**Q: What are advantages of Spring Boot ?  
A:** The advantages of Spring Boot are

* Reduce Developement, Testing time and efforts.
* Use of JavaConfig helps avoid usage of XML.
* Avoid lots of maven imports and the various version conflicts.
* Provide Opinionated Development approach.
* Quick start to development by providing defaults.
* No Separate Web Server Needed.Which means that you no longer have to boot up Tomcat, Glassfish, or anything else.
* Requires less configuration-Since there is no web.xml file. Simply add classes annotated with@Configuration and then you can add methods annotated with@Bean, and Spring will automagically load up the object and manage it like it always has. You can even add @Autowired to the bean method to have Spring autowire in dependencies needed for the bean.
* Environment Based Configuration-Using these properties, you can pass into the application which environment you are using with:-Dspring.profiles.active={enviornment}. Spring will then load up the subsequent application properties file at (application-{environment}.properties) after loading up the main application properties file.

**Q:How to reload my changes on Spring Boot without having to restart server?  
A:**This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers. One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart my server. This will eliminates the need for manually deploying the changes every time. Spring Boot doesn’t have this feature when it has released it’s first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment. It also provides H2-database console for better testing the application. The following dependency is used

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

The DevTool dependency usage for autorestart and H2 DB console is illustrated in this [example](http://www.javainuse.com/spring/SpringBootUsingPagination)

**Q:What is Actuator in Spring Boot?  
A:**Spring boot actuator is one of the important feature in spring boot framework. Spring boot actuator helps you to access the current state of the running application in production environment. There are several metrics that has to be checked and monitored in the production environment. Even some external applications may be using those services to trigger the alert message to concerned person. Actuator module exposes set of REST endpoints that can be directly accessed as a HTTP URL to check the status.

**Q:How to disable Actuator endpoint security in Spring Boot?  
A:** By default all sensitive HTTP endpoints are secured such that only users that have an ACTUATOR role may access them. Security is enforced using the standard HttpServletRequest.isUserInRole method.  
We can disable security using -   
**management.security.enabled=false**  
It is suggested to disable security only if the actuator endpoints are accessed behind firewall.  
  
**Q: How to run Spring boot application to custom port ?  
A:**In order to run a spring boot application on a custom port you can specify the port in application.properties.  
**server.port=8090**

**Q: Have you written Test cases using Spring Boot ?  
A:**Spring Boot provides the @SpringBootTest for writing Unit Test Cases

**Q: What is YAML ?  
A:**YAML is a human-readable data serialization language. It is commonly used for configuration files.  
Compared to properties file, YAML file is much more structured and less confusing in case we want to add complex properties in the configuration file. As can be seen YAML has hierarchical configuration data.

**Q: How to implement security for Spring boot application ?  
A:**For Implementing security for Spring Boot we use the **spring-boot-starter-security dependency** and have to add the Security config. It requires very little code. Config class will have to extend WebSecurityConfigurerAdapter and override its methods.

Add below dependency in pom.xml file

<dependency>

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

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

</dependency>

Configure the below class.

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

public void configure(AuthenticationManagerBuilder auth) throws Exception { auth.inMemoryAuthentication().withUser("javainuse") .password("javainuse").roles("USER");

}

@Override

public void configure(HttpSecurity http) throws Exception { http.antMatcher("/\*\*").authorizeRequests().anyRequest().hasRole("USER") .and().formLogin().loginPage("/login.jsp") .failureUrl("/login.jsp?error=1").loginProcessingUrl("/login") .permitAll().and().logout() .logoutSuccessUrl("/listEmployees.html");

}

}

**Q: Have you integrated Spring Boot and ActiveMQ ?  
A:**For integrating Spring Boot and ActiveMQ we use the **spring-boot-starter-activemq dependency**  
It requires very little configuration and no boilerplate code.

