# Introduction to Spring Boot

Spring is widely used for creating scalable applications. For web applications Spring provides  
Spring MVC which is a widely used module of spring which is used to create scalable web applications.  
But main **disadvantage of spring projects is that configuration is really time-consuming and can be a bit overwhelming for the new developers.** Making the application production-ready takes some time if you are new to the spring.

Solution to this is Spring Boot. Spring Boot is built on the top of the spring and contains all the features of spring. And is becoming favourite of developer’s these days because of it’s a rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and set up.

**Spring Boot is a microservice-based framework and making a production-ready application in it takes very less time.**Prerequisite for Spring Boot is the basic knowledge Spring framework.  
For revising the concepts of spring framework [read this article.](https://www.geeksforgeeks.org/introduction-to-spring-framework/)

### Features of Spring Boot

Spring Boot is built on the top of the conventional spring framework. So, it provides all the features of spring and is yet easier to use than spring.

* **It allows to avoid heavy configuration of XML which is present in spring:**  
  Unlike the [Spring MVC Project](https://www.geeksforgeeks.org/spring-mvc-with-jsp-view/), in spring boot everything is auto-configured. We just need to use proper configuration for utilizing a particular functionality.

**For example:** If we want to use [hibernate(ORM)](https://www.geeksforgeeks.org/introduction-to-hibernate-framework/) then we can just add **@Table annotation** above model/entity class(discussed later) and add **@Column annotation** to map it to table and columns in the database

* **It provides easy maintenance and creation of REST end points:**  
  Creating a [REST API](https://www.geeksforgeeks.org/rest-api-introduction/) is very easy in Spring Boot. Just the annotation **@RestController** and **@RequestMapping(/endPoint)** over the controller class does the work.
* **It includes embedded Tomcat-server:**

Unlike [Spring MVC project](https://www.geeksforgeeks.org/spring-mvc-with-jsp-view/) where we have to manually add and install the tomcat server, Spring Boot comes with an [embedded Tomcat server](https://www.geeksforgeeks.org/embedding-tomcat-server-in-maven-project/), so that the applications can be hosted on it.

* **Deployment is very easy, war and jar file can be easily deployed in the tomcat server:**  
  **war** or **jar** files can be directly deployed on the Tomcat Server and Spring Boot provides the facility to convert our project into war or jar files. Also, the instance of Tomcat can be run on the cloud as well.
* **Microservice Based Architecture:**  
  Microservice, as the name suggests is the name given to a module/service which focuses on a single type of feature, exposing an API(application peripheral interface).

Let us consider an example of a hospital management system.

* + In case of monolithic systems, there will be a single code containing all the features which are very tough to maintain on a huge scale.
  + But in the microservice-based system, each feature can be divided into smaller subsystems like service to handle patient registration, service to handle database management, service to handle billing etc.

Microservice based system can be easily migrated as only some services need to be altered which also makes debugging and deployment easy. Also, each service can be integrated and can be made in different technologies suited to them.

### Evolution of Spring Boot

1. Spring Boot came into existence when in October 2012, a client, **Mike Youngstrom** made a Jira request asking for bootstrapping the spring framework so that it can be quickly started. And hence in early 2013, Spring Boot was made.
2. In April 2014, Spring Boot **1.0** was created followed by various versions.
3. Spring Boot **1.1** on June 2014,
4. **1.2** in March 2015,
5. **1.3** in December 2016,
6. **1.4** in January 2017 and
7. Spring Boot **1.5** on February 2017.

### Spring Boot Architecture

To understand the architecture of Spring Boot, let us first see different layers and classes present in it.

* **Layers in Spring Boot:** There are four main layers in Spring Boot:
  + **Presentation Layer:** As the name suggests, it consists of views(i.e. frontend part)
  + **Data Access Layer:** CRUD (create, retrieve, update, delete) operations on the database comes under this category.
  + **Service Layer:** This consist of service classes and uses services provided by data access layers.
  + **Integration Layer:** It consists of web different web services(any service available over the internet and uses [XML](https://www.geeksforgeeks.org/html-and-xml-gq/) messaging system).
* Then we have utility classes, validator classes and view classes.
* All the services provided by the classes are implemented in their corresponding classes and are retrieved by implementing the dependency on those interfaces.

**Spring Boot flow architecture**:

* Since Spring boot uses all the features/modules of spring-like Spring data, [Spring MVC](https://www.geeksforgeeks.org/spring-mvc-with-jsp-view/) etc. so the architecture is almost the same as spring MVC, except for the fact that there is no need of **DAO** and **DAOImpl classes** in Spring boot.
* Creating a data access layer needs just a repository class instead which is implementing CRUD operation containing class.
* A client makes the https request(PUT/GET)
* Then it goes to controller and the controller mapped with that route as that of request handles it, and calls the service logic if required.
* Business logic is performed in the service layer which might be performing the logic on the data from the database which is mapped through JPA with model/entity class
* Finally, a JSP page is returned in the response if no error occurred.

### Setup Spring Boot:

1. Setup [Java JDK](https://www.geeksforgeeks.org/setting-environment-java/) from [Oracle’s official site](https://www.oracle.com/technetwork/java/javase/downloads/index.html" \t "https://www.geeksforgeeks.org/introduction-to-spring-boot/_blank).
2. Download and Setup [STS(Spring Tools Suite)](https://spring.io/tools3/sts/all" \t "https://www.geeksforgeeks.org/introduction-to-spring-boot/_blank).
3. Start a new spring starter project
   * Click on File -> New -> Spring starter project
   * Fill the appropriate details and add dependency and finish.
   * Edit the application properties.
   * Run the main file as a Java application.

# **Spring Boot – Architecture**

The **Spring Boot** is built on top of the core [Spring](https://www.geeksforgeeks.org/introduction-to-spring-framework/) framework. It is a simplified and automated version of the spring framework. The spring boot follows a **layered architecture** in which each layer communicates to other layers(Above or below in hierarchical order). The spring boot documentation provides the following definition to the Spring Boot Framework.

