Spring boot

Ref: <https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#using-boot-structuring-your-code>

Spring started many projects. This is the location of the projects references <https://spring.io/guides> . Each project gave us the wireframe of the solution. We can have it as the start point.

Why we adopt springboot?

SpringBoot features:

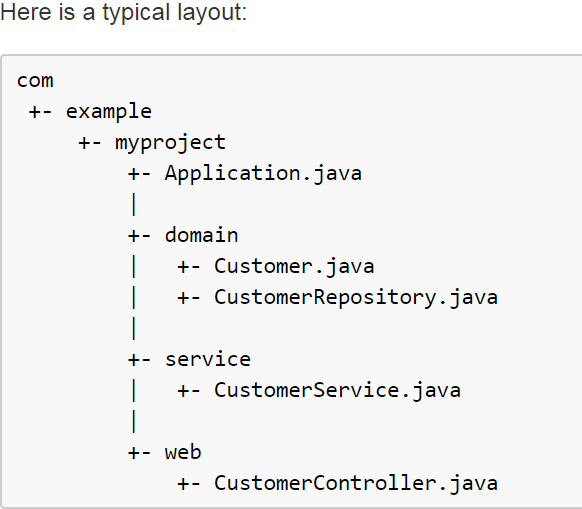
Our primary goals are:

* Provide a radically faster and widely accessible getting started experience for all Spring development.
* Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
* Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
* Absolutely no code generation and no requirement for XML configuration.

## Best practice:

. Structure code

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.



. one of the most important reasons of using spring boot is that spring boot allows you to externalize your configuration so you can work with the same application code in different env.

. another very cool function of spring boot is Hot swapping. JVM hot swapping is somewhat limited with the bytecode that is can replace, for a more solution JRebel and Spring loaded project can be used. in spring boot, we can include spring-boot-devtools module to make quick application restart. Just includes the dependency below:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

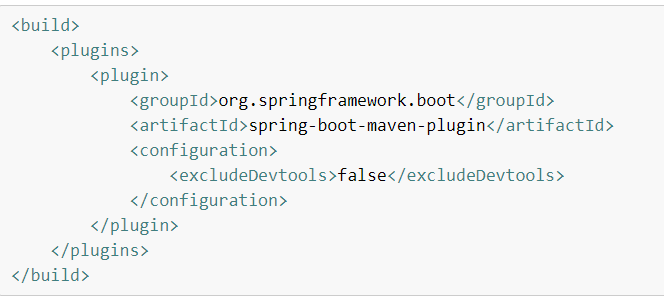
</dependency>

</dependencies>

Developer tools are automatically disabled when running a fully packaged application. If your application is launched using java –jar or if it’s started using a special classloader, then it is considered a ‘production application’. Flagging the dependency as optional is a best practice that prevents devtools from being transitively applied to other modules using your project. there are many special setting to customize the usage.

. spring-boot-devtools can be also used for remote debug.

1. Change pom.xml file

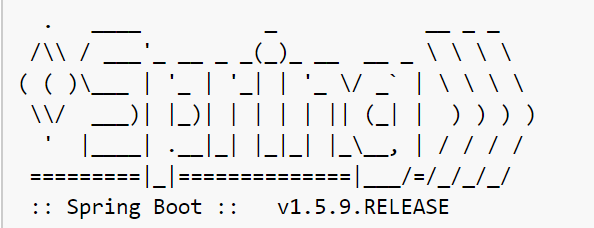


1. Set spring.devtools.remote.secret=mysecret property (after set and maven rebuilt, remotely server is functional)
2. Start client side by configuring remote server url

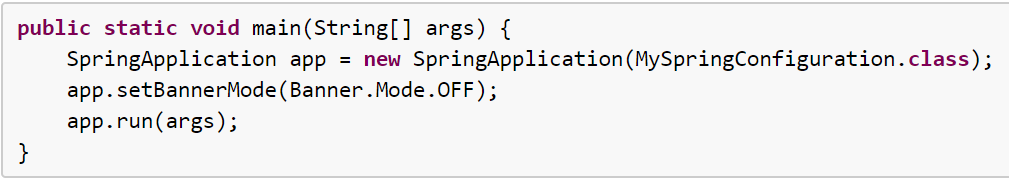
## Spring boot details:

. startup springboot app as debug mode – java -jar myproject.jar --debug

. customizing the banner (banner is sth like below when app started)



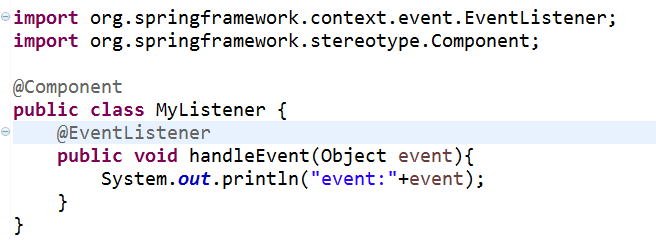
Turn off banner



. Spring Event (ref: <http://www.baeldung.com/spring-events> )

Spring event-based communication model is analogue to message queue solution. It is to loosely coupled communication.

Spring exposed some applicationContext level of event. We can register our own listeners in the events. (I think this is normally used in integration)



Read arguments from spring boot code

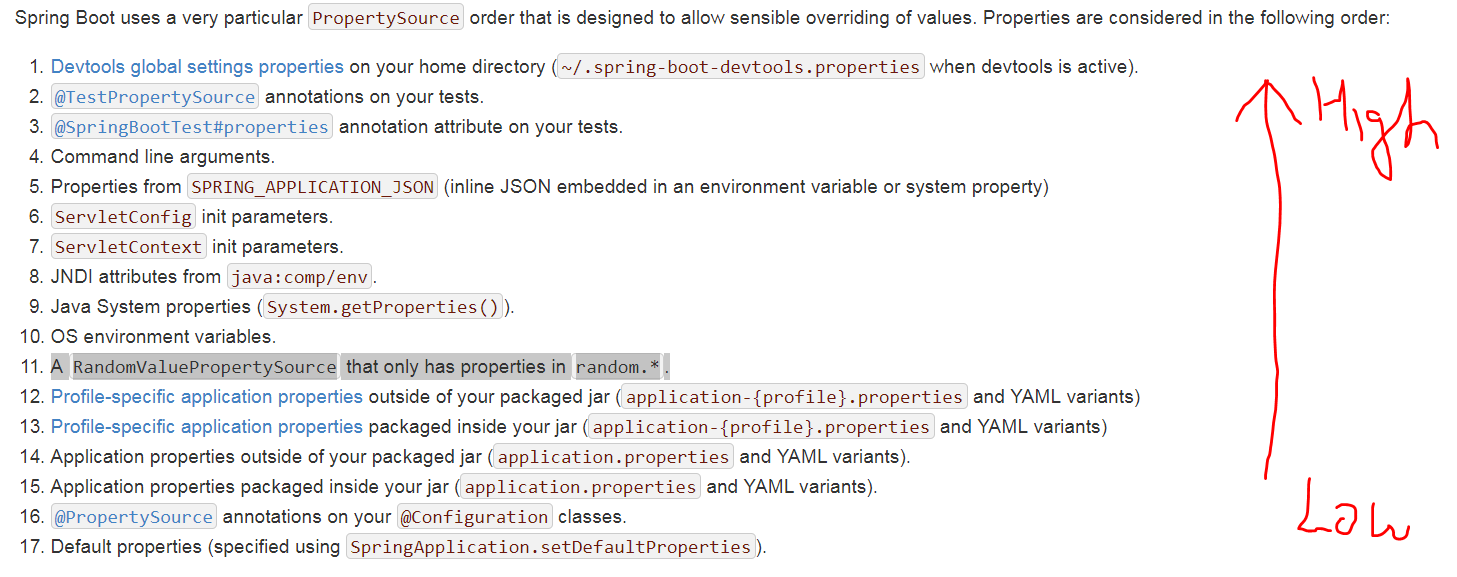




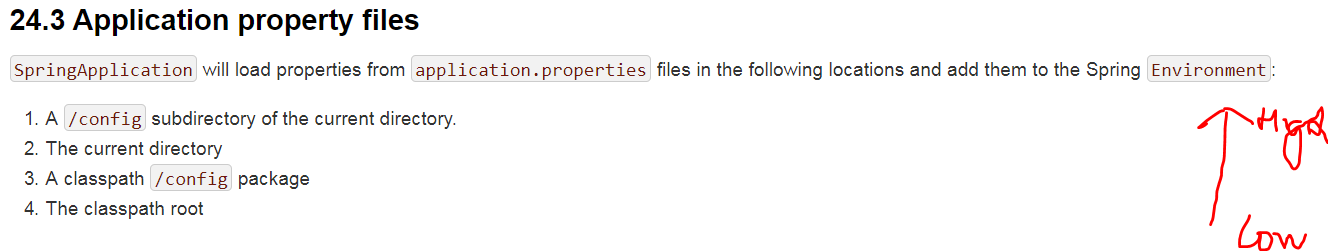
## . Externalized configuration

Ref: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config>

Spring boot allows us to externalize our configuration so that we can work with the same app code in different environments. We can use properties files, YAML files, environment variables and command line arguments to externalize configuration. Property values can be injected directly into our beans using the @Value annotation.



The precedence of loading application.properties file



In addition to using the default application.properties file name and location, we can define our own filename and location by defining spring property spring.config.location(where configfile is) and spring.config.name(what is the customized config filename), but it is not very suggested because it makes ourselves hard to maintain. If really like to do, we can refer to the document.

We can also define multiple application-{profile}.properties for one application. Therefore, it is able to work on different env. The profile is defined by spring.profiles.active property. (details see [ref](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config) in 24.4)

=======Jan 31 2018

->Before SpringBoot context initialization completion (<https://dzone.com/articles/spring-boot-applicationrunner-and-commandlinerunne> )

Spring Boot provides two interfaces, CommandLineRunner and ApplicationRunner, to run specific pieces of code when an application is fully started. These interfaces get called just before run()once SpringApplication completes.

