

# Spring Boot

# Spring Boot

Spring Boot is primarily considered as a framework from “The Spring Team” that eases the bootstrapping and the development of new Spring applications.

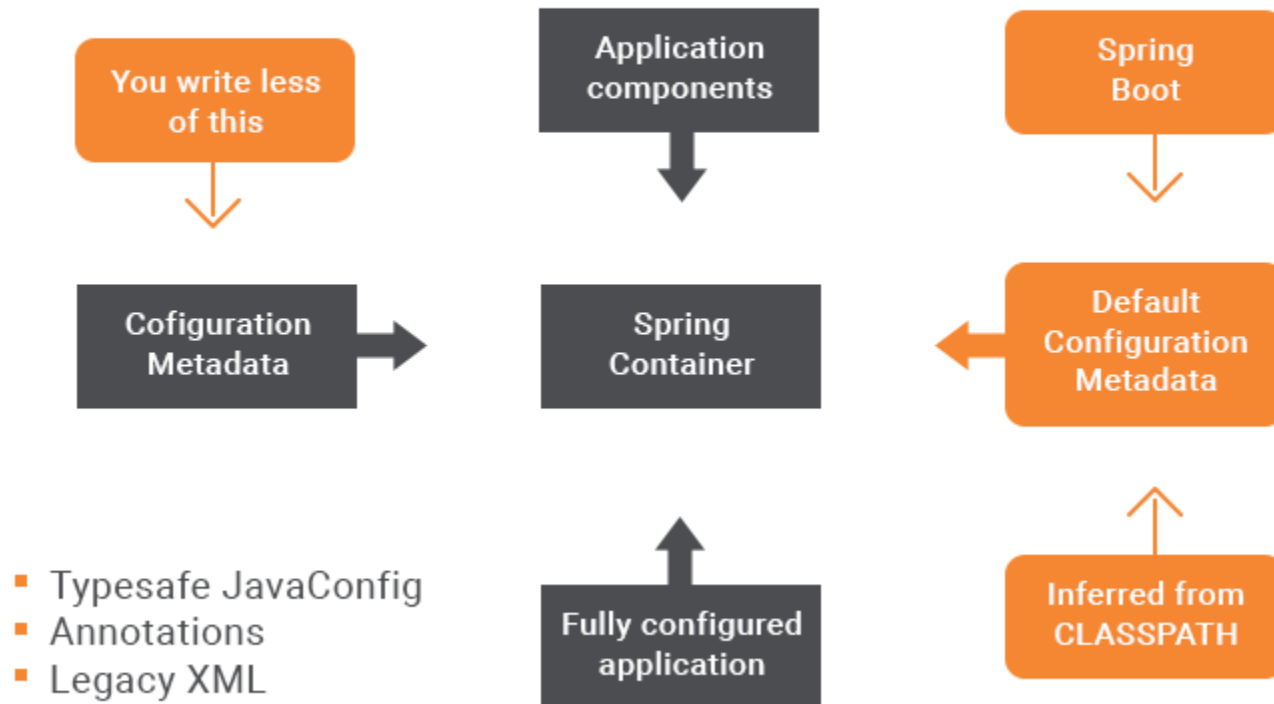
Spring Boot results into building production-ready Spring applications with feasible convention over configuration mechanism that allows to pre-configure the applications in a more opinionated way.

# Benefits of Spring Boot

- It makes it easier to develop Spring-based applications with Java or Groovy. Also, it reduces Developer's effort with the "Opinionated Defaults Configuration" approach.
- It minimizes writing multiple boilerplate codes, XML configuration and annotation, ultimately enhancing productivity while reducing lots of development time.
- It makes it easier to integrate the Spring Boot Application with the Spring Ecosystem that majorly includes Spring ORM, Spring JDBC, Spring Security, Spring Data and many other things.
- It tests the web applications easily with the help of different Embedded HTTP servers that includes Tomcat, Jetty and many others.
- It offers Command Line Interface (CLI) tool for developing and testing Spring Boot.
- It offers a number of plugins for developing and testing Spring Boot Applications easily using Maven/Gradle- the build tools.
- It offers a number of plugins for working with embedded and in-memory databases easily.

# Spring Boot Simplifies Configuration

## Spring Boot Simplifies Configuration



# Why to opt Spring Boot for building web applications

**Reason #1:** Spring Boot offers simpler dependency management to the starter projects as compared to CRUD web application.

**Reason #2:** It offers quick standalone mode along with auto configuration

**Reason #3:** It offers a highly opinionated as well as well tested set of dependencies that works great with Spring ecosystem for various projects.

