish to override them, need to configure any of these beans.

# **Enabling JPA repositories**

In the presence of spring-boot-starter-data-jpa POM, Spring Boot automatically looks for JPA repository definitions in (the same or sub-package of @SpringBootApplication class and will enable them. Therefore, you don't need to manually enable the JPA repositories either through XML or through @EnableJpaRepositories annotation.

Refer Spring site to know more about customizing data access properties.

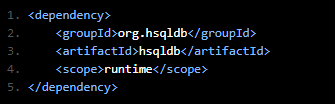
As you are using MySQL as your data source, you have to inform its location to Spring Boot. This could declare via properties file named application.properties file as below.



This file should be located under src/main/resources folder in your application. (Note: There are more places where you can add this file that you will learn about later)

# **Embedded database support**

Spring Boot supports in-memory embedded databases like H2, HSQL, and Derby. Therefore, actually, you don't need to provide any connection URL's for your data source as shown above, if you work with any of these embedded databases. You have to include just the build dependency for them. For example, if you want to connect with the HSQL database, include the below dependency in POM.



Populate the data every time the application is started and is lost when the application stops.

| **Spring Data JPA without Spring Boot** | **Spring Data JPA with Spring Boot** |
| --- | --- |
| The required version of Spring Data JPA jars needs to be added to the classpath | Spring-boot-starter-data-jpa takes care of loading the dependent jars with a default version |
| @EnableJpaRepositories is required | Not required, as JPA repositories will automatically be enabled. |
| Configuration of EntityManager, TransactionManager, and JpaVendorAdapter is required in the configuration file | It will be auto-configured |
| Configuration of DataSource is required in the configuration file | Configuration of the data source is not required in case if you are working with any of the embedded databases. You just need to provide the build dependency for the embedded database that you want. |

Exercise: Implementing repository layer with Spring Data JPA

Problem Statement:

Modify the MovieMax - Exercise 1 to use Spring Data JPA, it helps to persist the data in MySQL DB.

Theater details and movie details fetched from the theater table and movie table.

Movie booking should be updated with corresponding details in the below tables.

theatermovieshow table

booking table

The below table script helps to create the required tables:

create table user(userid varchar(8) primary key, username varchar(25), password varchar(10), emailid varchar(30), phone int);

create table theater(theaterid varchar(8) primary key, theatername varchar(30), seatsavailable int(3), city varchar(25));

create table movie(movieid varchar(8) primary key, moviename varchar(40), language varchar(15), category varchar(20), islive boolean);

create table theatermovieshow( showid varchar(8) primary key,

theaterid varchar(8) references theater(theaterid),

movieid varchar(8) references movie(movieid),

showtime varchar(8),

startdate date,

enddate date,

ticketrate double(6,2));

create table booking(bookingid int(5), userid varchar(8) references user(userid),

showid varchar(8) references theatermovieshow(showid),

noofseats int,

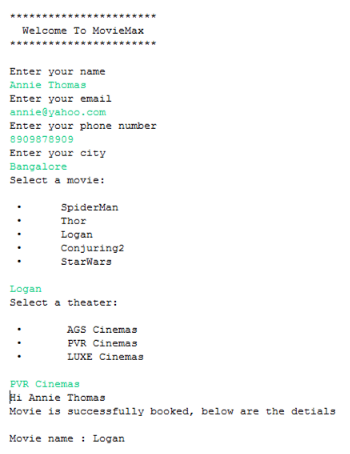
amountpaid double(8,2),

showdate date,

bookingdate date);

Verification:

* On execution, the details fetched from the database.
* If the booking is a success then check the corresponding tables to verify whether the details are persisted or not.



# Customize Spring Boot configuration

By now, you might have realized that auto-configuration is one of the important features in Spring Boot as it configures your Spring Boot application according to your classpath. However, sometimes, you may need to override auto-configuration with explicit configuration.

For example, Spring Boot auto-configures the data source to an embedded database, but how did you configure MySQL as your data source?

You have overridden the default data source by providing an application.properties file externally.

Is it the only way to customize the configuration?

Spring Boot has different options to provide explicit/external configuration. Few of them are:

* Using the application.properties file
* Using application.yml file (Will be discussed later)
* Using environment variables (Not discussed in this course)
* Using command-line arguments (Not discussed in this course)

There are different options (Refer Spring site) to externalize the configuration.

Remember that if none of the above options is used, Spring Boot already has default values for all those properties.

Let us see how to implement the application.properties file to customize Spring Boot configuration.

Additional Reference: Complete list of common application properties (Refer to Spring site).

**Application property files**

Spring Boot looks for application.properties or YAML files in the below locations.

1. A */config* subdirectory of the current directory
2. The current directory
3. A *classpath/config* package
4. The classpath root (In Maven project Spring Boot looks under *src/main/resources* folder)

The above list is ordered by priority (1 as highest and 4 as lowest). Properties defined in locations upper in the list override with those defined in lower locations.

For example, let us say you have an application.properties file under classpath root (*src/main/resources*) folder defining the root logger level as INFO and another application.properties file under *classpath /config* folder (*src/main/resources/config*) defining the root logger level as WARN. Here, the final level of root logger will be WARN. You can confirm this by injecting this property value into your Spring bean class as shown below.

package com.infosys.irs.model;

import org.springframework.stereotype.\*

import org.springframework.beans.factory.annotation.\*

@Component

public class TestBean {

@Value("${logging.level.root}")

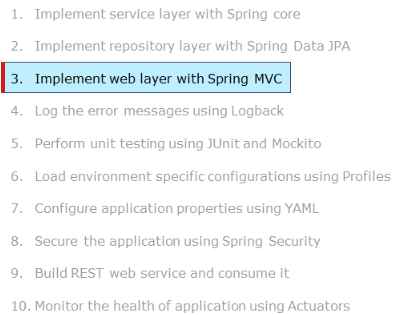
private String rootLogLevel;

// ...

}

# Implement web layer with Spring MVC

As of now, your IRS application is a console-based application. Let us see how to make it an online application by implementing a web layer for it. You are going to use Spring MVC for building the web layer.



# Demo: Implement web layer with Spring MVC

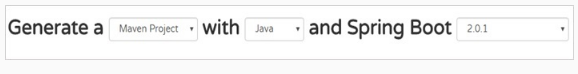
Highlights:

* How to create Spring Boot project structure for web application using Spring Initializr
* How to integrate Spring Boot with Spring MVC

Demosteps:

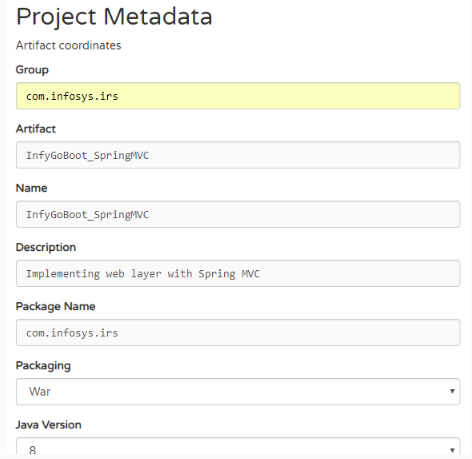
Step 1: Go to the Spring Initializr site.

Step 2: Generate a 'Maven' project with Spring Boot '2.0.1'.



Note: Click on 'Switch to the full version' to provide further details.

Step 3: Fill in the 'Project Metadata' details as shown below.



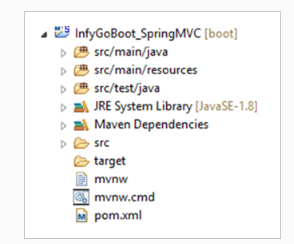
Step 4: Along with the dependencies that you have used for the repository layer ( Spring Data JPA, MYSQL) add the dependency for 'web' which will enable 'validation' as well.



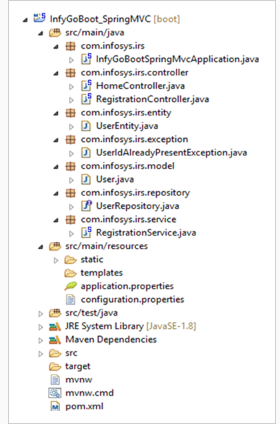
Step 5: Click on 'Generate Project'.

Step 6: Save the zip file in your system at any location and unzip it.

Step 7: Open STS and import the project created above by selecting the File menu and choosing Import -> Existing Maven Project -> Browse to the application and Finish. You can see the project structure below.



Step 8: Create the 'Java files' with the package structure as shown below.



Step 9: Reuse 'UserEntity.java', 'UserIdAlreadyPresentException.java' and 'UserRepository.java' from Demo 3.

Step 10: Implement 'User.java' as shown below.

package com.infosys.irs.model;

import javax.validation.constraints.Email;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

public class User {

@NotNull(message = "UserId must not be blank.")

@Size(min = 4, max = 15, message = "UserId must be between 4 to 15 Characters.")

private String userId;

@NotNull(message = "Password must not be blank.")

@Size(min = 8, max = 15, message = "Password must be between 8 to 15 Characters.")

private String password;

@NotNull(message = "Name must not be blank.")

@Size(min = 4, max = 15, message = "Name must be between 4 to 15 Characters.")

private String name;

@NotNull(message = "City must not be blank.")

private String city;

@NotNull(message = "Email must not be blank.")

@Email

private String email;

@NotNull(message = "PhoneNumber must not be blank.")

@Size(min = 10, max = 10, message = "PhoneNumber must be 10 digits.")

private String phone;

public String getUserId() {

return userId;

}

public void setUserId(String userId) {

this.userId = userId;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getPhone() {

return phone;

}

public void setPhone(String phone) {

this.phone = phone;

}

}

Step 11: Implement 'RegistrationService.java' as shown below.

package com.infosys.irs.service;

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

import org.springframework.stereotype.Service;

import com.infosys.irs.entity.UserEntity;

import com.infosys.irs.exception.UserIdAlreadyPresentException;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

@Service

public class RegistrationService {

@Autowired

private UserRepository userRepository;

public void registerUser(User user) throws UserIdAlreadyPresentException {

boolean ue = userRepository.existsById(user.getUserId());

if (ue)

throw new UserIdAlreadyPresentException("RegistrationService.USERID\_PRESENT");

UserEntity userEntity = new UserEntity();

userEntity.setCity(user.getCity());

userEntity.setEmail(user.getEmail());

userEntity.setName(user.getName());

userEntity.setPassword(user.getPassword());

userEntity.setPhone(user.getPhone());

userEntity.setUserId(user.getUserId());

userRepository.saveAndFlush(userEntity);

}

}

Step 12: Implement 'HomeController.java' as shown below.

package com.infosys.irs.controller;

import org.springframework.stereotype.Controller;

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class HomeController {

@GetMapping(value = "/")

public ModelAndView getHomeDetails() {

return new ModelAndView("infyGoHome", "", "");

}

}

Step 13: Implement 'RegistrationController.java' as shown below. It receives the HTTP request and invokes the Service class for business logic. As per the result of business logic, it returns a view.

package com.infosys.irs.controller;

import javax.validation.Valid;

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

import org.springframework.core.env.Environment;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.ui.ModelMap;

import org.springframework.validation.BindingResult;

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

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

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

import org.springframework.web.servlet.ModelAndView;

import com.infosys.irs.exception.UserIdAlreadyPresentException;

import com.infosys.irs.model.User;

import com.infosys.irs.service.RegistrationService;

@Controller

public class RegistrationController {

@Autowired

private RegistrationService registrationService;

@Autowired

private Environment environment;

private String command = "command";

private String register = "register";

@GetMapping(value = "/register")

public ModelAndView register(Model model) {

return new ModelAndView(register, command, new User());

}

@PostMapping(value = "registerUser")

public ModelAndView addCustomer(@Valid @ModelAttribute("command") User user, BindingResult result, ModelMap model) {

ModelAndView modelAndView = new ModelAndView();

if (result.hasErrors()) {

modelAndView = new ModelAndView(register, command, user);

} else {

try {

registrationService.registerUser(user);

modelAndView = new ModelAndView(register, command, user);

modelAndView.addObject("successMessage",

environment.getProperty("RegistrationController.SUCCESSFUL\_REGISTRATION"));

} catch (UserIdAlreadyPresentException e) {

if (e.getMessage().contains("RegistrationService")) {

modelAndView = new ModelAndView(register);

modelAndView.addObject(command, user);

modelAndView.addObject("message", environment.getProperty(e.getMessage()));

}

}

}

return modelAndView;

}

}

Step 14: Implement 'InfyGoBootSpringMvcApplication.java' as shown below.

package com.infosys.irs;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.PropertySource;

@SpringBootApplication

@PropertySource(value = { "classpath:configuration.properties" })

public class InfyGoBootSpringMvcApplication {

public static void main(String[] args) {

SpringApplication.run(InfyGoBootSpringMvcApplication.class, args);

}

}

Step 15: Add the properties in 'src/main/resources/application.properties' file as displayed below. (Note: Username and password should be the same as per your database account details)

server.port = 4343

server.servlet.context-path=/InfyGoBoot

spring.mvc.view.prefix = /WEB-INF/pages/

spring.mvc.view.suffix = .jsp

spring.datasource.url = jdbc:mysql://localhost:3306/db\_irs

spring.datasource.username = root

spring.datasource.password = root

Step 16: Add the 'configuration.properties' file under the folder 'src/main/resources' as shown below.

