# Tools/Software required

* **jdk1.8.0\_191**
* **apache-maven-3.6.0**
* Eclipse **Eclipse Versions:**2018-09 (4.9)
  + To enable the Spring Tool Suite-STS: download the plugin STS Spring Tools 4

<http://marketplace.eclipse.org/marketplace-client-intro?mpc_install=1794107>

* To enable the Angular plugin for Angular Project in Eclipse, download the same using the following link:

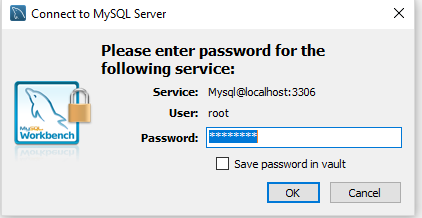
<http://marketplace.eclipse.org/marketplace-client-intro?mpc_install=3157661>

* Download Angular 7 Console (CLI) with Node JS
* DB Design Tool : <https://dbdiagram.io/d/5c8cc2e2f7c5bb70c72f4fbf>
  + User Id/Password: gmail credentials
* My SQL Server 8.0.13

User Id: root

Password: rajiv999

User Schema bjjd and Password: bjjd



# Technologies (implemented following technologies in order)

1. Spring Boot
2. Swagger: to create server stub for Spring Rest API using Swagger definition
3. Spring Rest
4. Spring Microservices
5. My SQL
6. Angular 7 with Node JS (with npm)

* npm

npm makes it easy for JavaScript developers to share and reuse code, and makes it easy to update the code that you’re sharing, so you can build amazing things.

* Install npm

npm is installed with Node.js

npm is distributed with [Node.js](https://nodejs.org/)- which means that when you download Node.js, you automatically get npm installed on your computer.

## **Architecture**angular-http-service-architecture spring-boot-angular-6-spring-rest-api-data-mysql-spring-server-architecturespring-boot-angular-6-spring-rest-api-data-mysql-angular-client-architecture

## **Spring Boot**

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

### Features

* Create stand-alone Spring applications
* Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
* Provide opinionated 'starter' dependencies to simplify your build configuration
* Automatically configure Spring and 3rd party libraries whenever possible
* Provide production-ready features such as metrics, health checks and externalized configuration
* Absolutely no code generation and no requirement for XML configuration

### Benefits

* **Automatic configuration**

Spring Boot can automatically provide configuration for application functionality common to many Spring applications.

* 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.
* We also added **spring-boot-starter-Thymeleaf**which not only adds the Thymeleaf library dependencies but also configures **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 classpath then SpringBoot 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.

* **Easy dependency Management- Starter dependencies**

You tell Spring Boot what kind of functionality you need, and it will ensure that the libraries needed are added to the build.

1. First thing to observe is we are using some dependencies named like **spring-boot-starter-\***. Remember I said “95% of the times I use the same configuration. So when you add **springboot-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**.
2. We have added 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.

* **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/**](http://localhost:8080/).
* **Where is the servlet container comes from?**

We have added **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-bootstarter-tomcat** from **spring-boot-starter-web** and include **spring-boot-starter-jetty**.

### Why Spring Boot

#### Without Spring Boot -Application Development Spring MVC and JPA (Hibernate) (Too much work)

If you are a Java developer then there is a high chance that you might have heard about Spring framework and probably have used it in your projects. Spring framework was created primarily as a Dependency Injection container but it is much more than that.

Spring is very popular for several reasons:

* Spring’s dependency injection approach encourages writing testable code
* Easy to use but powerful database transaction management capabilities
* Spring simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts/JSF/etc. web frameworks
* State of the art Web MVC framework for building web applications

Along with the Spring framework, there are many other Spring sister projects that help to build applications addressing modern business needs:

* Spring Data: Simplifies data access from relational and NoSQL data stores.
* Spring Batch: Provides powerful batch processing framework.
* Spring Security: Robust security framework to secure applications.
* Spring Social: Supports integration with social networking sites like Facebook, Twitter, LinkedIn, GitHub, etc.
* Spring Integration: An implementation of Enterprise Integration Patterns to facilitate integration with other enterprise applications using lightweight messaging and declarative adapters.

There are many other interesting projects addressing various other modern application development needs. For more information, take a look at <http://spring.io/projects>.

In the initial days, the Spring framework provided an XML-based approach for configuring beans. Later Spring introduced XML-based DSLs, Annotations, and JavaConfig-based approaches for configuring beans.

Let us take a quick look at how each of those configuration styles looks.

##### **XML-Based Configuration**

<bean id="userService" class="com.sivalabs.myapp.service.UserService">

<property name="userDao" ref="userDao"/>

</bean>

<bean id="userDao" class="com.sivalabs.myapp.dao.JdbcUserDao">

<property name="dataSource" ref="dataSource"/>

</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/test"/>

<property name="username" value="root"/>

<property name="password" value="secret"/>

</bean>

##### **Annotation-Based Configuration**

@Service

public class UserService

{

private UserDao userDao;

@Autowired

public UserService(UserDao dao){

this.userDao = dao;

}

...

...

}

@Repository

public class JdbcUserDao

{

private DataSource dataSource;

@Autowired

public JdbcUserDao(DataSource dataSource){

this.dataSource = dataSource;

}

...

...

}

##### **JavaConfig-Based Configuration**

@Configuration

public class AppConfig

{

@Bean

public UserService userService(UserDao dao){

return new UserService(dao);

}

@Bean

public UserDao userDao(DataSource dataSource){

return new JdbcUserDao(dataSource);

}

@Bean

public DataSource dataSource(){

BasicDataSource dataSource = new BasicDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/test");

dataSource.setUsername("root");

dataSource.setPassword("secret");

return dataSource;

}

}

Wow… Spring provides many approaches for doing the same thing, and we can even mix the approaches and use both JavaConfig- and Annotation-based configuration styles in the same application.

That is a lot of flexibility and it is both good and bad. People new to the Spring framework may get confused about which approach to follow. As of now, the Spring team suggests following a JavaConfig-based approach as it gives more flexibility.

But there is no one-size fits all kind of solution. One has to choose the approach based on their own application needs.

OK, now that you had a glimpse of how various styles of Spring bean configurations look.

Let us take a quick look at what the configuration of a typical SpringMVC + JPA/Hibernate web application configuration looks like.

##### A Web Application Using Spring MVC and JPA (Hibernate)

Before getting to know what Spring Boot is and what kind of features it provides, let us take a look at how a typical Spring web application configuration looks, the pain points, and how Spring Boot addresses those problems.

###### **Step 1: Configure Maven Dependencies**

The first thing we need to do is configure all the dependencies required in our **pom.xml**.

<?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/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.sivalabs</groupId>

<artifactId>springmvc-jpa-demo</artifactId>

<packaging>war</packaging>

<version>1.0-SNAPSHOT</version>

<name>springmvc-jpa-demo</name>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<failOnMissingWebXml>false</failOnMissingWebXml>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>1.9.2.RELEASE</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>1.7.13</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.13</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.13</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.190</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.38</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>4.3.11.Final</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.thymeleaf</groupId>

<artifactId>thymeleaf-spring4</artifactId>

<version>2.1.4.RELEASE</version>

</dependency>

</dependencies>

</project>

We have configured all our Maven jar dependencies to include Spring MVC, Spring Data JPA, JPA/Hibernate, Thymeleaf, and Log4j.

###### **Step 2: Configure Service/DAO Layer Beans Using JavaConfig**

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(basePackages="com.sivalabs.demo")

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

public class AppConfig

{

@Autowired

private Environment env;

@Bean

public static PropertySourcesPlaceholderConfigurer placeHolderConfigurer()

{

return new PropertySourcesPlaceholderConfigurer();

}

@Value("${init-db:false}")

private String initDatabase;

@Bean

public PlatformTransactionManager transactionManager()

{

EntityManagerFactory factory = entityManagerFactory().getObject();

return new JpaTransactionManager(factory);

}

@Bean

public LocalContainerEntityManagerFactoryBean entityManagerFactory()

{

LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();

HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

vendorAdapter.setGenerateDdl(Boolean.TRUE);

vendorAdapter.setShowSql(Boolean.TRUE);

factory.setDataSource(dataSource());

factory.setJpaVendorAdapter(vendorAdapter);

factory.setPackagesToScan("com.sivalabs.demo");

Properties jpaProperties = new Properties();

jpaProperties.put("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));

factory.setJpaProperties(jpaProperties);

factory.afterPropertiesSet();

factory.setLoadTimeWeaver(new InstrumentationLoadTimeWeaver());

return factory;

}

@Bean

public HibernateExceptionTranslator hibernateExceptionTranslator()

{

return new HibernateExceptionTranslator();

}

@Bean

public DataSource dataSource()

{

BasicDataSource dataSource = new BasicDataSource();

dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));

dataSource.setUrl(env.getProperty("jdbc.url"));

dataSource.setUsername(env.getProperty("jdbc.username"));

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

return dataSource;

}

@Bean

public DataSourceInitializer dataSourceInitializer(DataSource dataSource)

{

DataSourceInitializer dataSourceInitializer = new DataSourceInitializer();

dataSourceInitializer.setDataSource(dataSource);

ResourceDatabasePopulator databasePopulator = new ResourceDatabasePopulator();

databasePopulator.addScript(new ClassPathResource("data.sql"));

dataSourceInitializer.setDatabasePopulator(databasePopulator);

dataSourceInitializer.setEnabled(Boolean.parseBoolean(initDatabase));

return dataSourceInitializer;

}

}

In our AppConfig.java configuration class we have done the following:

* Marked it as a Spring Configuration class using **@Configuration** annotation.
* Enabled Annotation based transaction management using **@EnableTransactionManagement**
* Configured **@EnableJpaRepositories** to indicate where to look for Spring Data JPA repositories
* Configured PropertyPlaceHolder bean using **@PropertySource** annotation and **PropertySourcesPlaceholderConfigurer**bean definition, which loads properties from **application.properties** file.
* Defined beans for **DataSource**, JPA **EntityManagerFactory**, **JpaTransactionManager**.
* Configured **DataSourceInitializer**bean to initialize the database by executing **data.sql** script on application start up.

We need to configure property placeholder values in **application.properties** as follows:

jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/test

jdbc.username=root

jdbc.password=admin

init-db=true

hibernate.dialect=org.hibernate.dialect.MySQLDialect

hibernate.show\_sql=true

hibernate.hbm2ddl.auto=update

We can create a simple SQL script **data.sql** to populate sample data into **USER**table:

delete from user;

insert into user(id, name) values(1,'Siva');

insert into user(id, name) values(2,'Prasad');

insert into user(id, name) values(3,'Reddy');

We can create **log4j.properties** file with basic configuration as follows:

log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p %t %c{2}:%L - %m%n

log4j.category.org.springframework=INFO

log4j.category.com.sivalabs=DEBUG

###### **Step 3: Configure Spring MVC Web Layer Beans**

We will have to configure Thymeleaf **ViewResolver**, static **ResourceHandlers**, **MessageSource**for i18n etc.

@Configuration

@ComponentScan(basePackages = { "com.sivalabs.demo"})

@EnableWebMvc

public class WebMvcConfig extends WebMvcConfigurerAdapter

{

@Bean

public TemplateResolver templateResolver() {

TemplateResolver templateResolver = new ServletContextTemplateResolver();

templateResolver.setPrefix("/WEB-INF/views/");

templateResolver.setSuffix(".html");

templateResolver.setTemplateMode("HTML5");

templateResolver.setCacheable(false);

return templateResolver;

}

@Bean

public SpringTemplateEngine templateEngine() {

SpringTemplateEngine templateEngine = new SpringTemplateEngine();

templateEngine.setTemplateResolver(templateResolver());

return templateEngine;

}

@Bean

public ThymeleafViewResolver viewResolver() {

ThymeleafViewResolver thymeleafViewResolver = new ThymeleafViewResolver();

thymeleafViewResolver.setTemplateEngine(templateEngine());

thymeleafViewResolver.setCharacterEncoding("UTF-8");

return thymeleafViewResolver;

}

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry)

{

registry.addResourceHandler("/resources/\*\*").addResourceLocations("/resources/");

}

@Override

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer)

{

configurer.enable();

}

@Bean(name = "messageSource")

public MessageSource configureMessageSource()

{

ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();

messageSource.setBasename("classpath:messages");

messageSource.setCacheSeconds(5);

messageSource.setDefaultEncoding("UTF-8");

return messageSource;

}

}

In our **WebMvcConfig.java** configuration class we have done the following:

* Marked it as a Spring Configuration class using **@Configuration** annotation.
* Enabled Annotation based Spring MVC configuration using **@EnableWebMvc**
* Configured Thymeleaf ViewResolver by registering **TemplateResolver**, **SpringTemplateEngine**, **ThymeleafViewResolver**beans.
* Registered ResourceHandlers bean to indicate requests for static resources with URI**/resources/\*\***will be served from the location**/resources/** directory.
* Configured **MessageSource**bean to load i18n messages from ResourceBundle **messages-{country-code}.properties** from classpath.

For now we do not have any messages to be configured, so create an empty **messages.properties** file in **src/main/resources** folder.

###### **Step 4: Register Spring MVC FrontController Servlet DispatcherServlet**

Prior to Servlet 3.x specification we have to register Servlets/Filters in **web.xml**. Since Servlet 3.x specification we can register Servlets/Filters programatically using **ServletContainerInitializer**.

Spring MVC provides a convenient class **AbstractAnnotationConfigDispatcherServletInitializer**to register **DispatcherServlet**.

public class SpringWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer

{

@Override

protected Class<?>[] getRootConfigClasses()

{

return new Class<?>[] { AppConfig.class};

}

@Override

protected Class<?>[] getServletConfigClasses()

{

return new Class<?>[] { WebMvcConfig.class };

}

@Override

protected String[] getServletMappings()

{

return new String[] { "/" };

}

@Override

protected Filter[] getServletFilters() {

return new Filter[]{ new OpenEntityManagerInViewFilter() };

}

}

In our **SpringWebAppInitializer.java**configuration class we have done the following:

* We have configured **AppConfig.class** as **RootConfirationClasses**which will become the parent **ApplicationContext**that contains bean definitions shared by all child (**DispatcherServlet**) contexts.
* We have configured **WebMvcConfig.class** as **ServletConfigClasses**which is child**ApplicationContext**that contains WebMvc bean definitions.
* We have configured **”/”** as **ServletMapping**means all the requests will be handled by**DispatcherServlet**.
* We have registered **OpenEntityManagerInViewFilter**as a Servlet Filter so that we can lazy load the JPA Entity lazy collections while rendering the view.

