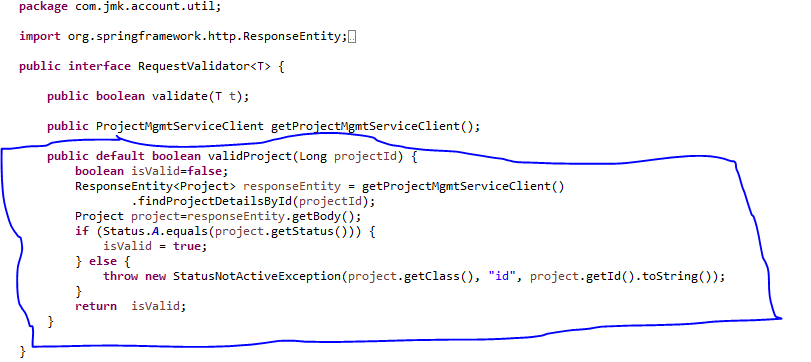
* Custom Exception Handling where
  + **EntityNotFoundException :** While interacting with people-mgmt-service and project-mgmt-service to validate the entity using feign client, EntityNotFoundException class (created in common-service) can be thrown if entity not present in database which is handled in RestExceptionHandler(created in common-service), But we are not able to catch this EntityNotFoundException exception from caller microservice as it throws FeignException so to catch the appropriate exception, need to handle this in FeignExceptionHandler class (created in common-service) we are creating the appropriate error message in json and send back to client.
  + **StatusNotFoundException:** This exception can be thrown when entity is present but its status is not active and it is handled in

RestExceptionHandler(created in common-service).

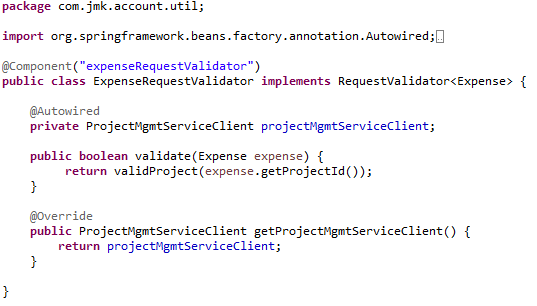
* Create the **RequestValidator** to validate the donation and expense request where we will validate the donor and project by invoking the people-mgmt-service and project-mgmt-service using feign client.



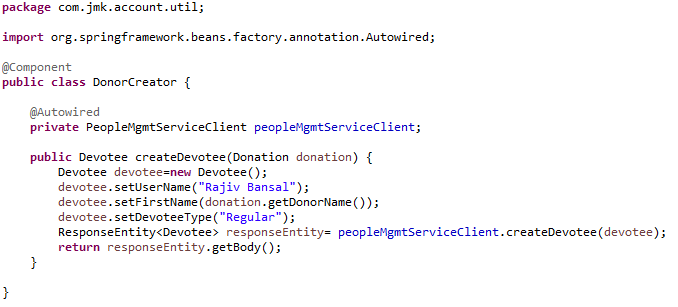
DonationRequestValidator implements RequestValidator where we are validating the donation request:



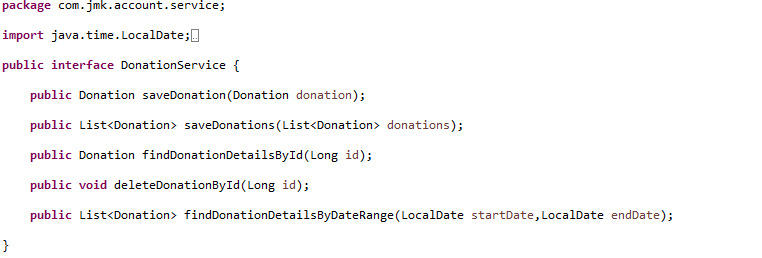
ExpenseRequestValidator implements RequestValidator where we are validating the donation request:



* Create DonorCreator class to create the donor by invoking the people-mgmt-service using feign client.



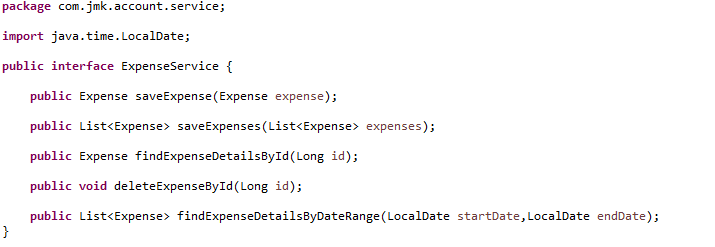
* DonationService and DonationServiceImpl Class should be exposed as following:
* DonationService interface



* DonationServiceImpl interface



* ExpenseService and ExpenseServiceImpl Class should be exposed as following:
* ExpenseService



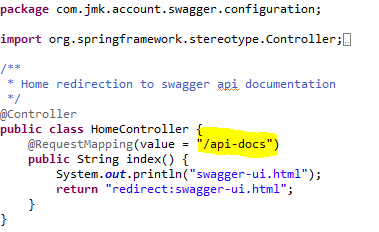
* ExpenseServiceImpl



* Made changes in the SwaggerDocumentationConfig



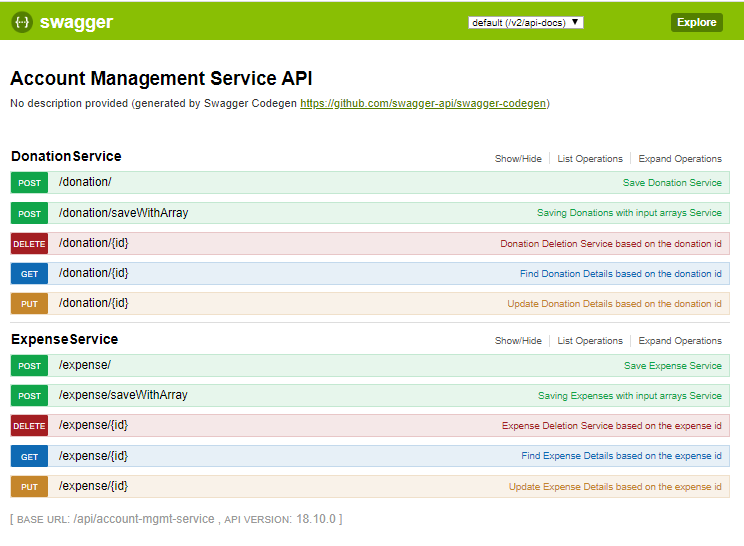
* Made changes in the Swagger’s HomeController



* After running the application, should be visible following functions for the following url: <http://localhost:7379/api/account-mgmt-service/swagger-ui.html>

Or

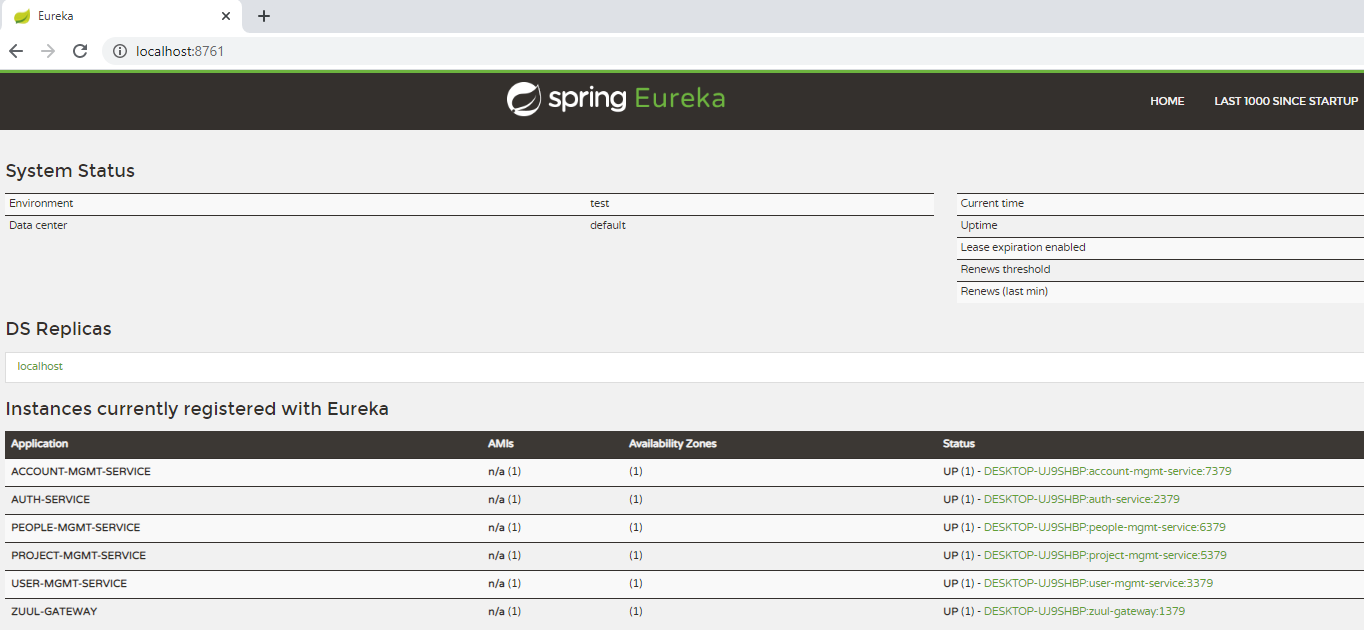
<http://localhost:7379/api/account-mgmt-service/api-docs>