#-----------SERVICE EXCEPTIONS-------------------

RegistrationService.USERID\_PRESENT=Given user id is already available. Please choose a different one!

#---------------------SUCCESS MESSAGES --------------------------------

RegistrationController.SUCCESSFUL\_REGISTRATION=You are successfully registered with InfyGo. Use your user id and password to login!

Step 17: Reuse the static resources like 'css', 'js', 'imgs' and 'images' from existing non-Spring Boot IRS application and place them inside 'src/main/resources/static' folder so that Spring Boot can pick up them automatically.

Step 18: Reuse the '.jsp' files 'infyGoHome.jsp', 'register.jsp', 'header.jsp' and 'footer.jsp' from existing non Spring Boot IRS application and place them inside 'src/main/webapp/WEB-INF/pages' folder.

Step 19: Change the 'commandName' attribute of <form:form > tag in 'register.jsp' as 'modelAttribute' .

Step 20: Add the dependency in 'pom.xml file' as shown below.

<dependency>

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

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

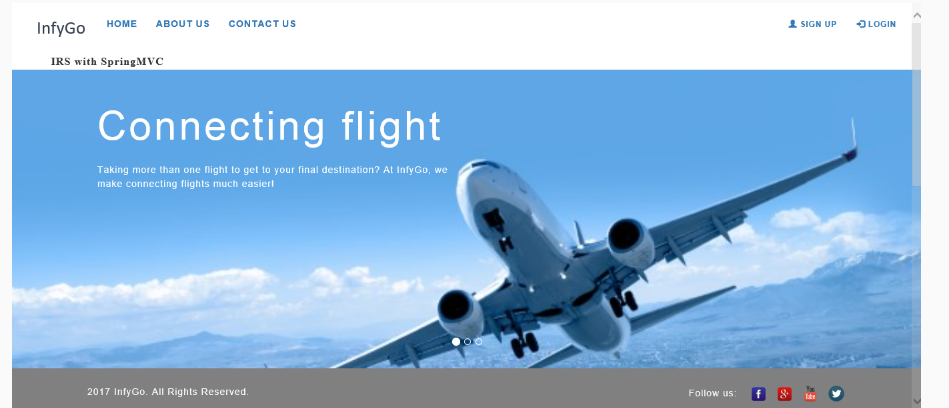
<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

Step 21: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App.

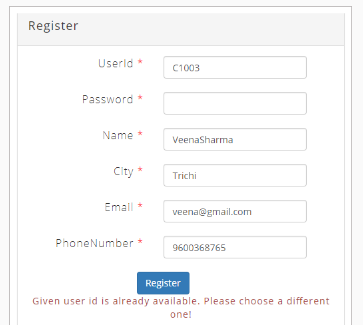
Step 22: Type the URL 'http://localhost:4343/InfyGoBoot' in the address bar of your browser. The 'infyGoHome.jsp' will be rendered for you that will have the 'SIGN UP' option as shown below.



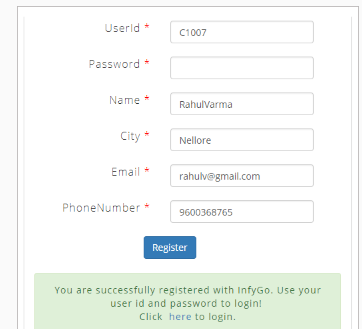
Step 23: Click on 'SIGN UP', which will render the 'register.jsp' as shown below.



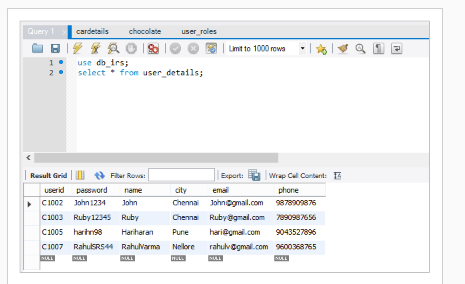
Step 24: If the given userId exists already, observe the 'register.jsp' page which will be rendered as shown below.



Step 25: If the given userId does not exists already, observe the 'register.jsp' page which will be rendered as shown below.



Step 26: Go to MySQL database and see the row inserted in the table as shown below.



Thus, you have implemented the web layer for the User Registration use case of IRS application.

Let us discuss few important things that you need to observe in this demo.

**Things to observe**

Have you realized that you have not done any of the below steps that you typically perform for any other SpringMVC applications?

* Adding dependent jar files in the classpath
* Writing XML files for Spring configuration
* Writing web.xml for deployment description
* Writing dispatcher-servlet.xml for Spring MVC configuration
* Deploying the application in an application server

Then how did it happen?

You already know that dependency management and Spring configuration are taken care of by Spring Boot starters and @SpringBootApplication.

Let us discuss,

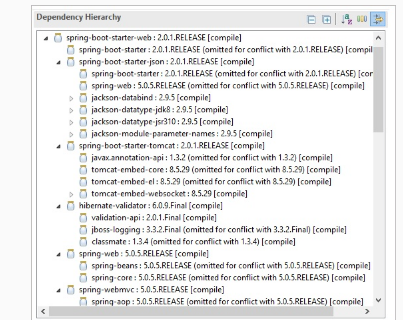
* Which starter is taking care of dependencies required for web applications?
* How the deployment description configured without web.xml?
* How the view resolvers configured without dispatcher-servlet.xml?
* Where is the application deployed?

**Spring-boot-starter-web**

As you select the web dependency in Spring Initializer, the pom.xml is added with spring-boot-starter-web dependency. This starter informs Spring Boot that this is a web application. Spring Boot forms the below opinion for your application accordingly.

* Embedded Tomcat container for deployment
* Hibernate for ORM
* Apache Jackson for JSON binding
* Spring MVC for web development

It delivers the below dependencies for your application.

****

**No XML files**

# No web.xml

The spring-boot-starter-web auto-configures the DispatcherServlet with its default. So, unless you wish to override the defaults, you must not need to use web.xml to configure the DispatcherServlet.

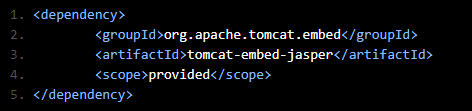
# No dispatcher-servlet.xml

Spring Boot will automatically pick up all the views if the views placed in the src/main/resources/templates folder. Therefore, you must not need any dispatcher-servlet.xml to configure view resolvers. But, it will work with ThymeLeaf, FreeMarker, Groovy, and Mustache template engines but not with JSP as Spring Boot has certain limitations with JSP.

Anyway, do not worry, still, you do not need a dispatcher-servlet.xml. The view resolver for JSP files configured in the application.properties file as below.



Insert the below dependency in the pom.xml file so that the embedded Tomcat container can process the view resolver.



Note: Packaging of the application done as war due to Spring Boot's limitation with JSP. Refer JSP limitations in Spring documentation to know further.

**Deployment**

The spring-boot-starter-web provides an embedded Tomcat container to run your application. Therefore, you do not need to maintain an external container to deploy your application.

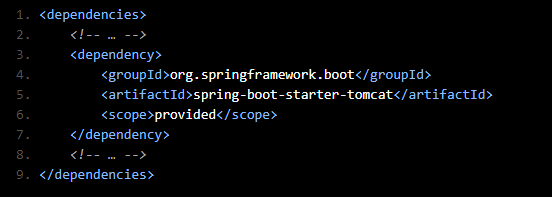
The embedded container will listen to the port on 8080 by default. . It is customized by providing your own server.port value in the application.properties file. The context path of the application can also be changed as below.

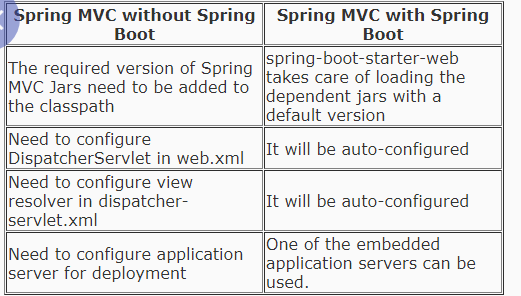


With the above properties, your application accessed by providing the URL http://localhost:4646/InfyGoBoot

# **Deploying into an external container**

Instead of using the embedded container, if you want to deploy your war file into an external container, you have to mark the embedded Tomcat container dependency as *<scope>provided</scope>* so that the embedded container does not clash with the external container.





# Exercise: Implement web layer with Spring MVC

# **Problem Statement:**

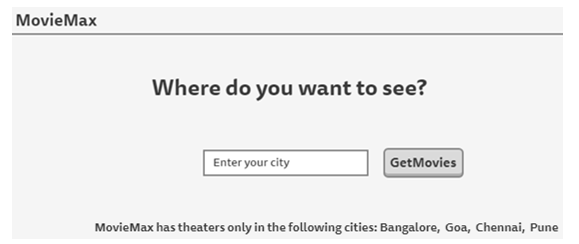
Modify exercise 2 the Spring boot application MovieMax to implement web layer using Spring MVC.

* Input validations need to be done with annotations.
* Create appropriate JSP files for each of the business functionalities.
* Success and failure messages must not hardcoded. Use the Properties file for the same.
* The user selects the movie with the preferable date and time. If seats are available for the selected movie and time slot, booking details are displayed to the user accordingly.

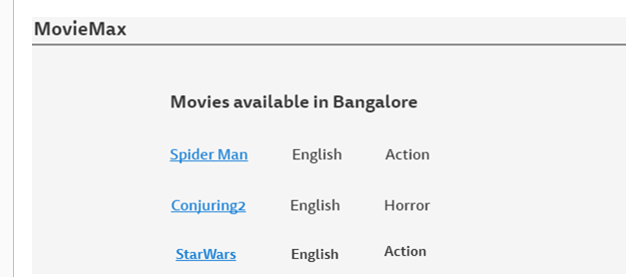
Verification:

* On execution, the application should prompt the user for a movie booking.

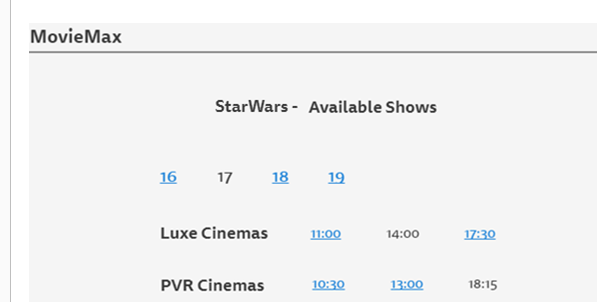
Sample UI is given below.



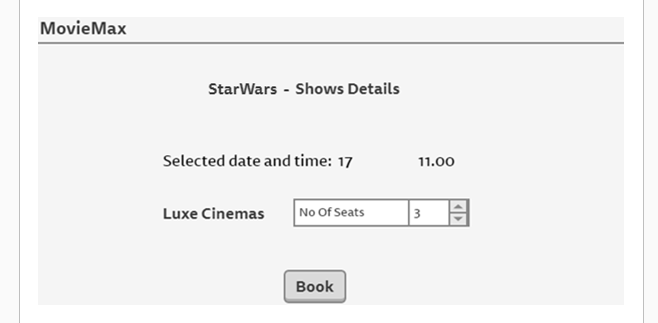
* After entering a valid city name, movie details for the given city will be rendered.



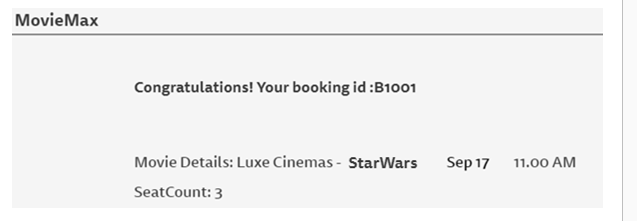
* On clicking a movie name, the dates and corresponding theater details will be rendered.



* On selecting a preferred date and showtime, the user prompted to enter a required number of seats.

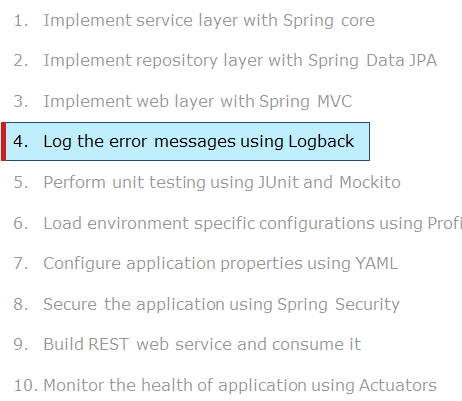


* The User has to select the number of seats and click on Book to confirm the booking.
* On booking, an appropriate success or error message should be displayed.



# Log the error messages using Logback

You know that logging is one of the important activities in any application. It helps in quick problem diagnosis, debugging, and maintenance. Let us learn the logging configuration in Spring Boot.



Spring Boot logging

While executing your Spring Boot application, have you seen things like the below getting printed on your console?



And guess what are these.

Yes, you are right. These are logging messaged logged on INFO level. However, you haven't written any code for logging in to your application. Then who does this?

By default, Spring Boot configures logging via Logback to log the activities of libraries that your application uses.

As a developer, you may want to log the information that helps in quick problem diagnosis, debugging and maintenance. So, let us see how to customize the default logging configuration of Spring Boot so that your application can log the information that you are interested in and in your own format.

