EInnoVator™
Software Engineering School

Spring Boot

Jorge Simão, Ph.D.

Content

Spring Boot Overview

Spring Boot is a *Rapid Application Development* (RAD) library-set for Spring enabled Java applications. Boot promotes a *configuration by convention* approach, by taking an "opinionated view" on how Spring projects should be setup and enabling suitable configuration defaults for most Spring modules and imported third-party dependencies. Boot differs notoriously from other RAD approaches, by not using source-code generation and not requiring any specific tooling/IDE support. Boot makes production-graded applications easy to get started, while allowing easy overriding of default configuration options.

Dependency Management

Spring Boot is best used in combination with project build systems – specially those that support dependency management. For Maven, the parent POM **spring-boot-starter-parent** can be used to automatically inherit version information for common third-party libraries (i.e. no need to specify the **<version>** element in dependencies).

» Example: Maven .pom for a Boot Project

```
<?xml version="1.0" encoding="UTF-8"?>
<project ... >
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.myorg</groupId>
  <artifactId>myapp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.3.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!--  other spring-boot-starter-* -->
    <!--  other dependencies -->
  </dependencies>
</project>
```

Spring Boot provides a wide range of *starter* dependency descriptors that import sets of related dependencies and allow to get started building specific kinds of applications easily (i.e. without having to enumerate all the imported dependencies). These dependency descriptors follow the naming convention **spring-boot-starter-***, where the wildcard * stand for the type of project. Starters also import **spring-boot-autoconfigure** dependency that supports auto-configuration features. Table below summaries some of the starters and the imported dependencies.

Starter	Description & Dependencies
spring-boot-starter	Boot Core, Auto-Configuration, Logging, YAML
spring-boot-starter-web	spring-web,spring-webmvc, Validation, Jackson, Tomcat
spring-boot-starter-test	spring-test, JUnit, Hamcrest, Mockito
spring-boot-starter-aop	spring-aop, aspectjrt, aspectjweaver
spring-boot-starter-thymeleaf	Thymeleaf VDL w/ SpringMVC integration
spring-boot-starter-jdbc	spring-jdbc, tomcat-jdbc
spring-boot-starter-data-jpa	spring-data-jpa, spring-orm, JPA API, Hibernate
spring-boot-starter-data-mongodb	spring-data-mongodb , MongoDB Java Driver
spring-boot-starter-redis	spring-data-redis, Redis Java driver (jedis)
spring-boot-starter-amqp	spring-rabbit, RabbitMQ Java driver
spring-boot-starter-security	spring-security-* (config, web)
spring-boot-starter-integration	spring-integration-* (core, file, http, ip, stream)
spring-boot-starter-batch	spring-batch-core, spring-jdbc, HSQLDB Java Driver
spring-boot-starter-cloud-connectors	spring-cloud-*-connector (local-config, cloudfoundy, heroku, spring-service)

Spring Boot Applications

A Spring Boot application can be written as a Java standalone app – with a **main()** method – and bootstrapped by calling method **SpringApplication.run()**, which lunches an embedded web container and automatically creates a Spring **ApplicationContext**. Alternatively, a Boot enabled web application can be deployed to a Servlet container by implemented Java **WebInitializer** that extends **SpringBootServletInitializer**.

» Example: Boot Standalone App

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

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v1.3.0.BUILD-SNAPSHOT

2015-05-15 12:00:30.000  INFO 12345 --- [main] myorg.MyAapp: Starting
MyApplication v0.1.0 on myhost with PID 1235 (/apps/myapp.jar …)
```

» Example: Boot Web Initializer

```
public class ServletInitializer extends
SpringBootServletInitializer {
   @Override
   protected SpringApplicationBuilder
configure(SpringApplicationBuilder app) {
      return app.sources(MyApp.class);
   }
}
```

Annotation **@EnableAutoConfiguration** is used to enable auto-configuration, or alternatively **@SpringBootApplication** which combines auto-configuration with annotations **@Configuration** and **@ComponentScan**. By placing this annotation in a class locate in the *root package* of the application – often the the main class, all classes annotated with Spring stereotype annotations will be picked-up during component scan (e.g. **@Component**, **@Repository**, **@Service**, **@Configuration**). For programmatic customization, an instance of **SpringApplication** can be created explicitly and initialized using property setters, before calling **run()**. Class **SpringApplicationBuilder** also supports the *builder design-pattern* with a fluent API.

» Example: Boot App w/ Programmatic Customization

