

Custom Starter with Spring Boot

Last updated: February 19, 2019 / [Spring Boot](#) / By [Umesh](#) / 2

[COMMENTS](#)



Introduction

[Spring Boot](#) provides several [starters](#) for most of the open source projects. It's possible to develop your own auto-configuration either for your projects or for your organization. We can also create Custom Starter with Spring Boot. Before the start, let's discuss understand how [Spring Boot autoconfiguration](#) works under the hood.

1. Spring Boot Auto Configuration

1.1 Locating Auto Configuration Classes

On starting our application, *Spring Boot* checks for a specific file named as *spring.factories*. This file is located in the *META-INF* directory. Here is an entry from the [spring-boot-autoconfigure](#).

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
```

All auto configuration classes should list under `EnableAutoConfiguration` key in the `spring.factories` property file. Let's pay our attention to few key points in the auto-configuration file entry.

- Based on the configuration file, Spring Boot will try to run all these configurations.
- Actual class configuration load will depend upon the classes on the classpath (e.g. if Spring find JPA in classpath, it will load JPA configuration class)

1.2 Conditional Annotation

Spring Boot use annotations to determine if an autoconfiguration class needs to be configured or not. The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations help Spring Boot to determine if an *auto-configuration* class needs to be included or not. In the similar fashion `@ConditionalOnBean` and `@ConditionalOnMissingBean` are used for spring bean level *autoconfiguration*.

Here is an example of *Mail.SenderAutoConfiguration* class

Here is an example of MailSenderAutoConfiguration class

```
@Configuration
@ConditionalOnClass({ MimeMessage.class, MimeType.class })
@ConditionalOnMissingBean(MailSender.class)
@Conditional(MailSenderCondition.class)
@EnableConfigurationProperties(MailProperties.class)
@Import(JndiSessionConfiguration.class)
public class MailSenderAutoConfiguration {
    // required configurations
}
```

1.3 Conditional Annotation

Spring Boot use default values for the beans initialization. These defaults are based on the *Spring environment properties*. *@EnableConfigurationProperties* is declared with MailProperties class. Here is the code snippet for this class

```
@ConfigurationProperties(prefix = "spring.mail")
public class MailProperties {
    private static final Charset DEFAULT_CHARSET =
        StandardCharsets.UTF_8;

    private Integer port;
}
```

Properties defined in the *MailProperties* file are the default properties for *MailSenderAutoConfiguration* class while initializing beans. Spring Boot allows us to override these configuration properties using *application.properties* file. To override default port, we need to add the following entry in our *application.properties* file.

```
spring.mail.port=445 . (prefix+property name)
```

2. Custom Starter with Spring Boot

To create our own custom starter, we require following components

- The *auto-configure module* with auto configuration class.
- The stater module which will bring all required dependencies using *pom.xml*

For this post, we are creating only a single module combining both *auto-configuration* code and starter module for getting all required dependencies. We will create a simple hello service stater with following features

1. the *hello-service-spring-boot-starter* with *HelloService* which takes the name as input to say hello.
2. *HelloService* will use the default configuration for the default name.
3. We will create Spring Boot demo application for using our *hello-service-starter*.

2.1 The Auto-Configure Module

The *hello-service-spring-boot-starter* will have the following classes and configurations

- *HelloServiceProperties* file for default name.
- *HelloService* interface and *HelloServiceImpl* class.
- *HelloServiceAutoConfiguration* to create *HelloService* Bean.
- The *pom.xml* file for bringing required dependencies to our custom starter.

2.2 Property and Service Class

```
package com.javadevjournal.service;

public interface HelloService {

    void hello();
}

//Impl Service
public class HelloServiceImpl implements HelloService {

    @Override
    public void hello() {
        System.out.println("Hello from the default starter");
    }
}
```

2.3 The AutoConfigure Module and Class

```
@Configuration
@ConditionalOnClass(HelloService.class)
public class HelloServiceAutoConfiguration {