# Demo: Log the error messages using LogbackHighlights:

# How to log the information of a Spring Boot application?

# How to log the exception messages using Logback?

# Demo steps:

# Step 1: Open STS and import the IRS project named 'InfyGoBoot\_SpringMVC' created in Demo 4. Rename the project as 'InfyGoBoot\_Logback'.

# Step 2: Create a Logger object and raise a log request in RegistrationController.java as shown below.

# package com.infosys.irs.controller;

# import javax.validation.Valid;

# import org.slf4j.Logger;

# import org.slf4j.LoggerFactory;

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

# import org.springframework.core.env.Environment;

# import org.springframework.stereotype.Controller;

# import org.springframework.ui.Model;

# import org.springframework.ui.ModelMap;

# import org.springframework.validation.BindingResult;

# import org.springframework.web.bind.annotation.GetMapping;

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

# import org.springframework.web.bind.annotation.PostMapping;

# import org.springframework.web.servlet.ModelAndView;

# import com.infosys.irs.exception.UserIdAlreadyPresentException;

# import com.infosys.irs.model.User;

# import com.infosys.irs.service.RegistrationService;

# @Controller

# public class RegistrationController {

# @Autowired

# private RegistrationService registrationService;

# @Autowired

# private Environment environment;

# private String command = "command";

# private String register = "register";

# @GetMapping(value = "/register")

# public ModelAndView register(Model model) {

# return new ModelAndView("register", "command", new User());

# }

# @PostMapping(value = "registerUser")

# public ModelAndView addCustomer(@Valid @ModelAttribute("command") User user, BindingResult result, ModelMap model) {

# final Logger logger;

# ModelAndView modelAndView = new ModelAndView();

# if (result.hasErrors()) {

# modelAndView = new ModelAndView(register, command, user);

# } else {

# try {

# registrationService.registerUser(user);

# modelAndView = new ModelAndView(register, command, user);

# modelAndView.addObject("successMessage",

# environment.getProperty("RegistrationController.SUCCESSFUL\_REGISTRATION"));

# } catch (UserIdAlreadyPresentException e) {

# logger = LoggerFactory.getLogger(this.getClass());

# if (e.getMessage().contains("RegistrationService")) {

# modelAndView = new ModelAndView(register);

# modelAndView.addObject(command, user);

# modelAndView.addObject("message", environment.getProperty(e.getMessage()));

# }

# logger.error(e.getMessage(), e);

# }

# }

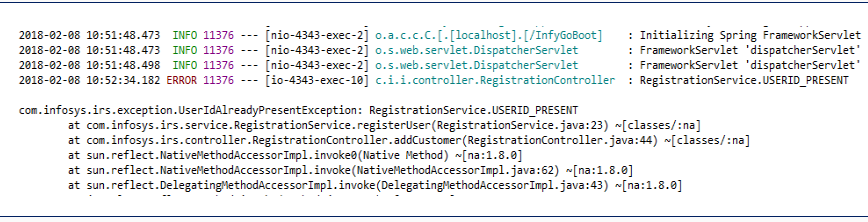
# return modelAndView;

# }

# }

# 

# Step 3: Run your application. Browse to the 'register.jsp' page and provide values such that the given user id is already existing in the database. It will render to the 'register.jsp' page with an error message. Now, go to the 'console' window of STS and observe that the exception message logged over there.



Step 4: Thus, we have implemented logging in our application. However, the code will look neat and clean if the logging is treated as a separate aspect and is implemented using AOP. Create a new package named 'com.infosys.irs.utility' and create a 'LoggingAspect.java' file in that package as given below. This file uses the 'AOP' feature to log the exception messages thrown by the service and repository layer of your application.

package com.infosys.irs.utility;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import com.infosys.irs.exception.InfyGoBootException;

@Component

@Aspect

public class LoggingAspect {

@AfterThrowing(pointcut = "execution(\* com.infosys.irs.repository.\*.\*(..))", throwing = "exception")

public void logExceptionFromRepository(InfyGoBootException exception) throws InfyGoBootException {

log(exception);

}

@AfterThrowing(pointcut = "execution(\* com.infosys.irs.service.\*.\*(..))", throwing = "exception")

public void logExceptionFromService(InfyGoBootException exception) throws InfyGoBootException {

if (exception.getMessage().contains("Service")) {

log(exception);

}

}

private void log(Exception exception) {

Logger logger = LoggerFactory.getLogger(this.getClass());

logger.error(exception.getMessage(), exception);

}

}

Step 5: Add the dependency in the 'pom.xml file' as shown below:

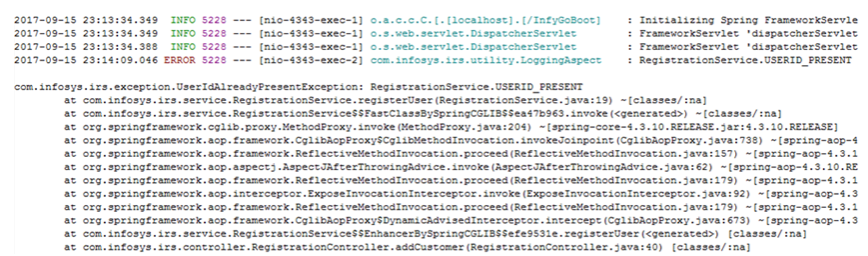
<dependency>

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

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

</dependency>

Step 6: Run your application. Browse to the 'register.jsp' page and provide values such that the given user id is already existing in the database. It will render to the 'register.jsp' page with an error message. Now, go to the 'console' window of STS and observe that the exception message is logged over there.



Thus, you have implemented logging in the IRS application.

**Things to observe**

Have you realized that you have not done any of the below activities for logging which you typically do in any Spring application?

* Adding dependent jars for logging
* Configuring logging through Java configuration or XML configuration

Still, you are able to log your messages. The reason is Spring Boot's default support for logging. The spring-boot-starter dependency includes spring-boot-starter-logging dependency, which configures logging via Logback to log to the console at INFO level.

Spring Boot uses Commons Logging API with default configurations for Java Util Logging, Log4j 2, and Logback implementation. Among these implementations, Logback configuration will enabled by default.

You, as a developer, have just created an object for Logger and raise a request to log with your own message in LoggingAspect.java as shown below.

private void log(Exception exception) {

Logger logger = LoggerFactory.getLogger(this.getClass());

logger.error(exception.getMessage(), exception);

}

@AfterThrowing(pointcut = "execution(\* com.infosys.irs.repository.\*.\*(..))", throwing = "exception")

public void logExceptionFromDAO(Exception exception) throws Exception {

log(exception);

}

@AfterThrowing(pointcut = "execution(\* com.infosys.irs.service.\*.\*(..))", throwing = "exception")

public void logExceptionFromService(Exception exception) throws Exception {

if (exception.getMessage().contains("Service")) {

log(exception);

}

}

The default log output contains the following information.

1. Date and Time



1. Log level



1. Process id



1. Separator



1. Thread name



1. Logger name



1. Log message



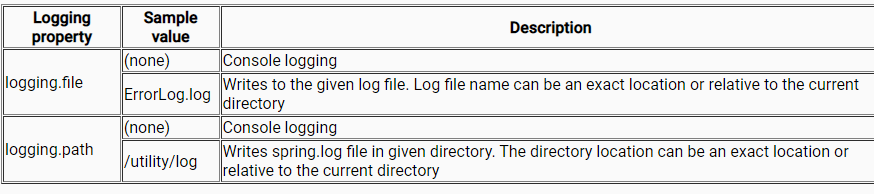
But, how to change this default configuration if you want to,

* log the message in a file rather than console
* log the message in your own pattern
* log the messages of a specific level
* use Log4j instead of Logback

Customizing logging configuration

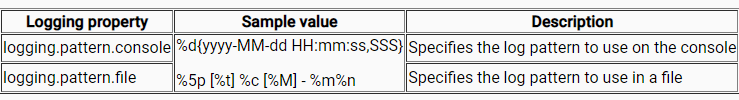
# **Log into file**

By default Spring Boot logs the message on console. To log into file, you have to include either logging.file or logging.path property in your application.properties file.



# **Custom log pattern**

Include logging.pattern.\* property in application.properties file to write the log message in your own format.



# **Custom log level**

By default logger is configured at INFO level. You can change this by configuring logging.level.\* property in application.properties file as shown below.

logging.level.root=WARN

logging.level.com.infosys.ars=ERROR

# **Log4j instead of Logback**

Since Spring Boot chooses Logback implementation by default, you need to exclude it and then include log4j 2 instead in your pom.xml.

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

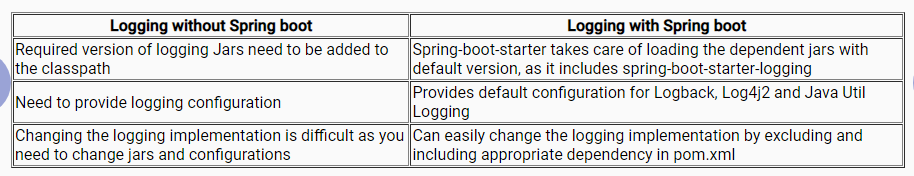
</dependency>

<dependency>

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

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>



# Exercise: Log the error messages using Logback

## **ProblemStatement:**

Modify exercise 3 the MovieMax Spring boot application to log the error messages using Logback.

* Use AOP to log the details of exceptions thrown from controller, service, and repository classes.
* Logging should be done in an external file named MovieMax\_ErrorLog.log

Verification:

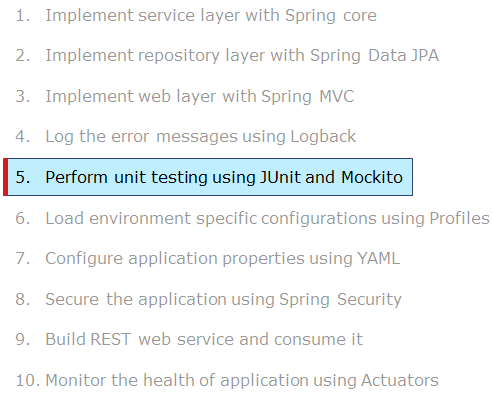
Verify the messages in MovieMax\_ErrorLog.log in the following scenarios.

* Entering invalid city name
* Selecting a show time where seats are not available

Perform unit testing using JUnit and Mockito

By now, you have completed the implementation of the user registration use case of the IRS application successfully. But, have you noticed that unit testing is not done yet for your code? It will be an overhead to fix the bugs at a later point in time as it incurs more time and effort.

Therefore, it is time to unit test your code. Though many tools are in existence to unit test an application, you are going to use, JUnit along with Mockito to unit test your application code.



# Demo: Perform unit testing with Junit and Mockito

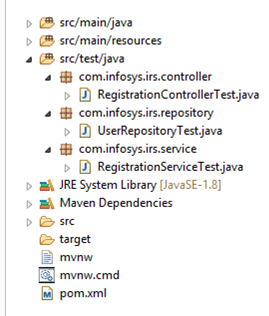
Highlights:

* How to perform unit testing in Spring Boot application
* How to write unit test cases with mock objects in Spring Boot application

Demo steps:

Step 1: Open STS and import the IRS project named 'InfyGoBoot\_Logback' created in Demo 5. Rename the project as 'InfyGoBoot\_UnitTesting'.

Step 2: Create the Java files in the 'src/test/java' package as shown below.