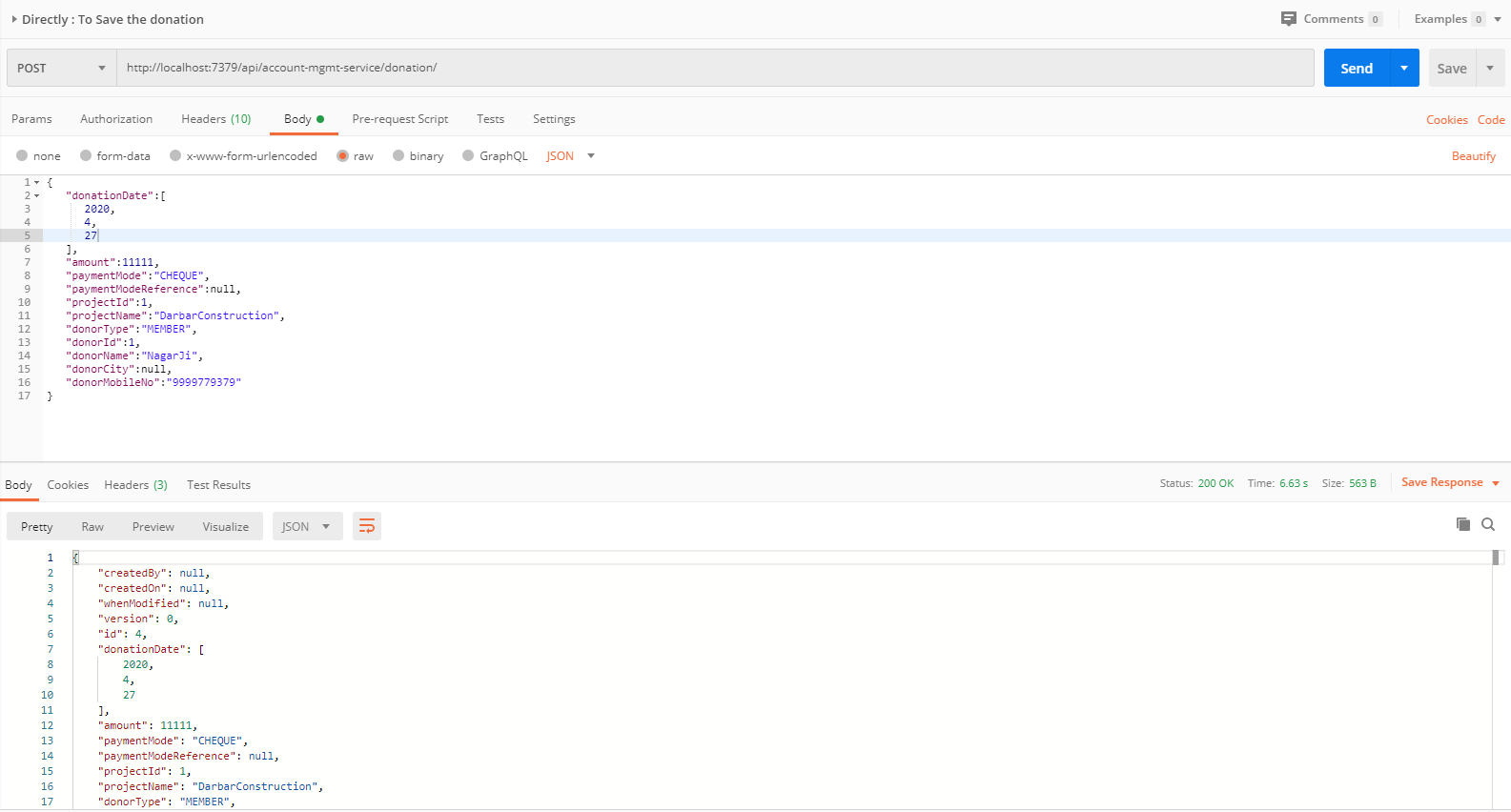
* Test the DonationApiTest using Junit



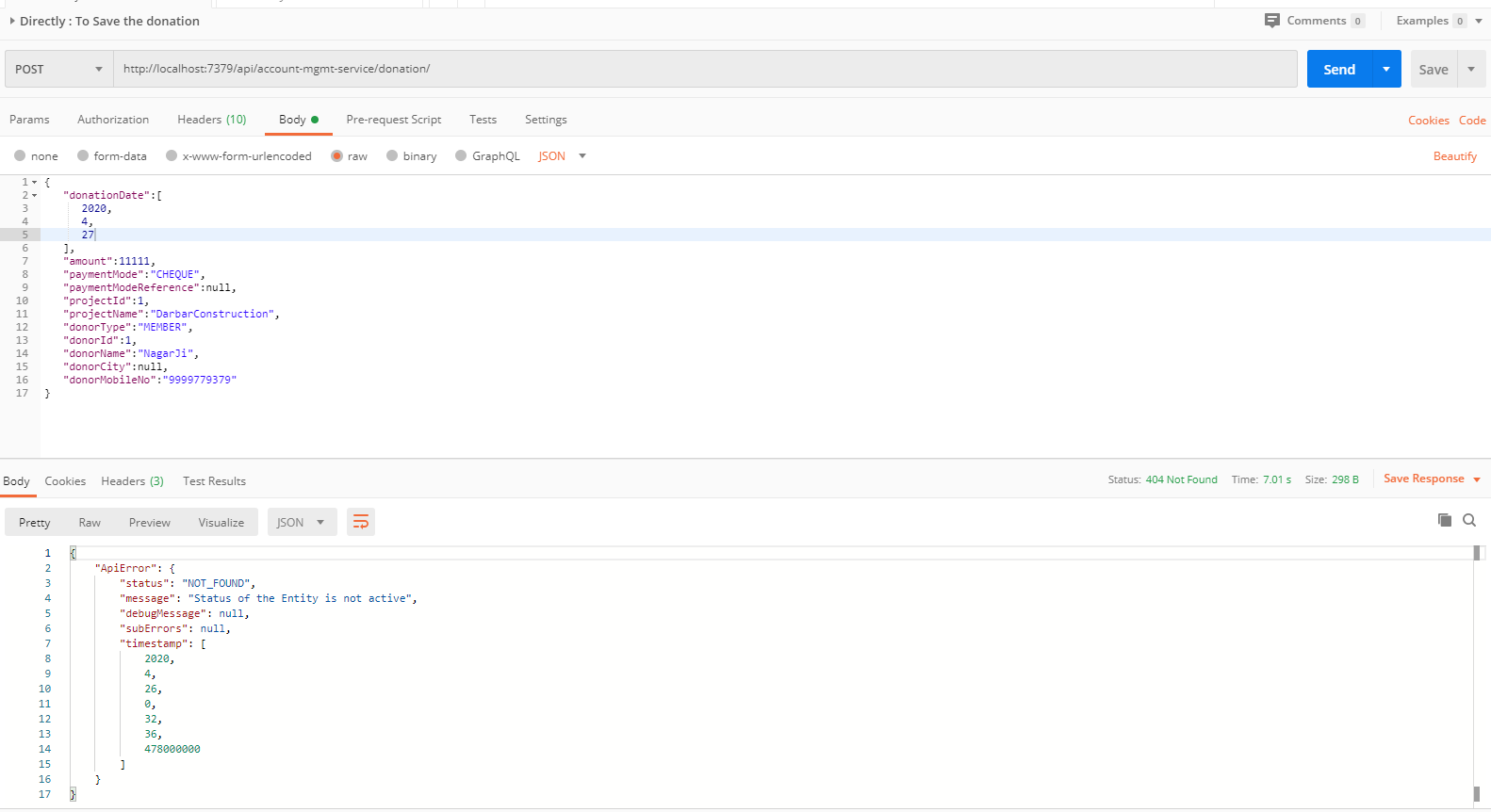
* Run the application as **Spring Boot App** in the following order
  + EurekaServerApplication
  + UserMgmtRestApplication
  + AuthServiceApplication
  + ZuulGatewayApplication
  + PeopleMgmtRestApplication
  + ProjectMgmtRestApplication
  + AccountMgmtRestApplication
* Open the browser of Eureka Server at localhost:8761, you should see the all the microservices running and enter the url: <http://localhost:8761/>



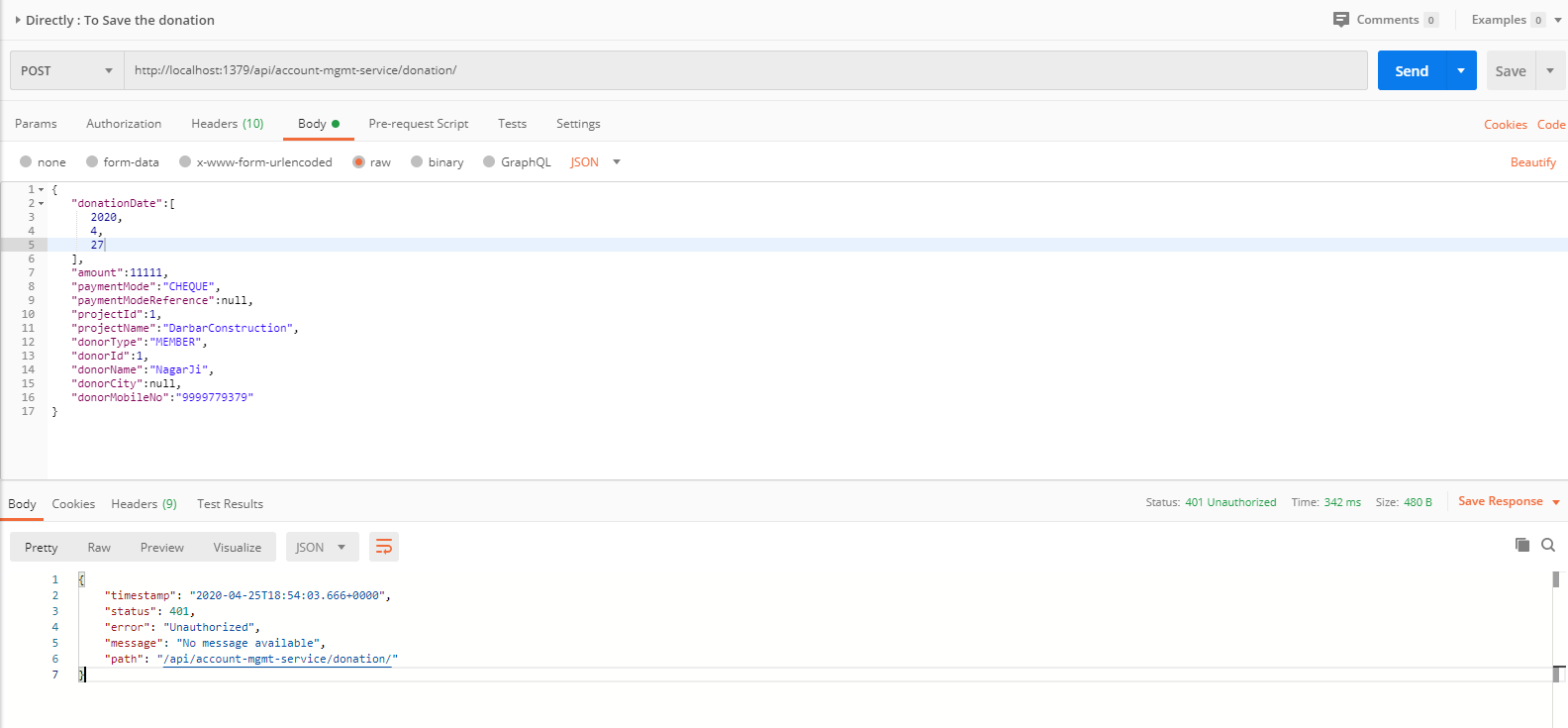
* Testing
  + Without authentication means directly hitting the account-mgmt-project running on 7379 and save the donation details



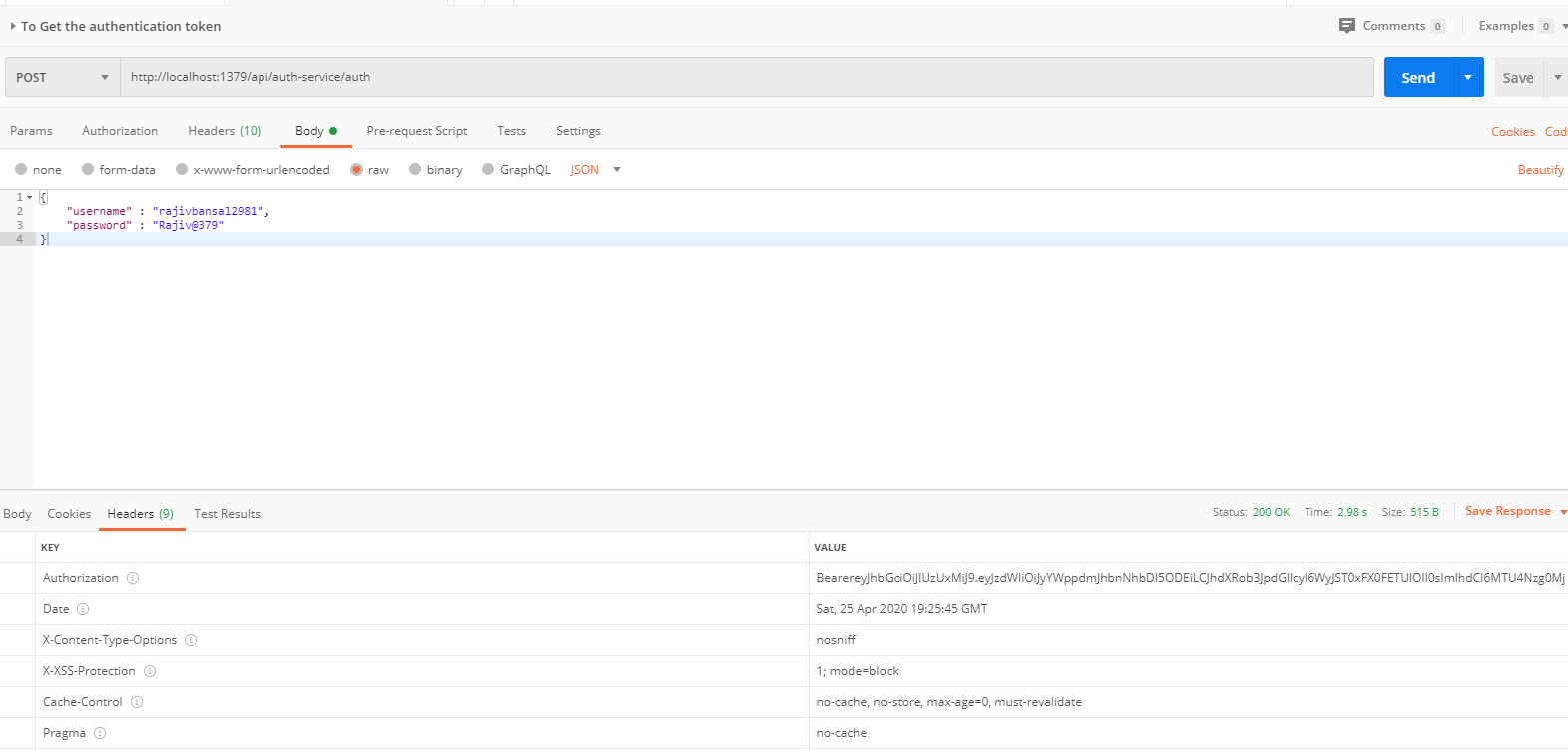
* + Without authentication means directly hitting the account-mgmt-project running on 7379 but member does not exist for whom we are saving the donation is not active.



