Spring Boot (II)

Created by :

Sangeeta Joshi

Configuration classes

Spring Boot favors Java-based configuration:

- *it is possible to call SpringApplication.run() with an XML source,*
- *However recommended is that primary source is a **@Configuration class**.*
- *Usually the class defines **the main method** is also a good candidate as the primary **@Configuration**.*

Configuration classes

Importing additional configuration classes

- *No need to put all @Configuration into a single class.*
- *The @Import annotation can be used to import additional configuration classes.*
- *Alternatively, we can use @ComponentScan to automatically pick up all Spring components, including @Configuration classes.*

Importing XML configuration

- *If you absolutely must, use XML based configuration, still better to start with a @Configuration class.*
- *You can then use an additional @ImportResource annotation to load XML configuration files.*

Annotations

Spring Beans and dependency injection

We are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies.

If we structure your code as our application class in a root package, we can add @ComponentScan without any arguments*.*

All of your application components (@Component, @Service, @Repository, @Controller etc.) will be automatically registered as Spring Beans.

Auto-configuration

Spring Boot auto-configuration attempts to automatically configure our Spring application based on the jar dependencies that we have added.

For example, If HSQLDB is on classpath, and we have not manually configured any database connection beans, then it will auto-configure an in-memory database.

We need to opt-in to auto-configuration by adding the @EnableAutoConfiguration or @SpringBootApplication annotations to one of our @Configuration classes.

[Tip]

We should only ever add one @EnableAutoConfiguration annotation.

Generally recommended to add it to primary @Configuration class.

Gradually replacing auto-configuration

Auto-configuration is noninvasive:

at any point we can start to define our own configuration to replace specific parts of the auto-configuration.

(For example, if we add DataSource bean, the default embedded database support will back away.)

If we need to find out what auto-configuration is currently being applied, and why, start application with the ***--debug switch****.*

This will enable debug logs for a selection of core loggers and log an auto-configuration report to the console.

Annotations

Using @SpringBootApplication annotation :

Many Spring Boot developers always have their main class annotated with @Configuration, @EnableAutoConfiguration and @ComponentScan. Since these annotations are so frequently used together,Spring Boot provides a convenient @SpringBootApplication alternative.

The ***@SpringBootApplication*** annotation is equivalent to using:

  @Configuration

  @EnableAutoConfiguration and

  @ComponentScan

with their default attributes:

Annotations

```
package com.example.myproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // same as @Configuration
@EnableAutoConfiguration @ComponentScan
public class Application {

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

}
```

Using Spring Boot CommandLineRunner

Using Spring Boot CommandLineRunner

a Spring Boot Interface called CommandLineRunner:

With CommandLineRunner we can perform tasks after all Spring Beans are created and the Application Context has been created.

From the Spring Boot Documentation:

If you want access to the raw command line arguments, or you need to run some specific code once the SpringApplication has started you can implement the CommandLineRunner interface.

The run(String… args) method will be called on all Spring beans implementing this interface.

You can additionally implement the @Ordered interface if several CommandLineRunner beans are defined that must be called in a specific order.

Using Spring Boot CommandLineRunner

```
@Component
public class ApplicationLoader implements CommandLineRunner {

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

    @Override
    public void run(String... strings) throws Exception {
        StringBuilder sb = new StringBuilder();
        for (String option : strings) {
            sb.append(" ").append(option);
        }
        sb = sb.length() == 0 ? sb.append("No Options Specified") : sb;
        logger.info(String.format("WAR launched with following options: %s", sb.toString()));
    }
}
```

Starters
spring-boot-starter-parent

- We can inherit from spring-boot-starter-parent project ***to obtain sensible defaults***.
- The parent project provides the following features:
  - Java 1.8 as the default compiler level.
  - UTF-8 source encoding.
  - A Dependency Management section, inherited from the spring-boot-dependencies pom, that manages the versions of common dependencies. This dependency management lets ***you omit <version> tags*** for those dependencies when used in your own pom.

POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
		<modelVersion>4.0.0</modelVersion>

		<groupId>com.example</groupId>
		<artifactId>myproject</artifactId>
		<version>0.0.1-SNAPSHOT</version>
		<!-- Inherit defaults from Spring Boot -->
		<parent>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-starter-parent</artifactId>
				<version>2.0.1.BUILD-SNAPSHOT</version>
		</parent>
```

POM

```
<!-- Add typical dependencies for a web application -->

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


            <!-- Package as an executable jar -->
            <build>
                        <plugins>
                                    <plugin>
                                                <groupId>org.springframework.boot</groupId>
                                                <artifactId>spring-boot-maven-plugin</artifactId>
                                    </plugin>
                        </plugins>
            </build>
```

Using Spring Boot without Parent POM

- spring-boot-starter-parent is a great way to use Spring Boot, but it might not be suitable all time.
- If we have own corporate standard parent that to use or we prefer to explicitly declare all Maven configuration.
- Sometimes we may need to inherit from a different parent POM

Using Spring Boot without Parent POM

- In those cases Use Spring Boot without the Parent POM” for an alternative solution that uses an import scope.
- We can still keep benefit of dependency management (***but not the plugin management***) by using a scope=import dependency

Using Spring Boot without Parent POM

```
•   <dependencyManagement>
•               <dependencies>
•               <dependency>
•               <!-- Import dependency management from Spring Boot -->
•                       <groupId>org.springframework.boot</groupId>
•                       <artifactId>spring-boot-dependencies</artifactId>
•                       <version>2.0.1.BUILD-SNAPSHOT</version>
•                       <type>pom</type>
•                       <scope>import</scope>
•               </dependency>
•       </dependencies>
•   </dependencyManagement>
```

Best Practices

Structuring your code

- *Avoid using the "default" package*

The use of the "default package" is generally discouraged, and should be avoided.

It can cause problems for Spring Boot applications that use :

@ComponentScan @EntityScan or @SpringBootApplication annotations

(since every class from every jar, will be read.)

Locating the main application class

- *Keep your main application class in a root package above other classes.*
- *The @EnableAutoConfiguration annotation is often placed on your main class, and it implicitly defines a base "search package" for certain items.*
- *For example, if you are writing a JPA application, the package of the @EnableAutoConfiguration annotated class will be used to search for @Entity items.*
- *Using a root package also allows the @ComponentScan annotation to be used without needing to specify a basePackage attribute.*
- *You can also use the @SpringBootApplication annotation if your main class is in the root package.*

Typical Layout

```
com
 +- example
     +- myproject
         +- Application.java
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- web
             +- CustomerController.java
```

Typical Layout

The Application.java file would declare the main method, along with the basic @Configuration.

```
package com.example.myproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {

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

}
```