Step 3: Implement 'RegistrationControllerTest.java' as below.

package com.infosys.irs.controller;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

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

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.test.context.junit4.SpringRunner;

import org.springframework.test.web.servlet.MockMvc;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

import com.infosys.irs.service.RegistrationService;

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

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

import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

@RunWith(SpringRunner.class)

@WebMvcTest(controllers = RegistrationController.class)

public class RegistrationControllerTest {

@MockBean

private UserRepository userRepository;

@MockBean

private RegistrationService registrationService;

@InjectMocks

private RegistrationController registrationController;

@Autowired

private MockMvc mockMvc;

private User user1;

private User user2;

private String command = "command";

private String dummy1 = "dummy1";

private String dummysuccess2 = "dummysuccess2";

public void user1() {

user1 = new User();

user1.setUserId(dummy1);

user1.setPassword(dummy1);

user1.setName(dummy1);

user1.setCity(dummy1);

user1.setEmail(dummy1);

user1.setPhone(dummy1);

}

public void user2() {

user2 = new User();

user2.setUserId(dummysuccess2);

user2.setPassword(dummysuccess2);

user2.setName(dummysuccess2);

user2.setCity("chennai");

user2.setEmail("dummysuccess2@emaill.com");

user2.setPhone("1234567890");

}

@Before

public void setUp() {

user1();

user2();

}

@Test

public void testRegistrationGet() throws Exception {

this.mockMvc.perform(get("/register")).andExpect(status().isOk()).andExpect(model().attributeExists(command))

.andExpect(view().name("register")).andDo(print());

}

@Test

public void testSubmitCustomerFailsValidation() throws Exception {

this.mockMvc

.perform(post("/registerUser").param("userId", user1.getUserId()).param("password", user1.getPassword())

.param("name", user1.getName()).param("city", user1.getCity()).param("email", user1.getEmail())

.param("phone", user1.getPhone()))

.andExpect(model().attributeHasFieldErrors(command)).andExpect(view().name("register"))

.andExpect(status().isOk()).andDo(print());

}

@Test

public void testSubmitCustomerPassesValidation() throws Exception {

this.mockMvc.perform(post("/registerUser").param("userId", user2.getUserId())

.param("password", user2.getPassword()).param("name", user2.getName()).param("city", user2.getCity())

.param("email", user2.getEmail()).param("phone", user2.getPhone()))

.andExpect(model().attributeHasNoErrors(command));

}

}

Step 4: Implement 'UserRepositoryTest.java' as below.

package com.infosys.irs.repository;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import org.springframework.test.context.junit4.SpringRunner;

import com.infosys.irs.entity.UserEntity;

import com.infosys.irs.model.User;

@RunWith(SpringRunner.class)

@DataJpaTest

@AutoConfigureTestDatabase(replace = Replace.NONE)

public class UserRepositoryTest {

@Mock

private UserRepository userRepository;

private User user1;

private User user2;

private String dummy1 = "dummy1";

private String dummy2 = "dummy2";

private String userId = "userId";

public void user1() {

user1 = new User();

user1.setUserId(dummy1);

user1.setPassword(dummy1);

user1.setName(dummy1);

user1.setCity(dummy1);

user1.setEmail(dummy1);

user1.setPhone(dummy1);

}

public void user2() {

user2 = new User();

user2.setUserId(dummy2);

user2.setPassword(dummy2);

user2.setCity(dummy2);

user2.setCity(dummy2);

user2.setEmail(dummy2);

user2.setPhone(dummy2);

}

@Before

public void setUp() {

user1();

user2();

}

@Test

public void saveAndFlushTest() {

UserEntity userEntity1 = new UserEntity();

userEntity1.setCity(user1.getCity());

userEntity1.setEmail(user1.getEmail());

userEntity1.setName(user1.getName());

userEntity1.setPassword(user1.getPassword());

userEntity1.setPhone(user1.getPhone());

userEntity1.setUserId(user1.getUserId());

UserEntity userEntity2 = new UserEntity();

userEntity2.setCity(user2.getCity());

userEntity2.setEmail(user2.getEmail());

userEntity2.setName(user2.getName());

userEntity2.setPassword(user2.getPassword());

userEntity2.setPhone(user2.getPhone());

userEntity2.setUserId(user2.getUserId());

Mockito.when(userRepository.saveAndFlush(userEntity1)).thenReturn(userEntity1);

UserEntity user = userRepository.saveAndFlush(userEntity1);

assertThat(user).hasFieldOrPropertyWithValue(userId, "dummy1");

Mockito.when(userRepository.saveAndFlush(userEntity2)).thenReturn(new UserEntity());

user = userRepository.saveAndFlush(userEntity2);

assertThat(user).hasFieldOrPropertyWithValue(userId, null);

}

@Test

public void findOneTest() {

UserEntity userEntity2 = new UserEntity();

userEntity2.setCity(user2.getCity());

userEntity2.setEmail(user2.getEmail());

userEntity2.setName(user2.getName());

userEntity2.setPassword(user2.getPassword());

userEntity2.setPhone(user2.getPhone());

userEntity2.setUserId(user2.getUserId());

Mockito.when(userRepository.getOne(user2.getUserId())).thenReturn(userEntity2);

UserEntity user = userRepository.getOne(user2.getUserId());

assertThat(user).hasFieldOrPropertyWithValue(userId, "dummy2");

}

}

Step 5: Implement 'RegistrationServiceTest.java' as below.

package com.infosys.irs.service;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Rule;

import org.junit.Test;

import org.junit.rules.ExpectedException;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.mockito.MockitoAnnotations;

import org.springframework.test.context.ContextConfiguration;

import com.infosys.irs.entity.UserEntity;

import com.infosys.irs.exception.UserIdAlreadyPresentException;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

@ContextConfiguration

public class RegistrationServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private RegistrationService registrationService;

@Rule

public ExpectedException expectedException = ExpectedException.none();

User user = new User();

UserEntity userEntity = new UserEntity();

public void inititalizeUserAndUserEntity() {

user.setCity("Chennai");

user.setEmail("anithamalar@infygo.com");

user.setName("anithamalar");

user.setPassword("anithamalar\_svs");

user.setPhone("9840567854");

user.setUserId("C1021");

userEntity.setCity(user.getCity());

userEntity.setEmail(user.getEmail());

userEntity.setName(user.getName());

userEntity.setPassword(user.getPassword());

userEntity.setPhone(user.getPhone());

userEntity.setUserId(user.getUserId());

}

@Before

public void initialWork() {

MockitoAnnotations.initMocks(this);

inititalizeUserAndUserEntity();

}

@Test

public void testRegisterUserPositive() {

Mockito.when(userRepository.getOne(user.getName())).thenReturn(null);

Mockito.when(userRepository.saveAndFlush(userEntity)).thenReturn(userEntity);

try {

registrationService.registerUser(user);

} catch (Exception e) {

Assert.fail("Exception " + e);

}

}

@Test

public void testRegisterUserNegative() throws UserIdAlreadyPresentException {

expectedException.expect(UserIdAlreadyPresentException.class);

expectedException.expectMessage("RegistrationService.USERID\_PRESENT");

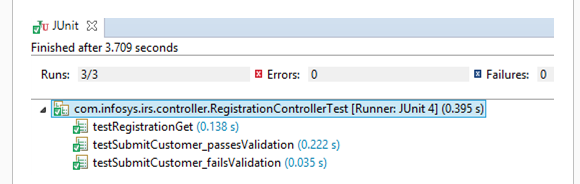
Mockito.when(userRepository.existsById(Mockito.anyString())).thenReturn(true);

registrationService.registerUser(user);

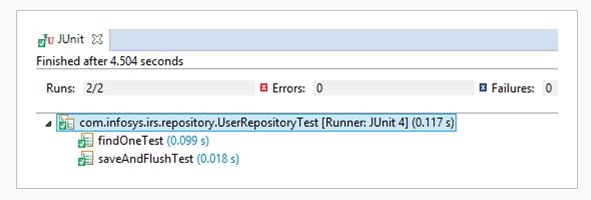
}

}

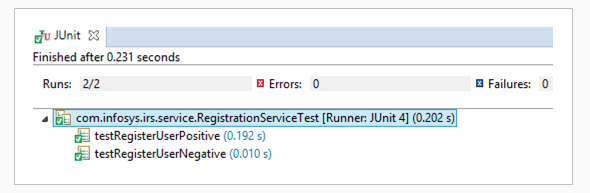
Step 6: Run the 'RegistrationControllerTest.java' by right-clicking on it and choosing Run As -> JUnit Test and observe the test result as below.



Step 7: Run the 'UserRepositoryTest.java' by right-clicking on it and choosing Run As -> JUnit Test and observe the test result as below.



Step 8: Run the 'RegistrationServiceTest.java' by right-clicking on it and choosing Run As -> JUnit Test and observe the test result as below.

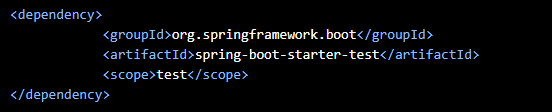


Thus you have performed Unit testing on RegistrationController.java, UserRepository.java, and RegistrationService.java for the User Registration use case of the IRS application.

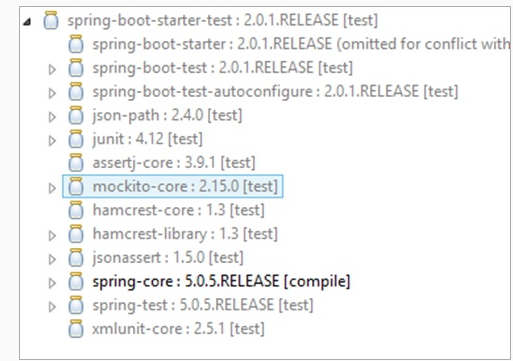
Let us discuss a few important things that you need to observe in this demo.

Have you noticed that no libraries related to Junit and Mockito added to the class path?

It is not required because of the spring-boot-starter-test dependency added in your POM.



The spring-boot-starter-test downloads the following bundle of libraries related to testing. Therefore, you do not need to add the libraries manually.



These are some common libraries that we usually find valuable when writing test cases. We can add other test dependencies if it does not suit for the needs.

Apart from this, Spring Boot provides a set of annotations to help when testing your applications. Let us see about few of those annotations in detail...

**Spring Boot testing annotations**

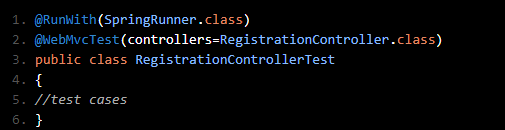
@RunWith (SpringRunner.class) annotaion:

This is a class-level annotation that informs JUnit to run using Spring's support for the test. SpringRunner provides the functionality to launch a Spring TestContext Framework. Every test class uses this annotation to utilize Spring's support for unit testing.

# Testing controllers

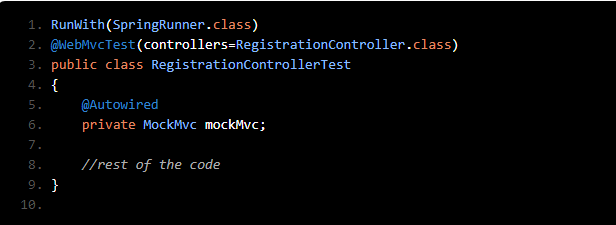
@WebMvcTest (controllers=RegistrationController.class) annotaion:

This is a class-level annotation that helps to implement unit testing in the Spring MVC application, a test focused on Spring MVC components alone. In RegistrationControllerTest, you want to launch only RegistrationController. So, other mappings and controllers do not get executed when this unit test is executed.



MockMvc

Spring Boot provides MockMvc to test the web layer of an application. MockMvc handles the incoming HTTP request and hands it off to the controller without the server it is started. So, it is preferable to test the RegistrationController with MockMvc. But, make sure MockMvc is autowired to the class that performs the test.



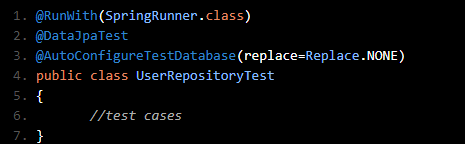
# **Testing repositories**

@DataJpaTest:

This class level annotation is helpful when the JPA repository classes are unit tested. This annotation configured with an in-memory embedded database scans for the classes annotated with @Entity and aligns Spring Data JPA repositories. In simple words, this annotation establishes an environment for JPA testing. In addition, this annotation is transactional in nature, and roll back happens after the unit test case finishes its execution.

@AutoConfigureTestDatabase(replace = NONE):

If you want your JPA repository unit test cases to effort with an in-memory embedded database, need to insert the same to your test classpath. In the absence of an in-memory database, this annotation is helpful to a great extent as it instructs that no replacement is needed for the actual data source.

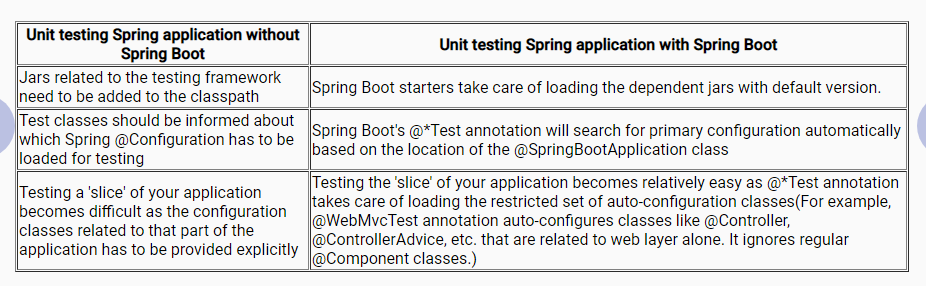


# **Testing services**

@SpringBootTest annotation is applied on RegistrationServiceTest class for loading the Spring configuration. This annotation looks for a focal configuration class (@SpringBootApplication), and use to start a Spring application context.

Apart from the configurations mentioned in the main configuration class, if you need additional beans or want to alter the existing configuration, you can use the @TestConfiguration class.

Additional Reference: Refer to Spring documentation to learn more about testing in the Spring Boot application.



# Exercise: Perform unit testing using JUnit and Mockito

## **ProblemStatement:**

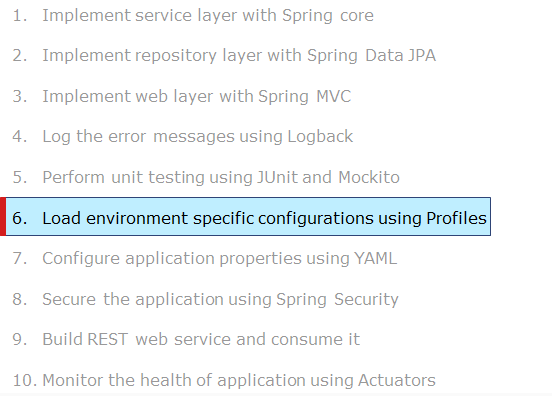
Modify the Spring boot application MovieMax of exercise 4 to perform unit testing on the controller, service, and repository classes using Junit and Mockito.

Verification:

* Verify that all the test cases are being passed for both positive and negative test conditions.
* Use any open source code coverage tool like Cobertura and confirm that at least 80% of your code should be covered under test cases.