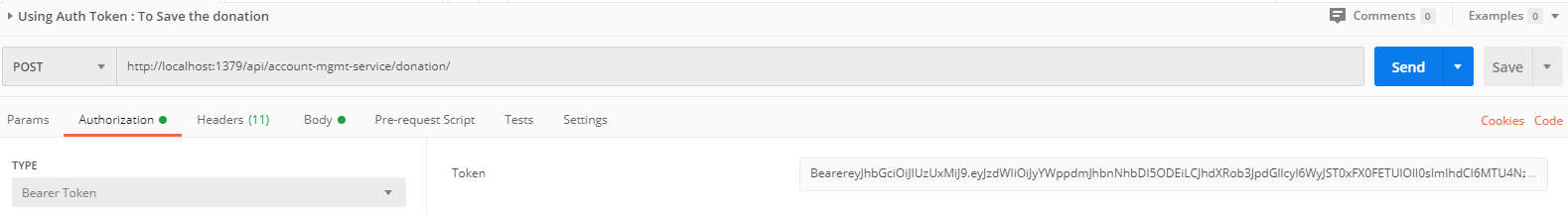
* + With authentication means every request to people-mgmt-service microservice will be hit via gateway running 1379
    - with authentication means every request to people-mgmt-service microservice will be hit via gateway running on 1379
      * getting member details based on the id but without token means no authorization code (Bearer token) in the header will give the error.



* Saving donation with token means authorization code (Bearer token) in the header will be provided using authentication service
  + 1. **To get authentication code in response header using authentication service**



* + 1. **Now hit the account-mgmt-service to** save the donation by providing the token means authorization code (Bearer token) in the header received in above step
  + See the **Authorization** tab in the donation request



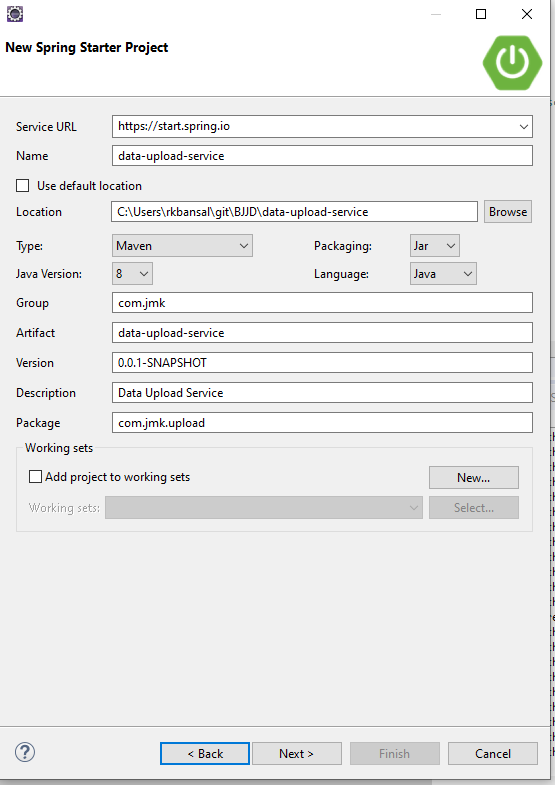
* + See the **Body** tab in the donation request and response can be seen.



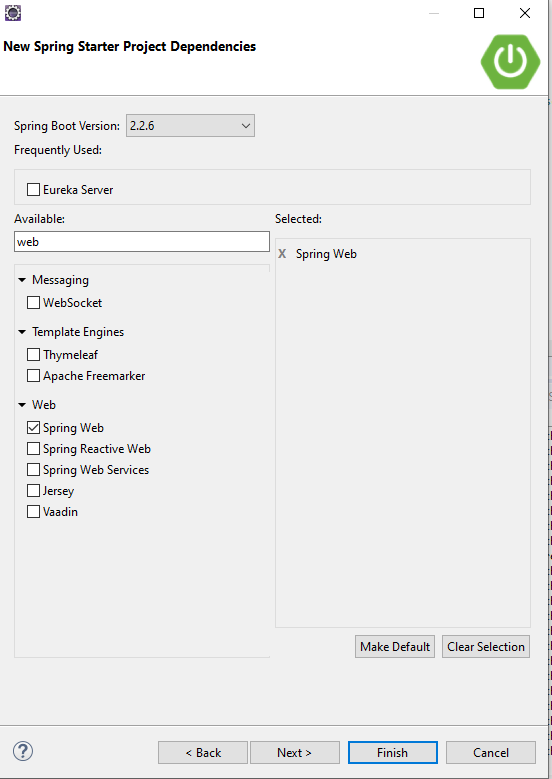
## **Data-Upload-Service**

This service will upload the data in bulk for Donations, Expenses, Darshan, Member, Devotee etc. in different format and call the appropriate microservice to save the data.

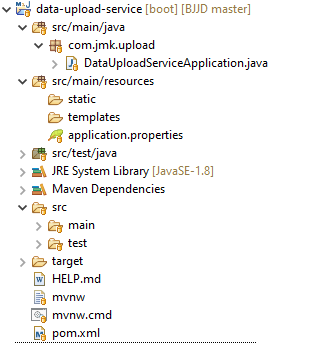
1. Create the Project using spring Starter Project.



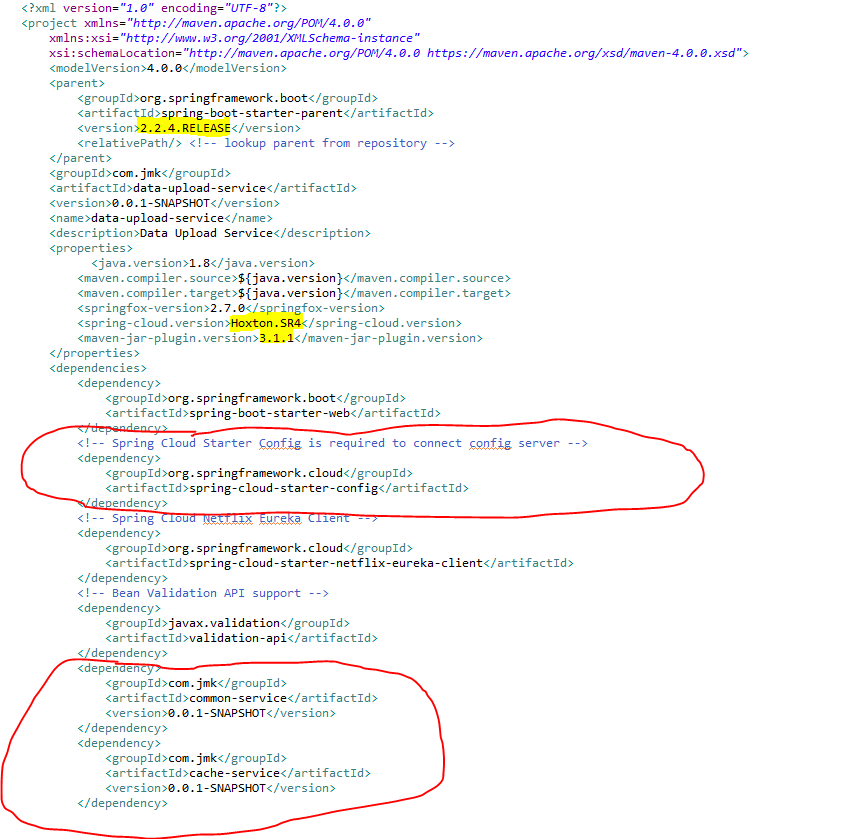
1. Click on Next and select the Spring Web Module.



1. Click on Finish and import the project.

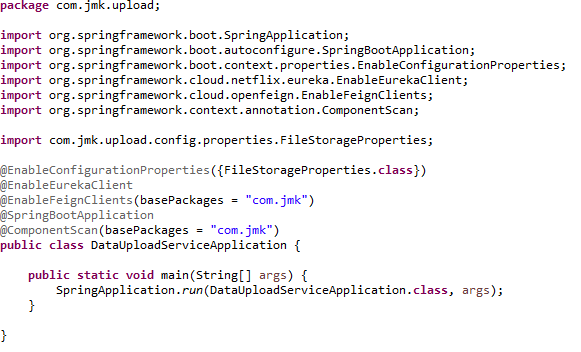


* Add the following dependencies in pom.xml with the following considerations:
  + Spring Cloud Starter Config: This dependency is required to connect to the Config Server
  + Spring Boot Starter Parent: update the version: 2.2.4.RELEASE highlighted in yellow colour.
  + Spring Cloud version: upgraded to Hoston.SR4. It is highlighted in yellow colour.
  + Excel File Processing API: used org.apache.poi API.
  + Cache-service and common-service: used internally created API: cache-service(to be discuss in next section) to cache the reference entities and common-service to use the model and utility classes.
  + Maven Jar Plugin Version: After the above updates, pom is giving compiler error in pom.xml so downgraded version of maven jar plugin to 3.1.1. It is highlighted in yellow colour.





1. **Update the main DataUploadServiceApplication application with the following details:**
2. Enable Eureka Client so that it can register with Eureka Server
3. Enable FeignClient to interact with people-mgmt-service, account-mgmt-service and project-mgmt-service
4. Enable JpaRepositories to store the data upload history.
5. To enable the ConfigurationProperties feature, you need to add [@EnableConfigurationProperties](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/EnableConfigurationProperties.html) annotation to any configuration class.



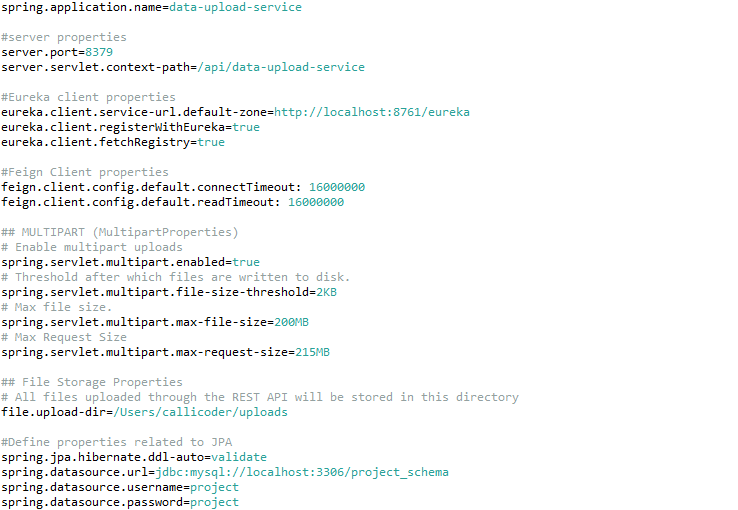
1. Configuring Server and File Storage Properties

**First thing First!**

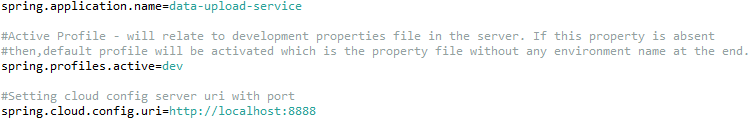
* + Let’s configure our Spring Boot application to enable Multipart file uploads, and define the maximum file size that can be uploaded. We’ll also configure the directory into which all the uploaded files will be stored.
  + Add the eureka client properties so that it can register with Eureka Server
  + Enable the feign client properties so that we can communicate with different microservices.

