Spring Breakout Session
Spring Boot

by Vlad Costel Ungureanu
for FII UAIC, Learn Stuff

Spring Breakout Session

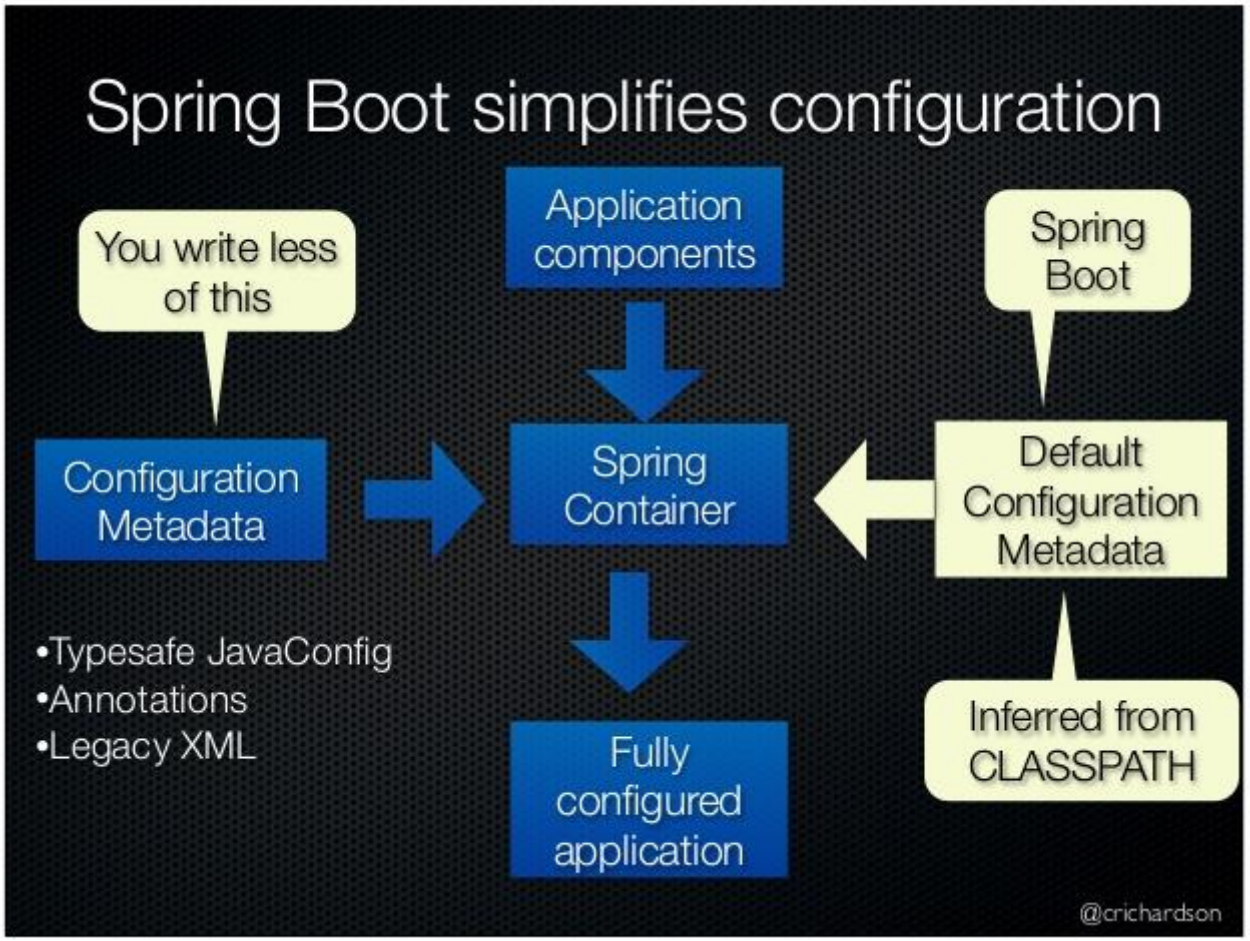
Spring Boot

- Features
- Requirements
- Spring Boot Modules
- Getting started
  - Starter POMs
  - Custom starter POMs
  - Main application class
  - Configurations
- Embedded Servers
- Spring MVC
- Logging
- Database
- Spring Security
- Error Handling
- Profiling

Spring Boot - Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Requirements



Getting Started

Project initializer: http://start.spring.io/ - bare bones Spring Boot project with Gradle/Maven support

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.5.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Spring Boot Modules

- **Cli** - Spring Boot software to run and test Spring Boot applications from command prompt;
- **Starters** - a set of convenient dependency descriptors that you can include in your application - spring-boot-starter-* ;
- **Autoconfigure** - attempts to automatically configure your Spring application based on the jar dependencies that you have added;
- **Actuator** - enable production-ready features to a Spring Boot application – without having to actually implement these things yourself;
- **Tools** - configuration and tools like Maven, Gradle;

Starter POMs

spring-boot-starter	The core Spring Boot starter, including auto-configuration support, logging and YAML.
spring-boot-starter-integration	Support for common spring-integration modules.
spring-boot-starter-security	Support for spring-security.
spring-boot-starter-test	Support for common test dependencies, including JUnit, Hamcrest and Mockito along with the spring-testmodule.
spring-boot-starter-web	Support for full-stack web development, including Tomcat and spring-webmvc.

