Spring Boot

Created by :
Sangeeta Joshi

Agenda

- What is Spring Boot
- Why Spring Boot
- How Spring Boot

What is Spring Boot

- Spring Boot is a project created by Spring Team to build production ready spring applications.
- Spring Boot favours convention over configuration
- It is designed to get you up and running as quickly as possible.

Why Spring Boot

Spring :

- a very popular framework for building Java web and enterprise applications.
- It provides a wide verity of features addressing the modern business needs(via its portfolio projects).
- Unlike many other frameworks which focus on only one area

Why Spring Boot

Spring :

- provides flexibility to configure beans in multiple ways such as: **XML**, **Annotations,** and **JavaConfig**.
- With number of features increased complexity also gets increased
- configuring Spring applications becomes tedious and error-prone.

Spring Boot :

Spring Boot is created to address complexity of configuration.

Why Spring Boot

Usecase :

We want to build a Web Application with:

Spring MVC , JPA(Hibernate) and MySql DB

Various configurations-steps needed:

- Maven Dependencies
- Service/DAO layer dependencies
- Web Layer MVC dependencies
- Log4j

Why Spring Boot

Problems while doing all those configurations:

- So many configurations so can not get up and run quickly
- If we want to develop another spring web app with similar technology stack ? (copy and tweak?)
- hunt for all the **compatible libraries** for the specific Spring version and configure
- 95% of the times we configure **DataSource, EntitymanagerFactory**,**TransactionManager** etc beans in the same way
- Also configure SpringMVC beans like **ViewResolver**, **MessageSource** etc in the same way most of the times.

Solution : ***an automated way to do it ALL***

Why Spring Boot

Solution : ***an automated way to do it ALL***

(If Spring can automatically do it for me? : that would be awesome!!!.)

- what if Spring is capable of configuring beans automatically?
- What if we can customize automatic configuration using simple customizable properties?

So basically we want Spring to do things automatically but provide flexibility to override default configuration in a simpler way? So that is :

SPRING BOOT

Structuring your code

Using the "default" package

When a class doesn't include a package declaration it is considered to be in the "default package".

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

It can cause particular 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

We generally recommend that you locate 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);
    }

}
```

Configuration classes

Spring Boot favors Java-based configuration. Although it is possible to call SpringApplication.run() with an XML source, we generally recommend that your primary source is a @Configuration class. Usually the class that defines the main method is also a good candidate as the primary @Configuration.

Configuration classes

Importing additional configuration classes

You don't need to put all your @Configuration into a single class. The @Import annotation can be used to import additional configuration classes. Alternatively, you can use @ComponentScan to automatically pick up all Spring components, including @Configuration classes.

Importing XML configuration

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

Annotations

Spring Beans and dependency injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. For simplicity, we often find that using @ComponentScan to find your beans, in combination with @Autowired constructor injection works well.

If you structure your code as suggested above (locating your application class in a root package), you can add @ComponentScan without any arguments. All of your application components (@Component, @Service, @Repository, @Controller etc.) will be automatically registered as Spring Beans.

Here is an example @Service Bean that uses constructor injection to obtain a required RiskAssessor bean.

Annotations

Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, If HSQLDB is on your classpath, and you have not manually configured any database connection beans, then we will auto-configure an in-memory database.

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

[Tip]

You should only ever add one @EnableAutoConfiguration annotation. We generally recommend that you add it to your primary @Configuration class.

Annotations

Gradually replacing auto-configuration

Auto-configuration is noninvasive, at any point you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own DataSource bean, the default embedded database support will back away.

If you need to find out what auto-configuration is currently being applied, and why, start your 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 the @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 (especially if you follow the best practices above), Spring Boot provides a convenient @SpringBootApplication alternative.

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes:

```
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

Published by Dave Burke on September 2, 2015

A quick post on a cool Spring Boot Interface called CommandLineRunner. With CommandLineRunner you 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()));
    }
}
```