###### **Step 5: Create a JPA Entity and Spring Data JPA Repository**

Create a JPA entity **User.java** and a Spring Data JPA repository for User entity.

@Entity

public class User

{

@Id @GeneratedValue(strategy=GenerationType.AUTO)

private Integer id;

private String name;

//setters and getters

}

public interface UserRepository extends JpaRepository<User, Integer>

{

}

###### **Step 6: Create a SpringMVC Controller**

Create a SpringMVC controller to handle URL**“/”**and render a list of users.

@Controller

public class HomeController

{

@Autowired UserRepository userRepo;

@RequestMapping("/")

public String home(Model model)

{

model.addAttribute("users", userRepo.findAll());

return "index";

}

}

###### **Step 7: Create a Thymeleaf View /WEB-INF/views/index.html to Render List of Users**

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="utf-8"/>

<title>Home</title>

</head>

<body>

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

</tr>

</thead>

<tbody>

<tr th:each="user : ${users}">

<td th:text="${user.id}">Id</td>

<td th:text="${user.name}">Name</td>

</tr>

</tbody>

</table>

</body>

</html>

We are all set now to run the application. But before that we need to download and configure the server like **Tomcat**or **Jetty**or **Wildfly**etc in your IDE.

You can download Tomcat 8 and configure in your favorite IDE, run the application and point your browser to **http://localhost:8080/springmvcjpa-demo**. You should see the list of users details in a table.

Yay…We did it.

##### **Problem with this approach**

**But wait..Isn’t it too much work to just show a list of user details pulled from a database table?**

**Let us be honest and fair. All this configuration is not just for this one use-case. This configuration is the basis for rest of the application also.**

**But again, this is too much of work to do if you want to quickly get up and running.**

**Another problem with it is, assume you want to develop another SpringMVC application with a similar technical stack?**

**Well, you copy-paste the configuration and tweak it. Right? But remember one thing: if you have to do the same thing again and again, you should find an automated way to do it.**

Apart from writing the same configuration again and again, do you see any other problems here?

**Well, let me list what are the problems I am seeing here.**

* You need to hunt for all the **compatible libraries** for the specific Spring version and configure them.
* 95% of the times we configure the **DataSource**, **EntitymanagerFactory**,**TransactionManager**etc beans in the same way. Wouldn’t it be great if Spring can do it for me automatically.
* Similarly we configure SpringMVC beans like **ViewResolver**, **MessageSource**etc in the same way most of the times.

#### With Spring Boot- Application Development Spring MVC and JPA (Hibernate) (less work)

**If Spring can automatically do it for me that would be awesome!!!.**

Imagine, what if Spring is capable of configuring beans automatically? What if you can customize the automatic configuration using simple customizable properties?

For example, instead of mapping DispatcherServlet url-pattern to “/” you want to map it to “/app/”. Instead of putting Thymeleaf views in “/WEB-INF/views” folder you may want to place them in “/WEB-INF/templates/” folder.

So basically you want Spring to do things automatically but provide the flexibility to override the default configuration in a simpler way?

Well, you are about to enter into the world of SpringBoot where your dreams come true!!!

##### A Quick Taste of Spring Boot

Welcome to Spring Boot! Spring Boot does what exactly you are looking for. It will do things automatically for you but allows you to override the defaults if you want to.

Instead of explaining in theory I prefer to explain by example.

So let us implement the same application that we built earlier but this time using SpringBoot.

##### **Step 1: Create a Maven-Based Spring Boot Project**

Create a Maven project and configure the dependencies as follows:

<?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/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.sivalabs</groupId>

<artifactId>hello-springboot</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>hello-springboot</name>

<parent>

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

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

<version>1.3.2.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

</dependencies>

</project>

Wow our **pom.xml** suddenly become so small!

##### **Step 2: Configure Datasource/JPA Properties in application.properties as Follows**

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=admin

spring.datasource.initialize=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

You can copy the same **data.sql** file into **src/main/resources** folder.

##### **Step 3: Create a JPA Entity and Spring Data JPA Repository Interface for the Entity**

Create **User.java, UserRepository.java** and **HomeController.java** same as in **springmvc-jpa-demo**application.

##### **Step 4: Create Thymeleaf View to Show List of Users**

Copy **/WEB-INF/views/index.html** that we created in **springmvc-jpa-demo** application into**src/-main/resources/templates** folder in our new project.

##### **Step 5: Create SpringBoot EntryPoint Class**

Create a Java class **Application.java** with main method as follows:

@SpringBootApplication

public class Application

{

public static void main(String[] args)

{

SpringApplication.run(Application.class, args);

}

}

Now run **Application.java**as a Java Application and point your browser to **http://localhost:8080/**.

You should see the list of users in table format. Coool!!!

Ok ok, I hear you are shouting “What is going on???”.

Let me explain what just happened.

* **Easy dependency Management**
* **Auto Configuration**
* **Embedded Servlet Container Support**

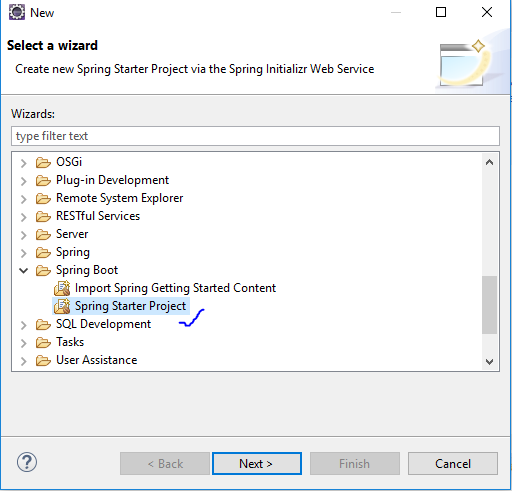
### Project Work from the scratch

In this article, I am going to explain how to develop a RESTful Spring Boot project, step-by-step.

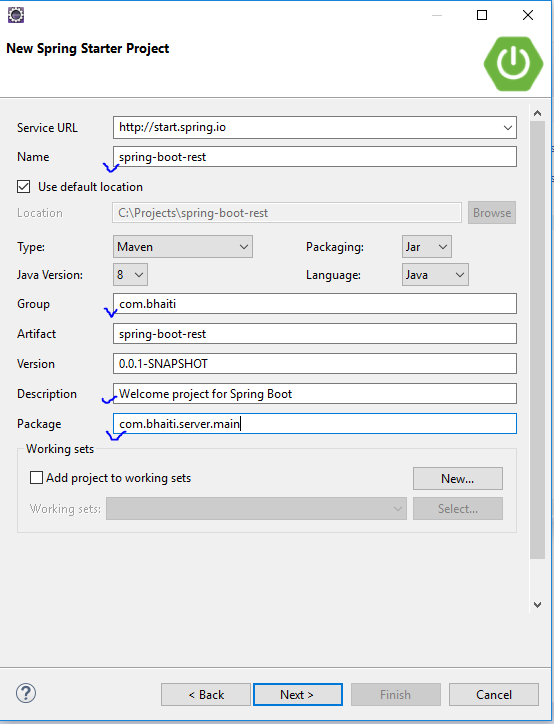
The first example I am going to explain is HTTP GET requests and the second example will be about HTTP POST requests and, in both, the example message format will be in [JSON representation](https://spring.io/understanding/JSON). You can download the project from https://github.com/prateekparallel/spring-boot-rest

Let’s start:

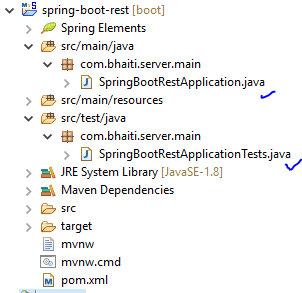
1. First, from the File menu select "New"  and then "other" and from wizard expand "Spring Boot" and select ‘Spring Starter Project’ (File->New->other->wizard->Spring Starter Project)



Now select next and provide the below information and click the Finish button.



Now you can see the below file structure in your project's explorer window.



Now, look that the **SpringBootRestApplication.java** file which is created by the STS plug-ins.

package com.bhaiti.server.main;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringBootRestApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootRestApplication.class, args);

}

}

This is a Spring Boot main class. A Spring Boot REST application loads through this class. We can also see that this class is created with the annotation @SpringBootApplication . As per the Spring documentation, the annotation @SpringBootApplication is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan, and these annotations are frequently used together. Most of the time in Spring Boot development, the main class is always annotated with all three of these important annotations.

So we are going to modify the @SpringBootApplication (given below in the Java class) with a component path. Without that, the application cannot find out the controller classes. We will learn more about controller classes in a few minutes.

@SpringBootApplication(scanBasePackages = {"com.bhaiti"})

publicclass SpringBootRestApplication {

publicstaticvoid main(String[] args) {

SpringApplication.run(SpringBootRestApplication.class, args);

}

}

1. To model the welcome representation, we are going to create a resource representation class here.

This project will be continue in next section………………….

## **Swagger**

### **Introduction**

Swagger is an open source software to build standard documentation in a human readable format for [REST APIs](https://restfulapi.net/). This provides a UI to easily understand service contract and consumer can interact with service without any detailed knowledge of underlying logic.

Swagger is developed by SmartBear software and comes with tools like

* **Swagger Editor**
* **Swagger CodeGen**
* **Swagger UI**
* **Swagger Inspector**.

Swagger has provided it’s specification known as OpenAPI specification to follow while documenting REST API.

Swagger can be integrated with REST APIs in below ways:

1. A **top-down** approach (Design First) – First API specification and then code generation
2. A **bottom-up** approach (Code First) – First API code and then Swagger integration. This is quite familiar and mostly useful when there is already an existing REST APIs built in and Swagger documentation needs to be integrated.

### **Top Down Approach (Design First)**

While consuming SOAP services we usually get the WSDL contract and generate fully annotated JAVA code. Like so if we are able to specify REST contract following [OpenAPI specification](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md) then we could use [Swagger CodeGen](https://swagger.io/swagger-codegen/) to create server stubs and client SDKs.

We will see how this can be done to have basic implementation of intended REST service.

Following are the steps in Top Down Approach

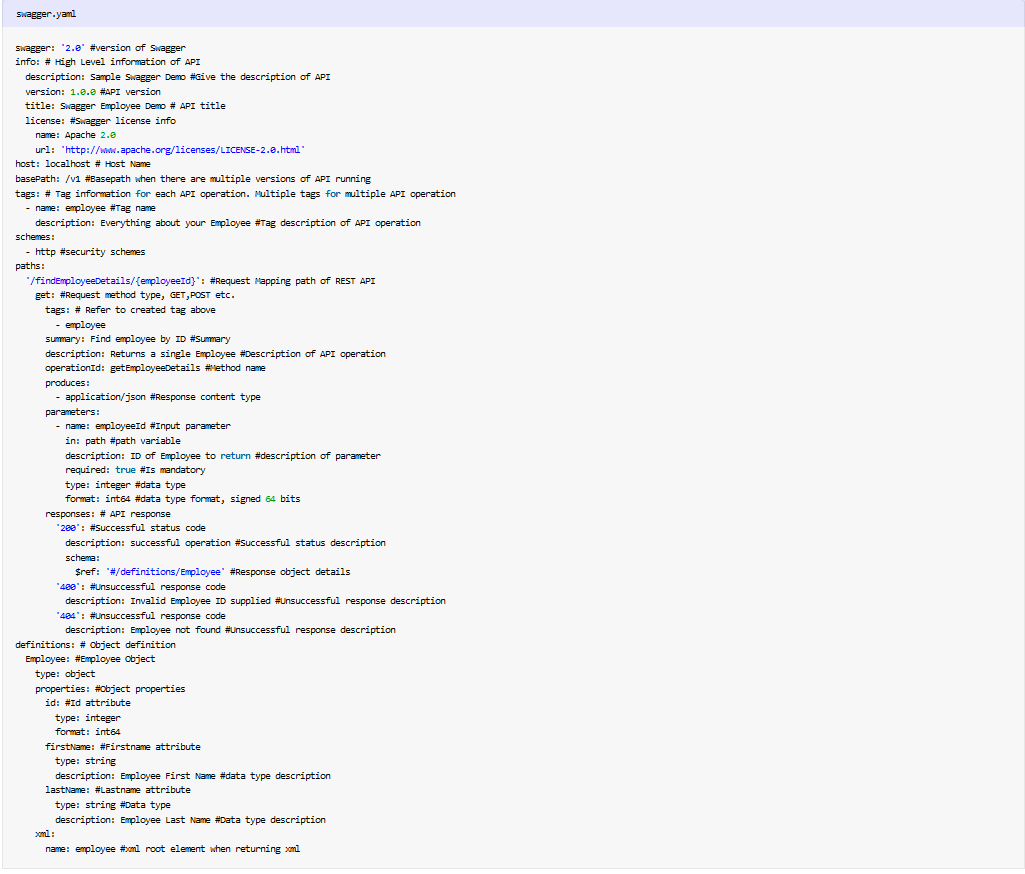
#### Create REST API contract

To create API contract, let’s use online [Swagger Editor](https://swagger.io/swagger-editor/) for the sake of simplicity. You can also download and install the same.

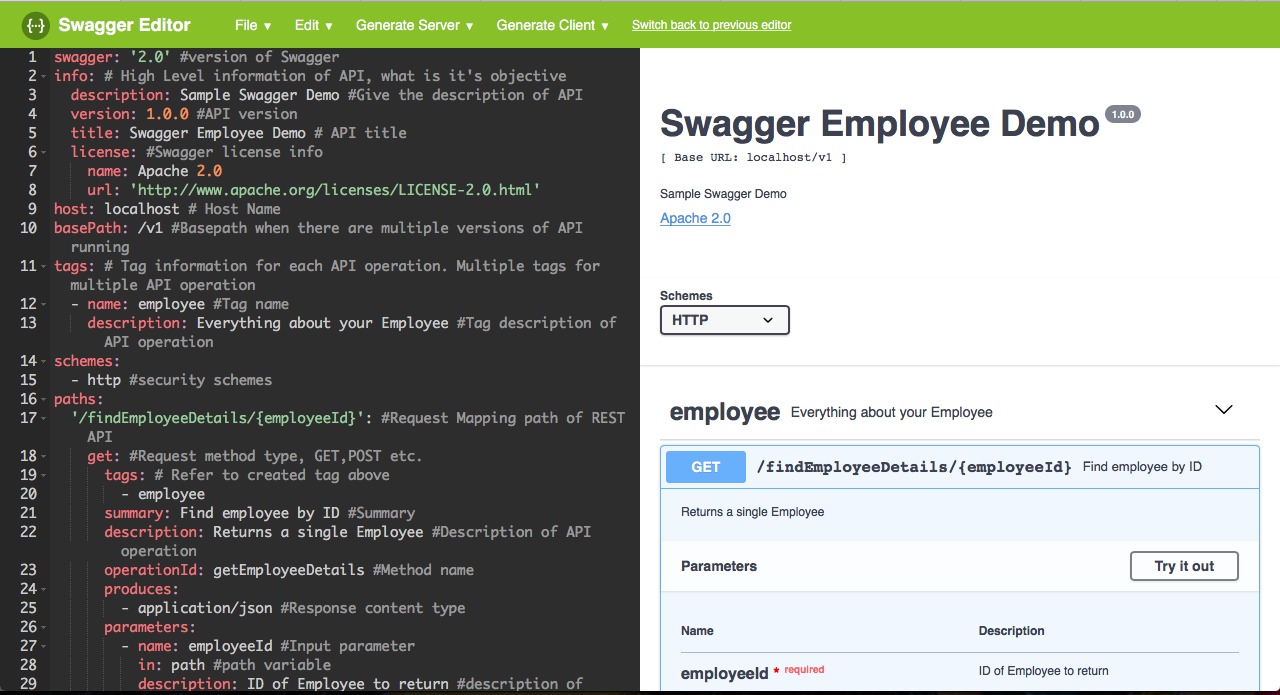
To create contract first have some understanding of [OpenAPI specification](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md). In this demo an Employee service contract has been created which will find employee details based on ID.

In the left pane of Swagger Editor write down the specification. Based on the specification outcome, right pane will show the UI documentation of Swagger.

Please follow the inline comments to understand below specification.



* Put the above specification in Swagger editor left pane and see the Swagger documentation visually.
* Save the specification as yaml file from editor File>Save as YAML. It will be saved as swagger.yaml.

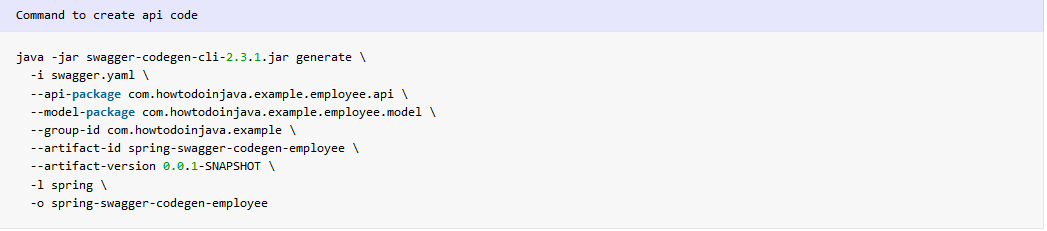


#### Generate API code with swagger codegen tool

In earlier step,we have saved the specification in yaml format. To further generate source code, this swagger.yaml file will be the source of input. To facilitate it, **Swagger CodeGen** tool is used.

Swagger provides utility jar to generate client REST clients for different programming languages and framework. The latest stable version jar can be downloaded from [Swagger Codegen](https://github.com/swagger-api/swagger-codegen/releases/tag/v2.3.1).

To generate client execute below command with **swagger cli**.



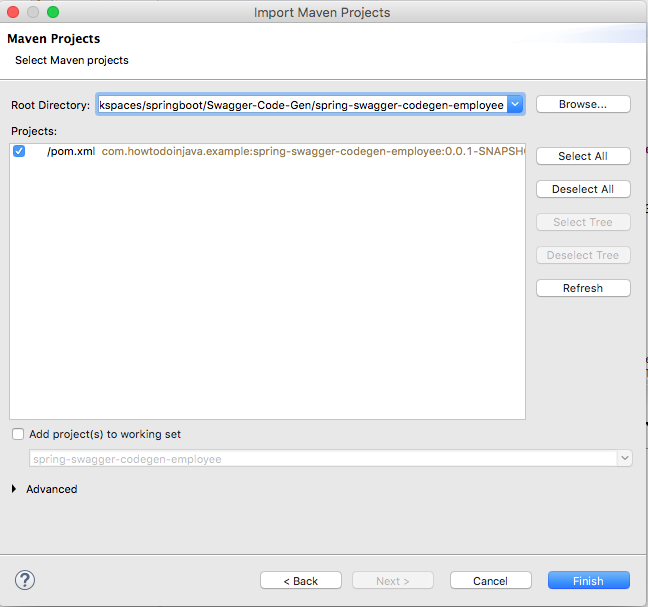
Description of arguments:

* **i**: Swagger specification source file
* **api-package**: Package information for generated API class
* **model-package**: Package information for generated model class
* **group-id**: Maven properties
* **artifact-id**: Maven properties
* **artifact-version**: Maven properties
* **l**: Implementation framework, here Spring is used, which by default provides spring-boot
* **o**: Output directory

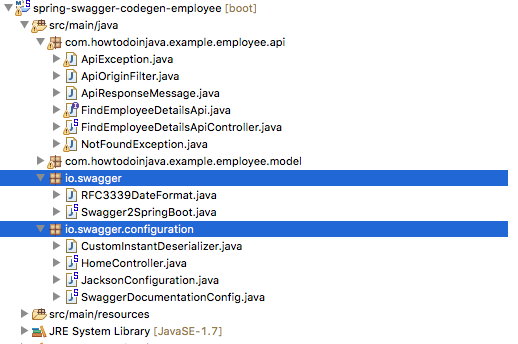
After successful execution of above command, a Spring boot maven projectspring-swagger-codegen-employee will be created.

#### Publish REST API

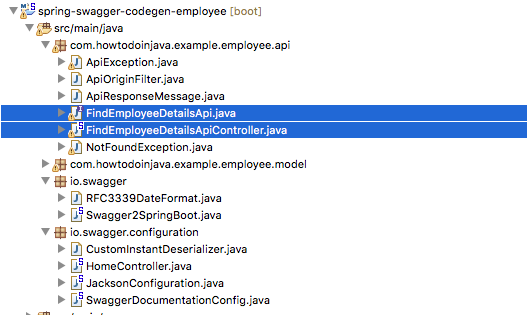
Open Spring tool suite and import as maven project we created in last step.



Once imported and project is built successfully, you can find swagger configuration classes are automatically created.



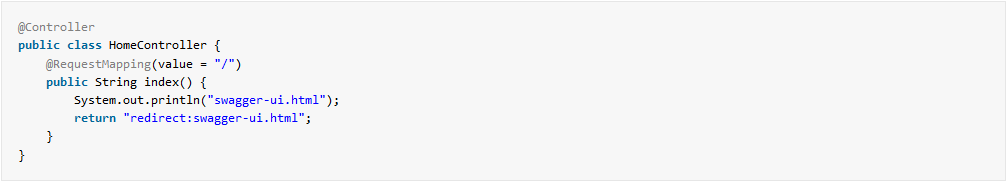
Verify the API controller FindEmployeeDetailsApiController.java and associated interfaces, you will see all Swagger annotations are already added.

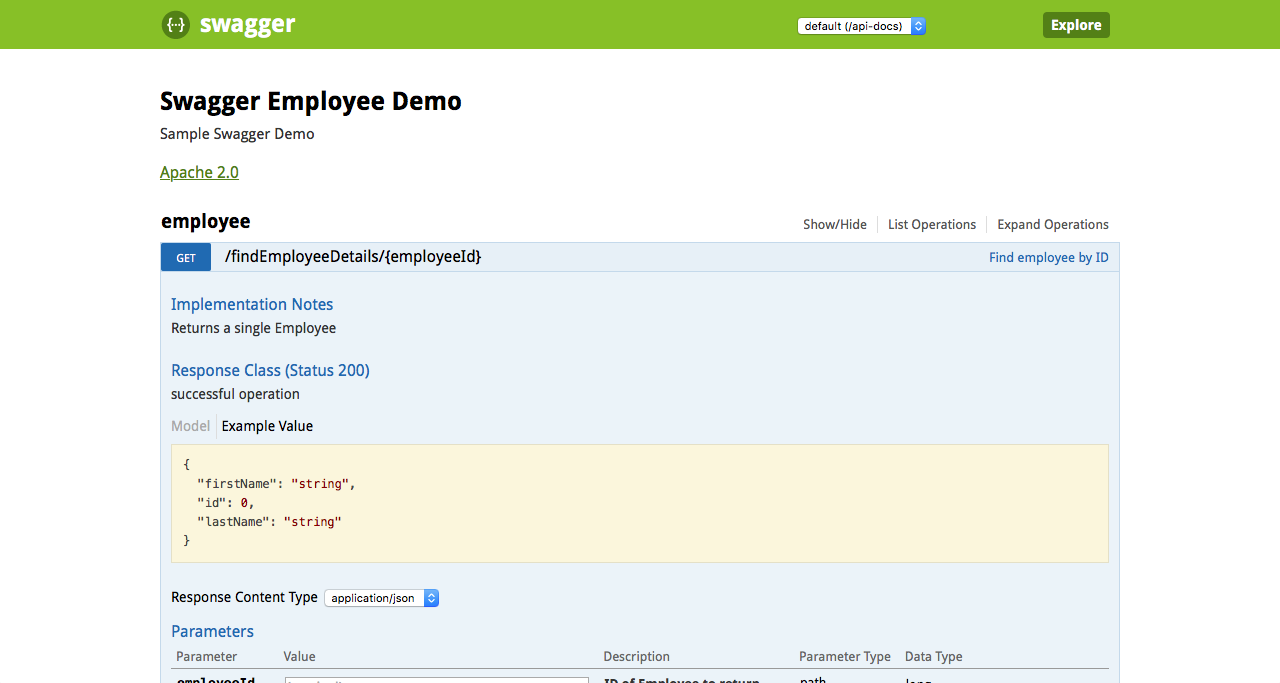


|  |
| --- |
|  |
|  |
|  |
|  |



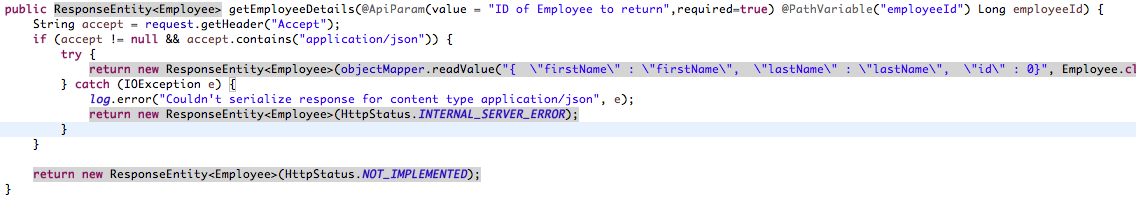
Next run the project and open Swagger UI accessible at http://localhost:8080/v1/swagger-ui.html. You can check all API details in Swagger UI.



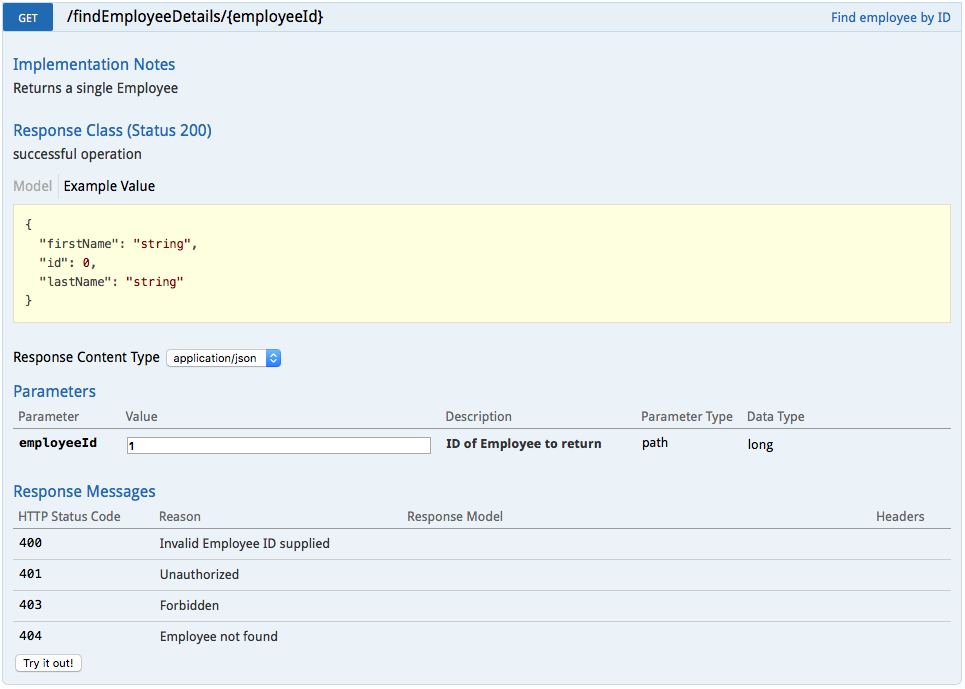


Now let’s access the default implemented service – findEmployeeDetails/{employeeid}. As there is no business logic implemented in controller class, you will see HTTP status code 501 (Not Implemented).

