IN28MINUTES

What is Spring Boot Auto Configuration?

This guide will help you understand Spring Boot Auto Configuration with examples. We will use a simple code example creating couple of simple rest services.

You will learn

- Why do we need Auto Configuration?
- What is Auto Configuration?
- A few examples of Spring Boot Auto Configuration
- How is Auto Configuration implemented in Spring Boot?
- How to debug Auto Configuration?

10 Step Reference Courses

- Spring Framework for Beginners in 10 Steps
- Spring Boot for Beginners in 10 Steps
- Spring MVC in 10 Steps
- JPA and Hibernate in 10 Steps
- Eclipse Tutorial for Beginners in 5 Steps
- Maven Tutorial for Beginners in 5 Steps
- JUnit Tutorial for Beginners in 5 Steps
- Mockito Tutorial for Beginners in 5 Steps
- Complete in28Minutes Course Guide

Tools you will need

- Maven 3.0+ is your build tool
- Your favorite IDE. We use Eclipse.
- JDK 1.8+

Complete Maven Project With Code Examples

> *Our Github repository has all the code examples - https://github.com/in28minutes/in28minutes.github.io/tree/master/code-zip-files*

- All other examples related to Restful Web Services
    - Website-springbootrestservices-all-examples.zip

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 component scan, dispatcher servlet, a view resolver, web jars(for delivering static content) among other things.

```
<bean
      class="org.springframework.web.servlet.view.InternalResourceVi
      <property name="prefix">
          <value>/WEB-INF/views/</value>
      </property>
      <property name="suffix">
          <value>.jsp</value>
      </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/"/>
```

Below code snippet shows typical configuration of a dispatcher servlet in a web application.

```
<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>
```

When we use Hibernate/JPA, we would need to configure a datasource, an entity manager factory, a transaction manager among a host of other things.

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSc
    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>
```

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

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

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

Above examples are typical with any Spring framework implementation or integration with other frameworks.

Spring Boot : Can we think different?

Spring Boot brings in 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 Hibernate jar is on the classpath?
- How about auto configuring a Dispatcher Servlet if Spring MVC jar is on the classpath?

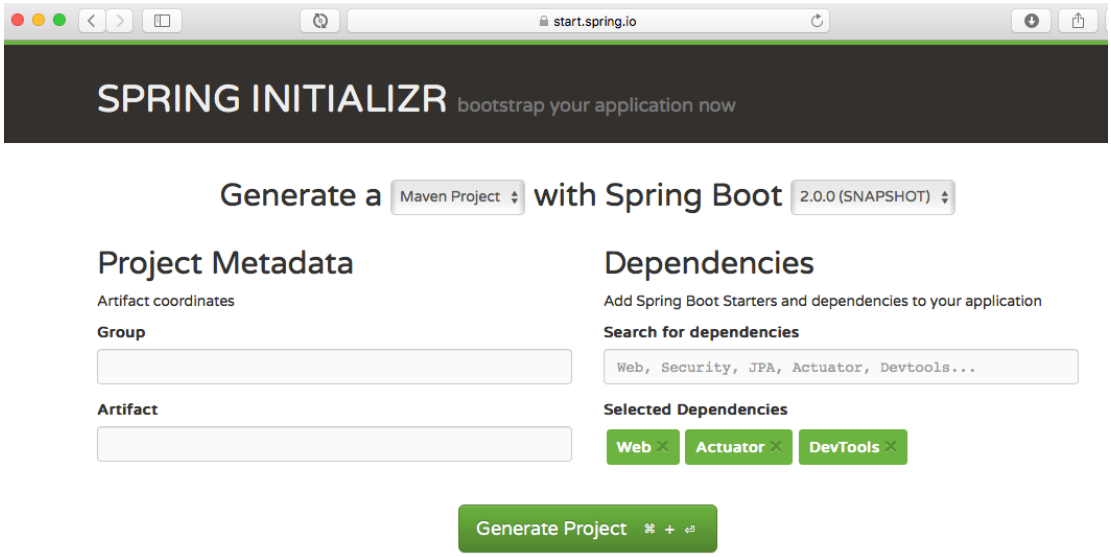
There would be provisions to override the default auto configuration.

> *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`.

To understand Auto Configuration further, lets bootstrap a simple Spring Boot Application using Spring Initializr.

Creating REST Services Application with Spring Initializr

> *Spring Initializr http://start.spring.io/ is great tool to bootstrap your Spring Boot projects.*



As shown in the image above, following steps have to be done.

- Launch Spring Initializr and choose the following
    - Choose `com.in28minutes.springboot` as Group
    - Choose `student-services` as Artifact
    - Choose following dependencies
        - Web
        - Actuator
        - DevTools
- Click Generate Project.
- Import the project into Eclipse.
- If you want to understand all the files that are part of this project, you can go here.

Spring Boot Auto Configuration in action.

When we run StudentServicesApplication.java as a Java Application, you will see a few important things in the log.

