Spring Boot from Kaushik

With spring boot we can actually have easy to start spring application.

Spring boot is a tool which lets you bootstrap a spring application from the scratch.

Definition from spring boot reference guide –

*Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you*

*can “just run”.*

Here production grade means it is not hello world application , but it is end to end application

Problems with spring

* Huge framework
* Multiple setup steps
* Multiple configuration steps
* Multiple build and deploy steps

Wouldn’t it be cool if we abstract these steps.All infrastructure concerns are abstracted away. All setups and configuration steps are abstracted out so that you just focus on your business logic.Is someone could tell you that for 80% of the cases this is the right way of doing it, and then for rest 20% cases we can configure it and tweak it a little bit so that it works differently. So if you address majority of use cases, you have some kind of opinionated framework.

**1.5 What spring boot gives us**

1. Opinionated - spring boot is opinionated which means it gives you a starting point meaning “start with this and configure other things if required”
2. Convention over configuration –If you belong to 80% use cases then no need of configuration. Only if you are in 20% category you want something different only then add configuration.
3. Standalone- You generate the spring application out of spring boot. Typically when you build a normal spring application you build a WAR file and deploy it in Tomcat/any servlet container. But with spring boot we have application that just runs and starts tomcat. You don’t have to find a servlet container to deploy to. configure it, deploy it- nothing is required.
4. Production Ready:You don’t have to do anything extra to get it ready for production. Just take that and deploy to production.

**1.6 Setting up development environment.**

**Maven –**

* download jars when you add dependency in pom.xml
* lets you create a starter project ,you don’t have to create individual files yourself, Maven gives you a template(archetype) which lets you start a simple project.

**1.8 Creating a spring boot application**

There are multiple ways in which we can create a spring application using spring boot.

Step 1. Add spring boot starter as a parent

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.3.RELEASE</version>

</parent>

The concept is- you can have a configuration defined in parent maven project and have the child project inherit the configuration.

Hence spring-boot-starter-parent has all default(opinionated set of) maven configurations . So now your project being child of this starter project inherits all default maven configurations.

**Step 2**. Spring boot provides meta dependency which means wrapper dependency which pulls in all dependency jars required for a web project so that we donot need to add each one of them one by one.

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**Step 3** Also add below property in properties section

<properties>

<java.version>1.8</java.version>

</properties>

**1.9 Starting a Spring boot application**

**Step 4.** Create a java class with main method. This standalone java program will create a servlet container, starts it and hosts your application.

**Step 5:** Annotate it with @SpringBootApplication- This tells spring boot that this is the starting point for our spring boot application.

**Step 6:**

Add the content to main method -

@SpringBootApplication

Public class CourseApiApp{

PSVM(String [] args){

SpringApplication.run(CourseApiApp.class,args);

**}**

**}**

**1.10**

SpringApplication is a static class and has a static method run which takes two arguments .One is the app class which is annotated with spring annotation. and other are pass thru arguments.

Spring annotation does the below things -

1. Setup default configuration ..addresses 80 % of cases.Spring boot is convention over configuration.
2. Starts Spring application context. In a normal spring application you have controllers, business services, data services. Spring acts as a container for them. This container is called application context.
3. Performs class path scan. Spring scans the class path and looks for annotations e. g @Service @Controller so that they can treated in right way. Spring infers all that from the annotation and does the right thing.
4. Starts tomcat server.

**1.11 Adding a rest controller.**

Controller-A Java class marked with annotation that has info about –

* What URL access triggers it
* What method to run when accessed.

The web layer in spring boot application leverages a spring MVC module. Since you have placed the class in classpath, There is no need of extra plugging. Spring f/w scans it while starting up. Finds such annotations and registers it.

