**SPRING MISC**

**What is CommandLineRunner in Spring ?**

CommandLineRunner is an interface used to create a simple console-based application using Spring Boot. This interface provides access to application arguments as string array. CommandLineRunner is a simple Spring Boot interface with a run method. Spring Boot will automatically call the run method of all beans implementing this interface after the application context has been loaded. Note that the run method is called after application context is loaded but before the execution of the main method is complete.

Most console applications will only have a single class that implements CommandLineRunner. If your application has multiple classes that implement CommandLineRunner, the order of execution can be specified using Spring's @Order annotation. You can also selectively filter some CommandLineRunners using active/inactive Spring Profiles (@ActiveProfiles annotation).

We should also specify the spring.main.web-application-type=NONE Spring property. This property will explicitly inform Spring that this isn't a web application.

***@Component***

***public class CommandLineAppStartupRunner implements CommandLineRunner {***

***private static final Logger logger = LoggerFactory.getLogger(CommandLineAppStartupRunner.class);***

***@Override***

***public void run(String...args) throws Exception {***

***logger.info("Application started with command-line arguments: {} . \n To kill this application, press Ctrl + C.", Arrays.toString(args));***

***}***

***}***

Typical scenario where CommandLineRunner can be used is to source data from other microservices via service discovery, before the execution of the main method completes.

**What is ApplicationRunner in Spring ?**

ApplicationRunner wraps the raw application arguments and exposes the ApplicationArguments interface, which has many convinent methods to get arguments, like getOptionNames() to return all the arguments' names, getOptionValues() to return the agrument value, and raw source arguments with method getSourceArgs().

***@Component***

***public class AppStartupRunner implements ApplicationRunner {***

***private static final Logger logger = LoggerFactory.getLogger(AppStartupRunner.class);***

***@Override***

***public void run(ApplicationArguments args) throws Exception {***

***logger.info("Your application started with option names : {}", args.getOptionNames());***

***}***

***}***

**What is JobLauncherCommandLineRunner in Spring ?**

org.springframework.boot.autoconfigure.batch.JobLauncherCommandLineRunner implements CommandLineRunner to register and start batch jobs at application startup (spring-batch).

**What is RetryTemplate in Spring retry?**

***RetryOperations***

Spring Retry provides RetryOperations interface which supplies a set of execute() methods:

***public interface RetryOperations {***

***<T> T execute(RetryCallback<T> retryCallback) throws Exception;***

***...***

***}***

The RetryCallback which is a parameter of the execute() is an interface that allows insertion of business logic that needs to be retried upon failure:

***public interface RetryCallback<T> {***

***T doWithRetry(RetryContext context) throws Throwable;***

***}***

***RetryTemplate Configuration***

The RetryTemplate is an implementation of the RetryOperations. Let's configure a RetryTemplate bean in our @Configuration class:

***@Configuration***

***public class AppConfig {***

***//...***

***@Bean***

***public RetryTemplate retryTemplate() {***

***RetryTemplate retryTemplate = new RetryTemplate();***

***FixedBackOffPolicy fixedBackOffPolicy = new FixedBackOffPolicy();***

***fixedBackOffPolicy.setBackOffPeriod(2000l);***

***retryTemplate.setBackOffPolicy(fixedBackOffPolicy);***

***SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();***

***retryPolicy.setMaxAttempts(2);***

***retryTemplate.setRetryPolicy(retryPolicy);***

***return retryTemplate;***

***}***

***}***

RetryPolicy determines when an operation should be retried. A SimpleRetryPolicy is used to retry a fixed number of times.

BackOffPolicy is used to control back off between retry attempts. A FixedBackOffPolicy pauses for a fixed period of time before continuing.

***Using the RetryTemplate***

To run code with retry handling we call the retryTemplate.execute():

***retryTemplate.execute(new RetryCallback<Void, RuntimeException>() {***

***@Override***

***public Void doWithRetry(RetryContext arg0) {***

***myService.templateRetryService();***

***...***

***}***

***});***

The same could be achieved using a lambda expression instead of an anonymous class:

***retryTemplate.execute(arg0 -> {***

***myService.templateRetryService();***

***return null;***

***});***

**What is Listener callback in Spring retry?**

Listeners provide additional callbacks upon retries. They can be used for various cross-cutting concerns across different retries.

