## **BJJD Project Work**

We can start project work in two parallel ways

1. Working with Spring Microservices (Spring Boot + Spring Rest + JPA + Swagger)
2. Working with Angular Project which will be used as our frond end to consume rest services and can be deployed at web, mobile and desktop.

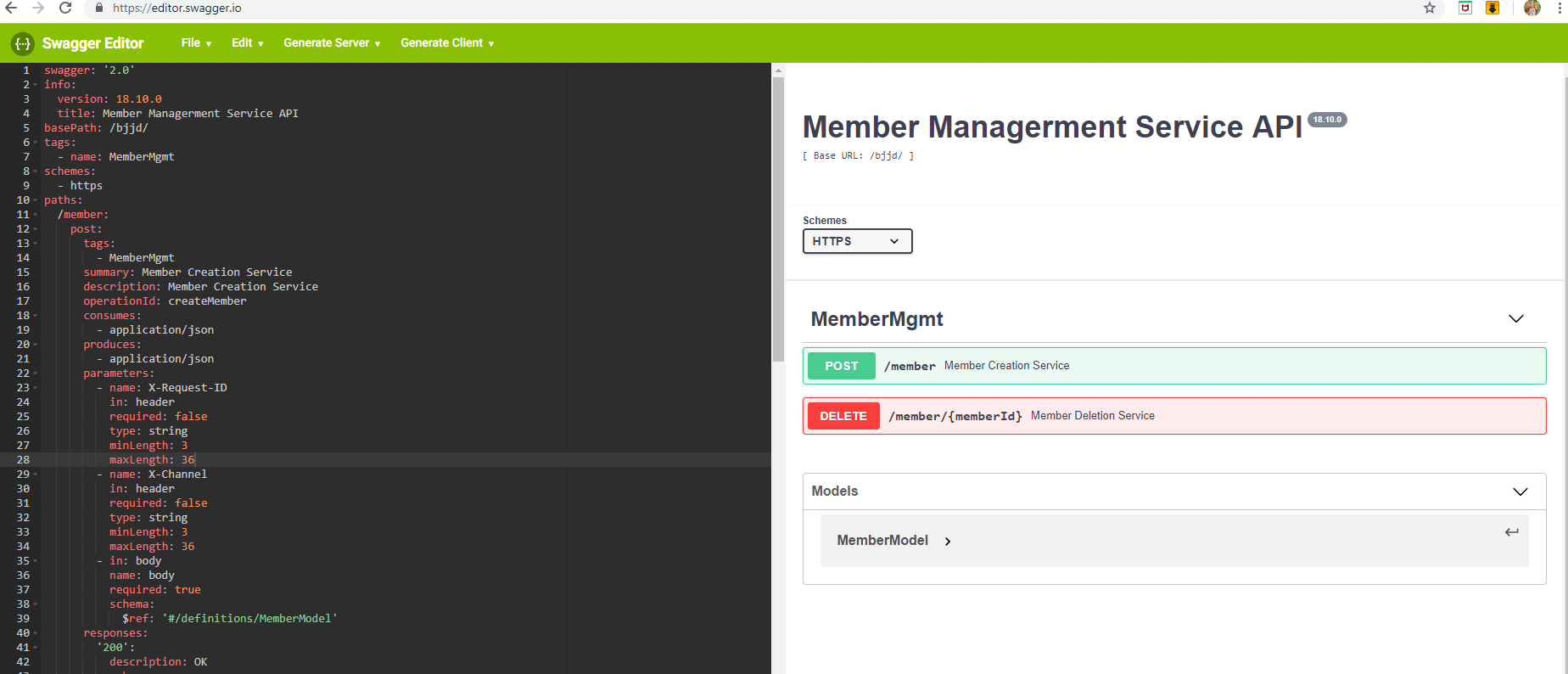
Technologies:

* Swagger-codegen-cli- 2.3.1
* Eclipse Jee 2018-09
* Sprint Boot: spring-boot-starter-parent: 2.1.6.RELEASE released in Nov-2018

# **Working with Spring Microservices**

First we will use the swagger specs (top down approach) to create the Spring Rest API with Spring Boot enabled project.

* Download the swagger-codegen-cli-2.3.1.jar from swagger website
* Create the member.yaml (swagger specs file as per the project requirement) 
* Validate the member.yaml file in the swagger editor: <https://editor.swagger.io/>



* Go to the command prompt where swagger-codegen-cli-2.3.1.jar and member.yaml files exists.
* Enter the following command (should be in one line) to create the **spring-boot** project named: **member-mgmt-service** in the current folder:

