Java Advanced – lesson 4 – Spring Boot tips & tricks

Wim Wiltenburg (willem.wiltenburg@inholland.nl)

Last week & questions

- @OneToOne

Objectives

- Properties
- Logging
- Lombok
- Maven profiles
- Jersey
- Developer tools

Properties

- When building an application we sometimes need to configure specific behavior or set specific defaults.
- We can configure our application in property files in src/main/resources
- There are different formats:
  - .properties files – flat file in format: {key} = {value}
    Example:
    ```
    spring.jpa.show-sql=false
    ```
  - .yml (yaml) files: flat file in tree format.
    Example:
    ```
    spring:
      jpa:
        show-sql: false
    ```

Create our own properties

We can define our own properties if we want

E.g.

```
shop:
  quantity: 5
```

We create a configuration class, that reads the property file, and adds the properties to the Spring Container

```
@Configuration
@EnableConfigurationProperties
@ConfigurationProperties(prefix = "shop")
public class Config {

    private int quantity;

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public int getQuantity() {
        return quantity;
    }
}
```

Using the property

We wire the Config object to whatever object that needs these properties, e.g. ApplicationRunner

```
@Component
public class MyApplicationRunner implements ApplicationRunner {

    private GuitarRepository guitarRepository;
    private StockRepository stockRepository;
    private Config config;

    public MyApplicationRunner(GuitarRepository guitarRepository, StockRepository stockRepository, Config config)
{
        this.guitarRepository = guitarRepository;
        this.stockRepository = stockRepository;
        this.config = config;
    }
```

Then we can use it where we want it:

```
List<Guitar> guitars = (List<Guitar>) guitarRepository.findAll();
guitars.stream()
        .forEach(a -> stockRepository.save(new Stock(a, config.getQuantity())));
```

Logging

A log is a file that contains structured data about what goes on in an application.

Every line should have a minimum of:

- Timestamp
- Data

```
2019-05-11 16:39:29,524 INFO  [main] o.s.b.w.e.tomcat.TomcatWebServer
TomcatWebServer: Tomcat initialized with port(s): 8080 (http)
```

We all know that's not always the case

(see stacktrace)

Logger

Java has its own logging framework in java.util.logging. At its basic level our application obtains a logging instance on our class:

```java
@SpringBootApplication
public class App
{
    private static final Logger logger = Logger.getLogger(App.class.getName());

    public static void main( String[] args ) {
        logger.log(Level.INFO, "Starting Jack's Guitar Shop application...");
        SpringApplication.run(App.class, args);
    }
}
```

The result:

```
mei 12, 2019 12:28:56 PM nl.inholland.App main
INFO: Starting Jack's Guitar Shop application...
```

This does not particularly look very good, but it's preferable to System.out.println()

Logback

Spring Boot standard includes logging framework Logback. It is configured in logback(-xxx).xml in src/main/resources and/or src/test/resources

Logback has a number of components:

- Appenders – defines log destination (e.g. console, or file system)
- Loggers – interacts with the class that logs
- Layout – formats the log message

Log levels

In logback there are 5 levels of logging. They are (sorted to more verbose to less verbose):

- TRACE
- DEBUG
- INFO
- WARN
- ERROR

The level can be set from root level (all logging) to a specific class overriding the root level

Example configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d %-5level [%thread] %class{36}: %msg%n</pattern>
        </encoder>
    </appender>

    <logger name="org.hibernate" level="error">
            <appender-ref ref="STDOUT" />
    </logger>

    <root level="info">
        <appender-ref ref="STDOUT"/>
    </root>

</configuration>
```

Here we define that the logs will not contain messages higher than INFO, that means no DEBUG or TRACE. For classes in package org.hibernate only ERROR will be logged; no WARN, INFO, DEBUG or TRACE. This comes in handy if we want to filter unwanted messages, or chatter in logs

Appenders

An appender defines the destination of the log. This is generally a local log file, or the console, but can be extended to a wide variety of destinations. Examples are:

- Remote logs over the network
- E-mail (STMP)
- Self written

Extensive configuration can be done, e.g. for RollingFileAppenders, which can rotate, based on specific parameters

Formatting the log lines

With an appender comes an encoder, that will format each log line according to a pattern. In our example we define:

- %d – timestamp formatted YYYY-MM-DD HH:mm:ss,SSS
- %5-level – the loglevel, e.g. INFO
- [%thread] – The name of the current thread, e.g. main (enclosed in square brackets_
- %class – the logging class
- %m%n – The message and a new line

Lombok

Project Lombok is an open source code generation project, designed to take away boilerplate code, like generic getters and setters, and logging.

It uses annotations, to create code “under water”, that will end up in the compiled project.

It can be added to the project with Maven:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
```

