Introduction

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

Advantage

Spring Boot offers the following advantages to its developers −

1. Spring Boot makes it easy to create a stand-alone enterprise application
2. Easy to understand and develop spring applications
3. Single class will run your entire application with an integrated webserver.
4. Reduces the development time
5. Microservices can be developed in spring boot

Goals

1. Spring Boot removes all these dependencies problems and having a separate tomcat/jetty server.
2. To avoid complex XML configuration in Spring.
3. To develop a production-ready Spring application in an easier way.
4. It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.

Why Spring Boot?

1. It offers annotation-based spring application
2. To ease the Java-based applications Development, Unit Test, and Integration Test Process.
3. To reduce Development, Unit Test, and Integration Test time by providing some defaults.

Spring Boot Features

1. Web Development
2. Spring Application
3. Application events and listeners
4. Admin features
5. Externalized Configuration
6. Properties Files
7. Security

Web Development

It is a well-suited Spring module for web application development. We can easily create a self-contained HTTP server using embedded Tomcat, Jetty or Undertow. We can use the spring-boot-starter-web module to start and running application quickly.

FAQ

Q: Why do you use SpringBoot?
A: It is used to develop MicroServices

Q: How do you create microservices in SpringBoot?
A: We create microservices using REST controller

Installation and Configuration

Maven Dependency

Parent tag inherits common spring dependencies from parent POM file. Parent tag provides inheritance between POM files to avoid redundancies or duplicate configurations.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>


JDK 1.8 and above is required by Spring boot support annotation
<!-- Spring boot need JDK Version 1.8 and above  -->
<properties>
    <java.version>1.8</java.version>
</properties>


Spring Boot Starters. Starters are a set of convenient dependency descriptors whic
h we can include in our application. Spring Boot provides built-in starters which
makes development easier and rapid. It is used for building web, including RESTful
, applications using Spring MVC.
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>


<!--  Spring Boot Starter Security dependency is used for Spring Security  -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

```
<!-- Spring Boot Actuator provides secured endpoints for monitoring and managing A
pp. To get production-ready features  -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

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

<!-- Spring JPA dependency -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!-- MySQL database driver dependency -->
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.12</version>
</dependency>

<!-- Spring mail dependency -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

```
<!-- Used for logger dependency  -->
<dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
</dependency>

<!-- Used to plug-in Tomcat -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
</dependency>
```

Start the application; Entry Point

Each application has one startup file that contains *main()* method. Startup class must be annotated by *@SpringBootApplication* annotation.

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

Spring application is started by running following static method:

```
SpringApplication.run(Application.class, args);
```

Annotation @SpringBootApplication includes Auto-Configuration, Component Scan, and Spring Boot Configuration.

@ComponentScan

Spring Boot application scans all the beans and package declarations when the application initializes. By default *@SpringBootApplication* annotation scans annotated components (classes) from current packages and its subpackages.

If you want to scan components from different then default packages then use *@ComponentScan* annotation.

```

```

```
@SpringBootApplication
@ComponentScan(basePackages = {
        "com.sunilos.utility",
        "com.sunilos.common
    }
)
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
   @Bean
   public RestTemplate getRestTemplate() {
      return new RestTemplate();
   }
}
```

Bean Configuration

You can configure custom beans using *@Configuration* and *@Bean* annotation in custom class. Keep new class in the same package as you keep startup class file.

```
@Configuration
public class AppConfig {


@Bean
 public UserDTO userBean() {
          UserDTO dto = new UserDTO();
          dto.setFirstName("Ram");
          dto.setLastName("Sharma");
          return dto;
 }
}
```

You can also define bean in startup file using @Bean tag. Separate configuration classes are used to group similar types of bean definitions.

Application properties

Spring boot keeps application configurable parameters into application.properties file. This file is located in root class-path.

Configurable parameters contain by this file are

- Server port number
- Data connection pool properties
- JPA properties
- Email server properties
- Custom properties

```
#Server info

server.port = 8080

spring.application.name = SunilOSApp


#Custom properties

page.size=10


##Data connection pool

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/ORS_P10?useSSL=false

spring.datasource.username = ram

spring.datasource.password = ram


## Hibernate/JPA  Properties

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format_sql = true

org.hibernate.dialect.MySQLDialect

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MariaDB53Dialect

spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate5.Sprin
gSessionContext


#Email properties

spring.mail.host=smtp.gmail.com

spring.mail.port=587

spring.mail.username=my@gmail.com

spring.mail.password=mypass

spring.mail.properties.mail.smtp.auth=true

spring.mail.properties.mail.smtp.starttls.enable=true
```

You can add application context path using following predefined property.

```
server.servlet.context-path =/SunilOSApp
```

Deploy App on external web server

You can create a stand-alone web application in spring boot that can be executed in-build tomcat.

You can also create WAR of spring boot application and deploy it on an external J2EE web server like JBoss, Tomcat, Jetty etc.

In order to deploy the application on an external server,

1) you need to extend your startup class by *SpringBootServletInitializer*.

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

2) Create WAR file and deploy

In order to create WAR file, you have to change the packing setting to WAR in POM.xml file.

```
<packaging>war</packaging>
```

Now Simple we need to Click on Our Application Right click >Run As> Maven Build > then It will open a window of Edit Configuration and Launch > in the Goal text type “clean package” then click on run button.

At the console, you will see progress for build creation started.

Create REST API

Rest controller is used to provide rest apis (endpoints). Rest endpoints are are used to send and receive data in form of JSON over HTTP protocol.

Rest controller is created @RestController annotation.

```
@RestController
@RequestMapping(value = "User")
public class LoginCtl {


	@PostMapping("login")
	public User login(@RequestBody User user) {
		User u = service.authenticate(user.getLoginId(), user.getPassword
());
		return u;
	}


	@GetMapping("fp/{login}")
	public String forgotPassword(@PathVariable String login) {
		return "Password has been sent to email id";
	}
}
```

Above controller will expose following end points to access rest controller:

http://localhost:8080/User/login
http://localhost:8080/User/fp/myloginid

1. Annotation @RequestMapping is used to map controller with URL.
2. Annotation @GetMapping is used to map GET HTTP method and sub-url with method of controller.
3. Annotation @PostMapping is used to map POST HTTP method and sub-url with method of controller.

Data Connection Pool

Spring boot by default configure **Hikari** connection pool for JPA. HikariCP is a lightweight and lightning fast JDBC connection pooling framework.

Make following configuration in application.properties file to configure DCP:

```
#Spring DATASOURCE (DataSourceAutoConfiguration &DataSourceProperties)
spring.datasource.driver-class-name = com.mysql.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/demoapp
spring.datasource.username = root
spring.datasource.password = root

#Configure Hibernate/JPA  Properties
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql = true

#Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto=update
Spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQLDialect
```

Mail configuration

1) Maven dependency "s*pring-boot-starter-mail*" will be included in *pom.xml* .
2) Make the following configuration in the *application.properties* file to configure email:

```
#Email properties
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=abcd@gmail.com
spring.mail.password=password
spring.mail.properties.mail.smtp.auth=true

# TLS , port 587
spring.mail.properties.mail.smtp.starttls.enable=true
```

You should inject JavaMailSender object to send your emails.

```
@RestController
public class SimpleMailController {
    @Autowired
    private JavaMailSender sender;

    @RequestMapping("/sendMail")
    public String sendMail() {
        MimeMessage message = sender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message);
        try {
            helper.setTo("demo@gmail.com");
            helper.setText("Greetings :)");
            helper.setSubject("Mail From Spring Boot");
        } catch (MessagingException e) {
            e.printStackTrace();
            return "Error while sending mail ..";
        }
        sender.send(message);
        return "Mail Sent Success!";  }    }
```

Annotations

Spring framework uses following annotations to create beans and inject dependencies. The Spring container will then recursively scan all components in the given package & its sub-packages.

Defining a Controller

@Controller annotation is used to mark any of our Spring MVC controller classes.

```
@Controller

public class MyController {

}
```

@RestController annotation is used to define RESTful controllers.

```
@RestController

public class MyRestController {

}
```

Rest controllers always return response in JSON format whereas normal controllers may return response into xml, html, tiles of any other format. Rest controllers may configured to send response other then JSON.

Map URL with Controller

@RequestMapping
Request URLs are mapped with Controller classes and its methods. Annotation @RequestMapping is used to map the URLs. This annotation provides several options to customize its behavior.

1. method: denotes one the HTTP request methods – PUT, GET, POST, DELETE, PATCH
2. value: the mapped URL
3. params: filters the request based on the HTTP parameters
4. headers: filters the request based on the HTTP headers
5. produces: defines the media type of the HTTP responses
6. consumes: specifies the media type of the HTTP request

```
@RequestMapping("/User")

public class UserController {

    @RequestMapping(value = "/login", method = RequestMethod.POST)

    public User login() {

        …

    }


    @RequestMapping(value = "/logout", method = RequestMethod.GET)

    public User logout() {

        …   }  }
```

@RequestParam

We can bind an HTTP request parameter to the method parameter using this annotation.

```
@GetMapping("/message")
String message(@RequestParam("userName") String userName) {
    return "Hello " + userName;
}
```

@PathVariable

We can bind URIor path variables with method parameters using this annotation

```
@GetMapping("/message/{userName}")
public String message(@PathVariable String userName) {
   Return “Ok”;
}
```

Also, we can choose to mark a path variable as optional by setting required is equal to false.

```
@GetMapping("/message/{userName}")
public String message(@PathVariable(required=false) String userName){
 return “Ok”;
}
```

@RequestBody

@RequestBody annotation maps JSON object from HttpRequest POST request body to Java object.. Spring automatically deserializes the JSON into a Java type.

```
@PostMapping("addUser")
          public ORSResponse detailsAmt(@RequestBody UserForm form) {
                    return res;
          }
}
```

@SessionAttribute

Annotation to bind a method parameter to a session attribute.@SessionAttribute used to pass value across different requests through the session. Rather than using HttpSession object directly, using this annotation can benefit auto type conversion and optional/required check.

```
@GetMapping("/message")
public String handle(@SessionAttribute("user") User user) {
 //
}
```

@ModelAttribute

@ModelAttribute is used to bind POST/GET request parameters with Java object. After binding request parameter, java object is stored in Spring MVC model object.

Data stored in Model is available to View component.

```
@PostMapping("/addUser")
public String addUser(@ModelAttribute("user") User user) {
}
```

The annotation is used to define objects which should be part of a Model. So if you want to have a User object referenced in the Model you can use the following method.

If you want to store return object of a method into model object then you can apply @ModelAttribute annotation on the method:

```
@ModelAttribute("user")
public User getUser() {
         return  new User();
}
```

Response-Handling Annotations

@ResponseBody

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

Following example will convert User object into JSON and send it to http response

```
@GetMapping("/login")
@ResponseBody
public User login() {
    return new User();
}
```

@ExceptionHandler

We can write custom exception handler methods. These methods will be invoked when an exception of its type is raised during the method execution.

```
@ExceptionHandler(RuntimeException.class)
public static void applicationException(RuntimeException ex) {
    // On User Denied Exception
}
```

@ResponseStatus

With this, we can map the desired HTTP response status to the methods in our controller

```
@ExceptionHandler(RuntimeException.class)
@ResponseStatus(HttpStatus.FORBIDDEN)
public static void applicationException(RuntimeException ex) {
    // On User Denied Exception
}
```

Enable CORS

Cross-Origin Resource Sharing (CORS) is a security policy that uses HTTP concept. It restricting the resources accessed from web browsers. It prevents the JavaScript code from producing or consuming requests against different origins.

Spring Boot CORS

The following Spring Boot application uses Angular for the frontend. The Angular SPA is run on localhost:4200 and makes a request to the Spring Boot backend, which runs on localhost:8080. For this to work, we need to enable CORS in the Spring Boot application.

A web page can embed cross-origin images, stylesheets, scripts, iframes, and videos. Some cross-domain requests, notably Ajax requests, are forbidden by default by the same-origin security policy.

Global CORS Configuration in Spring Boot main class

```
@SpringBootApplication
public class MyApplication {
   public static void main(String[] args) {
      SpringApplication.run(MyApplication.class, args);
   }
   @Bean
   public WebMvcConfigurer corsConfigurer() {
      return new WebMvcConfigurerAdapter(){
         @Override
          public void addCorsMappings(CorsRegistry registry) {
                    CorsRegistration cors = registry.addMapping("/**");
                    cors.allowedOrigins("http://localhost:4200");
                    cors.allowedHeaders("*");
                    cors.allowCredentials(true);
          }
      };
   }
}
```

FAQ

Q: How do you enable the CORS?
A: We will enable it with help of WebMvcConfigurerAdapter. We write code inside the startup class annotated by @SpringBootApplication.

Map Static Resources

You can map static resources like images, js, HTML, or other files using WebMvcConfigurerAdapter.addResourceHandlers method.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
  WebMvcConfigurer w = new WebMvcConfigurer() {
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry reg){
        ResourceHandlerRegistration hander = reg.addResourceHandler("/**");
          hander.addResourceLocations("classpath:/public/");
    }
  };
  return w;
}
```

Q: How do you enable the static resources?
A: We will enable it with help of WebMvcConfigurer. We write code inside the startup class annotated by @SpringBootApplication.

Q: What are the static resources?

Enable Interceptor

Front controller like interceptors can be configured using WebMvcConfigurerAdapter.addInterceptors method.

```
@Autowired
FrontCtl frontCtl;


@Bean
public WebMvcConfigurer corsConfigurer() {
  WebMvcConfigurer w = new WebMvcConfigurer() {
    public void addInterceptors(InterceptorRegistry reg) {
      InterceptorRegistration inter = reg.addInterceptor(frontCtl);
      inter.addPathPatterns("/**");
      inter.excludePathPatterns("/Auth/**");
    }
  return w;
}
```

Q: How do you enable the intercepter?
A: We will enable it with help of WebMvcConfigurer. We will use addInterceptors() method. We write code inside the startup class annotated by @SpringBootApplication.
Q: Why did you use interceptor in your application?

Scheduling

To schedule job in spring boot application to run periodically, spring boot provides @EnableScheduling and @Scheduled annotations

The simple rules that need to be followed to annotate a method with @Scheduled are:

- a method should have void return type
- a method should not accept any parameters

Add @EnableScheduling annotation to your spring boot application class. @EnableScheduling is a Spring Context module annotation. It internally imports the SchedulingConfiguration via the @Import(SchedulingConfiguration.class) instruction.

```
@SpringBootApplication

@EnableScheduling

public class SpringBootWebApplication {

}
```

Execute a task at a fixed interval of time

```
@Scheduled(initialDelay = 1000, fixedRate = 10000)

public void run() {

    logger.info("Current time is :: " + Calendar.getInstance().getTime());

}
```

Schedule task at fixed delay

Configure a task to run after a fixed delay. In given example, the duration between the end of last execution and the start of next execution is fixed. The task always waits until the previous one is finished.

```
@Scheduled(fixedDelay = 10000)

public void run() {

  system.out.println("Current time"+Calendar.getInstance().getTime());

}
```