***Adding Callbacks***

The callbacks are provided in a RetryListener interface:

***public class DefaultListenerSupport extends RetryListenerSupport {***

***@Override***

***public <T, E extends Throwable> void close(RetryContext context,***

***RetryCallback<T, E> callback, Throwable throwable) {***

***logger.info("onClose);***

***...***

***super.close(context, callback, throwable);***

***}***

***@Override***

***public <T, E extends Throwable> void onError(RetryContext context,***

***RetryCallback<T, E> callback, Throwable throwable) {***

***logger.info("onError");***

***...***

***super.onError(context, callback, throwable);***

***}***

***@Override***

***public <T, E extends Throwable> boolean open(RetryContext context,***

***RetryCallback<T, E> callback) {***

***logger.info("onOpen);***

***...***

***return super.open(context, callback);***

***}***

***}***

The open and close callbacks come before and after the entire retry, and onError applies to the individual RetryCallback calls.

***Registering the Listener***

Next, we register our listener (DefaultListenerSupport) to our RetryTemplate bean:

***public class AppConfig {***

***...***

***@Bean***

***public RetryTemplate retryTemplate() {***

***RetryTemplate retryTemplate = new RetryTemplate();***

***...***

***retryTemplate.registerListener(new DefaultListenerSupport());***

***return retryTemplate;***

***}***

***}***

**How Spring security works by default without any Security configuration?**

In order to add security to our Spring Boot application, we need to add the security starter dependency.

This will include the SecurityAutoConfiguration class – containing the initial/default security configuration.

When Spring-boot-starter-security dependency is added, we get default generated security password in console logs on server startup.

default username is 'user' for generated password.

Simply put, by default, the Authentication gets enabled for the Application. Also, content negotiation is used to determine if basic or formLogin should be used.

Default username and password can be configured by adding below properties in application.properties

spring.security.user.name=username

spring.security.user.password=password

Some of the endpoints that were unsecured by default in Spring Boot 1 are now secured by default in Spring 2.

These endpoints include static resources such as /css/\*\*, /js/\*\*, /images/\*\*, /webjars/\*\*, /\*\*/favicon.ico, and the error endpoint. If we need to allow unauthenticated access to these endpoints, we can explicitly configure that.

***Disabling the Auto-Configuration:***

To discard the security auto-configuration and add our own configuration, we need to exclude the SecurityAutoConfiguration class.

This can be done via a simple exclusion:

***@SpringBootApplication(exclude = { SecurityAutoConfiguration.class })***

***public class SpringBootSecurityApplication {***

***public static void main(String[] args) {***

***SpringApplication.run(SpringBootSecurityApplication.class, args);***

***}***

***}***

Or by adding some configuration into the application.properties file:

***Spring.autoconfigure.exclude= = org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration***

There are also some particular cases in which this setup isn't quite enough.

For example, almost each Spring Boot application is started with Actuator in the classpath. This causes problems because another auto-configuration class needs the one we've just excluded, so the application will fail to start.

In order to fix this issue, we need to exclude that class; and, specific to the Actuator situation, we need to exclude ManagementWebSecurityAutoConfiguration.

**How properties file for application configuration work in Spring and Spring Boot?**

***Register a Properties File via Java Annotations -***

The new @PropertySource annotation, as a convenient mechanism for adding property sources to the environment. This annotation is to be used in conjunction with Java based configuration and the @Configuration annotation.

***@Configuration***

***@PropertySource("classpath:foo.properties")***

***public class PropertiesWithJavaConfig {***

***//...***

***}***

One other very useful way of registering a new properties file is using a placeholder to allow you to dynamically select the right file at runtime; for example:

***@PropertySource({***

***"classpath:persistence-${envTarget:mysql}.properties"***

***})***

The @PropertySource annotation is repeatable according to Java 8 conventions. Therefore, if we're using Java 8 or higher, we can use this annotation to define multiple property locations:

***@PropertySource("classpath:foo.properties")***

***@PropertySource("classpath:bar.properties")***

***public class PropertiesWithJavaConfig {***

***//...***

***}***

We can also use the @PropertySources annotation and specify an array of @PropertySource. This works in any supported Java version, not just in Java 8 or higher:

***@PropertySources({***

***@PropertySource("classpath:foo.properties"),***

***@PropertySource("classpath:bar.properties")***

***})***

***public class PropertiesWithJavaConfig {***

***//...***

***}***

In either case, it's worth noting that in the event of a property name collision, the last source read takes precedence.

*Injecting a property with the @Value annotation is straightforward:*

***@Value( "${jdbc.url}" )***

***private String jdbcUrl;***

A default value of the property can also be specified:

***@Value( "${jdbc.url:aDefaultUrl}" )***

***private String jdbcUrl;***

we can obtain the value of a property using the Environment API:

***@Autowired***

***private Environment env;***

***...***

***dataSource.setUrl(env.getProperty("jdbc.url"));***

***Properties with Spring Boot -***

In Spring boot, application.properties is the Default Property File in “src/main/resources” directory, and it will be auto-detected.