java -jar swagger-codegen-cli-2.3.1.jar generate

-i member.yaml

--api-package org.jmk.bjjd.member.api

--model-package org.jmk.bjjd.member.model

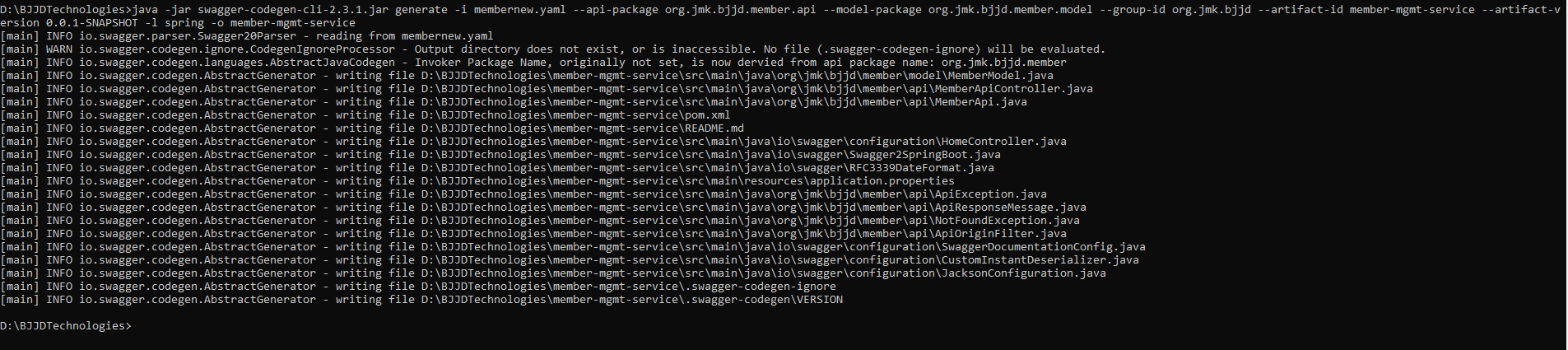
--group-id org.jmk.bjjd

--artifact-id member-mgmt-service

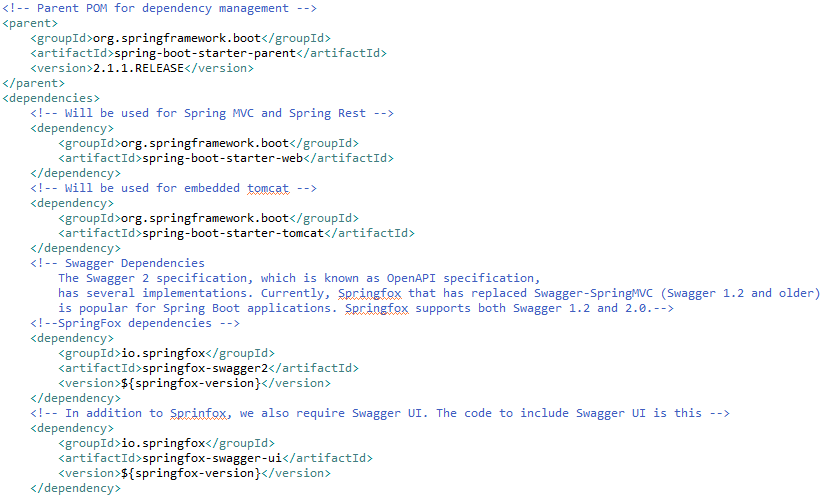
--artifact-version 0.0.1-SNAPSHOT

-l spring

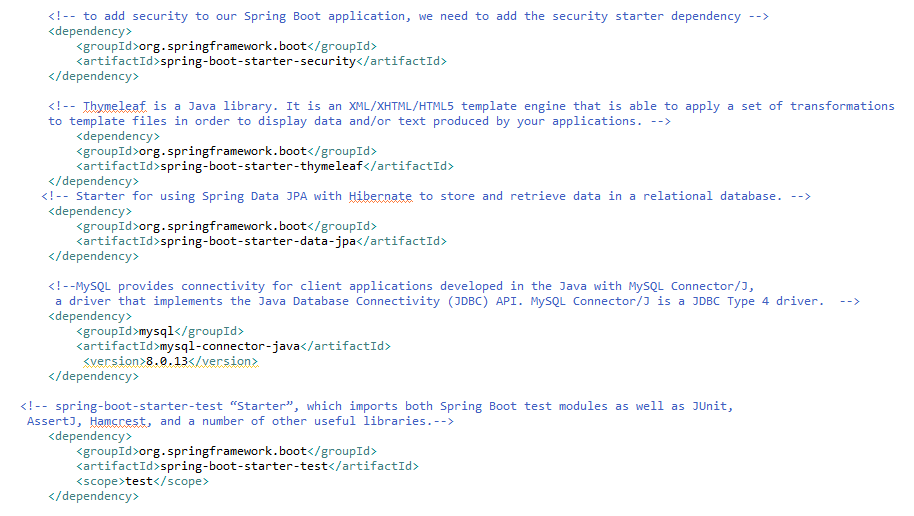
-o member-mgmt-service

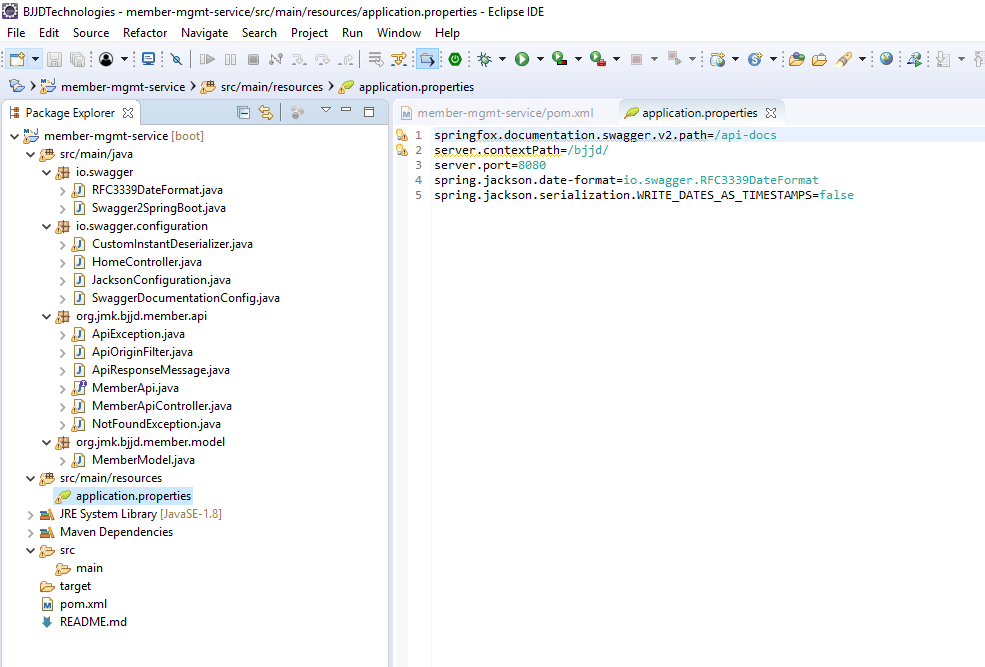


* Import this **member-mgmt-service as maven** into the Eclipse(enabled with Spring STS and Maven).
* This project comes with the following dependencies can be seen in pom.xml:



* Now update the pom.xml
  + Update the <java.version>1.7</java.version> to latest, in our case can be replaced with 1.8.
  + Update the version of spring-boot-starter-parent from 1.5.9.RELEASE to latest, in our case 2.1.1.RELEASE released in Nov-2018.
  + Add the following dependencies



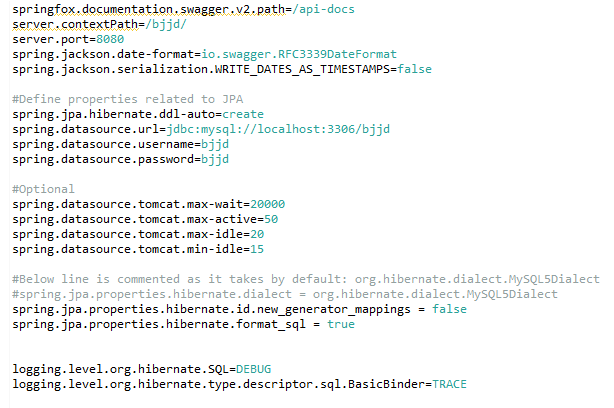
* Build or Update the project with Maven
* Now project code will be displayed as per the below screen shot.
* If we look into the MemberApiController class, need to make the following changes:
  + Annotation @Controller(Spring MVC) will be replaced with @RestController(Sprin Rest) annotation introduced in Spring 4.0 marks MemberApiController as a REST API controller. Under the hood, @RestController works as a convenient annotation to annotate the class with the @Controller and @ResponseBody.
  + Return type of every services should be ResponseEntity.
* Create the database named: root and user: bjjd using mysql command line or UI

User Id: root

Password: rajiv999

1. mysql> create database root; -- Create the new database/schema
2. mysql> create user 'bjjd'@'%' identified by 'bjjd'; -- Creates the user
3. mysql> grant all on root.\* to bjjd@'%'; -- Gives all the privileges to the new user on the newly created database/schema

* Define JPA Properties in src\main\resources\application.properties

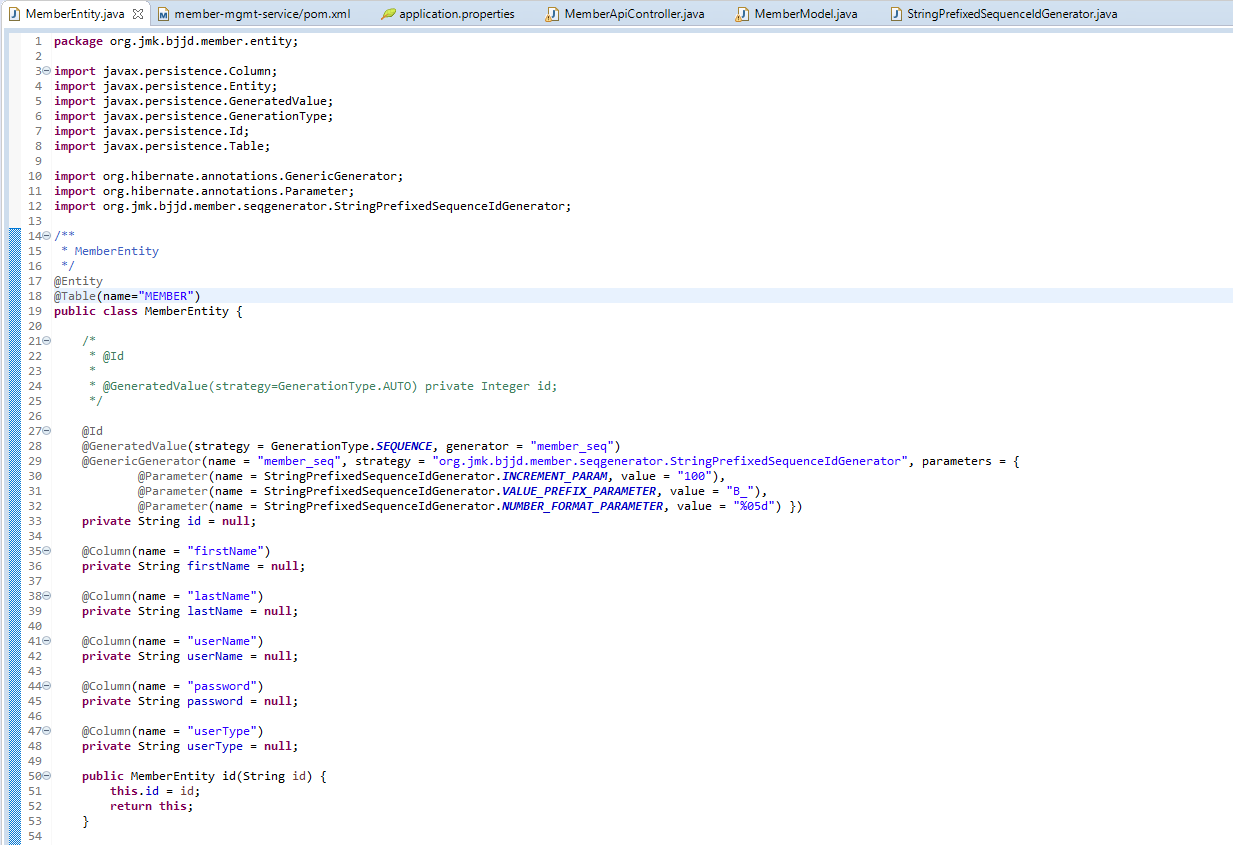


spring.jpa.hibernate.ddl-auto:

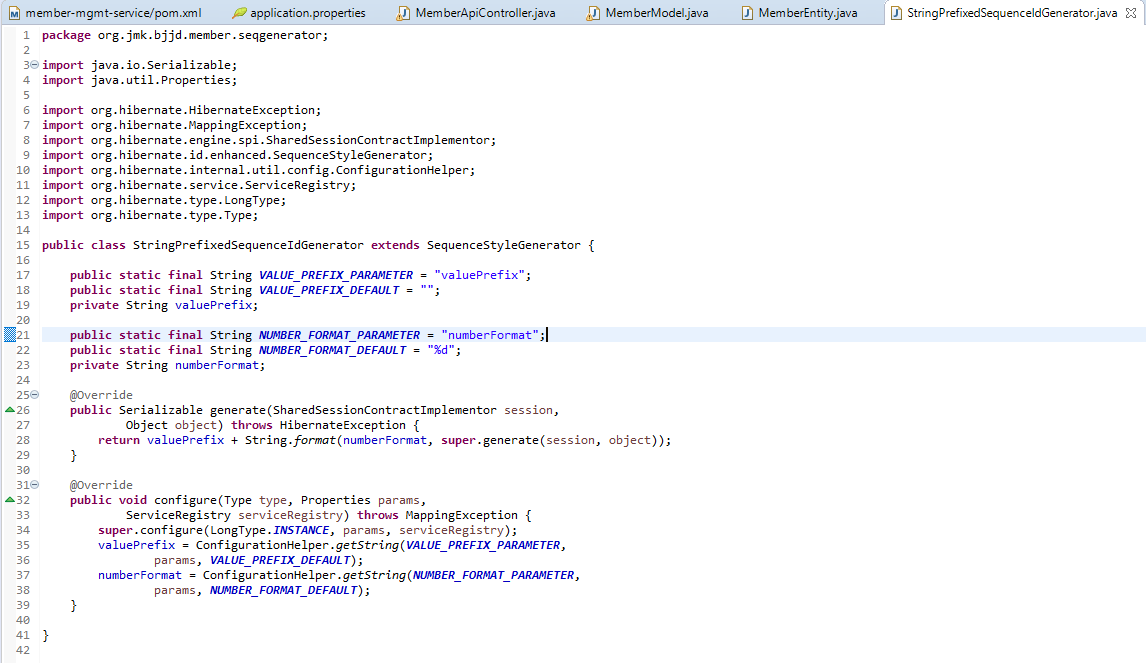
1. none This is the default for MySQL, no change to the database structure.
2. update Hibernate changes the database according to the given Entity structures.
3. create Creates the database every time, but don’t drop it when close.
4. create-drop Creates the database then drops it when the SessionFactory closes.

The data source properties starting with spring.datasource.\* will automatically be read by Spring boot JPA. To change the Hibernate properties we will use prefix spring.jpa.properties.\* with Hibernate property name. On the basis of given data source URL, Spring boot can automatically identify data source driver class. So we need not to configure diver class.   
Find the properties to configure JpaBaseConfiguration and HibernateJpaAutoConfiguration in application.properties.   
**spring.data.jpa.repositories.enabled**: It enables JPA repositories. The default value is **true**.   
**spring.jpa.database**: It targets database to operate on. By default embedded database is auto-detected.   
**spring.jpa.database-platform**: It is used to provide the name of database to operate on. By default it is auto- detected.   
**spring.jpa.generate-ddl**: It is used to initialize schema on startup. By default the value is **false**.   
**spring.jpa.hibernate.ddl-auto**: It is DDL mode used for embedded database. Default value is **create-drop**.   
**spring.jpa.hibernate.naming.implicit-strategy**: It is Hibernate 5 implicit naming strategy fully qualified name.   
**spring.jpa.hibernate.naming.physical-strategy**: It is Hibernate 5 physical naming strategy fully qualified name.   
**spring.jpa.hibernate.use-new-id-generator-mappings**: It is used for Hibernate IdentifierGenerator for AUTO, TABLE and SEQUENCE.   
**spring.jpa.open-in-view**: The default value is **true**. It binds a JPA EntityManager to the thread for the entire processing of the request.   
**spring.jpa.properties.\***: It sets additional native properties to set on the JPA provider.   
**spring.jpa.show-sql**: It enables logging of SQL statements. Default value is **false**.

* Define JPA Entity: MemberEntity (similar to MemberModel)

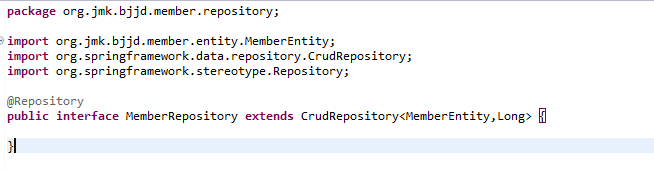


* Define custom sequence generator with string prefix:



* Create member\_seq in mysql database and no need to create member as we have defined spring.jpa.hibernate.ddl-auto=create in application.properties file but after one execution we will change this value from create to update

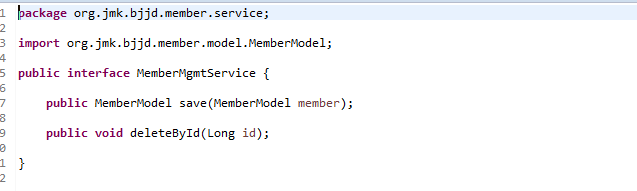
## Create the repository



This is the repository interface, this will be automatically implemented by Spring in a bean with the same name with changing case The bean name will be userRepository.

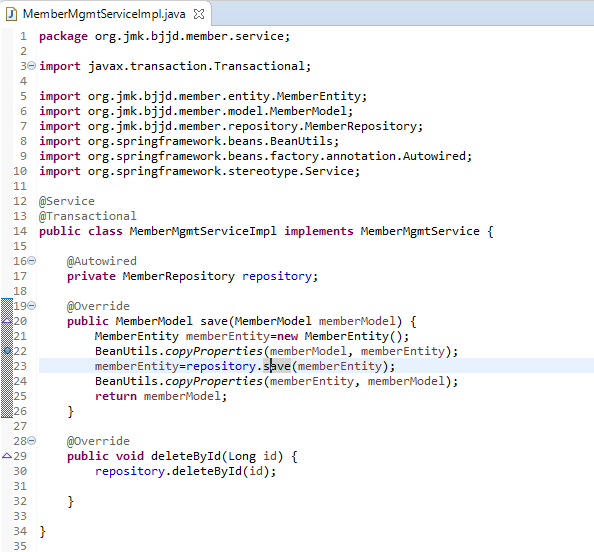
## Define Service Classes for Interacting With Data Access Objects

Pay attention to member management service interface, which defines the service API.