<dependency>

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

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

</dependency>

@SpringBootApplication

public class SpringBootHelloWorldApplication {

public static void main(String[] args) {

ApplicationContext ctx = SpringApplication.run( SpringBootHelloWorldApplication.class, args);  **JmsTemplate jms = ctx.getBean(JmsTemplate.class); jms.convertAndSend("javainuse", "test message");**

}

}

**Q: Have you integrated Spring Boot and Apache Kafka ?  
A:**For integrating Spring Boot and Apache Kafka we use the **spring-kafka** dependency.

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class KafkaSender {

@Autowired

private KafkaTemplate<String, String> kafkaTemplate;

String kafkaTopic = "java\_topic";

public void send(String message) {

kafkaTemplate.send(kafkaTopic, message);

}

}

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

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.4.0</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.4.0</version>

</dependency>

@Configuration

@EnableSwagger2

**public** **class** SwaggerConfig {

@Bean

**public** Docket api() {

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.select().apis(RequestHandlerSelectors

.*basePackage*("com.owners.wsseller.controller"))

.paths(PathSelectors.*any*()).build().apiInfo(apiInfo());

}

**private** ApiInfo apiInfo() {

Contact contact = **new** Contact("Owner Sell Team",

"http://www.owners.com/", "help@owners.com");

**return** **new** ApiInfoBuilder()

.title("Owner sell API")

.description("Below api's having all functionality related sell side")

.termsOfServiceUrl("http://www.owners.com/about/terms")

.contact(contact)

.version("2.0").build();

}

}

**Q.  How does path=”users”, collectionResourceRel=”users” work with Spring Data Rest?**  
@RepositoryRestResource(collectionResourceRel = "users", path = "users")  
public interface UserRestRepository extends  
PagingAndSortingRepository<User, Long>

* path – The path segment under which this resource is to be exported.
* collectionResourceRel – The rel value to use when generating links to the collection resource. This is used when generating HATEOAS links.

**Q16. How does Spring enable creating production ready applications in quick time?**  
Spring Boot aims to enable production ready applications in quick time. Spring Boot provides a few non functional features out of the box like caching, logging, monitoring and embedded servers.

* spring-boot-starter-actuator – To use advanced features like monitoring & tracing to your application out of the box
* spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat – To pick your specific choice of Embedded Servlet Container
* spring-boot-starter-logging – For Logging using logback
* spring-boot-starter-cache – Enabling Spring Framework’s caching support

**Q23. Why do we need spring-boot-maven-plugin?**  
spring-boot-maven-plugin provides a few commands which enable you to package the code as a jar or run the application

* spring-boot:run runs your Spring Boot application.
* spring-boot:repackage repackages your jar/war to be executable.
* spring-boot:start and spring-boot:stop to manage the lifecycle of your Spring Boot application (i.e. for integration tests).
* spring-boot:build-info generates build information that can be used by the Actuator.

**How can I enable auto reload of my application with Spring Boot?**

Use Spring Boot Developer Tools.

Adding Spring Boot Developer Tools to your project is very simple.

Add this dependency to your Spring Boot Project pom.xml

<dependency>

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

<artifactId>spring-boot-devtools</artifactId> <scope>runtime</scope>

</dependency>

Add the following entry in application.properties file

spring.devtools.restart.additional-paths=.

**Error : HAL browser gives me unauthorized error - Full authentication is required to access this resource. How can I fix it?**

{ "timestamp": 1488656019562, "status": 401, "error": "Unauthorized", "message": "Full authentication is required to access this resource.", "path": "/beans" }

Two options

**Option 1 : Disable security**

application.properties

management.security.enabled: FALSE

**Option 2 : Search for password in the log and pass it in the request header**

**What is best way to expose custom application configuration with Spring Boot?**

The problem with @Value is that you would have your configuration values distributed through out your application. A better option would be to have a centralized approach.

You can define a configuration component using **@ConfigurationProperties**.

