# SPRING BOOT

## DOCUMENTATION

# Table of Contents

# Spring Boot

## 1. Introduction to Spring Boot

Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications. This document will give you an introduction to Spring Boot and familiarizes you with its basic concepts.

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that can **just run**. Users can get started with minimum configurations without the need for an entire spring configuration setup.

### 1.1 Spring MVC vs Spring Boot

| Spring Boot | Spring MVC |
|---|---|
| Avoids boiler plate code. Wraps dependencies together in a single unit. e.g spring-boot-starter-web | Specify each and every dependency seperately. This takes more time. |
| Can be packaged as a jar with embedded server by default. | Will need a lot of configuration to be written to achieve the same. |
| Reduces development time and increases productivity so that production ready applications can be built quickly. | Takes significantly more time to achieve the same - example of time consuming tasks would be error prone dependancy additions where versions don't match. |

## 2. Advantages

Spring Boot offers the following advantages to its developers –

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

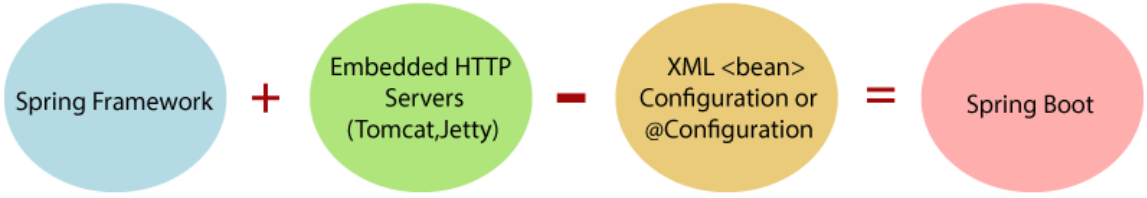
## 3. Goals

Spring Boot is designed with the following goals –

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way

- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application
- It includes Embedded Servlet Container

## 4. Working with Spring Boot




Spring Boot automatically configures your application based on the dependencies you have added to the project by using **@EnableAutoConfiguration** annotation.

The entry point of the spring boot application is the class contains **@SpringBootApplication** annotation and the main method.

Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation.

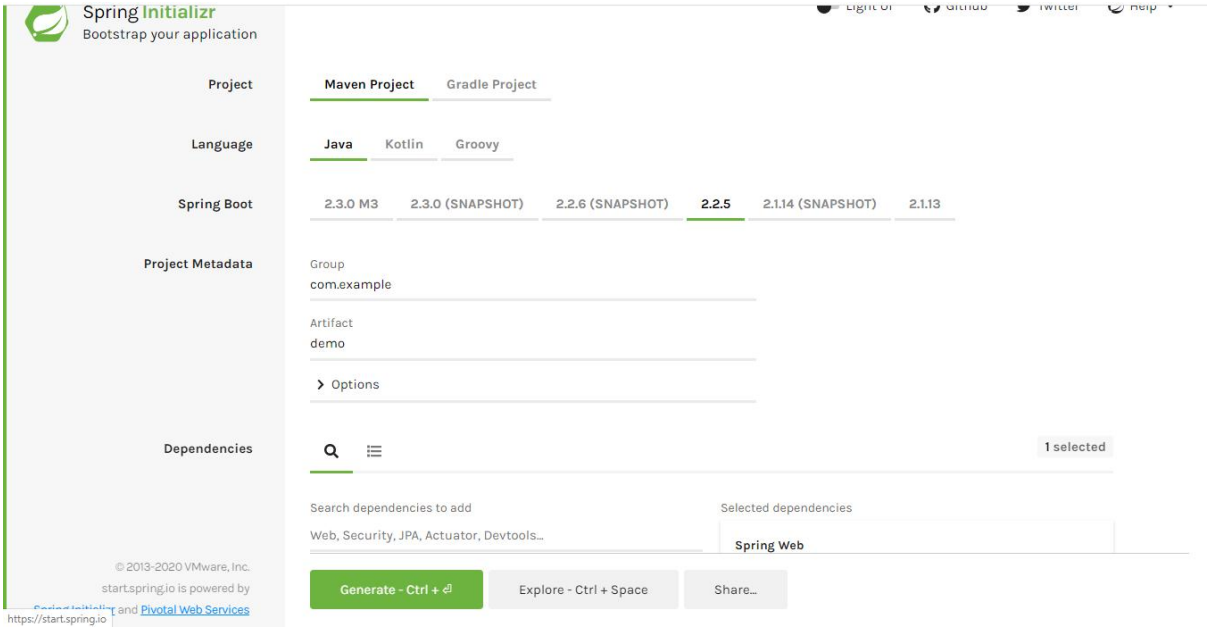
### 4.1 Prerequisite of Spring Boot

To create a spring boot application the following are the prerequisites.

- Java 1.8 +
- Starter project from **https://start.spring.io/** or Spring Tool Suite or CLI
- IDE

### 4.2 Downloading Spring Boot Starter Project

Go to spring initializr website and select the maven project with java language , spring boot version and download the spring boot starter project with project meta data as given and also with the required dependencies selected.




## 4.3 Setup the Starter Project in Eclipse IDE

Import the downloaded project into workspace and give maven update.