Open src/main/resources/application.properties file, and add the following properties to it.

* Moved the following main application.properties along with different profiles to git repository: **bjjd-config-server-git-repo**



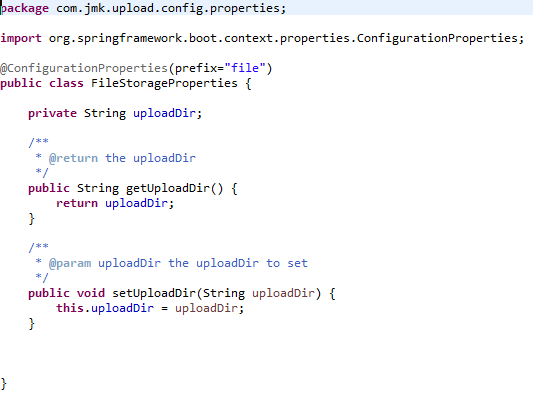
* In this application, still there will be an application.properties file but it will have the following minimum details:
  + Name of the application
  + Active Default Profile
  + Cloud Config server URI with port



1. **Automatically binding properties to a POJO class**

Spring Boot has an awesome feature called [@ConfigurationProperties](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html" \l "boot-features-external-config-typesafe-configuration-properties) using which you can automatically bind the properties defined in the application.properties file to a POJO class.

Let’s define a POJO class called FileStorageProperties to bind all the file storage properties.



The @ConfigurationProperties(prefix = "file") annotation does its job on application startup and binds all the properties with prefix file to the corresponding fields of the POJO class.

If you define additional file properties in future, you may simply add a corresponding field in the above class, and spring boot will automatically bind the field with the property value.

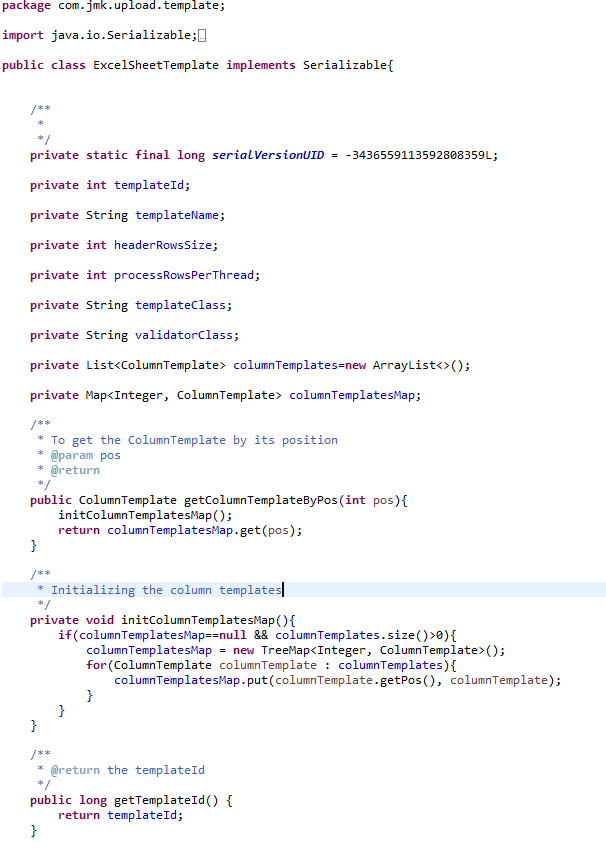
1. Excel File Template(json format) to upload the excel file data in proper format so that it can be validated and processed. Basic format should be like this: excelSheetTemplate.json



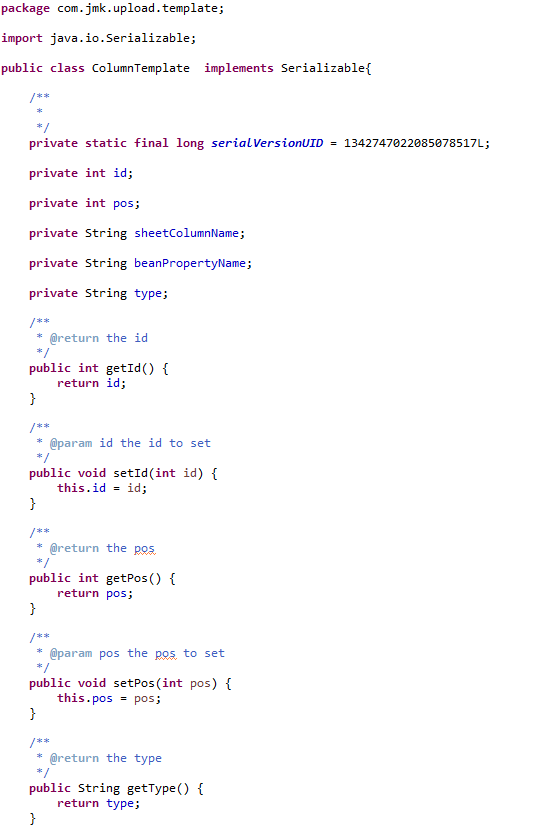


1. To handle the data coming in excelFileTemplate we will have two classes:
   * ExcelSheetTemplate.java
   * ColumnTemplate.java

* ExcelSheetTemplate.java



* ColumnTemplate.java



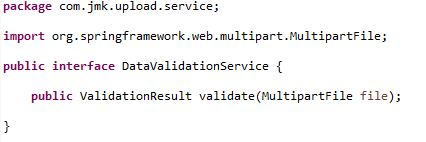
1. Read Excel Sheet Template using class: ExcelSheetTemplateReader.java



1. Write REST FileUploadController to upload the single or multiple excel files from UI (Browser or Application).



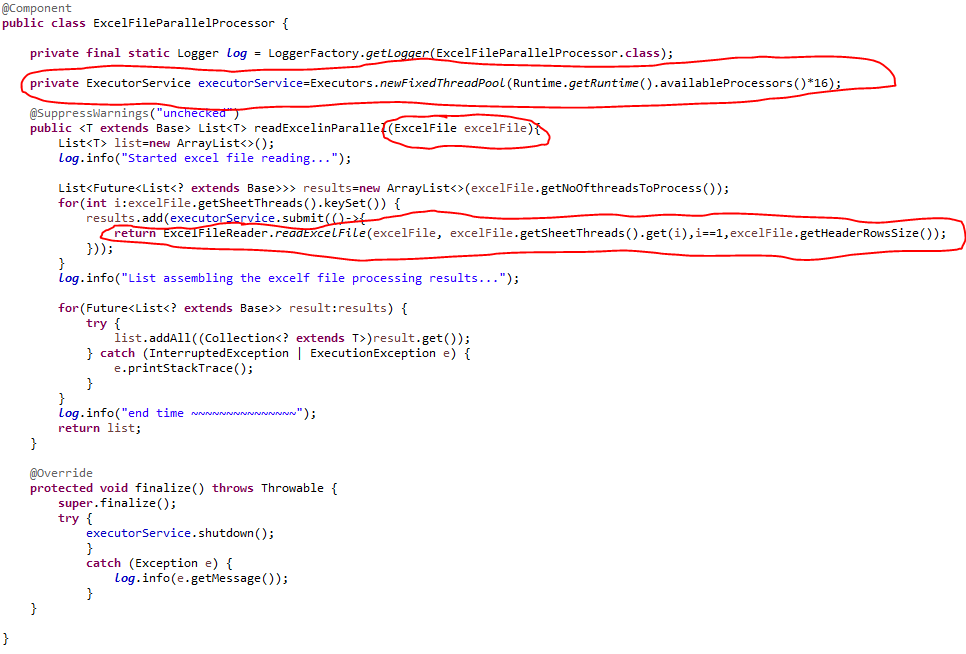
* DataValidationService Interface and its implementation class DataValidationServiceImpl to validate the uploaded excel file.
  + DataValidationService.java



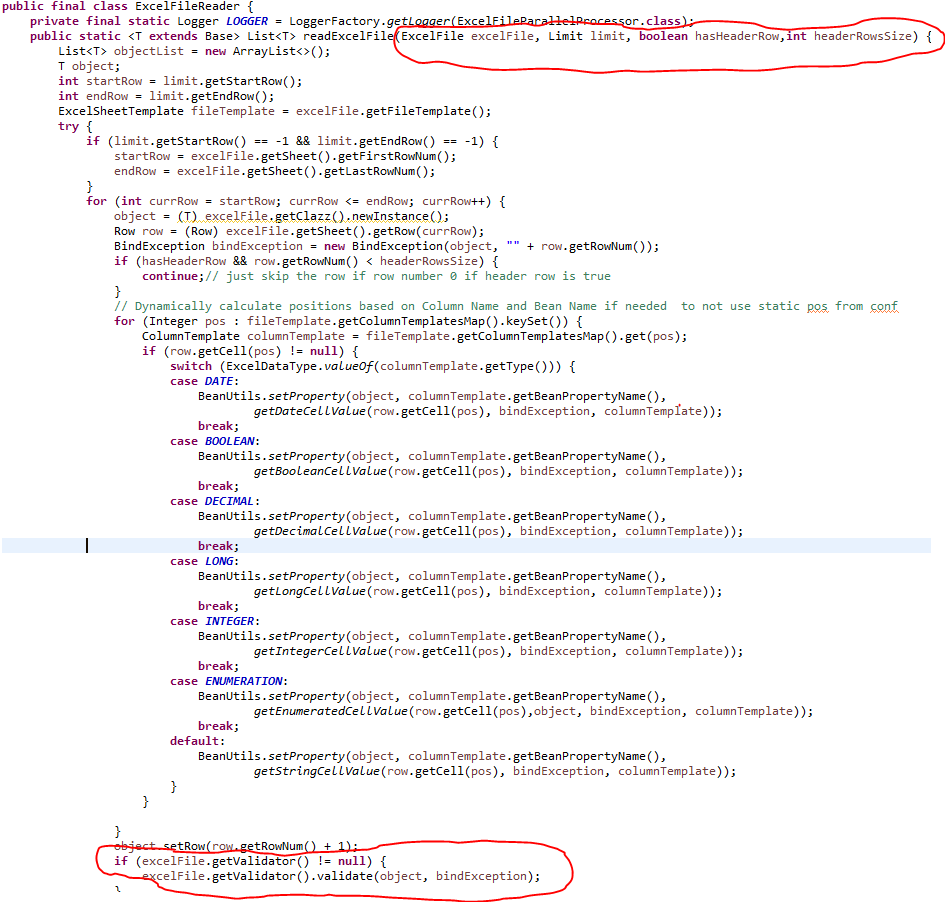
* DataValidationServiceImpl.java



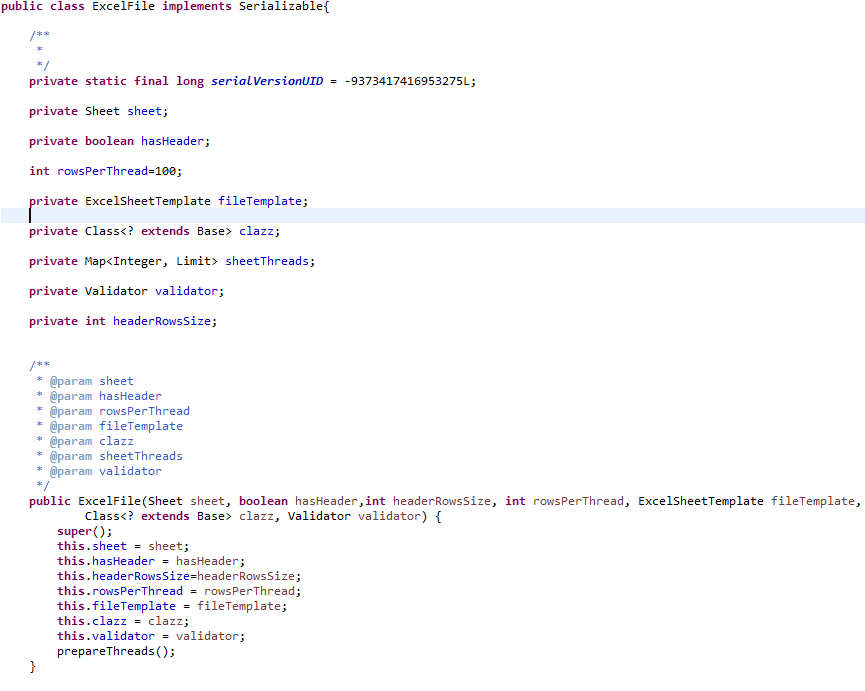
* ExcelFileParallelProcessor to process the excel file by reading the excelfile using ExcelFileReader in parallel using multithreading: ExecutorService



