**File Management**

**Back-End Development:**

This Project is all about learning how to perform CRUD operations on files. We were given a manual containing user stories to initiate the project and the concepts of spring such as spring security (JWT), spring batch to implement on the project.

This Spring Boot application is designed to efficiently process and manage daily files. It handles the ingestion of fixed-length data, stages it in a database, and provides a RESTful API for external applications to monitor and manage the file load process. The system leverages Spring Batch for robust file processing, Spring JDBC and Spring Data JPA for database interactions, and Spring Security with JWT for secure access. Open API is used to generate API documentation.

The basic packages to perform CRUD operations on the files are:

1. **Parent** – The main program of project from where the project starts.
2. **API** – Controller layers that contains the end points.
3. **DAO** – The repositories used in the project that implements JPA repository.
4. **Entity** – The Entity classes for which the tables are created in database.
5. **Service** – Contains classes where actual business logic is performed.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig:** *Basic packages used in the project*

The Project was started by creating the model classes used in the project or for which tables have to be created in the database. The Entity class named “FileLoad” is placed in Entity package:

A computer screen shot of text

AI-generated content may be incorrect.

**Fig:** *FileLoad Entity class*

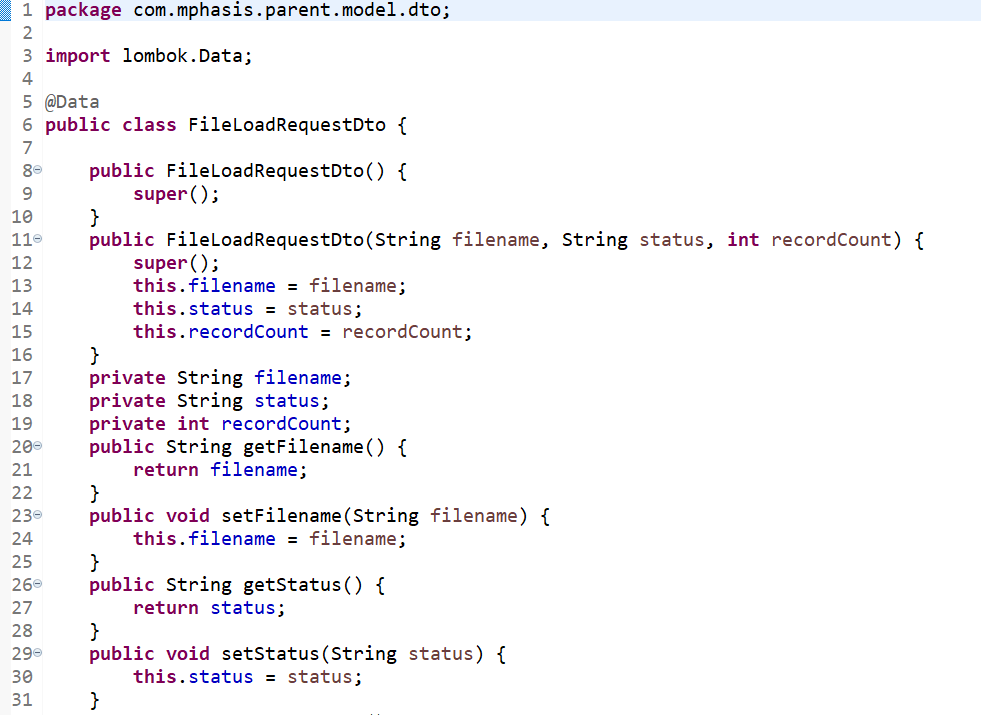
The Second most important thing was to create a repository class named “FileLoadRepository” that implements the JPA repository that connects to the database (MySQL). The class looks like:

A screenshot of a computer program

AI-generated content may be incorrect.

**Fig:** *FileLoadRepository class*

I have also created a FileLoad Request and Response DTOs (Data Transfer Objects) to record the requests from the end user. DTOs are used to transfer objects across layers. These are placed inside another package named “model.dto”. The FileLoad Request Dto contains the parameters like filename, status, record count and setters and getters of those variables in order to fetch data from end-user to the database or vice versa. The FileLoad Request Dto looks like:

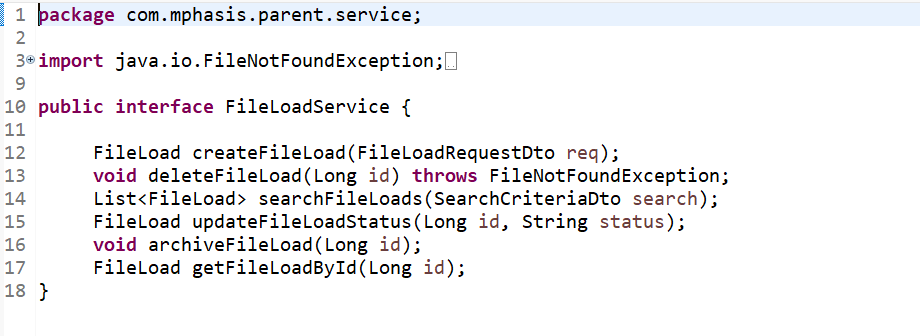


**Fig:** *FileLoad Request Dto to accept parameters*

Before creating a service layer to write business logic, I created a FileLoad Service interface for several reasons:

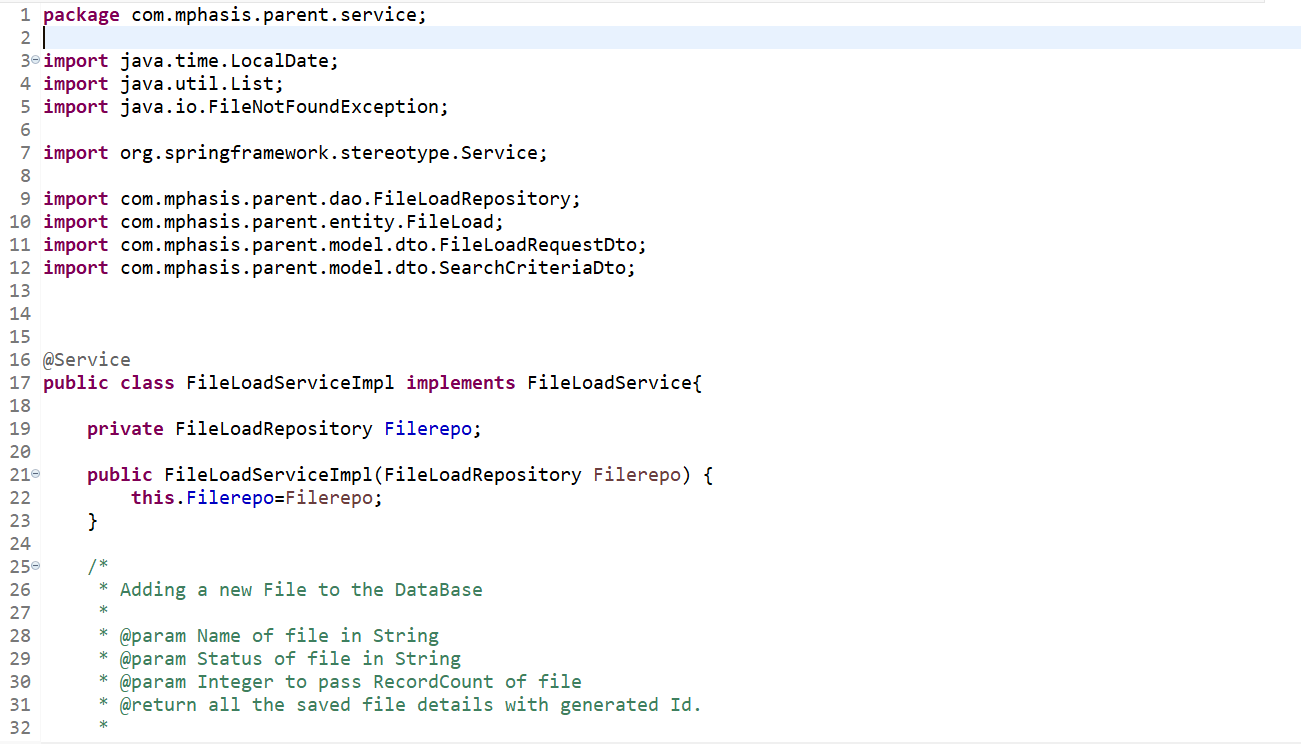
1. Loose coupling and Flexibility
2. Supports Dependency Injection
3. Enhances Testability
4. Encourages Code Maintainability

My FileLoad Service Interface looks like this:

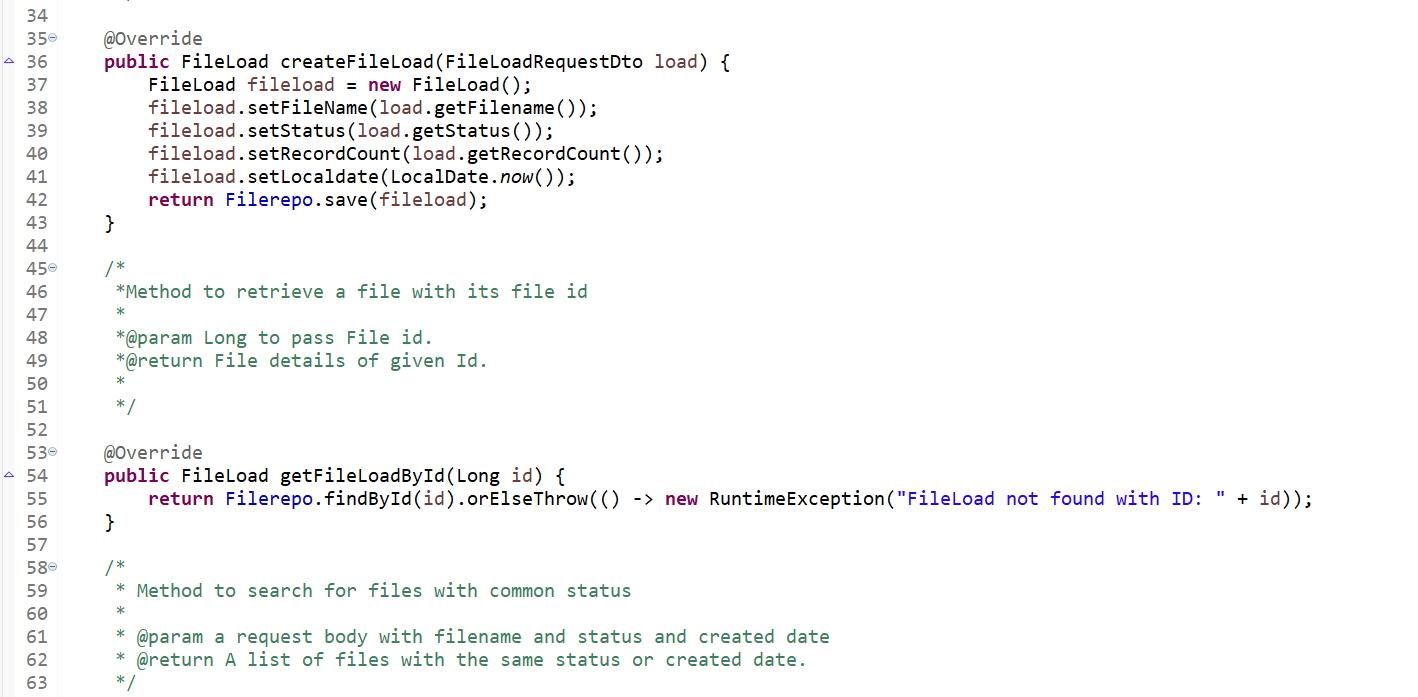


**Fig:** *FileLoad Service Interface*

After creating an interface for Service class, I started working on the business logic in “FileLoadServiceImpl” class by overriding the methods that has been written in service interface, The service class is annotated with @Service annotation to create beans for the methods that have been created in the class and the FileLoadServiceImpl class looks like below:



**Fig:**  *FileLoadServiceImpl class*



**Fig:** *An Overridden method in Service class*

After working on business logic, I focused on writing the Controller class where all the end points are mapped with the methods to perform CRUD operations on files.

The Controller class is annotated with “@RestController” annotation. RestController is a convenience annotation that is itself annotated with @Controller and @Responsebody.

Types that carry this annotation are treated as controllers where RequestMapping methods assume ResponseBody semantics by default.

I have used PostMapping, an Annotation for mapping HTTP POST requests onto specific handler methods.