- After maven update the dependencies in pom.xml will be downloaded into local repository from the central repository and project structure will be updated with spring boot project structure.
- Add a controller in src/main/java folder to create an rest endpoint and run simple boot application with embedded server tomcat.
- By default by adding spring-boot-starter-web dependency will add the jars required to build web application and rest web service.
- Spring Boot uses Maven Build Tool for building the project using maven plug-in added in pom.xml
- Maven is build management tool that looks at pom.xml and manage dependencies and generates war/jar according to packing where default packaging is jar.
- Maven has different lifecycles. They are default cycle fir deployment, clean cycle for cleaning the project and site for documentation
- The default cycle phases are

  validate -> compile -> test -> package ->verify ->install ->deploy

  when we run command to install all it's previous phases (validate, compile, test, package, verify ) will be executed before executing install.

## 4.4 Required and optional Dependencies

Spring-boot-starter-web dependency is mandatory dependency for web application and there are other optional dependencies with specific purpose like Lombok dependency to avoid boiler plate code in model, tomcat –jasper dependency support to view jsp pages.

- spring-boot-starter-actuator to manage and analyse metrics of application endpoints.
- Spring boot requires maven-compiler-plug-in for build management to build the project and can run on maven profiles defining profiles in pom.

## 4.5 Running Spring Boot Application

Right click on project and click on run configurations and type clean install to build the application which will generate war file and install it on local repository and later the war should be deployed on any server or give spring - boot : run to run spring boot application in eclipse on embedded tomcat server.

## 4.6 Spring Boot with rest endpoint

Now, your main Spring Boot Application class file will look like as shown in the code given below −

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {
public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }
@RequestMapping(value = "/")
   public String hello() {
  return "Hello World";
}
}
```

The endpoint will return String HelloWorld on accessing the url : locahost:8080/ from any browser.

## 4.7 Spring Boot application file

Every Spring Boot project will contain Application.properties file which is used to override the defaults set by spring boot.

- When tomcat has to be runned on random port then add the key value pair into application.properties file as below
  server.port=0

- When the developer want to specify the port on which tomcat should run the value can be given like server.port=9090 , now tomcat will start on port 9090 instead of it's default port 8080.
- In order to run on different profiles in spring boot application.properties file can be used, create copies of application.properties file with names
  1. Application.properties
  2. Application-dev.properties
  3. Application-uat.properties
  4. Application-prod.properties

  As the Database credentials vary from environment to environment we can maintain database credentials for different environments in different properties file and can run the application in profiles so that dev environment will access dev database and prod environment will access prod database dynamically on runtime.

  When there is no key like export spring_profiles_active is given in application.properties file then the default application.properties file will be read at runtime

  When the value for key export spring_profiles_active=dev then dev credentials will be accessed during runtime connecting to dev database.

  Here are some of common properties used in application.properties

1. Server.port=9090 (will set embedded server tomcat port)
2. Spring.application.name=DemoApp (will register in service registry name of service in eureka service registery used while building microservices)
3. spring_profiles_active=dev used to run application in different profiles
4. debug=true logs in debug mode
5. logging.path = /var/tmp/
6. logging.file = /var/tmp/mylog.log
7. logging.level.root = WARN
8. management.security.port=9211 port where actuator endpoint accessed
9. management.security.enabled =false disabling security to actuator endpoints
10. can refer the link for more properties https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html

## 4.8 Spring Boot Annotations

Spring Boot basic annotations includes

- @Bean - indicates that a method produces a bean to be managed by Spring.

- @Service - indicates that an annotated class is a service class.
- @Repository - indicates that an annotated class is a repository, which is an abstraction of data access and storage.
- @Configuration - indicates that a class is a configuration class that may contain bean definitions.
- @Controller - marks the class as web controller, capable of handling the requests.
- @RequestMapping - maps HTTP request with a path to a controller method.
- @Autowired - marks a constructor, field, or setter method to be autowired by Spring dependency injection.
- @SpringBootApplication - enables Spring Boot autoconfiguration and component scanning.
- @ComponentScan –Scans the packages for bean auto configuration.
- @RestController - marks the class as REST controller, capable of handling the REST API requests.
- @PostMapping – It is equal to @RequestMapping annotation with POST request
- @GetMapping – It is equal to @RequestMapping annotation with GET request
- @PutMapping –It is equal to @RequestMapping annotation with PUT Request
- @DeleteMapping - It is equal to @RequestMapping annotation with DELETE Request

@Component is a generic stereotype for a Spring managed component. It turns the class into a Spring bean at the auto-scan time. Classes decorated with this annotation are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases.

There are also Hibernate @Entity, @Table, @Id, and @GeneratedValue annotations.

## 4.9 Spring Boot Actuator

Spring boot actuators are in-built HTTP endpoints available for any boot application for different **monitoring and management purposes**. Before spring framework, if we had to introduce this type of monitoring functionality in our applications then we had to manually develop all those components and that too were very specific to our need. But with spring boot we have Actuator module which makes it very easy.

Spring boot's module Actuator allows you to monitor and manage application usages in production environment, without coding and configuration for any of them. These monitoring and management information is exposed via REST like endpoint URLs.

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

Some of important and widely used actuator endpoints are given below:

**/env -** Returns list of properties in current environment

**/health -** Returns application health information.

**/auditevents -** Returns all auto-configuration candidates and the reason why they 'were' or 'were not' applied.

**/beans -** Returns a complete list of all the Spring beans in your application.

**/trace -** Returns trace logs (by default the last 100 HTTP requests).

**/dump -** It performs a thread dump.

**/metrics -** It shows several useful metrics information like JVM memory used, system CPU usage, open files, and much more.

In application.properties specificying key value pair as shown below will expose the actuator endpoint with given value.

```
management.context-path=/manage
```

**example :** Now you will be able to access all actuator endpoints under new URL.

- /manage/health
- /manage/dump
- /manage/env
- /manage/beans

For including the endpoints for exposure we can use include and exclude properties as shown below.

- management.endpoints.web.exposure.include=*
- management.endpoints.web.exposure.exclude=env,beans

For more reference on actuators

https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html

## 4.9 Spring Boot with MVC Flow

Download spring boot starter project and add controller

```
@Controller
    public class WebController {
        @RequestMapping(value="/loginform" )
        public String welcomeLogin() {
                return "login";
        }


        @RequestMapping(value = "/login", method = RequestMethod.POST)
public ModelAndView submit(@ModelAttribute("LoginForm")
LoginFormloginForm,HttpServletRequest request) {
ModelAndView mv=new ModelAndView();
                boolean status=false;
                if(loginForm.getPassword().equals("123")) {
                        status=true;
                                                        }
mv.addObject("status", ((status==true)? "Success" : "Failed"  ));
mv.addObject("username", loginForm.getUsername());
                mv.setViewName("LoginStatus");
                return mv;

                                        }

    @RequestMapping(value="/logout",method = RequestMethod.GET)
                public String logOut(HttpSession session,HttpServletResponse
                response,HttpServletRequest request) throws IOException{
                 return "redirect:/loginform";
                }
```

**In application.properties**

spring.mvc.view.prefix=/

spring.mvc.view.suffix=.jsp

In folder src/main/webapp add jsp pages

login.jsp

```
<!DOCTYPE html>
<html>
```

```
<style>
input[type=text], select {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  border-radius: 4px;
  box-sizing: border-box;
}

input[type=submit] {
  width: 100%;
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

input[type=submit]:hover {
  background-color: #45a049;
}

div {
  border-radius: 5px;
  background-color: #f2f2f2;
  padding: 20px;
}
</style>
<body>

<h3>Login Form Demo</h3>

<div>
  <form modelAttribute="loginForm" method="POST" action="/login" >
    <label for="fname">User Name</label>
    <input type="text" id="uname" name="username" placeholder="Your name..">

    <label for="lname">Password</label>
    <input type="text" id="pwd" name="password" placeholder="Your Password..">
```

```
   <input type="submit" value="Submit">
  </form>
</div>

</body>
</html>
```

loginStatus.jsp

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

<html>
<style>
input[type=text], select {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  border-radius: 4px;
  box-sizing: border-box;
}

input[type=submit] {
  width: 100%;
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

input[type=submit]:hover {
  background-color: #45a049;
}

div {
  border-radius: 5px;
  background-color: #f2f2f2;
  padding: 20px;
```

```
}



.divred{
  border-radius: 5px;
  background-color: red;
  padding: 20px;
}
</style>
<body>


<div>
<h3>
    <c:if test = "${status == 'Success' }">
                <div>
                ${username}, ur Login is ${status}
                </div>
    </c:if>

    <c:if test = "${status != 'Success' }">
                <div class="divred">
                ${username}, ur Login is ${status}
                </div>
    </c:if>
<a href="/logout"> logout</a>



</h3>

</div>

</body>
</html>
```

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.2.6.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>demoone</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>demoone</name>
        <description>Demo project for Spring Boot</description>

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

        <dependencies>

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

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-devtools</artifactId>
                        <scope>runtime</scope>
                        <optional>true</optional>
                </dependency>

        <dependency>
                <groupId>org.apache.tomcat</groupId>
                <artifactId>tomcat-jasper</artifactId>
                <version>9.0.33</version>
        </dependency>
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>jstl</artifactId>
        </dependency>
        <dependency>
```

```
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
   <!-- <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
        </dependency> -->
 <!-- <dependency>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-jaxrs</artifactId>
  <version>1.5.0</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.0.1</version>
  </dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.0.1</version>
</dependency>  -->
                <dependency>
                        <groupId>com.h2database</groupId>
                        <artifactId>h2</artifactId>
                        <scope>runtime</scope>
                </dependency>
                        <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>

                <!-- <dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
 -->


                <!-- <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
```

```
                <exclusions>
                    <exclusion>
                        <groupId>org.junit.vintage</groupId>
                        <artifactId>junit-vintage-engine</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
        --></dependencies>

        <build>
            <plugins>
                    <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <!-- <version>3.8.0</version> -->
                    <configuration>
                        <source>${java.version}</source>
                        <target>${java.version}</target>
                        <compilerVersion>${java.version}</compilerVersion>
                        <verbose>true</verbose>
                        <fork>true</fork>
                        <executable>C:\Program
Files\Java\jdk1.8.0_191\bin\javac</executable>
                        <meminitial>${initial.memory}</meminitial>
                        <maxmem>${maximum.memory}</maxmem>

        <useIncrementalCompilation>true</useIncrementalCompilation>
                    </configuration>
                </plugin>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                    <configuration>

        <mainClass>com.example.demoone.DemooneApplication</mainClass>
                        <executable>true</executable>
                    </configuration>
                </plugin>
                </plugins>
        </build>