```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        new SpringApplicationBuilder()
              .showBanner(false)
              .sources(MyApp.class)
              .run(args);
    }
}
```

» Example: Importing XML Configuration File

```
@SpringBootApplication
@ImportResource("classpath:infrastructure-config.xml")
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

Auto-Configuration

Spring Boot uses the mechanism of *Condition Beans*, introduced in **Core Spring Framework 4.1**, to have required configuration components automatically defined – if not explicitly defined by the application. Presence of a dependency in the *classpath* is enough to trigger auto-configuration (e.g. the presence of a DB driver in the *classpath*, such as HSQLDB, MySql, or MariaDB, will make a **java.sql.DataSource** bean to be automatically defined if none is explicitly defined in the **ApplicationContext**.) Conversely, Boot modules refrain from defining a component if they are defined by the application (e.g. in a Java-Spring Config class) . Additionally, the attribute **exclude()** in annotations **@EnableAutoConfiguration and @SpringBootApplication**, can be used to disable a **AutoConfiguration** class.

» Example: Disabling Specific AutoConfiguration Classes

```
import org.springframework.boot.autoconfigure.*;
import org.springframework.boot.autoconfigure.jdbc.*;

@SpringBootApplication(exclude={DataSourceAutoConfigura
tion.class})
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

Library developers can also provide support for auto-configuration of custom bean types using the mechanisms of conditional beans of Spring 4.1 – **@Conditional** annotation and **Condition** interface, including the many annotations Boot provides for specific conditions – such as **@ConditionalOnClass**, **@ConditionalOnMissingBean**, and **@ConditionalOnProperty**.

» Example: Custom Auto-Configuration

```
@Configuration
@ConditionalOnClass({ MyBean.class })
```


```
@EnableConfigurationProperties(MyProperties.class)
public class MyAutoConfiguration {
  @Bean
  @ConditionalOnProperty(prefix="myorg.mylib", name =
"enabled", havingValue = "true", matchIfMissing = true)
  @ConditionalOnMissingBean(MyBean.class)
  public MyBean mybean(MyProperties props) {
    return new MyBean(props);
  }
}
```

Externalized Configuration

Spring Boot allows the externalization of application configuration using conventionally-named property files and YAML files, command-line arguments, in addition to the usual OS environment variables, or JVM properties. Specifically, Spring Boot installs a **PropertySource** that looks up for files named **application.properties** and **application.yml**, in several locations including project root, *classpath* directories, and inside directory **/config** on the root or a *classpath* directory. Spring *profile* specific files are also located and loaded with names **application-*{profile}*.properties** and **application-*{profile}*.yml.** Command line argument with syntax **--propertyName=value** are also accepted. These configuration files can be located inside or outside the JAR where and application is bundled. Precedence is as follows:

- Command line arguments: **--propertyName=value**
- JNDI attributes from: **java:comp/env**
- JVM properties: **–DpropertyName=value**
- OS environment variables
- A **RandomValuePropertySource** for: **random.*** settings
- Profile-specific **application-{profile}.properties** and **application-{profile}.yaml** (outside JAR, followed by inside JAR);
- **application.properties** and **application.yml** (ouside JAR, followed by inside JAR)
- Additional **@PropertySource** in **@Configuration** classes
- Defaults set w/ **SpringApplication.setDefaultProperties()**

» Example: Configuring Boot and App Properties

```
spring.main.show_banner=false
db.url=jdbc:mysql:server-a.myorg/mydb
db.slaves[0]=jdbc:mysql:server-b.myorg/mydb
db.slaves[1]=jdbc:mysql:server-c.myorg/mydb
```

» Example: Configuring App with YAML

```
db:
  url: jdbc:mysql:server-a.myorg/mydb
  username: dba
  password: pa$$
  slaves:
    - dbc:mysql:server-a.myorg/mydb
    - jdbc:mysql:server-c.myorg/mydb
```

All settings are registered in the **Environment** of the Spring **ApplicationContext**. Relaxed naming of properties is supported: case-insensitive, *camelCase* to *XML-notation* equivalence, and word-separation with '.' or '_'. Settings can by used to configure components of Spring Boot modules, to resolve property placeholder in **@Value** annotations, and to perform injection of properties in structured objects/beans annotated with **@ConfigurationProperties**. Annotation **@EnableConfigurationProperties** should be used in a **@Configuration** class to enable **@ConfigurationProperties** injection. JSR-303 validation annotations can also be used to validate property values. (Because Spring Boot uses a **DataBinder** to perform injection, property getters&setter methods should be defined.)

» Example: Injecting Individual Properties

