Table of Contents

[***1. Spring Boot Initial Steps:*** 2](#_Toc526465948)

[***2. Spring Boot Initializers:*** 4](#_Toc526465949)

[***3.*** ***Working of Spring Boot:*** 5](#_Toc526465950)

[***4.*** ***Why Container less? :*** 6](#_Toc526465951)

[***5.*** ***A Sample Restful Web App:*** 6](#_Toc526465952)

[***6.*** ***Integrating UI Client:*** 7](#_Toc526465953)

[***7.*** ***Server Side code – Rest Controller:*** 8](#_Toc526465954)

[***8.*** ***Overriding Default behavior - Application Properties and Environmental Configurations:*** 9](#_Toc526465955)

[***8.1.*** ***application.properties*** 9](#_Toc526465956)

[***8.2.*** ***Environmental Configurations:*** 9](#_Toc526465957)

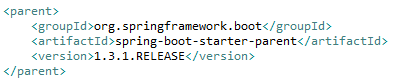
[***9.*** ***Spring Boot Data Integration*** 10](#_Toc526465958)

[***10.*** ***Spring Boot Testing*** 16](#_Toc526465959)

### 1. Spring Boot Initial Steps:

1. Create a new maven Project with maven archetype (maven-quickstart 1.1)
2. In POM.xml

**Define Parent as**

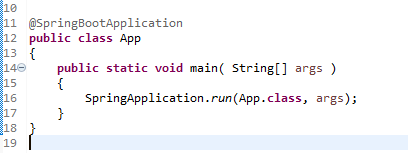


**And add new dependency**

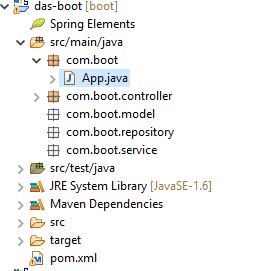


We do not have to enter version here because the version is obtained from parent.

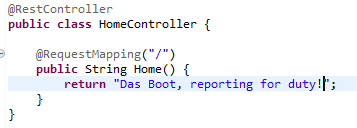
1. Then in Java Application write the following code. This is the starter class.



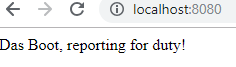
1. Create 4 packages to separate the concerns



1. Write a Rest controller in the controller package.



1. Run the app as a java app and type <https://localhost:8080>



### 2. Spring Boot Initializers:

1. Web Initializer

<https://start.spring.io/>

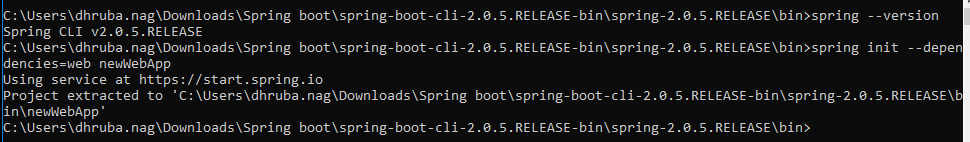
Creates a maven project with all necessary components and downloads as a zip file.

1. Spring Boot CLI(command line)

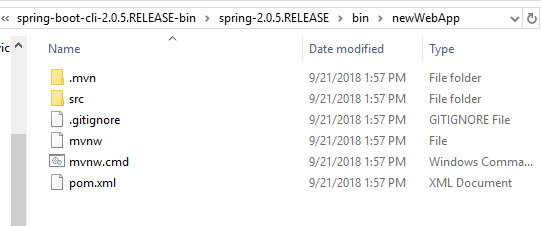
<https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.0.5.RELEASE/spring-boot-cli-2.0.5.RELEASE-bin.zip>

Download the CLI

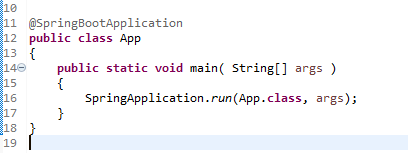
Navigate to the bin folder through command line and run the following commands.



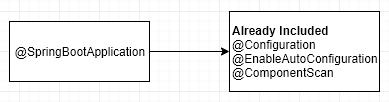
A new project will be created at the location mentioned.



