My Spring-Boot Notes

From O'Reilly and Manning books.

Core Spring

Spring is a web dev framework. It revolves around Aspect Oriented Programming. AOP is like a blanket where you write your code and then you tie your code with other modules like logging, security, etc. This is a great way of programming.

You can the use XML, or JAVA or Autowiring, which is a great Dependency Injection way. In order to use autowiring you need to declare beans using the @Bean and @Configuration decorators.

Sometimes you want to have different Beans for different environments, development, testing and production. You can use the @Profile('dev') decorator in order to create a profile. Test can be set to run with a specific profile using @ActiveProfile('dev'). Profile can be used on classes as well as methods starting with spring 3.2

Beans creation can be conditional using @Conditional which requires a class that implements Condition.

When Spring can't find a unique bean it will throw an exception, in order to make spring get a bean use the @Primary decorator in order to indicate that the primary bean should be always used, but when there are two primary beans another exception pops out, in order to solve this, when using @Autowired use it together with @Qualifier("iceCream").

In Spring beans can have different scopes: Singleton, Prototype (uniq instance), Session (request session web attached to a single user), Request (single request) To configure a bean scope use: @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) Bean can also take the following arguments @Bean(initMethod="", destroyMethod="") and if you have access to the bean's source code you could create the methods yourself and anotate them wiht @PreConstruct, @PostDestroy.

In order to retrieve values from Spring's environment you can use the following methods:

```
String getProperty(String key)
String getProperty(String key, String defaultValue) : env.getProperty("disc.title",
"Rattle and Hum"),
T getProperty(String key, Class<T> type)
T getProperty(String key, Class<T> type, T defaultValue) :
env.getProperty("db.connection.count", Integer.class, 30);
```

Aspect Oriented Programming

AOP has the following terminology:

ADVICE - The job on an aspect, the what and when.

Five kinds of advices:

- Before: Before the execution of a method.
- After: After the execution of a method.

- Around: Before and after the execution of a method.
- After-Returning: After the successfull execution of a method.
- After-Throwing: After the not successfull execution of a mehtod.

JOIN POINT - An opportunity of where an advice should be applied.

POINT CUTS - Points cuts help narrow the JOIN POINTS, like a condition

ASPECT - This is a merger of advice and point cuts.

INTRODUCTION - This helps you adding new methods or attributes to a class.

WEAVING - This is the process of applying aspects to a target object to create new proxied object.

- Compile Time: requires special compiler
- Class Load Time: class loaded into jvm
- Runtime: at the runtime, dinamically generates a proxy object that delegates to the target object.

Configuration

By using @EnableAutoConfiguration springs brings in a lot of dependencies that check if other classes are present using the @Condition annotation.

@SpringBootApplication is a shortcut for @Configuration @EnableAutoConfiguration @ComponentScan

To import an configuration, if you don’t use component scan use @Import(Myconfiguration.class)

Application Properties

You an use application properties in order to apply custom configuration to your Spring Boot application, for example changing the port number or the base path. You can also create custom properties and refference them in code by using the @Value("${property_name}") annotation. Values can also be random numbers. ${random.int}

Profiles

Profilings are used to configure at runtime which methods to run based on what profiles. @Profile("profile_name"), you can set the active profile using the following property: spring.profiles.active="dev" and you could use --spring.profiles.active=prod as an argument when running the program.

You can also remove the @Profile annotation and create a new "application-prod.properties" file, this will determine which values from there the program will used, based on the active profile, and the "application.properties" file will remain the default fallback option.

A third option is to use the @ConfigurationProperties(prefix="my") java annotation on the class,

which let's you specify a prefix for the properties that should be used, for example: my.messageValue=secret will be injected into the the annotated class, the last step is to use @EnableConfigurationProperties(value=MyMessage.class) to enable property injection for the selected class. Classes annotated /w @Component don't need to be registered.