# Creating Spring Boot Projects With Eclipse and Maven

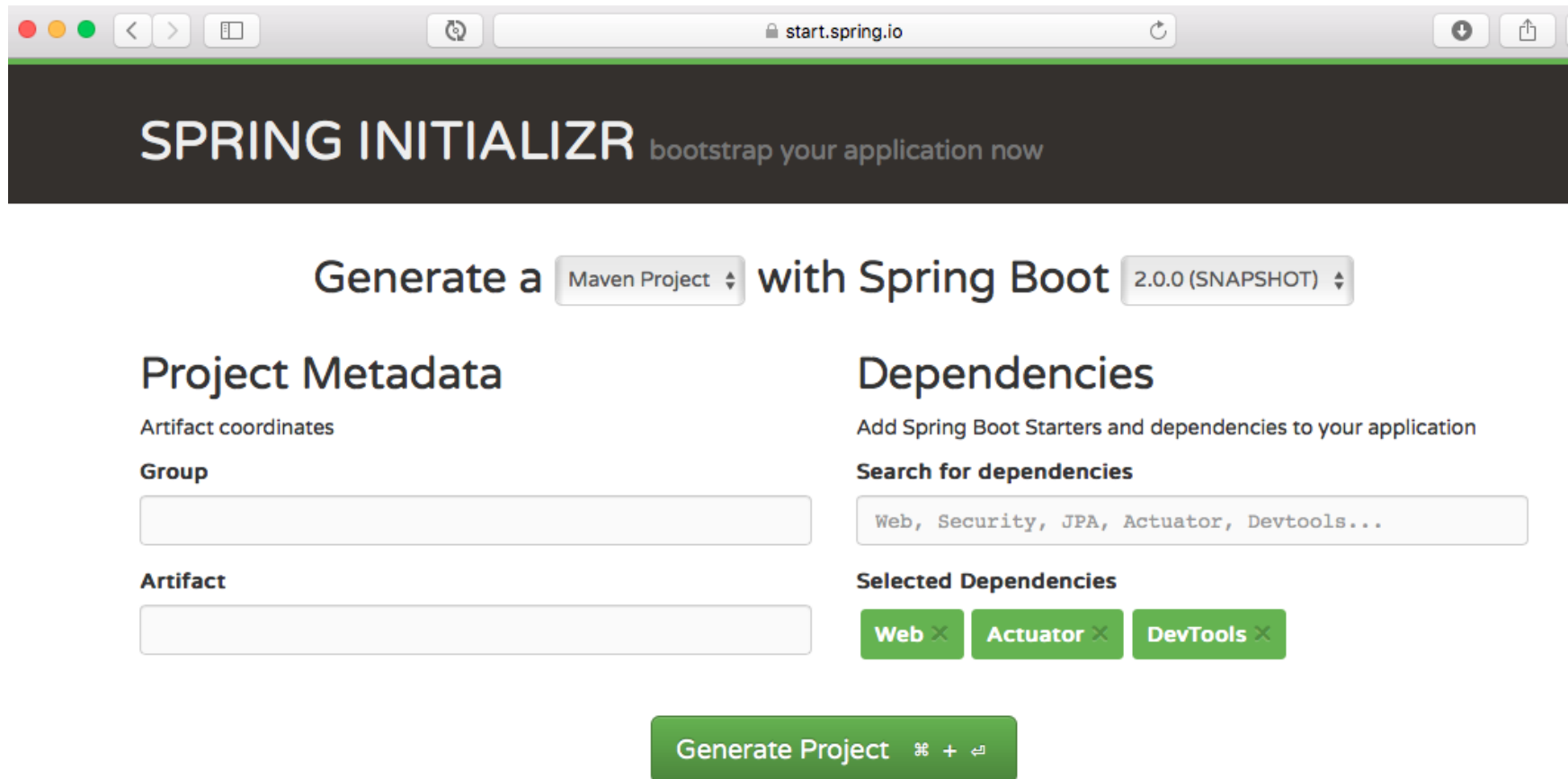
There are three options to create Spring Boot Projects with Eclipse and Maven:

- [Spring Initializr](#)
- Use STS or STS Eclipse Plugin and create a Spring Boot Maven project directly from Eclipse
- Manually create a Maven project and add Spring Boot starter dependencies

# Bootstrap Spring Boot Project With Spring Initializr

[Spring Initializr](https://start.spring.io/) is a easiest way to bootstrap your Spring Boot projects.

1. Go to <http://start.spring.io/>



The screenshot shows the Spring Initializr web application in a browser window. The browser's address bar displays 'start.spring.io'. The page has a dark header with the text 'SPRING INITIALIZR bootstrap your application now'. Below the header, there is a form to generate a project. The form is divided into two main sections: 'Project Metadata' and 'Dependencies'. In the 'Project Metadata' section, there are two input fields: 'Group' and 'Artifact'. In the 'Dependencies' section, there is a search bar with the text 'Web, Security, JPA, Actuator, Devtools...' and a list of 'Selected Dependencies' which includes 'Web', 'Actuator', and 'DevTools'. At the bottom of the form is a large green button labeled 'Generate Project'.

Generate a Maven Project with Spring Boot 2.0.0 (SNAPSHOT)

**Project Metadata**  
Artifact coordinates

**Group**

**Artifact**

**Dependencies**  
Add Spring Boot Starters and dependencies to your application

**Search for dependencies**

**Selected Dependencies**  
Web Actuator DevTools

**Generate Project**

# Bootstrap Spring Boot Project With Spring Initializr

*As shown in the image, following steps have to be done:*

- Launch Spring Initializr and choose the following
  - Choose **com.vu** as **Group**
  - Choose **SpringBootWebApp** as **Artifact**
  - Choose from the following dependencies:
    - Web
    - Actuator
    - DevTools
- Click **Generate Project**.

This would download a ZIP file to your local machine.