# Load environment specific configuration using Profiles

Let us now learn about another important feature of Spring Boot called Spring Profiles.



Consider the below configuration requirements to be used under various environments for IRS application execution.



How can it be achieved?

For every execution, the database configuration and logging configuration defined in the application.properties file need to be changed according to the environment in which it is running. But, it is a time-consuming job and it is difficult to keep track of changes as well.

Spring Boot provides a feature called Profiles, which can load the application properties according to the environment in which it is running.

Let us use Spring Boot profiles in IRS application to connect with MySQL database in the development environment and embedded HSQL database in a testing environment.

# Demo: Load environment specific configuration using Profiles

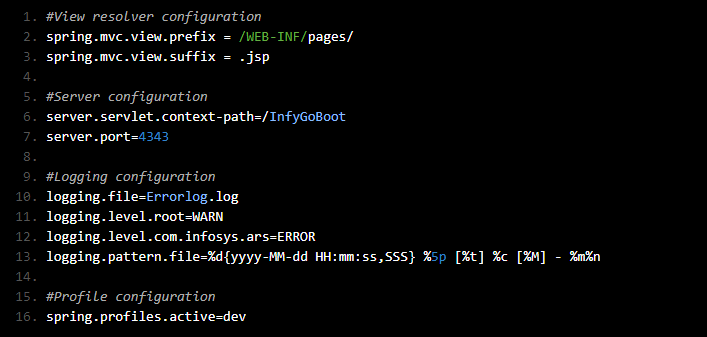
# Highlights:

* How to use Spring Boot Profiles to load environment-specific properties?
* How to use MySQL for development and HSQL for testing using Profiles?

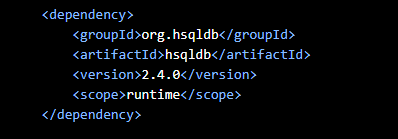
Demo steps:

Step 1: Open STS and import the IRS project named 'InfyGoBoot\_UnitTesting' created in Demo 6. Rename the project as 'InfyGoBoot\_Profiles'.

Step 2: Add the properties in the 'src/main/resources/application.properties' file as shown below.

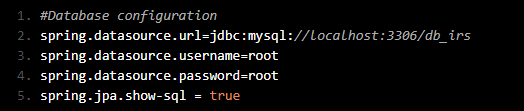


Step 3: Add hsql database dependency in pom.xml

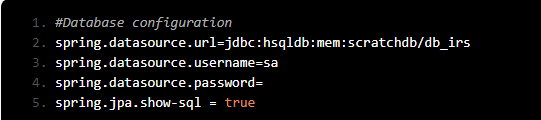


Step 4: Create the properties files under the 'src/main/resources' folder as shown below. (Note: Username and password should be given as per your database account details)

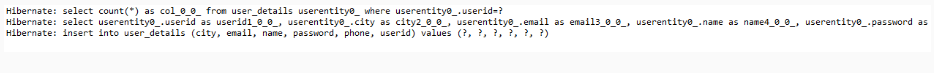
application-dev.properties



Application-test.properties



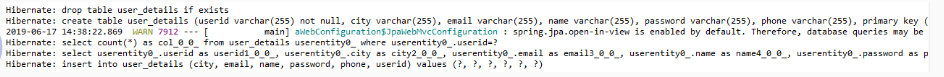
Step 5: Run the application and open the signup page, provide the new user details. Observe that it performs database operations on the MySQL database also displays the database commands on the console as below.



Step 6: Update 'spring.profiles.active' as 'test' in 'application.properties' file as shown below.



Step 7: Run your application. Observe that it performs database operations on the HSQL database on the console as shown below.



Thus you have loaded environment specific configuration in IRS application using Profiles.

**Things to observe**

Creating environment specific properties

To run the Spring Boot application in various environments with different configurations, you need to provide application.properties file for each environment individually using application-<environment>.properties naming convention. For example,

application-dev.properties - To use in development environment

application-test.properties - To use in testing environment

The <environment> string can be anything. Use a meaningful name for better understanding.

Adding active profile

By default, Spring Boot will use application.properties file to load the property values. You need to use spring.profiles.active property in application.properties file to load environment specific properties. For ex, if you wish to load the properties specified under application-dev.properties file for development environment, then your application.properties file should have the below property.



When an active profile is added into default properties file, which is application.properties, the values configured in active profile will override the default properties.

Programmatically setting active profile

Instead of setting active profile statically in application.properties file, you can set it programmatically as well in two ways as shown below

1. Calling SpringApplication.setAdditionalProfiles(String...profiles)



2. Using ConfigurableEnvironment interface



@Profile

You can make your application to use or ignore any @Component or @Configuration or bean objects by applying @Profile annotation. For example the below component will be used only if the active profile is set to development environment. Other environments will ignore this component object.



Additional Reference: Read more on Spring Boot Profiles in Spring official site.

# Exercise: Load environment specific configurations using Profiles

## **Problem Statement:**

Modify the Spring boot application MovieMax of exercise 5 to configure database and logging details as per the environment in which the application is being executed.

Development environment:

* Use MySQL database
* Log the messages on console with INFO level

Production environment:

* Use MongoDB database
* Log the error messages in an external file

Verification:

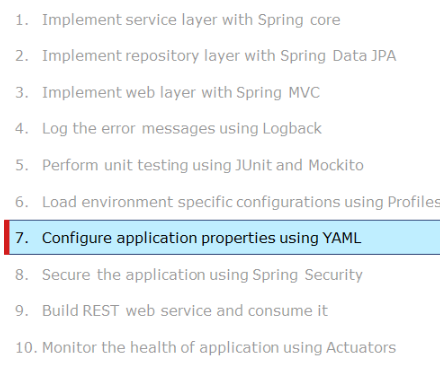
* MovieMax is running in the development environment:
  + Run the application to book a movie. Corresponding tables should be updated in the MySQL database. Log messages should be displayed on the console.
* MovieMax is running in a production environment:
  + Run the application to book a movie. Corresponding tables should be updated in the MongoDB database. Exception messages are available in an external file.

# Configure application properties using YAML

Instead of creating and maintaining a separate application.properties file for each environment, you can use YAML for configuration as it allows you to specify all the environment-specific properties in a single file.

YAML is a superset of JSON and is more readable because of its convenient syntax. It provides values in a hierarchical format.

Let us see how to use YAML instead of the application.properties file.



# Demo: Configure application properties using YAML

Highlights:

* How to use the YAML file in the Spring Boot application?
* How to configure properties in YAML?

Demo steps:

Step 1: Open STS and import the IRS project named 'InfyGoBoot\_Profiles' created in Demo 7. Rename the project as 'InfyGoBoot\_YAML'.

Step 2: Remove all the existing 'application.properties' file and create the 'application.yml' file under the 'src/main/resources' folder as shown below.

spring:

profiles:

active: dev

mvc:

view:

prefix: /WEB-INF/pages/

suffix: .jsp

server:

servlet.context-path: /InfyGoBoot

port: 4343

logging:

file: Errorlog.log

level:

root: warn

com.infoys.ars: error

pattern:

file: "%d{yyyy-MM-dd HH:mm:ss,SSS} %5p [%t] %c [%M] - %m%n"

---

spring:

profiles: dev

datasource:

url: jdbc:mysql://localhost:3306/db\_irs

datasource:

username: root

password: root

jpa:

show-sql: true

---

spring:

profiles: test

datasource:

url: jdbc:hsqldb:mem:scratchdb/db\_irs

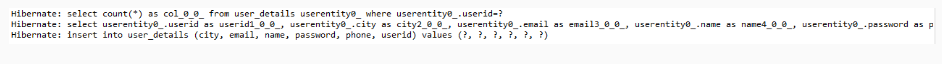
username: sa

password:

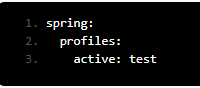
jpa:

show-sql: true

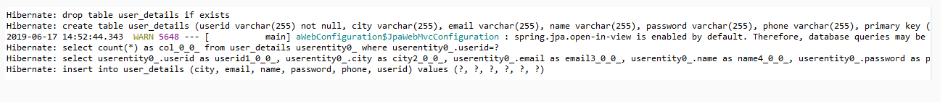
Step 3: Run your application. Observe that it performs database operations on MySQL database on the console as shown below.



Step 4: Update 'spring.profiles.active' as 'test' in 'application.yml' file as shown below.



Step 5: Run your application.Observe that it performs database operations on HSQL database on the console as shown below.

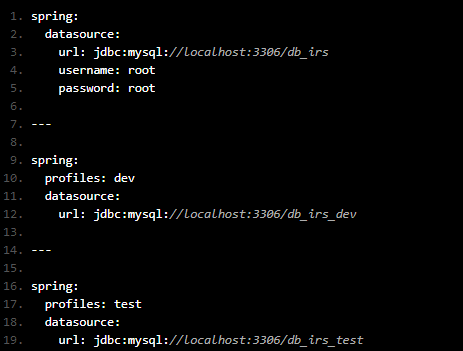


Thus you have loaded all the required properties using the YAML document.

**Properties or YAML?**

To use YAML instead of Properties, you need to create application.yml file under src/main/resources folder of your application. The YAML file will be parsed by Snake YAML library provided via spring-boot-starter. So there is nothing you need to do to enable YAML support in Spring Boot.

The single YAML file used in previous page is actually a sequence of documents separated by three dashes "---", where each document represents the properties for a specific profile. For example:



In this example, the first document is not attached to any profile, the second document is attached to development profile and third one is attached to test profile.

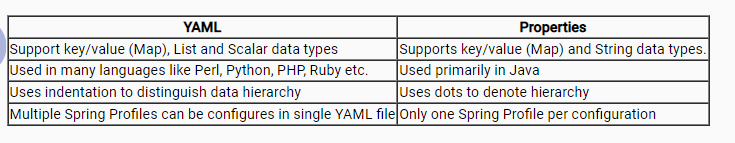
In this case, the datasource username and password will be same for all the profiles whereas the url value will be "jdbc:mysql://localhost:3306/db\_irs\_dev " if development profile is active and it will be "jdbc:mysql://localhost:3306/db\_irs\_test" if the test profile is active. If neither development nor test profile is enabled, then the url value will be "jdbc:mysql://localhost:3306/db\_irs".

Additional Reference: Read more on using YAML instead of Properties (Refer Spring documentation).

**Things to observe**

Does configuring multiple spring profiles in a single YAML file is the only advantage of YAML?

To know better, let us see a comparison between YAML and Properties.



In this example, the first document is not attached to any profile, the second document is attached to development profile and third one is attached to test profile.

# Exercise: Configure application properties using YAML

## **Problem Statement:**

Modify the Spring boot application MovieMax of exercise 6 to use YAML for Spring configurations.

Configure the below details in application.yml:

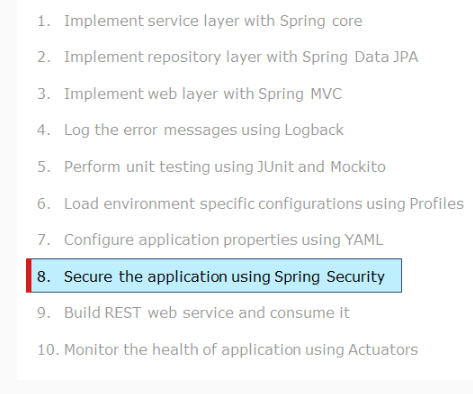
* Database credentials
* Server details
* Logging details
* View resolver
* Spring Profile details

Verification:

MovieMax application should not have application.properties file in it. Instead it should contain application.yml. Execute your application and verify that business functionalities are executed as expected, in the presence of application.yml.

# Secure the application using Spring Security

Until now, you have learned the Spring Boot features just by implementing a user registration use case. Now, implement the rest of the use cases of IRS application using Spring Boot and secure the application with proper authentication using Spring Security.



# Demo: Secure the application using Spring Security

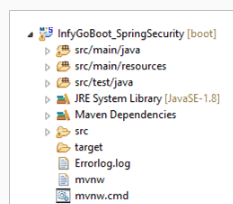
Highlights:

* How to create Spring Boot project structure for web application using Spring Initializr
* How to secure application using Spring Security

Demo steps:

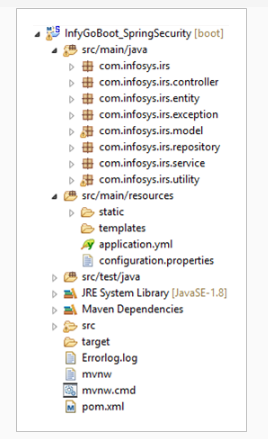
Step 1: Execute the existing table script for the tables ('flight\_Details', 'passenger\_details', 'creditcard\_details', 'ticket\_details') from the non-Spring Boot IRS application Demo:1 (Refer InfyGoMaven\InfyGoMaven\src\com\infosys\irs\resources folder for table script).