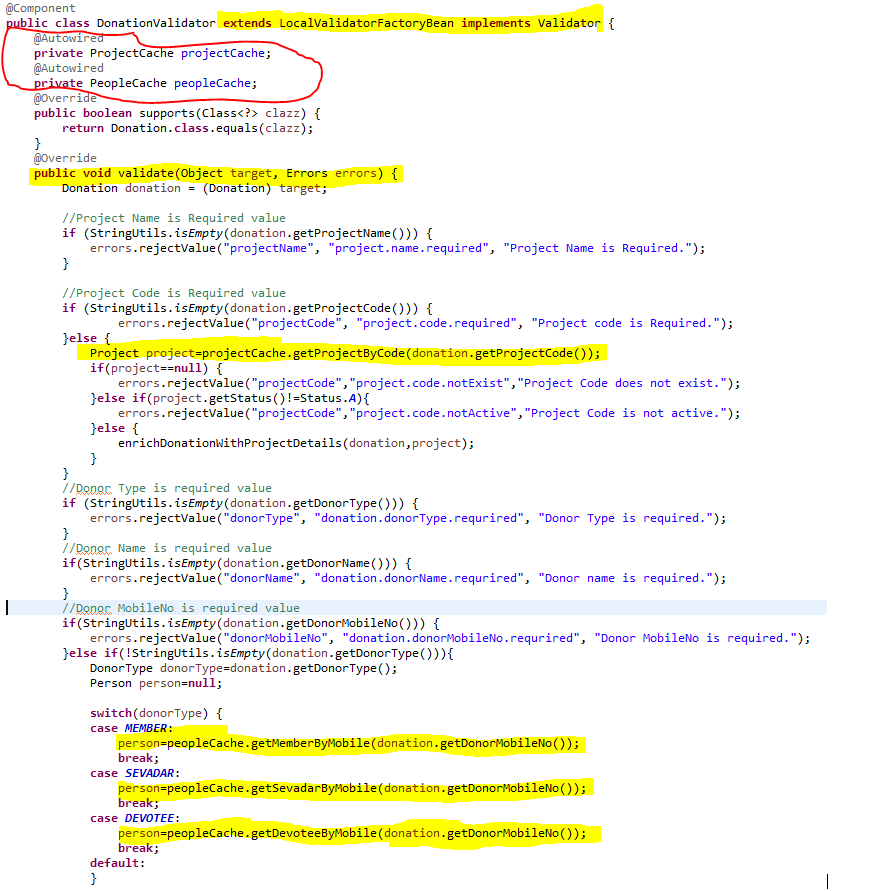
* ExcelFileReader to read the excel file



* ExcelFileReader will use ExcelFile object to read the object: ExcelFile.java



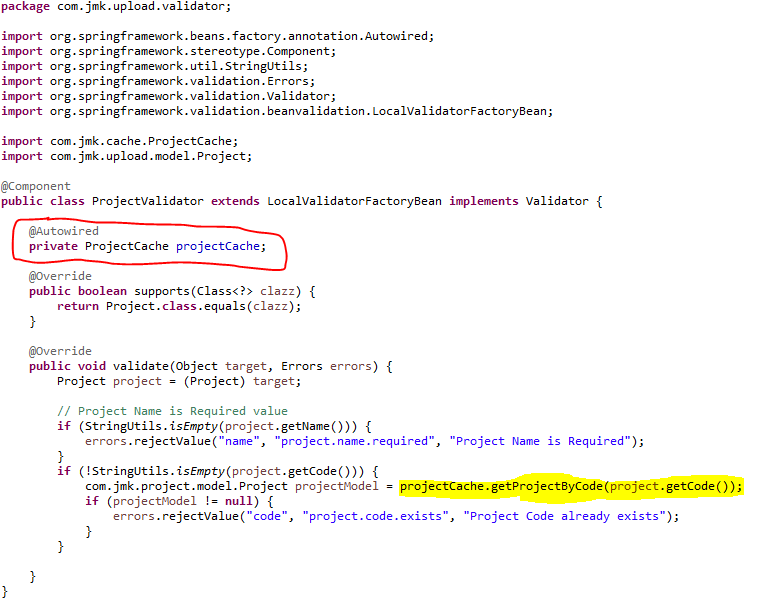
* ExcelFileReader will use the following validators and many more in future to validate the data after reading the excel file.
  + DonationValidator
  + ExpenseValidator
  + ProjectValidator
    - DonationValidator



* + - ExpenseValidator



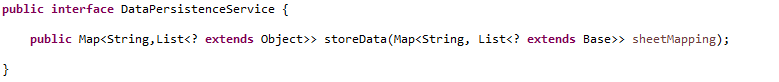
* + - ProjectValidator



* To store the Validation result, used the ValidationResult class



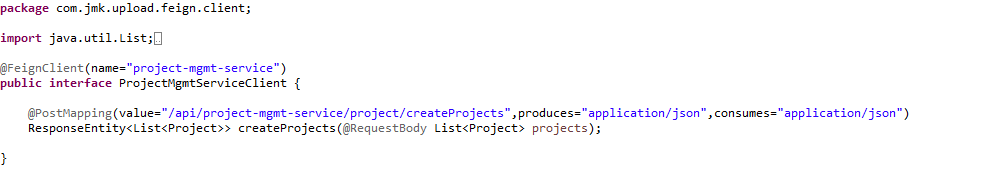
* **DataPersistenceService: Save the excel file data after successfully validating the excel file**
* DataPersistenceService



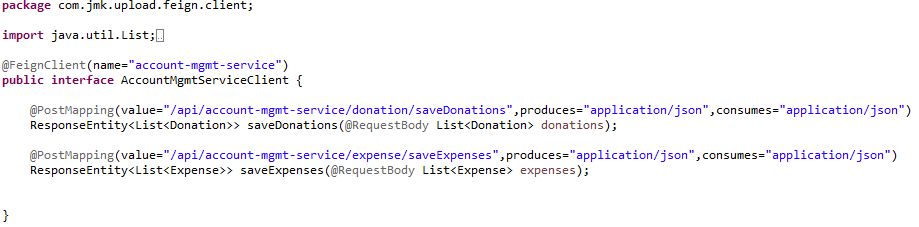
* DataPersistenceServiceImpl



* **FeignClient to interact with other microservices to save the data.**
* ProjectMgmtServiceClient
* AccountMgmtServiceClient
  + ProjectMgmtServiceClient