### Working of Spring Boot:



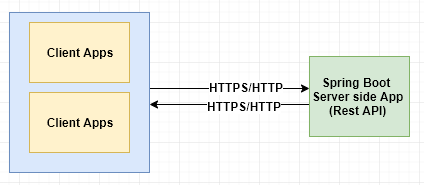
|  |  |  |
| --- | --- | --- |
| Index | Components | Functions |
| 1 | Public static void main | Starting point of a Spring Boot Application just like older java applications |
| 2 | @SpringBootApplication | A convenience annotation that wraps annotations commonly used with Spring Boot |
| 2.1 | @Configuration | Spring Configuration on startup. This will be used to configure the spring context by Spring Boot |
| 2.2 | @EnableAutoConfiguration | Auto Configures other compatible frameworks on class Path |
| 2.3 | @ComponentScan | Scans directory and subdirectory of package containing this class. So this class should be at the top. |
| 3 | SpringApplication.run(className.class, args); | Starts spring, creates spring context, applies annotations and configurations, places any spring components in spring context and sets up embedded container |



### Why Container less? :

|  |  |
| --- | --- |
| **With Containers** | **Without Containers** |
| Pre Set up and Configuration. The container and the server needs to be setup and configured to run the application | Only needs Java. Can be run in any host which has Java. The container is embedded inside the application. We just have a jar file to run. |
| Deployment Descriptors – web.xml  This is needed to tell the container how to deploy and serve up the application | No deployment descriptors are needed since the container is embedded inside the application. The spring boot property files can be used to configure setup of application. |
| All environment settings need to be provided as external settings. e.g. JNDI configurations set up in server | All environment settings are internal |

### A Sample Restful Web App:



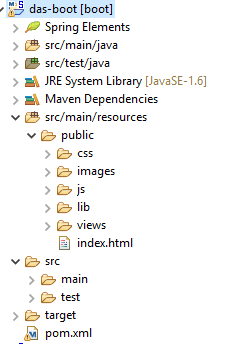
### Integrating UI Client:

Default Static Locations: /static, /public, /resources

We can put static files for UI layer (angular-JS files and HTML) here under

Src>main>resources> (static/public/resources)

e.g.:



After src/main/resources/ (static/public/resources) is created, run a maven update to include these folder and its contents in project class path.

### Server Side code – Rest Controller:

**Base URL:**



|  |
| --- |
| **Endpoints** |
| GET /api/v1/shipwrecks (list) |
|  |
| POST /api/v1/shipwrecks (add) |
|  |
| GET /api/v1/shipwrecks/ {id} (view) |
|  |
| PUT /api/v1/shipwrecks/ {id} (update) |
|  |
| DELETE /api/v1/shipwrecks/ {id} (delete) |
|  |

|  |
| --- |
| **Role of Spring Boot** |
| Enables Auto configuration.  It tells spring mvc to set up view resolvers like content negotiating view resolvers.  It sets up Jackson – Json library to handle views for application/json types |
| It sets up spring mvc to serve static content from class path root / (static/public/resources) folder |
| It sets up spring mvc http message converters so that it can convert JSON objects to java and vice versa. |
| The basic string encoding is set to UTF 8 out of the box with Spring boot |
| It leaves a way to override default behavior by means of programmable hooks. |

### Overriding Default behavior - Application Properties and Environmental Configurations:

Src/main/resources/application.properties

The application properties can be standard java properties or YAML format (for YAML we need to add Snake-YAML dependency)

Environmental Configuration:

application-{profile}.properties e.g. : application-dev.properties

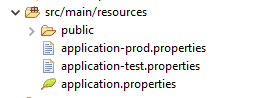
This overrides the properties defined in application.properties based on environment.

### application.properties

|  |  |
| --- | --- |
| Property | Function |
| logging.level.org.springframework.web=debug | This helps us to set logging level without log4j configuration |
| server.port=8181 | The port the application is running on. If nothing is mentioned, default is 8080. |

### Environmental Configurations:

Step 1:

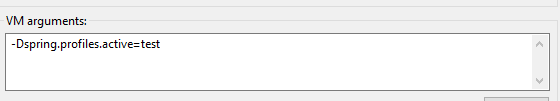


Test has server.port = 9090

Prod has server.port = 80

Step 2:

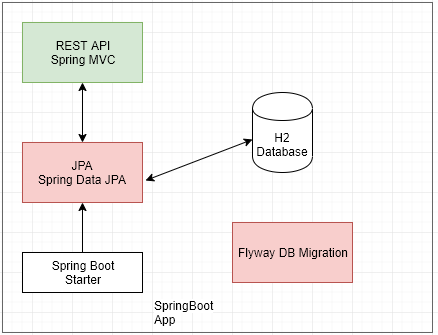
Run Congurations>Run as>Arguments>VM Arguments



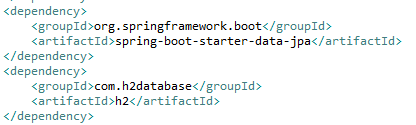
Common Spring Boot Application properties:

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

### Spring Boot Data Integration



1. In pom.xml add the following dependencies.

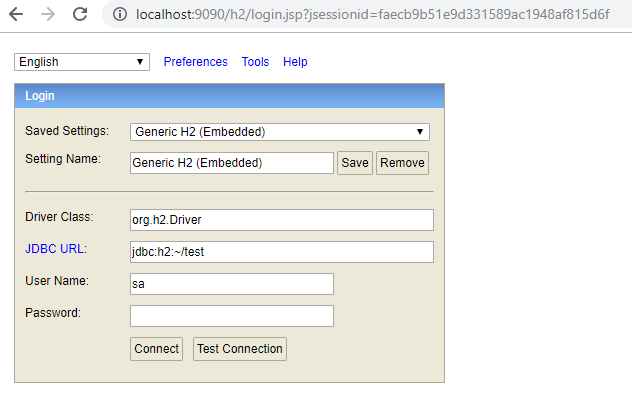


1. In application.properties file add the below configuration



1. Access H2 database with the following URL and default creds

<http://localhost:9090/h2>

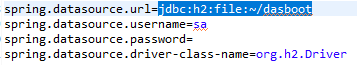


Database configuration and pooling libraries need to be defined in application.properties.

By default spring boot starter data jpa uses tomcat-jdbc as default pooling strategy.

commons-dbcp, commons-dbcp2 can also be integrated.

1. Add the data source properties in the application.properties file.



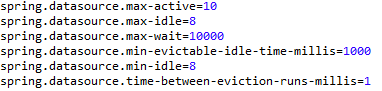
1. Connect to H2 through browser and change the DB URL



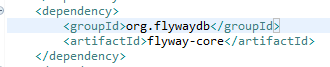
1. Create some test data



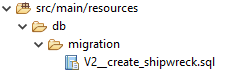
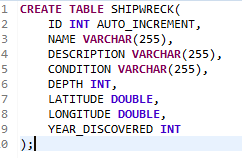
1. Pooling properties



1. Flyway Integration
2. Add POM dependency



1. Create Migration Scripts under src>main>resources

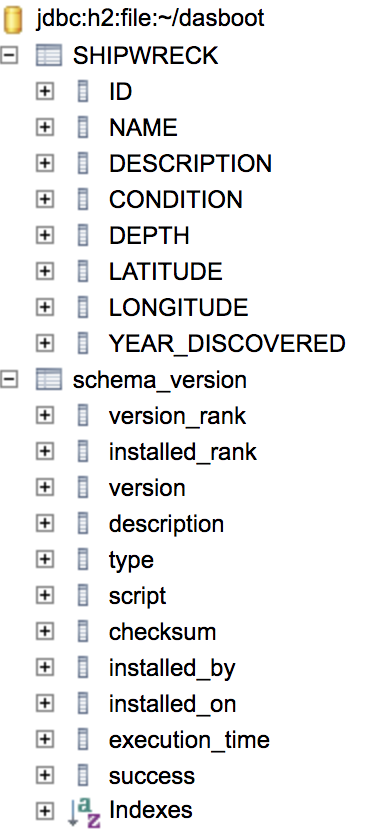
 

This will create a table in database next time application is fired.