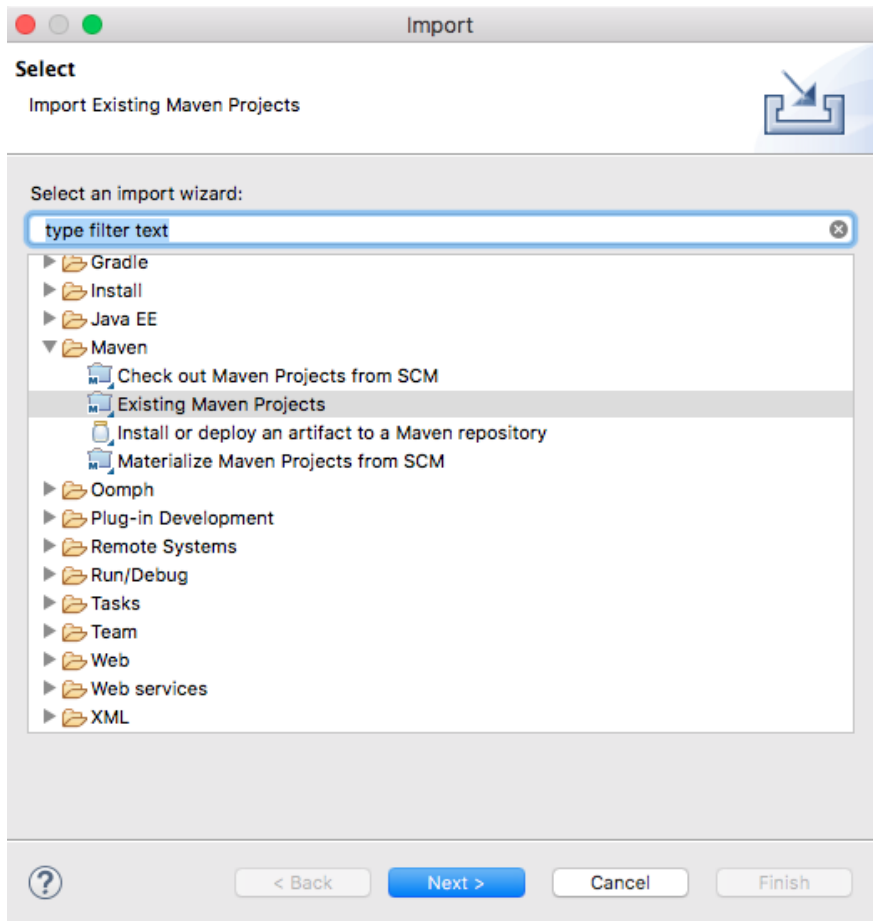
Unzip the zip file and extract to a folder.

In Eclipse, Click **File > Import > Existing Maven Project**

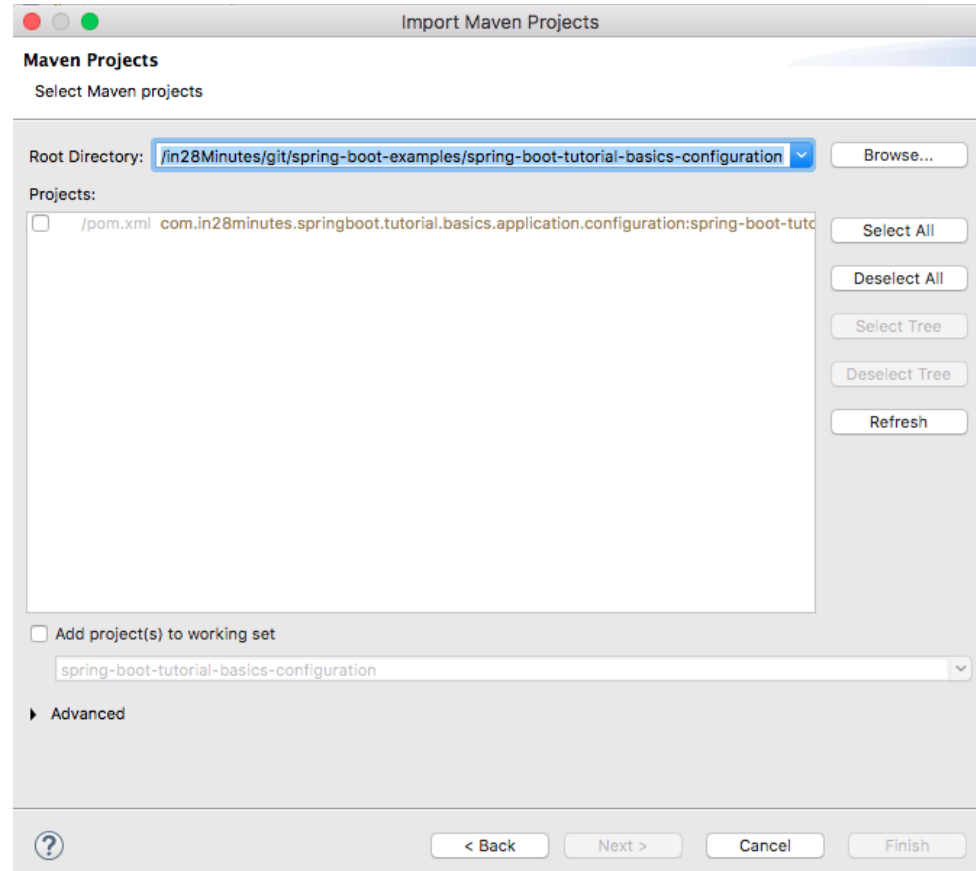


# Building Spring Boot App

In Eclipse, Click **File > Import > Existing Maven Project** as shown below.



Navigate or type in the path of the folder where you extracted the ZIP file on the next screen.



Once you click **Finish**, Maven will take some time to download all the dependencies and initialize the project.

# Spring Boot Starter Web Dependencies

Dependencies can be classified into:

- Spring - core, beans, context, app
- Web MVC - (Spring MVC)
- Jackson - for JSON Binding
- Validation - Hibernate Validator, Validation API
- Embedded Servlet Container - Tomcat
- Logging - logback, slf4j

# Building Spring Boot Web Application

Spring Boot Starter Web provides all the dependencies and the auto configuration needed to develop web applications. We should use the first dependency.

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

We want to use JSP as the view. Default embedded servlet container for Spring Boot Starter Web is tomcat. To enable support for JSP's, we would need to add a dependency on tomcat-embed-jasper.

```
<dependency>  
  
<groupId>org.apache.tomcat.embed</groupId>  
  
<artifactId>tomcat-embed-jasper</artifactId>  
  
<scope>provided</scope>  
  
</dependency>
```

# Building Spring Boot Web Application

Go to project properties, Right-Click on context-root->properties

Properties-> project facets.

Select Web module 3.1 and configure.

Note: Provide the name of the folder as **webapp** against the default name, **WebContent** folder. and WEB-INF folder in it. Spring boot follows convention over configuration.

Spring Boot Starter Web auto configures the basic things we needed to get started.