If you have a string `messageValue` the properties must follow the naming convetions: messageValue, MESSAGE_VALUE or message-value.

Executing Code at Startup

Application Runner and Command Line Runner

```
@Component
public class MyCommandLineRunner implements CommandLineRunner {
  @Override
  public void run(String... args) throws Exception {
    // Code that shall be run at runtime
  }
}
```

The difference is that we don't get the args as a string, we get them from the application context.

```
@Component
public class MyApplicationRunner implements ApplicationRunner {
  @Override
  public void run(String... args) throws Exception {
    // Code that shall be run at runtime
  }
}
```

ApplicationArguments can be injected in any bean.

Template Support

Springs makes it easy to use multiple template engines, all you need to do is configure them, usually using the app properties file. This is a trivial task and it is easily accomplished.

Caching

Spring Boot and MVC provides us /w an easy mechanism to control the caching issues. We just need to use the application properties:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
```

This will add a hash to the filename and when the contents change, the 1 year cache is invalidated. There's another way to do this with the `strategy.fixed` option, this applies a fixed version number to the files, needs constant updating. This is usefull when you can't change the filename of a client resource.

```
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/**
spring.resources.chain.strategy.fixed.version=v5
```

Embedded Container Configuration

The Spring Web Starter uses Tomcat by default, this is a dependecy. If we want to use somethign else, we can, in eclipse to exclude the maven artifact. Of course if you don't have Eclipse, you can just exclude the dependency by placing: `<exclusions>` tag in the xml node of Spring Boot Web Starter. All you need to do now, is to add a dependency in the starter pom. We can also provide a number of properties that customizes our server controller, like the address and port. To enable compression we can add additional properties.

```
server.address=titan
server.port=8080
server.context-path=/spring

server.compression.enabled=true
server.compression.min-response-size=0
```

Spring boot has support for Tomcat, Jetty and Undertoe. :)

Custom Servlet

Sometimes we need custom servlets that do request processing for specific use cases. We can create a class `MyFilter implements Filter` from the Java Servlet interface. Don't forget to register the filter as a `@Component` in order to be discovered by Spring Boot.

Another way is to use the @WebFilter Java EE annotation and @ServletCompoenentScan annotation on our Spring Boot Application class.

Programatic Configuration

We can override Spring's EmbeddedServletContainerFactory Bean by creating a `@Configuration` class and a bean:

```
@Bean
public EmbeddedServletContainerFactory factory() {
  TomcatEmbeddedServletContainerFactory factory = new
TomcatEmbeddedServletContainerFactory();
  factory.setContextPath("/app"); // sets the serve path to /app
  return factory;
}
```

Spring security

By default all endpoints will be protected by http-basic authentication but this is nok. We need to override the default Spring Security authentication in order to get an usable application. By default, the password will be logged everytime the app starts to STDOUT. The dafault username is: user.

Spring Security will ignore by default a CSS directory, JS directory and Images directory.

To customize this, we can use `security.user.name` and `security.user.password` in the application.properties file.

Spring Security publishes several events to indicate key security events. To listen to those events you need to create a class that implements `ApplicationListener<AbstractAuthenticationFailureEvent>` then implements the method provided by the interface. You can print the `event.getException().getMessage()`. Be carefull, as this will only listen to the AbstractAuthenticationFailureEvent event.

To configure Spring Seucurity you need to extend the `WebSecurityConfigureAdapter` then you can override the methods that you may see fit. The class must then be anotated with @EnableWebSecurity. For example:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests().anyRequest().authenticated().and().httpBasic();
}
```

Relational Database Support

To add support for relational databases add the following dependencies: Web, H2 and JDBC JDBC templates is like simplified SQL, I don't like it. I'm only going to use ORM like methods.

H2 Console is usefull when debugging the database, the console is browser based and can be exposed by setting somethign properties in the application.properties: `spring.h2.console.enabled=true`. The console can be found at /h2-console Using the `spring.datasource.url=jdbc:h2:~/test` property we can specify the path to our database.