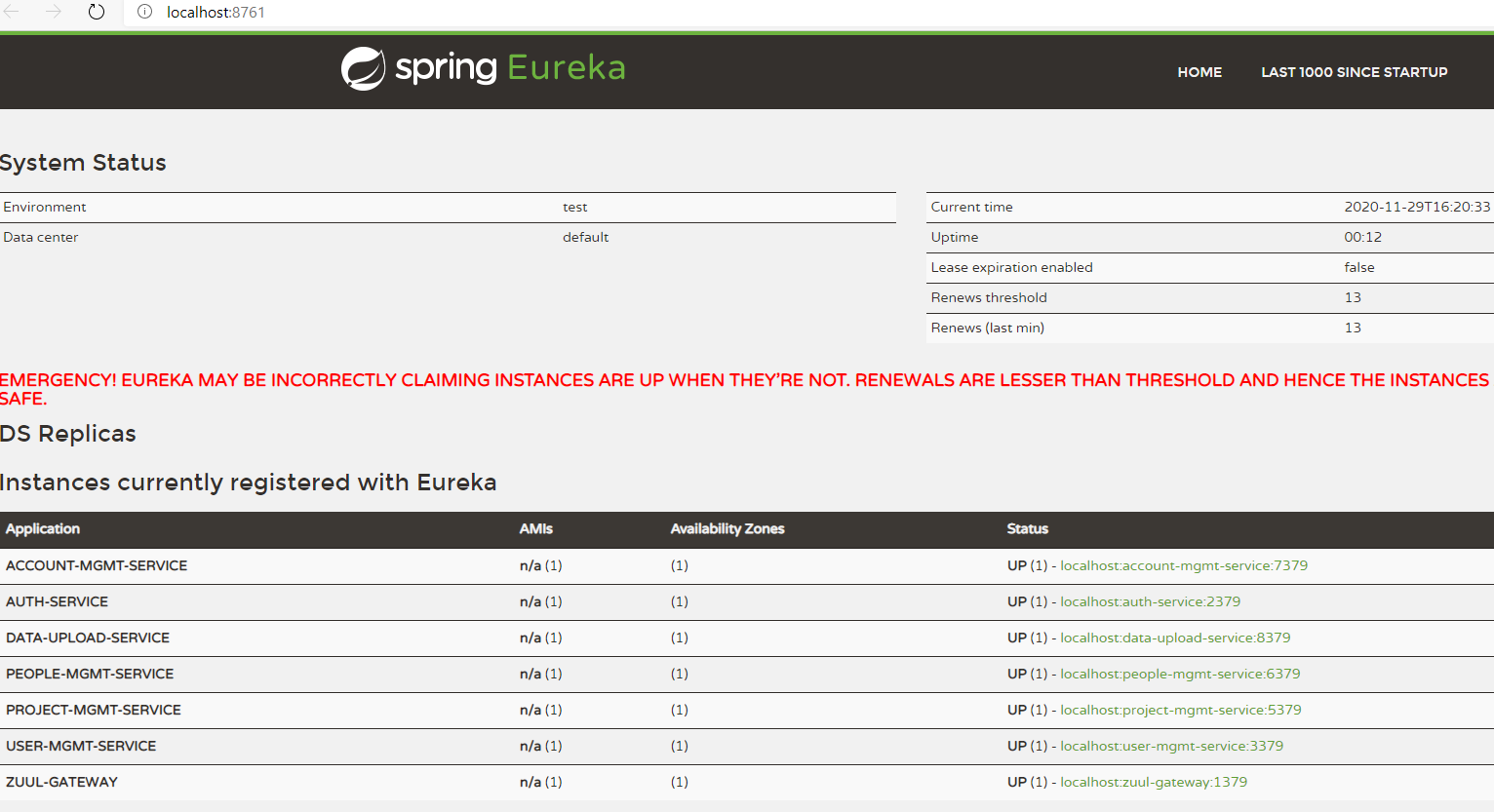
* + AccountMgmtServiceClient



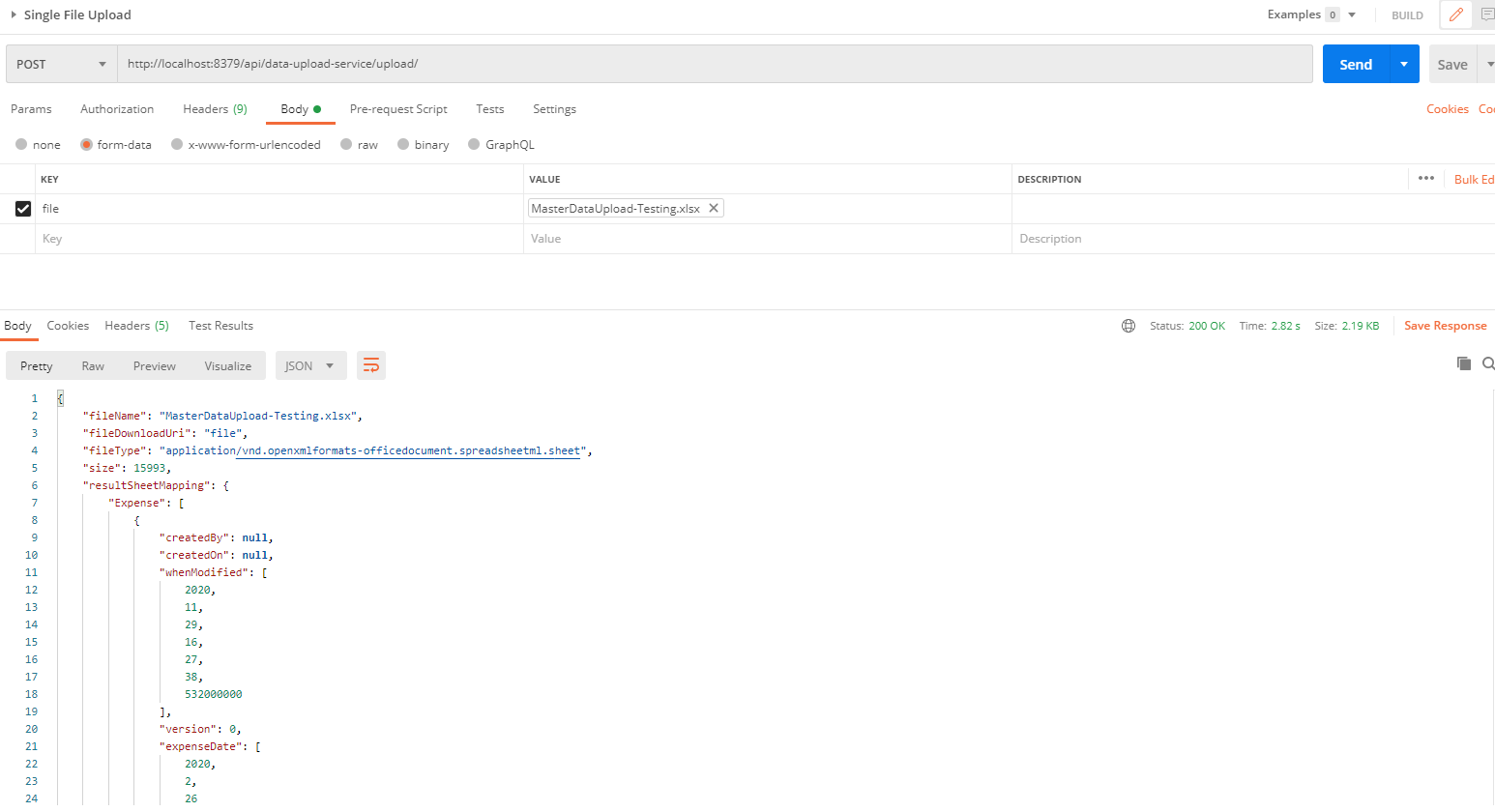
* + **To send the result to the client, use UploadFileResponse class**



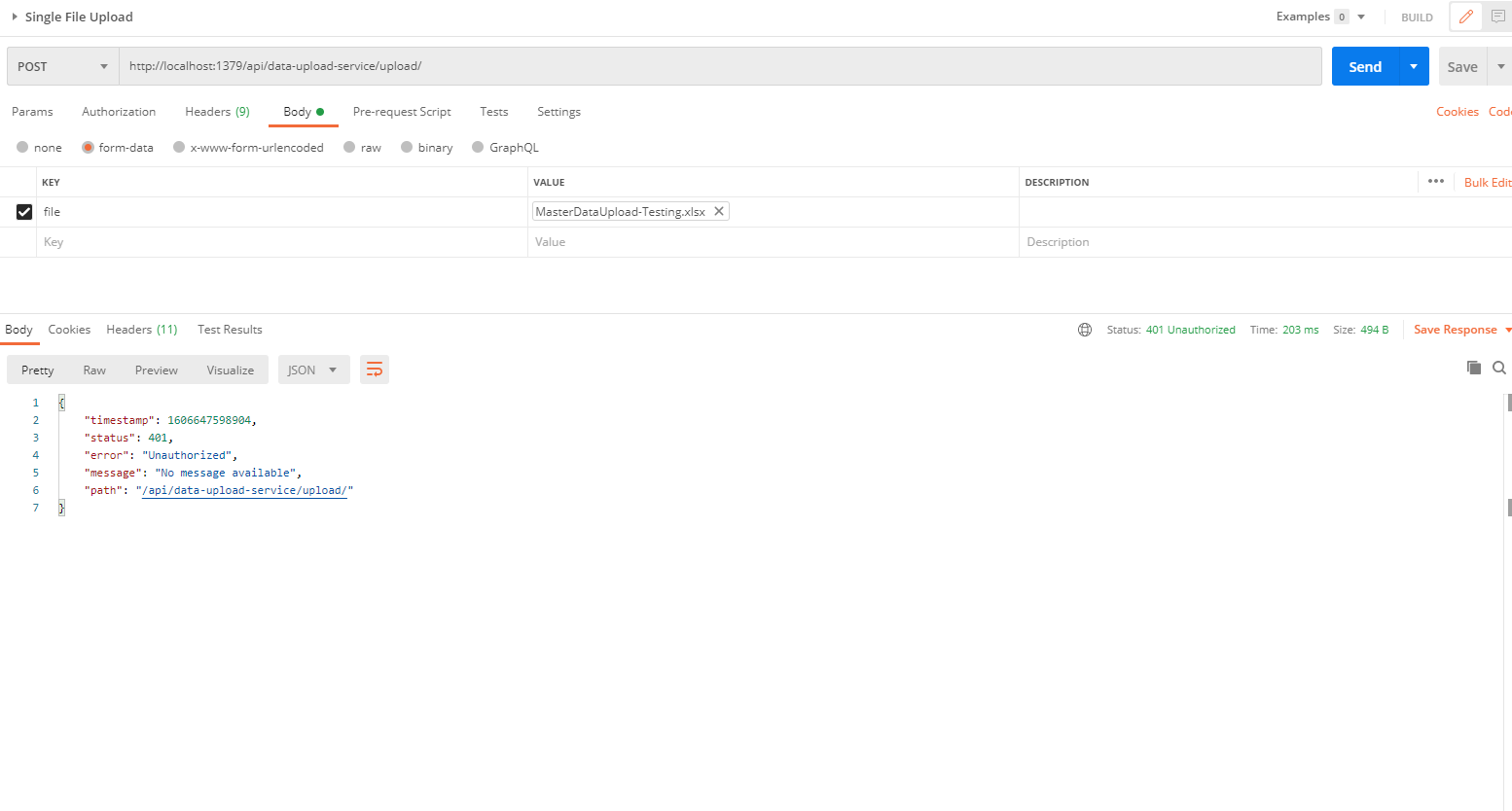
* Run the application as **Spring Boot App** in the following order
  + ConfigServerApplication
  + EurekaServerApplication
  + UserMgmtRestApplication
  + AuthServiceApplication
  + ZuulGatewayApplication
  + PeopleMgmtRestApplication
  + ProjectMgmtRestApplication
  + AccountMgmtRestApplication
  + DataUploadServiceApplication
* Open the browser of Eureka Server at localhost:8761, you should see the all the microservices running and enter the url: <http://localhost:8761/>



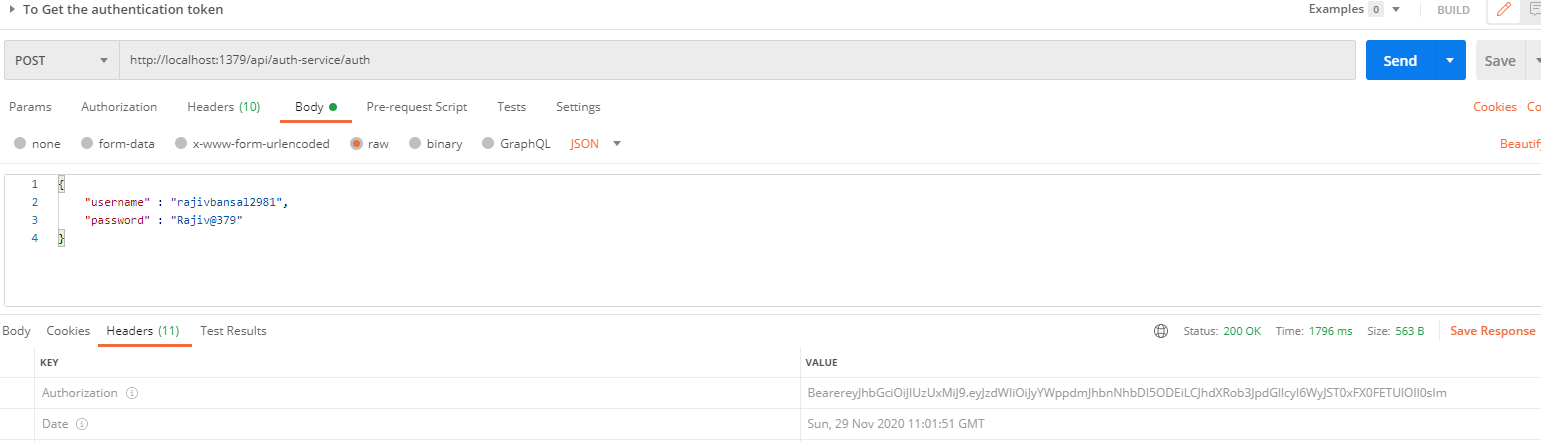
* Testing
  + Without authentication means directly hitting the data-upload-service running on 4379 and save the darshan details



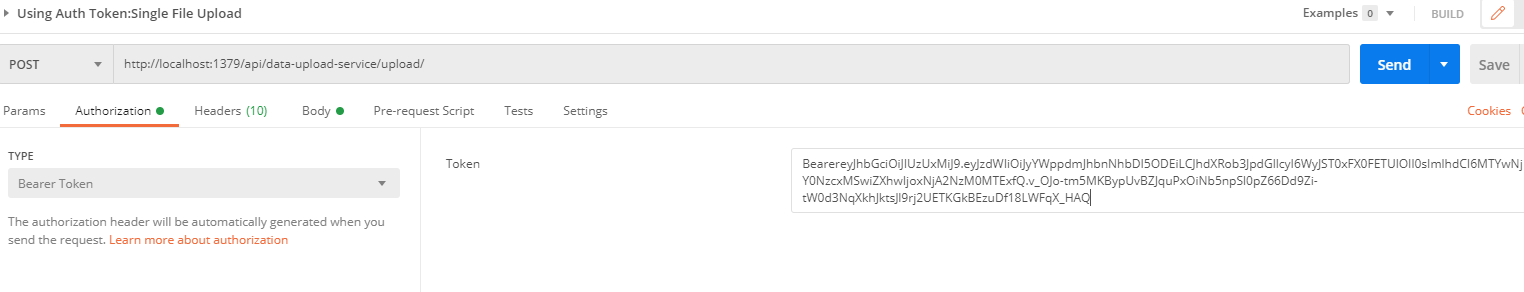
* + With authentication means every request to data-upload-service microservice will be hit via gateway running 1379
    - with authentication means every request to data-upload-service microservice will be hit via gateway running on 1379
      * uploading the data but without token means no authorization code (Bearer token) in the header will give the error.