</project>
```

## 4.10 Spring Boot Dev Tools Dependency

Dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

The spring-boot-devtools module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. Precondition is that your browser should have supported extention for it. You can find such browser extentions in this link

By default, live reload is enabled. If you wish to disable this feature for some reason, then set spring.devtools.livereload.enabled property to false.

If you want to disable auto-reload in browser for files in few of these paths, then use spring.devtools.restart.exclude property

There may be few files not in classpath, but you still may want to watch those addtional files/paths to reload the application. To do so, use the spring.devtools.restart.additional-paths property.

Auto-restart means reloading the java classes and configuration at server side. After the server side changes are re-deployed dynamically, server restart happen and load the modified code and configuration.

To disable the logging of the report, set the following property:

spring.devtools.restart.log-condition-evaluation-delta = false

Automatic restarts may not be desirable on every file change and sometimes can slower down development time due to frequent restarts. To solve this problem, you can use a trigger file. Spring boot will keep monitoring that file and once it will detect any modification in that file, it will restart the server and reload all your previous changes.

spring.devtools.restart.trigger-file = c:/workspace/restart-trigger.txt

## 4.11 Spring Boot Schedulers

Scheduling is a process of executing the tasks for the specific time period. Spring Boot provides a good support to write a scheduler on the Spring applications.

The @EnableScheduling annotation is used to enable the scheduler for your application. This annotation should be added into the main Spring Boot application class file.

The @Scheduled annotation is used to trigger the scheduler for a specific time period.

```
@Scheduled(cron = "0 * 9 * * ?")
public void cronJobSch() throws Exception {
}
```

**Fixed Rate**

Fixed Rate scheduler is used to execute the tasks at the specific time. It does not wait for the completion of previous task. The values should be in milliseconds. The sample code is shown here –

```
@Scheduled(fixedRate = 1000)
public void fixedRateSch() {
}
```

**Fixed Delay**

Fixed Delay scheduler is used to execute the tasks at a specific time. It should wait for the previous task completion. The values should be in milliseconds. A sample code is shown here –

```
@Scheduled(fixedDelay = 1000, initialDelay = 1000)
public void fixedDelaySch() {
}
```

Dynamic scheduling can be done by implementing interface SchedulingConfigurer

Sample code for Dynamic Scheduler that runs twice in a day in given interval

```
@Override
public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.addTriggerTask(new Runnable() {
        @Override
        public void run() {
                System.out.println("Running Schedular JobLogic....."+new Date());
        }
    },  new Trigger() {

                        @Override
                        public Date nextExecutionTime(TriggerContext triggerContext) {
            Calendar nextExecutionTime = new GregorianCalendar();
            System.out.println("let::"+triggerContext.lastActualExecutionTime());
            Date lastActualExecutionTime = triggerContext.lastActualExecutionTime();
            boolean lastExDtTom=false;
            if(lastActualExecutionTime!=null ) {
                    lastExDtTom=lastActualExecutionTime.before(new Date());
```

```
                System.out.println("lastExDtTom:"+lastExDtTom);


            }
            if(count==1 && lastActualExecutionTime==null) {
                lastExDtTom=true;
                System.out.println("count = 1");
            }
            if(!lastExDtTom && count>2) {
                System.out.println("count > 2");
                lastExDtTom=false;

            }
            else {
                System.out.println("count = 2");
                lastExDtTom=true;
            }
            if(lastExDtTom ) {
                        nextExecutionTime.setTime(lastActualExecutionTime != null ?
                        lastActualExecutionTime : new Date());
                            nextExecutionTime.add(Calendar.MILLISECOND,getNewExecuti
                            onTime());
            count++;
            System.out.println("count:"+count);
            System.out.println("nET::"+nextExecutionTime.getTime());
            return nextExecutionTime.getTime();
        }
            else {
                        System.out.println("null---");
                return null;


                }}

        });
}
//running job dynmically twice per day in 20sec interval
private int getNewExecutionTime() {
        //Load Your execution time from database or property file
        System.out.println("nextExecutionTime:"+nextExecutionTime);
        return 20000;
}
```

## 5. Limitations of Spring Boot

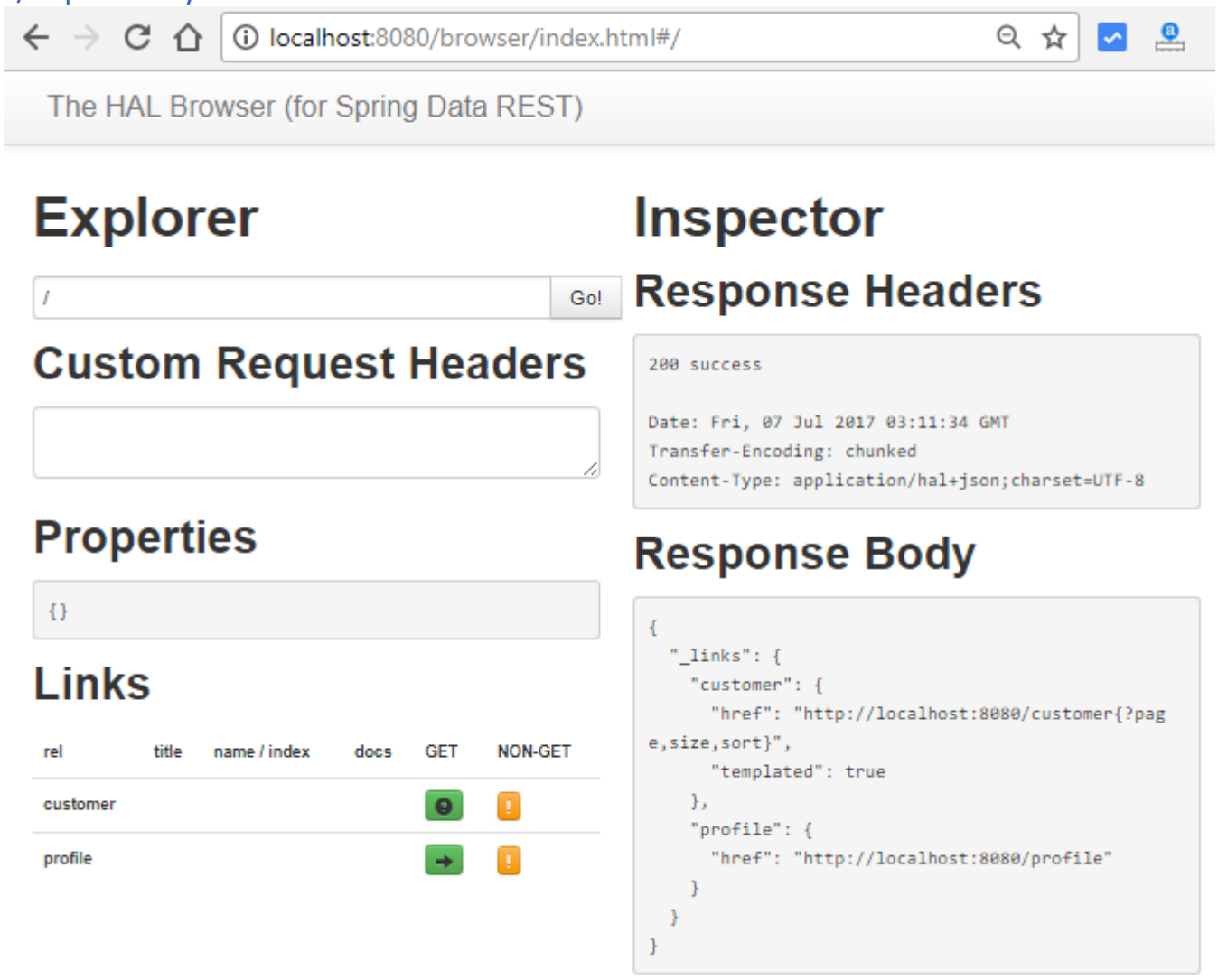
Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application. It needs lots of time and efforts to turn legacy/ existing spring application into spring boot application but new spring boot projects can be created easily.

## 6. HAL Browser

JSON Hypertext Application Language, or HAL, is a simple format that **gives a consistent and easy way to hyperlink between resources in our API**. Including HAL within our REST API makes it much more explorable to users as well as being essentially self-documenting.

HAL Dependency :

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```



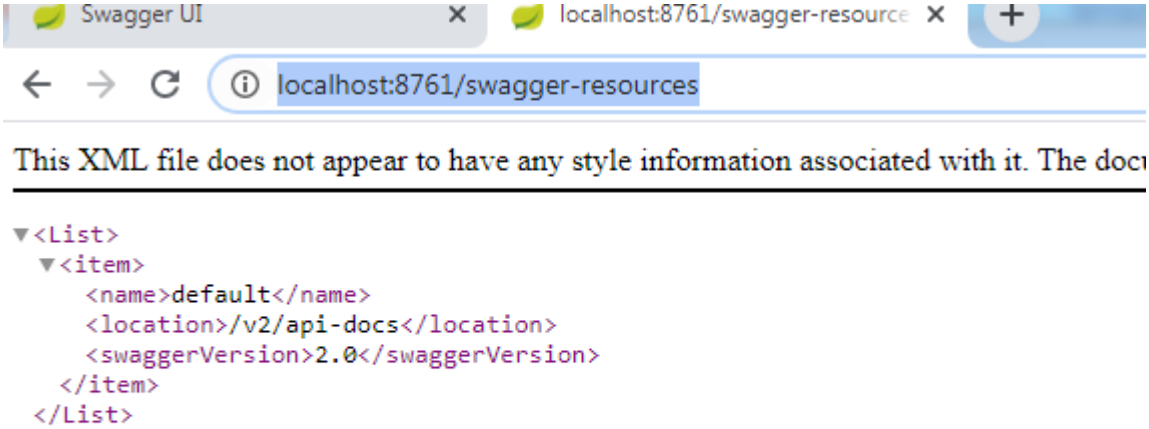
## 7. Swagger

Swagger is widely used for visualizing APIs, and with Swagger UI it provides online sandbox for frontend developers. For the tutorial, we will use the Springfox implementation of the Swagger 2 specification. Swagger