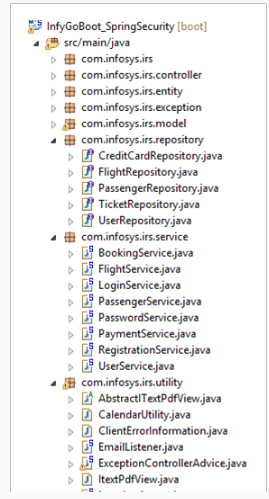
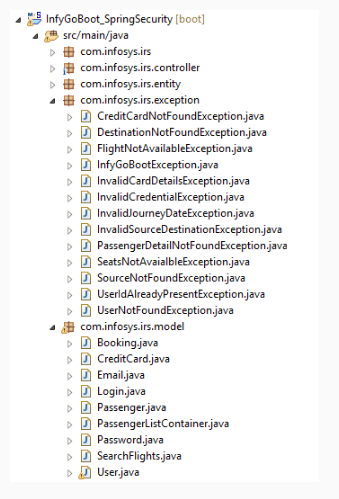
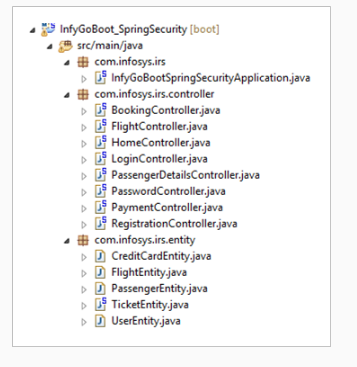
Step 2: Open STS and import the IRS project named 'InfyGoBoot\_YAML' created in Demo 8. Rename the project as 'InfyGoBoot\_SpringSecurity'. You can see the project structure below.



Step 3: Include all the '.java' files and '.jsp' files from the existing non Spring Boot IRS application. Place them in appropriate folders to see the package structure as shown below.

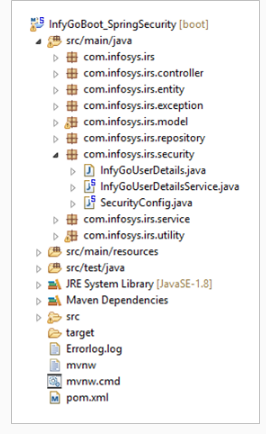
Note: Replace the fineOne() method with getOne() in all service classes.







Step 4: Include a package called 'com.infosys.irs.security' to see the package structure as shown below.



Step 5: Implement 'InfyGoUserDetails.java' as below.

package com.infosys.irs.security;

import java.util.Collection;

import java.util.List;

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.AuthorityUtils;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.util.StringUtils;

import com.infosys.irs.model.User;

@SuppressWarnings("serial")

public class InfyGoUserDetails implements UserDetails {

private List<String> userRoles;

private transient User user;

public InfyGoUserDetails(User user, List<String> userRoles) {

this.user = user;

this.userRoles = userRoles;

}

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

String roles = StringUtils.collectionToCommaDelimitedString(userRoles);

return AuthorityUtils.commaSeparatedStringToAuthorityList(roles);

}

@Override

public String getUsername() {

return user.getUserId();

}

@Override

public boolean isAccountNonExpired() {

return true;

}

@Override

public boolean isAccountNonLocked() {

return true;

}

@Override

public boolean isCredentialsNonExpired() {

return true;

}

@Override

public boolean isEnabled() {

return true;

}

@Override

public String getPassword() {

return user.getPassword();

}

}

Step 6: Implement 'InfyGoUserDetailsService.java' as below.

package com.infosys.irs.security;

import java.util.Arrays;

import java.util.List;

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

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Service;

import com.infosys.irs.entity.UserEntity;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

@Service

public class InfyGoUserDetailsService implements UserDetailsService {

@Autowired

private UserRepository userRepository;

@Autowired

private PasswordEncoder passwordEncoder;

@Override

public UserDetails loadUserByUsername(String username) {

UserEntity userEntity = userRepository.findByName(username);

if (userEntity == null) {

throw new UsernameNotFoundException(username);

}

User user = new User();

user.setName(userEntity.getName());

user.setPassword(passwordEncoder.encode(userEntity.getPassword()));

user.setUserId(userEntity.getUserId());

List<String> userRoles=Arrays.asList("USER");

return new InfyGoUserDetails(user, userRoles);

}

}

Step 7: Implement 'SecurityConfig.java' as below.

package com.infosys.irs.security;

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

import org.springframework.boot.autoconfigure.security.SecurityProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.annotation.Order;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration

@Order(SecurityProperties.BASIC\_AUTH\_ORDER)

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private InfyGoUserDetailsService userDetailsService;

@Autowired

public void configAuthentication(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService);

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/","/login","/register","/imgs/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.formLogin()

.loginPage("/login")

.defaultSuccessUrl("/flights",true)

.permitAll()

.and()

.logout()

.permitAll();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

}

Step 8: Update the below method in FlightService.java

public void updateFlight(String flightId, String noOfSeats) throws InfyGoBootException {

FlightEntity flight = flightsRepository.getOne(flightId);

if (flight == null) {

throw new InfyGoBootException("No flight for the given details");

} else {

int count = flight.getSeatCount() - Integer.valueOf(noOfSeats);

flight.setSeatCount(count);

flightsRepository.saveAndFlush(flight);

}

}

Step 8: Add the dependency on 'pom.xml' as below.

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.itextpdf</groupId>

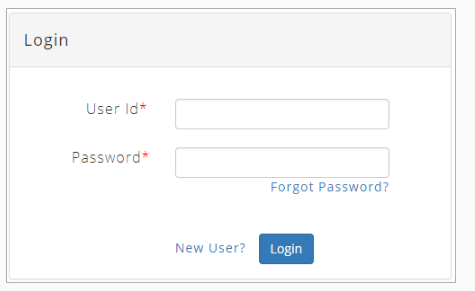
<artifactId>itextpdf</artifactId>

<version>5.1.0</version>

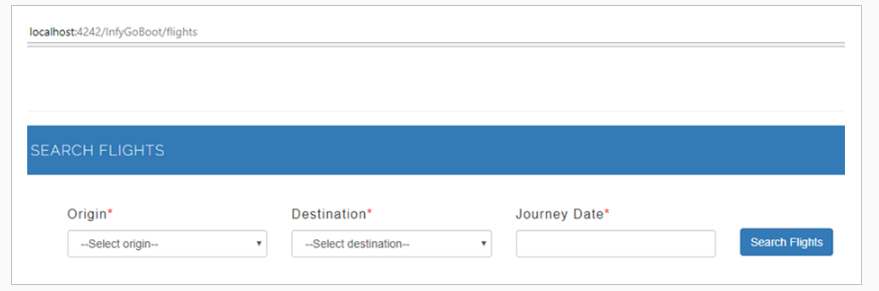
</dependency>

Step 9: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App.

Step 10: Type the url 'http://localhost:4242/InfyGoBoot/flights' in the address bar of your browser. You can see the request redirected to the url 'http://localhost:4242/InfyGoBoot/login' and rendering the 'login.jsp' page as shown below.



Step 11: After giving the right credentials, 'searchFlights.jsp' page will be rendered as shown below.



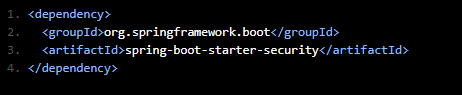
Thus, you have secured the application using Spring Security with MySQL database.

Let us discuss few important things that you need to observe in this demo.

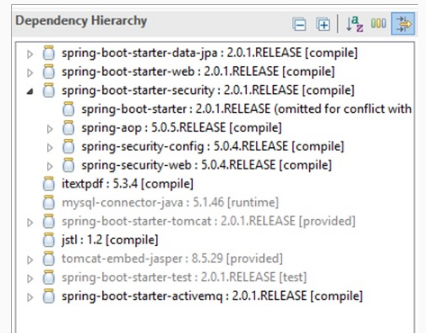
**Things to observe**

Have you noticed that no libraries related to Spring Security are added to the classpath?

It is not required because of the spring-boot-starter-security dependency added in your POM.



Due to the above dependency, relevant jar files as shown below, will be downloaded into the project's class path automatically.



With this dependency, Spring Boot secures all HTTP endpoints with basic authentication automatically. That means, all the pages of the IRS application is prevented from unauthorized access. When the end-user tries to access any of the pages of IRS, it renders a default login page and prompts the user to enter a username and password. On entering valid credentials as explained in the demo, the user can proceed further.

It happens because of the below auto-configuration provided by Spring Boot.

* AuthenticationManager instance with single-user and in-memory store
* Secured HTTP endpoints except static resource locations like /css/\*\*, /images/\*\* and etc.
* Caching and CSRF features
* ApplicationEventPublisher instance to publish the security events

But, how to override these default settings?

For example,

* How to use IRS's own login page instead of the default one, to authenticate the user via Spring Security?
* How to prevent unauthorized access to selected pages?

The next page gives the solution...

To use your own custom page for authentication and to restrict access to the selected pages, you have to provide your own security configuration as shown below.

@Configuration

@Order(SecurityProperties.BASIC\_AUTH\_ORDER)

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

public void configAuthentication(AuthenticationManagerBuilder auth) throws Exception {

auth

.inMemoryAuthentication()

.withUser("C1001").password("Jack1234").roles("USER");

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/","/login","/register","/imgs/\*\*","/FlightAPI/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.formLogin()

.loginPage("/login")

.permitAll()

.and()

.logout()

.permitAll();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

}

The configAuthentication() method uses an in-memory store with a single user-populated with username, password, and role.

The configure() method defines which URL paths require authentication and which do not require. The code says that access to "/","/register", and "/login" paths are permitted to everyone and the rest of the paths are secured with authentication.

When the user tries to access any of the secured pages, let us say "/flights", the user will be redirected to "/login" page as specified by loginPage(). On entering valid credentials as configured in the in-memory store, it will redirect the user to the previously requested page "/flights".

Thus you have secured your application by preventing access to restricted pages. But, it is configured for only one user with an in-memory store which is not acceptable in a real scenario. So, let us see how to provide secured access to all the registered users.

Additional Reference: Refer to the documentation on Spring official site.

UserDetailsService is another option to authenticate the user. Here, you first have to create a class that implements UserDetails. UserDetails is a standard interface provided by Spring framework to represent core user information.

public class InfyGoUserDetails implements UserDetails {

private List<String> userRoles;

private User user;

public InfyGoUserDetails(User user, List<String> userRoles) {

this.user = user;

this.userRoles = userRoles;

}

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

String roles = StringUtils.collectionToCommaDelimitedString(userRoles);

return AuthorityUtils.commaSeparatedStringToAuthorityList(roles);

}

@Override

public String getUsername() {

return user.getUserId();

}

@Override

public String getPassword() {

// TODO Auto-generated method stub

return user.getPassword();

}

}

Then, create a bean class of type UserDetailsService as shown below.

package com.infosys.irs.security;

import java.util.Arrays;

import java.util.List;

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

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Service;

import com.infosys.irs.entity.UserEntity;

import com.infosys.irs.model.User;

import com.infosys.irs.repository.UserRepository;

@Service

public class InfyGoUserDetailsService implements UserDetailsService {

@Autowired

private UserRepository userRepository;

@Autowired

private PasswordEncoder passwordEncoder;

@Override

public UserDetails loadUserByUsername(String username) {

UserEntity userEntity = userRepository.findByName(username);

if (userEntity == null) {

throw new UsernameNotFoundException(username);

}

User user = new User();

user.setName(userEntity.getName());

user.setPassword(passwordEncoder.encode(userEntity.getPassword()));

user.setUserId(userEntity.getUserId());

List<String> userRoles=Arrays.asList("USER");

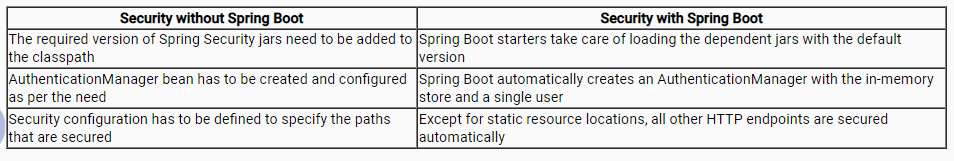
return new InfyGoUserDetails(user, userRoles);

}

}

This class connects with the database through UserRepository and fetches the user details for the username entered on the login page. It returns the retrieved user details as an instance of UserDetails. Note that the user role is hardcoded as "USER" as IRS application does not manage different types of roles.

Finally, inject the UserDetailsService object into AuthenticationManager which will authenticate the user by comparing the password submitted in the login page against the one loaded by UserDetailsService.



# Exercise: Secure the application using Spring Security

## **Problem Statement:**

Modify the Spring boot application MovieMax of exercise 7 to secure the application using Spring Security.

* Create a model object named User
* Create corresponding entity, service, and repository classes named UserEntity, UserService, and UserRepository.
* Implement necessary JSP pages and controller classes for a user to register and login into the MovieMax application. Only registered users can book movies after logging into the application.

Use the table script given below to create User table.

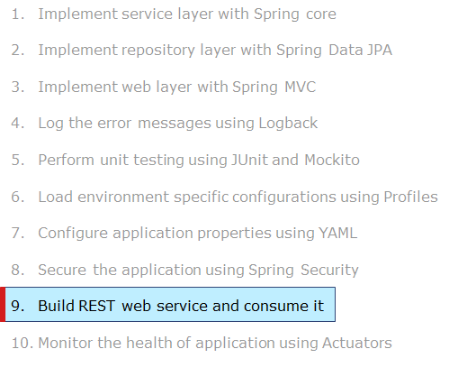


Verification:

* The home page, login page, and register pages should be accessible to everyone.
* The rest of the pages should be accessible only after logging in. They should not be accessible by its url directly typed or pasted in the address bar of the browser.