Pay attention to some of the following:

* [@Service annotation](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Service.html) is used to indicate a service implementation class or a business service facade. This annotation serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.
* [@Transactional annotation](https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/transaction.html#transaction-declarative-annotations) to declare transaction configuration:



* See the MemberApi Class create by swagger.

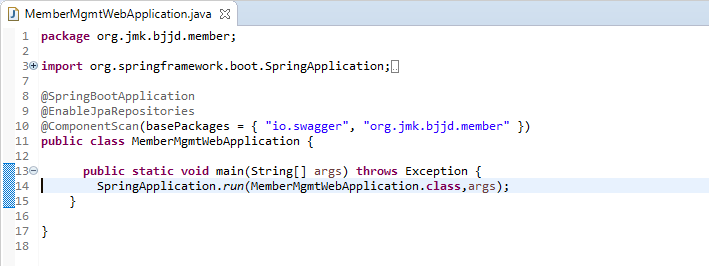


* Update the MemberApiControllerClass which implements MemberApi so that it can invoke appropriate service methods



## Starting the Application

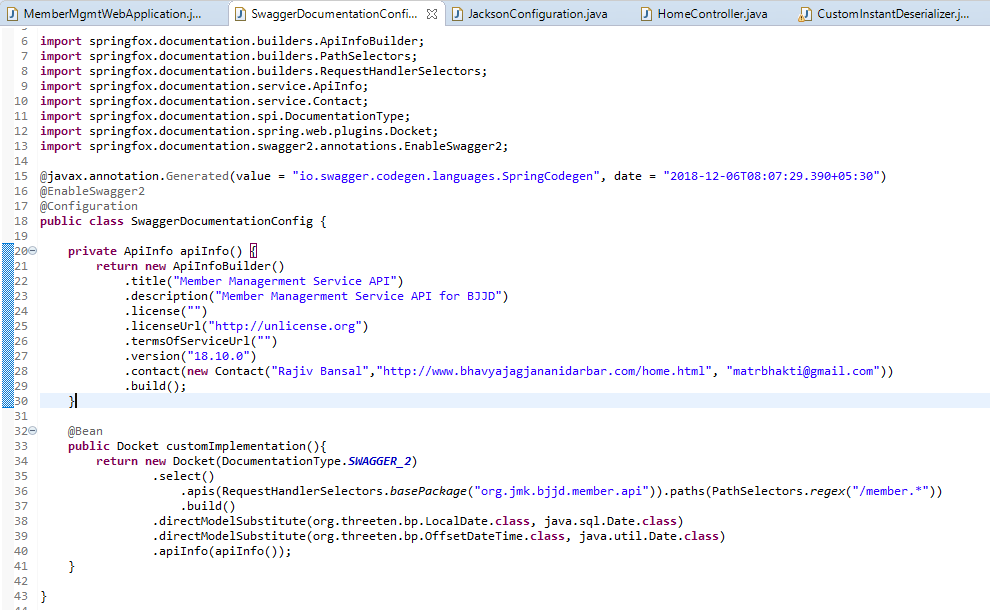
Swagger2SpringBoot class is automatically created when we had generated the Spring Boot project using Swagger API. Now need to make few changes in this class. Now we will rename this class and move to the main package where all the application classes exist and refactored the code as shown below:



## 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.

SwaggerDocumentationConfig is automatically created when we had generated the Spring Boot stub using swagger api. Now need to make few changes in this class.

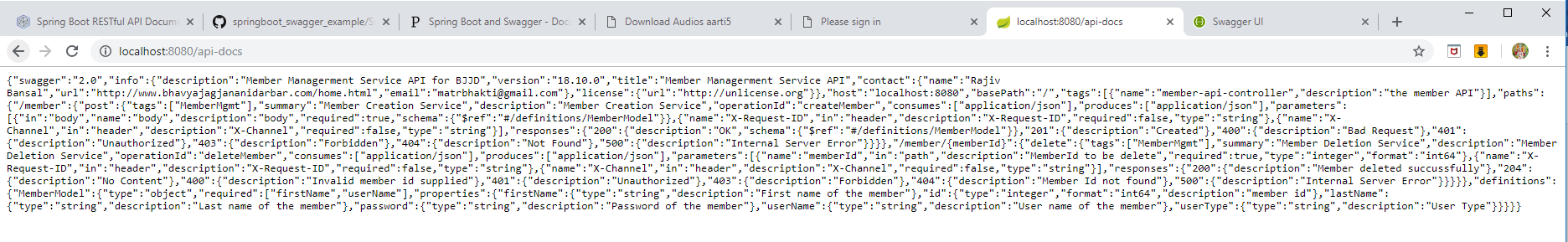


In this configuration class, the @EnableSwagger2 annotation enables Swagger support in the class. The select() method called on the Docket bean instance returns an ApiSelectorBuilder, which provides the apis() and paths() methods that are used to filter the controllers and methods that are being documented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

In the code, the RequestHandlerSelectors.basePackage predicate matches the org.jmk.bjjd.member.api 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 /member.

* @Configuration - This file contains Spring configuration.
* @EnableSwagger2 - Annotation to Enable Swagger Documentation on the API
* public static final Contact DEFAULT\_CONTACT - Has the contact information of the API. This will be exposed as part of the Swagger Documentation.
* public static final ApiInfo DEFAULT\_API\_INFO - Meta information about the API - Description, Licensing etc. This will be exposed as part of the Swagger Documentation.
* private static final Set<String> DEFAULT\_PRODUCES\_AND\_CONSUMES - What content types does your API support?
* public Docket api() { - Docket to decide what kind of APIs you would want to document. In this example, we are documenting all APIs. You can filter out APIs you do not want to document with Swagger.