```
Mapping servlet: 'dispatcherServlet' to [/]

Mapped "{[/error]}" onto public org.springframework.http.ResponseEntit

Mapped URL path [/webjars/**] onto handler of type [class org.springfr
```

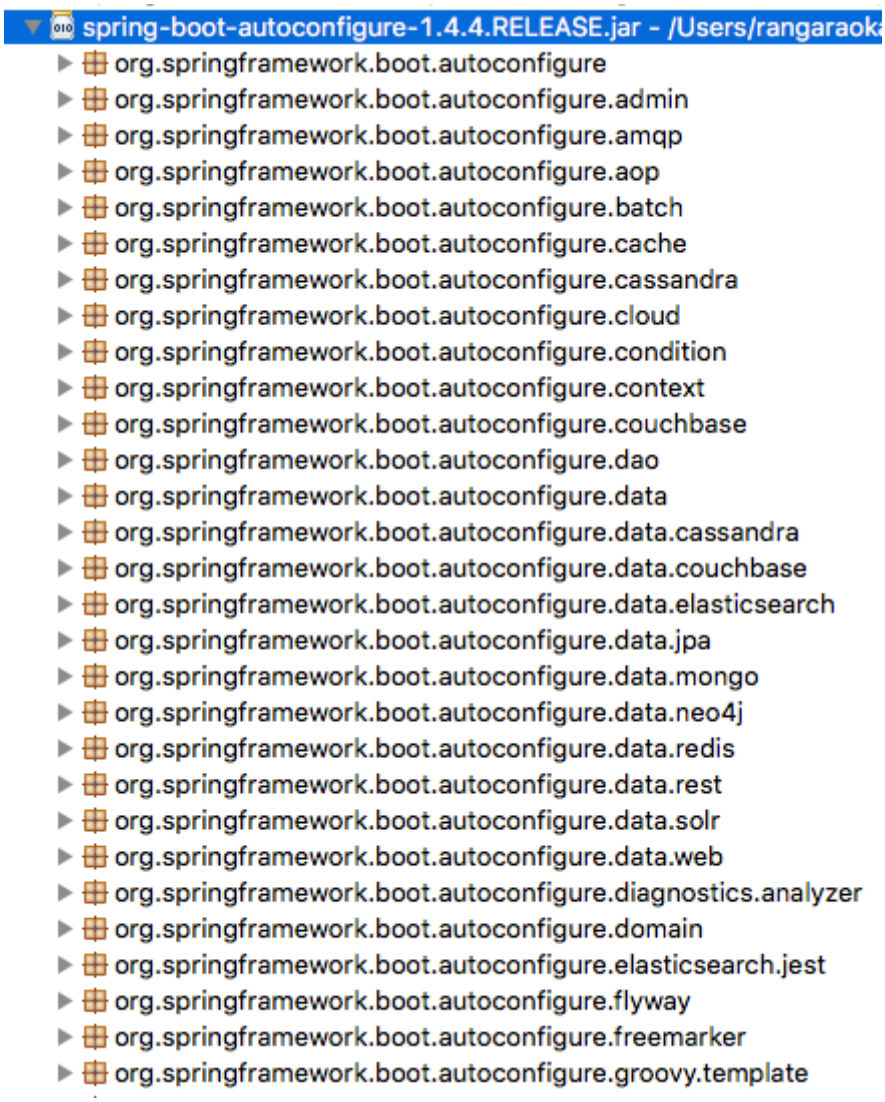
Above log statements are good examples of `Spring Boot Auto Configuration` in action.

As soon as we added in Spring Boot Starter Web as a dependency in our project, Spring Boot Autoconfiguration sees that Spring MVC is on the classpath. It autoconfigures dispatcherServlet, a default error page and webjars.

If you add Spring Boot Data JPA Starter, you will see that Spring Boot Auto Configuration auto configures a datasource and an Entity Manager.

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.



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. A few of the important auto configurations are listed below.

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfigur
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguratic
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguratic
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfigurat
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfigur
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfigurat
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManag
org.springframework.boot.autoconfigure.security.SecurityAutoConfigurat
org.springframework.boot.autoconfigure.security.SecurityFilterAutoConf
org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfig
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAut
org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\
```

Example Auto Configuration

We will take a look at DataSourceAutoConfiguration.

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. Annotations like @ConditionalOnClass, @ConditionalOnMissingBean help in providing these features!

`@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })` : This configuration is enabled only when these classes are available in the classpath.