```
@Configuration
public class DBConfig {
    @Value("${db.url}") private String url;
    @Value("${db.username}") private String username;
    @Value("${db.password}") private String password;
}
```

» Example: Property Injection in Structured Objects

```
@Configuration
@ConfigurationProperties(prefix="db")
public class DBConfig {
    @NotNull
    private String url;
    private String username;
    private String password;
    private List<String> slaves;
    //getters and setters...
}
```

» Example: Enabling Configuration Properties Injection

```
@Configuration
@EnableConfigurationProperties(DBConfig.class)
public class MyConfiguration {
}
```

Table below summarizes some of the basic properties that can be set to configure Spring Boot.

Property	Description (*Default*)
spring.config.location	Directory with properties/yaml files (*classpath:,classpath:/config,file:,file:config/*)
spring.config.name	Filename for properties files (*application*)
banner.location	Banner file location (*classpath:banner.txt*)
banner.encoding	Banner file encoding (*UTF-8*)

logging.file	Logging file location (Log to console only, by default)
logging.path	Directory for log file (default filename is *spring.log*)
logging.level.*	Logging level (*INFO*)
logging.config	Log configuration file (*logback.{xml,groovy}* | *log4j.{properties,xml}* | *log4j2.xml* | *logging.properties*)
spring.main.web_environment	Enable/Disable WebApplicationContext (*true*)
spring.main.show_banner	Show/Hide start banner (*true*)
spring.application.name	Application name (*the class name*)

Web Applications and REST-WS

Spring Boot simplifies the development of web applications by configuring Spring MVC with sensible defaults when the starter dependency **spring-boot-starter-web** is imported. Auto-configured features include:

- Serving of static resources (e.g img, js, css) located in pre-defined directories: **index.html**, **/static**, **/public**, **/resources**, **/META-INF/resources**, and **/webjars/**** (for JAR packaging)
- **MessageConverter**s for REST-WS (XML, JSON, ...)
- View Resolvers: **ContentNegotiatingViewResolver** and **BeanNameViewResolver**

Auto-configuration of several VDL is also supported by importing appropriate starters, including: *Velocity*, *FreeMarker*, *Groovy*, and *Thymeleaf*. View files should be located under **src/main/resources/templates**.

Auto-Configuration of Spring MVC can be suppressed altogether using annotation **@EnableWebMvc** in a **@Configuration** class. To refine the Spring Boot base auto-configuration class **WebMvcConfigurerAdapter** can be extended and desired methods override.

» Example: Refining MVC Auto-Configuration

```
@EnableWebMvc
@Configuration
public class MvcConfig extends WebMvcAutoConfigurationAdapter {
  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addRequestInterceptor(new MyInterceptor());
  }
}
```

Spring Boot applications run with an embedded *Servlet container* (Java web server), when deployed as a **main()** (standalone) applications. By default, *Tomcat* is used as container, but *Jetty* and *Undertow* are also supported out-of-the-box by importing appropriate starters. Table below summarizes Boot properties specific to web applications.

Property	Description (*Default*)
server.port	HTTP server port (*8080*)
server.address	IP interface address (*all*)
server.sessionTimeout	HTTP Session timeout (in ms)
server.ssl.*	SSL setting (key/cert files&pass)
server.tomcat.*	Tomcat specific settings
spring.jackson.*	Jackson mapping (JSON) settings
spring.thymeleaf.*	Thymeleaf specific settings

Security

Spring Boot performs auto-configuration of Spring Security when starter dependency **spring-boot-starter-security** is imported. All HTTP endpoints are protected to require BASIC authentication credentials, and an authentication provider is configured with a simple in-memory database with a single user named *user* and a random password (logged with level INFO).

Auto-Configuration of Spring Security can be suppressed using annotation **@EnableWebSecurity** in a custom **@Configuration** class, or refined by extending class **WebSecurityConfigurerAdapter**.

» Example: Refining Security Auto-Configuration

