Contents

[**Step 1:** 2](#_Toc456240318)

[**Step 2:** 2](#_Toc456240319)

[**For Spring MVC:** 2](#_Toc456240320)

[**Spring Boot Initializers:** 3](#_Toc456240321)

[**How does Spring Boot work ?** 3](#_Toc456240322)

[**Why Move to Containerless Deployments?** 5](#_Toc456240323)

[RestFul Web App: 5](#_Toc456240324)

[Spring MVC Integration Overview: 6](#_Toc456240325)

[Properties and Environmental Configuration: 7](#_Toc456240326)

[Demo: Configuring App with Properties: 7](#_Toc456240327)

[Provisioning and Integrating a DB: 8](#_Toc456240328)

[Setting up H2 Database: 8](#_Toc456240329)

[Configuring a Data Source: 9](#_Toc456240330)

[Flyway Integration: 10](#_Toc456240331)

[spring.jpa.hibernate.ddl-auto=false 11](#_Toc456240332)

[URL to Launch the application: 12](#_Toc456240333)

[Define Spring Beans in Java: 12](#_Toc456240334)

[DataSource setup: 12](#_Toc456240335)

**Step 1:**

Create a QuickStart Maven application.

**Step 2:**

Add the below Patent element in the pom.xml.

<parent>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-parent</artifactId>  
<version>1.3.5.RELEASE</version>  
</parent>

**For Spring MVC:**

Add the below dependency.

<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-web</artifactId>  
</dependency>

To enable Spring Boot:

**@SpringBootApplication** -> This annotation scans the project for spring components and autowires the most of the spring libraries by enabling auto configuration.

**SpringApplication.run(App.class, args)** -> This line actually fires up the Spring Boot application as the main method here run as a pure Java application.

@SpringBootApplication  
public class App  
{  
public static void main( String[] args )  
{  
SpringApplication.run(App.class, args);  
}  
}

**Spring Boot Initializers:**

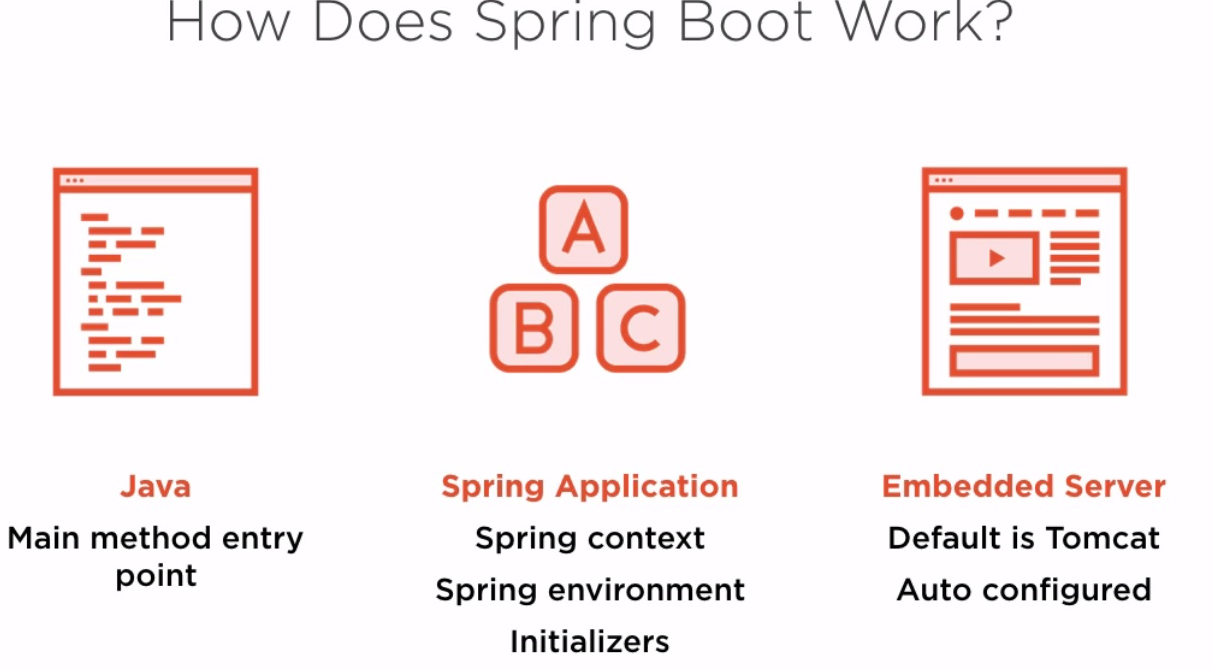
This is to generate a spring boot project based on your preference.

Go to the below link and then you can configure the starter based on the requirements.<http://start.spring.io/>

The below link has some sample projects that are configured and ready to use.