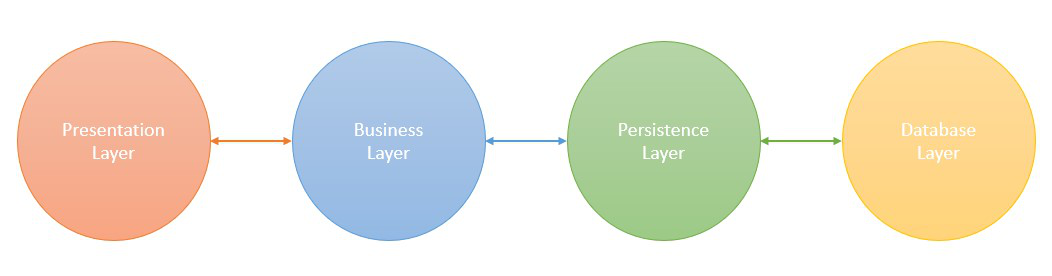
*Spring Boot makes it easy to create stand-alone, production-grade Spring based application that you can “Just Run”*

The main aim of spring boot is to remove the XML and annotations-based configuration settings from the application. Along with this spring boot provides the following benefits such as opinionated(options to later change the configuration), convention over configuration, stand-alone, and production-ready.

### ****Spring Boot Layers****

The spring boot consists of the following four layers:

1. **Presentation Layer** – Authentication & Json Translation
2. **Business Layer** – Business Logic, Validation & Authorization
3. **Persistence Layer** – Storage Logic
4. **Database Layer** – Actual Database



**1. Presentation Layer**

The presentation layer is the top layer of the spring boot architecture. It consists of Views. i.e., the front-end part of the application. It handles the HTTP requests and performs authentication. It is responsible for converting the JSON field’s parameter to Java Objects and vice-versa. Once it performs the authentication of the request it passes it to the next layer. i.e., the business layer.

**2. Business Layer**

The business layer contains all the business logic. It consists of services classes. It is responsible for validation and authorization.

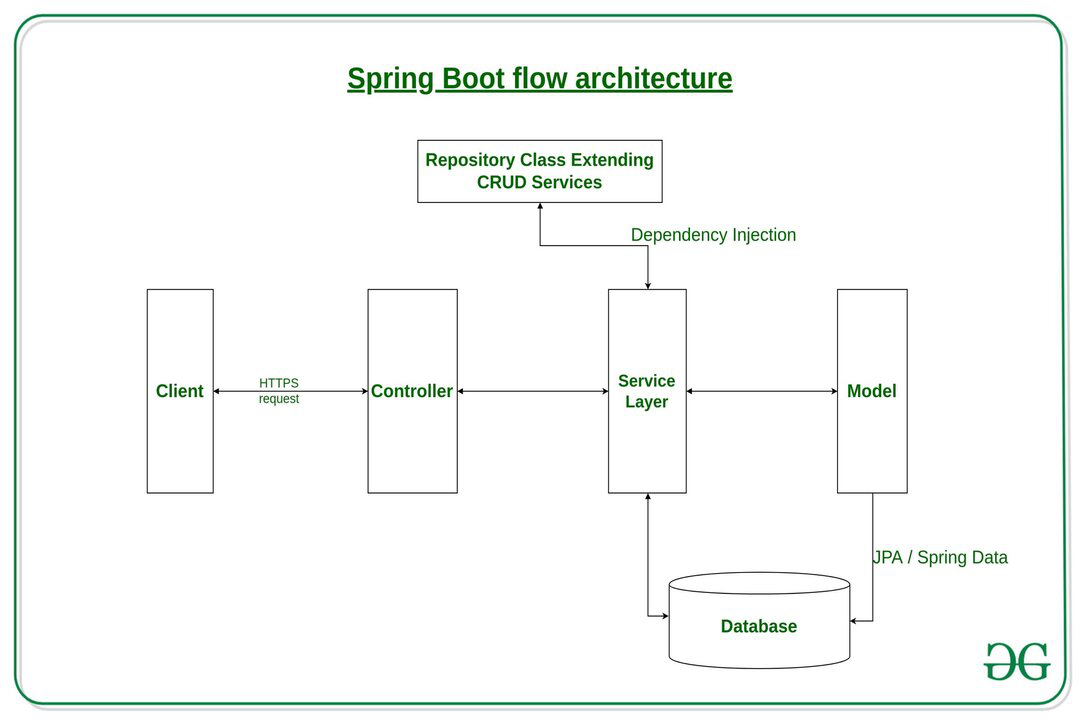
**3. Persistence Layer**

The persistence layer contains all the database storage logic. It is responsible for converting business objects to the database row and vice-versa.

**4. Database Layer**

The database layer contains all the databases such as [MySql](https://www.geeksforgeeks.org/sql-tutorial/), [MongoDB](https://www.geeksforgeeks.org/mongodb-an-introduction/), etc. This layer can contain multiple databases. It is responsible for performing the [CRUD](https://www.geeksforgeeks.org/spring-boot-crud-operations/) operations.

### Spring Boot Flow Architecture



*Fig 2 – Spring boot flow architecture*

**Explanation:**

* The Client makes an **HTTP** request(GET, PUT, POST, etc.)
* The HTTP request is forwarded to the **Controller**. The controller maps the request. It processes the handles and calls the server logic.
* The business logic is performed in
* the **Service layer**. The spring boot performs all the logic over the data of the database which is mapped to the spring boot model class through [Java Persistence Library](https://www.geeksforgeeks.org/spring-boot-spring-data-jpa/)(**JPA**).
* The [JSP](https://www.geeksforgeeks.org/introduction-to-jsp/) page is returned as Response from the controller.

# **How to Create a Spring Boot Project?**

[Spring Boot](https://www.geeksforgeeks.org/introduction-to-spring-boot/) is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time. Following are some of the features of Spring Boot:

* It allows avoiding heavy configuration of XML which is present in spring
* It provides easy maintenance and creation of REST endpoints
* It includes embedded Tomcat-server
* Deployment is very easy, war and jar files can be easily deployed in the tomcat server

For more information please refer to this article: [Introduction to Spring Boot](https://www.geeksforgeeks.org/introduction-to-spring-boot/)

Generally, to develop a Spring Boot Application we choose **Eclipse, Spring Tool Suite**,and **IntelliJ IDEA**IDE. So in this article, we are going to create our spring boot project in these 3 IDEs.

# Spring Boot – Annotations

[Spring Boot](https://www.geeksforgeeks.org/introduction-to-spring-boot/) is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time. Following are some of the features of Spring Boot:

* It allows avoiding heavy configuration of XML which is present in spring
* It provides easy maintenance and creation of REST endpoints
* It includes embedded Tomcat-server
* Deployment is very easy, war and jar files can be easily deployed in the tomcat server

**Spring Annotations** are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program. So in this article, we are going to discuss annotations that are available in Spring Boot with examples.