**Build REST web service and consume it**

InfyGo wants to expose its flight details as a web service. So, let us learn to integrate REST API with Spring Boot.



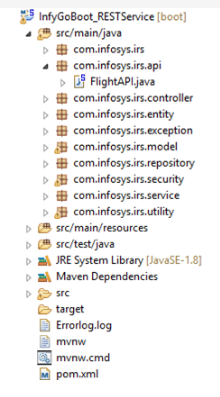
Highlights:

* How to create Spring Boot web application for building and consuming REST service
* How to integrate Spring Boot with Spring REST

Demo Steps:

Step 1: Open STS and import the IRS project named 'InfyGoBoot\_YAML' created in Demo 8. Rename the project as 'InfyGoBoot\_RESTService\_WithoutSecurity'

Step 2: Include a package 'com.infosys.irs.api' and see the package structure as shown below.



Step 3: Implement 'FlightAPI.java' as below.

package com.infosys.irs.api;

import java.util.Calendar;

import java.util.List;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

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

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

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

import com.infosys.irs.model.SearchFlights;

import com.infosys.irs.service.FlightService;

import com.infosys.irs.utility.CalendarUtility;

@RestController

@RequestMapping("FlightAPI")

public class FlightAPI {

@Autowired

private FlightService flightService;

@GetMapping(value = "/{source}/{destination}/{journeyDate}", headers = "Accept=application/json")

public List<SearchFlights> searchFlights(@PathVariable String source, @PathVariable String destination,

@PathVariable String journeyDate) throws Exception {

Calendar jDate = CalendarUtility.getCalendarFromString(journeyDate);

List<SearchFlights> availableFlights = flightService.getFlights(source, destination, jDate);

return availableFlights;

}

@RequestMapping(value = "/source", method = RequestMethod.GET, headers = "Accept=application/json")

public List<String> getSources() throws Exception {

try {

List<String> s1 = flightService.getSources();

return s1;

} catch (Exception e) {

e.printStackTrace();

throw e;

}

}

@RequestMapping(value = "/destination", method = RequestMethod.GET, headers = "Accept=application/json")

public ResponseEntity<List<String>> getDestinations() throws Exception {

List<String> s2 = flightService.getDestinations();

return new ResponseEntity<List<String>>(s2, HttpStatus.OK);

}

}

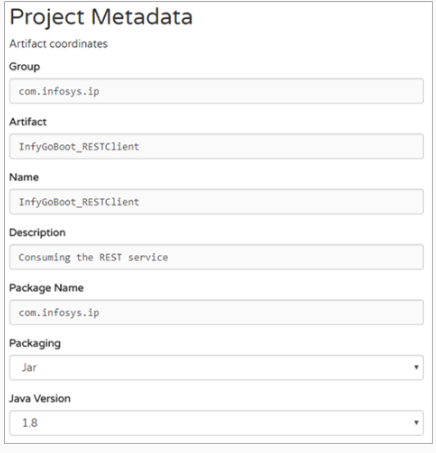
Step 4: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App.

Step 5: Go to 'http://start.spring.io/'.

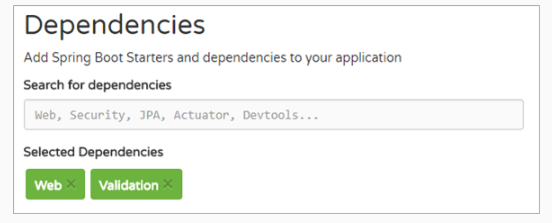
Step 7: Generate a 'Maven' project with Spring Boot.

Step 8: Click on 'Switch to the full version' to provide further details.

Step 9: Fill in the 'Project Metadata' details as shown below.



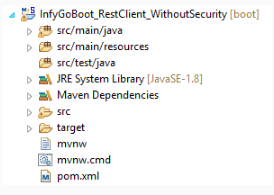
Step 10: Add the dependencies as shown below.



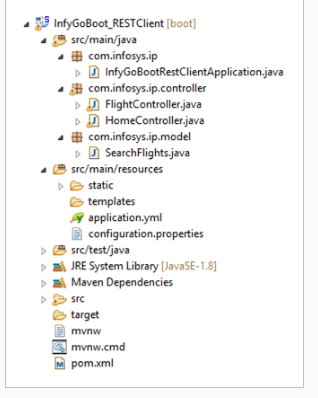
Step 11: Click on 'Generate Project'.

Step 12: Save the zip file in your system at any location and unzip it.

Step 13: Open STS and import the project created above by selecting the File menu and choosing Import -> Existing Maven Project -> Browse to the application and Finish. Rename the project as InfyGoBoot\_RESTClient\_WithoutSecurity. You can see the project structure as below.



Step 14: Create the Java files with the package structure as shown below.



Step 15: Implement 'FlightController.java' as shown below.

package com.infosys.ip.controller;

import java.util.List;

import javax.validation.Valid;

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

import org.springframework.boot.web.client.RestTemplateBuilder;

import org.springframework.core.env.Environment;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.validation.BindingResult;

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

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

import org.springframework.web.client.RestTemplate;

import org.springframework.web.servlet.ModelAndView;

import com.infosys.ip.model.SearchFlights;

@Controller

public class FlightController {

@Autowired

private Environment environment;

private final RestTemplate restTemplate;

public FlightController(RestTemplateBuilder restTemplateBuilder) {

this.restTemplate = restTemplateBuilder.build();

}

@PostMapping(value = "/searchFlights")

public ModelAndView flightSearch(@Valid @ModelAttribute("command") SearchFlights searchFlights,

BindingResult bindingResult, ModelMap model) {

String restUrl = environment.getProperty("RestUrl");

ModelAndView modelAndView = null;

if (bindingResult.hasErrors()) {

new ModelAndView("searchFlights", "command", searchFlights);

}

@SuppressWarnings("unchecked")

List<com.infosys.ip.model.SearchFlights> availableFlights = restTemplate

.getForObject(restUrl + "/FlightAPI/" + searchFlights.getSource() + "/" + searchFlights.getDestination()

+ "/" + searchFlights.getJourneyDate(), List.class);

List<String> sources = restTemplate.getForObject(restUrl + "/FlightAPI/source", List.class);

List<String> destinations = restTemplate.getForObject(restUrl + "/FlightAPI/destination", List.class);

model.addAttribute("availableFlights", availableFlights);

model.addAttribute("size", availableFlights.size());

modelAndView = new ModelAndView("searchFlights", "command", searchFlights);

modelAndView.addObject("sourceList", sources);

modelAndView.addObject("destinationList", destinations);

return modelAndView;

}

}

Step 16: Implement 'HomeController.java' as shown below.

package com.infosys.ip.controller;

import java.util.List;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpSession;

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

import org.springframework.boot.web.client.RestTemplateBuilder;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

import org.springframework.stereotype.Controller;

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

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

import org.springframework.web.bind.support.SessionStatus;

import org.springframework.web.client.RestTemplate;

import org.springframework.web.servlet.ModelAndView;

import com.infosys.ip.model.SearchFlights;

@Controller

@SessionAttributes("contextPath")

@PropertySource("classpath:configuration.properties")

public class HomeController {

@Autowired

private Environment environment;

private final RestTemplate restTemplate;

public HomeController(RestTemplateBuilder restTemplateBuilder) {

this.restTemplate = restTemplateBuilder.build();

}

@GetMapping(value = "/flight")

public ModelAndView getSourceDestinationDetails(HttpSession session, HttpServletRequest request,

SessionStatus status) {

String restUrl = environment.getProperty("RestUrl");

session.setAttribute("contextPath", request.getContextPath());

status.setComplete();

List<String> sources = restTemplate.getForObject(restUrl + "/FlightAPI/source", List.class);

List<String> destinations = restTemplate.getForObject(restUrl + "/FlightAPI/destination", List.class);

ModelAndView modelAndView = new ModelAndView("searchFlights", "command", new SearchFlights());

modelAndView.addObject("sourceList", sources);

modelAndView.addObject("destinationList", destinations);

return modelAndView;

}

}

Step 17: Reuse 'SearchFlights.java' from Demo 11 and place it inside 'com.infosys.ip.model' folder.

Step 18: Add the 'configuration.properties' file under the folder 'src/main/resources' as shown below.

RestUrl=http://localhost:4242/InfyGoBoot

Step 19: Add the 'application.yml' file under the folder 'src/main/resources' as shown below.

spring:

mvc:

view:

prefix: /WEB-INF/pages/

suffix: .jsp

server:

servlet.context-path: /InfyGoBootRestClient

port: 4848

Step 20: Reuse the static resources like 'css', 'js', 'imgs' and 'images' from existing non-Spring Boot IRS application and place them inside 'src/main/resources/static' folder so that Spring Boot can pick up them automatically.

Step 21: Reuse the '.jsp' files 'searchFlights.jsp', 'header2'.jsp, 'header.jsp' and 'footer.jsp' from existing non Spring Boot IRS application and place them inside 'src/main/webapp/WEB-INF/pages' folder.

Step 22: Add the dependency in the 'pom.xml' file as shown below.

<dependency>

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

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

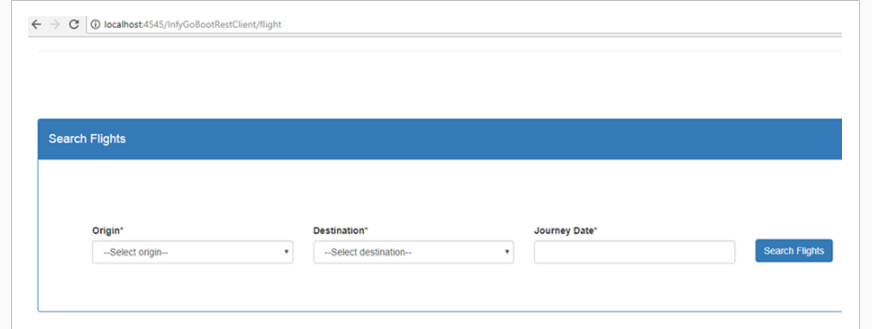
<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

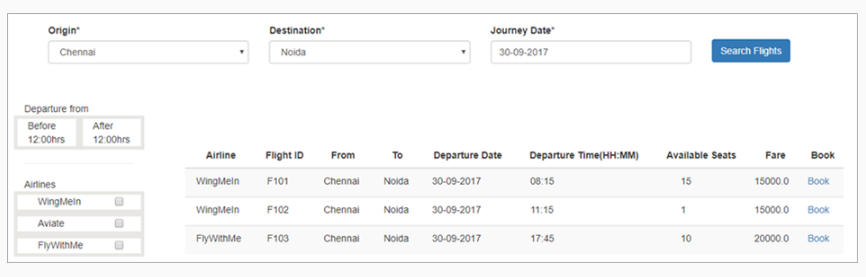
Step 23: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App.

Step 24: Type the url 'http://localhost:4545/InfyGoBootRestClient/flight' in the address bar of your browser and observe the 'searchFlights.jsp' page will be rendered for you as shown below.



Step 25: Give Origin, Destination, and Journey Date.

Step 26: Click on 'Search Flights' and observe the result rendered accordingly as shown below.



Thus you have built and consumed RESTful web service using Spring Boot.

**Things to observe**

Building RESTful web services in Spring Boot is the same as how you build in Spring MVC except for the way how you bootstrap the application. Adding spring-boot-starter-web dependency in your POM enables REST API for your application.

# **Exposing REST API**

FlightAPI.java is a @RestController annotated class. This annotation exposes the methods defined in this class as HTTP endpoints using @RequestMapping annotation.

@RestController

@RequestMapping("FlightAPI")

public class FlightAPI {

@RequestMapping(value = "/{source}/{destination}/{journeyDate}", method = RequestMethod.GET,headers="Accept=application/json")

public ResponseEntity<List<SearchFlights>> searchFlights(@PathVariable String source, @PathVariable String destination, @PathVariable String journeyDate) throws Exception{

Calendar jDate = CalendarUtility.getCalendarFromString(journeyDate);

List<SearchFlights> availableFlights = flightService.getFlights(source, destination, jDate);

return new ResponseEntity<List<SearchFlights>>(availableFlights,HttpStatus.OK);

}

# **Consuming REST API**

The InfyGoBoot\_RESTClient Spring Boot application consumes InfyGo's web service as below.

@Controller

public class FlightController {

@Autowired

private Environment environment;

private final RestTemplate restTemplate;

public FlightController(RestTemplateBuilder restTemplateBuilder){

this.restTemplate = restTemplateBuilder.build();

}

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

public ModelAndView flightSearch(@Valid @ModelAttribute("command") SearchFlights searchFlights,

BindingResult bindingResult, ModelMap model) {

String restUrl=environment.getProperty("RestUrl");

ModelAndView modelAndView = null;

List<SearchFlights> availableFlights = restTemplate.getForObject( restUrl + "/FlightAPI/" + searchFlights.getSource()

+ "/" + searchFlights.getDestination() + "/" + searchFlights.getJourneyDate(), List.class);

model.addAttribute("availableFlights", availableFlights);

model.addAttribute("size", availableFlights.size());

modelAndView = new ModelAndView("searchFlights", "command", searchFlights);

return modelAndView;

}

}

It consumes the web service using RestTemplate class in the same way how you do in the Spring MVC application. Spring Boot doesn't provide any auto-configured RestTemplate bean as it needs to be customized in most of the applications before it is being used. However, Spring Boot auto-configures RestTemplateBuilder class which can be used to configure and customize the RestTemplate instance as per the need.