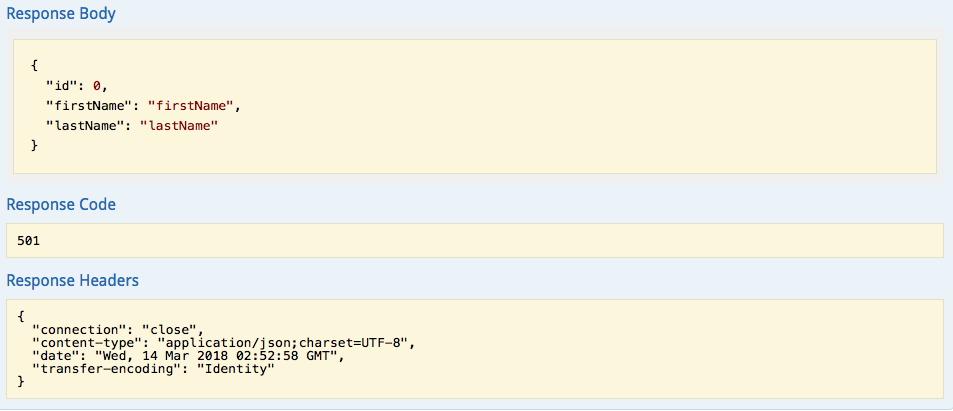
Let’s access the REST service through Swagger and REST client to see the default response.



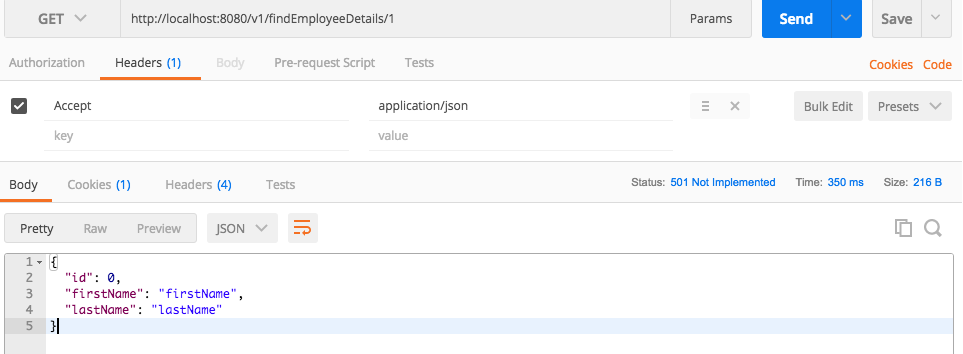
Swagger UI Request:



Swagger UI Response:



REST Client Response: Snapshot of Postman client



Now customise the service method as per business need while actual implementation. Notice the effort it is saving to integrate Swagger, also this is quite useful to get structured code ready during implementation of any new API.

### Bottom-Up Approach (Code First)

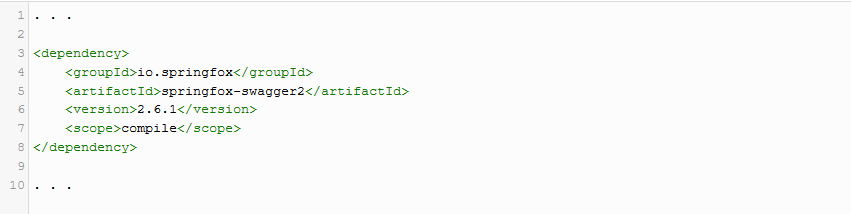
Swagger 2 is an open-source project used to describe and document RESTful APIs.

I’ll cover how to use [Swagger 2](http://swagger.io/) to generate REST API documentation for a Spring Boot project.

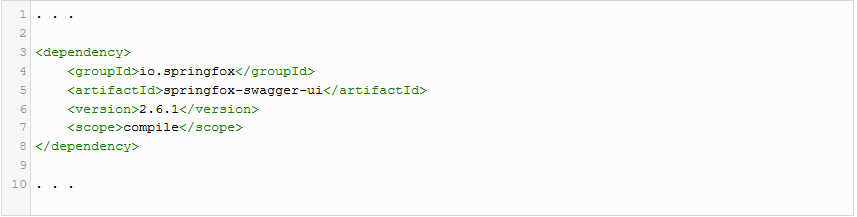
The Swagger 2 specification, which is known as [OpenAPI specification](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md), has several implementations. Currently, [Springfox](https://springfox.github.io/springfox/) that has replaced Swagger-SpringMVC (Swagger 1.2 and older) is popular for Spring Boot applications. Springfox supports both Swagger 1.2 and 2.0.

We will be using Springfox in our project.

To bring it in, we need the following dependency declaration in our Maven POM.



In addition to Sprinfox, we also require Swagger UI. The code to include Swagger UI is this.\



#### The Spring Boot RESTful Application

Our application implements a set of REST endpoints to manage products. We have a Product JPA entity and a repository named ProductRepository that extends CrudRepository to perform CRUD operations on products against an in-memory H2 database.

The service layer is composed of a ProductService interface and a ProductServiceImpl implementation class.

The Maven POM of the application is this.

pom.xml:

<?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>guru.springframework</groupId>

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

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>Spring Boot Web Application</name>

<description>Spring Boot Web Application</description>

<parent>

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

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

<version>1.4.2.RELEASE</version>

<relativePath/>

<!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.jayway.jsonpath</groupId>

<artifactId>json-path</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.6.1</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.6.1</version>

<scope>compile</scope>

</dependency>

<!--WebJars-->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

The controller of the application, ProductController, defines the REST API endpoints. The code of ProductController is this:

. . .

@RestController

@RequestMapping("/product")

public class ProductController {

private ProductService productService;

@Autowired

public void setProductService(ProductService productService) {

this.productService = productService;

}

@RequestMapping(value = "/list", method= RequestMethod.GET)

public Iterable list(Model model){

Iterable productList = productService.listAllProducts();

return productList;

}

@RequestMapping(value = "/show/{id}", method= RequestMethod.GET)

public Product showProduct(@PathVariable Integer id, Model model){

Product product = productService.getProductById(id);

return product;

}

@RequestMapping(value = "/add", method = RequestMethod.POST)

public ResponseEntity saveProduct(@RequestBody Product product){

productService.saveProduct(product);

return new ResponseEntity("Product saved successfully", HttpStatus.OK);

}

@RequestMapping(value = "/update/{id}", method = RequestMethod.PUT)

public ResponseEntity updateProduct(@PathVariable Integer id, @RequestBody Product product){

Product storedProduct = productService.getProductById(id);

storedProduct.setDescription(product.getDescription());

storedProduct.setImageUrl(product.getImageUrl());

storedProduct.setPrice(product.getPrice());

productService.saveProduct(storedProduct);

return new ResponseEntity("Product updated successfully", HttpStatus.OK);

}

@RequestMapping(value="/delete/{id}", method = RequestMethod.DELETE)

public ResponseEntity delete(@PathVariable Integer id){

productService.deleteProduct(id);

return new ResponseEntity("Product deleted successfully", HttpStatus.OK);

}

}

. . .

In this controller, the @RestController annotation introduced in Spring 4.0 marks ProductController as a REST API controller. Under the hood, @RestController works as a convenient annotation to annotate the class with the @Controller and @ResponseBody.

The @RequestMapping class-level annotation maps requests to /product onto the ProductController class. The method-level @RequestMapping annotations map web requests to the handler methods of the controller.

#### Configuring Swagger 2 in the Application

For our application, we will create a Docket bean in a Spring Boot configuration to configure Swagger 2 for the application. A Springfox Docket instance provides the primary API configuration with sensible defaults and convenience methods for configuration. Our Spring Boot configuration class, SwaggerConfig is this.

. . .

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket productApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.select() .apis(RequestHandlerSelectors.basePackage("guru.springframework.controllers"))

.paths(regex("/product.\*"))

.build();

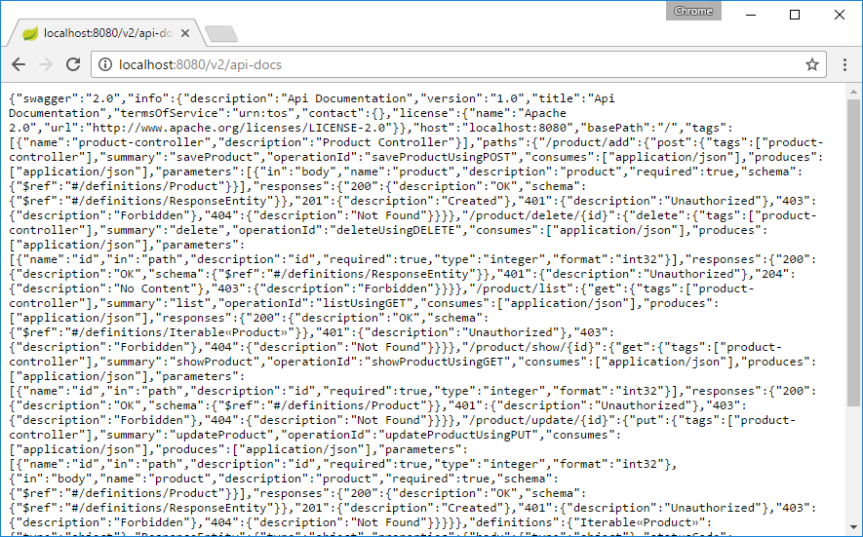
}

}

. . .

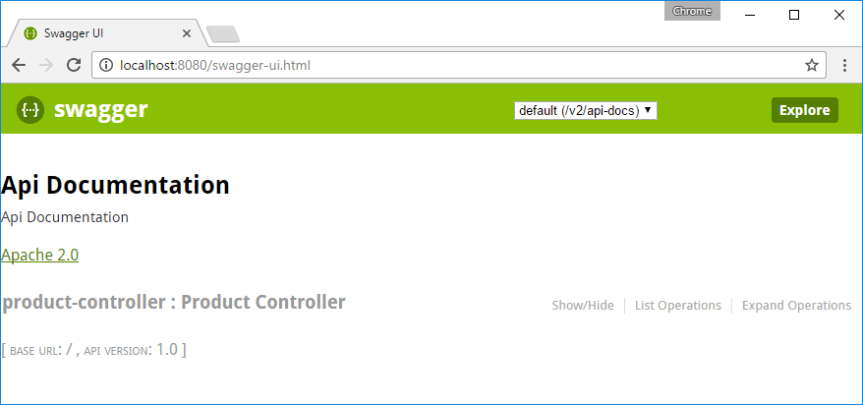
In this configuration class, the @EnableSwagger2 annotation enables Swagger support in the class. The select() method called on the Docket bean instance returns an ApiSelectorBuilder, which provides the apis() and paths() methods that are used to filter the controllers and methods that are being documented using String predicates.

In the code, the RequestHandlerSelectors.basePackage predicate matches the guru.springframework.controllers base package to filter the API. The regex parameter passed to paths()acts as an additional filter to generate documentation only for the path starting with /product.

At this point, you should be able to test the configuration by starting the app and pointing your browser to http://localhost:8080/v2/api-docs.  
[](https://i1.wp.com/springframework.guru/wp-content/uploads/2017/02/Swagger_JSON_Output.png?ssl=1)Obviously, the above JSON dump that Swagger 2 generates for our endpoints is not something we want.

What we want is some nice human readable structured documentation, and this is where Swagger UI takes over.

On pointing your browser to http://localhost:8080/swagger-ui.html, you will see the generated documentation rendered by Swagger UI, like this:

[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/02/Swagger_Default_Documentation.png?ssl=1)

As you can see, Swagger 2 used sensible defaults to generate the documentation of our ProductController.

Then, Swagger UI wrapped everything up to provide us an intuitive UI. This was all done automatically. We did not write any code or other documentation to support Swagger.

#### Customizing Swagger

So far, we’ve been looking at Swagger documentation as it comes out of the box — but Swagger 2 has some great customization options.

Let’s start customizing Swagger by providing information about our API in the SwaggerConfig class like this.