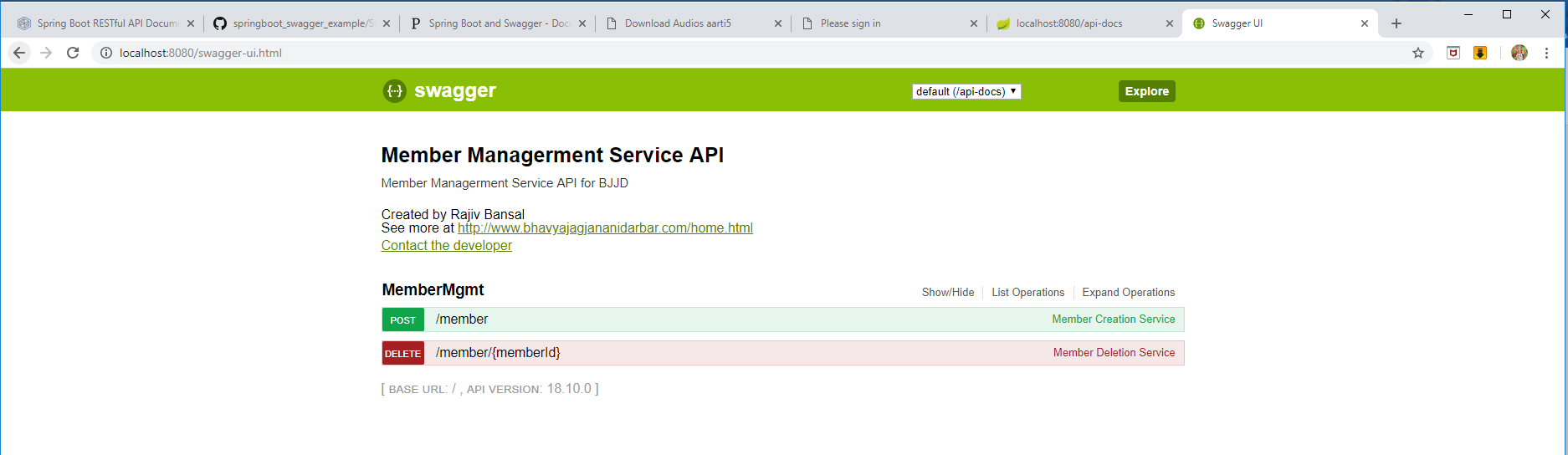
At this point, you should be able to test the configuration by starting the app and pointing your browser to <http://localhost:8080/api-docs> .



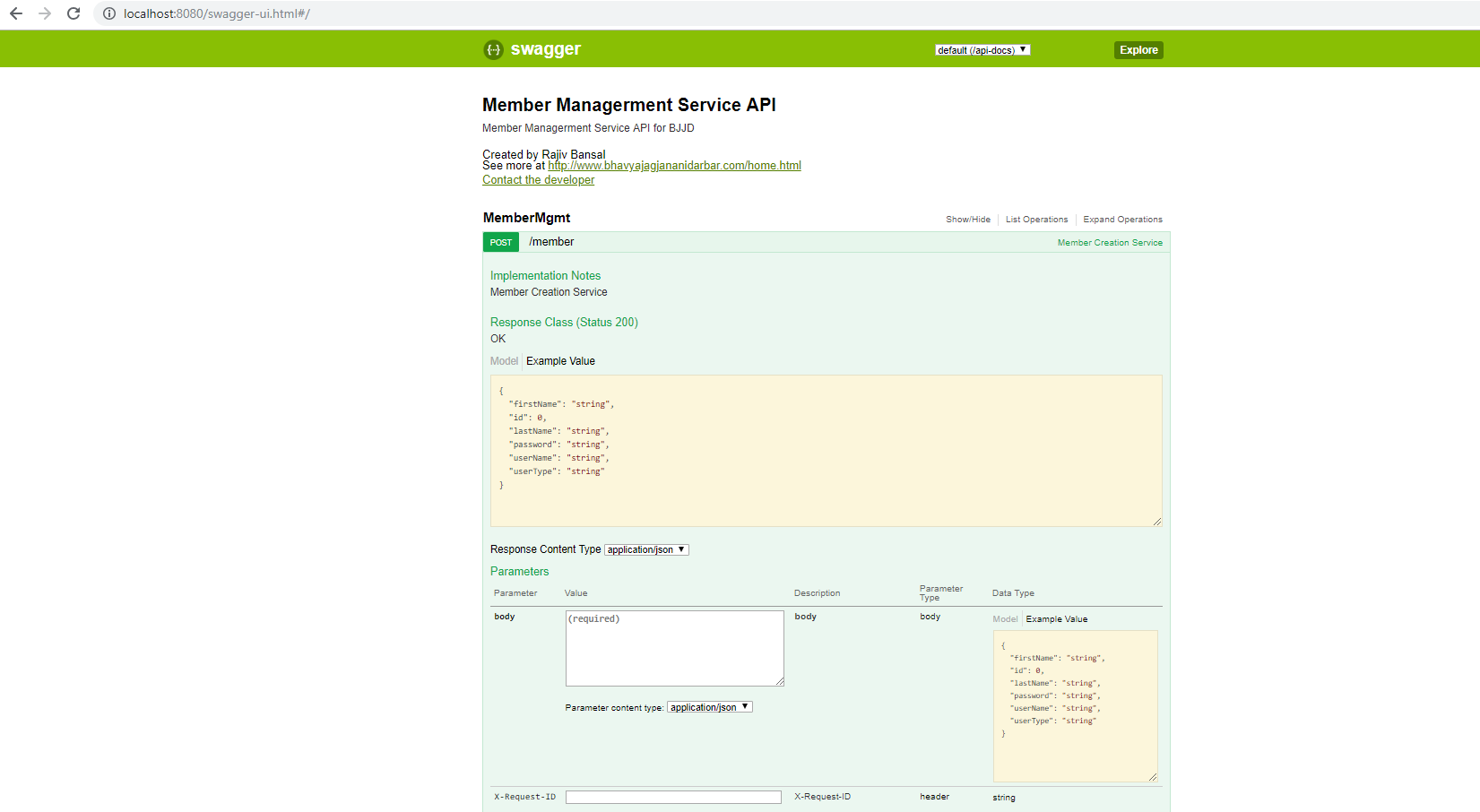
Obviously, the above JSON dump that Swagger 2 generates for our endpoints is not something we want.