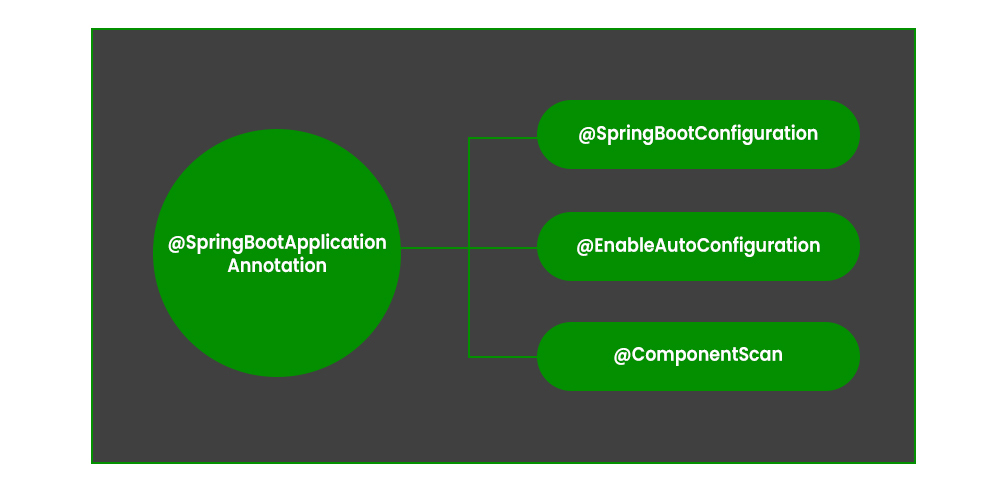
### [Spring Boot Annotations](https://www.geeksforgeeks.org/spring-framework-annotations/)

Spring annotations present in the ***org.springframework.boot.autoconfigure*** and ***org.springframework.boot.autoconfigure.condition***packages are commonly known as Spring Boot annotations. Some of the annotations that are available in this category are:

* @SpringBootApplication
* @SpringBootConfiguration
* @EnableAutoConfiguration
* @ComponentScan
* Auto-Configuration Conditions
  + @ConditionalOnClass, and @ConditionalOnMissingClass
  + @ConditionalOnBean, and @ConditionalOnMissingBean
  + @ConditionalOnProperty
  + @ConditionalOnResource
  + @ConditionalOnWebApplication and @ConditionalOnNotWebApplication
  + @ConditionalExpression
  + @Conditional

**1. @SpringBootApplication Annotation**

This annotation is used to mark the main class of a Spring Boot application. It encapsulates **@SpringBootConfiguration**, **@EnableAutoConfiguration**, and **@ComponentScan** annotations with their default attributes.



**Example:**

* sJava

|  |
| --- |
| @SpringBootApplication    // Class  **public** **class** DemoApplication {        // Main driver method  **public** **static** **void** main(String[] args)      {            SpringApplication.run(DemoApplication.**class**, args);      }  } |

**2. @SpringBootConfiguration Annotation**

It is a class-level annotation that is part of the Spring Boot framework. It implies that a class provides Spring Boot application configuration. It can be used as an alternative to Spring’s standard **@Configuration** annotation so that configuration can be found automatically. Most Spring Boot Applications use @SpringBootConfiguration via @SpringBootApplication. If an application uses @SpringBootApplication, it is already using @SpringBootConfiguration.

**Example:**

* Java

|  |
| --- |
| @SpringBootConfiguration  **public** **class** Application {    **public** **static** **void** main(String[] args) {          SpringApplication.run(Application.**class**, args);      }        @Bean  **public** StudentService studentService() {  **return** **new** StudentServiceImpl();      }  } |

**3. @EnableAutoConfiguration Annotation**

This annotation auto-configures the beans that are present in the classpath. It simplifies the developer’s work by assuming the required beans from the classpath and configure it to run the application. This annotation is part of the spring boot framework. For example, when we illustrate the **spring-boot-starter-web** dependency in the classpath, Spring boot auto-configures [Tomcat](https://www.geeksforgeeks.org/embedding-tomcat-server-in-maven-project/), and [Spring MVC](https://www.geeksforgeeks.org/difference-between-spring-mvc-and-spring-boot/). The package of the class declaring the @EnableAutoConfiguration annotation is considered as the default. Therefore, we need to apply the @EnableAutoConfiguration annotation in the root package so that every sub-packages and class can be examined.

**Example:**

* Java

|  |
| --- |
| @Configuration  @EnableAutoConfiguration  **public** **class** Application {    **public** **static** **void** main(String[] args) {          SpringApplication.run(Application.**class**, args);      }    } |

**4. @ComponentScan Annotation**

@ComponentScan tells Spring in which packages you have annotated classes that should be managed by Spring. So, for example, if you have a class annotated with [@Controller](https://www.geeksforgeeks.org/spring-controller-annotation-with-example/) which is in a package that is not scanned by Spring, you will not be able to use it as a Spring controller. So we can say @ComponentScan enables Spring to scan for things like configurations, controllers, services, and other components that are defined. Generally, @ComponentScan annotation is used with @Configuration annotation to specify the package for Spring to scan for components.

**Example:**

* Java

|  |
| --- |
| @Configuration  @ComponentScan    // Main class  **public** **class** Application {        // Main driver method  **public** **static** **void** main(String[] args)      {            SpringApplication.run(Application.**class**, args);      }  } |

**5. @ConditionalOnClass Annotation and @ConditionalOnMissingClass Annotation**

@ConditionalOnClass Annotation used to mark auto-configuration bean if the class in the annotation’s argument is present/absent.

**Example:**

* Java

|  |
| --- |
| @Configuration  @ConditionalOnClass(MongoDBService.**class**)    **class** MongoDBConfiguration {      // Insert code here  } |

**6. @ConditionalOnBean Annotation and @ConditionalOnMissingBean Annotation**

These annotations are used to let a bean be included based on the presence or absence of specific beans.

**Example:**

* Java

|  |
| --- |
| @Bean  @ConditionalOnMissingBean(type = "JpaTransactionManager")    JpaTransactionManager jpaTransactionManager(      EntityManagerFactory entityManagerFactory)  {      // Insert code here  } |

**7. @ConditionalOnProperty Annotation**

These annotations are used to let configuration be included based on the presence and value of a Spring Environment property.

**Example:**

* Java

|  |
| --- |
| @Bean  @ConditionalOnProperty(name = "usemongodb",                         havingValue = "local")    DataSource  dataSource()  {      // Insert code here  }    @Bean  @ConditionalOnProperty(name = "usemongodb",                         havingValue = "prod")    DataSource  dataSource()  {      // Insert code here  } |

**8. @ConditionalOnResource Annotation**

These annotations are used to let configuration be included only when a specific resource is present in the classpath.

