**Without spring boot the problems are**:

1) We need to hunt for all the **compatible libraries** for the specific Spring version and add them.

2) Most of the times we have to configure **DataSource**, **JdbcTemplate**, **TransactionManager, DispatcherServlet, HandlerMapping, ViewResolver**, etc beans in the same way.

3) We should always deploy in external server.

4) The problem with Spring component-scanning and autowiring is that it’s hard to see how all of the components in an application are wired together.

**With Spring Boot the advantages are**:

1) **Starters** help easy dependency management.

2) **Auto configuration** for most of the commonly used built-in classes such as DataSource, JdbcTemplate, TransactionTemplate, DispatcherServlet, ViewResolver, HandlerMappig, etc using customizable properties. We need to enable auto configuration by adding either @EnableAutoConfiguraiton or @SpringBootApplication.

**3) Embedded Server**

The spring-boot-starter-web automatically pulls spring-boot-starter-tomcat which starts tomcat as an embedded server. So we don’t have to deploy our application on any externally installed tomcat server.

**4) Actuators**

The actuators let us look inside of our bean dependencies, autoconfig details, environment variables, configuration properties, memory usage, garbage collection, web requests, and data source usage.

**Note**: a) Spring boot increases the speed of development because of Starters and autoconfiguration.

b) One of the great outcomes of Spring Boot is that it almost eliminates the need to have traditional XML configurations.

Spring Boot newly added in Spring 4 and offers following main features:

**Spring Boot Starters**

Without Spring Boot, we need to configure all the dependencies required in our pom.xml.

Spring Boot starter **aggregates** common groupings of dependencies into **single starter dependency** that can be added to a project’s Maven or Gradle build.

#**pom.xml**

<project>

<dependencies>

<dependency>

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

<artifactId>**spring-boot-starter-jdbc**</artifactId>

<version>1.5.3.RELEASE</version>

</dependency>

<dependency>

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

<artifactId>**spring-boot-starter-test**</artifactId>

<version>1.5.3.RELEASE</version>

</dependency>

<dependency>

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

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

<version>1.5.3.RELEASE</version>

</dependency>

</dependencies>

</project>

**Note**: The list of starters found in <artifactId>**spring-boot-starters**</artifactId>

<modules>

<module>spring-boot-starter</module>

<module>spring-boot-starter-aop</module>

<module>spring-boot-starter-jdbc</module>

<module>spring-boot-starter-test</module>

<module>spring-boot-starter-data-jpa</module>

<module>spring-boot-starter-data-web</module>

…

</modules>

**The parent element is one of the interesting aspects in the pom.xml file**.

<parent>

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

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

<version>1.5.3.RELEASE</version>

</parent>

**Below is the modified pom file after adding starter parent**:

#**pom.xml**

<project>

<parent>

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

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

<version>1.5.3.RELEASE</version>

</parent>

<dependencies>

**<dependency>**

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

**<artifactId>spring-boot-starter-jdbc</artifactId>**

**</dependency>**

**<dependency>**

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

**<artifactId>spring-boot-starter-test</artifactId>**

**</dependency>**

**<dependency>**

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

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

**</dependency>**

</dependencies>

</project>

**Auto configuration**

Spring Boot uses **convention over configuration** by scanning the dependent libraries available in the class path. For each spring-boot-starter-\* dependency in the POM file, Spring Boot executes a default AutoConfiguration classes

Spring Boot provides ‘**spring-boot-autoconfigure**’ module (**spring-boot-autoconfigure-<version>.jar**) which contains many configuration classes to autoconfigure beans. The above jar file contains **META-INF/spring.factories** file which contains list of autoconfigure classes.

# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\

org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\

org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\

JpaBaseConfiguration#transactionManager,\

JpaBaseConfiguration#jpaVendorAdapter

org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

….

….

Since spring boot provides many autoconfigure classes hence reduces the complexity of configuration.

Spring Boot auto-configuration is a runtime (more accurately, application startup-time) process that considers several factors to decide what Spring configuration should and should not be applied. For example, **Is Spring’s JdbcTemplate available on the classpath**? If so and if there is a DataSource bean, then auto-configure a JdbcTemplate bean.

**Example**:

@Configuration

**@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })**

public class JdbcTemplateAutoConfiguration {

}

The **@EnableAutoConfiguration** enables the magic of auto configuration.

Spring-Boot checks tomcat-jdbc (default), HikariCP, Commons DBCP and Common DBCP2 in this sequence order i.e., Spring Boot checks the availability of the following data source classes and uses the first one that is available in classpath.

1. org.apache.tomcat.jdbc.pool.DataSource

2. com.zaxxer.hikari.HikariDataSource

3. org.apache.commons.dbcp.BasicDataSource

4. org.apache.commons.dbcp2.BasicDataSource

The starter-jdbc automatically pulls tomcat-jdbc-{version}.jar.