SwaggerConfig.java:

package guru.springframework.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.RequestHandlerSelectors;

import springfox.documentation.service.ApiInfo;

import springfox.documentation.service.Contact;

import springfox.documentation.spi.DocumentationType;

import springfox.documentation.spring.web.plugins.Docket;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

import static springfox.documentation.builders.PathSelectors.regex;

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket productApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.basePackage("guru.springframework.controllers"))

.paths(regex("/product.\*"))

.build()

.apiInfo(metaData());

}

private ApiInfo metaData() {

ApiInfo apiInfo = new ApiInfo(

"Spring Boot REST API",

"Spring Boot REST API for Online Store",

"1.0",

"Terms of service",

new Contact("John Thompson", "https://springframework.guru/about/", "john@springfrmework.guru"),

"Apache License Version 2.0",

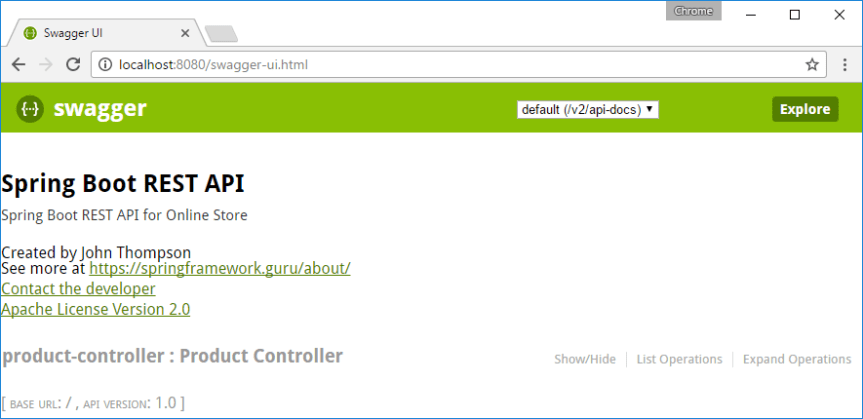
"https://www.apache.org/licenses/LICENSE-2.0");

return apiInfo;

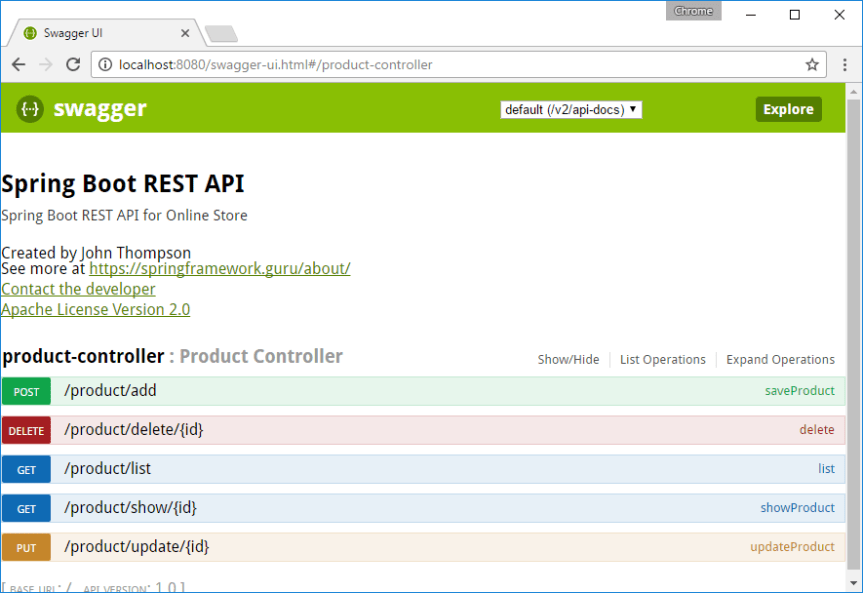
}

}

In the SwaggerConfig class, we have added a metaData() method that returns and ApiInfo object initialized with information about our API. Line 23 initializes the Docket with the new information.

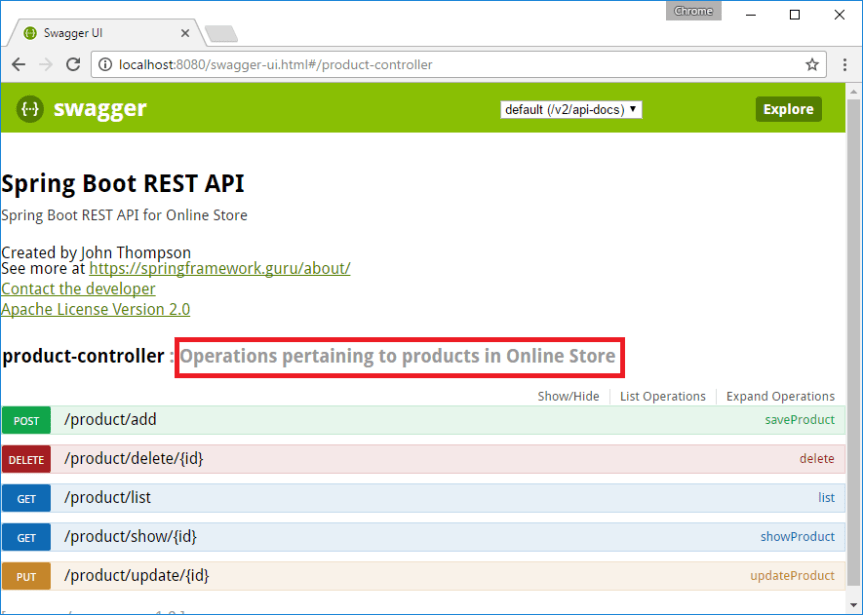
The Swagger 2-generated documentation now looks similar to this:  
[](https://i1.wp.com/springframework.guru/wp-content/uploads/2017/02/swagger-ui_with_API_information-2.png?ssl=1)

#### Swagger 2 Annotations for REST Endpoints

At this point, if you click the product controller link, Swagger UI will display the documentation of our operation endpoints, like this:  
[](https://i1.wp.com/springframework.guru/wp-content/uploads/2017/02/swagger-ui_with_default_endpoint_documentation.png?ssl=1)

We can use the @Api annotation on our ProductController class to describe our API.

@RestController @RequestMapping("/product") @Api(value="onlinestore", description="Operations pertaining to products in Online Store") public class ProductController { . . . . }

The Swagger UI-generated documentation will reflect the description and now looks like this:  
[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/02/swagger-ui_with_API_Description.png?ssl=1)For each of our operation endpoints, we can use the @ApiOperation annotation to describe the endpoint and its response type, like this:

. . .

@ApiOperation(value = "View a list of available products", response = Iterable.class)

@RequestMapping(value = "/list", method= RequestMethod.GET,produces = "application/json")

public Iterable list(Model model){

Iterable productList = productService.listAllProducts();

return productList;

}

. . .

Swagger 2 also allows overriding the default response messages of HTTP methods. You can use the @ApiResponse annotation to document other responses, in addition to the regular HTTP 200 OK, like this.

. . .

@ApiOperation(value = "View a list of available products", response = Iterable.class)

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved list"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@RequestMapping(value = "/list", method= RequestMethod.GET, produces = "application/json")

public Iterable list(Model model){

Iterable productList = productService.listAllProducts();

return productList;

}

. . .

One undocumented thing that took quite some of my time was related to the value of Response Content Type. Swagger 2 generated \*/\*, while I was expecting application/json for Response Content Type. It was only after updating the @RequestMapping annotation, which produces = "application/json", that the desired value got generated. The annotated ProductController is below.

ProductController.java:

package guru.springframework.controllers;

import guru.springframework.domain.Product;

import guru.springframework.services.ProductService;

import io.swagger.annotations.Api;

import io.swagger.annotations.ApiOperation;

import io.swagger.annotations.ApiResponse;

import io.swagger.annotations.ApiResponses;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/product")

@Api(value="onlinestore", description="Operations pertaining to products in Online Store")

public class ProductController {

private ProductService productService;

@Autowired

public void setProductService(ProductService productService) {

this.productService = productService;

}

@ApiOperation(value = "View a list of available products",response = Iterable.class)

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved list"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@RequestMapping(value = "/list", method= RequestMethod.GET, produces = "application/json")

public Iterable<Product> list(Model model){

Iterable<Product> productList = productService.listAllProducts();

return productList;

}

@ApiOperation(value = "Search a product with an ID",response = Product.class)

@RequestMapping(value = "/show/{id}", method= RequestMethod.GET, produces = "application/json")

public Product showProduct(@PathVariable Integer id, Model model){

Product product = productService.getProductById(id);

return product;

}

@ApiOperation(value = "Add a product")

@RequestMapping(value = "/add", method = RequestMethod.POST, produces = "application/json")

public ResponseEntity saveProduct(@RequestBody Product product){

productService.saveProduct(product);

return new ResponseEntity("Product saved successfully", HttpStatus.OK);

}

@ApiOperation(value = "Update a product")

@RequestMapping(value = "/update/{id}", method = RequestMethod.PUT, produces = "application/json")

public ResponseEntity updateProduct(@PathVariable Integer id, @RequestBody Product product){

Product storedProduct = productService.getProductById(id);

storedProduct.setDescription(product.getDescription());

storedProduct.setImageUrl(product.getImageUrl());

storedProduct.setPrice(product.getPrice());

productService.saveProduct(storedProduct);

return new ResponseEntity("Product updated successfully", HttpStatus.OK);

}

@ApiOperation(value = "Delete a product")

@RequestMapping(value="/delete/{id}", method = RequestMethod.DELETE, produces = "application/json")

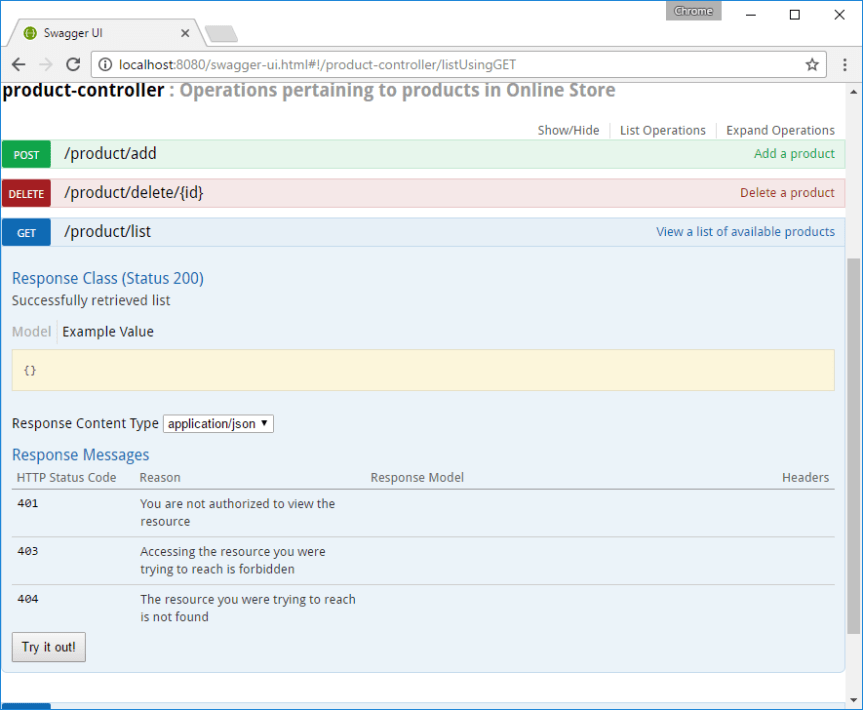
public ResponseEntity delete(@PathVariable Integer id){

productService.deleteProduct(id);

return new ResponseEntity("Product deleted successfully", HttpStatus.OK);

}

}

The output of the operation endpoints on the browser is this:  
[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/02/swagger-ui_with_endpoint_documentation.png?ssl=1)The current documentation is missing one thing: documentation of the Product JPA entity. We will generate documentation for our model next.

#### Swagger 2 Annotations for Model

You can use the @ApiModelProperty annotation to describe the properties of the Product model. With @ApiModelProperty, you can also document a property as required.

The code of our Product class is this.

Product.java:

package guru.springframework.domain;

import io.swagger.annotations.ApiModelProperty;

import javax.persistence.\*;

import java.math.BigDecimal;

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@ApiModelProperty(notes = "The database generated product ID")

private Integer id;

@Version

@ApiModelProperty(notes = "The auto-generated version of the product")

private Integer version;

@ApiModelProperty(notes = "The application-specific product ID")

private String productId;

@ApiModelProperty(notes = "The product description")

private String description;

@ApiModelProperty(notes = "The image URL of the product")

private String imageUrl;

@ApiModelProperty(notes = "The price of the product", required = true)

private BigDecimal price;

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public Integer getVersion() {

return version;

}

public void setVersion(Integer version) {

this.version = version;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

public String getImageUrl() {

return imageUrl;

}

public void setImageUrl(String imageUrl) {

this.imageUrl = imageUrl;

}

public BigDecimal getPrice() {

return price;

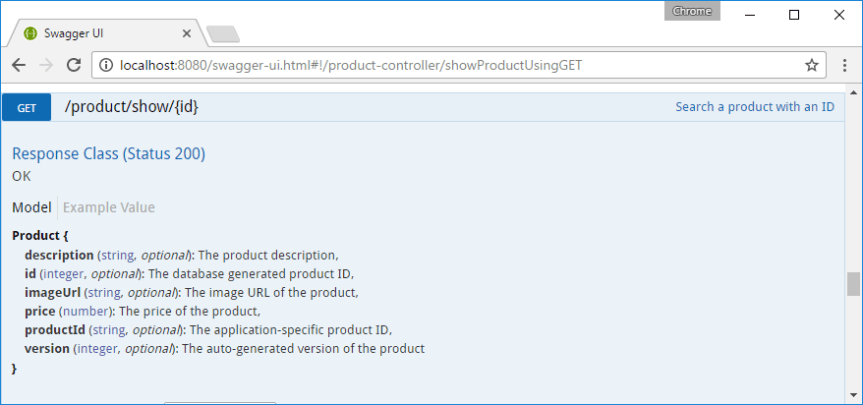
}

public void setPrice(BigDecimal price) {

this.price = price;

}

}

The Swagger 2 generated documentation for Product is this:  
[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/02/swagger_model_documentation.png?ssl=1)

#### Summary

Besides REST API documentation and presentation with Swagger Core and Swagger UI, Swagger 2 has a whole lot of other uses beyond the scope of this post. One of my favorites is [Swagger Editor](http://swagger.io/swagger-editor/), a tool to design new APIs or edit existing ones. The editor visually renders your Swagger definition and provides real-time error-feedback. Another one is [Swagger Codegen](http://swagger.io/swagger-codegen/), a code generation framework for building Client SDKs, servers, and documentation from Swagger definitions.

Swagger 2 also supports Swagger definition through JSON and YAML files. It is something you should try if you want to avoid implementation-specific code in your codebase by externalizing them in JSON and YAML files — something that I will cover in a future post.

## 3. **Spring Rest**

I n Spring’s approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the [@RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation, and the GreetingController below handles GET requests for /greeting by returning a new instance of the Greeting class:



This controller is concise and simple, but there’s plenty going on under the hood. Let’s break it down step by step.

The @RequestMapping annotation ensures that HTTP requests to /greeting are mapped to the greeting() method.

|  |  |
| --- | --- |
|  | The above example does not specify GET vs. PUT, POST, and so forth, because @RequestMapping maps all HTTP operations by default. Use @RequestMapping(method=GET) to narrow this mapping. |

@RequestParam binds the value of the query string parameter name into the name parameter of the greeting() method. If the name parameter is absent in the request, the defaultValue of "World" is used.

The implementation of the method body creates and returns a new Greeting object with idand content attributes based on the next value from the counter, and formats the given name by using the greeting template.