To understand the features Spring Boot Starter Web brings in, *Run Application.java* as a *Java Application* and review the log.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

# @SpringBootApplication

```
@SpringBootApplication
```

```
public class Application {  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
}
```

@SpringBootApplication which is a combination of the following more specific spring annotations -

- [@Configuration](#) : Any class annotated with @Configuration annotation is bootstrapped by Spring and is also considered as a source of other bean definitions.
- [@EnableAutoConfiguration](#) : This annotation tells Spring to automatically configure your application based on the dependencies that you have added in the pom.xml file.  
For example, If spring-data-jpa or spring-jdbc is in the classpath, then it automatically tries to configure a DataSource by reading the database properties from ***application.properties*** file.
- [@ComponentScan](#) : It tells Spring to scan and bootstrap other components defined in the current package (ex. *com.vus*) and all the sub-packages, **but have to specify the package name.**

Note: The `main()` method calls Spring Boot's `SpringApplication.run()` method to launch the application.

# resources folder

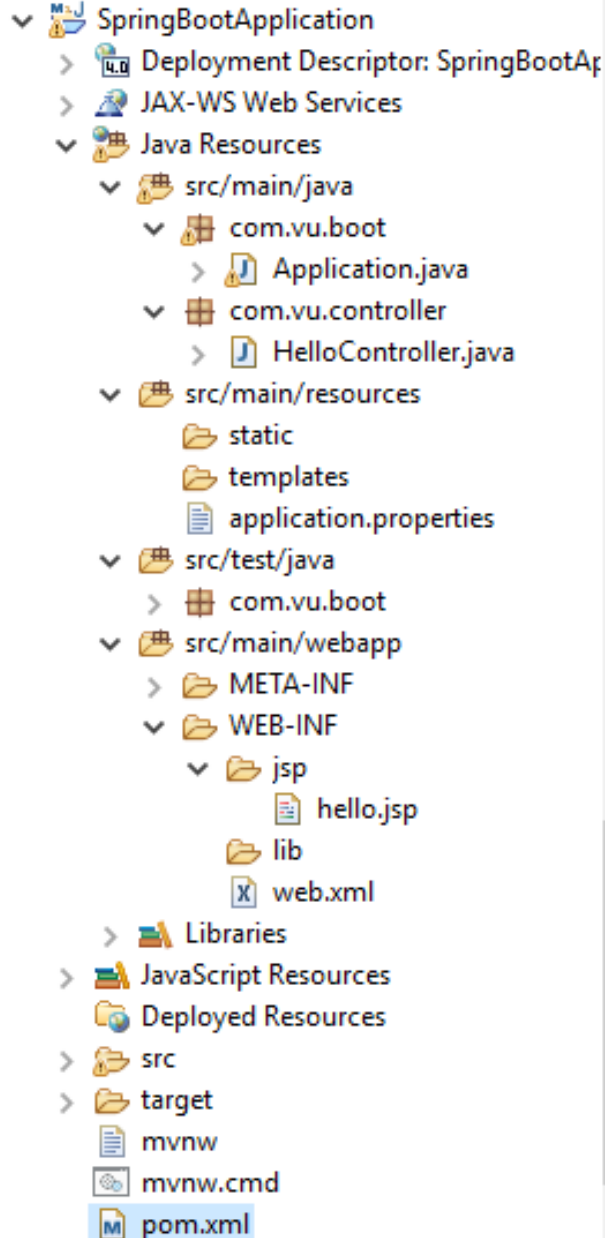
This directory, as the name suggests, is dedicated to all the static resources, templates and property files.

- **resources/static** - contains static resources such as css, js and images.
- **resources/templates** - contains server-side templates which are rendered by Spring.
- **resources/application.properties** - This file is very important as it contains application-wide properties.

Spring reads the properties defined in this file to configure your application. You can define server's default port, server's context path, database URLs etc, in this file.

- Sample Spring Boot Application
- Sample Spring Boot Web Application

# Sample Spring Boot Application



1. Go to <http://start.spring.io/> . Select required dependencies (**Web, DevTools, Actuator**) and generate project.
2. A zip file will be saved into your local directory.
3. Unzip.
4. In Eclipse, File->Import->New->Existing Maven Projects.

***Note : webapp folder not required for this application***



# Sample Spring Boot Application

pom.xml

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-
parent</artifactId>
<version>2.0.3.RELEASE</version>
<relativePath/> <!-- lookup parent from repository
-->
</parent>
<properties>
<project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
<java.version>1.8</java.version>
</properties>

<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<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>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>

</dependencies>
```

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
```

# Sample Spring Boot Application

Application.java

```
@SpringBootApplication(scanBasePackages="com.varaunited")
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

```
@RestController
public class HelloController {
    @Value("${application.message:Hello World}")
    private String helloMessage;