1. Add properties to disable spring JPA behaviour and generate metadata for flyway.

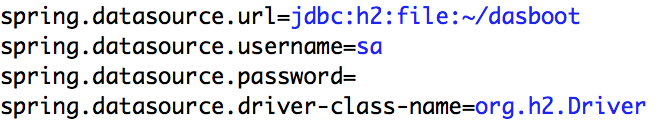


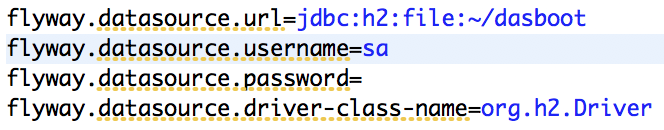
1. Run application

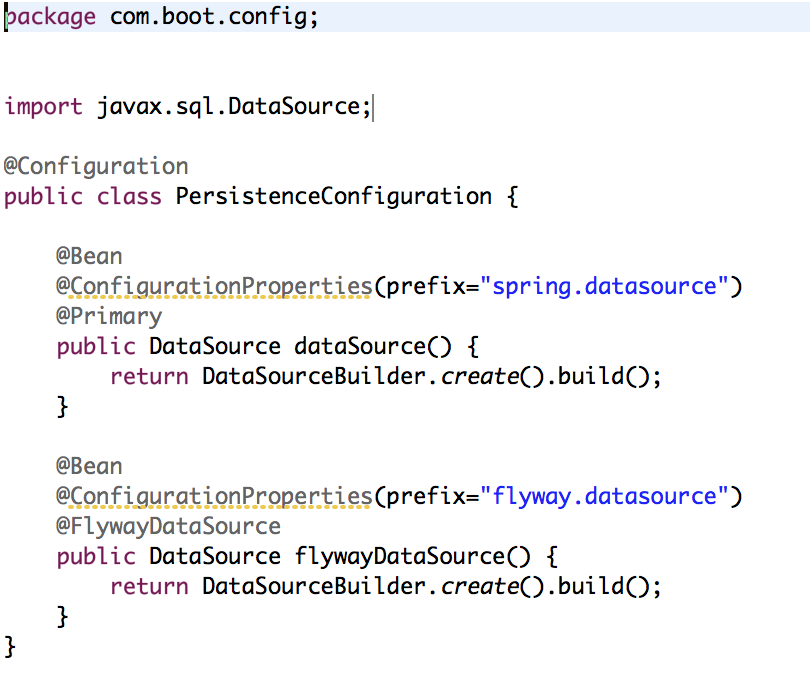


1. Configuring Data Sources.

The datasources defined in the property files need to be configured. Earlier it was done using xml. Now Java configuration can be used.



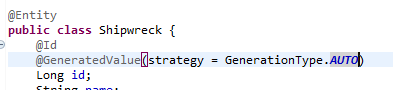




|  |  |
| --- | --- |
| Annotations | Functions |
| @Configuration | Denoted to Spring boot that this is a configuration class |
| @Bean | A bean needs to be created |
| @ConfigurationProperties  (prefix=”spring.datasource” | The properties to be used while configuring the bean |
| @Primary | This is the primary data source. Should not be ambiguous. |
| **@FlywayDataSource** | To mark a data source to be used by Flyway |

1. Spring Data and Repository
2. Convert ShipWreck.java under model to a JPA Entity.





|  |  |
| --- | --- |
| Annotations | Functions |
| @Entity | Defines the class as a JPA entity |
| @Id | Primary Key |
| @GeneratedValue(strategy = GenerationType.AUTO) | Generates a sequence automatically for primary key |

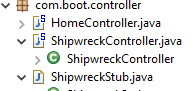
1. Create a repository interface under com.boot.repository.



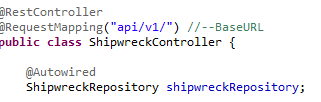
This may extend JPARepository



1. Go to the ShipWreck Controller and remove references to all methods using shipWreckStub

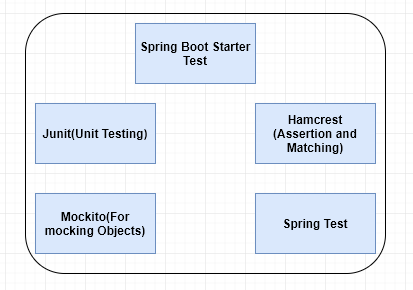


First autowire ShipwreckRepository and since it extends Jparepository , it will inherit some JPA methods

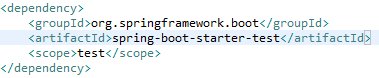


|  |  |
| --- | --- |
| CRUD | CREATE/READ/UPDATE/DELETE |
| CREATE | saveAndFlush(shipWreck) |
| READ | findAll()    findOne(id) |
| UPDATE | findOne(id) and then saveAndFlush |
| DELETE | findOne(id) and then delete(Entity) |