@Component

@ConfigurationProperties("basic")

public class BasicConfiguration {

private boolean value;

private String message;

private int number;

}

The values can be configured in application.properties

basic.value: true

basic.message: Dynamic Message

basic.number: 100  
  
**Q : What is the need for Profiles?**

Enterprise application development is complex. You have multiple environments

* Dev
* QA
* Stage
* Production

You want to have different application configuration in each of the environments.

Spring and Spring Boot provide features where you can specify

* What is the configuration for various environments in different profiles?
* Set the active profile for a specific environment.

Spring Boot would pick up the application configuration based on the active profile that is set in a specific environment.

**Q : How can you use profiles to configure environment specific configuration with Spring Boot?**

Profile is nothing but a key to identify an environment.

We would want to customize the application.properties for dev profile. We would need to create a file with name application-dev.properties and override the properties that we would want to customize.

application-dev.properties

basic.message: Dynamic Message in DEV

Similarly you can configure properties for prod profile.

application-prod.properties

basic.message: Dynamic Message in Prod

Once you have profile specific configuration, you would need to set the active profile in an environment.

There are multiple ways of doing this

* Using -Dspring.profiles.active=prod in VM Arguments
* Use spring.profiles.active=prod in application.properties

**How to disable specific auto-configuration in spring boot?**

You can use exclude property as shown below to disable specific auto configuration

@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

* **What is Hot swapping in spring boot?**

Reloading the changes without restarting the server is called hot swapping, Modern IDEs (Eclipse, IDEA, etc.) all support hot swapping of bytecode,  so if you make a change that doesn’t affect class or method signatures it should reload cleanly with no side effects.

* **How do you Switch off the Spring Boot security configuration?**

If you define a @Configuration with @EnableWebSecurity anywhere in your application it will switch off the default webapp security settings in **Spring Boot**.

**Custom Endpoint using Spring Boot 1.x**

To implement a new endpoint for our application using Spring Boot 1.x, we should expose the instance of the custom endpoint class as a bean.We need to implement Endpoint<T> interface.

@Component

public class CustomEndpoint implements Endpoint {

@Override

public String getId() {

return "custom-endpoint";

}

@Override

public boolean isEnabled() {

return true;

}

@Override

public boolean isSensitive() {

return false;

}

@Override

public String invoke() {

return "This is a custom end point for demo purpose";

}

}

To access our custom endpoint, use the id field (for our example, it is “*custom-endpoint*“).

{

This is a custom end point for demo purpose

}

**Custom Endpoint with Spring Boot 2.x**

Spring Boot 2 provides an easy way to create custom endpoints.Spring Boot 2.x introduced @Endpoint annotation.Spring Boot automatically expose endpoints with@Endpoint, @WebEndpoint, or @WebEndpointExtension over HTTP using Jersey, Spring MVC, or Spring WebFlux.

Spring Boot 2.x Actuator support CURD model, it supports read, writes and delete operation with the endpoints.The @Endpoint annotation can be used in combination with @ReadOperation,@WriteOperation and @DeleteOperation to develop endpoints.

2.1 Creating Custom Endpoint

We are creating a custom health endpoint, this endpoint will provide a custom information to the client.

@JsonInclude(JsonInclude.Include.NON\_EMPTY)

public class CustomHealth {

private Map<String, Object> healthDetails;

@JsonAnyGetter

public Map<String, Object> getHealthDetails() {

return this.healthDetails; }

}

**Custom Health endpoint.**

@Component

@Endpoint(id="custom-health")

public class CustomHealthEndPoint {

@ReadOperation

public CustomHealth health() {

Map<String, Object> details = new LinkedHashMap<>(); details.put("CustomHealthStatus", "Everything looks good"); CustomHealth health = new CustomHealth(); health.setHealthDetails(details);

return health;

}

@ReadOperation

public String customEndPointByName(@Selector String name) { return "custom-end-point";

}

@WriteOperation

public void writeOperation(@Selector String name) {

//perform write operation

}

@DeleteOperation

public void deleteOperation(@Selector String name){ //delete operation }

}

* The Id property of the @Endpoint annotation determines the mapping of our endpoint (in our example it is /custom-endpoint).
* @ReadOperation – HTTP Get method.
* @WriteOperation – POST method.
* @DeleteOperation – HTTP DELETE operation.