Exercise: Build REST web service and consume it

## **Problem Statement:**

Modify the Spring boot application MovieMax of exercise 8 to expose the movie and flight details as RESTful web service.

* Implement MovieAPI class to expose below movie details as JSON data
  + All movie details
  + Movie details by language
  + Movie details by category
  + Movie details by city
* Implement TheaterAPI class to expose below theater details as JSON data
  + All theater details
  + Theater details city
  + Theater details by movie name
* Implement a client application named MovieScanner.
* Use RestTemplate object to raise an HTTP request to consume MovieAPI and TheaterAPI
* Display the JSON data received on the console.

Verification:

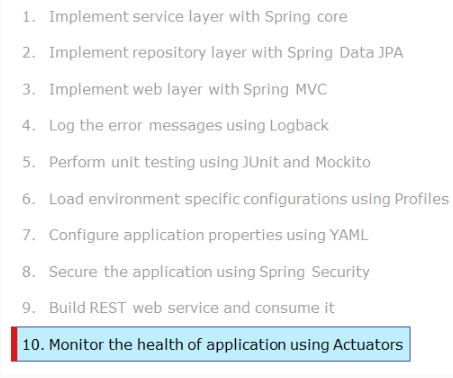
* Deploy MovieMax application on Tomcat server
* Run MovieScanner application. Observe the Movie and Theater details getting displayed on the console as JSON values

# Monitor the health of application using Actuator

So, you have successfully developed the IRS Spring Boot application and now it has to be deployed into production. How the production team will monitor your application? How will they get an insight into the running application's health, metrics, thread dump, environment variables, etc.,

Spring Boot's Actuator offers production-ready features to monitor the application and to get its metrics.

Let us see how to use Actuator.



Highlights:

* How to enable Actuator?

Demo Steps:

Step 1: Open STS and import the IRS project named 'InfyGoBoot\_SpringMVC'.

Step 2: Add the dependency in the 'pom.xml' as shown below.

<dependency>

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

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

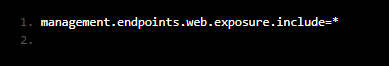
</dependency>

Step 3: Run the project by right-clicking on your project and choosing Run As -> Spring Boot App.

Step 4: Type the URL 'http://localhost:4343/InfyGoBoot/actuator/' and see the details in the browser as shown below.



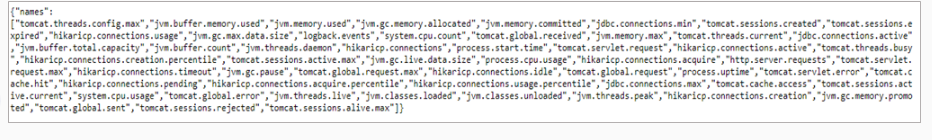
Step 5: Include the below line in the application.yml file.



Step 6: Type the URL 'http://localhost:4343/InfyGoBoot/actuator/beans' and see the bean details in the browser as shown below.



Step 7: Type the URL 'http://localhost:4343/InfyGoBoot/actuator/metrics' and see the bean details in the browser as shown below.



Thus actuator has been enabled in the application.

**Things to observe**

Actuators can easily be added to your application just by adding spring-boot-actuator dependency to the pom.xml file.

<dependency>

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

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

</dependency>

Once Actuator is enabled in your application with the above dependency in pom.xml, it provides several web endpoints through which the internals of the running application can be seen. These endpoints are exposed over HTTP in the Spring MVC application using the 'id' of endpoints as the URL path. For Spring Boot 2.0, the two default endpoints are /health and /info.

Other endpoints could be enabled by adding the below line in the application.properties file:

management.endpoints.web.exposure.include=\*

Any specific endpoint can be hidden as shown below:

management.endpoints.web.exposure.include=\*

management.endpoints.web.exposure.exclude=env

Any specific endpoint can be enabled as shown below:

management.endpoints.web.exposure.include=env,beans

For example,

* If you run your application and access the endpoint URI at /beans as http://localhost:4343/InfyGoBoot/actuator/beans, you should see all the Spring beans that are used in your application.
* If you access http://localhost:4343/InfyGoBoot/actuator/metrics, you should see 'metrics' information of your application, where you can view how much memory is used, how much memory is available, the size of the heap being used, the number of thread used, and so on.

Similar to /beans and /metrics endpoints, Spring Boot supports several endpoints.

Additional Reference: Refer Spring site to explore the complete list of Actuator endpoints and their details.

Actuators in Spring Boot

Changes in Spring Boot Actuator 2.0

* All actuator endpoints are now placed under the /actuator path by default.
* Improved JSON Structures
* Simplified process for creating User-Defined Endpoints

Additional Reference: For complete details on Spring Boot 2.0 actuator changes, refer to the Spring site.

# Create new endpoint

You can create your own endpoint by creating an Endpoint bean type as shown below.

import java.util.HashMap;

import java.util.Map;

import org.springframework.boot.actuate.endpoint.annotation.Endpoint;

import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;

import org.springframework.stereotype.Component;

@Component

@Endpoint(id="myendpoint")

public class MyEndpoint {

private Map<String, String> myUsers = new HashMap<>();

@ReadOperation

public Map<String, String> features() {

myUsers.put("FirstUser", "Abhinaya");

myUsers.put("SecondUser", "Viveka");

return myUsers;

}

@WriteOperation

public String writeOperation( String name) throws MalformedURLException{

return name + " \t in HTTP POST method" ;

}

@DeleteOperation

public String deleteOperation(){

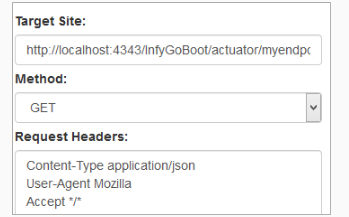
return "HTTP DELETE method"; }

}

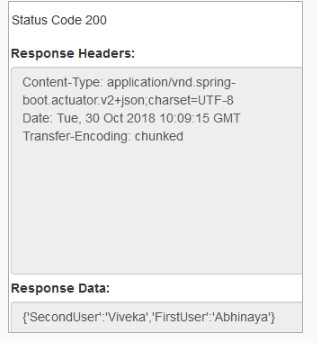
MyEndpoint mentioned above has the below:

* A method defined with @ReadOperation which will be mapped to HTTP GET method and automatically be exposed over HTTP.
* A method defined with @WriteOperation which will be mapped to HTTP POST method and automatically be exposed over HTTP. The methods that are annotated with @WriteOperation can take parameters in JSON format alone.
* A method defined with @DeleteOperation which will be mapped to HTTP DELETE method and automatically be exposed over HTTP.
* Technology-specific endpoints defined with @WebEndpoint/@JmxEndpoint. Forex, @WebEndpoint is exposed over HTTP only.

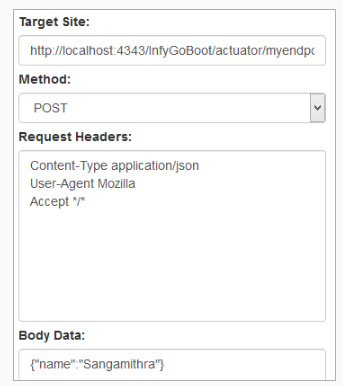
HTTP GET method - Request:



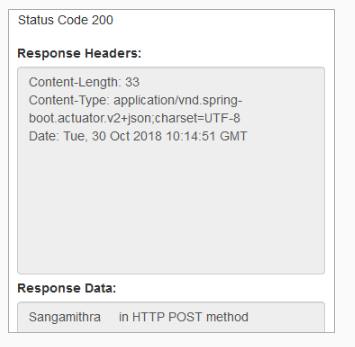
HTTP GET method - Response:



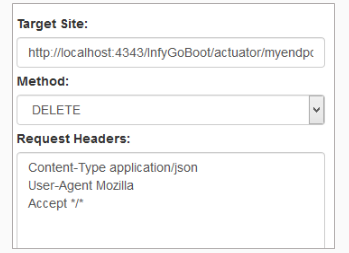
HTTP POST method - Request:



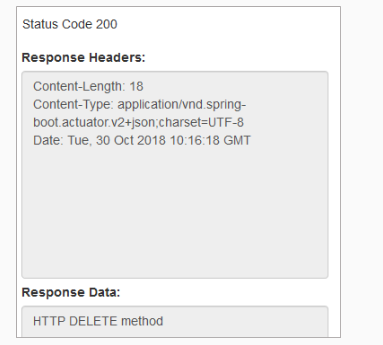
HTTP POST method - Response:



HTTP DELETE method - Request:



HTTP DELETE method - Response:



# Exercise: Monitor the health of application using Actuators

## **Problem Statement:**

Modify the Spring boot application MovieMax of exercise 9 to use Actuators.

* Implement a custom actuator with an endpoint named /maxmovies to display the list of movie names that are currently live.

Verification:

* Run your application.
* Invoke /maxmovies and verify that movie names are getting displayed

**Spring Boot Best Practices**

Below are the best practices to be followed in Spring Boot applications.



Use Embedded-configurations:

* Spring Boot provides a lot of embedded features like server, database, etc.
* To start the application quickly you can use the preconfigured database and server from Boot and later if required you can configure the new database and server in the Boot application.

Create new Spring Boot projects Via Spring Initializr :

* Spring Initializr site provides the best easy way to start a new Spring Boot project and load the required dependencies.
* Creating your application with the Spring Initializr ensures that you are selecting the approved and tested dependencies that will work fine with Spring auto-configuration.

Make @Controller as focused:

Controllers should be very thin. Create the controllers to manage and delegate, rather than to execute actual business logic.

Here are the best practices:

* Controllers should be stateless! By default, the controllers are singletons and giving them any state can cause massive issues.
* Controllers are not supposed to execute any business logic but rely on delegation.
* Controllers are connected to the HTTP layer of the application. This should not be passed down to Services.

Create @Service with business capabilities:

* Services are one of the most important core parts of Spring Boot where the business logic can be implemented, and it will be invoked from the controller.
* Naming conventions for the service classes: In the application name the services around the use cases rather than the functionality. Forex, EmployeeService, CustomerService, TransactionService are more appropriate than ValidationService, DatabaseService, etc.
* You can decide to achieve one-to-one mapping between Services and Controllers. That’s the ideal way but that does not mean that the Services can’t use each other!

Use Auto-configuration:

* The topmost feature of Spring Boot Auto-configuration. It will make your code easy to work. It will get triggered whenever a specific jar file is detected on the classpath.
* This will always rely on the Spring Boot Starters.
* Spring Boot auto-configuration tries to configure the Spring application automatically based on the included jar dependencies. For Ex, If HSQLDB is included in the classpath and it's not configured with any database connection beans manually, then it will auto-configure an in-memory database.

Use global exception handling:

* Spring Boot provides two ways to handle exceptions.
  + To define global exception you can use HandlerExceptionResolver.
  + Use @ExceptionHandler in controller advice, this will be useful to make specific exception handlers.

Logging framework:

* We are using a Logger for logging the messages instead of capturing the messages manually using System.out.println(). This can be achieved very easily in Spring Boot without any configuration.
* Simply include the logger instance like below:
  + Logger logger = LoggerFactory.getLogger(Demo.class);
  + Then we can set different logging levels as required.
* For starting the application quickly you can use the embedded logger of Boot i.e., Logback which provides very good support for Util Logging, Common Logging, Log4J, and SLF4J. You can customize with a new logger if required.
* The best practice to use the logger with different tracing levels in the application to capture all kinds of messages, this will help you to trace the application status whenever required.

Spring Bean Naming Conventions:

* The best practice to be followed when naming beans is standard Java field name convention i.e., bean names could start with a lowercase letter and are camel-cased from then on. For example, customerDao, customerEntity, registrationController, loginController and loginService, and so on.
* Naming beans will make your configuration easier to understand.

Constructor-based and setter-based DI in Spring or Spring boot applications:

* Spring provides three ways of Dependency Injection:

1. Field-based dependency injection
2. Constructor-based dependency injection
3. Setter-based dependency injection

* The best thumb rule: Use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.
* Most of the developers prefer to use constructor injection over setter-based injection because this will make the bean object as immutable.

Spring Boot Actuator:

Actuator provides support to monitor the below metrics in the Boot application.

* Gathering the metrics, analyzing the traffic and database status etc.,
* The major benefit of actuator is providing the support of production-grade tools without doing any actual implementations.
* It is majorly used to expose the information about the running application like health information, metrics, dump, env, etc.
* Once Actuator dependency is added several endpoints are accessible for us out of the box.
* The best practice to be followed while using an actuator we should not expose all the endpoints.
* By default, all the endpoints are enabled except shutdown.
* If required you can expose/hide the endpoints using include and exclude properties.

Make Proper Code Structure:

Follow the proper structure for your source code in the application.

* Do not use the default package. Ensures everything lives in a properly named package. So that there will not be any issue related to component scan and autowiring.
* Make your entry class i.e., Application.java (your Main Class) in the top-level of the source folder.
* The best practice is to keep all controllers in one package and all service classes in another package. Forex, Save all service classes together on com.infy.service package and all controller classes in com.infy.controller package.