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

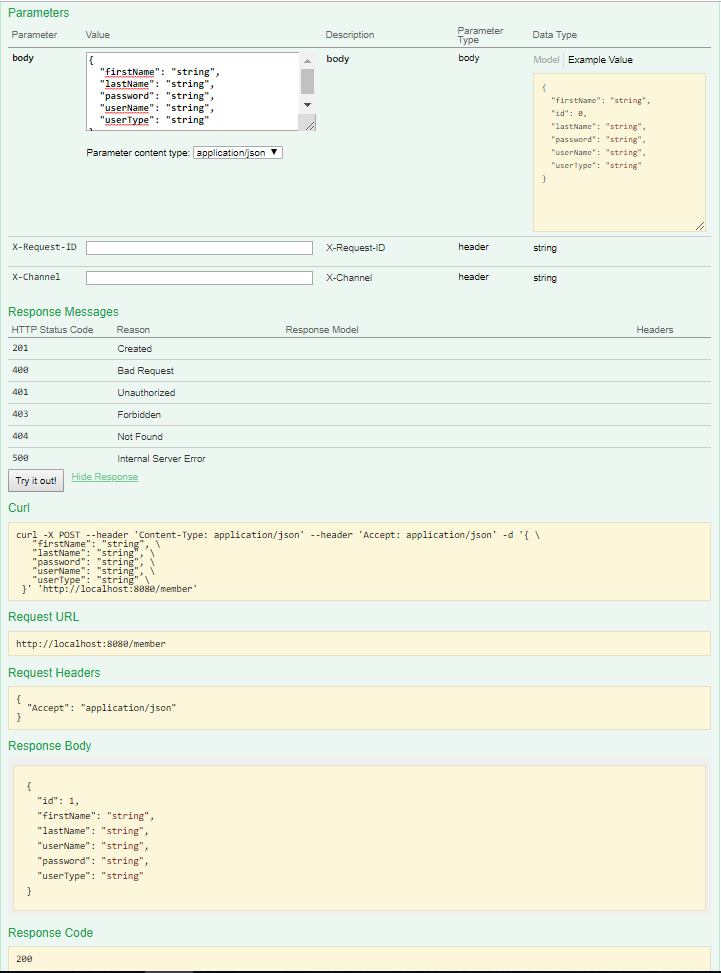
On pointing your browser to [*http://localhost:8080/swagger-ui.html*](http://localhost:8080/swagger-ui.html), you will see the generated documentation rendered by Swagger UI, like this:



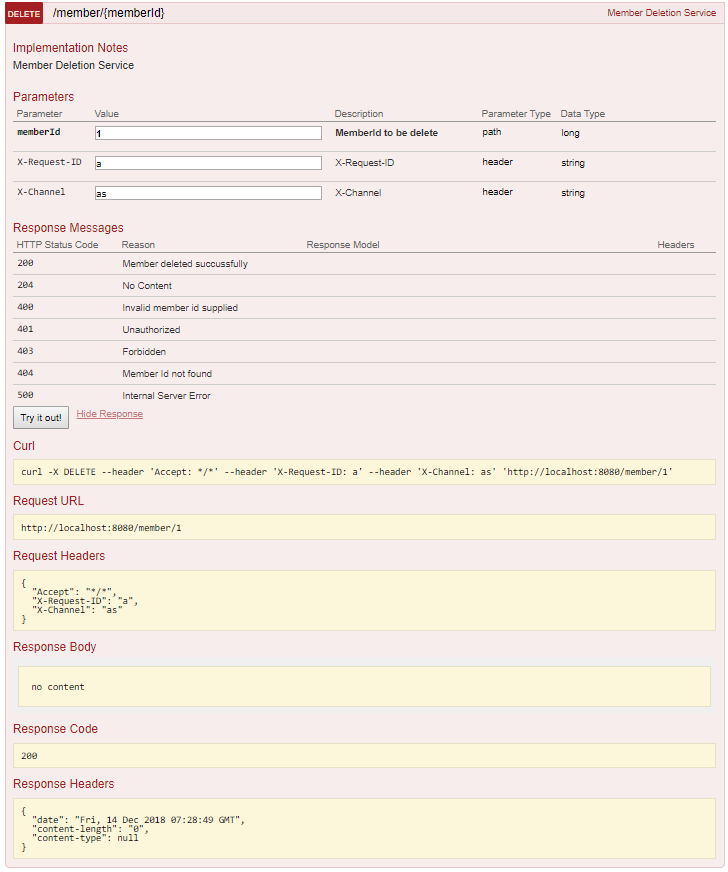
If you will click on Expand Operations



* Testing the application using Swagger UI:
  + Open browser and enter the URL: [*http://localhost:8080/swagger-ui.html*](http://localhost:8080/swagger-ui.html)
  + Expand the list of operations.
    - Add the new member by entering the following details:
      * Body
      * X-channel id
      * X-request id

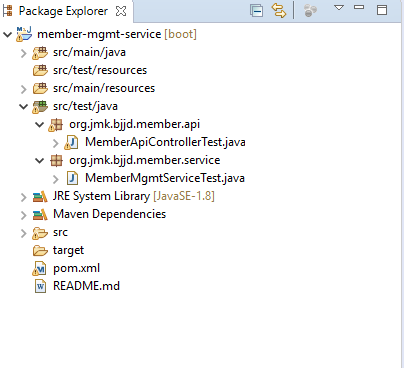


* + - Delete the member:
      * Provide the memberId
      * X-Channel Id
      * X-request id