is a tool, a specification and a complete framework implementation for producing the visual representation of RESTful Web Services. It enables documentation to be updated at the same pace as the server. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Thus Swagger removes the guesswork in calling the service.

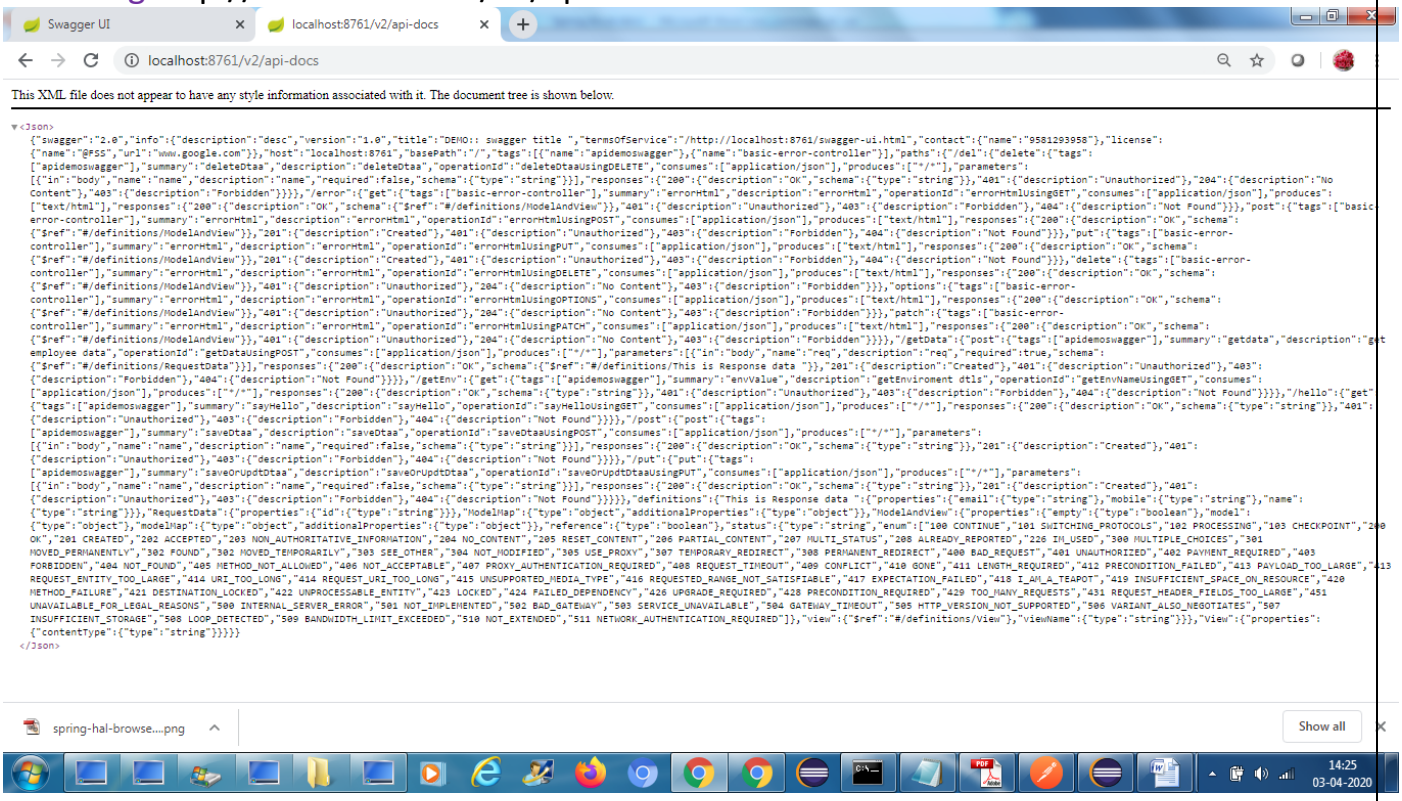
```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.0.1</version>
  </dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.0.1</version>
```

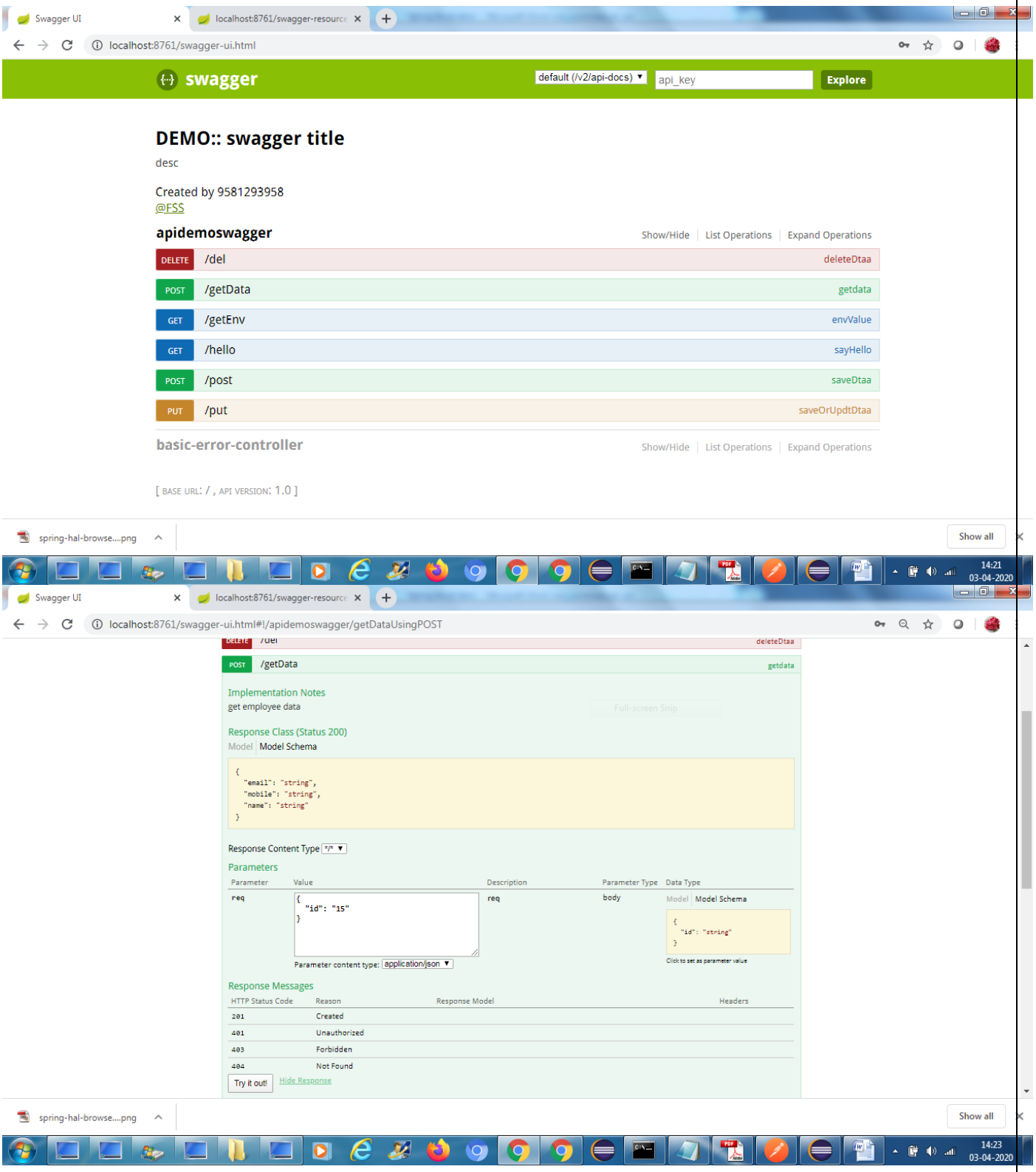
To enable the Swagger 2 we use the annotation @EnableSwagger2. A Docket bean is defined and using its select() method we get an instance of ApiSelectorBuilder. ApiSelectorBuilder we configure the endpoints exposed by Swagger.
After the Docket bean is defined, its select() method returns an instance of ApiSelectorBuilder, which provides a way to control the endpoints exposed by Swagger. Using the RequestHandlerSelectors and PathSelectors we configure the predicates for selection of RequestHandlers.

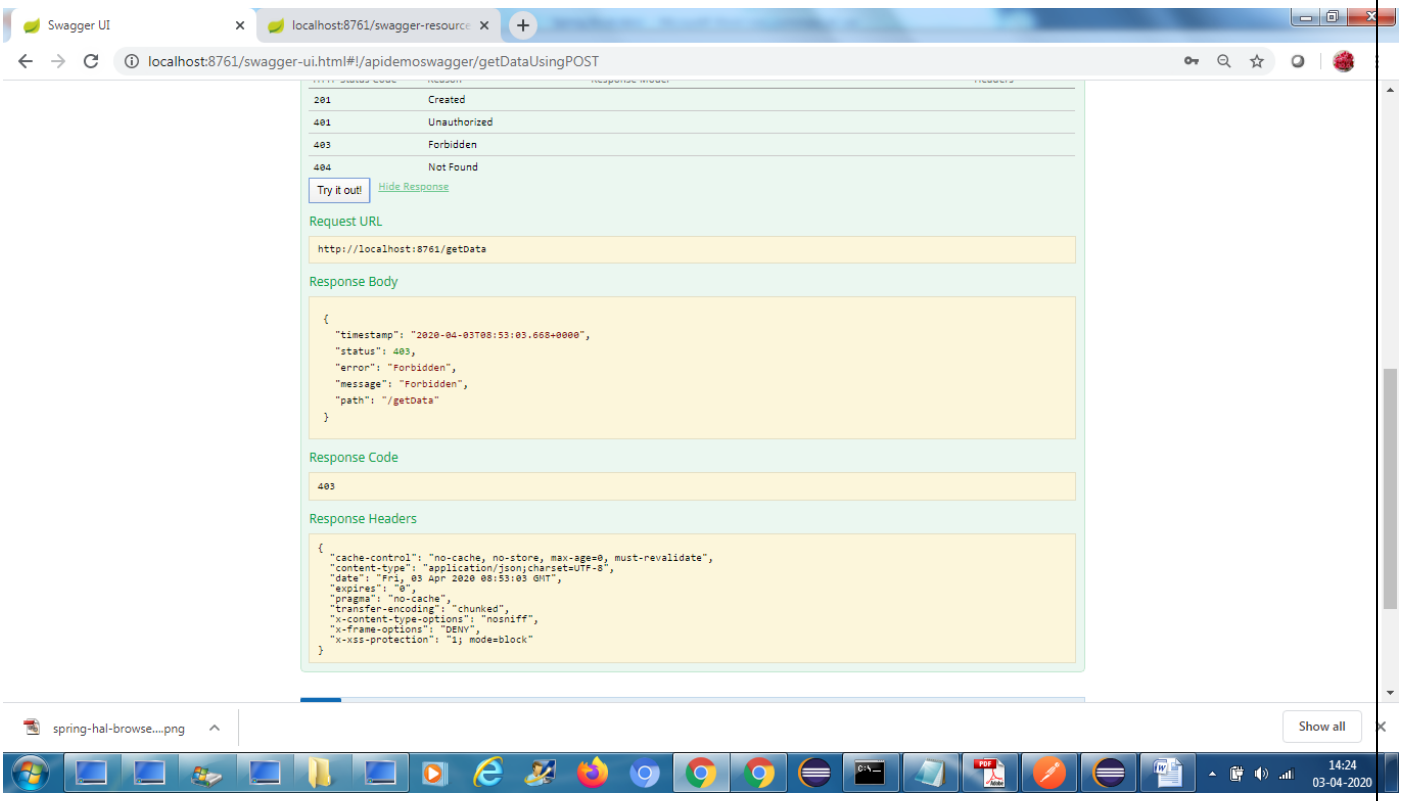
To Access Resources : http://localhost:8761/swagger-resources




Accessing http://localhost:8761/v2/api-docs