CommandLineRunner

This interface provides access to application arguments as string array. Let's see the example code for more clarity.

@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));

}

}

ApplicationRunner

ApplicationRunner wraps the raw application arguments and exposes the ApplicationArguments interface, which has many convenient 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(). Let's see an example:

@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());

}

}

Sometimes, we’d like to execute some piece of code exactly before the application startup completes. In one of our projects, we used these to source data from other microservices via service discovery, which was registered in Consul.

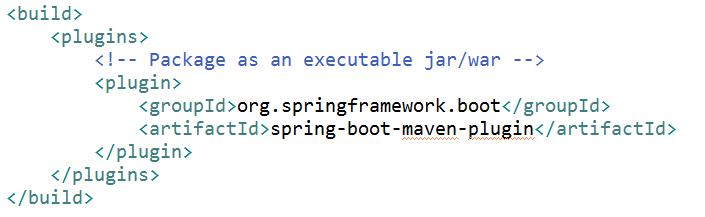
## Ordering

You can register as many application/command line runners as you want. You just need to register them as Beans in the application context. Then, Spring will automatically pick them up. You can order them as well either by extending interface org.springframework.core.Ordered or via the @Order annotation.

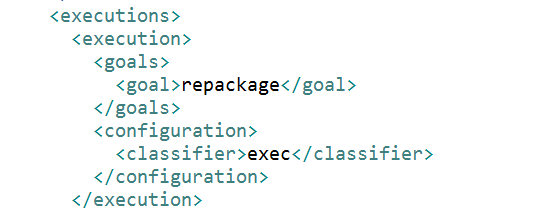
This is all about the application/command line runners. You can also see org.springframework.boot.autoconfigure.batch.JobLauncherCommandLineRunner in **spring-batch**, which implements CommandLineRunner to register and start batch jobs at application startup. I hope you find this informative and helpful. You can grab the full example code on [GitHub](https://github.com/gauravrmazra/gauravbytes/tree/master/spring-boot-hook-runner).

## Spring boot maven plugin

Spring boot can automatically generate the executable jar or war file for us once spring-boot-maven-plugin is included in pom.xml file.



Latterly, we have the requirement to share the spring boot project as a common lib for other projects. The executable artifact cannot be directly used. The configuration of pom should be sth like below.



This configuration generates 2 artifacts. One is executable. The other one is the pure shareable library.