<dependency>

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

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

</dependency>

We need to exclude tomcat-jdbc from classpath If we want to use other datasources.

<dependency>

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

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

**<exclusions>**

**<exclusion>**

**<groupId>org.apache.tomcat</groupId>**

**<artifactId>tomcat-jdbc</artifactId>**

**</exclusion>**

**</exclusions>**

</dependency>

We need to add following dependency to use HikariDataSource.

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

</dependency>

We need to add following dependency to use commons-dbcp.

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

</dependency>

We need to add following dependency to use commons-dbcp2.

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

</dependency>

Boot automatically configures above data sources based on corresponding **jar file present in classpath** and connection properties should be configured in application.pro perties file. Various properties can be specified inside our **application.properties/application.yml** file. This section provides a list of common Spring Boot properties and references to the underlying classes that consume them.

# **src/main/resources/application.properties**

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe

spring.datasource.username=system

spring.datasource.password=manager

#**tomcat-connection settings**

#spring.datasource.tomcat.initialSize=20

#spring.datasource.tomcat.max-active=25

# **Hikari settings**

#spring.datasource.hikari.maximum-pool-size=20

# **dbcp settings**

#spring.datasource.dbcp.initial-size=20

#spring.datasource.dbcp.max-active=25

# **dbcp2 settings**

spring.datasource.dbcp2.initial-size=20

spring.datasource.dbcp2.max-total=25

# **src/main/resources/application.yml**

spring:

datasource:

driver-class-name: oracle.jdbc.driver.OracleDriver

url: jdbc:oracle:thin:@localhost:1521:xe

username: system

password: manager

tomcat:

initialSize: 20

max-active: 25

If we want to use other than above datasources then we need to configure explicitly.

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

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

#**connection.properties**

jdbc.driverClass=oracle.jdbc.driver.OracleDriver

jdbc.url=jdbc:oracle:thin:@localhost:1521:xe

jdbc.username=system

jdbc.password=manager

jdbc.initPoolSize=15

jdbc.maxPoolSize=25

**@PropertySource(value={"classpath:connection.properties"})**

public class SpringJdbcConfig {

@Autowired

private Environment env;

@Bean

public DataSource dataSource() {

**ComboPooledDataSource ds = new ComboPooledDataSource()**;

try{

ds.setDriverClass(env.getProperty("jdbc.driverClass"));

ds.setJdbcUrl(env.getProperty("jdbc.url"));

ds.setUser(env.getProperty("jdbc.username"));

ds.setPassword(env.getProperty("jdbc.password"));

ds.setInitialPoolSize(env.getProperty("jdbc.initPoolSize", Integer.class));

ds.setMaxPoolSize(env.getProperty("jdbc.maxPoolSize", Integer.class));

}catch(Exception e){

e.printStackTrace();

}

return ds;

}

}

By default SpringBoot features such as external properties, logging, etc are available in the ApplicationContext only if we use SpringApplication. So, SpringBoot provides @SpringBootTest annotation to configure the ApplicationContext for tests which uses SpringApplication behind the scenes.

@SpringBootApplication

public class Application {

public static void main(String[] args) {

**SpringApplication.run**(**Application.class**, args);

}

}

Application.class is passed as a parameter to tell Spring Boot that this is the primary component

**Note**: As an alternate to application.properties, we can use a .**yaml** file. YAML provides a JSON-like structured configuration compared to the flat properties file.

#application.yaml

Server:

port: 9080

@**SpringBootApplication =@SpringBootConfiguration+@ComponentScan+@EnableAutoConfiguration**

The @**SpringBootApplication** enables **Spring component-scanning** and **Spring Boot auto-configuration**. In fact, @SpringBootApplication combines three other useful annotations:

1. **@SpringBootConfiguration**—This annotation hints that the contained class declares one or more @Bean definitions. It can be used as an alternative to the Spring's standard **@Configuration** annotation. The @Configuration is a specialization of @Component hence candidate for component scanning i.e., needs to give configuration class package name in test class. But @SpringBootConfiguration can be found automatically (for example in tests) hence need not to give configuration class package name in test class.

2. **@ComponentScan**—Enables component-scanning so that the web controller classes and other components we write will be automatically discovered and registered as beans in the Spring application context. This annotation is save as <context:component-scan/> element.

3. **@EnableAutoConfiguration**—This enables the magic of Spring Boot auto-configuration.

**Embedded Container**

The spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don’t have to deploy our application on any externally installed tomcat server.

That’s exactly what Spring Boot’s @**WebIntegrationTest** annotation does. By annotating a test class with @WebIntegrationTest, we declare that we want Spring Boot to not only create an application context for our test, but also to start an embedded servlet container. Once the application is running along with the embedded container, we can issue real HTTP requests against it and make assertions against the results.