@RestController

Public class HelloController {

@RequestMapping(“/hello”)

Public String sayHi(){

Return “hi”;

}

}

**1.12 Returning Objects from comtroller**

@RequestMapping(“/topics”)

Public list<Topic> getTopics{}

List<Topic> is going to get converted to JSON, since the class is annotated with @RestController. The generated JSON has key names corresponding to the property names of the Topic class.The JSON values are the values of these properties.We get array in JSON format. This JSON conversion is done by spring MVC.

**1.13 Whats happening here :Bill of materials**

Spring-boot-starter-parent provides preset list of possible combination of jars.The version number which you mentioned for it

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.3.RELEASE</version>

</parent>

Downloads accordingly the jars with compatible(that work well) version – BILL OF MATERIALS.You know that there are certain combination of jars and their versions that work well.You don’t have to worry about individual version numbers anymore. You just have to worry about version number of parent that itself instructs maven that what are the different combinations that do work well.

**1.14 WHats happening here: Embedded Servlet Container**

Convenience -No need to download Tomcat, deploy or all that stuff.

Servlet container config is now application config –Earlier you needed to do some servlet configuration in tomcat to have it run your application, but now it has become part of application config. Whatever configuration you add as a part of application

Standalone application

Useful for microservices architecture.

**2.1 How Spring MVC works**

In the application when we added springboot starter web which added spring MVC to our app. Without controller we get error from tomcat while hitting the request .So, we built controller which are simple java classes that map a URI and http method (get or post) to some functionality(java method).Hence if we have multiple controllers., spring MVC will look for the controllers in the class path and will find the appropriate controller which has URI mapping of that of incoming request and the corresponding java method is executed and whatever is the return type , it converts it into proper response and sends it back. Hence java values to JSON conversion is handled automatically by spring MVC.

**2.2 The REST Api we will build**

In order to establish for your API, you need to first identify resources.And then identify the ways in which consumers can access these resources using different HTTP methods.

e.g in our case resources are Topic,Lesson,Course.

A topic can have multiple courses and a course can have multiple lesssons.These are nouns/entities in your system.

Take on resource for now, say Topic. Operations that can be performed are –

The standard REST conventions are -

**GET**- /topics – gets all topics

**GET** -/topics/id –Gets the topic

**POST** /topics-create a new topic.

**PUT** /topics/id- updates the topic

**DELETE** /topics/id – deletes the topic

<https://www.mkyong.com/spring-boot/spring-boot-which-main-class-to-start/>

**To identify start class -**If Spring Boot project contains multiple main classes, Spring Boot will fail to start or package for deployment.

1.1 Define single main class via start-class properties

pom.xml

<properties>

<!-- The main class to start by executing java -jar -->

<start-class>com.mkyong.SpringBootWebApplication</start-class>

</properties>

1.2 Alternatively, define the main class in the spring-boot-maven-plugin

pom.xml

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<mainClass>com.mkyong.SpringBootWebApplication</mainClass>

</configuration>

</plugin>

</plugins>

</build>

To create an executable jar we need to add the spring-boot-maven-plugin to our pom.xml. Insert the following lines just below the dependencies section:

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

The spring-boot-starter-parent POM includes <executions> configuration to bind the repackage goal. If you are not using the parent POM you will need to declare this configuration yourself.

|  |  |
| --- | --- |
|  | <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <version>1.1.3.RELEASE</version>  <configuration>  <mainClass>my.package.MyStartClass</mainClass>  <layout>ZIP</layout>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions>  </plugin> |

In other words

The spring-boot-maven-plugin can be used to create an executable ‘fat’ JAR. If you are using the spring-boot-starter-parent POM you can simply declare the plugin and your jars will be repackaged:

**<build>**

**<plugins>**

**<plugin>**

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

**<artifactId>**spring-boot-maven-plugin**</artifactId>**

**</plugin>**

**</plugins>**

**</build>**

If you are not using the parent POM you can still use the plugin, however, you must additionally add an <executions> section:

**<build>**

**<plugins>**

**<plugin>**

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

**<artifactId>**spring-boot-maven-plugin**</artifactId>**

**<version>**1.5.6.RELEASE**</version>**

**<executions>**

**<execution>**

**<goals>**

**<goal>**repackage**</goal>**

**</goals>**

**</execution>**

**</executions>**

**</plugin>**

**</plugins>**

**</build>**

To see the jar use command-

$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar

Important annotations

@EnableConfigurationProperties

@Configuration

@ComponentScan

@EnableTransactionManagement

@EnableJpaRepositories

@EnableJpaAuditing

@Autowired

@Conditional(WorkerApiServiceAppOnlyCondition.**class**)

@EnableWebMvc

@EnableAutoConfiguration (exclude = { DataSourceAutoConfiguration.**class** })

To do -7.10 in spring f/w reference

Auto-configuration is implemented with standard @Configuration classes. Additional

@Conditional annotations are used to constrain when the auto-configuration should apply

1. how to conditionally register a bean based on System Property.

@Configuration

public class AppConfig

{

@Bean

@Conditional(MySQLDatabaseTypeCondition.class)

public UserDAO jdbcUserDAO(){

return new JdbcUserDAO();

}

@Bean

@Conditional(MongoDBDatabaseTypeCondition.class)

public UserDAO mongoUserDAO(){

return new MongoUserDAO();

}

}

We can implement the Condition **MySQLDatabaseTypeCondition**to check whether the System Property **dbType**is **"MYSQL"** as follows:

public class MySQLDatabaseTypeCondition implements Condition

{

@Override

public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata)

{

String enabledDBType = System.getProperty("dbType");

return (enabledDBType != null && enabledDBType.equalsIgnoreCase("MYSQL"));

}

}

We can implement the Condition **MongoDBDatabaseTypeCondition**to check whether the System Property **dbType**is "**MONGODB**" as follows:

public class MongoDBDatabaseTypeCondition implements Condition

{

@Override

public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata)

{

String enabledDBType = System.getProperty("dbType");

return (enabledDBType != null && enabledDBType.equalsIgnoreCase("MONGODB"));

}

}

public interface UserDAO

{

List<String> getAllUserNames();

}

public class JdbcUserDAO implements UserDAO

{

@Override

public List<String> getAllUserNames()

{

System.out.println("\*\*\*\* Getting usernames from RDBMS \*\*\*\*\*");

return Arrays.asList("Siva","Prasad","Reddy");

}

}

public class MongoUserDAO implements UserDAO

{

@Override

public List<String> getAllUserNames()

{

System.out.println("\*\*\*\* Getting usernames from MongoDB \*\*\*\*\*");

return Arrays.asList("Bond","James","Bond");

}

}

If we run the application like**java -jar myapp.jar -DdbType=MYSQL** then only the **JdbcUserDAO**bean will be registered.But if you set the System property like **-DdbType=MONGODB** then only the **MongoUserDAO**bean will be registered.

1. Suppose we want to register **MongoUserDAO**bean only when **MongoDB**java driver class **"com.mongodb.Server"** is available on classpath, if not we want to register **JdbcUserDAO**bean.then matches method will have below implementation –

public class MongoDriverPresentsCondition implements Condition

{

@Override

public boolean matches(ConditionContext conditionContext,AnnotatedTypeMetadata metadata)

{

try {

Class.forName("com.mongodb.Server");

return true;

} catch (ClassNotFoundException e) {

return false;

}

}

}

public class MongoDriverNotPresentsCondition implements Condition

{

@Override

public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata)

{

try {

Class.forName("com.mongodb.Server");

return false;

} catch (ClassNotFoundException e) {

return true;

}

}

}

1. What if we want to register the **MongoUserDAO**bean only if no other Spring bean of the type **UserDAO**is already registered.

public class UserDAOBeanNotPresentsCondition implements Condition

{

@Override

public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata)

{

UserDAO userDAO = conditionContext.getBeanFactory().getBean(UserDAO.class);

return (userDAO == null);

}

}

@Configuration

**public class** MyAutoConfiguration {

@Bean

@ConditionalOnMissingBean

**public** MyService myService() { ... }

}

In the example above, the myService bean is going to be created if no bean of type MyService is already contained in the ApplicationContext.