    //conditional bean creation
    @Bean
    @ConditionalOnMissingBean
    public HelloService helloService(){

        return new HelloServiceImpl();
    }
}
```

```
}  
}
```

The final piece of our auto-configuration is the addition of this class in the *spring.factories* property file located in the */src/main/resources/META-INF*.

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.javadevjournal.config.HelloServiceAutoCc
```

On application startup

- *HelloServiceAutoConfiguration* will run if *HelloService* class is available in the classpath. (*@ConditionOnClass* annotation).
- *HelloService* Bean will be created by *Spring Boot* if it is not available (*@ConditionalOnMissingBean*).
- If developer defines their own *HelloService* bean, our *customer starter* will not create *HelloService* Bean.

2.4 The pom.xml

The last part of the custom starter is the *pom.xml* to bring in all the required dependencies.

```
<?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>  
  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starters</artifactId>  
    <version>1.5.9.RELEASE</version>  
  </parent>  
  
  <groupId>com.javadevjournal</groupId>  
  <artifactId>hello-service-spring-boot-starter</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
  <packaging>jar</packaging>  
  
  <name>hello-service-spring-boot-starter</name>  
  <description>Custom Starter for Spring Boot</description>  
  
  <properties>  
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
    <java.version>1.8</java.version>  
  </properties>  
  
  <dependencies>  
    <dependency>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-autoconfigure</artifactId>
```

```
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Let's cover some interesting points in the pom.xml

- We defined the parent as spring-boot-starters. We needed to pull in required dependencies.

For more information on parent pom. Please read our article [Spring Boot Starter Parent](#)

2.4 Naming Convention

While creating a custom starter with Spring Boot, read below guidelines for the naming convention.

- Your *custom starter module* should not start with Spring Boot.
- Use name-spring-boot-starter as a guideline. In our case, we named our starter as hello-service-spring-boot-starter.

3. Using the custom starter

Let's create a [sample Spring Boot application](#) to use our custom starter. Once We create starter app, add the *custom starter* as a dependency in *pom.xml*.

```
<dependency>
<groupId>com.javadevjournals</groupId>
<artifactId>hello-service-spring-boot-starter</artifactId>
<version>0.0.1-SNAPSHOT</version>
</dependency>
```

Here is our Spring Boot starter class

```
@SpringBootApplication
public class CustomStarterAppApplication implements CommandLineRunner {
```

```

@Autowired
HelloService service;

public static void main(String[] args) {

SpringApplication.run(CustomStarterAppApplication.class, args);
}

@Override
public void run(String... strings) throws Exception {
    service.hello();
}

```

If we run our application, you will see following output in the console

```

018-01-23 20:27:52.138 INFO 20441 --- [      main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.
Hello from the default starter
2018-01-23 20:27:52.620 INFO 20441 --- [      main] c.j.CustomStarterAppApplication      : Started CustomS

```

We have defined no *HelloService* in our demo application. When *Spring Boot* started, auto-configuration did not find any *custom bean definition*. Our *custom starter* auto configuration class created default “*HelloService*” bean. (as visible from the output). To understand *Spring Boot auto-configuration* logic and functionality, let’s create a custom *HelloService* bean in our sample application

```

public class CustomHelloService implements HelloService {

    @Override
    public void hello() {
        System.out.println("We are overriding our custom Hello Service");
    }
}

//bean bean definition
@SpringBootApplication
public class CustomStarterAppApplication implements CommandLineRunner {

    @Autowired
    HelloService service;

    public static void main(String[] args) {

SpringApplication.run(CustomStarterAppApplication.class, args);
}


    @Override
    public void run(String... strings) throws Exception {
        service.hello();
    }
}

```

```
@Bean
public HelloService helloService(){
    return new CustomHelloService();
}
}
```

Here is the output on running this application