    @RequestMapping("/")
    public String hello() {
        return "Hello! Welcome To Spring Boot";
        //return helloMessage;
    }
}
```

HelloController.java

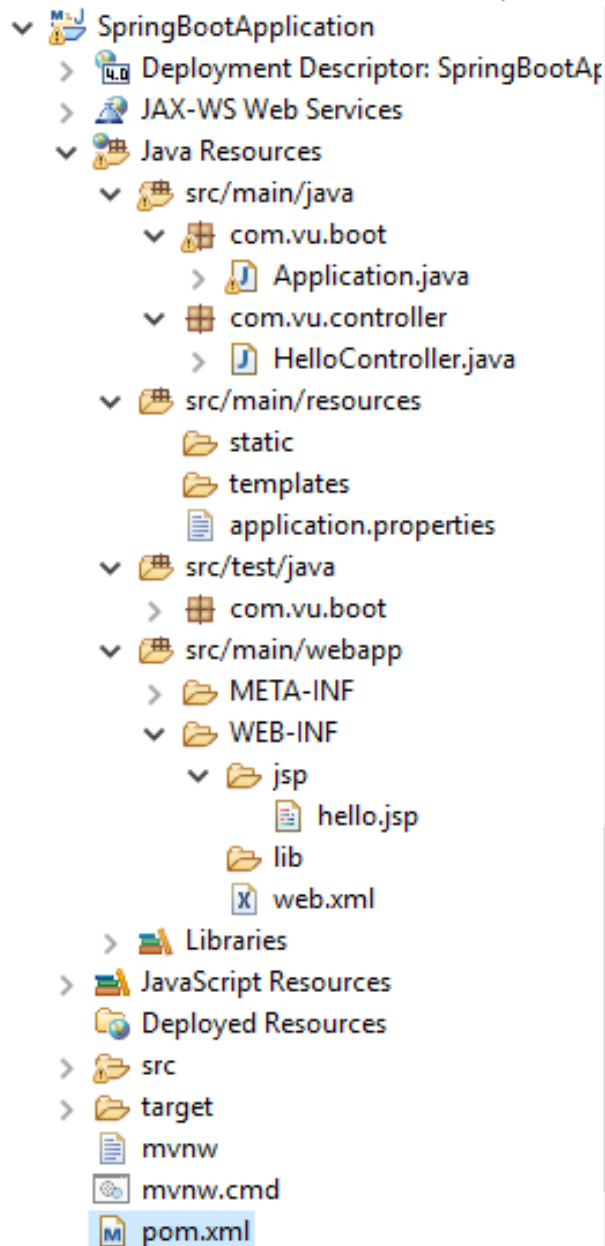
application.properties

application.message: Hello World!!

1. Run Application.java, which will launch tomcat and deploy our application on it.
2. Run <http://localhost:8080/> on browser.

**Note:** ContextRoot (project name) not required since only one project is deployed in the container

# Sample Spring Boot Web Application



1. Go to <http://start.spring.io/> . Select required dependencies (**Web, DevTools, Actuator**) and generate project.
2. A zip file will be saved into your local directory.
3. Unzip.
4. In Eclipse, File->Import->New->Existing Maven Projects.
5. Right-click on project->properties->project facets->web module->3.1(or above)
6. Re-name WebContent folder as webapp. Following is the project structure.
7. **Drag webapp folder into src/main folder.**
8. Update pom.xml, include following additional dependencies.
  1. tomcat-embed-jasper
  2. Jstl

Note: **webapp** folder should be in **src/main** source folder

# Sample Spring Boot Application

pom.xml

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-
parent</artifactId>
<version>2.0.3.RELEASE</version>
<relativePath/> <!-- lookup parent from repository
-->
</parent>
<properties>
<project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
<java.version>1.8</java.version>
</properties>

<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<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>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
<!--
https://mvnrepository.com/artifact/org.apache.tomcat.e
mbed/tomcat-embed-jasper -->
<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>
</dependencies>
```

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
```

# Sample Spring Boot Web Application

@Controller

```
public class HelloController {  
    @Value("${application.message:Hello World}")  
    private String helloMessage;
```

HelloController.java

Note: Need not update web.xml

```
@RequestMapping("/")  
public String hello(Model model) {  
    model.addAttribute("message", helloMessage);  
    return "hello";  
}  
}
```

application.properties

spring.mvc.view.prefix=/WEB-INF/jsp/  
spring.mvc.view.suffix=.jsp

application.message: Hello World!!

hello.jsp