<https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples>

**How does Spring Boot work ?**



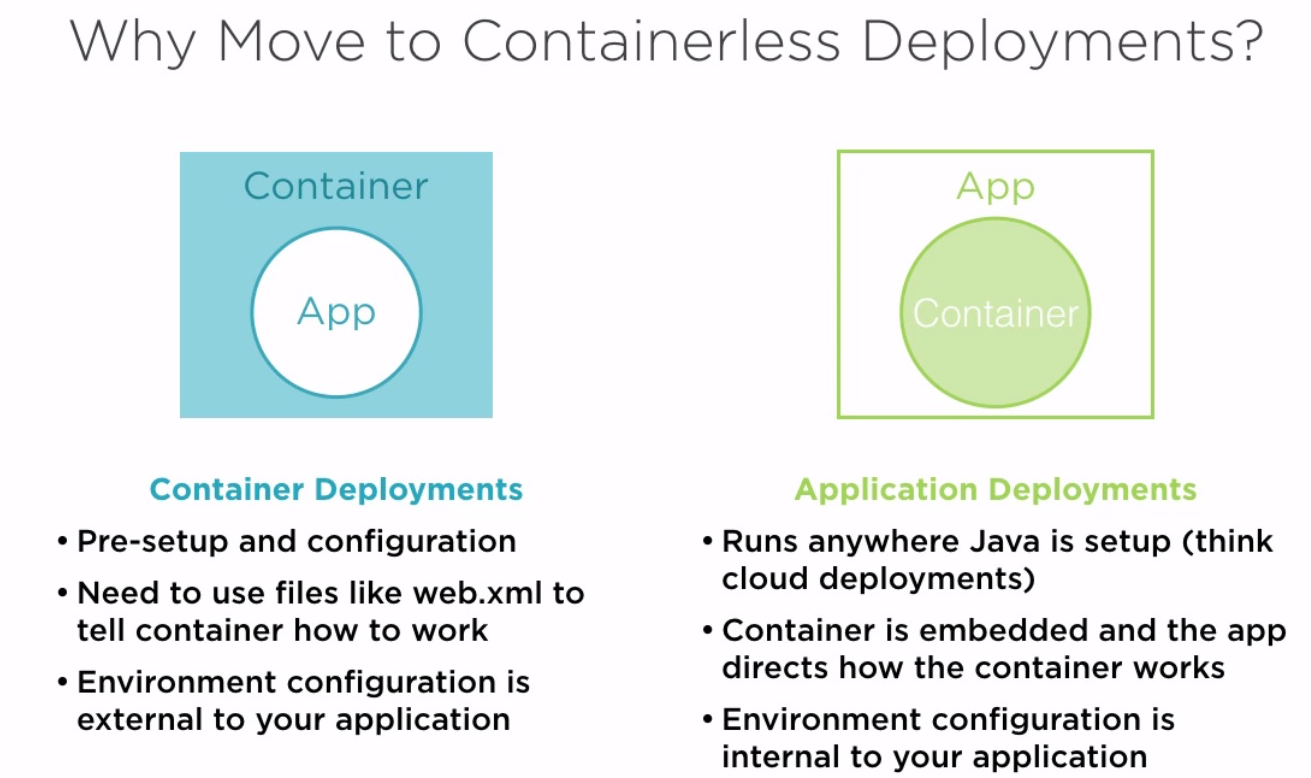
**1)** Public static void main()  
Starts Java and then the application.

**2)**@SpringBootApplication Annotation:  
A convenience annotation that wraps commonly used annotations with Spring boot.It sets up the spring mvc features for us.

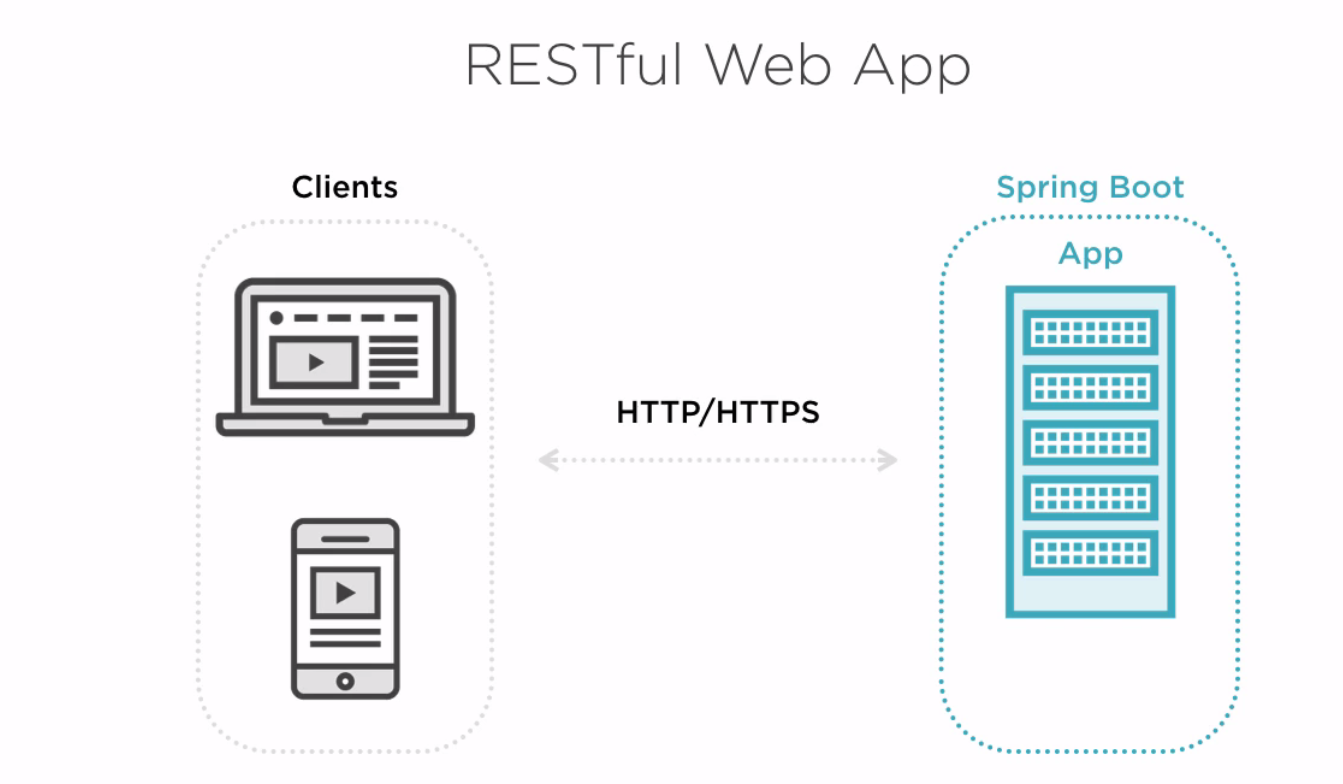
**3)**SpringBootApplication annotation has the following annotation behind the scenes. @Configuration - it acts as a Spring XML configuration. Instead of using Xmls to define the configuration here we are using Java to define the configurations.  
@EnableAutoConfiguration - This is a Spring boot specific annotation.Based on the type of project it autowires all the dependencies. If it is a web(spring mvc - spring-boot-starter-web) then it autowires all the web related spring libraries.  
@ComponentScan - It scans for the classes that are annotated with @controllers, @Services etc,.