### Difference between MVC Controller and Restful Web Service

A key difference between a traditional MVC controller and the RESTful web service controller above is the way that the HTTP response body is created.

Rather than relying on a [view technology](https://spring.io/understanding/view-templates) to perform server-side rendering of the greeting data to HTML, this RESTful web service controller simply populates and returns a Greeting object. The object data will be written directly to the HTTP response as JSON.

he **@RestController** annotation in Spring MVC is nothing but a combination of @Controller and @ResponseBody annotation. It was added into Spring 4.0 to make the development of RESTful Web Services in Spring framework easier. If you are familiar with the [REST web services](http://www.java67.com/2017/04/3-great-books-to-learn-java-web-services-soap-and-restful.html) you know that the fundamental difference between a web application and a REST API is that the response from a web application is generally view (HTML + CSS + JavaScript) while REST API just return data in form of JSON or XML. This difference is also obvious in the **@Controller** and **@RestController** annotation. The job of @Controller is to create a Map of model object and find a view but @RestController simply return the object and object data is directly written into HTTP response as JSON or XML.  
  
This can also be done with traditional @Controller and use @ResponseBody annotation but since this is the default behavior of RESTful Web services, Spring introduced @RestController which combined the behavior of @Controller and @ResponseBody together.  
  
In short, following two code snippet are equal in Spring MVC:

@Controller

@ResponseBody

public class MVCController {

.. your logic

}

@RestController

public class RestFulController {

.... your logic

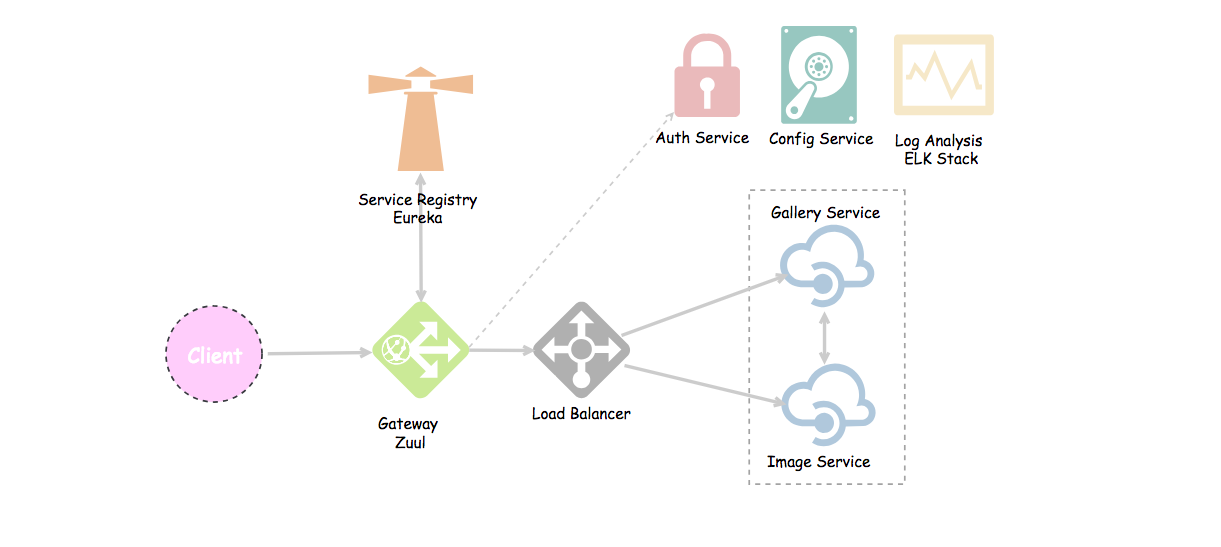
}

This code uses Spring 4’s new [@RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.

The Greeting object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because [Jackson 2](http://wiki.fasterxml.com/JacksonHome) is on the classpath, Spring’s [MappingJackson2HttpMessageConverter](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/json/MappingJackson2HttpMessageConverter.html) is automatically chosen to convert the Greeting instance to JSON.

## Spring Microservices

*A microservice is an engineering approach focused on****decomposing****applications into****single-function****modules with****well-defined interfaces****which are* ***independent****ly deployed and operated by****small teams****who own the****entire lifecycle****of the service.*



Benefits:

#### **Decomposing**

So, instead of having one large application, we decompose it into separate, different, mini-applications (services).

Each service handles a specific business domain (logging, auth, orders, customers) and provides the implementation for user interface, business logic, and connection to database.

class UserApp() {  
 User getUser() {  
 // 1. auth user  
 // 2. get user data  
 // 3. log user actions  
 }  
}

class UserApp() {  
 void authUser(User user) { ... }  
 User getUserData() { ... }  
 void logUserActions() { ... }  
}

#### Single-function

Each and every service has a specific function, or responsibility. And yes, a service can do many tasks, but all of them are nevertheless relevant to this single function.

#### **Well-defined interfaces**

Services must provide an interface that defines how can we communicate with it. This basically defines a list of methods, and their inputs and outputs.

#### Independent

Independent means services doesn’t know about each other implementation. They can get tested, deployed, and maintained independenly.

It might be the case where services are implemented using different language stacks, and communicate with different databases.

But that doesn’t mean they don’t work together. They do, in order to complete there required operation.

class UserApp() {  
 void authUser(User user) {  
 // log user login action (success or failure)   
 // using logUserActions  
 }  
 User getUserData() { ... }  
 void logUserActions() { ... }  
}

#### Small Teams

We split the work up and team across the services. Each team focuses on a specific service, they don’t need to know about internal workings of other teams.

Those teams are can work efficiently, communicate easily, and each service can be deployed rapidly as soon as it’s ready.

#### Entire **Lifecycle**

The team is resposible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.

In traditional application, we may have a team for coding, and another one for deployment. In microservices, that’s not the case.

#### **Minimizing Communication**

Minimizing communication doesn’t mean that the team members should ignore each other. It means the only essential cross-team communication should be through the interface that each service provides.

They all need to agree on the external interface, so that communication between services is clearly defined.

#### **The scope and risk of change**

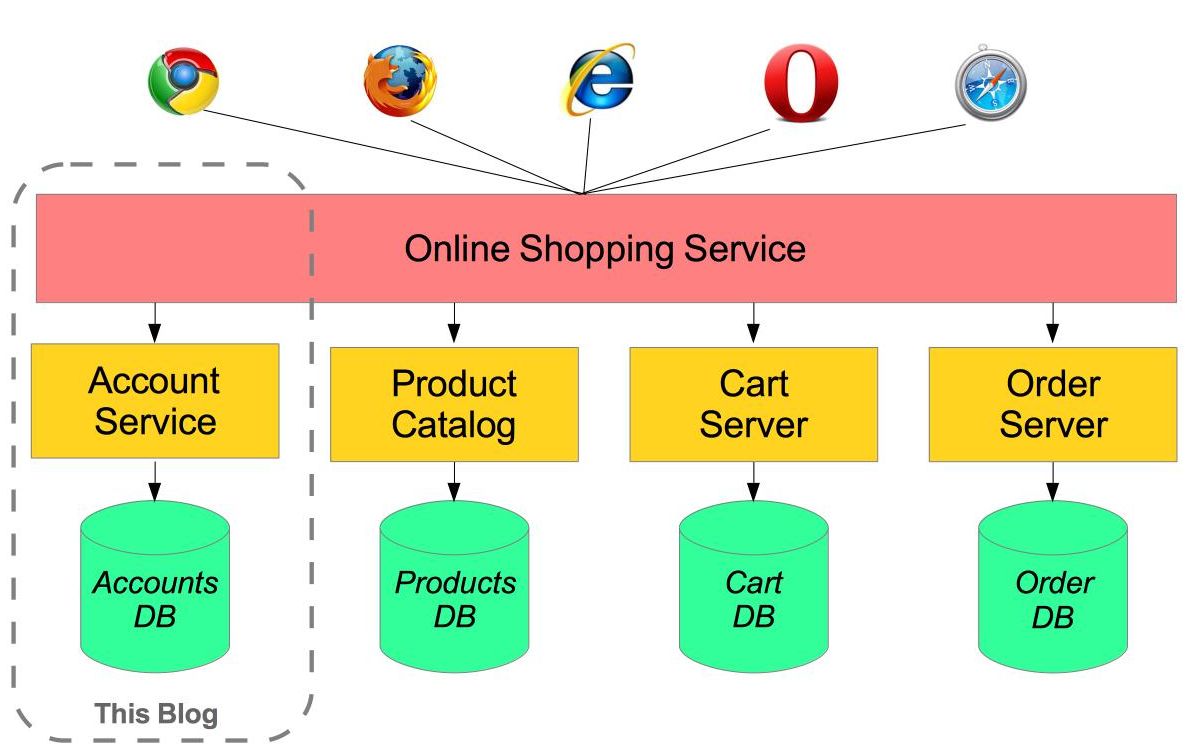
Services should be changed without breaking other services. And so long as we don’t change the external interface there will be no problem for other services.

As a result of changes, the versions of services are updating individually, and there is no relationship between them.

Example:

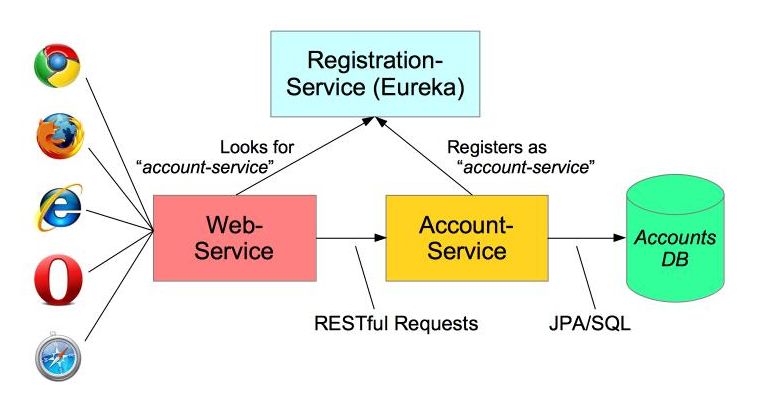
A simple example of setting up a microservices system using Spring, Spring Boot and Spring Cloud.

[Microservices](http://martinfowler.com/articles/microservices.html) allow large systems to be built up from a number of collaborating components. It does at the process level what Spring has always done at the component level: loosely coupled processes instead of loosely coupled components.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/shopping-system.jpg)

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts:

Inevitably there are a number of moving parts that you have to setup and configure to build such a system. How to get them working together is not obvious - you need to have good familiarity with Spring Boot since Spring Cloud leverages it heavily, several Netflix or other OSS projects are required and, of course, there is some Spring configuration “magic”!

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/mini-system.jpg)

In this article I aim to clarify how things work by building the simplest possible system step-by-step. Therefore, I will only implement a small part of the big system - the user account service.

The Web-Application will make requests to the Account-Servicemicroservice using a RESTful API. We will also need to add a discovery service – so the other processes can find each other.

The code for this application is here: <https://github.com/paulc4/microservices-demo>.

The description of how it works is deliberately detailed. Impatient readers may prefer to simply look at the [code](https://github.com/paulc4/microservices-demo). Note that it contains three microservices in a single project.

### Follow-Up 1: Other Resources

This article only discusses a minimal system. For more information, you might like to read Josh Long’s blog article [Microservice Registration and Discovery with Spring Cloud and Netflix’s Eureka](https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka) which shows running a complete microservice system on Cloud Foundry.

The Spring Cloud projects are [here](https://projects.spring.io/spring-cloud/).

### Follow Up 2: SpringOne Platform 2018

Book your place at [SpringOne2 Platform in Washington DC, USA](https://springoneplatform.io/) this September - simply the best opportunity to find out first hand all that’s going on and to provide direct feedback. The name has changed, from Spring One, to reflect the growth of Spring in platform services (such as the Spring Cloud projects).

### Updates (June 2018)

A number of changes since I originally wrote this blog:

1. A [discussion](https://spring.io/blog/2015/07/14/microservices-with-spring#configuration-options) of using multiple instances of the same service on the same host.. Demo application updated to match.
2. A [discussion](https://spring.io/blog/2015/07/14/microservices-with-spring#load-balanced-resttemplate) of @LoadBalanced - how this works has changed since the Brixton release-train ([Spring Cloud](https://projects.spring.io/spring-cloud) 1.1.0.RELEASE).
3. Refactored [configuration](https://spring.io/blog/2015/07/14/microservices-with-spring#accountsconfiguration-class) of Accounts microservice into its own class AccountsConfiguration.
4. Upgraded to Spring Boot 2, so a few Boot classes have changed package.
5. Upgraded [demo application](https://spring.io/blog/2015/07/14/microservices-with-spring#running-the-system) to Spring Cloud Finchley release-train (including various fixes from the comments at the end - thanks for the feedback).
6. The Eureka server dependency has changed to spring-cloud-starter-netflix-eureka-server.

Previous version, using Spring Boot 1.5.10 and Spring Cloud Edgeware SR3, is available as git tag v1.2.0.

OK, let’s get started …

### Service Registration

When you have multiple processes working together they need to find each other. If you have ever used Java’s RMI mechanism you may recall that it relied on a central registry so that RMI processes could find each other. Microservices has the same requirement.

The developers at Netflix had this problem when building their systems and created a registration server called Eureka (“I have found it” in Greek). Fortunately for us, they made their discovery server open-source and Spring has incorporated into Spring Cloud, making it even easier to run up a Eureka server. Here is the complete discovery-server application:

@SpringBootApplication

@EnableEurekaServer

public class ServiceRegistrationServer {

public static void main(String[] args) {

// Tell Boot to look for registration-server.yml

System.setProperty("spring.config.name", "registration-server");

SpringApplication.run(ServiceRegistrationServer.class, args);

}

}

It really is that simple!