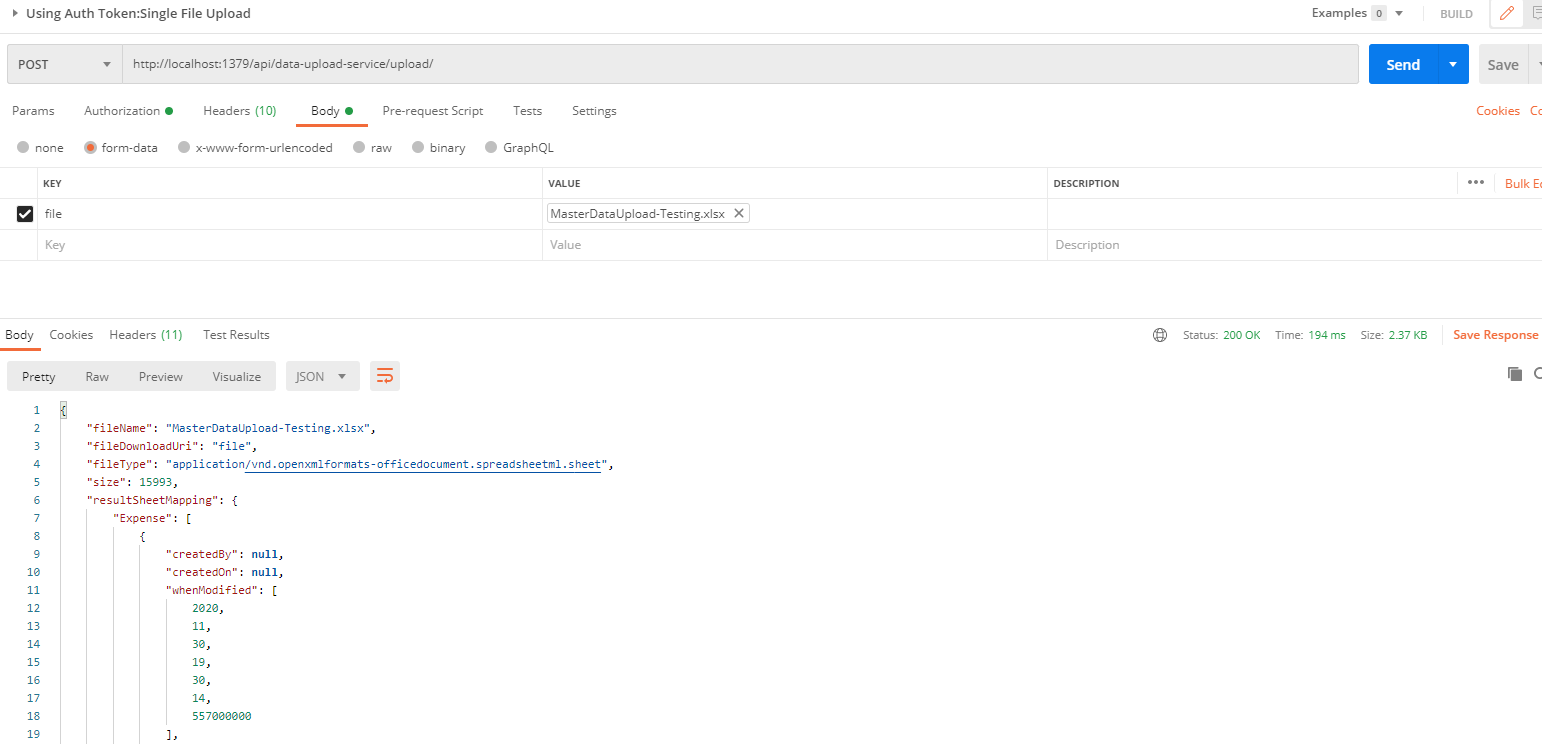
* Uploading file with token means authorization code (Bearer token) in the header will be provided using authentication service
  + 1. **To get authentication code in response header using authentication service**



* + 1. **Now hit the** data-upload-service **to** upload the file by providing the token means authorization code (Bearer token) in the header received in above step
  + See the **Authorization** tab in the darshan request



* + See the **Body** tab in the darshan request and response can be seen.



## **Message Sender Service (Kafka)**

**Why Kafka?**

Traditional messaging queues like ActiveMQ, RabbitMQ can handle high throughput usually used for long-running or background jobs and communicating between services.

Kafka is a stream-processing platform built by LinkedIn and currently developed under the umbrella of the Apache Software Foundation. Kafka aims to provide low-latency ingestion of large amounts of event data.

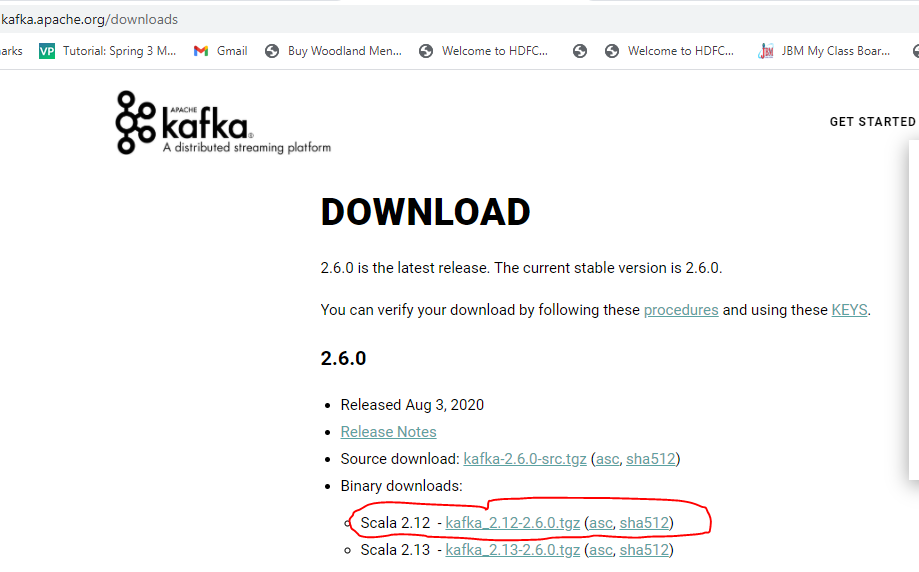
**We can use Kafka when we have to move a large amount of data and process it in real-time**. An example would be when we want to process user behavior on our website to generate product suggestions or monitor events produced by our micro-services.

Kafka is built from ground up with horizontal scaling in mind. We can scale by adding more brokers to the existing Kafka cluster.

**Kafka Vocabulary**

Let’s look at the key terminologies of Kafka:

1. **Producer**: A producer is a client that sends messages to the Kafka server to the specified topic.
2. **Consumer**: Consumers are the recipients who receive messages from the Kafka server.
3. **Broker**: Brokers can create a Kafka cluster by sharing information using Zookeeper. A broker receives messages from producers and consumers fetch messages from the broker by topic, partition, and offset.
4. **Cluster**: Kafka is a distributed system. A Kafka cluster contains multiple brokers sharing the workload.
5. **Topic**: A topic is a category name to which messages are published and from which consumers can receive messages.
6. **Partition**: Messages published to a topic are spread across a Kafka cluster into several partitions. Each partition can be associated with a broker to allow consumers to read from a topic in parallel.
7. **Offset**: Offset is a pointer to the last message that Kafka has already sent to a consumer.
8. To work with Kafka, need to install and setup Kafka.
9. To download and install Kafka, please refer the official guide [https://kafka.apache.org/quickstart](https://kafka.apache.org/quickstart" \t "_blank) . Click on the following selected link.



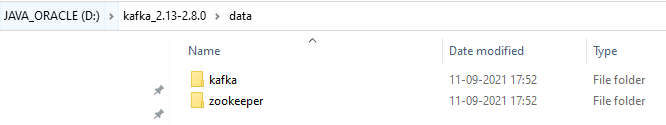
1. Click on the highlighted text to download Kafka: **[kafka\_2.13-2.8.0.tgz](https://dlcdn.apache.org/kafka/2.8.0/kafka_2.13-2.8.0.tgz)**



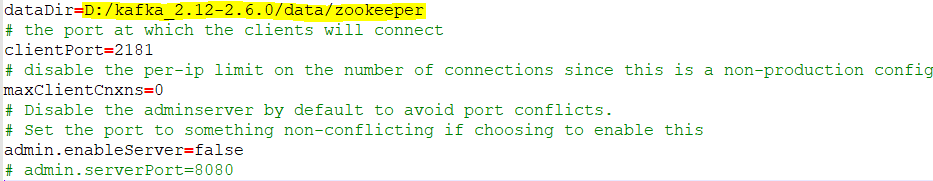
1. Open the powershell and unzip the **[kafka\_2.13-2.8.0.tgz](https://dlcdn.apache.org/kafka/2.8.0/kafka_2.13-2.8.0.tgz)** file.

Command: tar -xf .\kafka\_2.13-2.8.0.tgz

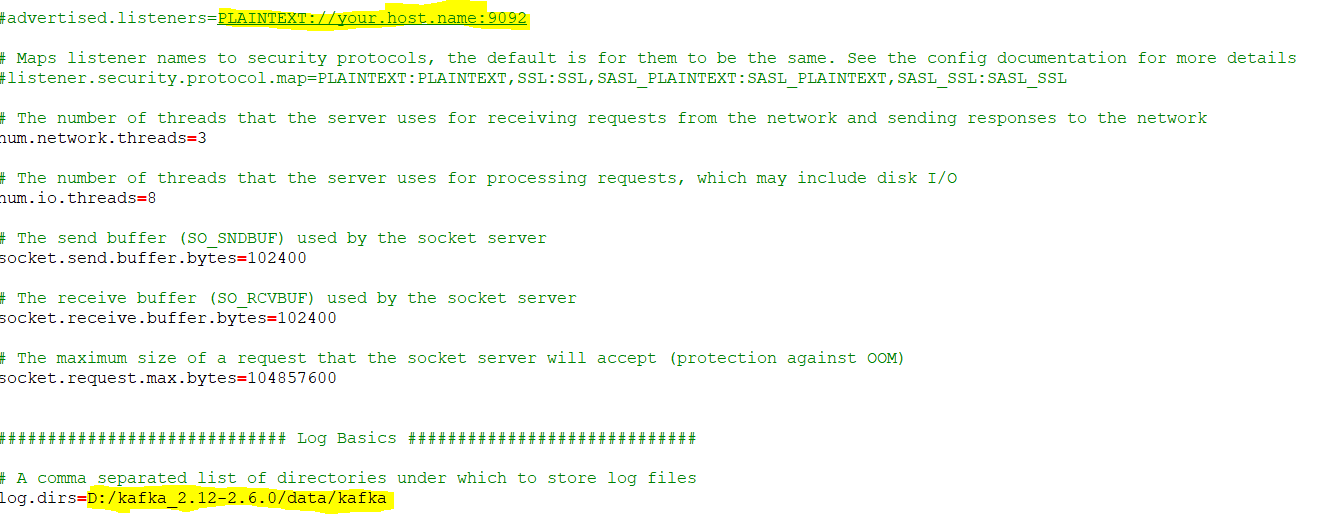
1. **Create data folder and subfolders: zookeeper and kafka in the main directory: kafka\_2.13-2.8.0**



1. **Change the default configuration value**
   1. Update dataDir in zookeeper.properties file located in “D:\kafka\_2.12-2.6.0\config\zookeeper.properties” configuration file.



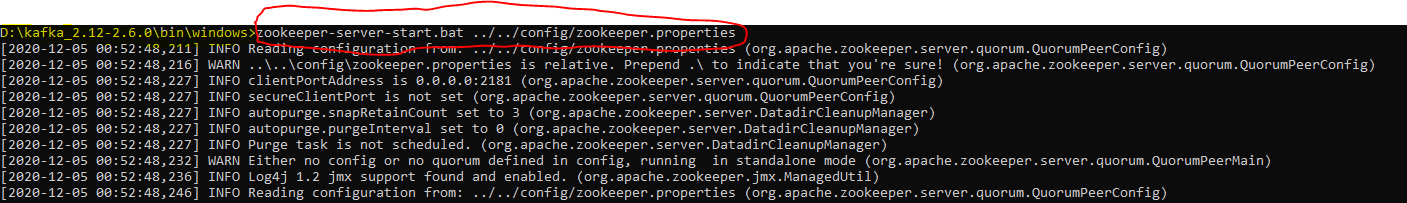
* 1. Update logs.dir in server.properties file located in “D:\kafka\_2.12-2.6.0\config\server.properties” configuration file.