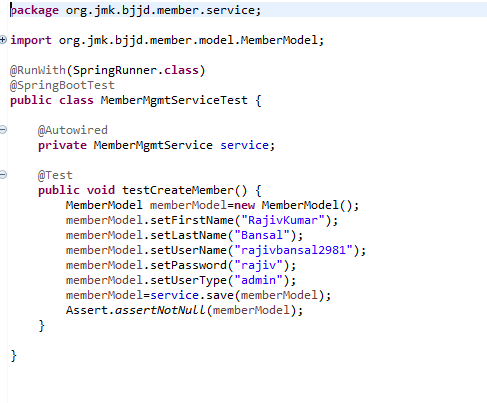


* Testing with Spring(<artifactId>spring-boot-starter-test</artifactId>)

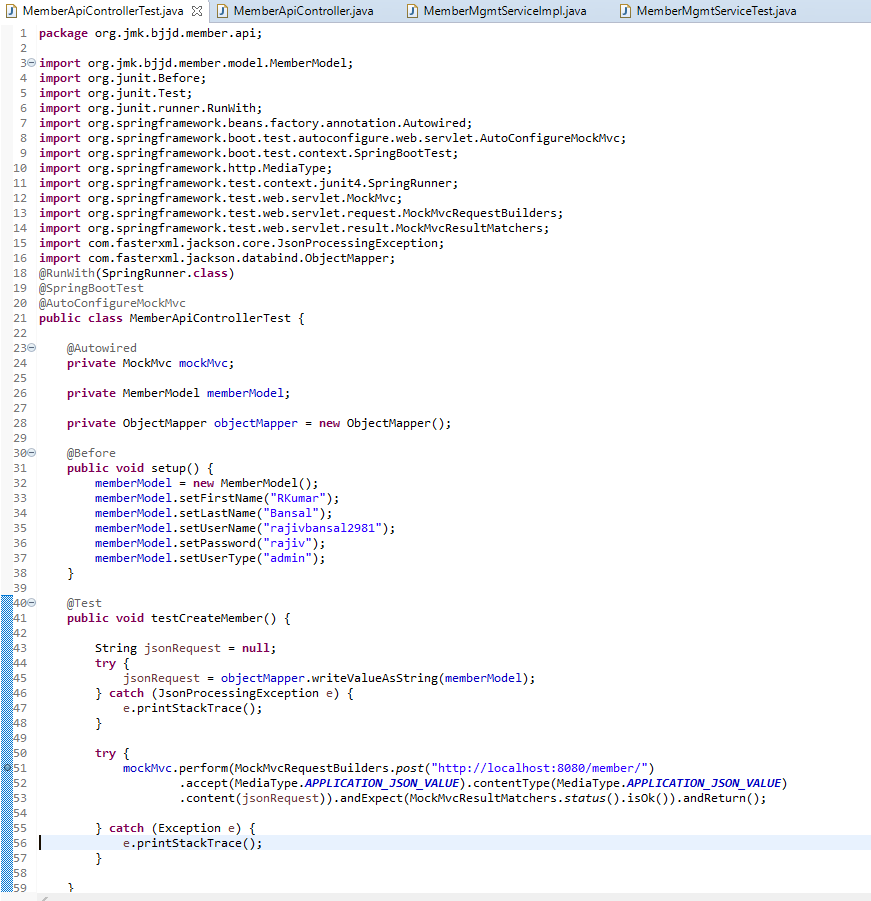
Create test structure in the project as mentioned in the below screen:



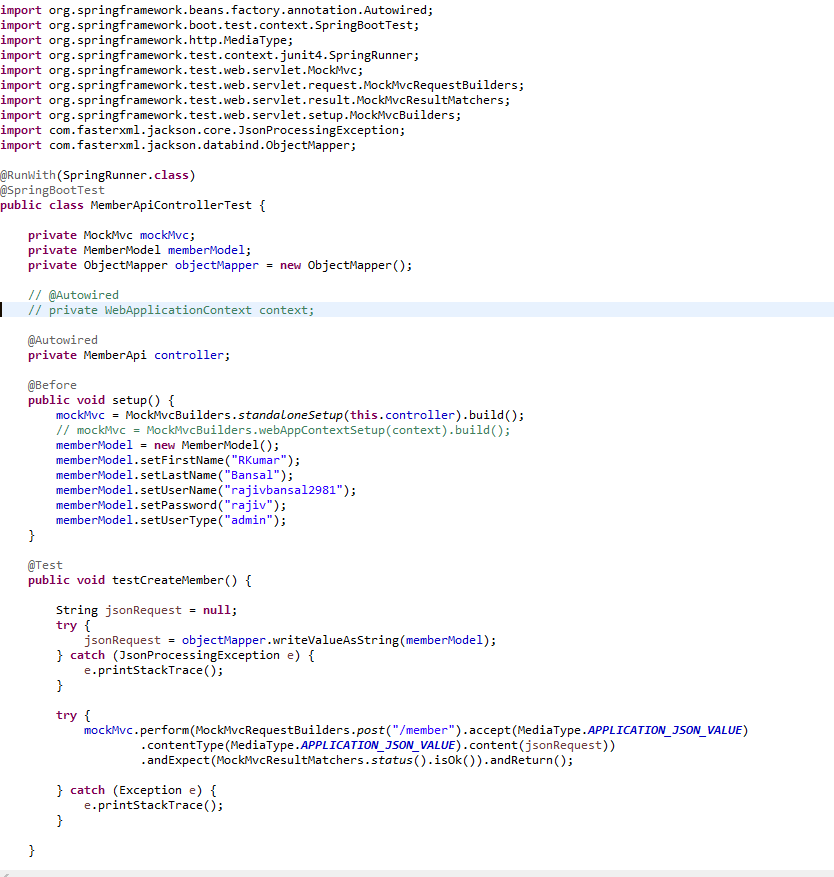
Service Integration Test



Controller Integration Test



* Without @AutoConfigurMockMvc



Finally, Member Management Service Application is Microservice which is complete because microservice focuses on single responsibility which we have already developed. So now we will put this application on Spring Cloud.