```
...  
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
...  
</head>  
<body>  
<h1><font color="green"> ${message}</font></h1>  
..
```

1. Run Application.java, which will launch tomcat and deploy our application on it.
2. Run <http://localhost:8080/> on browser.

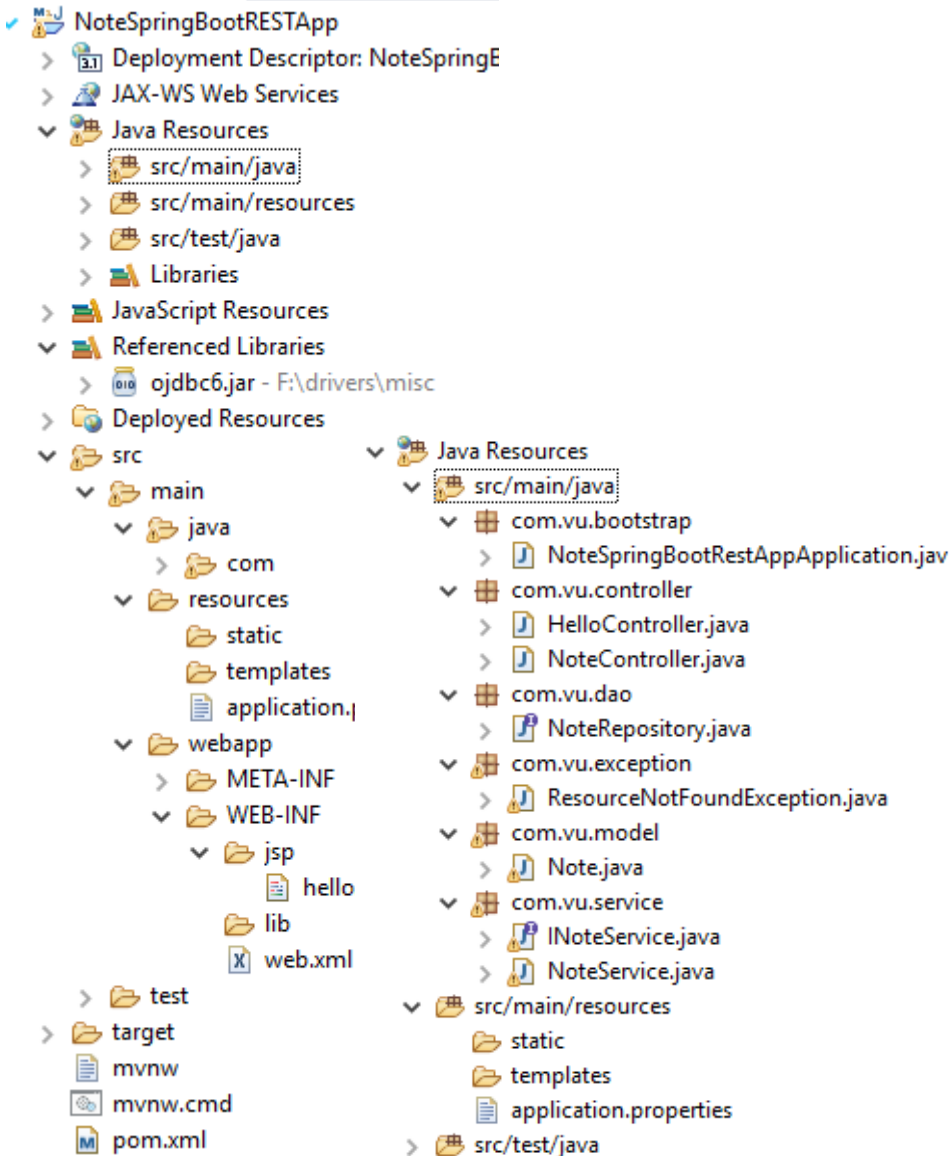
Note: Spring boot will implicitly pickup the jsp location from *application.properties* file

# Creating RESTful Application with Spring Boot and JPA



NoteSpringBootRESTApp.zip

# NoteSpringBootRESTApp



1. Go to <http://start.spring.io/> . Select required dependencies (**Web, JPA, and DevTools** ) and generate project.
  2. A zip file will be saved into your local directory.
  3. Unzip.
  4. In Eclipse, File->Import->New->Existing Maven Projects.
  5. Right-click on project->properties->project facets->web module->3.1(or above)
  6. While configuring change the folder name WebContent to webapp.
  7. **drag webapp folder into src/main folder**
  8. Update pom.xml, include following additional dependencies.
    1. tomcat-embed-jasper
    2. Jstl
  9. Add ojdbc6.jar as external jar file to projects WEB-INF/lib folder( via. Deployment assembly)
  10. Configure project facets, re-name WebContent folder as webapp
- Note: **webapp** folder should be placed under **src/main** source folder

# Configuring Oracle Database

Spring Boot tries to auto-configure a DataSource if spring-data-jpa is in the classpath by reading the database configuration from application.properties file.

So, we just have to add the configuration and Spring Boot will take care of the rest.

Open **application.properties** file and add the following properties to it.

```
## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url = jdbc:oracle:thin:@//localhost:1521/myoracle
spring.datasource.username = scott
spring.datasource.password = tiger
## Hibernate Properties
# SQL dialect makes Hibernate generate better SQL for chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.Oracle10gDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```



# Model Class : Note

```
@Entity
@Table(name = "notes_vu")
@EntityListeners(AuditingEntityListener.class)
@JsonIgnoreProperties(value = {"createdAt", "updatedAt"}, allowGetters = true)
public class Note implements Serializable {
    @Id
    @SequenceGenerator(name="notes_vu_sg",sequenceName="notes_vu_seq",allocationSize=1)
    @GeneratedValue(generator="notes_vu_sg",strategy=GenerationType.SEQUENCE)
    private Long id;
    @NotBlank
    private String title;
    @NotBlank
    private String content;
    @Column(nullable = false, updatable = false) @Temporal(TemporalType.TIMESTAMP)
    @CreatedDate
    private Date createdAt;
    @Column(nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    @LastModifiedDate
    private Date updatedAt;
    // Getters and Setters ... }
```

# Enable JPA Auditing

In Note model class, we have annotated ***createdAt*** and ***updatedAt*** fields with **@CreatedDate** and **@LastModifiedDate** annotations respectively.

Due to this, the fields automatically get populated whenever we create or update an entity.

To achieve this, we need to do two things –

1. **Add Spring Data JPA's AuditingEntityListener to the domain model(Note class).**

```
@EntityListeners(AuditingEntityListener.class)
public class Note{
}
```

2. **Enable JPA Auditing in the main application.**

Decorate Application.java with **@EnableJpaAuditing** annotation.

# Starter/Bootstrap Application

```
@SpringBootApplication(scanBasePackages="com.vu")
@EnableJpaAuditing
@EnableJpaRepositories(basePackages="com.vu.dao")
@EntityScan("com.vu.model")
public class NoteSpringBootTestAppApplication {

    public static void main(String[] args) {
        SpringApplication.run(NoteSpringBootTestAppApplication.class, args);
    }
}
```

To autowire NoteRepository, decorate starter class with the following annotations along with package names where they are located.

**@EnableJpaRepositories(basePackages="com.vu.dao") and  
@EntityScan("com.vu.model")**

Note: Without this ,autowiring will not be done.

# Creating NoteRepository to access data from the database

Spring boot comes with a [JpaRepository](#) interface which defines methods for all the CRUD operations on the entity, and a default implementation of JpaRepository called [SimpleJpaRepository](#).

```
@Repository
public interface NoteRepository extends JpaRepository<Note, Long>
{

}
```

That is all you have to do in the repository layer. You will now be able to use JpaRepository's methods like **save()**, **findOne()**, **findAll()**, **count()**, **delete()** etc.

You don't need to implement these methods.

They are already implemented by Spring Data JPA's SimpleJpaRepository. This implementation is plugged in by Spring automatically at runtime.