But there is even more elegant way to implement Conditions using Annotations. Instead of creating a Condition implementation for both MYSQL and MongoDB, we can create aDatabaseType annotation as follows:

@Target({ ElementType.TYPE, ElementType.METHOD })

@Retention(RetentionPolicy.RUNTIME)

@Conditional(DatabaseTypeCondition.class)

public @interface DatabaseType

{

String value();

}

Then we can implement **DatabaseTypeCondition**to use the **DatabaseType**value to determine whether to enable or disable bean registration as follows:

public class DatabaseTypeCondition implements Condition

{

@Override

public boolean matches(ConditionContext conditionContext,

AnnotatedTypeMetadata metadata)

{

Map<String, Object> attributes = metadata.getAnnotationAttributes(DatabaseType.class.getName());

String type = (String) attributes.get("value");

String enabledDBType = System.getProperty("dbType","MYSQL");

return (enabledDBType != null && type != null && enabledDBType.equalsIgnoreCase(type));

}

}

Now we can use the **@DatabaseType** annotation on our bean definitions as follows:

@Configuration

@ComponentScan

public class AppConfig

{

@DatabaseType("MYSQL")

public UserDAO jdbcUserDAO(){

return new JdbcUserDAO();

}

@Bean

@DatabaseType("MONGO")

public UserDAO mongoUserDAO(){

return new MongoUserDAO();

}

}

Spring boot provides various auto configuration classes.

@Configuration

@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })

@EnableConfigurationProperties(DataSourceProperties.class)

@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })

public class DataSourceAutoConfiguration

{

Typically **AutoConfiguration**classes are annotated with

1. **@Configuration** to mark it as a Spring configuration class and annotated with

Here,

1. **DataSourceAutoConfiguration**is annotated with **@ConditionalOnClass({ DataSource.class,EmbeddedDatabaseType.class })** which means that the AutoConfiguration of beans within **DataSourceAutoConfiguration**will be considered only if the **DataSource.class**and **EmbeddedDatabaseType.class**classes are available on classpath.
2. The class is also annotated with **@EnableConfigurationProperties(DataSourceProperties.class)**which enables binding the properties in **application.properties** to the properties of **DataSourceProperties** class automatically.

@ConfigurationProperties(prefix = DataSourceProperties.PREFIX)

public class DataSourceProperties implements BeanClassLoaderAware, EnvironmentAware, InitializingBean {

public static final String PREFIX = "spring.datasource";

...

...

private String driverClassName;

private String url;

private String username;

private String password;

...

//setters and getters

}

With this configuration all the properties that starts with **spring.datasource.\*** will be automatically binds to **DataSourceProperties**object.

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=secret

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

Other important file inside spring-boot-autoconfigure.jar is /META-INF/spring.factories. This file lists all the auto configuration classes that should be enabled under the EnableAutoConfiguration key.

If you don’t like application.properties as the configuration file name you can switch to

another by specifying a spring.config.name environment property. You can also refer to an

explicit location using the spring.config.location environment property

$ java -jar myproject.jar --spring.config.name=myproject

SpringApplication will load properties from application.properties files in the following

locations and add them to the Spring Environment:

1. A /config subdirectory of the current directory.

2. The current directory

3. A classpath /config package

4. The classpath root

The list is ordered by precedence (properties defined in locations higher in the list override those defined

in lower locations).

Spring Boot 1.3 has introduced devtools, a module to improve the development-time experience when working on Spring Boot applications. To enable it, just add the following dependency to your project:

<dependencies>

<dependency>

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

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

<version>1.5.6.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

an uber-jar is an "over-jar", one level up from a simple "jar", defined as one that contains both your package and all its dependencies in one single JAR file.