We can also configure a different file at runtime if we need to, using an environment property:

java -jar app.jar --spring.config.location=classpath:/another-location.properties

***Environment Specific Properties File***

We can simply define an “application-environment.properties” file in the “src/main/resources” directory – and then set a Spring profile with the same environment name. This env file will be loaded and will take precedence over the default property file. Note that the default file will still be loaded, it's just that when there is a property collision the environment specific property file takes precedence.

***Test Specific Properties File***

Spring Boot handles this for us by looking in our “src/test/resources” directory during a test run. Again, default properties will still be injectable as normal but will be overridden these ones if there is a collision.

If we need more granular control over test properties, then we can make use of the @TestPropertySource annotation.

This allows us to set test properties for a specific test context, taking precedence over the default property sources:

***@ContextConfiguration***

***@TestPropertySource("/my-test.properties")***

***public class IntegrationTests {***

***// tests***

***}***

If we don't want to use a file, we can specify names and values directly:

***@ContextConfiguration***

***@TestPropertySource("foo=bar", "bar=foo")***

***public class IntegrationTests {***

***// tests***

***}***

We can also achieve a similar effect using the properties argument of the @SpringBootTest annotation:

***@SpringBootTest(properties = {"foo=bar", "bar=foo"})***

***public class IntegrationTests {***

***// tests***

***}***

***Hierarchical Properties***

If we have properties which are grouped together, we can make use of the @ConfigurationProperties annotation which will map these property hierarchies into Java objects graphs.

Let's take some properties used to configure a database connection:

database.url=jdbc:postgresql:/localhost:5432/instance

database.username=foo

database.password=bar

And then let's use the annotation to map them to a database object:

***@ConfigurationProperties(prefix = "database")***

***public class Database {***

***String url;***

***String username;***

***String password;***

***// standard getters and setters***

***}***

Spring Boot applies it's convention over configuration approach again, automatically mapping between property names and their corresponding fields. All that we need to supply is the property prefix.

***Alternative – YAML Files***

YAML files are also supported.

All the same naming rules apply for test specific, environment specific, and default property files. The only difference is the file extension, and a dependency on the SnakeYAML library being on your classpath.

YAML is particularly good for hierarchical property storage; the following property file:

database.url=jdbc:postgresql:/localhost:5432/instance

database.username=foo

database.password=bar

secret: foo

Is synonymous to the following YAML file:

database:

url: jdbc:postgresql:/localhost:5432/instance

username: foo

password: bar

secret: foo

It's also worth mentioning that YAML files do not support the @PropertySource annotation, so if use of the annotation was required it would constrain us to using a properties file.

***Properties from Command Line Arguments -***

As opposed to using files, properties can be passed directly on the command line:

java -jar app.jar --property="value"

You can also do this via system properties, which are provided before the -jar command rather than after it:

java -Dproperty.name="value" -jar app.jar

***Properties from Environment Variables***

Spring Boot will also detect environment variables, treating them as properties:

export name=value

java -jar app.jar

***Randomization of Property Values***

If we don't want determinist property values, RandomValuePropertySource can be used to randomise the values of properties:

random.number=${random.int}

random.long=${random.long}

random.uuid=${random.uuid}

Besides the convenient methods of getting properties into Spring – annotations and the XML namespace – the property configuration bean can also be defined and registered manually. Working with the PropertySourcesPlaceholderConfigurer gives us full control over the configuration, with the downside of being more verbose and most of the time, unnecessary.

**How to tell Spring Boot to look for myapp.properties instead of application.properties?**

***Command line arguments-***

Use command line argument "--spring.config.name" to pass the renamed file name to the Spring Boot application as shown below:

java -jar myAppBuild.jar --spring.config.name=myapp

***Environment variables-***

In the second approach, you can configure your externalized configuration details into environment variables and your Spring Boot application will read it from your environment as shown below:

set SPRING\_CONFIG\_NAME=myapp

java -jar myAppBuild.jar

***Programatically loading configurations-***

***@SpringBootApplication***

***public class SpringBootWebApplication {***

***public static void main(String[] args) throws Exception {***

***ConfigurableApplicationContext applicationContext = new SpringApplicationBuilder(SpringBootWebApplication.class)***

***.properties("spring.config.name:myapp")***

***.build()***

***.run(args);***

***}***

***}***

**OR**

***@SpringBootApplication***

***public class MyApplication {***

***public static void main(String[] args) {***

***System.setProperty("spring.config.name", "myapp");***

***SpringApplication.run(MyApplication.class, args);***

***}***

***}***