Whats Spring Boot?

Spring Boot enables developers to focus on the application logic rather than being bogged down by the intricacies of configuration.

Spring has always prioritised convention over configuration as a model for simpler programming and Spring Boot P oject emphasizes a similar discipline.

Specifically, there are four main features that come with Spring Boot Project:
1. Starter Dependencies
2. Automatic Configuration
3. CLI
4. The Actuator

Why Spring Boot?

The problem in Spring framework is that there are too many on the verge of making an application development den e with numerous configuration codes.

If we consider EJB a mess of heavyweight components, the Spring framework is definitely a mess of configuration.

Apart from these, meeting the right dependency for the project is another tricky problem.

So, Spring has to solve not only the configuration issues but also the problem associated with library dependencies.

In most project development, we heavily need boilerplated code, such as a project structure with similar dependenci s defined with Maven or Gradle.

And, the project falls into one of the many known categories which require dependencies such as Spring MVC, Serv et API, JDBC, ORM, JPA, and so forth.

If it is a Web application, we need an XML file to initiate the application, a controller class that responds to the HTT request, and a Web application server such as Tomcat to deploy the application.

There is actually very little code that is new to the application; the rest are repetitive, reusable, boilerplate code.

So, Spring thought why not bootstrap them; provide these functionality behind the scene with minimal user's interve tion as possible.

This is exactly what the Spring Boot project is all about.

Whats Boilerplate Code?

Boilerplate code or boilerplate refers to sections of code that have to be included in many places with little or no alte ation.

What are the advantages of Spring Boot?

Spring Boot enables developers to focus more on the business logic of the application than project infrastructure, wh ch is taken care of by Spring Boot.

For example, Spring Boot automatically finds the specific beans declared in the project.

There is no need to configure them explicitly; it automatically embeds Tomcat as the Web application server.

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

The Spring Boot Project provides four key features to begin it.

They are typically called: starter dependencies, CLI, Automatic configuration, and the actuator.

Let's get a brief overview on each of them

Starter Dependencies

The starters are basically a set of dependency descriptors tagged under a single banner, called starter name, such as s ring-boot-starter-web.

This starter includes all the dependent libraries required for developing a Spring Web application. Additional depend ncies may be added, but in most cases the starter is sufficient for a particular category of project.

Also, there is no harm in using more than one starter in pom.xml. Similarly, there is a starter called spring-boot-start r-test.

This starter automatically includes almost all the libraries usually required for testing: Spring Test, JUnit, Hamcrest, nd Mockito.

Although dependencies can be added manually, Spring Boot Starters are rather more convenient.

For example, we can add spring-boot-starter-web for Web application development as follows.

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

Adding spring-boot-starter-test:

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

Adding Data JPA starter with embedded h2 database:

```
<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>
```

CLI (Command Line Interface)

Spring Boot provides a command line tool, called CLI (Command Line Interface), to quickly prototype a Spring app ication using Groovy Scripts.

As mentioned, Spring Boot CLI is ideal for quick prototyping; production grade applications are rarely created using Spring Boot CLI.

To use Spring Boot CLI, one needs to install a CLI distribution and create a Groovy file of the required application.

Automatic Configuration

Autoconfiguration is enabled with @EnableAutoConfiguration annotation

This feature tries to automatically configure the application based upon the dependent libraries added to the project c ass path.

Automatic configuration means. Spring Boot implicitly scans the application class path and detects the required data ase library and provides the necessary configuration to use it.

If part of the code includes JdbcTemplate, it is also automatically configured. An automatic configuration scheme is ot restricted to database use only.

The Actuator

The actuator basically enables inspection of a production-grade application by enabling auditing, health monitoring, nd metric gathering features.

The other Spring Boot features are primarily targeted towards development whereas the actuator exposes the internal runtime operational information, such as:

Beans configured in the Spring Application Context
Spring Boot's auto-configuration
Available environment variables, system and configuration properties, and the like
Trace of a recent HTTP request
Metrics of memory usage, garbage collection, data source usages, or Web request

To enable the actuator, we may add the dependency as follows:

```
<dependency>
```

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

Embedded server

Spring boot applications always include tomcat as embedded server dependency.

You can exclude tomcat and include any other embedded server if you want.

```
<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 Boot Annotations

The spring boot annotations are mostly placed in org.springframework.boot.autoconfigure and org.springframework.
oot.autoconfigure.condition packages.

1. @SpringBootApplication

@SpringBootApplication annotation enable all able things in one step. It enables the three features:

@EnableAutoConfiguration : enable auto-configuration mechanism
@ComponentScan : enable @Component scan
@SpringBootConfiguration : register extra beans in the context

The java class annotated with @SpringBootApplication is the main class of a Spring Boot application and applicatio
starts from here.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

```
    }
}
```

2. @EnableAutoConfiguration

This annotation enables auto-configuration of the Spring Application Context, attempting to guess and configure bea s that we are likely to need based on the presence of predefined classes in classpath.

As this annotation is already included via @SpringBootApplication, so adding it again on main class has no impact.

It is also advised to include this annotation only once via @SpringBootApplication.

3. @SpringBootConfiguration

It indicates that a class provides Spring Boot application configuration. It can be used as an alternative to the Spring' standard @Configuration annotation so that configuration can be found automatically.

Application should only ever include one @SpringBootConfiguration and most idiomatic Spring Boot applications ill inherit it from @SpringBootApplication.

4. @ImportAutoConfiguration

It import and apply only the specified auto-configuration classes.

The difference between @ImportAutoConfiguration and @EnableAutoConfiguration is that later attempts to config re beans that are found in the classpath during scanning, whereas @ImportAutoConfiguration only runs the configur tion classes that we provide in the annotation.

@ImportAutoConfiguration example

```
@ComponentScan("path.to.your.controllers")
@ImportAutoConfiguration({WebMvcAutoConfiguration.class
   ,DispatcherServletAutoConfiguration.class
   ,EmbeddedServletContainerAutoConfiguration.class
   ,ServerPropertiesAutoConfiguration.class
   ,HttpMessageConvertersAutoConfiguration.class})
public class App
{
   public static void main(String[] args)
   {
      SpringApplication.run(App.class, args);
   }
}
```

5. @AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder
---
We can use the @AutoConfigureAfter or @AutoConfigureBefore annotations if our configuration needs to be appli d in a specific order (before of after).

If we want to order certain auto-configurations that should not have any direct knowledge of each other, we can also se @AutoConfigureOrder

@AutoConfigureAfter Example

```
@Configuration
@AutoConfigureAfter(CacheAutoConfiguration.class)
public class RedissonCacheStatisticsAutoConfiguration
{
   @Bean
   public RedissonCacheStatisticsProvider redissonCacheStatisticsProvider(){
      return new RedissonCacheStatisticsProvider();
   }
}
```

6. Condition Annotations

6.1. @ConditionalOnBean and @ConditionalOnMissingBean

These annotations let a bean be included based on the presence or absence of specific beans.

In below example, bean JpaTransactionManager will only be loaded if a bean of type JpaTransactionManager is not lready defined in the application context.

```
@Bean
@ConditionalOnMissingBean(type = "JpaTransactionManager")
JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory)
{
   JpaTransactionManager transactionManager = new JpaTransactionManager();
   transactionManager.setEntityManagerFactory(entityManagerFactory);
   return transactionManager;
}
```

6.2. @ConditionalOnClass and @ConditionalOnMissingClass

These annotations let configuration classes be included based on the presence or absence of specific classes.

```
@Configuration
@ConditionalOnClass(EmbeddedAcmeService.class)
static class EmbeddedConfiguration
{

   @Bean
   @ConditionalOnMissingBean
   public EmbeddedAcmeService embeddedAcmeService() { ... }

}

@Configuration
@ConditionalOnClass(EmbeddedAcmeService.class)
static class EmbeddedConfiguration
{

   @Bean
   @ConditionalOnMissingBean
```

```
    public EmbeddedAcmeService embeddedAcmeService() { ... }

}
```

6.3. @ConditionalOnNotWebApplication and @ConditionalOnWebApplication
---
These annotations let configuration be included depending on whether the application is a “web application” or not.

6.4 @ConditionalOnProperty

This annotation lets configuration be included based on the presence and value of a Spring Environment property.

```
@Bean
@ConditionalOnProperty(name = "env", havingValue = "local")
DataSource dataSource()
{
    // ...
}

@Bean
@ConditionalOnProperty(name = "env", havingValue = "prod")
DataSource dataSource()
{
    // ...
}
```

6.5. @ConditionalOnResource

This annotation lets configuration be included only when a specific resource is present in the classpath. Resources ca be specified by using the usual Spring conventions.

```
@ConditionalOnResource(resources = "classpath:vendor.properties")
Properties additionalProperties()
{
    // ...
}
```

-Ddebug=true as VM argument

Now when you run the application, you will get lots of debug logs having similar information :

DataSource Configuration

Maven

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

<dependency>
    <groupId>com.h2database</groupId>
```

```
    <artifactId>h2</artifactId>
    <version>2.4.1</version>
    <scope>runtime</scope>
</dependency>
```

application.properties

```
# H2
spring.datasource.url=jdbc:h2:file:C:/temp/test
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# MySQL
#spring.datasource.url=jdbc:mysql://localhost:3306/test
#spring.datasource.username=dbuser
#spring.datasource.password=dbpass
#spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect

# Oracle
#spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
#spring.datasource.username=dbuser
#spring.datasource.password=dbpass
#spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
#spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect

# SQL Server
#spring.datasource.url=jdbc:sqlserver://localhost;databaseName=springbootdb
#spring.datasource.username=dbuser
#spring.datasource.password=dbpass
#spring.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
#spring.jpa.hibernate.dialect=org.hibernate.dialect.SQLServer2012Dialect
```

DataSource Bean

```
@Configuration
public class JpaConfig {

    @Bean
    public DataSource getDataSource()
    {
        DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
        dataSourceBuilder.driverClassName("org.h2.Driver");
        dataSourceBuilder.url("jdbc:h2:file:C:/temp/test");
        dataSourceBuilder.username("sa");
        dataSourceBuilder.password("");
        return dataSourceBuilder.build();
    }
}
```

Multiple Datasources with Spring boot

```
@Configuration
public class JpaConfig {

   @Bean(name = "h2DataSource")
   public DataSource h2DataSource()
   {
      DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
      dataSourceBuilder.driverClassName("org.h2.Driver");
      dataSourceBuilder.url("jdbc:h2:file:C:/temp/test");
      dataSourceBuilder.username("sa");
      dataSourceBuilder.password("");
      return dataSourceBuilder.build();
   }

   @Bean(name = "mySqlDataSource")
   @Primary
   public DataSource mySqlDataSource()
   {
      DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
      dataSourceBuilder.url("jdbc:mysql://localhost/testdb");
      dataSourceBuilder.username("dbuser");
      dataSourceBuilder.password("dbpass");
      return dataSourceBuilder.build();
   }
}
```

Autowire primary datasource

```
@Autowired
DataSource dataSource;
```

Autowire NON-primary datasource

```
@Autowired
@Qualifier("h2DataSource")
DataSource dataSource;
```

Spring boot security

Learn to use basic authentication to secure rest apis created inside a Spring boot application.

The secured rest api will ask for authentication details before giving access the data it secure.

First step is to include required dependencies e.g. spring-boot-starter-security.

Second step is to configure WebSecurityConfigurerAdapter and add auth details.

Maven dependency

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

Configure WebSecurityConfigurerAdapter

To enable authentication and authorization support in spring boot rest apis, we can configure a utility class WebSecu ityConfigurerAdapter.

It helps in requiring the user to be authenticated prior to accessing any configured URL (or all urls) within our applic tion.

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter
{
   @Override
   protected void configure(HttpSecurity http) throws Exception
   {
      http
       .csrf().disable()
       .authorizeRequests().anyRequest().authenticated()
       .and()
       .httpBasic();
   }

   @Autowired
   public void configureGlobal(AuthenticationManagerBuilder auth)
         throws Exception
   {
      auth.inMemoryAuthentication()
         .withUser("admin")
         .password("{noop}password")
         .roles("USER");
   }
}
```

Access rest api with 'authorization' header

Upon passing authorization request header with encoded basic-auth user name and password combination, we will b able to access the rest api response.

Logging in Spring Boot

Logging in spring boot is very flexible and easy to configure. Spring boot supports various logging providers throug some simple configuration.

Default Zero Configuration Logging

Spring boot active enabled logging is determined by spring-boot-starter-logging artifact and it’s auto-configuration

hich enables anyone of supported logging providers (Java Util
Logging, Log4J2, and Logback) based on configuration provided.

If we do not provide any logging specific configuration, we will still see logs printed in “console”. These are becaus
of default logging support provided in spring boot which uses Logback.

Maven Dependency

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-log4j2</artifactId>
    </dependency>
```

log4j2.properties
---
log4j2.properties
```
status = error
name = PropertiesConfig

#Make sure to change log file path as per your need
property.filename = C:\\logs\\debug.log

filters = threshold

filter.threshold.type = ThresholdFilter
filter.threshold.level = debug

appenders = rolling

appender.rolling.type = RollingFile
appender.rolling.name = RollingFile
appender.rolling.fileName = ${filename}
appender.rolling.filePattern = debug-backup-%d{MM-dd-yy-HH-mm-ss}-%i.log.gz
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
appender.rolling.policies.type = Policies
appender.rolling.policies.time.type = TimeBasedTriggeringPolicy
appender.rolling.policies.time.interval = 1
appender.rolling.policies.time.modulate = true
appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
appender.rolling.policies.size.size=10MB
appender.rolling.strategy.type = DefaultRolloverStrategy
appender.rolling.strategy.max = 20

loggers = rolling

#Make sure to change the package structure as per your application

logger.rolling.name = com.emexo
logger.rolling.level = debug
logger.rolling.additivity = false
logger.rolling.appenderRef.rolling.ref = RollingFile
```

Log4j2HelloWorldExample.java

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4j2HelloWorldExample
{
   private static final Logger LOGGER = LogManager.getLogger(Log4j2HelloWorldExample.class.getName());

   public static void main(String[] args)
   {
      LOGGER.debug("Debug Message Logged !!!");
      LOGGER.info("Info Message Logged !!!");
      LOGGER.error("Error Message Logged !!!", new NullPointerException("NullError"));
   }
}
```

Logging Level

ERROR, WARN, INFO, DEBUG

MVC Annotations

@RestController
@Controller
@ResponseBody

@RequestMapping

@PathVariable http://localhost:8080/emp/employees/10
@RequestParam http://localhost:8080/emp/employees?id=10&name=Regu
@RequestBody

HTTP Methods

Get
Post
put
Delete

Spring RestTemplate – Spring REST Client