**4)**SpringApplication.run: Starts Spring, Creates spring context, applies annotation and sets up container.

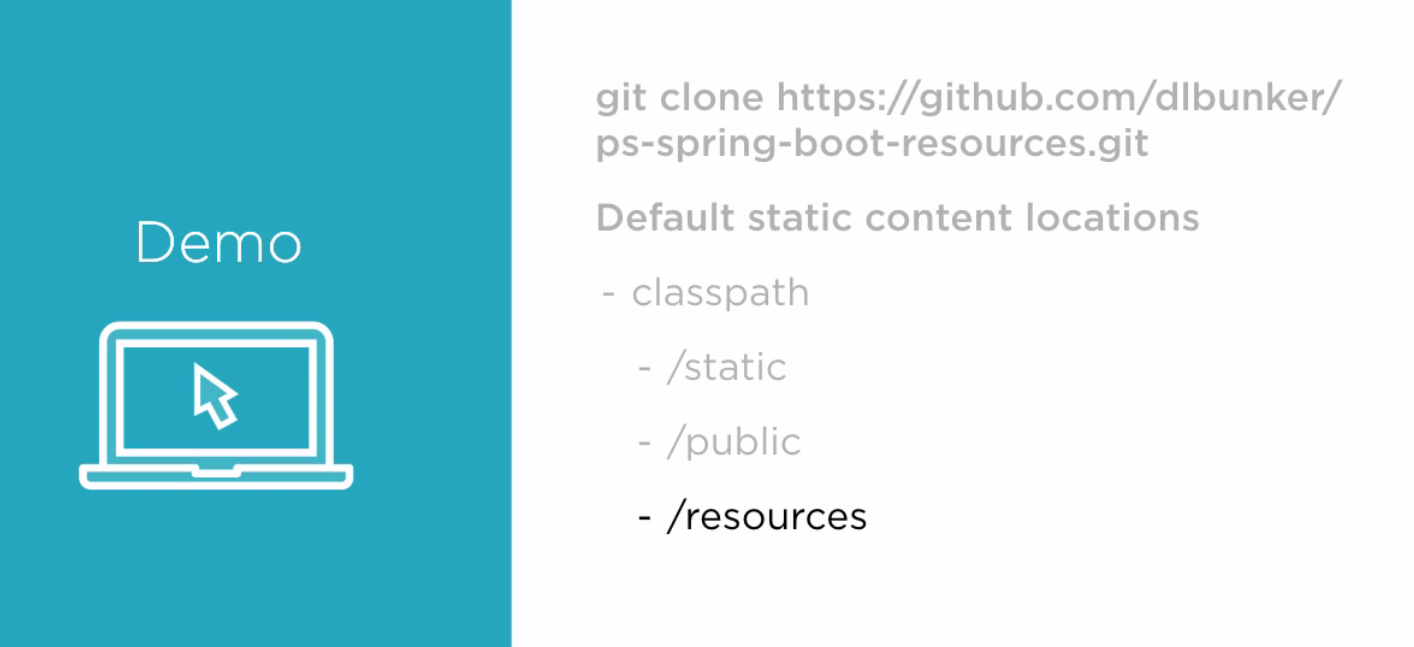
**Why Move to Containerless Deployments?**



## RestFul Web App:



Below is the location in which github looks for the files by default.

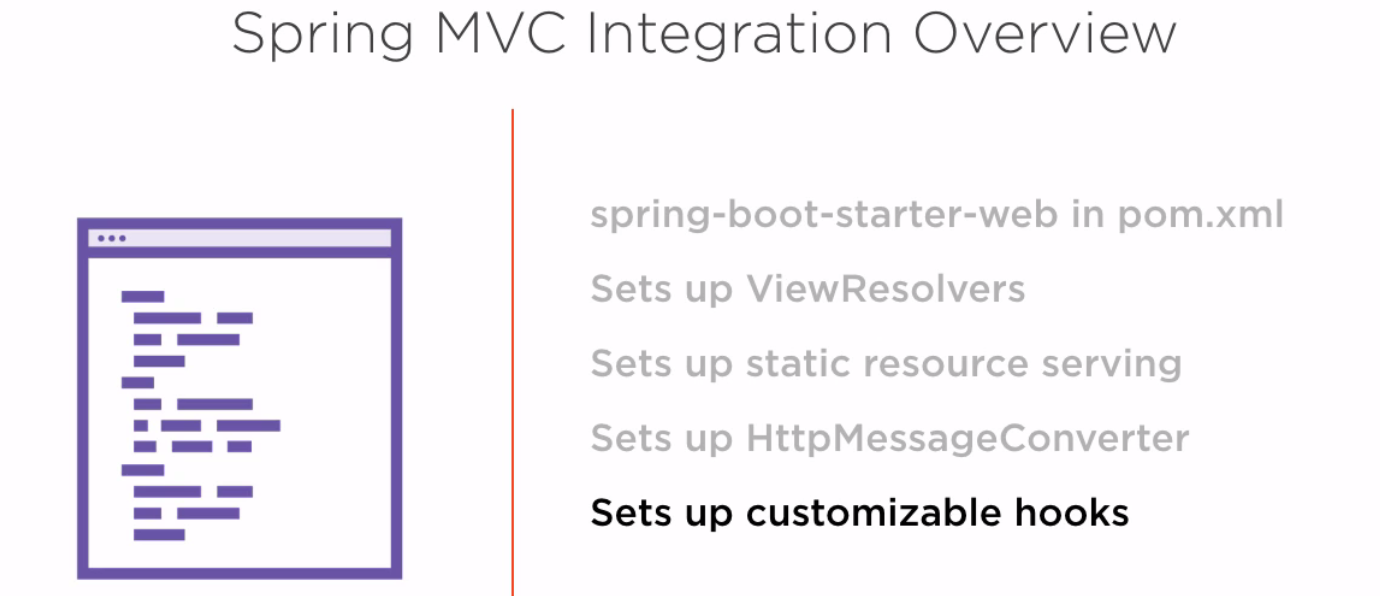


Go inside the www folder and copy the contents of the www folder.

C:\Dilip\Study\Spring\_Boot\ps-spring-boot-resources-master\ps-spring-boot-resources-master\client\www

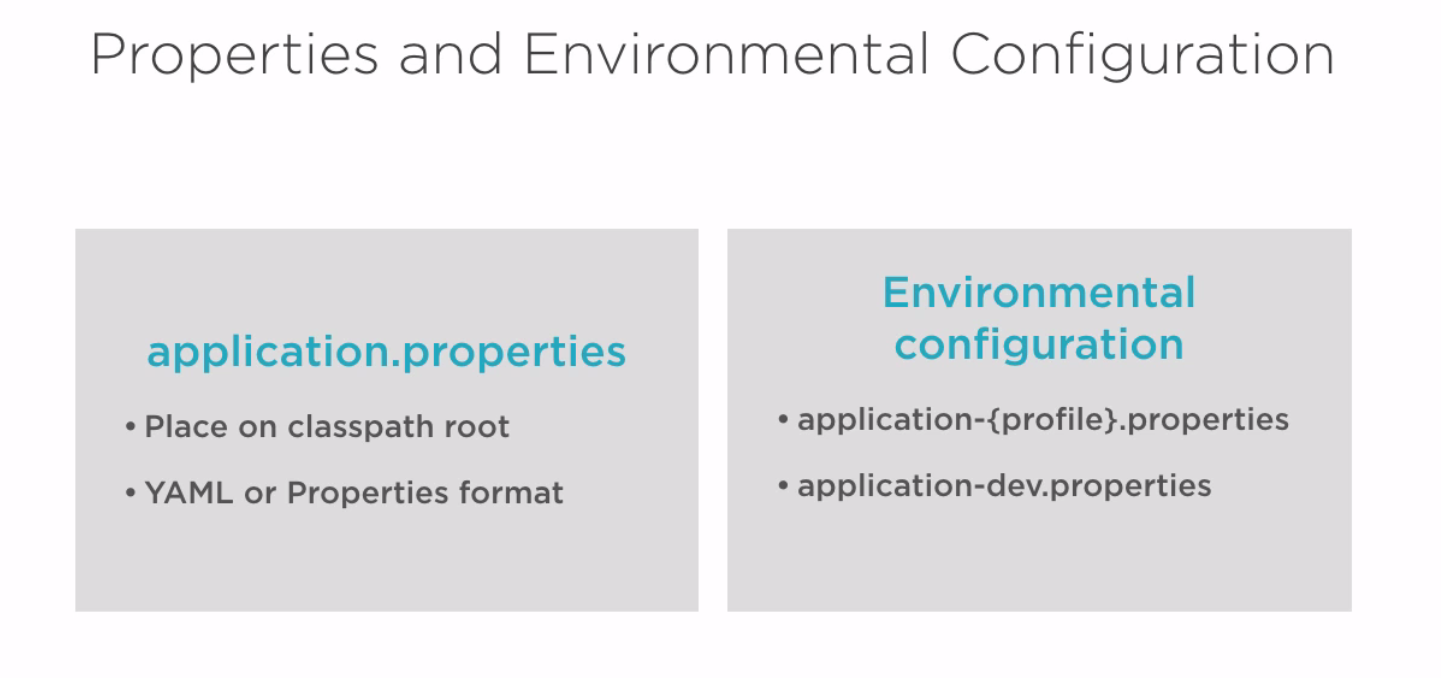
## Spring MVC Integration Overview:

Sets up HttpMessageConverter -> Jackson Mapper ( Converting Json to Java Object and vice versa).

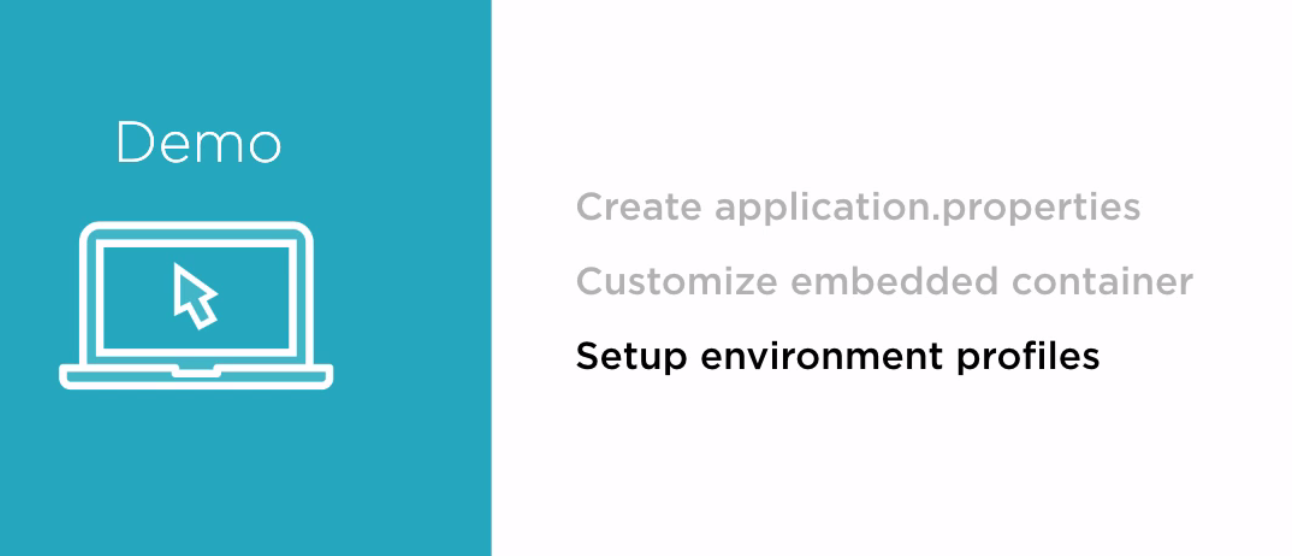


## Properties and Environmental Configuration:

Profile – It refers to the environment like dev, test, stage or prod.



