**Before we start**

I have listed down the challenges which we may face while setting up the environment

1) Download and set up a server (e.g. Tomcat)

2) Download and install a DB

3) Download required Spring jars

4) Download Hibernate jar compatible with the Spring version

5) Download jar for MySQL/Oracle driver

6) Setting up web.xml to use spring MVC

**Why Spring Boot**

* Provides a radically faster and widely accessible ‘getting started’ experience for all Spring development

• No clumsy XML Configuration by developers

• Provide opinionated ‘starter’ POMs to simplify your Maven configuration

• Uses project management tool such as MAVEN or GRADLE

* Helps fast development and production ready code

• Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)

**1. Easy dependency Management**

First thing to observe is we are using some dependencies named like spring-boot-starter-\*.

“95% of the times we use the same configuration.

So when you add spring-boot-starter-web dependency by default it will pull all the commonly

used libraries while developing Spring MVC applications such as spring-webmvc, jackson-json, validation-api and tomcat.

If you want to add spring-boot-starter-data-jpa dependency.

This pulls all the spring-data-jpa dependencies and also adds Hibernate libraries because the majority of the applications use Hibernate as JPA implementation.

**2. Auto Configuration**

Not only the spring-boot-starter-web adds all these libraries but also configures the commonly registered beans like DispatcherServlet, ResourceHandlers,

MessageSource etc beans with sensible defaults.

ThymeleafViewResolver beans as well automatically.

We haven’t defined any of the DataSource, EntityManagerFactory, TransactionManager etc beans but they are automatically gets created.

How? If we have any in-memory database drivers like H2 or HSQL in our class path then Spring Boot will automatically create an in-memory

DataSource and then registers EntityManagerFactory, TransactionManager beans automatically with sensible defaults. But we are using MySQL,

so we need to explicitly provide MySQL connection details. We have configured those MySQL connection details in application.properties file and SpringBoot

creates a DataSource using these properties.

**3. Embedded Servlet Container Support**

The most important and surprising thing is we have created a simple Java class annotated with some magical

annotation @SpringApplication having a main method and by running that main we are able to run the application and access it at http://localhost:8080/.

Where is the servlet container comes from?

When We add spring-boot-starter-web which pulls the spring-boot-starter-tomcat automatically

and when we run the main() method it started tomcat as an embedded container so that we don’t have to deploy our application on any externally installed tomcat server.

By the way have you observe that our packaging type in pom.xml is ‘jar’ not ‘war’. Wonderful!

Ok, but what if I want to use Jetty server instead of tomcat?

Simple, exclude spring-boot-starter-tomcat from spring-boot-starter-web and include spring-boot-starter-jetty.

**What problems are we trying to solve?**

* Developers keep re-solving the same problems.

• The point of coding is to build your business value-added services.

• Any time spent doing anything else is (usually) wasted.

* Development with Java is mostly boiler-plate.

• Web-applications are very common and require lots of configuration/setup.

• Packaging, deployment, and monitoring take time (every time!). This is a waste to your productivity

• There are too many custom variations of essentially the same configuration, project layout, and deployment (makes standardization/training hard).

**What is Spring Boot?**

Spring Boot makes it easy to create standalone, production-grade Spring based applications that you can just run.

We take an opinionated view of the platform and third-party libraries so that you can get started with minimum fuss.

**Advantages of Spring Boot**

It is very easy to develop Spring Based applications with Java.

It reduces lots of development time and increases productivity.

It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC,

Spring ORM, Spring Data, Spring Security etc.

Takes an opinionated view of building production-ready Spring applications.

Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible.

**What are some important features of using Spring Boot**

**1. Starter dependency**

This feature aggregates common dependencies together. For example, if you want to develop Spring MVC based RESTful services then instead of including Spring MVC JAR and Jackson JAR file into class path you can just specify spring-boot-web-starter and it will automatically download both those JAR files. Spring Boot comes with many such starter dependencies to improve productivity.

**2. Auto-Configuration**

This is another awesome features of Spring Boot which can configure many things for you. For example, If you are developing Spring web application and Thymeleaf.jar is present on the class path then it can automatically configure Thyme leaf template resolver, view resolver, and other settings. A good knowledge of auto-configuration is required to become an experienced Spring Boot developers.

**3. Spring Initializer**

A web application which can create initial project structure for you. This simplifies initial project setup part.

**4. Spring Actuator**

This feature provides a lot of insights of a running Spring boot application. For example, you can use Actuator to find out which beans are created in Spring's application context and which request path are mapped to controllers.

**5. Spring CLI**

This is another awesome feature of Spring Boot which really takes Spring development into next level. It allows you to use Groovy for writing Spring boot application which means a lot more concise code.

**Key Features**

* Get Spring web-services running with just couple lines of code (literally).
* Embed Tomcat or Undertow directly into them, providing a runnable JAR.
* Automatically configure Spring wherever possible (you can actually run without any configuration out of the box).
* Make your WAR function as an in it service.
* Add standard metrics, health checks, and externalized configuration.
* Integrate with a huge number of things out of the box.

for example:

• Console for discoverability/distributed configuration.

• Admin console for managing spring apps.