To use a MySQL Database we need to add the dependency mysql-connector.java dependecy. The mysql db will run as a service in the background. To connect to the mysql db use:

```
spring.datasource.url=jdbc:mysql:localhost/test_schema
spring.datasource.user=test_user
spring.datasource.password=test_password
spring.datasource.driver-class-name=com.mysql.jdbc.driver
spring.datasource.schema=classpath:load.sql

spring.datasource.test-on-borrow=true
spring.datasource.validation-query=/* ping */
```

Spring Data

Spring Data makes working /w databases very easy. You only need to extend an class. Create a new interface: UserRepository extends JpaRepository<User.class, Long>{} → `JpaRepository<Entity.class, IdType>` Spring will then implement all the CRUD methods behind your back.

We can configure the several properties that are related to Spring Data.

If you want to work /w Mongo DB, you may find the Mongo DB starter quite usefull. There are several ways to work /w a MongoDB database, mongo raw db objects, mongo template and the monst convenient: Mongo Repository.

Caching Support

Spring Boot has support for different caching providers, like Ehcache, Redis, Guava, and JCache.

To get started all you need to do is add the `@CacheResult` annotation to the method that you want to be cached and the `@EnableCaching` to the SpringBootApplication class.

You also need to add the `cache-api` dependency to your pom.xml.

If you need to use a more powerful caching mechanism you need to add the appropriate dependecies. Ehcache for example needs a `ehcache.xml` file in the resource folder. The xml file can contain something like the following:

```
<ehcache>
  <cach name="price" maxEntriesLocalHeap="200"></cache>
</ehcache>
```

All that's left now is to add the name on the `@CacheResult(cacheName="price")` annotation.

You can also use a CacheManager bean, autowired in the class, in order to retrieve a cache value.

```
@Autowired
private CacheManager manager;

public double getPriceWithManager(String symbol) {
  Cache cache = manager.getCache("price");
  return Double.valueOf(cache.get(symbol).getObjectValue().toString());
}
```

Spring Boot Extras

Automatic Restarts are provided by Spring Boot! To use it you need to include the Dev Tools dependency. To dissable automatic reloading set the `spring.devtools.restart.enabled=false` property.

Livereload can be added with a browser extension.

To configure the logging behaviour you can use the application properties. We can also configure logback via xml.

```
logging.level.root=DEBUG
logging.level.org.springframework.web=DEBUG // rest is info
logging.file=myLog.log // set log file
logging.path=logs // set log directory
```

Spring Boot Accuator

After adding the Accuator dependency, navigate to /mappings in order to see every route that's mapped to the application.

- /trace can be used to trace latest http authorizeRequests
- /dump can be used to execute a thread dump

You can customize accuator by using properties:

```
endpoints.autoconfig.path=/ac - set autoconfig endpoint
endpoints.autoconfig.enabled=false - e/d autoconfig
management.context-path=manage - set accuator endpoints path
endpoints.health.sensitive=true - set health sensitive info about application

info.appname=your app name with spaces is ok
info.framework=spring boot
info.name=@project.artifactId@
```

You can customize the health indicator by creating a class `implements HealthIndicator`.

The info endpoint will display only information from app properties.

If you have configured logging to a logfile, you can access the log via the browser using the /logfile endpoint.

`spring-boot-starter-hateoas` gives us a bunch of discoverable links that proves a map for our rest api's. If we visit the /acuator we can see all rest endpoits.

Accuator is also very extensible, it allows us to create custom endpoits, if you need to create a custom endpoint you need to implement the `Endpoit<T>` interface. The endpoit ID will be the path of the endpoint. The endpoint needs to be a bean to be discovered, add the `@Component` annotation.

You can check the `/metrics` endpoint to check the metrics.

Next Steps

- Write code.
- Research traditional Spring configuration
- Expand skillset: AOP, ByteCode Manipulation, Scala?