# Creating Custom Business Exception

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    private String resourceName;
    private String fieldName;
    private Object fieldValue;
    public ResourceNotFoundException( String resourceName, String fieldName, Object fieldValue) {
        super(String.format("%s not found with %s : '%s'", resourceName, fieldName, fieldValue));
        this.resourceName = resourceName;
        this.fieldName = fieldName;
        this.fieldValue = fieldValue;
    }
    public String getResourceName() { return resourceName; }
    public String getFieldName() { return fieldName; }
    public Object getFieldValue() { return fieldValue; }
}
```

Notice the use of **@ResponseStatus** annotation in the above exception class. This will cause Spring boot to respond with the specified HTTP status code whenever this exception is thrown from your controller.

# Service Layer

## Create interface and its implementation

```
public interface INoteService {  
    public List<Note> getAllNotes();  
    public Note createNote(Note note);  
    public Note getNoteById(Long noteId);  
    public Note updateNote(Long noteId, Note noteDetails);  
    public ResponseEntity<?> deleteNote(Long noteId);  
}
```

**@Service**

**public class NoteService implements INoteService{**

**@Autowired**

**NoteRepository noteRepository;**

```
public List<Note> getAllNotes() {  
    return noteRepository.findAll();  
}
```

```
public Note createNote(Note note) {  
    return noteRepository.save(note);  
}
```

```
public Note getNoteById(Long noteld) {  
    return noteRepository.findById(noteld).orElseThrow(() -> new  
        ResourceNotFoundException("Note", "id", noteld));  
}
```

# Service Layer Contd.

```
public Note updateNote(Long notelId, Note noteDetails) {  
    Note note = noteRepository.findById(notelId).orElseThrow(() ->  
        new ResourceNotFoundException("Note", "id", notelId));  
    note.setTitle(noteDetails.getTitle());  
    note.setContent(noteDetails.getContent());  
    Note updatedNote = noteRepository.save(note);  
    return updatedNote;  
}
```

```
public ResponseEntity<?> deleteNote(Long notelId) {  
    Note note = noteRepository.findById(notelId).orElseThrow(() ->  
        new ResourceNotFoundException("Note", "id", notelId));  
    noteRepository.delete(note);  
    return ResponseEntity.ok().build();  
}  
}
```

# Creating NoteController

```
@RestController
@RequestMapping("/api")
public class NoteController {
    @Autowired
    INoteService noteService;

    // Get All Notes
    // Create a new Note
    // Get a Single Note
    // Update a Note
    // Delete a Note
}
```

**@RestController** annotation is a combination of Spring's **@Controller** and **@ResponseBody** annotations.

The **@Controller** annotation is used to define a controller and the **@ResponseBody** annotation is used to indicate that the return value of a method should be used as the response body of the request.

**@RequestMapping("/api")** declares that the url for all the apis in this controller will start with /api.



# NoteController API

Let's now look at the implementation of all the apis one by one.

## 1. Get All Notes (GET /api/notes)

```
// Get All Notes
@GetMapping("/notes")
public List<Note> getAllNotes() {
    return noteService.getAllNotes();
}
```

The above method is pretty straightforward.

It calls corresponding Service method which in turn invokes JpaRepository's findAll() method ( NoteRepository extends JpaRepository) to retrieve all the notes from the database and returns the entire list.

Also note that the **@GetMapping("/notes")** annotation is a short form of **@RequestMapping(value="/notes", method= RequestMethod.GET)**

# NoteController API

## 2. Create a new Note (POST /api/notes)

```
{  
  "id": 0,  
  "title": "My Nth Note",  
  "content": "Spring Microservices with Spring Boot"  
}
```

```
// Create a new Note @PostMapping("/notes")  
public Note createNote(@Valid @RequestBody Note note) {  
    return noteService.createNote(note);  
}
```

The `@RequestBody` annotation is used to bind the request body with a method parameter.

The `@Valid` annotation makes sure that the request body is valid. Remember, we had marked Note's title and content with `@NotBlank` annotation in the Note model.

If the request body doesn't have a title or a content, then spring will return a 400 `BadRequest` error to the client.

# NoteController API

## 3. Get a Single Note (Get /api/notes/{noteId})

// Get a Single Note

```
@GetMapping("/notes/{id}")
```

```
public Note getNoteById(@PathVariable(value = "id") Long noteId) {
```

```
    return noteService.getNoteById(noteId);
```

```
}
```

NoteService

```
public Note getNoteById(Long noteId) {  
    return noteRepository.findById(noteId).orElseThrow(() ->  
        new ResourceNotFoundException("Note", "id", noteId));  
}
```

The `@PathVariable` annotation, as the name suggests, is used to bind a path variable with a method parameter.

In the above method, we are throwing a `ResourceNotFoundException` whenever a `Note` object with the given id is not found.

This will cause Spring Boot to return a 404 Not Found error to the client (Remember, we had added a `@ResponseStatus(value = HttpStatus.NOT_FOUND)` annotation to the `ResourceNotFoundException` class).

# NoteController API

## 4. Update a Note (PUT /api/notes/{noteId})

// Update a Note

@PutMapping("/notes/{id}")

public Note updateNote(@PathVariable(value = "id") Long noteId, @Valid

@RequestBody Note noteDetails) {

return noteService.updateNote(noteId, noteDetails);

}

**NoteService**

```
public Note updateNote(Long noteId, Note noteDetails) {  
    Note note = noteRepository.findById(noteId) .orElseThrow(() ->  
        new ResourceNotFoundException("Note", "id", noteId));  
    note.setTitle(noteDetails.getTitle());  
    note.setContent(noteDetails.getContent());  
    Note updatedNote = noteRepository.save(note);  
    return updatedNote;  
}
```

# NoteController API

## 5. Delete a Note (DELETE /api/notes/{noteId})

// Delete a Note

```
@DeleteMapping("/notes/{id}")
public ResponseEntity<?> deleteNote(@PathVariable(value = "id") Long noteId) {
    return noteService.deleteNote(noteId);
}
```

### NoteService