<dependency>

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

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

</dependency>

**Spring Boot Actuators**

The problem with Autodiscovery and Autoconfiguration is that it’s difficult to know which beans were configured and how these beans wired together.

The Spring Boot Actuators provide us details such as which beans have been configured, bean dependencies, autoconfig report (which cotains both positive and negative maches), environment variables, health, configuration properties, memory usage, garbage collection, web requests, and data source usage.

The following starter dependency should be added in pom.xml file:

<dependency>

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

<artifactId>**spring-boot-starter-actuator**</artifactId>

</dependency>

Spring Boot Acturaltors provide following details:

1. What beans have been configured in the Spring application context

2. What decisions were made by Spring Boot’s auto-configuration

3. What environment variables, system properties, configuration properties, and command-line arguments are available to our application.

4. A trace of recent HTTP requests handled by our application.

5. Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

|  |  |
| --- | --- |
| The Actuator provides following REST endpoints: **REST End Point** | **Description** |
| /beans | Describes all beans in the application context and their relationship to each other. |
| /autoconfig | Provides an auto-configuration report describing what autoconfiguration conditions passed and failed. |
| /env | Retrieves all environment properties. |
| /health | Reports health metrics for the application, as provided by HealthIndicator implementations. |
| /metrics | |
| … | |

**Spring Boot MVC**

The spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don’t have to deploy our application on any externally installed tomcat server.

Below dependency needed to get web starter:

<dependency>

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

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

</dependency>

Below dependency is needed to process JSP pages:

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>**tomcat-embed-jasper**</artifactId>

</dependency>

Below dependency is needed to process html pages:

<dependency>

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

<artifactId>**spring-boot-starter-thymeleaf**</artifactId>

</dependency>

**CommandLineRunner**

If we need to execute some custom code just before boot application starting up? We can make that happen with a runner i.e., Spring Boot provides CommandLineRunner interface to run specific pieces of code when an application is fully started. When we want to execute some piece of code exactly before the application startup completes, we can use it then.

@SpringBootApplication

public class Hello implements **CommandLineRunner**{

public static void main(String args){

SpringApplication.run(Hello.class, args);

}

@Override

**public void run(String... arg0) throws Exception** {

…

}

}

**Fat Jar**

The below plug-in should be added in pom.xml file to create Fat jar:

<!--Build configuration -->

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

<finalName>test</finalName>

</build>

Dependency for Spring Boot Starter Web

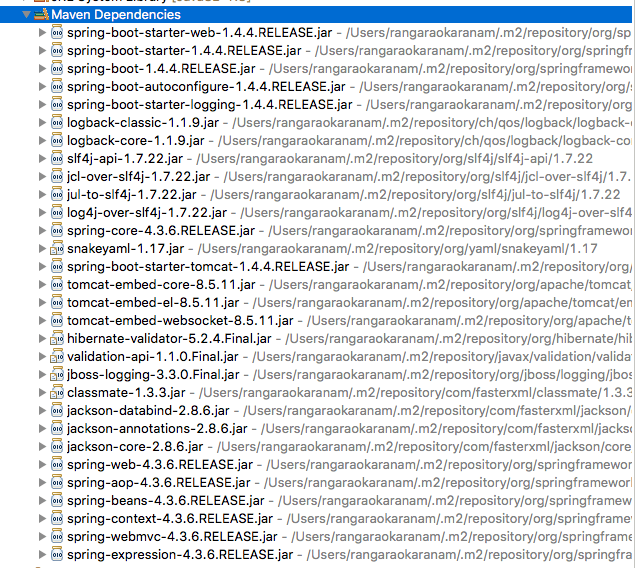
<dependency>

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

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

</dependency>

Following screenshot shows the different dependencies that are added in to our application



Dependencies can be classified into:

* Spring - core, beans, context, aop
* Web MVC - (Spring MVC)
* Jackson - for JSON Binding
* Embedded Servlet Container - Tomcat
* Logging - logback, slf4j

Any typical web application would use all these dependencies. Spring Boot Starter Web comes pre packaged with these. As a developer, I would not need to worry about either these dependencies or their compatible versions.

**Spring Boot Starter Project Options**

As we see from Spring Boot Starter Web, starter projects help us in quickly getting started with developing specific types of applications.

* spring-boot-starter-web-services - SOAP Web Services
* spring-boot-starter-web - Web & RESTful applications
* spring-boot-starter-test - Unit testing and Integration Testing
* spring-boot-starter-jdbc - Traditional JDBC
* spring-boot-starter-security - Authentication and Authorization using Spring Security
* spring-boot-starter-data-jpa - Spring Data JPA with Hibernate
* spring-boot-starter-cache - Enabling Spring Framework’s caching support
* spring-boot-starter-data-rest - Expose Simple REST Services using Spring Data REST