**Example:**

* Java

|  |
| --- |
| @ConditionalOnResource(resources                         = "classpath:mongodb.properties")    Properties  additionalProperties()  {      // Insert code here  } |

**9. @ConditionalOnExpression Annotation**

These annotations are used to let configuration be included based on the result of a**SpEL (Spring Expression Language) expression.**

***SpEL (Spring Expression Language):****It is a powerful expression language that supports querying and manipulating an object graph at runtime.*

**Example:**

* Java

|  |
| --- |
| @Bean  @ConditionalOnExpression("${env} && ${havingValue == 'local'}")    DataSource dataSource()  {      // Insert code here  } |

**10. @ConditionalOnCloudPlatform Annotation**

These annotations are used to let configuration be included when the specified cloud platform is active.

**Example:**

* Java

|  |
| --- |
| @Configuration  @ConditionalOnCloudPlatform(CloudPlatform.CLOUD\_FOUNDRY)    **public** **class** CloudConfigurationExample  {    // Insert code here  } |

# **Spring Boot Actuator**

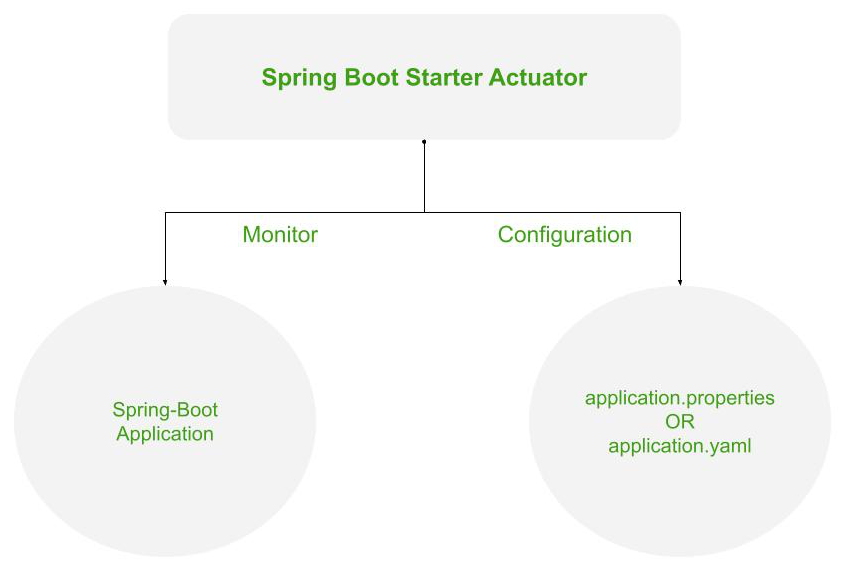
The Spring Framework is the most used platform which was released in October 2002 for building effective and efficient web-based applications. On top of it, the Spring Boot framework was released in April 2014 to overcome the cumbersome effort of manual configuration. The main moto of Spring Boot was to achieve the Auto-Configuration feature. With the help of this and other features, we are able to create a stand-alone Spring web application. Developing and Managing an application are the two most important aspects of the application’s life cycle. It is very crucial to know what’s going on beneath the application. Also when we push the application on production, managing it gradually becomes critically important. Therefore, it is always recommended to monitor the application both while at the development phase and at the production phase.

### ****Advantages of Monitoring/Managing the Application****

1. It increases customer satisfaction.
2. It reduces downtime.
3. It boosts productivity.
4. It improves Cybersecurity Management.
5. It increases the conversion rate.

### ****Spring Boot – Actuator****

* With the help of Spring Boot, we can achieve the above objectives.
* Spring Boot’s ‘Actuator’ dependency is used to monitor and manage the Spring web application.
* We can use it to monitor and manage the application with the help of HTTP endpoints or with the JMX.



*Working of the Spring’s Actuator*

To use the ‘Actuator’ add the following dependency in your application’s project build.

***Maven* -> pom.xml**

<dependencies>

<dependency>

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

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

</dependency>

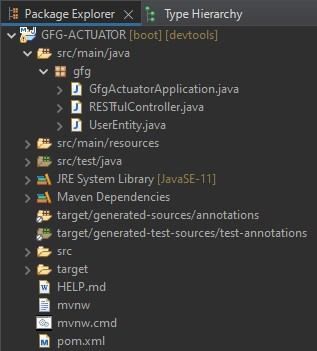
</dependencies>

***Gradle* -> build.gradle**

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-actuator'

}



*Project Structure – Maven*

**pom.xml** (Configuration of the Web Application)

## 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](http://maven.apache.org/POM/4.0.0%C2%A0)  <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**>2.6.4</**version**>          <**relativePath**/> <!-- lookup parent from repository -->      </**parent**>      <**groupId**>sia</**groupId**>      <**artifactId**>GFG-ACTUATOR</**artifactId**>      <**version**>0.0.1-SNAPSHOT</**version**>      <**name**>GFG-ACTUATOR</**name**>      <**description**>Spring Boot Starter Actuator</**description**>      <**properties**>          <**java.version**>11</**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**>              <**optional**>true</**optional**>          </**dependency**>          <**dependency**>              <**groupId**>org.projectlombok</**groupId**>              <**artifactId**>lombok</**artifactId**>              <**optional**>true</**optional**>          </**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**>                  <**configuration**>                      <**excludes**>                          <**exclude**>                              <**groupId**>org.projectlombok</**groupId**>                              <**artifactId**>lombok</**artifactId**>                          </**exclude**>                      </**excludes**>                  </**configuration**>              </**plugin**>          </**plugins**>      </**build**>  </**project**> |

**GfgActuatorApplication.java** (Bootstrapping of the application)

## Java

|  |
| --- |
| **package** gfg;    **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;    @SpringBootApplication  **public** **class** GfgActuatorApplication {    **public** **static** **void** main(String[] args)      {          SpringApplication.run(GfgActuatorApplication.**class**,                                args);      }  } |

**UserEntity.java** (Entity class representing the model data)

* This class acts as a simple java bean whose properties are returned as JSON response by the REST API’s get() method.
* ‘Lombok’ library is used to generate GETTER/SETTER methods automatically at runtime using ‘*@Data*‘ annotation.
* ‘*@RequiredArgsConstructor*‘ annotation is used to generate a zero-argument constructor and if final or ‘*@NonNull’*fields are present, then respective arguments constructor is created.
* To add the ‘Lombok’ library in your application, add the following dependency in your application’s project build.