To access our custom endpoint, use *http://host:port/actuator/custom-health* to check the output.

{ "CustomHealthStatus":"Everything looks good" }

2.2 Controller Endpoints

Spring Boot Actuator provides an alternate way to create custom endpoints that are only exposed by Spring MVC or Spring WebFlux.Use @ControllerEndpoint and @RestControllerEndpoint for this.While using this approach, we should use standard Spring MVC annotations like @RequestMapping and @GetMapping, with the endpoint’s ID being used as a prefix for the path.

@Component

@RestControllerEndpoint(id = "rest-end-point")

public class RestCustomEndPoint {

@GetMapping("/custom")

public @ResponseBody ResponseEntity customEndPoint(){ return new ResponseEntity<>("REST end point", HttpStatus.OK); }

}

Use *http://host:port/actuator/rest-end-point/custom* to access this custom endpoint.

## Introduction

Filters as the name suggest use to perform filtering on either the request to a resource or on the response from a resource, or both. Spring Boot provides few options to register custom filters in the Spring Boot application. Let’s take a look at the different options.

## 1. Define Spring Boot Filter and Invocation Order

Simply implement *Filter* interface to create a new filter in Spring Boot.

@Configuration

@Order(Ordered.HIGHEST\_PRECEDENCE)

public class CustomFilter implements Filter {

private static final Logger LOGGER = LoggerFactory.getLogger(CustomFilter.class);

@Override public void init(FilterConfig filterConfig) throws ServletException {

LOGGER.info("########## Initiating Custom filter ##########"); }

@Override public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest;

HttpServletResponse response = (HttpServletResponse) servletRequest;

LOGGER.info("Logging Request {} : {}", request.getMethod(), request.getRequestURI()); //call next filter in the filter chain filterChain.doFilter(request, response); LOGGER.info("Logging Response :{}", response.getContentType()); }

@Override public void destroy() { // TODO: 7/4/18 } }

Let’s quickly take a look at some of the important points in the above code

* The filter registered by @Component annotation.
* To fire filters in the right order – we needed to use the @Order annotation.

Highest order filter run first. This is really useful when we want to execute our custom filters on predefined order.

@Component

@Order(1)

public class CustomFirstFilter implements Filter { } @Component

@Order(2) public class CustomSecondFilter implements Filter { }

In the above code, CustomFirstFilter will run before the CustomSecondFilter.

*The lower the number, the higher the precedence*

## 2. URL Pattern

If the convention-based mapping is not flexible enough, we can use FilterRegistrationBean for the complete control of the application. In this case, don’t use @Component annotation for the filter class but register the filter using a FilterRegistrationBean.

public class CustomURLFilter implements Filter {

private static final Logger LOGGER = LoggerFactory.getLogger(CustomURLFilter.class);

@Override public void init(FilterConfig filterConfig) throws ServletException {

LOGGER.info("########## Initiating CustomURLFilter filter ##########");

}

@Override public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest; HttpServletResponse response = (HttpServletResponse) servletResponse;

LOGGER.info("This Filter is only called when request is mapped for /customer resource");

//call next filter in the filter chain

filterChain.doFilter(request, response);

}

@Override public void destroy() { } }

Register the custom Filter using FilterRegistrationBean.

@Configuration public class AppConfig {

@Bean public FilterRegistrationBean < CustomURLFilter > filterRegistrationBean() {

FilterRegistrationBean < CustomURLFilter > registrationBean = new FilterRegistrationBean();

CustomURLFilter customURLFilter = new CustomURLFilter(); registrationBean.setFilter(customURLFilter); registrationBean.addUrlPatterns("/greeting/\*"); registrationBean.setOrder(2); //set precedence return registrationBean; }

}