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguratic
public class DataSourceAutoConfiguration {
```

`@ConditionalOnMissingBean` : This bean is configured only if there is no other bean configured with the same name.

```
@Bean
@ConditionalOnMissingBean
public DataSourceInitializer dataSourceInitializer() {
        return new DataSourceInitializer();
}
```

Embedded Database is configured only if there are no beans of type DataSource.class or XADataSource.class already configured.

```
@Conditional(EmbeddedDatabaseCondition.class)
@ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
@Import(EmbeddedDataSourceConfiguration.class)
protected static class EmbeddedDatabaseConfiguration {
}
```

Debugging Auto Configuration

There are two ways you can debug and find more information about auto configuration.

- Turning on debug logging
- Using Spring Boot Actuator

Debug Logging

You can turn debug logging by adding a simple property value to application.properties. In the example below, we are turning on Debug level for all logging from org.springframework package (and sub packages).

```
logging.level.org.springframework: DEBUG
```

When you restart the application, you would see an auto configuration report printed in the log. Similar to what you see below, a report is produced including all the auto configuration classes. The report separates the positive matches from negative matches. It will show why a specific bean is auto configured and also why something is not auto configured.

```
=========================
AUTO-CONFIGURATION REPORT
=========================

Positive matches:
-----------------
DispatcherServletAutoConfiguration matched
 - @ConditionalOnClass classes found: org.springframework.web.servlet.
 - found web application StandardServletEnvironment (OnWebApplicationC


Negative matches:
-----------------
ActiveMQAutoConfiguration did not match
 - required @ConditionalOnClass classes not found: javax.jms.Connectic

AopAutoConfiguration.CglibAutoProxyConfiguration did not match
 - @ConditionalOnProperty missing required properties spring.aop.proxy
```

Spring Boot Actuator

Other way to debug auto configuration is to add spring boot actuator to your project. We will also add in HAL Browser to make things easy.

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

<dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```

HAL Browser auto configuration http://localhost:8080/actuator/#http://localhost:8080/autoconfig would show the details of all the beans which are auto configured and those which are not.

```
  "negativeMatches": {
    "CacheStatisticsAutoConfiguration": [
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnBean (types: org.springframework.cache.C
acheManager; SearchStrategy: all) found no beans"
      }
    ],

"CacheStatisticsAutoConfiguration.CaffeineCacheStatisticsProviderConfigura
tion": [
      {
        "condition": "OnClassCondition",
        "message": "required @ConditionalOnClass classes not found: com.gi
thub.benmanes.caffeine.cache.Caffeine,org.springframework.cache.caffeine.C
affeineCacheManager"
      },
      {
        "condition": "ConditionEvaluationReport.AncestorsMatchedCondition"
,
        "message": "Ancestor 'org.springframework.boot.actuate.autoconfigu
re.CacheStatisticsAutoConfiguration' did not match"
      }
    ],
```

```
{
  "positiveMatches": {
    "AuditAutoConfiguration#auditListener": [
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnMissingBean (types: org.springframework.
boot.actuate.audit.listener.AbstractAuditListener; SearchStrategy: all) fo
und no beans"
      }
    ],
    "AuditAutoConfiguration#authenticationAuditListener": [
      {
        "condition": "OnClassCondition",
        "message": "@ConditionalOnClass classes found: org.springframework
.security.authentication.event.AbstractAuthenticationEvent"
      },
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnMissingBean (types: org.springframework.
boot.actuate.security.AbstractAuthenticationAuditListener; SearchStrategy:
all) found no beans"
      }
    ],
    "AuditAutoConfiguration#authorizationAuditListener": [
      {
        "condition": "OnClassCondition",
        "message": "@ConditionalOnClass classes found: org.springframework
.security.access.event.AbstractAuthorizationEvent"
      },
```

> *Congratulations! You are reading an article from a series of 50+ articles on Spring Boot and Microservices. We also have 20+ projects on our Github repository. For the complete series of 50+ articles and code examples, click here.*

Next Steps

- Join 100,000 Learners and Become a Spring Boot Expert - 5 Awesome Courses on Microservices, API's, Web Services with Spring and Spring Boot. Start Learning Now
- Learn Basics of Spring Boot - Spring Boot vs Spring vs Spring MVC, Auto Configuration, Spring Boot Starter Projects, Spring Boot Starter Parent, Spring Boot Initializr
- Learn RESTful and SOAP Web Services with Spring Boot
- Learn Microservices with Spring Boot and Spring Cloud
- Watch Spring Framework Interview Guide - 200+ Questions & Answers

> *Join our free Spring Boot in 10 Steps Course.*

> *Find out how in28Minutes reached 100,000 Learners on Udemy in 2 years. The in28minutes Way - Our approach to creating awesome learning experiences.*