***Maven* -> pom.xml**

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

‘@Component’ annotation is used so that this bean automatically gets registered in Spring’s application context.

## Java

|  |
| --- |
| **package** gfg;    **import** lombok.Data;  **import** lombok.RequiredArgsConstructor;  **import** org.springframework.stereotype.Component;    @Component  @Data  @RequiredArgsConstructor  **public** **class** UserEntity {      String id = "1";      String name = "Darshan.G.Pawar";      String userName = "@drash";      String email = "drash@geek";      String pincode = "422-009";  } |

**RESTfulController.java** (A REST API controller)

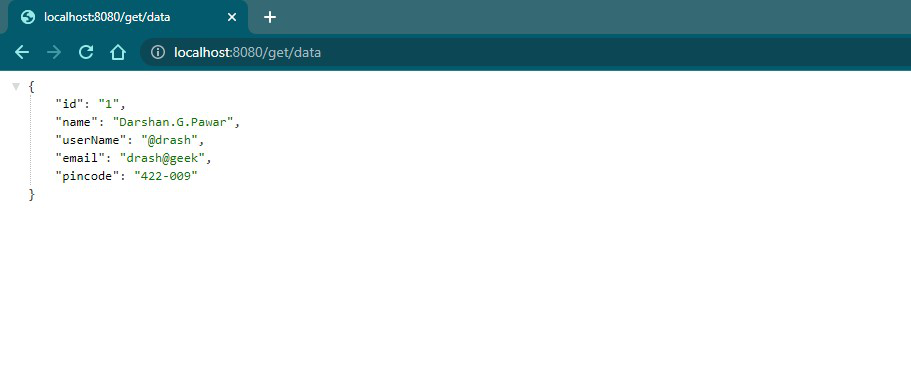
This controller’s get() method uses the UserEntity bean to return JSON response. UserEntiy bean is outsourced through ‘*@Autowired*‘ annotation which was registered in Spring’s application context.

## Java

|  |
| --- |
| **package** gfg;    **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;    @RestController  @RequestMapping("/get")  **public** **class** RESTfulController {        @Autowired      UserEntity entity;        @GetMapping("/data") **public** UserEntity getEntity()      {  **return** entity;      }  } |

**Output:**

Here, the JSON Formatter Chrome extension is used to automatically parse the JSON body. Further, it will be required to work with ‘Actuator’.

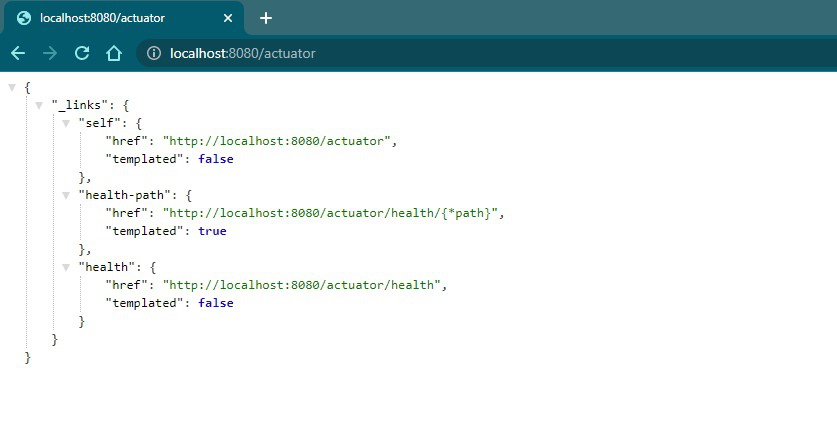


*JSON response returned by REST API*

**Working with Spring Boot Actuator**

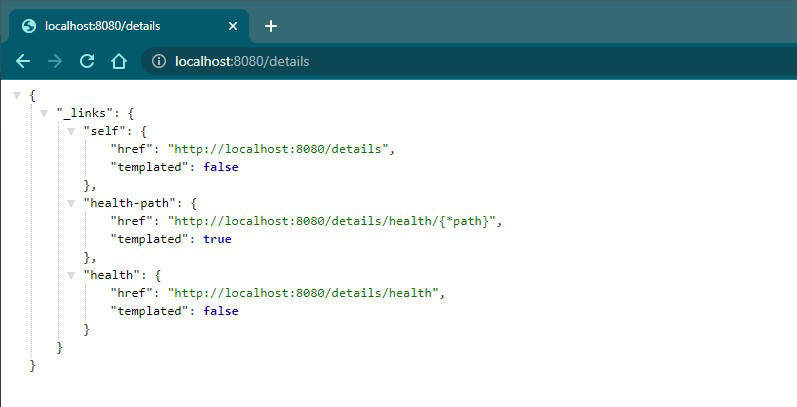
To access the ‘Actuator’ services, you will have to use the HTTP endpoint as it becomes reliable to work with. The default endpoint is ‘/actuator’.

**Example:**

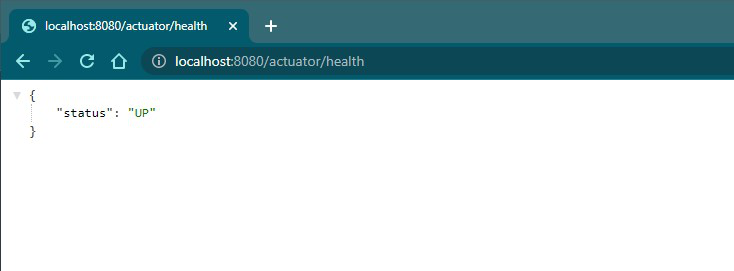


You can also change the default endpoint by adding the following in the application.properties file.

management.endpoints.web.base-path=/details



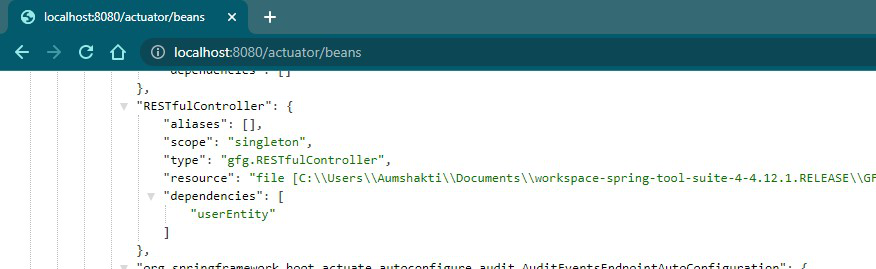
You can click on these above links and see the respective information. Additionally, you can activate other Actuator IDs and use them after ‘/actuator’ to see more information. For example, ‘health’ ID is activated by default. Therefore you can click the link in the image or directly use ‘http://localhost:8080/actuator/health’.