Spring Cloud is built on Spring Boot and utilizes parent and starter POMs. The important parts of the [POM](https://github.com/paulc4/microservices-demo/blob/master/pom.xml) are:

<parent>

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

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

<version>2.0.1.RELEASE</version>

</parent>

<dependencies>

<dependency>

<!-- Setup Spring Boot -->

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

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

</dependency>

<dependency>

<!-- Setup Spring MVC & REST, use Embedded Tomcat -->

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

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

</dependency>

<dependency>

<!-- Spring Cloud starter -->

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter</artifactId>

</dependency>

<dependency>

<!-- Eureka for service registration -->

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

<!-- Spring Cloud dependencies -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

This POM has changed since I originally wrote the article to use Spring Boot as its parent not Spring Cloud. Spring Cloud dependencies are provided via the dependency management section.

An sample gradle build file is also included in the github code.

**Note:***Finchley.RELEASE* is the current "release train" - a set of co-ordinated releases -- see note on Spring Cloud [home page](https://projects.spring.io/spring-cloud/#release-trains).

By default Spring Boot applications look for an application.properties or application.ymlfile for configuration. By setting the spring.config.name property we can tell Spring Boot to look for a different file - useful if you have multiple Spring Boot applications in the same project - as I will do shortly.

This application looks for registration-server.properties or registration-server.yml. Here is the relevant configuration from registration-server.yml:

# Configure this Discovery Server

eureka:

instance:

hostname: localhost

client: # Not a client, don't register with yourself (unless running

# multiple discovery servers for redundancy)

registerWithEureka: false

fetchRegistry: false

server:

port: 1111 # HTTP (Tomcat) port

By default Eureka runs on port 8761, but here we will use port 1111 instead. Also by including the registration code in my process I might be a server or a client. The configuration specifies that I am not a client and stops the server process trying to register with itself.

### Using Consul

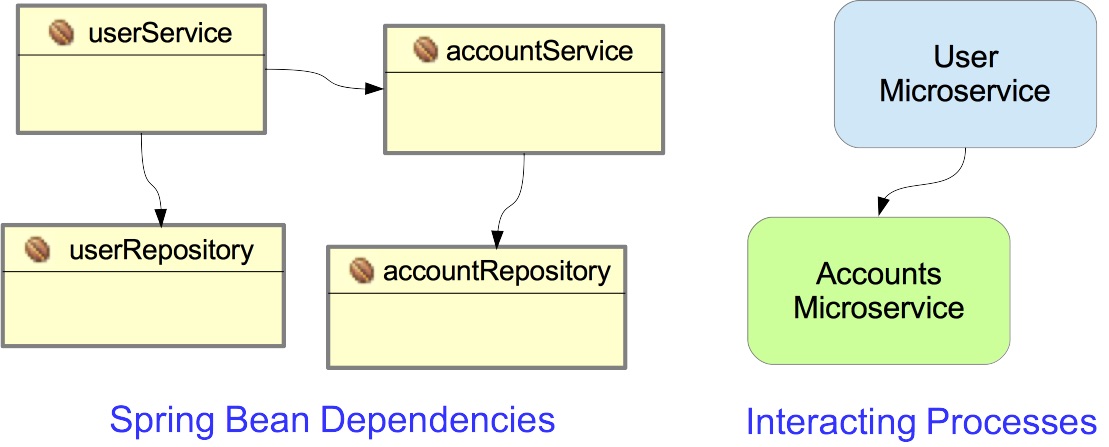
Spring Cloud also supports [Consul](https://www.consul.io/) as an alternative to Eureka. You start the Consul Agent (its registration server) using a script and then clients use it to find their microservices. For details, see this blog [article](https://spring.io/blog/2015/05/27/spring-cloud-consul-1-0-0-m1-available-now) or project [home page](https://cloud.spring.io/spring-cloud-consul).

Try running the RegistrationServer now (see [below](https://spring.io/blog/2015/07/14/microservices-with-spring#running-the-system) for help on running the application). You can open the Eureka dashboard here: [http://localhost:1111](http://localhost:1111/) and the section showing Applications will be empty.

From now on we will refer to the discovery-serversince it could be Eureka or Consul (see side panel).

### Creating a Microservice: **Account-Service**

A microservice is a stand-alone process that handles a well-defined requirement.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/beans-vs-processes.jpg)

When configuring applications with Spring we emphasize Loose Coupling and Tight Cohesion, These are not new concepts (Larry Constantine is credited with first defining these in the late 1960s - [reference](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29)) but now we are applying them, not to interacting components (Spring Beans), but to interacting processes.

In this example, I have a simple Account management microservice that uses Spring Data to implement a JPA AccountRepository and Spring REST to provide a RESTful interface to account information. In most respects this is a straightforward Spring Boot application.

What makes it special is that it registers itself with the discovery-server at start-up. Here is the Spring Boot startup class:

@EnableAutoConfiguration

@EnableDiscoveryClient

@Import(AccountsWebApplication.class)

public class AccountsServer {

@Autowired

AccountRepository accountRepository;

public static void main(String[] args) {

// Will configure using accounts-server.yml

System.setProperty("spring.config.name", "accounts-server");

SpringApplication.run(AccountsServer.class, args);

}

}

The annotations do the work:

1. @EnableAutoConfiguration - defines this as a Spring Boot application.
2. @EnableDiscoveryClient - this enables service registration and discovery. In this case, this process registers itself with the discovery-server service using its application name (see below).
3. @Import(AccountsWebApplication.class) - this Java Configuration class sets up everything else (see [below](https://spring.io/blog/2015/07/14/microservices-with-spring#accountswebapplication-configuration) for more details).

What makes this a microservice is the registration with the discovery-server via @EnableDiscoveryClient and its YML configuration completes the setup:

# Spring properties

spring:

application:

name: accounts-service

# Discovery Server Access

eureka:

client:

serviceUrl:

defaultZone: http://localhost:1111/eureka/

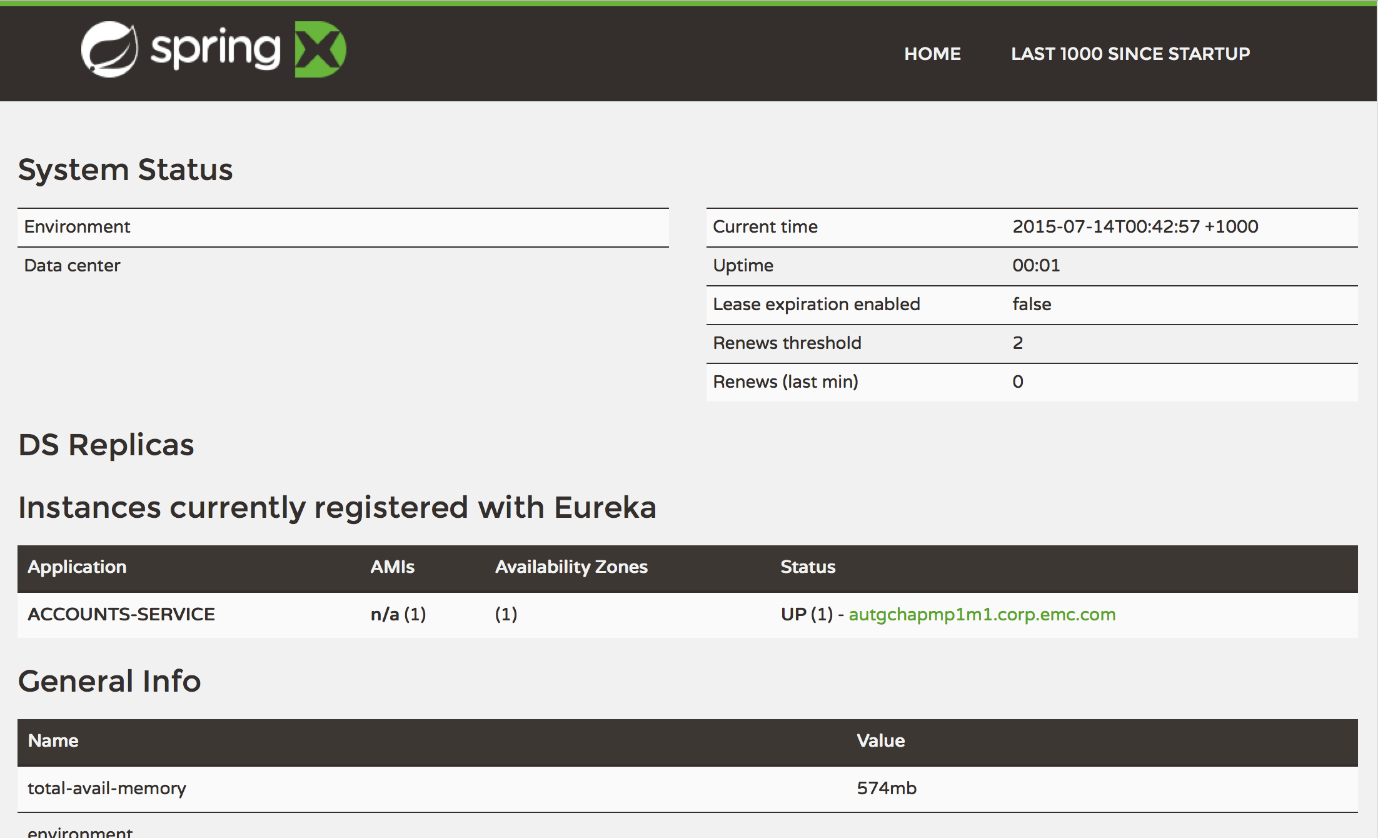
# HTTP Server

server:

port: 2222 # HTTP (Tomcat) port

Note that this file

1. Sets the application name as accounts-service. This service registers under this name and can also be accessed by this name - see below.
2. Specifies a custom port to listen on (2222). All my processes are using Tomcat, they can’t all listen on port 8080.
3. The URL of the Eureka Service process - from the previous section.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/dashboard.png)

Run the AccountsServiceapplication now and let it finish initializing. Refresh the dashboard [http://localhost:1111](http://localhost:1111/) and you should see the ACCOUNTS-SERVICE listed under Applications. Registration takes up to 30 seconds (by default) so be patient - check the log output from RegistrationService

**Warning:**Do not try to display XML output using the internal web-viewer of Eclipse/STS because it cannot do so. Use your favorite web browser instead.

For more detail, go here: <http://localhost:1111/eureka/apps/> and you should see something like this:

<applications>

<versions\_\_delta>1</versions\_\_delta>

<apps\_\_hashcode>UP\_1\_</apps\_\_hashcode>

<application>

<name>ACCOUNTS-SERVICE</name>

<instance>

<hostName>autgchapmp1m1.corp.emc.com</hostName>

<app>ACCOUNTS-SERVICE</app>

<ipAddr>172.16.84.1</ipAddr><status>UP</status>

<overriddenstatus>UNKNOWN</overriddenstatus>

<port enabled="true">3344</port>

<securePort enabled="false">443</securePort>

...

</instance>

</application>

</applications>

Alternatively go to <http://localhost:1111/eureka/apps/ACCOUNTS-SERVICE> and see just the details for AccountsService - if it’s not registered you will get a 404.

### Configuration Options

**Registration Time:** Registration takes up to 30s because that is the default client refresh time. You can change this by setting the eureka.instance.leaseRenewalIntervalInSecondsproperty to a smaller number (in the demo application I have set it to 5). This is not recommended in [production](https://cloud.spring.io/spring-cloud-static/docs/1.0.x/spring-cloud.html#_why_is_it_so_slow_to_register_a_service). See [also](https://github.com/spring-cloud/spring-cloud-netflix/issues/373).

eureka:

instance:

leaseRenewalIntervalInSeconds: 5 # DO NOT DO THIS IN PRODUCTION

**Registration Id:** A process (microservice) registers with the discovery-service using a unique id. If another process registers with the same id, it is treated as a restart (for example some sort of failover or recovery) and the first process registration is discarded. This gives us the fault-tolerant system we desire.

To run multiple instances of the same process (for load-balancing and resilience) they need to register with a unique id. When I first wrote this blog, that was automatic and since the Brixtonrelease-train, it is again.

Under the Angel release train, the instance-id, used by a client to register with a discovery server, was derived from the client’s service name (the same as the Spring application name) and also the client’s host name. The same processes running on the same host would therefore have the same id, so only one could ever register.

Fortunately you could set the id property manually via the client’s Eureka metadata map, like this:

eureka:

instance:

metadataMap:

instanceId: ${spring.application.name}:${spring.application.instance\_id:${server.port}}

Since the Brixton release train, this is now the default. So what does it do?

We are setting the instanceId to application-name:instance\_id, but if instance\_id is not defined, we will use application-name::server-port instead. Note that the spring.application.instance\_id is only set when using Cloud Foundry but it conveniently provides a unique id number for each instance of the same application. We can do something similar when running elsewhere by using the server-port (since different instances on the same machine must listen on different ports. Another example you will often see is ${spring.application.name}:${spring.application.instance\_id:${random.value}} but I personally find using the port number makes each instance easy to identify - the random values are just long strings that don’t mean anything.

**Note:** The syntax ${x:${y}} is Spring property shorthand for ${x} != null ? ${x} : ${y}.

Since the Brixton release there is also a dedicated property for this:

eureka:

instance:

instanceId: ${spring.application.name}:${spring.application.instance\_id:${random.value}}

#### Accessing the Microservice: **Web-Service**

To consume a RESTful service, Spring provides the RestTemplate class. This allows you to send HTTP requests to a RESTful server and fetch data in a number of formats - such as JSON and XML.

**Note:**The Accounts microservice provides a RESTful interface over HTTP, but any suitable protocol could be used. Messaging using [AMQP](https://rabbitmq.docs.pivotal.io/) or JMS is an obvious alternative (in which case the Discovery Server is no longer needed - instead processes need to know the names of the queues to talk to, consider using the [Spring Cloud Configuration Server](https://cloud.spring.io/spring-cloud-config/) for this).

Which formats can be used depends on the presence of marshaling classes on the classpath - for example JAXB is always detected since it is a standard part of Java. JSON is supported if Jackson jars are present in the classpath.

A microservice (discovery) client can use a RestTemplate and Spring will automatically configure it to be microservice aware (more of this in a moment).