Swagger UI
localhost:8761/swagger-ui.html
swagger
default (/v2/api-docs)
api_key
Explore
DEMO:: swagger title
desc
Created by 9581293958
@FSS
apidemoswagger
Show/Hide | List Operations | Expand Operations
DELETE /del deleteDtaa
POST /getData getdata
GET /getEnv envValue
GET /hello sayHello
POST /post saveDtaa
PUT /put saveOrUpdtDtaa
basic-error-controller
Show/Hide | List Operations | Expand Operations
[ BASE URL: / , API VERSION: 1.0 ]
spring-hal-browse....png
Show all
14:21
03-04-2020
Swagger UI
localhost:8761/swagger-ui.html#!/apidemoswagger/getDataUsingPOST
POST /getData
getdata
Implementation Notes
get employee data
Response Class (Status 200)
Model Schema
{
"email": "string",
"mobile": "string",
"name": "string"
}
Response Content Type */*
Parameters
Parameter Value Description Parameter Type Data Type
req
{
"id": "15"
}
req
body
Model Schema
{
"id": "string"
}
Parameter content type: application/json
Response Messages
HTTP Status Code Reason Response Model Headers
201 Created
401 Unauthorized
403 Forbidden
404 Not Found
Try it out!
Hide Response
spring-hal-browse....png
Show all
14:23
03-04-2020




**Swagger Annotations :**

**@Api – decorated on top of controller to describe about the service**