```
public ResponseEntity<?> deleteNote(Long noteId) {
    Note note = noteRepository.findById(noteId) .orElseThrow(() ->
        new ResourceNotFoundException("Note", "id", noteId));
    noteRepository.delete(note);
    return ResponseEntity.ok().build();
}
```

Note:

Run the application.

Test using chrome plugin **Postman**

# Accessing data from public API



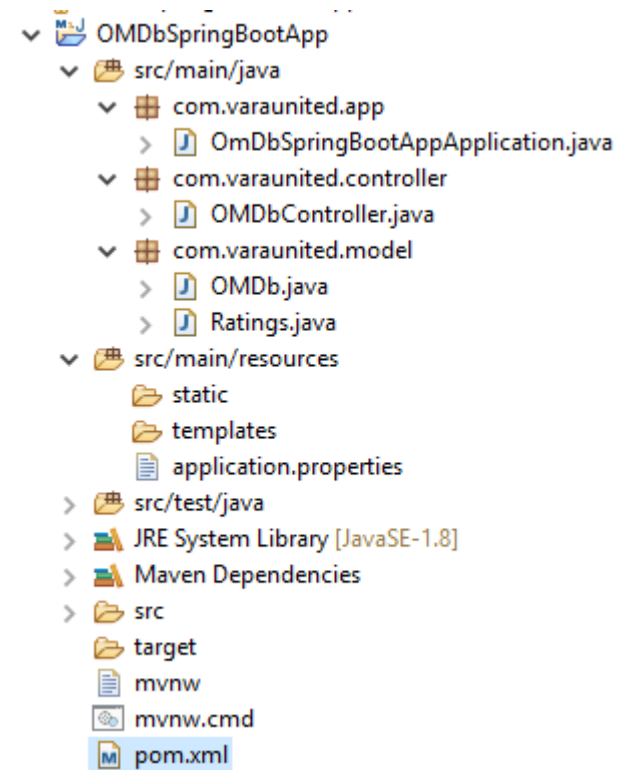
OMDbSpringBootTestApp.zip

1. Go to <http://start.spring.io/> . Select required dependencies (**Web, Actuator and DevTools** ) and generate project.
2. A zip file will be saved into your local directory.
3. Unzip.
4. In Eclipse, File->Import->New->Existing Maven Projects.
5. Update pom.xml, include following additional dependency.

```
<!-- https://mvnrepository.com/artifact/org.json/json -  
->  
<dependency>  
    <groupId>org.json</groupId>  
    <artifactId>json</artifactId>  
    <version>20180130</version>  
</dependency>
```

# Accessing data from public API

```
package com.varaunited.model;
import java.io.Serializable;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
/* Connect to http://www.omdbapi.com/ to get public key for their REST APIs Using http://pojo.sodhanalibrary.com/ to get
Java POJO class from JSON data. */
/* @JsonIgnoreProperties : Annotation that can be used to either suppress serialization of properties (during
serialization), or ignore processing of JSON properties read (during de-serialization).
*/
@JsonIgnoreProperties
public class OMDb implements Serializable{
    private static final long serialVersionUID = 1L;
    private String Released;
    private String Website;
    private String Type;
    private String imdbVotes;
    private Ratings[] Ratings;
    private String Runtime;
    private String Response;
    private String Poster;
    private String imdbID;
    private String Country;
    private String BoxOffice;
    private String Title;
    private String DVD;
    private String imdbRating;
    private String Year;
    private String Rated;
    private String Actors;
    private String Plot;
    private String Metascore;
    private String Writer;
    private String Production;
    private String Genre;
    private String Language;
    private String Awards;
    private String Director;
    //getters and setters
}
```



# Accessing data from public API

```
package com.varaunited.model;

public class Ratings{
    private String Source;

    private String Value;

    public String getSource ()    {
        return Source;
    }

    public void setSource (String Source)    {
        this.Source = Source;
    }

    public String getValue ()    {
        return Value;
    }

    public void setValue (String Value)    {
        this.Value = Value;
    }

    @Override
    public String toString()    {
        return "ClassPojo [Source = "+Source+", Value = "+Value+"]";
    }
}
```



# Accessing data from public API

```
@RestController
@RequestMapping("/api")
public class OMDbController {
    //http://localhost:8080/api/movie/Gravity
    @SuppressWarnings({ "rawtypes", "unchecked", "unused" })
    @GetMapping("/movie/{title}")
    public ResponseEntity getMovieDetailsByTitle(@PathVariable(value="title") String title ) {
        String uri="http://www.omdbapi.com/?apikey=97b78279&"+ "t="+ title;
        RestTemplate restTemplate = new RestTemplate();
        String str= restTemplate.getForObject(uri, String.class);
        try {
            //String object to JSON object
            JSONObject json = new JSONObject(str);
            //JSON object to Java object
            OMDb oMDb=new OMDb();
            oMDb.setActors(json.getString("Actors"));
            oMDb.setDirector(json.getString("Director"));
            oMDb.setTitle(json.getString("Title"));
            oMDb.setGenre(json.getString("Genre"));
            oMDb.setImdbRating(json.getString("imdbRating"));
            oMDb.setAwards(json.getString("Awards"));
            oMDb.setPoster(json.getString("Poster"));
            if(oMDb !=null) {
                return new ResponseEntity(oMDb,HttpStatus.FOUND);
            }else {
                throw new JSONException("Invalid Title");
            }
        } catch (JSONException e) {
            e.printStackTrace();
            return new ResponseEntity("Invalid Title",HttpStatus.NOT_ACCEPTABLE);
        }
    }
}
```

# Accessing data from public API

```
@SpringBootApplication(scanBasePackages="com.varaunited")  
public class OmDbSpringBootApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(OmDbSpringBootApplication.class, args);  
    }  
}
```

<http://localhost:8080/api/movie/Gravity>



Thank You!