#### Encapsulating Microservice Access

Here is part of the WebAccountService for my client application:

@Service

public class WebAccountsService {

@Autowired // NO LONGER auto-created by Spring Cloud (see below)

@LoadBalanced // Explicitly request the load-balanced template

// with Ribbon built-in

protected RestTemplate restTemplate;

protected String serviceUrl;

public WebAccountsService(String serviceUrl) {

this.serviceUrl = serviceUrl.startsWith("http") ?

serviceUrl : "http://" + serviceUrl;

}

public Account getByNumber(String accountNumber) {

Account account = restTemplate.getForObject(serviceUrl

+ "/accounts/{number}", Account.class, accountNumber);

if (account == null)

throw new AccountNotFoundException(accountNumber);

else

return account;

}

...

}

Note that my WebAccountService is just a wrapper for the RestTemplate fetching data from the microservice. The interesting parts are the serviceUrl and the RestTemplate.

#### Accessing the Microservice

As shown below, the serviceUrl is provided by the main program to the WebAccountController (which in turn passes it to the WebAccountService):

@SpringBootApplication

@EnableDiscoveryClient

@ComponentScan(useDefaultFilters=false) // Disable component scanner

public class WebServer {

// Case insensitive: could also use: http://accounts-service

public static final String ACCOUNTS\_SERVICE\_URL

= "http://ACCOUNTS-SERVICE";

public static void main(String[] args) {

// Will configure using web-server.yml

System.setProperty("spring.config.name", "web-server");

SpringApplication.run(WebServer.class, args);

}

@LoadBalanced // Make sure to create the load-balanced template

@Bean

RestTemplate restTemplate() {

return new RestTemplate();

}

/\*\*

\* Account service calls microservice internally using provided URL.

\*/

@Bean

public WebAccountsService accountsService() {

return new WebAccountsService(ACCOUNTS\_SERVICE\_URL);

}

@Bean

public WebAccountsController accountsController() {

return new WebAccountsController

(accountsService()); // plug in account-service

}

}

A few points to note:

1. The WebController is a typical Spring MVC view-based controller returning HTML. The application uses Thymeleaf as the view-technology (for generating dynamic HTML)
2. WebServer is also a @EnableDiscoveryClient but in this case as well as registering itself with the discovery-server (which is not necessary since it offers no services of its own) it uses Eureka to locate the account service.
3. The default component-scanner setup inherited from Spring Boot looks for @Componentclasses and, in this case, finds my WebAccountController and tries to create it. However, I want to create it myself, so I disable the scanner like this @ComponentScan(useDefaultFilters=false).
4. The service-url I am passing to the WebAccountController is the name the service used to register itself with the discovery-server - by default this is the same as the spring.application.name for the process which is account-service - see account-service.yml above. The use of upper-case is not required but it does help emphasize that ACCOUNTS-SERVICE is a logical host (that will be obtained via discovery) not an actual host.

#### Load Balanced RestTemplate

The RestTemplate bean will be intercepted and auto-configured by Spring Cloud (due to the @LoadBalanced annotation) to use a custom HttpRequestClient that uses Netflix [Ribbon](http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html) to do the microservice lookup. Ribbon is also a load-balancer so if you have multiple instances of a service available, it picks one for you. (Neither Eureka nor Consul on their own perform load-balancing so we use Ribbon to do it instead).

**Note:** From the Brixton Release Train (Spring Cloud 1.1.0.RELEASE), the RestTemplate is no longer created automatically. Originally it was created for you, which caused confusion and potential conflicts (sometimes Spring can be too helpful!).

Note that this instance is qualified using @LoadBalanced. (The [annotation](https://github.com/spring-cloud/spring-cloud-commons/blob/master/spring-cloud-commons/src/main/java/org/springframework/cloud/client/loadbalancer/LoadBalanced.java) is itself annotated with @Qualifier - see [here](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-autowired-annotation-qualifiers) for details). Thus if you have more than one RestTemplate bean, you can make sure to inject the right one, like this:

@Autowired

@LoadBalanced // Make sure to inject the load-balanced template

protected RestTemplate restTemplate;

If you look in the [RibbonClientHttpRequestFactory](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonClientHttpRequestFactory.java) you will see this code:

String serviceId = originalUri.getHost();

ServiceInstance instance =

loadBalancer.choose(serviceId); // loadBalancer uses Ribbon

... if instance non-null (service exists) ...

URI uri = loadBalancer.reconstructURI(instance, originalUri);

The loadBalancer takes the logical service-name (as registered with the discovery-server) and converts it to the actual hostname of the chosen microservice.

A RestTemplate instance is thread-safe and can be used to access any number of services in different parts of your application (for example, I might have a CustomerService wrapping the same RestTemplate instance accessing a customer data microservice).

### Configuration

Below the relevant configuration from web-server.yml. It is used to:

1. Set the application name
2. Define the URL for accessing the discovery server
3. Set the Tomcat port to 3333

# Spring Properties

spring:

application:

name: web-service

# Discovery Server Access

eureka:

client:

serviceUrl:

defaultZone: http://localhost:1111/eureka/

# HTTP Server

server:

port: 3333 # HTTP (Tomcat) port

#### How to Run the Demo

A small demo of this system is at [http://github.com/paulc4/microservices-demo](https://github.com/paulc4/microservices-demo). Clone it and either load into your favorite IDE or use maven directly. Suggestions on how to run the demo are included in the [README](https://github.com/paulc4/microservices-demo/blob/master/README.md) on the project homepage.

#### Extra Notes

Some notes about Spring Boot usage by these applications. If you are not familiar with Spring Boot, this explains some of the “magic”!

### View Templating Engines

The Eureka dashboard (inside RegistrationServer) is implemented using FreeMarker templates but the other two applications use Thymeleaf. To make sure each uses the right view engine, there is extra configuration in each YML file.

This is at the end of registration-server.yml to disable Thymeleaf.

...

# Discovery Server Dashboard uses FreeMarker. Don't want Thymeleaf templates

spring:

thymeleaf:

enabled: false # Disable Thymeleaf spring:

Since both AccountService and WebService use thymeleaf, we also need to point each at their own templates. Here is part of account-server.yml:

# Spring properties

spring:

application:

name: accounts-service # Service registers under this name

freemarker:

enabled: false # Ignore Eureka dashboard FreeMarker templates

thymeleaf:

cache: false # Allow Thymeleaf templates to be reloaded at runtime

prefix: classpath:/accounts-server/templates/

# Template location for this application only

...

web-server.yml is similar but its templates are defined by

prefix: classpath:/web-server/templates/

Note the / on the end of each spring.thymeleaf.prefix classpath - this is crucial.

### Command-Line Execution

The jar is compiled to automatically run io.pivotal.microservices.services.Main when invoked from the command-line - see [Main.java](https://github.com/paulc4/microservices-demo/blob/master/src/main/java/io/pivotal/microservices/services/Main.java).

The Spring Boot option to set the start-class can be seen in the [POM](https://github.com/paulc4/microservices-demo/blob/master/pom.xml):

<properties>

<!-- Stand-alone RESTFul application for testing only -->

<start-class>io.pivotal.microservices.services.Main</start-class>

</properties>

### AccountsConfiguration class

@SpringBootApplication

@EntityScan("io.pivotal.microservices.accounts")

@EnableJpaRepositories("io.pivotal.microservices.accounts")

@PropertySource("classpath:db-config.properties")

public class AccountsWebApplication {

...

}

This is the main configuration class for AccountService which is a classic Spring Boot application using Spring Data. The annotations do most of the work:

1. @SpringBootApplication - defines this as a Spring Boot application. This convenient annotation combines @EnableAutoConfiguration, @Configuration and @ComponentScan(which, by default, causes Spring to search the package containing this class, and its sub-packages, for components - potential Spring Beans: AccountController and AccountRepository) .
2. @EntityScan("io.pivotal.microservices.accounts") - because I am using JPA, I need to specify where the @Entity classes are. Normally this is an option you specify in JPA’s persistence.xml or when creating a LocalContainerEntityManagerFactoryBean. Spring Boot will create this factory-bean for me because the spring-boot-starter-data-jpadependency is on the class path. So an alternative way of specifying where to find the @Entity classes is by using@EntityScan. This will find Account.
3. @EnableJpaRepositories("io.pivotal.microservices.accounts")- look for classes extending Spring Data’s Repository marker interface and automatically implement them using JPA - see [Spring Data JPA](https://projects.spring.io/spring-data-jpa).
4. @PropertySource("classpath:db-config.properties") - properties to configure my DataSource – see [db-config.properties](https://github.com/paulc4/microservices-demo/blob/master/src/main/resources/db-config.properties).

### Configuring Properties

As mentioned above, Spring Boot applications look for either application.properties or application.yml to configure themselves. Since all three servers used in this application are in the same project, they would automatically use the same configuration.

To avoid that, each specifies an alternative file by setting the spring.config.name property.

For example here is part of WebServer.java.

public static void main(String[] args) {

// Tell server to look for web-server.properties or web-server.yml

System.setProperty("spring.config.name", "web-server");

SpringApplication.run(WebServer.class, args);

}

At runtime, the application will find and use web-server.yml in src/main/resources.

### Logging

Spring Boot sets up INFO level logging for Spring by default. Since we need to examine the logs for evidence of our microservices working, I have raised the level to WARN to reduce the amount of logging.

To do this, the logging level would need to be specified in each of the xxxx-server.ymlconfiguration files. This is usually the best place to define them as logging properties cannotbe specified in property files (logging has already been initialized before @PropertySource directives are processed). There is a note on this in the Spring Boot manual, but it’s easy to miss.

Rather than duplicate the logging configuration in each YAML file, I instead opted to put it in the logback configuration file, since Spring Boot uses logback - see [src/main/resources/logback.xml](https://github.com/paulc4/microservices-demo/blob/master/src/main/resources/logback.xml). All three services will share the same logback.xml.

## My SQL

My SQL is similar to Oracle database and can be downloaded from <https://dev.mysql.com/downloads/mysql/>.

## Angular 7 with Node JS

To start with Angular 7, need to install Node JS.

### **Getting started**

Welcome to Angular! Angular helps you build modern applications for the web, mobile, or desktop.

This guide shows you how to build and run a simple Angular app. You'll use the [Angular CLI tool](https://angular.io/cli) to accelerate development, while adhering to the [Style Guide](https://angular.io/guide/styleguide) recommendations that benefit every Angular project.

This guide takes less than 30 minutes to complete. At the end of this guide—as part of final code review—there is a link to download a copy of the final application code. (If you don't execute the commands in this guide, you can still download the final application code.)

### **Prerequisites**

Before you begin, make sure your development environment includes Node.js® and an npm package manager.

#### **Node.js**

Angular requires Node.js version 8.x or 10.x.

* To check your version, run node -v in a terminal/console window.
* To get Node.js, go to [nodejs.org](https://nodejs.org/).

#### **2. npm package manager**

Angular, the Angular CLI, and Angular apps depend on features and functionality provided by libraries that are available as [npm packages](https://docs.npmjs.com/getting-started/what-is-npm). To download and install npm packages, you must have an npm package manager.

This Quick Start uses the [npm client](https://docs.npmjs.com/cli/install) command line interface, which is installed with Node.js by default.

To check that you have the npm client installed, run npm -v in a terminal/console window.

### **Step 1: Install the Angular CLI**

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

Install the Angular CLI globally.

To install the CLI using npm, open a terminal/console window and enter the following command:

npm install -g @angular/cli

### **Step 2: Create a workspace and initial application**

You develop apps in the context of an Angular [**workspace**](https://angular.io/guide/glossary#workspace). A workspace contains the files for one or more [**projects**](https://angular.io/guide/glossary/#project). A project is the set of files that comprise an app, a library, or end-to-end (e2e) tests.

To create a new workspace and initial app project:

1. Run the CLI command ng new and provide the name my-app, as shown here:

ng new my-app

1. The ng new command prompts you for information about features to include in the initial app project. Accept the defaults by pressing the Enter or Return key.

The Angular CLI installs the necessary Angular npm packages and other dependencies. This can take a few minutes.

It also creates the following workspace and starter project files:

* A new workspace, with a root folder named my-app
* An initial skeleton app project, also called my-app (in the src subfolder)
* An end-to-end test project (in the e2e subfolder)
* Related configuration files

The initial app project contains a simple Welcome app, ready to run.

### **Step 3: Serve the application**

Angular includes a server, so that you can easily build and serve your app locally.

1. Go to the workspace folder (my-app).
2. Launch the server by using the CLI command ng serve, with the --open option.

cd my-app

ng serve --open

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

The --open (or just -o) option automatically opens your browser to [http](https://angular.io/api/common/http)://localhost:4200/.

Your app greets you with a message:



### **Step 4: Edit your first Angular component**

[Components](https://angular.io/guide/glossary#component) are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

#### **Angular components**

The page you see is the application shell. The shell is controlled by an Angular **component** named AppComponent.

Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

#### **Make changes to the application**

Open the project in your favorite editor or IDE and navigate to the src/app folder to make some changes to the starter app.

You'll find the implementation of the shell AppComponent distributed over three files:

1. app.component.ts— the component class code, written in TypeScript.
2. app.component.html— the component template, written in HTML.
3. app.component.css— the component's private CSS styles.

As part of the initial app, the CLI created the first Angular component for you. It is the root component, and it is named app-root.

1. Open ./src/app/app.component.ts.
2. Change the title property from 'my-app' to 'My First Angular App'.

src/app/app.component.ts

@[Component](https://angular.io/api/core/Component)({

selector: 'app-root',

templateUrl: './app.component.html',

[styleUrls](https://angular.io/api/core/Component#styleUrls): ['./app.component.css']

})

export class AppComponent {

title = 'My First Angular App!';

}

The browser reloads automatically with the revised title. That's nice, but it could look better.

1. Open ./src/app/app.component.css and give the component some style.

src/app/app.component.css

h1 {

color: #369;

font-family: Arial, Helvetica, sans-serif;

font-size: 250%;

}

Looking good!