```
@EnableWebSecurity
@Configuration
public class SecurityConfiguration extends
WebSecurityConfigurerAdapter {
  @Override
  public void configure(WebSecurity web) throws Exception { … }
  …
}
```

SQL/Relational Data-Access

Spring Boot auto-configures a **javax.sql.DataSource**, if not defined explicitly as an application bean, and starters **spring-boot-starter-jdbc** or **spring-boot-starter-data-jpa** is imported plus a suitable driver. Driver H2, HSQL, Derby create an embedded data-base, while drivers MySql or MariaDB can be used for production. Settings **spring.datasource.*** can be used to configure the connection details. Connection pooling is automatically enabled (using Tomcat connection pooler, or other available in the classpath). An instance of **JdbcTemplate**

and **NamedParameterJdbcTemplate** are also created and configured automatically.

» Example: Imports for Data-Source Auto-Configuration

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

» Example: Setting DataSource Connection Details

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Importing starter **spring-boot-starter-data-jpa** auto-configures a JPA **EntityManagerFactory**, and a **JpaTransactionManager**. Scanning of **@Entity**, **@MappedSuperclass**, and **@Embedded** classes is done automatically below the package of class annotated with **@EnableAutoConfiguration** or **@SpringBootApplication**. Spring Data JPA repository classes are also automatically created if component-scanning finds interfaces extending **Repository** (or **CrudRepository**). Table below summarizes Boot properties for **DataSource** and JPA auto-configuration:

Property	Description (*Default*)
spring.datasource.url	DataSource URL
spring.datasource.username	DataSource username
spring.datasource.passoword	password
spring.datasource.diver-class-name	driver (for individual connections)
spring.datasource.jndi-name	JNDI name for DataSource
spring.jpa.hibernate.ddl-auto	Hibernate auto-schema creation (DDL execution)
spring.jpa.properties.*	Any JPA property
spring.datasource.name	Name for embedded DataSource (*testdb*)
spring.datasource.data	SQL initialization script
spring.datasource.initialize	Initialize with *{data}.sql* (*true*)

NoSql Data-Access

Spring Boot has auto-configuration starters for several NoSql DBs supported by Spring Data, including: MondoDB, Redis, Elasticsearch, and Solr. For MondoDB, a beans of type **Mongo**, **MongoDbFactory**, and **MongoTemplate,** are automatically created (connecting by default to **mongodb://localhost/test**).

For Redis, beans of type **RedisConnectionFactory**, **StringRedisTemplate** or **RedisTemplate** are automatically created (connecting by default to **localhost:6379**).

For Solr, an instance of **SolrServer** is auto-created (connected to *localhost:8983/solr*). For Elasticsearch, and instance of **ElasticsearchTemplate** and Elasticsearch **Client** is auto-created (connected to an in-memory server). Spring Data JPA repository classes are automatically created for all of these NoSQL DBs when the starter dependency is imported. Table below summarizes Boot properties for MongoDB and Redis auto-configuration.

Property	Description (*Default*)
spring.data.mongodb.uri	MongoDB server URI
spring.data.mongodb.host	MongoDB server (*localhost*)
spring.data.mongodb.port	MongoDB server port (*27017*)
spring.data.mongodb.database	MongoDB database name (*test*)
spring.redis.host	Redis server host (*localhost*)
spring.redis.port	Redis server port (*6379*)
spring.redis.password	Password for Redis server
spring.redis.pool.*	Jedis connection pool settings
spring.redis.sentinel.*	Master&slaves addresses

Messaging

Spring Boot has auto-configuration starters for messaging-based communication based on JMS and AMQP. The core Boot starter has built-in support for ActiveMQ broker – importing ActiveMQ driver&server to the *classpath*, makes an **ActiveMQConnectionFactory** to be automatically configured. An embedded server is started, unless property **spring.activemq.broker-url** is set or if running inside an application server. For HornetQ broker the starter **spring-boot-starter-hornetq** should be imported. In both cases, a JMS **ConnectionFactory** and a **JmsTemplate** is auto-configure. If annotation **@JmsListener** is used for asynchronous reception, a **JmsListenerContainerFactory** is also auto-created (without need to explicitly use **@EnableJms**). Table below summarizes Boot properties for JMS based messaging.

Property	Description (*Default*)
spring.activemq.broker-url	URL to ActiveMQ broker
spring.activemq.user	Username in ActiveMQ
spring.activemq.password	Password in ActiveMQ
spring.jms.jndi-name	ConnectionFactory JDNI name (*java:/JmsXA*, *java:/XAConnectionFactory*)
spring.hornetq.mode	URL to HornetQ broker (*native*)
spring.hornetq.host	Host of HornetQ broker
spring.hornetq.port	Port of HornetQ broker (*9876*)