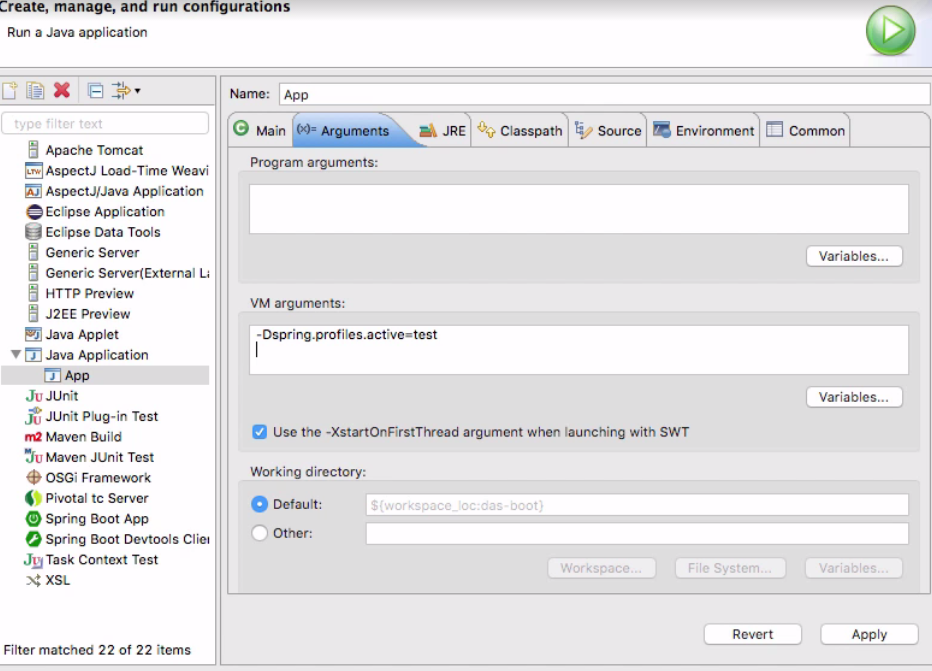
## Demo: Configuring App with Properties:



Place the application.properties in the class path. Place it in the resources package.

Passing the environment as a Vm argument

-Dspring.profiles.active=test



## Provisioning and Integrating a DB:



### Setting up H2 Database:

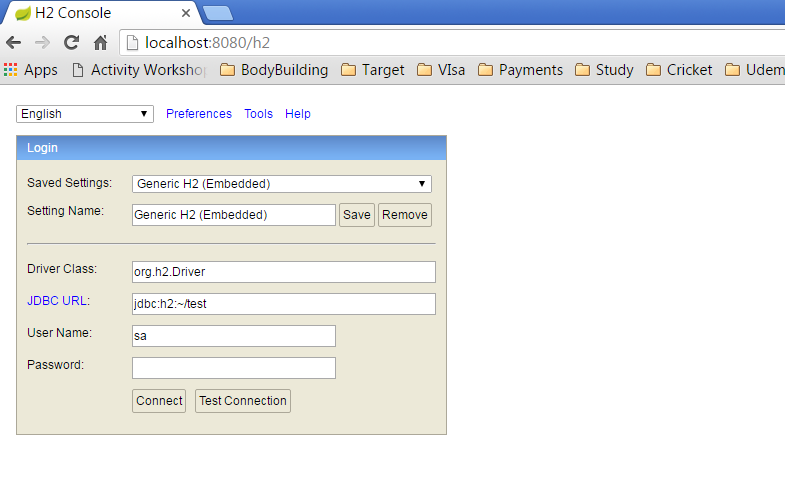
By providing the below values in the application.properties file.

spring.h2.console.enabled=true

spring.h2.console.path=/h2

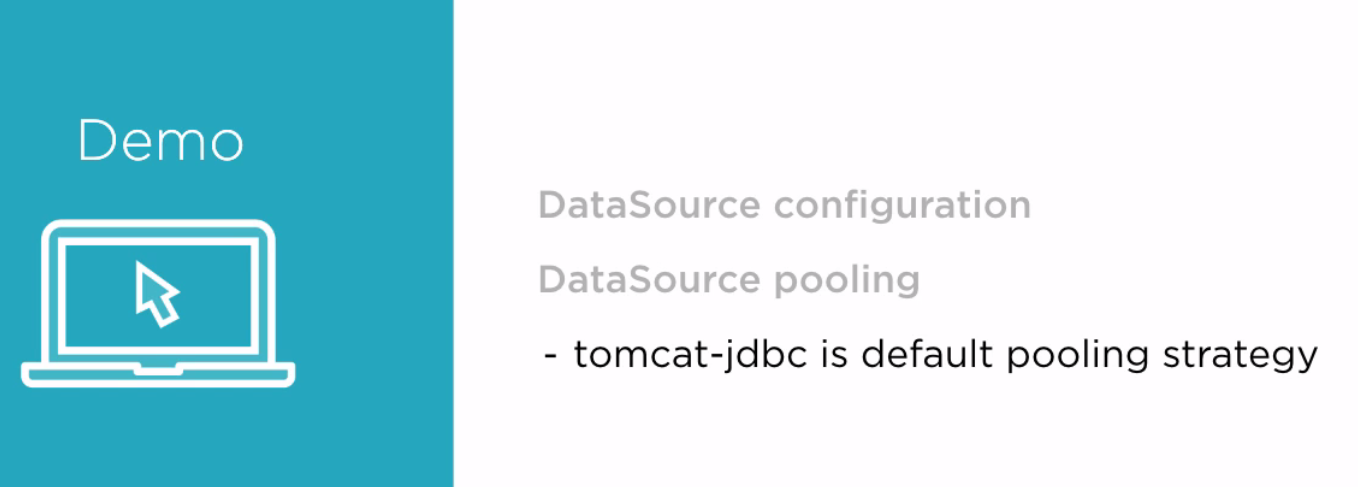
You can access the database from the below URL and click connect.