Note: if you expect it to work in your IDE, install the lombok plugin

Lombok annotations

Common lombok annotations are:

- @Log – will generate private static final Logger (java.util.logging). Other loggers are available
- @Getter – will generate generic getters for all fields
- @Setter – will do the same for setters. Default setters can be overridden, e.g. for id.
- @Builder – will generate a builder pattern
- @ToString – seeing the pattern here?
- @Data - @Getter + @Setter + @ToString, and more

Lombok example

```
@Entity
@Table(uniqueConstraints = {@UniqueConstraint(columnNames = {"brand", "model"})})
@Getter
@Setter
@NoArgsConstructor
@ToString
@Log
public class Guitar{

    @Id
    @SequenceGenerator(name="guitar_seq", initialValue = 1000001)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "guitar_seq")
    long id;
    String brand;
    String model;
    int price;

    public Guitar(String brand, String model, int price) {
        this.brand = brand;
        this.model = model;
        this.price = price;
    }

    public void setId(long id) {
        log.warning("Method setId called, but ignored");
    }
}
```

Maven profiles

It's possible that our application should at one time behave differently than other times, depending on:

- Operating System
- Environment (DTAP)
- Dependency set – e.g. Functional Tests
- External properties

For this situations we can define different profiles in Maven, where we can configure our application for that target situation

Example: a tale of two databases

During our developing process, we would like to compare databases for our applications. We define two profiles: h2 and derby:

```
<profiles>
  <profile>
    <id>h2</id>
    <dependencies>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
  </profile>
  <profile>
    <id>derby</id>
    <dependencies>
      <dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derby</artifactId>
      </dependency>
    </dependencies>
  </profile>
</profiles>
```

Profile configuration properties

These two profiles are inside the <profiles> tag. Each profile has an ID and a different dependency.

If we run the H2 profile, the application will behave properly, because H2 doesn’t need (but can have) additional configuration.

Derby is not that forgiving. It needs configuration, and if we run that profile, it will not start up.

We configure that application in application-<profile-id>.properties (or .yml)

Derby configuration

Derby needs at a minimum:

- Database Type:
  - Hibernate DDL definition
- Datasource:
  - url: e.g. jdbc:derby:jacks;create=true
  - driverClassName: org.apache.derby.jdbc.EmbeddedDriver
  - Username/password.

Derby can also run externally, so different drivers are needed for that occasion. The EmbeddedDriver is needed when running in memory

Example: application-derby.yml

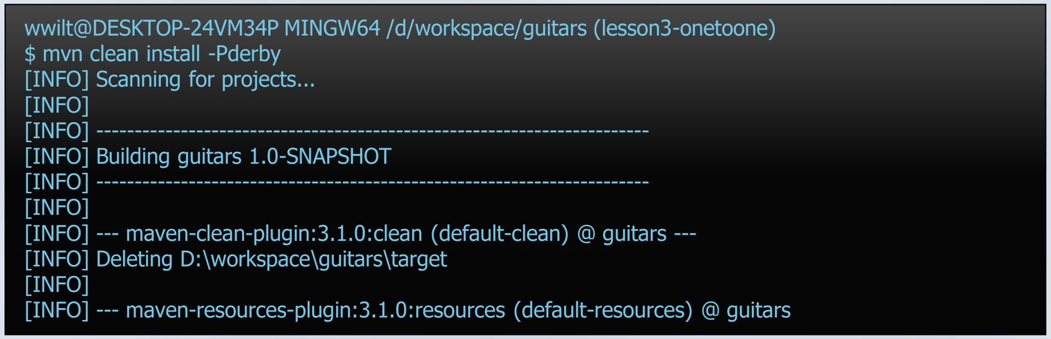
```yaml
spring:
  jpa:
    database: DERBY
    show-sql: true
    hibernate:
      ddl-auto: create-drop

  datasource:
    platform: derby
    url: jdbc:derby:jacks;create=true
    username: sa
    password:
    driverClassName: org.apache.derby.jdbc.EmbeddedDriver
```

Working with the profile

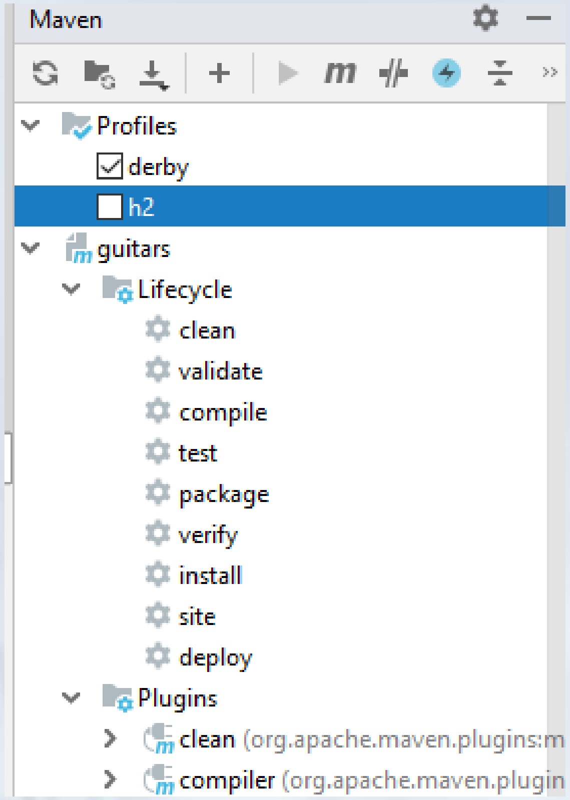
In IntelliJ we can choose the profile in the Maven tab:

We can build a jar file by clicking the clean and install goals, and running it. It will build a jar with the correct dependencies

On the commandline we type:



```
wwilt@DESKTOP-24VM34P MINGW64 /d/workspace/guitars (lesson3-onetoone)
$ mvn clean install -Pderby
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building guitars 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ guitars ---
[INFO] Deleting D:\workspace\guitars\target
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ guitars
```



Default profile

When we want to make one profile default, we can add that to our profile configuration:

```
<profile>
  <id>h2</id>
  <dependencies>
      <dependency>
          <groupId>com.h2database</groupId>
          <artifactId>h2</artifactId>
          <scope>runtime</scope>
      </dependency>
  </dependencies>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
</profile>
```

If we now type mvn clean install, then by default the h2 profile will be built.

Jersey

JAX-RS (or Jersey) is a webcontainer that adheres a little closer to the REST API standards, than the default Spring container, also called Spring MVC.

It uses the namespace javax.ws.rs as opposed to org.springframework.web, and different annotations. To start using Jersey we need to change the dependency from:

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

to

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

Registering controllers

In Jersey controllers are not annotated as @RestController, but as mere @Component. Jersey does not automatically recognize these components as controllers. Therefore we need to register them.

```
@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(GuitarController.class);
        register(StockController.class);
    }
}
```

We use @Component to let the Spring container scan this at startup. The ResourceConfig is a Jersey class that among others allows us to say that certain classes are controllers.

JAX-RS annotations

The controller itself has different annotations. Apart from just @Component we specify our url with @Path, and what we produce with @Produces. Get requests are annotated with @Get, Postrequests with @Post, etc.

Example:

```
@Component
@Path("/guitars")
@Produces(MediaType.APPLICATION_JSON_VALUE)
public class GuitarController {


    @GET
    @Path("")
    public Iterable<Guitar> getAllGuitars() {
        return service.getAllGuitars();
    }
```

Some remarks about JAX-RS

- If the controller has no javax.ws.rs annotations (like @GET, @POST, etc.) then the registering of the class just gets ignored
- Proper REST API return codes (HTTP 204, etc.) are standard. No additional configuration like @ResponseStatus is needed
- It is allowed in the final project, but research what you're doing. It could be a little more complicated than Spring MVC. Don't expect everything to work out-of-the-box

Developer Tools

When actively coding, it can be irksome to write something, reload, write something, reload and repeat. Spring Developer Tools to the rescue. It uses a separate class loader for classes on the classpath that are actively changing and reload them. Comes in handy when developing frontend, too.

Add dependency:

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

Configuring developer tools

For developer tools to work we must:

- Update the IntelliJ Registry. Enter ctrl + shift + a. Lookup Registry… and find entry: compiler.automake.allow.when.app.running. Enable that entry.
- Goto settings > Build, Execution, Deployment > Compiler. Enable “Build project automatically”

Homework