1. **Start Zookeeper**

Now time to start zookeeper from command prompt. Change your directory to **D:\kafka\_2.12-2.6.0\bin\windows** and execute **zookeeper-server-start.bat** command with config/zookeeper.properties configuration file.

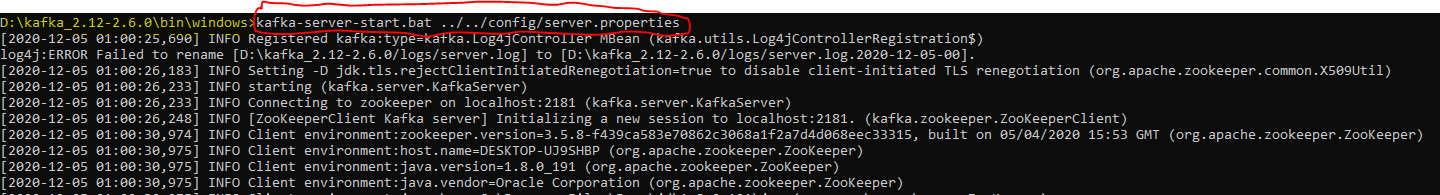
D:\kafka\_2.12-2.6.0\bin\windows>zookeeper-server-start.bat ../../config/zookeeper.properties



1. **Start Kafka**

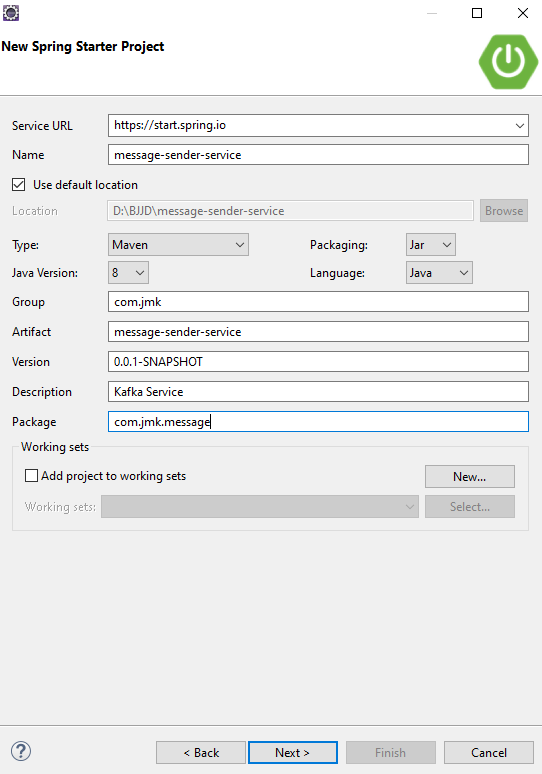
Finally time to start Apache Kafka from command prompt. Change your directory to **D:\kafka\_2.12-2.6.0\bin\windows** and run **kafka-server-start.bat**command with kafka config/server.properties configuration file.

D:\kafka\_2.12-2.6.0\bin\windows>kafka-server-start.bat ../../config/server.properties

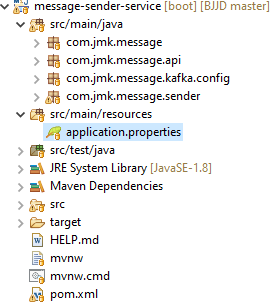


1. Now, by default, the Kafka server starts on localhost:9092.

1. Create the Project using spring Starter Project.

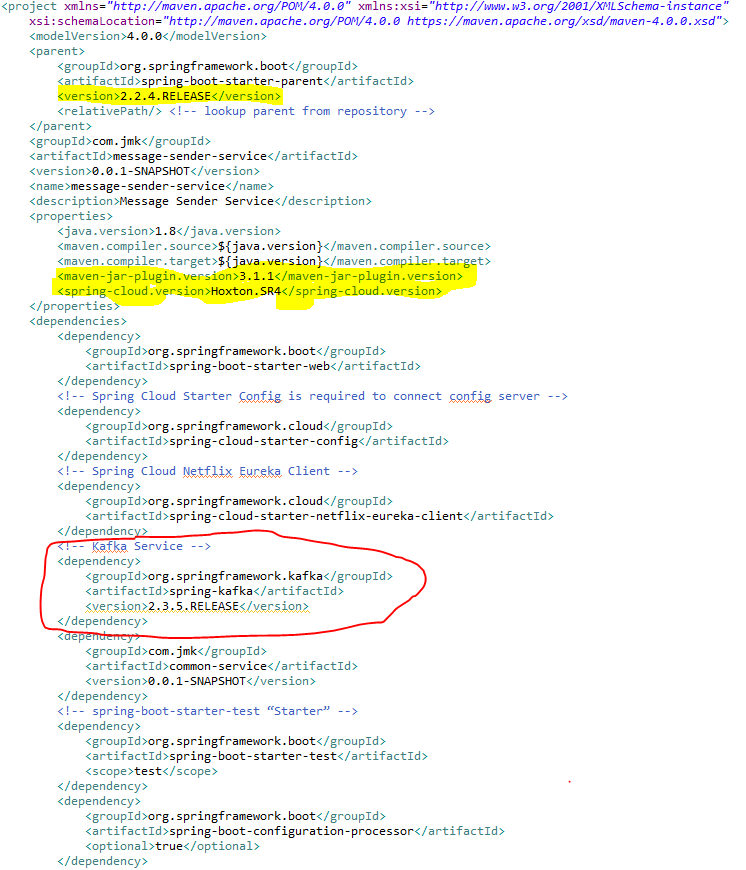


1. Click on Next and Finish.



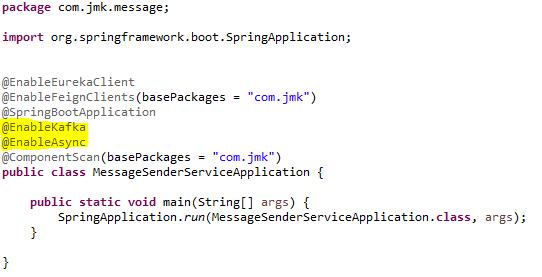
1. Import the Project

* Add the following dependencies in pom.xml with the following considerations:
  + Spring Boot Starter Parent: update the version: 2.2.4.RELEASE highlighted in yellow colour.
  + Spring Cloud version: upgraded to Hoston.SR4. It is highlighted in yellow colour.
  + Maven Jar Plugin Version: After the above updates, pom is giving compiler error in pom.xml so downgraded version of maven jar plugin to 3.1.1. It is highlighted in yellow colour.
  + Spring Kafka:  used to publish and subscribe the messages based on the fault-tolerant messaging system.  It is fast, scalable and distributed by design.





* Update the MessageSenderServiceApplication.java with the following details:
  + Enable Eureka Client so that it can be register with Eureka Server
  + Enable JpaRepositories
  + Enable FeignClient to interact with user-mgmt-service, people-mgmt-service.
  + Enable EnableSwagger2 so that we can view the document api
  + Enable Kafka and Async communication



* application.properties : Make application.properties empty as it is not required here and will be referred in other microservices. Will define the properties of spring.kafka in other microservices.
* Spring Boot does most of the configuration automatically, so we can focus on building the listeners and producing the messages. It also provides the option to override the default configuration through application.properties. The Kafka configuration is controlled by the configuration properties with the prefix spring.kafka.\*:



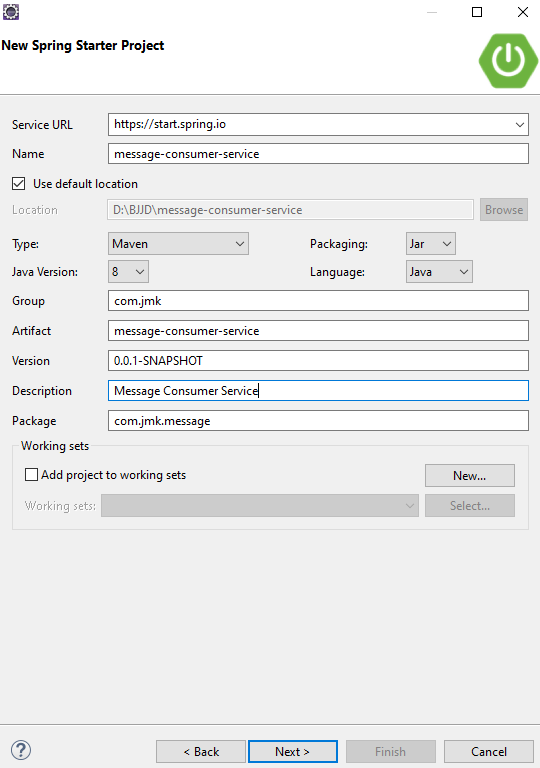
1. To split up responsibilities, we have created two projects:
   * Message-sender-service: It is related to pushing the message on the topic using Spring Producer Configuration. i.e. current section going on.
   * Message-consumer-service: It is related to listening the message received on the topic using kafka consumer configuration. It will be the next topic.
2. Kafka Message Producer Configuration

Let’s have a look at the producer configuration first.

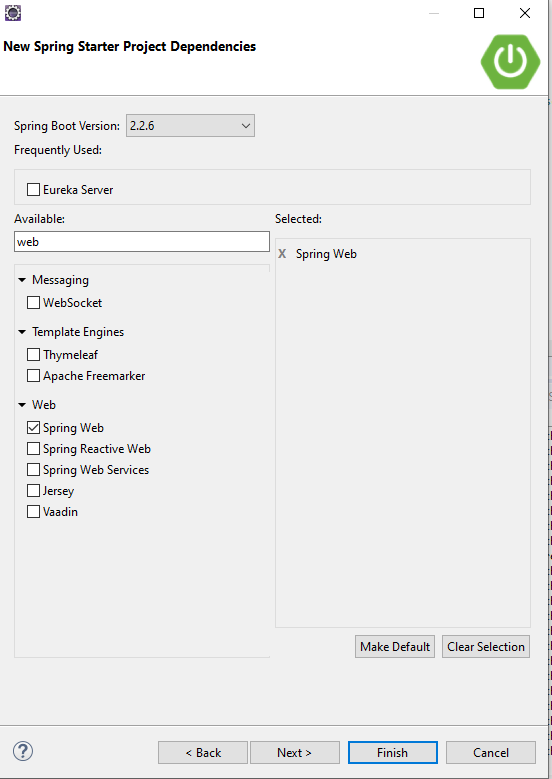
## **Messaging Consumer Service (Kafka and Twilio)**

This service will send the message to the user using the following medium:

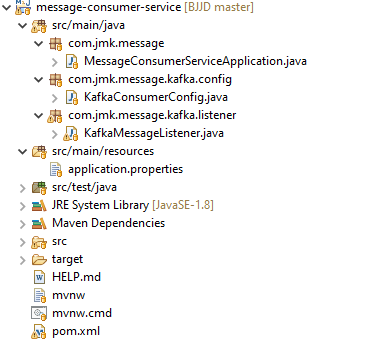
1. WhatsApp
2. Email
3. SMS
4. Create the Project using spring Starter Project.



1. Click on Next and select the Spring Web Module.



1. Click on Finish and import the project.



1. Import the Project

* Add the following dependencies in pom.xml with the following considerations:
  + Spring Boot Starter Parent: update the version: 2.2.4.RELEASE highlighted in yellow colour.
  + Spring Cloud version: upgraded to Hoston.SR4. It is highlighted in yellow colour.
  + Maven Jar Plugin Version: After the above updates, pom is giving compiler error in pom.xml so downgraded version of maven jar plugin to 3.1.1. It is highlighted in yellow colour.
  + Spring Kafka:  used to publish and subscribe the messages based on the fault-tolerant messaging system.  It is fast, scalable and distributed by design.