<http://localhost:8080/h2>



## Configuring a Data Source:

Data Source pooling is **tomcat-jdbc** by default.



Below configuration **application.properties** is needed to wire the app and the Data base.

jdbc:h2:file:~/dasboot -> This is to store the data base in a file. H2 data base is an in memory database. In order to store the data base across multiple sessions/restarts we are storing the database in a file.

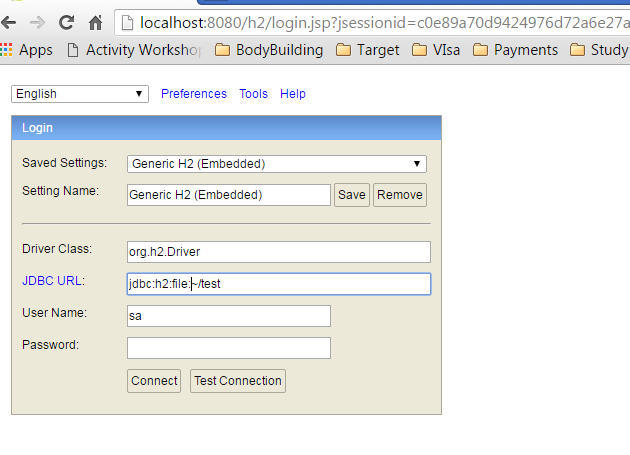
spring.datasource.url=jdbc:h2:file:~/dasboot

spring.datasource.username=sa

spring.datasource.password=

spring.datasource.driver-class-name=org.h2.driver

In the **JDBC Url** field change the below value from default.



Below details are needed for the data base pooling configuration in the **“application.properties”**.

#Database Pooling

spring.datasource.max-active= 10

spring.datasource.max-idle= 8

spring.datasource.max-wait= 10000

spring.datasource.min-evictable-idle-time-millis= 1000

spring.datasource.min-idle=8

spring.datasource.time-between-eviction-runs-millis= 1

## Flyway Integration:

Flyway will run the script mentioned in the resources path and creates the DB. Since Hibernate is also added as an JPA manager and we need to explicitly disable the hibernate to run the script.

Add the below changes in the application.properties.

flyway.baseline-on-migrate=true

## spring.jpa.hibernate.ddl-auto=false

Add the below pom dependency.

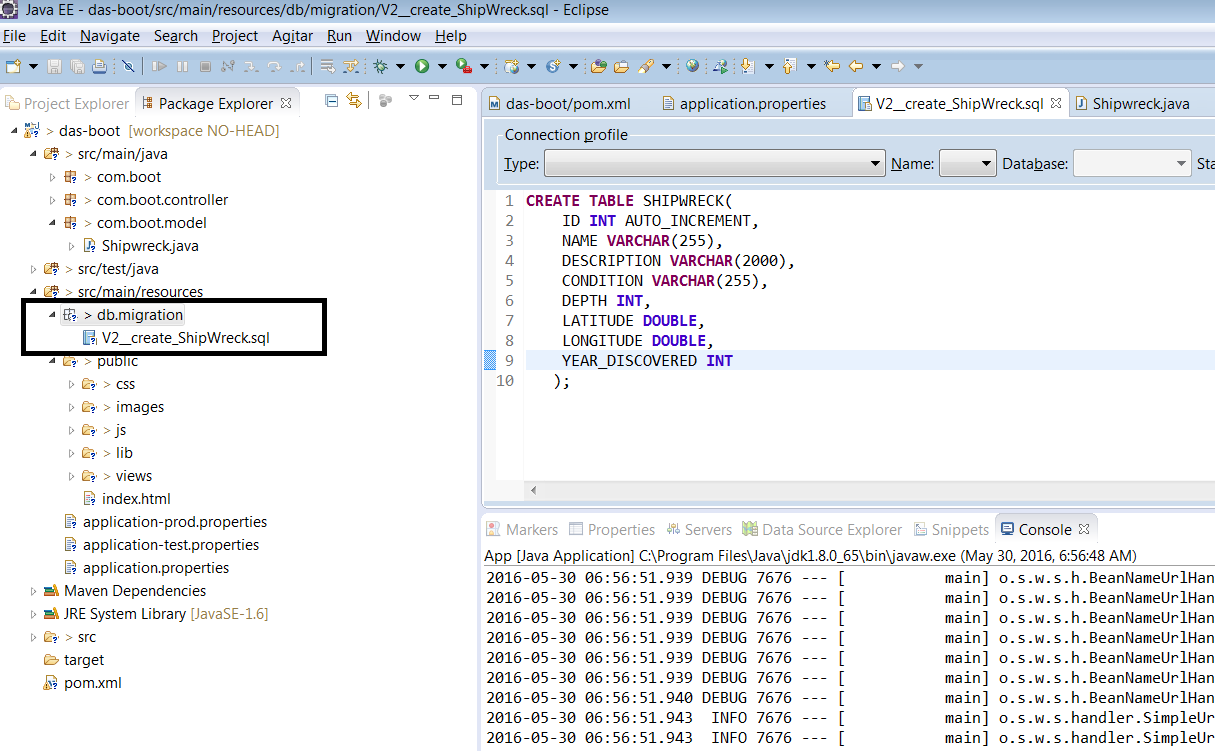
<dependency>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-core</artifactId>