### Spring Boot Testing

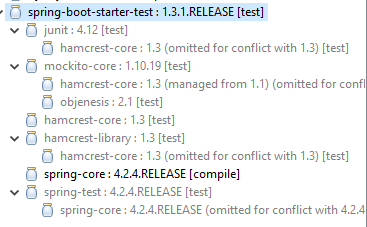


1. Dependencies

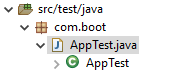


This pulls in the following dependencies for junit, hamcrest, mockito and

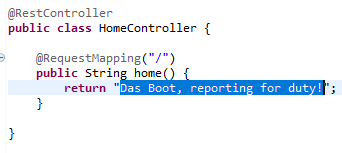
Spring test. This dependency is defined with the scope of test which means it will not be included during deployment. It is only available during test phase.



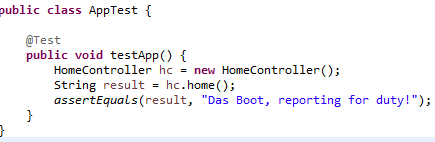
1. 1st test class(Modify archetype generated AppTest.java



Code to test on:



Test code:



Tests can be run in 3 ways

1) Right Click – run as Junit Test

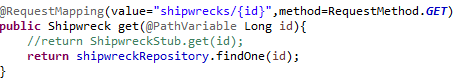
2) Right Click on project – run as maven test

3) Command line, navigate to directory containing POM e.g. cd C:\work\eclipse\_workspace\TEST\das-boot and type mvn test

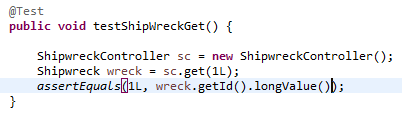
1. Mock objects – Mockito

We now try to test the ShipWreckControllerClass

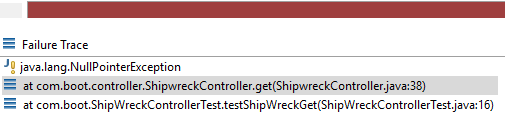
Code to Test:



Test Code:



When we run this, we get null pointer exception because we manually instantiated the controller and dependencies like shipwreckRepository were not injected by spring into it





So we modify the code using Mockito.



Line 21 to Line 26 involves declaring mocks

Line 27 to Line 30 runs before each @Test and injects the mocks

Line 35 to 37: An object is stubbed. A shipwreck is created and id 1L is set to it.

Line 39: whenever a mock behavior is invoked, stubbed object is returned.

Whenever findOne(1L) is invoked on mock object shipWreckRepository, the stubbed shipwreck is returned.

Line 41: Actual business logic is executed.

Line 43 checks whether due to business logic execution, findOne(1L) has been invoked on mock object shipWreckRepository

Line 45 compared value returned during business logic execution to value in stubbed object

1. Hamcrest – Better readable matching



The commented line is replaced by code written using hamcrest making it more readable.

The imports needed are as follows:



1. Integration testing

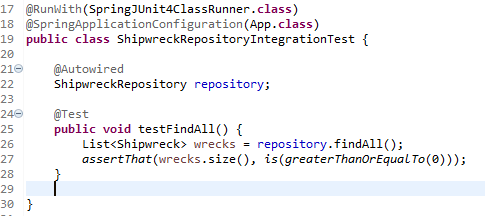
This refers to testing all pieces of the app together as it should run in an environment.

Before spring boot, it was difficult to test the container and prop up the spring context.

Spring boot bundles apps as a jar with a built in container and also makes propping up spring context easier due to all the starter classes.

But testing time is still slow and we still face issues while testing with a database because value in databases may have already changed compared to our test code.

Integration Test Code:



Step 1. Line 17 states that it must be run with SpringJunit4Runner from spring test starter

Step 2. Line 18 defines the startup class for spring boot app. Every time the test is run, the app boots up making the test very slow.

Step 3. Required objects are auto wired by spring boot app.

1. Web Integration testing

They involve testing the actual rest APIs.

WebIntegrationTestCode:



Line 23 states that this is a webintegrationtest.

A rest URL is invoked programmatically in Line 29/30 and the response is checked for status 200 in line 31.

The response in then converted to Json and since there are no records, the value of the Json response is matched to [](empty).