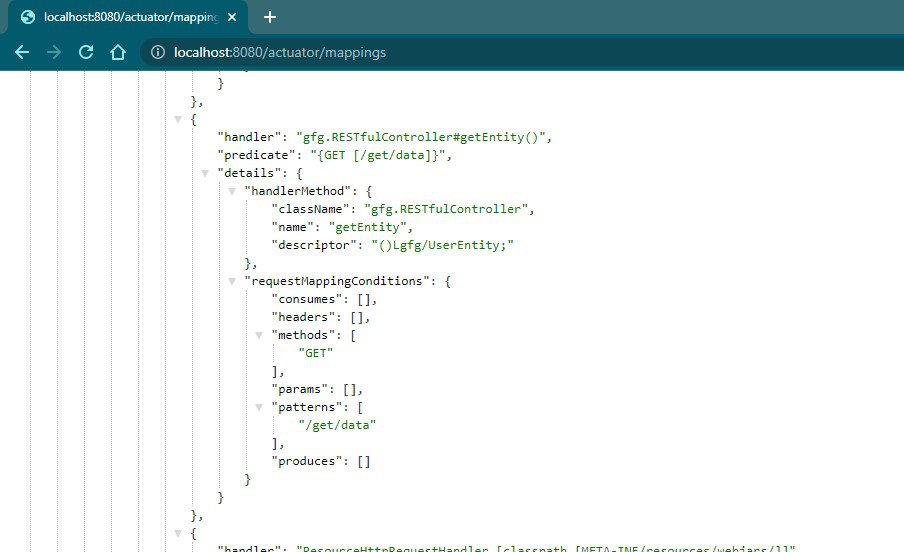
*The health of an application*

‘UP’ means the application’s health is good. There are a total of 25 IDs out of which the commonly used are listed out here –

| **ID** | **Description** |
| --- | --- |
| beans | Displays a complete list of all the Spring beans in your application. |
| caches | Exposes available caches. |
| conditions | Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match. |
| health | Shows application health information. |
| httptrace | Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges). Requires an HttpTraceRepository bean. |
| loggers | Shows and modifies the configuration of loggers in the application. |
| mappings | Displays a collated list of all @RequestMapping paths. |
| sessions | Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a servlet-based web application that uses Spring Session. |
| threaddump | Performs a thread dump. |



*Accessing ‘beans’ ID of the above project*



*Accessing ‘mappings’ ID of the above project*

**Including IDs/Endpoints**

By default, all IDs are set to false except for ‘health’. To include an ID, use the following property in the application.properties file.

management.endpoint.<id>.enabled

**Example ->** management.endpoint.metrics.enabled=true

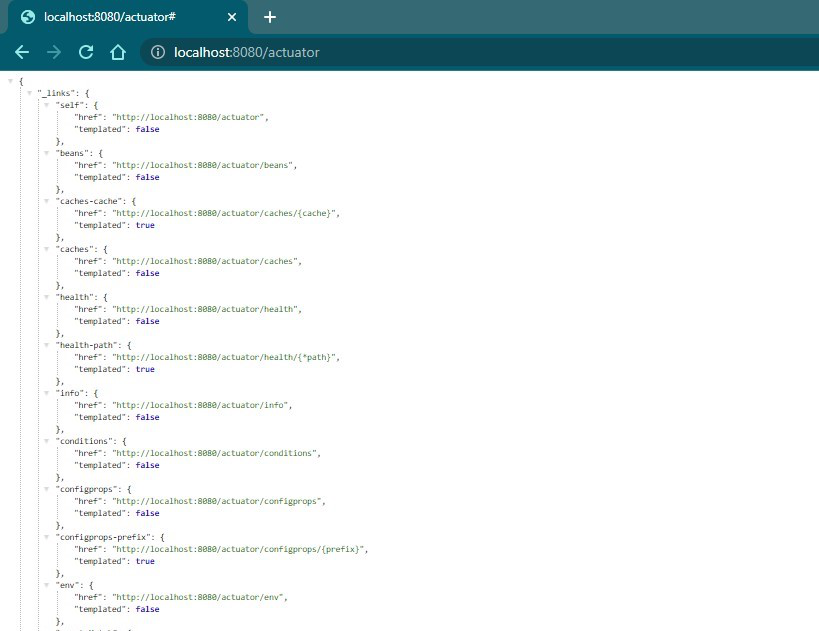
OR, you can just list down all IDs that you want to include which are separated by a comma.

management.endpoints.web.exposure.include=metrics,info

This will include only metrics and info IDs and will exclude all others (‘health’ too). To add/include all ID information about your application, you can do it in the application.properties file by simply adding the following –

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

**Output:**



*All the IDs or the Endpoint are now enabled*

**Excluding IDs/Endpoints**

To exclude an ID or endpoint, use the following property and list out the respective IDs separated by a comma in the application.properties file.

management.endpoints.web.exposure.exclude

**Example ->** management.endpoints.web.exposure.exclude=info

Use ‘\*’ in place of IDs in property to exclude all the IDs or endpoints.

***Notes:***

1. *Before setting the management.endpoints.web.exposure.include, ensure that the exposed actuators do not contain sensitive information.*
2. *They should be secured by placing them behind a firewall or are secured by something like Spring Security.*

# **Spring Boot – Introduction to RESTful Web Services**

REST stands for REpresentational State Transfer. It was developed by Roy Thomas Fielding, one of the principal authors of the web protocol HTTP. Consequently, REST was an architectural approach designed to make the optimum use of HTTP protocol. It uses the concepts and verbs already present in HTTP to develop web services. This made REST incredibly easy to use and consume, so much so that it is the go-to standard for building web services today. Unlike SOAP, REST does not have a standard messaging format. We can build REST web services using many formats, including both XML and JSON, although JSON is the more popular option. An important thing to consider is that REST is not a standard but a style whose purpose is to constrain our architecture to a client-server architecture and is designed to use stateless communication protocols like HTTP. The main methods of HTTP we build web services for are:

**GET**

The default request method for HTTP. We don’t have any request body with this method, but we can define multiple request parameters or path variables in the URL. This method is used for getting obtaining some resources. Depending on the presence of an ID parameter, either we can fetch a specific resource or fetch a collection of resources in the absence of the parameter. Sample GET request in Spring Boot Controller:

@GetMapping("/user/{userId}")

public ResponseEntity<Object> getUser(@PathVariable int userId) {

UserEntity user = userService.getUser(userId);

return new ResponseEntity<>(user, HttpStatus.OK);

}

**POST**

The POST method of HTTP is used to create a resource. We have a request body in this method and can also define multiple request parameters or path variables in the URL. Sample POST request in Spring Boot Controller:

@PostMapping(value = "/user")

public ResponseEntity<Object> addUser(@RequestBody UserEntity user) {

userService.saveOrUpdate(user);

return new ResponseEntity<>("User is created successfully", HttpStatus.CREATED);

}

**PUT**

The PUT method of HTTP is used to update an existing resource. We have a request body in this method and can also define multiple request parameters or path variables in the URL. Sample PUT request in Spring Boot Controller:

@PutMapping("/user/{userId}")

public ResponseEntity<Object> getUser(@RequestBody UserEntity user) {

userService.saveOrUpdate(user);

return new ResponseEntity<>("User is updated successfully", HttpStatus.OK);

}

**DELETE**

The DELETE method of HTTP is used to remove a resource. We don’t have a request body in this method but can define multiple request parameters or path variables in the URL. We can delete multiple or single records, usually based on whether we have an ID parameter or not. We can delete multiple or single records, usually based on whether we have an ID parameter or not. Sample DELETE request in Spring Boot Controller:

@DeleteMapping(value = "/user")

public ResponseEntity<Object> addUser(@PathVariable int userId) {

userService.deleteUser(userId);

return new ResponseEntity<>("User is deleted successfully", HttpStatus.OK);

}

REST web services use the Status-Line part of an HTTP response message to inform clients of their request’s ultimate result. The status codes defined in HTTP are the following:

* **200**: Success
* **201**: Created
* **401**: Unauthorized
* **404**: Resource Not Found
* **500**: Server Error

### ****When to use REST?****

The web services are completely stateless. The service producer and consumer have a mutual understanding of the context and content being passed along. When there is already some catching infrastructure present since we can use those to enhance performance in a REST API. This is so since idempotent requests like GET, PUT, and DELETE are all cacheable. Often Bandwith is of significant importance to organizations. Rest is instrumental then as there are no overhead headers from the SOAP XML payload. Web service delivery or aggregation into existing websites can be enabled easily with a RESTful style. It’s simply not required to overhaul the architecture since we can expose the API as an XML and consume the HTML web pages, thus still maintaining the external contract of the service.

### ****Principles of RESTful web services****

The following are the main principles rest services follow, which makes them fast, lightweight, and secure are:

* Resource Identification through URI- A RESTful web service provides an independent URI/ global ID  for every resource.
* Uniform Interface- Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE.
* Self-descriptive messages- Resources and representations are decoupled in a RESTful web service. This allows us to represent the payload in various formats such as HTML, XML, plain text, PDF, JPEG, JSON, and others based on our use case.
* Stateful Interaction through hyperlinks- Every interaction with a resource is stateless; that is, request messages are self-contained.

### ****Advantages of RESTful web services****

Some of the primary advantages of using RESTful web services are

* Easy to Built: REST APIs are simpler to build than a corresponding SOAP API.  Hence, REST is the better choice if we want to develop APIs quickly.
* Independent:: Since the client and server are decoupled in RESTful web services, it allows for independent development across projects.
* Scalability: Stateless communication and a replicated repository provide a high level of scalability. Compared to SOAP, scaling up an existing website is easier with the REST web services.
* Layered System: REST web services have their application divided into multiple layers forming a hierarchy. It makes the application both modular and scalable.

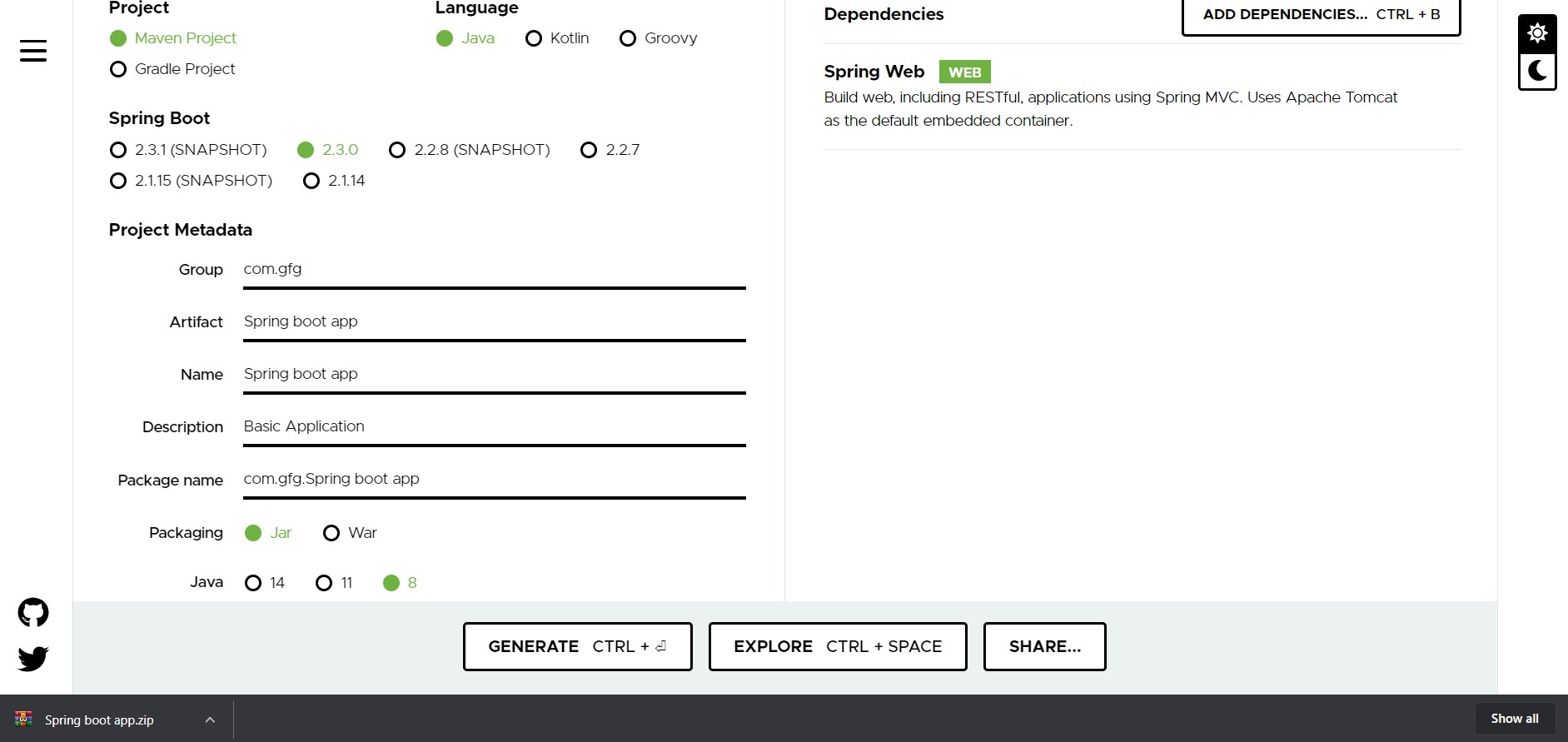
# **How to create a basic application in Java Spring Boot**