Many Spring Boot developers always have their classes annotated with

**@Configuration**, **@EnableAutoConfiguration** and **@ComponentScan**

Spring Boot provides a convenient **@SpringBootApplication** alternative.

* **@Configuration** is used to turn a class into a JavaConfig source so you can define beans in it, etc. The class used in SpringApplication.run() should be a configuration class (though XML configs are possible).
* **@EnableAutoConfiguration** attempts to guess and configure beans that you are likely to need based on our code and class-path. E.g. if Tomcat is present due to web dependencies in the POM, it will set it up and use it for you.
* **@ComponentScan** is used to tell Spring to automatically search for and wire up classes based on their annotations (e.g. make @Controllers register themselves, populate @Value variables, etc.)

**Adding Actuator**

Spring boot actuator provides production grade metrics, auditing, and monitoring features to your application.

Just adding the actuator dependency makes it available on our 8080 port. If we add hateoas as well, it will make it restfully navigable under /actuator. No code is required.

**Maven Dependency Scopes**

Maven is one of the most popular build tools in the Java ecosystem, and one of its core features is dependency management.

## ****Dependency Scopes****

Dependency scopes can help to limit transitivity of the dependencies and they modify class path for different built tasks. **Maven has 6 default dependency scopes**.

### **Compile**

**This is the default scope when no other scope is provided.**

Dependencies with this scope are available on the class path of the project in all build tasks and they're propagated to the dependent projects.

|  |  |  |  |
| --- | --- | --- | --- |
|  | <dependency>      <groupId>commons-lang</groupId>      <artifactId>commons-lang</artifactId>  <version>2.6</version>  </dependency> **Provided** This scope is used to mark **dependencies that should be provided at runtime by JDK or a container**  <dependency>      <groupId>javax.servlet</groupId>      <artifactId>servlet-api</artifactId>      <version>2.5</version>      <scope>provided</scope>  </dependency> **Runtime** **The dependencies with this scope are required at runtime,**  but they're not needed for compilation of the project code. Because of that,  dependencies marked with the runtime scope will be present in runtime and test class path,  but they will be missing from compile class path.  A good example of dependencies that should use the runtime scope is a JDBC driver:   |  |  | | --- | --- | |  | <dependency>      <groupId>mysql</groupId>      <artifactId>mysql-connector-java</artifactId>      <version>6.0.6</version>      <scope>runtime</scope>  </dependency> | |

### **4.Test**

This scope is used to indicate that dependency isn't required at standard runtime of the application, but is used only for test purposes. **Test dependencies aren't transitive and are only present for test and execution class paths.**

The standard use case for this scope is adding test library like JUnit to our application:

|  |  |  |  |
| --- | --- | --- | --- |
|  | <dependency>      <groupId>junit</groupId>      <artifactId>junit</artifactId>      <version>4.12</version>      <scope>test</scope>  </dependency> **5.System** System**scope is much similar to the**provided**scope.**  The main difference between those two scopes is that system requires us to directly point to specific jar on the system.  The important thing to remember is that building the project with system scope dependencies  may fail on different machines if dependencies aren't present or are located in a different place  than the one system Path points to:   |  |  | | --- | --- | |  | <dependency>      <groupId>com.ecosystem</groupId>      <artifactId>custom-dependency</artifactId>      <version>1.3.2</version>      <scope>system</scope>      <systemPath>${project.basedir}/libs/custom-dependency-1.3.2.jar</systemPath>  </dependency>  **6. Import**  This scope was added in Maven 2.0.9 and it’s only available for the dependency type pom  <dependency>      <groupId>com.beans</groupId>      <artifactId>custom-project</artifactId>      <version>1.3.2</version>      <type>pom</type>      <scope>import</scope>  </dependency> | |

**Spring Boot Annotations**

**@EnableAutoConfiguration** – It enable auto-configuration mechanism.

**@ComponentScan** – enable component scanning in application class path.

**@SpringBootApplication** – enable all 3 above three things in one step i.e.

enable auto-configuration mechanism, enable component scanning and

register extra beans in the context.

**@ImportAutoConfiguration** – imports and apply only the specified auto-configuration classes.

We should use this when we don’t want to enable the default auto-configuration.

**@AutoConfigureBefore**, **@AutoConfigureAfter**, **@AutoConfigureOrder** – shall be used if the configuration needs to be applied in a specific order (before of after).

**@Conditional** – annotations such as

**@ConditionalOnBean,** **@ConditionalOnWebApplication** or **@ConditionalOnClass**

allow to register a bean only when the condition meets.

## LocaleResolver

## Internationalization is a process that makes your application adaptable to different languages and regions without engineering changes on the source code. In either words, Internationalization is a readiness of Localization.

We need to determine default Locale of your application. We need to add the LocaleResolver bean in our Spring Boot application.

# **Spring i18n – ResourceBundleMessageSource**

For an application to support internationalization (i18n), it requires the capability of **resolving text messages for different locales**. Spring’s application context is able to resolve text messages for a target locale by their keys. Typically, the messages for one locale should be stored in one separate properties file. The properties file which are maintaining for storing messages that file is called a **resource bundle.**