```
2018-01-23 20:36:48.991 INFO 20529 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering bean
We are overriding our custom Hello Service
2018-01-23 20:36:49.000 INFO 20529 --- [main] c.j.CustomStarterAppApplication : Started CustomS
```



When we defined our custom bean, Spring Boot default *HelloService* is no longer available. This enables developers to completely override default bean definition by creating/ providing their own bean definition.

4. Video Tutorial

Watch our video tutorial on *Custom Starter with Spring Boot*.

Summary

In this post, we create *Custom Starter with Spring Boot*. We learned how to use these *custom starters* in our application. We covered *how Spring Boot auto-configuration works* with starters. This is a simple example of *creating a custom starter*. In our next post, we will cover *creating custom configurator using Spring Boot* with following features.

- Custom auto-configuration module.
- Custom starter.
- Integrating these two modules to work together.

Umesh

Hello!! I am Umesh- an engineer by profession and a photographer by passion.I like to build stuff on the web using OSS and love to capture the world through my lens.



follow me on:

Further Reading

12 Mar, 2019

[Spring Boot With Ehcache](#)

08 Mar, 2019

[Custom Validation MessageSource in Spring Boot](#)

05 Mar, 2019

[Spring Boot Rest Example](#)

01 Mar, 2019

[Spring Boot Example](#)

26 Feb, 2019

[Dockerizing Spring Boot Application](#)

22 Feb, 2019

[How Spring Boot auto-configuration works](#)

2

Leave a Reply

Join the discussion...

1 Comment threads

1 Thread replies

0 Followers

Most reacted comment

Hottest comment thread



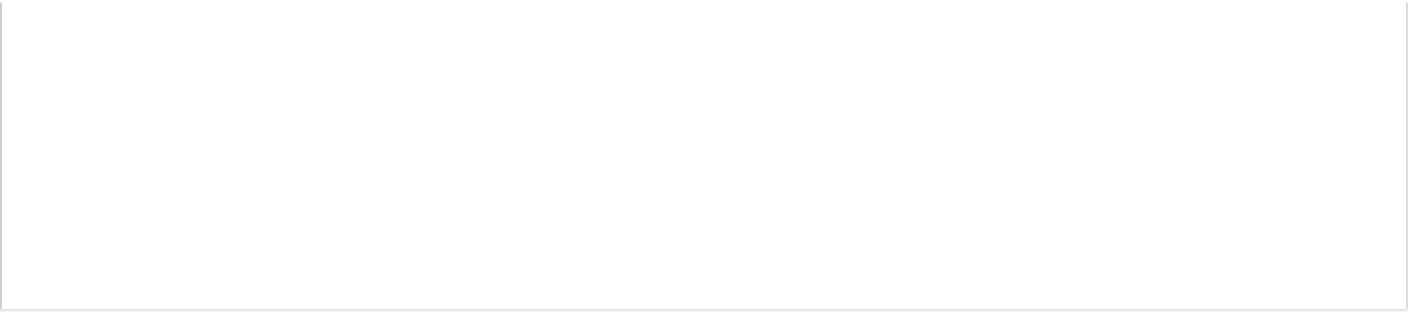
Recent comment authors

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

Subscribe

newest **oldest** most voted





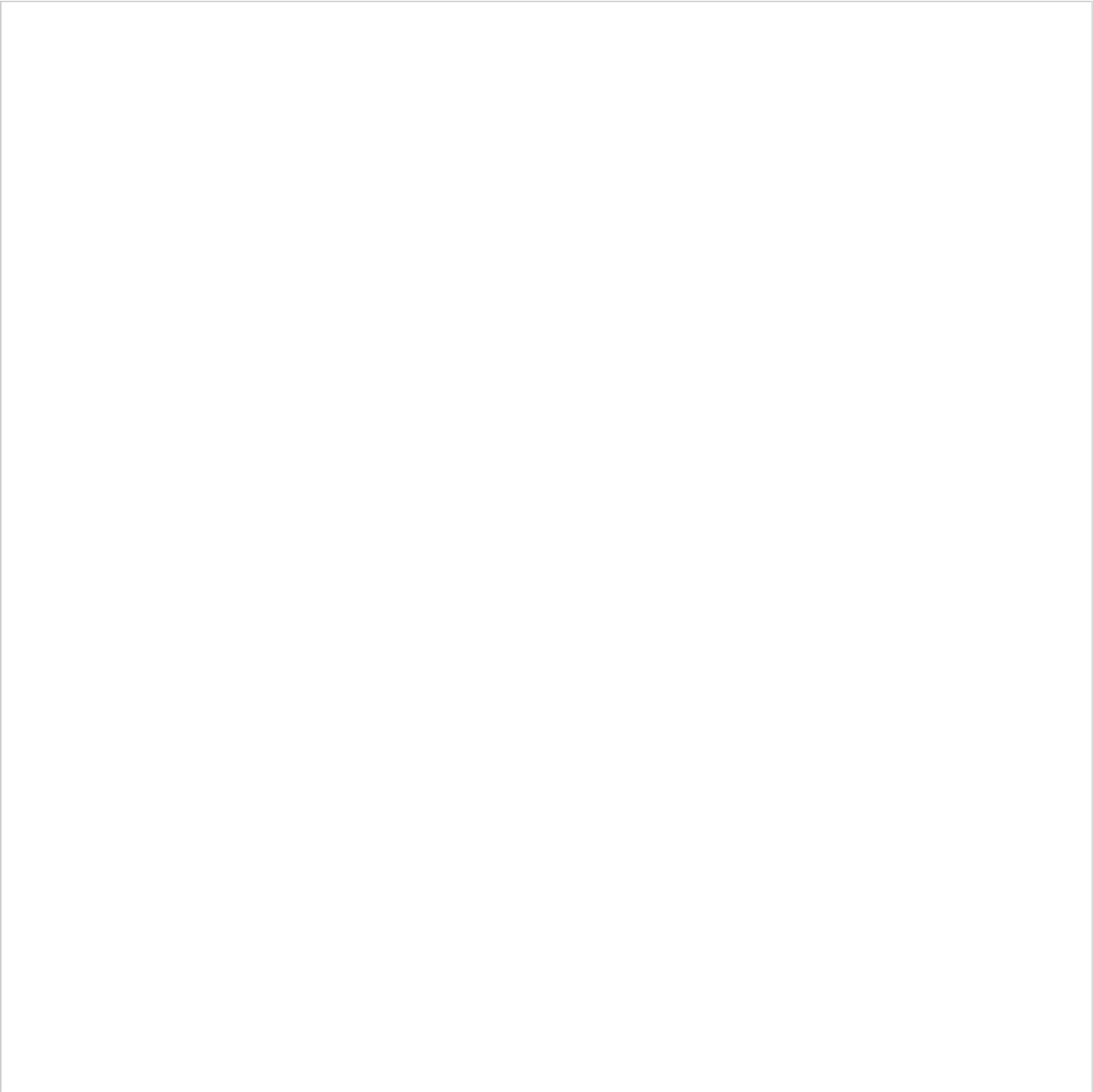
Guest

Fathzer

Thanks a lot for this easy to understand working example ☺

Vote Up0Vote Down [Reply](#)

6 months ago



Admin

Umesh Awasthi

I am happy that this post was helpful to you ☺

Vote Up0Vote Down [Reply](#)

6 months ago

 wpDiscuz

SERIES

- [BUILD REST API WITH SPRING](#)
- [JAVA AND SPRING INTERVIEW QUESTIONS](#)
- [SPRING BOOT TUTORIALS](#)
- [SPRING MVC TUTORIAL](#)

CATEGORIES

- [JAVA](#)
- [NEWSLETTER](#)
- [REST](#)
- [SHOPIZER](#)
- [SHOPIZER SETUP](#)
- [SPRING](#)
- [SPRING BOOT](#)
- [SPRING MVC](#)

ABOUT

- [ABOUT US](#)
- [BUILD REST API WITH SPRING](#)
- [DISCLAIMER](#)
- [JAVA AND SPRING INTERVIEW QUESTIONS](#)
- [PRIVACY POLICY](#)
- [SPRING BOOT TUTORIALS](#)
- [SPRING MVC TUTORIAL](#)
- [WRITE AND EARN](#)

Java Development Journal

Copyright 2018 by Java
Development Journal.

Connect With Me:

-
-
-