</dependency>

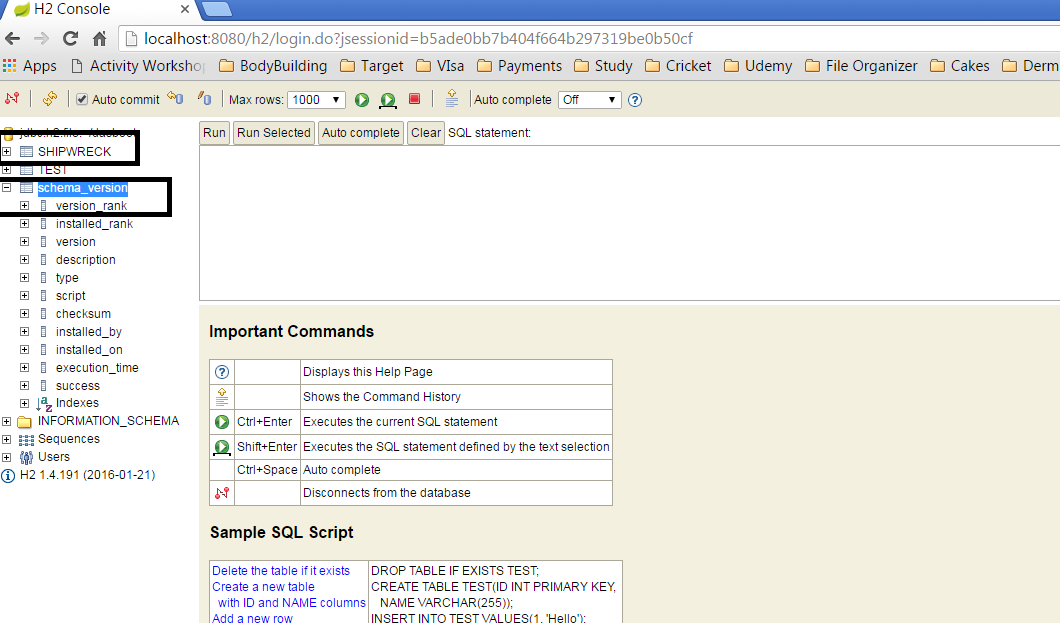
In the below screenshot you can notice that the db.migration has a ddl script from which the flyway Db will pick up the Db Script and run the script in the DB server.



After the run of the application the below tables are created.

Schema\_version is the meta table that gets created by flyway.

SHIPWRECK is the table that is created for the application.



# URL to Launch the application:

<http://localhost:8080/index.html#/>

# Define Spring Beans in Java:

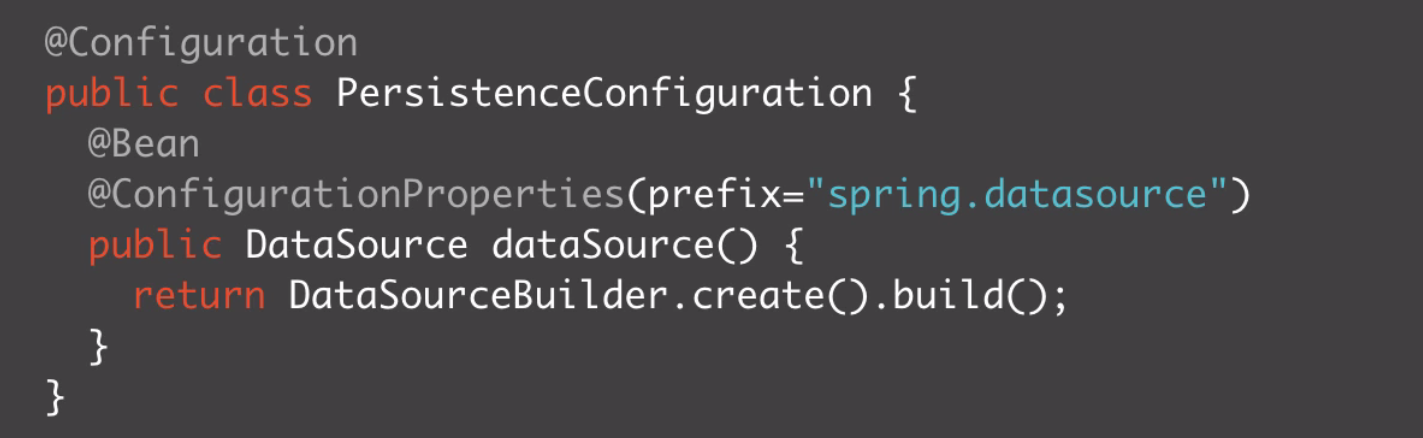
In older versions of spring most of the configurations are handled in spring xml file.

We can achieve all those in java code. This removes the XML dependency and allows us to configure any kind of app configuration programmatically on the app start up.

## DataSource setup:

We can create the configuration class and create the data source programmatically.

Any configuration java class that is defined with the **@Bean** annotation will be loaded during application start up and those beans will be available in the spring application context.

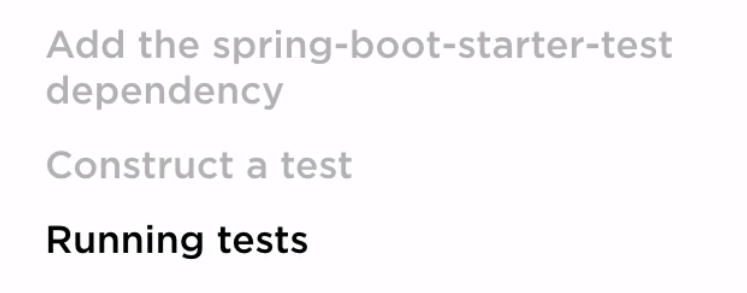


It is advisable to name all the **Configuration** classes to end with **Configuration**.