[Spring MVC](https://www.geeksforgeeks.org/introduction-to-spring-framework/) is a widely used module of spring which is used to create scalable [web applications](https://www.geeksforgeeks.org/how-to-choose-a-technology-stack-for-web-application-development/). But the main disadvantage of spring projects is that configuration is really time-consuming and can be a bit overwhelming for the new developers. Making the application production-ready takes some time if you are new to the spring. The solution to this is [Spring Boot](https://www.geeksforgeeks.org/introduction-to-spring-boot/). Spring Boot is built on the top of the spring and contains all the features of spring. In this article, we will see how to create a basic spring boot application.  
**Spring Initializr** is a web-based tool using which we can easily generate the structure of the Spring Boot project. It also provides various different features for the projects expressed in a metadata model. This model allows us to configure the list of dependencies which are supported by [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). Here, we will create the structure of an application using spring initializr and then use an [IDE](https://www.geeksforgeeks.org/what-will-be-the-best-java-ides-in-2020/) to create a sample GET route. Therefore, to do this, the following steps are followed: 

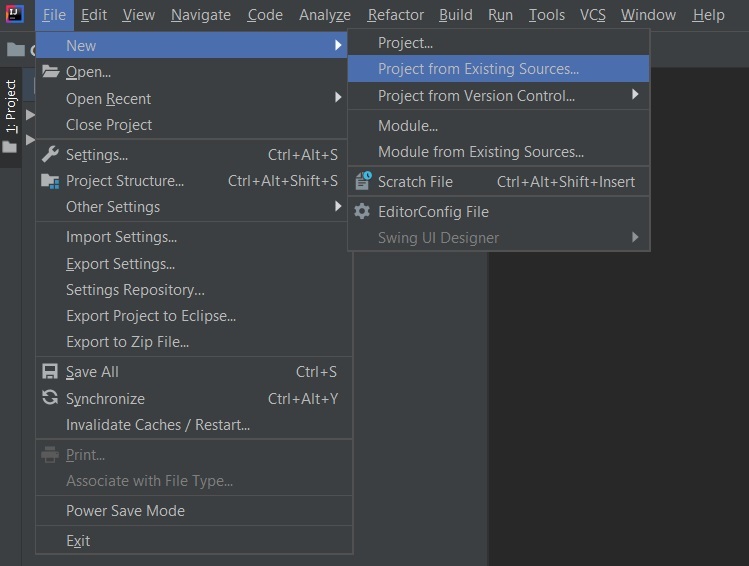
1. Go to [Spring Initializr](https://start.spring.io/" \t "https://www.geeksforgeeks.org/how-to-create-a-basic-application-in-java-spring-boot/_blank)
2. Fill in the details as per the requirements. For this application:

*Project: Maven   
Language: Java   
Spring Boot: 2.2.8   
Packaging: JAR   
Java: 8   
Dependencies: Spring Web*

1. Click on Generate which will download the starter project.



1. Extract the zip file. Now open a suitable IDE and then go to *File*->*New*->*Project from existing sources*->*Spring-boot-app* and select pom.xml. Click on import changes on prompt and wait for the project to sync.

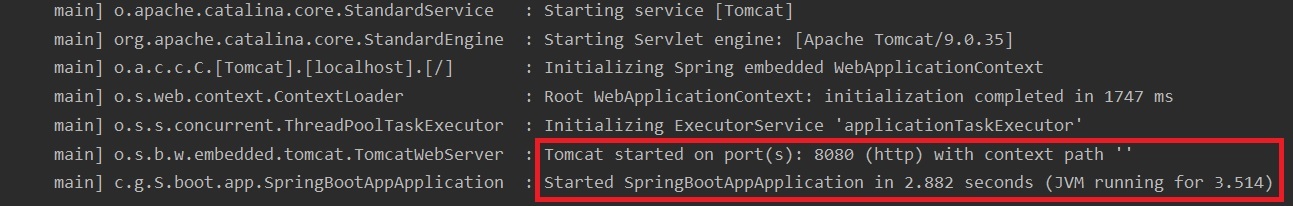


1. **Note:** In the Import Project for Maven window, make sure you choose the same version of JDK which you selected while creating the project.
2. Go to *src*->*main*->*java*->*com.gfg.Spring.boot.app*, create a java class with name   
   as Controller and add the annotation *@RestController*. Now create a GET API as shown below:

* Java

|  |
| --- |
| @RestController  **public** **class** Controller {        // One syntax to implement a      // GET method      @GetMapping("/")  **public** String home()      {          String str              = "<html><body><font color=\"green\">"                + "<h1>WELCOME To GeeksForGeeks</h1>"                + "</font></body></html>";  **return** str;      }        // Another syntax to implement a      // GET method      @RequestMapping(          method = { RequestMethod.GET },          value = { "/gfg" })    **public** String info()      {          String str2              = "<html><body><font color=\"green\">"                + "<h2>GeeksForGeeks is a Computer"                + " Science portal **for** Geeks. "                + "This portal has been "                + "created to provide well written, "                + "well thought and well explained "                + "solutions **for** selected questions."                + "</h2></font></body></html>";  **return** str2;      }  } |

1. This application is now ready to run. Run the *SpringBootAppApplication* class and wait for the Tomcat server to start.



1. **Note:** The default port of the Tomcat server is 8080 and can be changed in the *application.properties* file.
2. Now go to the browser and enter the URL **localhost:8080**. Observe the output and now do the same for **localhost:8080/gfg**