The advantage is that you can distribute your uber-jar and not care at all whether or not dependencies are installed at the destination, as your uber-jar actually has no dependencies.

The easiest way to take complete control over MVC configuration is to provide your own

@Configuration with the **@EnableWebMvc** annotation. This will leave all MVC configuration in your

hands.

Test

support is provided by two modules; spring-boot-test contains core items, and spring-boottest-

autoconfigure supports auto-configuration for tests. Most developers will just use the spring-boot-starter-test ‘Starter’ which imports both Spring

Boot test modules as well has JUnit, AssertJ, Hamcrest and a number of other useful libraries

A new lifecycle option is available for intercepting @ResponseBody and ResponseEntity methods

just after the controller method returns and before the response is written. To take advantage declare

an @ControllerAdvice bean that implements ResponseBodyAdvice

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*KT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Vantage decoupling

microservices

Are your web services individually deployable? Individually scalable? Individually monitored?

while web services is an implementation of the service-oriented architecture pattern.

So, SOA is an architectural pattern in which application components provide services to other components. However, in SOA those components can belong to the same application. On the other hand, in microservices these components are suites of independently deployable services.

Microservices is the architectural specialisation of SOA, driven by DevOps practices. The individually deployable services made it easier to apply Continuous Integration / Continuous Deployment

Microservices can decouple code and resources, make builds smaller, releases safer, and APIs more stable.

-monolith comparison- UI +DB+business service layer,scaling

-independently deployable-Better deployability,Continuous deployment ,Better testability ,new team member,Improved fault isolation,Eliminates any long-term commitment to a technology stack.

Spring boot

1. Opinionated - spring boot is opinionated which means it gives you a starting point meaning “start with this and configure other things if required”

2. Convention over configuration –If you belong to 80% use cases then no need of configuration. Only if you are in 20% category you want something different only then add configuration. Autoconfiguration. Trying to minimize the amount of config a Spring app requires to get ready to go and favoring convention over configuration

3. Standalone- You generate the spring application out of spring boot. Typically when you build a normal spring application you build a WAR file and deploy it in Tomcat/any servlet container. But with spring boot we have application that just runs and starts tomcat. You don’t have to find a servlet container to deploy to. configure it, deploy it- nothing is required.

4. Production Ready:You don’t have to do anything extra to get it ready for production. Just take that and deploy to production.

1)spring boot starter parent takes care of versions of all spring dependencies. Hence spring-boot-starter-parent has all default(opinionated set of) maven configurations . So now your project being child of this starter project inherits all default maven configurations.

2)starters-

You should need to specify only the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to upgrade to another Spring Data release train, you would add the following to your pom.xml:

<properties>

<spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>

</properties>

3)Spring Boot Maven plugin

It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.

It searches for the public static void main() method to flag as a runnable class.

It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot’s chosen set of versions.

4). Create a java class with main method. Annotate it with @SpringBootApplication-

SpringApplication.run(CourseApiApp.class,args);

a)Starts Spring application context, Performs class path scan. Spring scans the class path and looks for annotations e. g @Service @Controller so that they can treated in right way. Spring infers all that from the annotation and does the right thing.

b)Starts tomcat server.

@enableautoconfiguration - annotation auto-configures the beans that are present in the classpath. registers the beans that are matching various Conditions. e.g JmxAutoConfiguration ,DataSourceAutoConfiguration,DispatcherServletAutoConfiguration

@SpringBootApplication which combines the three annotations @Configuration,@EnableAutoConfiguration and code>@ComponentScan.

While the traditional MVC controller relies on the View technology, the RESTful web service controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.

Spring lets you return data directly from the controller, without looking for a view, using the @ResponseBody annotation on a method. Beginning with Version 4.0, this process is simplified even further with the introduction of the @RestController annotation.

When you use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the http response automatically. Each method in the Controller class must be annotated with @ResponseBody.

By annotating the controller class with @RestController annotation, you no longer need to add @ResponseBody to all the request mapping methods