@Configuration classes are candidates for component scanning.

# Testing the Spring Boot App:

Follow the below steps to perform the testing in spring boot app.



## Spring boot starter test:

This dependency provides the necessary artifacts for the unit testing.

The scope is defines as “**test**” which means that when the app is bundled any dependency that is declared with test scope will not be part of it. This makes sure that the test scope dependencies are available only in the development mode or in maven test scope.

<dependency>

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

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

<scope>test</scope>

</dependency>

## Test using Mockito:

Updated a separate document for Mockito.

## Hamcrest Matchers:

This is similar to Junit assert but the below piece of code is more readable.

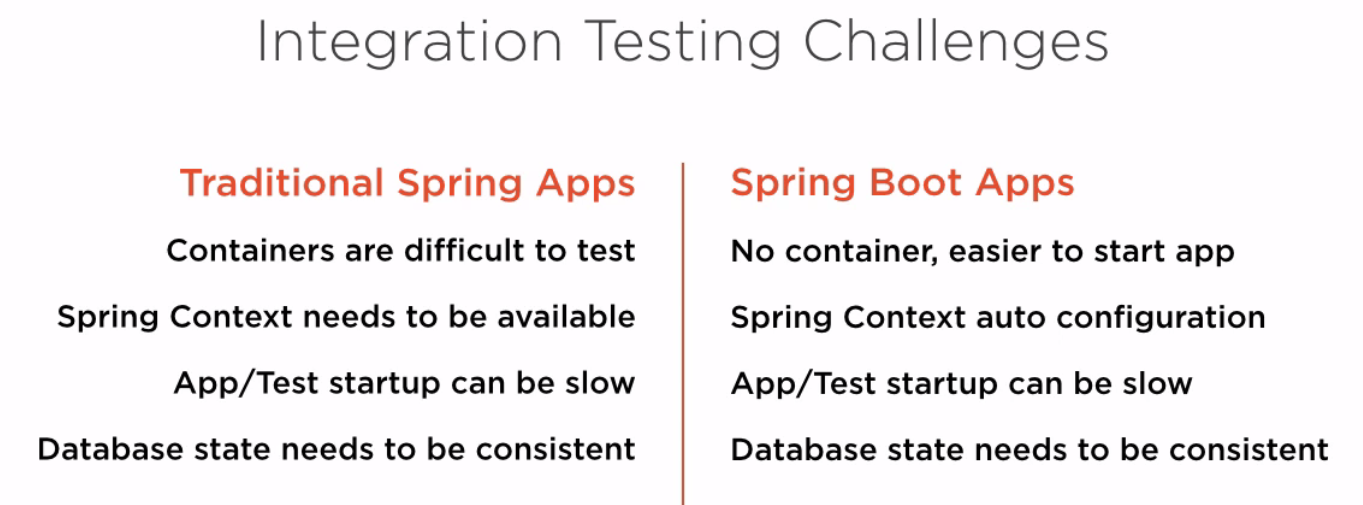
assertThat(shipwreck1.getId(), is(1l));

In order to have assertThat works we need to add the below static imports.

**import** **static** org.hamcrest.Matchers.\*;

**import** **static** org.hamcrest.MatcherAssert.*assertThat*;

## Testing Challenges:



Below annotations are required for running test cases for Spring boot applications.

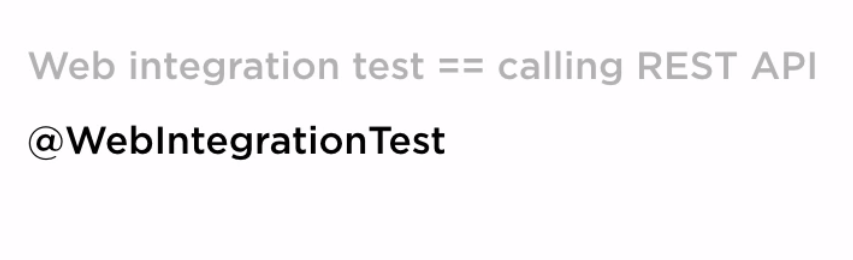


@RunWith(SpringJUnit4ClassRunner.class) -> SpringJUnit4ClassRunner is a custom extension of JUnit's BlockJUnit4ClassRunner which provides functionality of the *Spring TestContext Framework* to standard JUnit tests by means of the [TestContextManager](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/context/TestContextManager.html) and associated support classes and annotations.

@SpringApplicationConfiguration(App.class) -> This loads the spring container because the App.class is the starting point of this spring boot application.

By doing this you will get the complete spring context which is necessary to perform the integration testing.

## Web Integration Testing:



@WebIntegrationTest -> This is to test the API calls and to check their responses.

@RunWith(SpringJUnit4ClassRunner.**class**)

@SpringApplicationConfiguration(App.**class**)

@WebIntegrationTest

**public** **class** ShipWreckControllerWebIntegrationTest {

@Test

**public** **void** findAll(){

RestTemplate restTemplate = **new** TestRestTemplate();

ResponseEntity<String> response = restTemplate.getForEntity("http://localhost:8080/api/v1/shipwrecks", String.**class**);

*assertThat*(response.getStatusCode(),*equalTo*(HttpStatus.***OK***));

ObjectMapper mapper= **new** ObjectMapper();

**try** {

JsonNode responseJson = mapper.readTree(response.getBody());

*assertThat*(responseJson.isMissingNode(), *is*(**false**));

*assertThat*(responseJson.toString(), *not*("[]"));

} **catch** (JsonProcessingException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

## 