Custom Starter POM

A full Spring Boot starter for a library may contain the following components:

- The **autoconfigure** module that contains everything that is necessary to get started with the library. It may also contain configuration keys definition (@ConfigurationProperties) and any callback interface that can be used to further customize how the components are initialized.

- The **starter** module that provides a dependency to the autoconfigure module as well as the library and any additional dependencies that are typically useful. In a nutshell, adding the starter should be enough to start using that library

Third party starters should not start with spring-boot as it is reserved for official Spring Boot artifacts. A third-party starter for endava will be typically named endava-spring-boot-starter.

Main Application Class

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

Main Application Class

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

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

Auto-Configuration

- Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, when we include spring-boot-starter-thymeleaf, Spring Boot has the opinion that we'll probably want embedded Tomcat, Jackson JSON support, JSR 303 validation, and Spring Web MVC. So, it adds them as required dependencies.

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

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

Externalized Configuration

- Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables and command-line arguments to externalize configuration.

- Properties are considered in the the following order:

  - Command line arguments.
  - JNDI attributes from java:comp/env.
  - Java System properties (System.getProperties()).
  - OS environment variables.
  - A RandomValuePropertySource that only has properties in random.*.
  - Profile-specific application properties outside of your packaged jar (application-{profile}.properties and YAML variants)
  - Profile-specific application properties packaged inside your jar (application-{profile}.properties and YAML variants)
  - Application properties outside of your packaged jar (application.properties and YAML variants).
  - Application properties packaged inside your jar (application.properties and YAML variants).
  - @PropertySource annotations on your @Configuration classes.
  - Default properties (specified using SpringApplication.setDefaultProperties).

Embedded Servers

- Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers.
- The Spring Boot starters generally uses **Tomcat as the default embedded server**. If that needs to be changed – you can exclude the Tomcat dependency and include Jetty or Undertow instead:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Spring Services

Spring Boot provides auto-configuration for Spring MVC that works well with most applications;
JSON REST service - @RestController (requires Jackson2 is on the classpath);
XML REST service

Jackson XML :

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Woodstox:

```
<dependency>
        <groupId>org.codehaus.woodstox</groupId>
        <artifactId>woodstox-core-asl</artifactId>
</dependency>
```

JAXB by default if Jacskon XML is missing;

LEARN stuff IO

Logging

- **Logging**
  - uses Common Logging, but leaves the underlying log implementation open
  - default configurations are provided for Java Util Logging, Log4J, Log4J2 and Logback;
  - The logger levels can be set in the Spring Environment (application.properties) using 'logging.level.*=LEVEL' where 'LEVEL' is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF.
    - logging.level.org.springframework.web=DEBUG

      logging.level.org.hibernate=ERROR
- **Templating (for MVC)**
  - auto-configuration support for the following templating engines: FreeMarker, Groovy, Thymeleaf, Velocity, Mustache;
  - due to some several known limitations, JSPs should be avoided if possible;
  - templates will be picked automatically from src/main/resources/templates

Database

- SQL databases
  - autoconfigure embedded H2, HSQL, and Derby databases;

```
<dependency>
        <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
        <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
</dependency>
```

- NoSQl databases
  - Additional projects for connecting to MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire,Couchbase and Cassandra databases.

Database configuration

- To provide configurations for your application's datasource it is enough to add the following lines to your application.properties

```
#postgres configuration details
spring.datasource.url=jdbc:postgresql://localhost:5432/db_name
spring.datasource.username=db_username
spring.datasource.password=db_password
spring.datasource.driver-class-name=org.postgresql.Driver
```

Spring Security

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

The basic features you get out of the box in a web application are:

- An AuthenticationManager bean with in-memory store and a single user
- Ignored (unsecure) paths for common static resource locations (/css/**, /js/**, /images/** and **/favicon.ico).
- HTTP Basic security for all other endpoints.
- Security events published to Spring's ApplicationEventPublisher (successful and unsuccessful authentication and access denied).
- Common low-level features (HSTS, XSS, CSRF, caching) provided by Spring Security are on by default.

Spring Security

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Autowired
   private UserDetailsService userDetailsService;

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests()
            .formLogin().loginPage("/login")
            .failureUrl("/login?error").usernameParameter("email")
            .permitAll().and().logout()
            .logoutUrl("/logout") .logoutSuccessUrl("/")
            .permitAll();
   }
   @Override
   public void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(userDetailsService)
            .passwordEncoder(new BCryptPasswordEncoder());
   }
}
```

Error handling

- /error mapping by default that handles all errors in a sensible way, and it is registered as a 'global' error page in the servlet container
- replacing this default mapping can be done :
  - implementing ErrorController and register a bean definition of that type;
  - adding a bean of type ErrorAttributes to use the existing mechanism but replace the contents;
  - using regular Spring MVC features like @ExceptionHandler methods and @ControllerAdvice;
- If you want more specific error pages for some conditions, the embedded servlet containers support a uniform Java DSL for customizing the error handling

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer(){
   return new MyCustomizer();
}
private static class MyCustomizer implements EmbeddedServletContainerCustomizer {
   @Override
   public void customize(ConfigurableEmbeddedServletContainer container) {
      container.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
   }
}
```

Profiling

You can provide profile-specific configurations using

configuration classes

```
@Configuration//@Component
@Profile("production")
public class ProductionConfiguration {
  // ...
}
```

specific application.properties files using the following naming convention: application-{profile}.properties

Assignments

- ✓Remove the configuration from the previous application and implement the configuration using Spring Boot

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –