Spring Boot

Agenda

- What is Spring Boot
- Why Spring Boot
- How Spring Boot

What is Spring?

- It is an application framework : unlike single tire framework like hibernate, struts.
- It's the ***only framework to address all architectural tiers*** of typical j2ee application
- It also offers a comprehensive range of service as well as lightweight container

Spring Framework

- a very popular framework for building Java web and enterprise applications.
- It provides a wide variety of features addressing the modern business needs(via its portfolio projects).
- Unlike many other frameworks which focus on only one area

What is Spring Boot

- Spring Boot is a project created by Spring Team to build production ready spring applications.
- Spring Boot favours convention over configuration
- It is designed to get you up and running as quickly as possible.

Introducing Spring Boot

- Spring Boot makes it easy to create stand-alone, production-grade Spring-based Applications that we can run.
- We have **an opinionated view** of the Spring platform and third-party libraries, so that we can get started with minimum fuss.
- Most Spring Boot applications need very little Spring configuration.
- We can use Spring Boot to create Java applications that can be started by using java -jar or more traditional war deployments.
- Spring provides a command line tool that runs “spring scripts”.

Introducing Spring Boot

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 (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

Configuration

- Beans : Your application objects which are managed by DI/IOC spring container
- Bean definition : Configuration metadata that is used by container to manage your beans
- Info required by container :

  [1] How to create a bean

  [2] Bean's lifecycle

  [3] Bean's dependencies.

Three ways of Configuration

- Beans Configuration — *XML*
- Beans Configuration — *Annotation*

  Annotation configuration is not turned on by default. To enable it we add following to our Spring configuration file

  <context:component-scan base-package= "com.demo. “ >
  </context:component-scan>

- Beans Configuration — *Java Code*

Java Configuration

@Configuration:

- tells the Spring IoC container to use it as a source of bean definitions.(java class)

- So container can process the class and generate & manage beans to be used in the application.

- This annotation is part of the spring core framework.

@Configuration

```
package com.sj.spring;

import org.springframework.context.annotation.Bean;
Import org.springframework.context.annotation.Configuration;

@Configuration
public class MyConfiguration {

    @Bean
    public MyBean myBean() {
                                return new MyBean();
                }

}
```

Configuration classes

Spring Boot favors Java-based configuration:

- *it is possible to call SpringApplication.run() with an XML source,*
- *However recommended is that primary source is a **@Configuration class**.*
- *Usually the class defines **the main method** is also a good candidate as the primary **@Configuration**.*

Configuration classes

Importing additional configuration classes

- *No need to put all @Configuration into a single class.*
- *The* @Import *annotation can be used to import additional configuration classes.*
- *Alternatively, we can use* @ComponentScan *to automatically pick up all Spring components, including* @Configuration *classes.*

Importing XML configuration

- *If its absolutely must, use XML based configuration, still better to start with a* @Configuration *class.*
- *You can then use an additional* @ImportResource *annotation to load XML configuration files.*

Why Spring Boot

Spring :

- provides flexibility to configure beans in multiple ways such as: **XML**, **Annotations,** and **JavaConfig**.
- With number of features increased complexity also gets increased
- configuring Spring applications becomes tedious and error-prone.

Spring Boot :

Spring Boot is created to address complexity of configuration.

Think Differently :Auto-configuration

Spring Boot auto-configuration :
Why do we need Spring Boot Auto Configuration?

- *Spring based applications have a lot of configuration.*
- *When we use Spring MVC, we need to configure :*
  - *a component scan*
  - *the dispatcher servlet*
  - *a view resolver*
  - *web JARs (for delivering static content)*
  - *and many more*

Typical ViewResolver Configuration

```
• <bean class=
  "org.springframework.web.servlet.view.InternalResour
  ceViewResolver“ >
        <property name="prefix">
          <value>/WEB-INF/views/</value>
        </property>
        <property name="suffix">
          <value>.jsp</value>
        </property>
• </bean>
 <mvc:resources mapping="/webjars/**"
      location="/webjars/"/>
```

Typical Dispatcher Servlet Configuration

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
       org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>/WEB-INF/todo-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
 </servlet-mapping>
```

Typical JPA/Hibernate Configuration

```
<bean id="dataSource" class="com.sj.MyDataSource"
    destroy-method="close">
    <property name="driverClass" value="${db.driver}" />
    <property name="jdbcUrl" value="${db.url}" />
    <property name="user" value="${db.username}" />
    <property name="password" value="${db.password}" />
 </bean>
 <jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:config/schema.sql" />
    <jdbc:script location="classpath:config/data.sql" />
 </jdbc:initialize-database>
```

Typical JPA/Hibernate Configuration

```
  <bean
     class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
           id="entityManagerFactory">
     <property name="persistenceUnitName" value="hsql_pu" />
     <property name="dataSource" ref="dataSource" />
</bean>

<bean id="transactionManager"
       class="org.springframework.orm.jpa.JpaTransactionManager">
     <property name="entityManagerFactory" ref="entityManagerFactory" />
    <property name="dataSource" ref="dataSource" />
</bean>

<tx:annotation-driven
          transaction-manager = "transactionManager"/>
```

Spring Boot: Can We Think Differently?

- Spring Boot brings in a new thought process around this.
- Can we bring more intelligence into this?
- When a Spring MVC JAR is added into an application, can we auto configure some beans automatically?
- How about auto configuring a Data Source if a Hibernate JAR is on the classpath?
- How about auto configuring a Dispatcher Servlet if a Spring MVC JAR is on the classpath?

There would be provisions to override the default auto configuration.

Auto Configuration

Spring Boot looks at

– a) Frameworks available on the CLASSPATH

– b) Existing configuration for the application.

Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks.

This is called Auto Configuration.

Auto-configuration

Spring Boot auto-configuration :

- *attempts to automatically configure our Spring application based on the jar dependencies that we have added.*

- *We need to opt-in to auto-configuration by adding the @EnableAutoConfiguration to one of our @Configuration classes.*

[Tip]
We should only ever add one @EnableAutoConfiguration annotation.
Generally recommended to add it to primary @Configuration class.

Gradually replacing auto-configuration

Auto-configuration is non-invasive:

at any point we can start to define our own configuration to replace specific parts of the auto-configuration.

(For example, if we add DataSource bean, the default embedded database support will back away.)

*If we need to find out what auto-configuration is currently being applied, and why, start application with the **--debug switch**.*

This will enable debug logs for a selection of core loggers and log an auto-configuration report to the console.

Spring Boot Auto Configuration

Where Is Spring Boot Auto Configuration Implemented?

- All auto configuration logic is implemented in spring-boot-autoconfigure.jar.
- All auto configuration logic for MVC, data, JMS, and other frameworks is present in a single JAR.

Spring Boot Auto Configuration

- Typically, all auto configuration classes look at other classes available in the classpath.

- If specific classes are available in the classpath, then configuration for that functionality is enabled through auto configuration.

Why Spring Boot

Usecase :

We want to build a Web Application with:

Spring MVC , JPA(Hibernate) and MySql DB

Various configurations-steps needed:

- Maven Dependencies
- Service/DAO layer dependencies
- Web Layer MVC dependencies
- Log4j

Why Spring Boot

Problems while doing all those configurations:

- So many configurations so can not get up and run quickly
- If we want to develop another spring web app with similar technology stack ? (copy and tweak?)
- hunt for all the **compatible libraries** for the specific Spring version and configure
- 95% of the times we configure **DataSource, EntitymanagerFactory,TransactionManager** etc beans in the same way
- Also configure SpringMVC beans like **ViewResolver**, **MessageSource** etc in the same way most of the times.

Solution : ***an automated way to do it ALL***

Why Spring Boot

Solution : ***an automated way to do it ALL***

(If Spring can automatically do it for me? : that would be awesome!!!.)

- what if Spring is capable of configuring beans automatically?
- What if we can customize automatic configuration using simple customizable properties?

So basically we want Spring to do things automatically but provide flexibility to override default configuration in a simpler way? So that is :

SPRING BOOT

BUILD ANYTHING WITH SPRING BOOT

- Spring Boot is starting point for building all Spring-based applications.
- It is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring.
    - Get started in seconds using Spring Initializr
    - Build anything - REST API, WebSocket, Web, Streaming, Tasks, and more
    - Simplified Security
    - Rich support for SQL and NoSQL

BUILD ANYTHING WITH SPRING BOOT

- Embedded runtime support - Tomcat, Jetty, and Undertow
- Developer productivity tools such as live reload and auto restart
- Curated dependencies that just work
- Production-ready features such as tracing, metrics and health status
- Works in any IDE - Spring Tool Suite, IntelliJ IDEA and NetBeans

Annotations

Spring Beans and dependency injection

We are free to use any of the standard Spring Framework techniques to define our beans and their injected dependencies.

If we structure our code as our application class in a root package, we can add @ComponentScan without any arguments*.*

All of your application components (@Component, @Service, @Repository, @Controller etc.) will be automatically registered as Spring Beans.

@SpringBootApplication

Using @SpringBootApplication annotation :

- Spring Boot developers always have their main class annotated with: @Configuration, @EnableAutoConfiguration and @ComponentScan.
- Since these annotations are so frequently used together,Spring Boot provides a convenient *@SpringBootApplication* alternative.

The ***@SpringBootApplication*** annotation is equivalent to using:

@Configuration

@EnableAutoConfiguration and

@ComponentScan

with their default attributes:

@SpringBootApplication

```
package com.example.myproject;

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);
    }

}
```

Starters
spring-boot-starter-parent

- We can inherit from spring-boot-starter-parent project *to obtain sensible defaults*.
- The parent project provides the following features:
    - Java 1.8 as the default compiler level.
    - UTF-8 source encoding.
    - A Dependency Management section, inherited from the *spring-boot-dependencies pom*, that manages the versions of common dependencies. This dependency management lets *you omit <version> tags* for those dependencies when used in your own pom.

POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.example</groupId>
        <artifactId>myproject</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <!-- Inherit defaults from Spring Boot -->
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.0.1.BUILD-SNAPSHOT</version>
        </parent>
```

POM

```
<!-- Add typical dependencies for a web application -->

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


        <!-- Package as an executable jar -->
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>
```