RabbitMQ/AMQP auto-configuration is enabled by importing **spring-boot-starter-amqp**. This will automatically creates beans of type Spring AMQP **ConnectionFactory**, a **RabbitTemplate**, and **RabbitAdmin**. Table below summarizes Boot properties for RabbitMQ/AMQP based messaging:

Property	Description (*Default*)
spring.rabbitmq.host	Host for RabbitMQ broker
spring.rabbitmq.port	Port for RabbitMQ broker (*5672*)
spring.rabbitmq.username	Username in RabbitMQ
spring.rabbitmq.password	Password in RabbitMQ
spring.rabbitmq.virtualHost	Virtual host in RabbitMQ
spring.rabbitmq.addresses	Comma-separated list of server addresses (in cluster)

Mail

Boot support for sending email messages is enabled by importing **spring-boot-starter-mail**. This auto-configures a **JavaSendMail** bean if property **spring.mail.host** is defined. Table below summarizes Boot properties for Mail.

Property	Description (*Default*)
spring.mail.host	Host of SMTP server
spring.mail.port	Port of SMTP server
spring.mail.user	Username in SMTP server
spring.mail.password	Password in SMTP server
spring.mail.defaultEncoding	Default char encoding (*UTF-8*)

Distributed Transactions

Boot support auto-configuration of JTA distributed transactions, using either an JEE application-server or an embedded standalone JTA provider. Providers Atomikos and Bitronix are supported, using starters **spring-boot-starter-jta-atomikos** and **spring-boot-starter-jta-bitronix**, respectivelly. Instances of JTA's **UserTransaction** and **TransactionManager** (package **javax.transaction.***) are automatically created. Beans of type **XADataSourceWrapper** and **XAConnectionFactoryWrapper** are also auto-configured as XA-enabled **DataSource** and **ConnectionFactory**. For deployments in a JEE application-server, a JNDI lookup is performed on names **java:comp/UserTransaction** and **java:comp/TransactionManager**. Table below summarizes Boot properties for JTA transactions. Further customization of Atomikos and Bitroniz can be done by defining beans of type **AtomikosProperties** and **bitronix.tm.Configuration**.

Property	Description (*Default*)
spring.jta.log-dir	Log dir for JTA provider (*transaction-logs*)
spring.jta.transaction-manager-id	Distributed-UUID for JTA transaction manager (*IP*)
spring.jta.enabled	JTA enable/disable flag (*true*)

Spring Integration

Basic auto-configuration of Spring Integration is done by importing **spring-boot-starter-integration** (which makes optional annotation **@EnableIntegration**). If dependency **spring-integration-jmx** is also imported, message processing statistics are also published over JMX (unless property **spring.jmx.enabled=false**).

JMX

Boot core will automatically auto-configures a **MBeanServer** and a **MBeanExporter** – that exports to the JMX server all beans annotated with **@ManagedResource**. A default naming strategy for MBeans is used, unless a bean of type **ObjectNamingStrategy** is defined. Setting property **spring.jmx.enabled=false**, disables this JMX auto-configuration behavior.

Testing

Integration testing of Boot enabled applications is done with usual mechanisms of a Spring app, including the **SpringJUnit4ClassRunner**, but annotation **@SpringApplicationConfiguration** should be be used to specify configuration resources (rather than the usual

@ContextConfiguration). To perform the integration test in a real embedded Servlet container use annotation **@WebIntegrationTest** (or **@IntegrationTest** with **@WebAppConfiguration**). To use a mock Servlet container (**MockServletContext**) use only **@WebAppConfiguration**.

» Example: Web Integration Testing (Embedded Server)

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MyApp.class)
@WebIntegrationTest
public class MyIntegrationTests { … }
```

Actuator - Production-Ready Features

Spring Boot provides several monitoring and management features useful in production-ready applications, including: configuration info, health-checks, metrics-gathering, and remote-shell. Importing **spring-boot-starter-actuator** enables all these features, and exports HTTP endpoints to access the collected info. By default, all HTTP endpoints are enabled, but they can be individually disabled with **endpoints.*name*.enabled=false** or globally with **endpoints.enabled=false**. Table below summarizes some of these HTTP endpoints.

Endpoint	Description
/info	Generic app info
/metrics	Get metrics info
/health	Get health info
/trace	Get trace of last HTTP requests
/mappings	Get list of mapped HTTP endpoints
/autoconfig	Get list of auto-configured beans
/configprops	Get list of auto-configured properties

Resources

- Spring Boot project: http://projects.spring.io/spring-boot/
- Spring Boot Reference Documentation: http://docs.spring.io/spring-boot/docs/1.3.0.BUILD-SNAPSHOT/reference/htmlsingle/
- Spring Boot GitHub repository: https://github.com/spring-projects/spring-boot

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org