```
@RestController @RequestMapping("/product") @Api(value="onlinestore",
description="Operations pertaining to products in Online Store") public class
ProductController { . . . . }
```

**@ApiOperation – used to describe API Endpoint and it's response type**

```
@ApiOperation(value = "View a list of available products", response = Iterable.class)
@RequestMapping(value = "/list", method= RequestMethod.GET,produces =
"application/json")
public Iterable list(Model model){..}
```

**@ApiResponse**

Swagger 2 also allows overriding the default response messages of HTTP methods. You can use the @ApiResponse annotation to document other responses, in addition to the regular HTTP 200 OK, like this.

```
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "Successfully retrieved list"),
    @ApiResponse(code = 401, message = "You are not authorized to view the resource"),
    @ApiResponse(code = 403, message = "Accessing the resource you were trying to
reach is forbidden"),
    @ApiResponse(code = 404, message = "The resource you were trying to reach is not
found")
```

```
}
```

**@ApiModelProperty**

ApiModelProperty annotation to describe the properties of the Product model.
With @ApiModelProperty, you can also document a property as required.

```
public class Product {

   @ApiModelProperty(notes = "The database generated product ID")
   private Integer id;
   @Version
   @ApiModelProperty(notes = "The auto-generated version of the product")
   private Integer version;
}
```

## 8. H2 Data Base

H2 is one of the popular in-memory databases written in Java. It can be embedded in Java applications or run in the client-server mode.

- Spring Boot provides excellent integration support for H2 using simple properties configuration.
- To make it even more useful, H2 also provides a console view to maintain and interact with the database tables and data.

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

By default, the console view of H2 database is disabled. We must enable it to view and access it in browser. Note that we can customize the URL of H2 console which, by default, is '/h2'.

Application.properties

```
spring.datasource.url=jdbc:h2:mem:dbname
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

spring.h2.console.enabled=true

The pojo must contain annotation @entity and @table annotations on class to map with tables in H2 tables and @column to mark columns in table and @id indicates it as key for the table.

Create interface that extends crudRepository as shown below

```
@Component
public interface StudentRepository  extends CrudRepository<Student, Integer>{
}
```

In Service
Returns all students in student table
studentRepository.findAll().forEach(student -> students.add(student));
Return student detail by id from student table
studentRepository.findById(id).get();
Save student object
studentRepository.save(student);
delete student by id from student table
studentRepository.deleteById(id);

//In this way we can work with h2 database.

## 9. Basic Security Authentication

In spring boot basic security authentication can be done just by adding spring-start-security dependency in pom.xml. this dependency will add security to each and every endpoint of the application where the user who is accessing the api should use basic auth to access the URL and by default user will be the username and password will be unique auto generated value in console during app-start-up. In case if password and username has to be changed, then application.properties file can be used to mention the username and password as shown below.

```
spring.security.user.name=user
spring.security.user.password=password@123
```