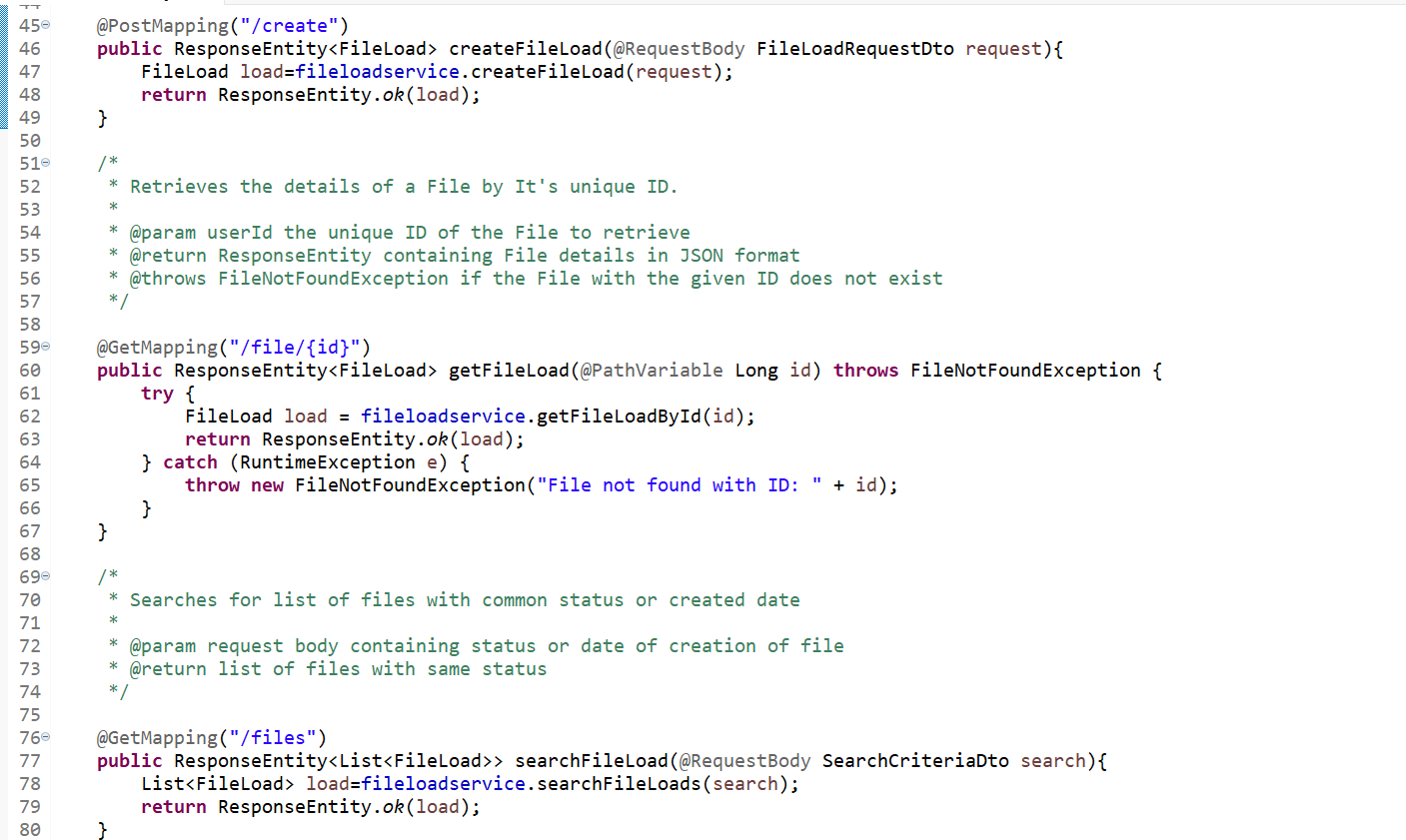
Specifically, PostMapping is a *composed annotation* that acts as a shortcut for RequestMapping (method = RequestMethod.POST).

Some Other RequestMapping methods such as PostMapping, GetMapping, PostMapping and DeleteMapping are used to perform different tasks such as writing into database, fetching from database, updating records in database and deleting data from the database respectively.

All the methods present in the service class are called in the controller class and given end points to them, so that they can be accessed by the end user. The controller class looks like this:

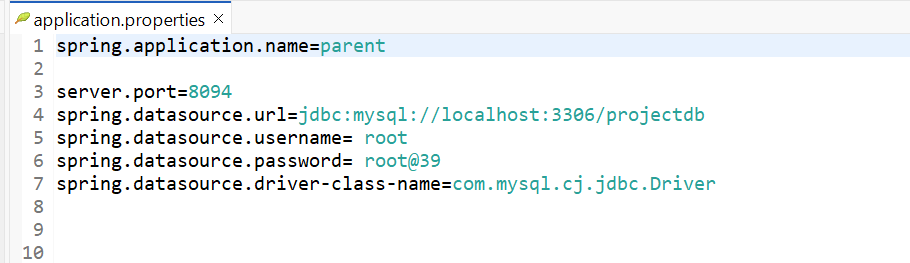


**Fig:** *Controller class in our application*



**Fig:** *End points for different purposes*

To connect with the database, it has to be connected to the database and the sql commands to connect to the database are written in application.properties under src/main/resources. The server port where the application has to be run is also mentioned in this file only and username and password of the sql database is also provided to connect the java application with the database and the commands in application.properties looks like this:

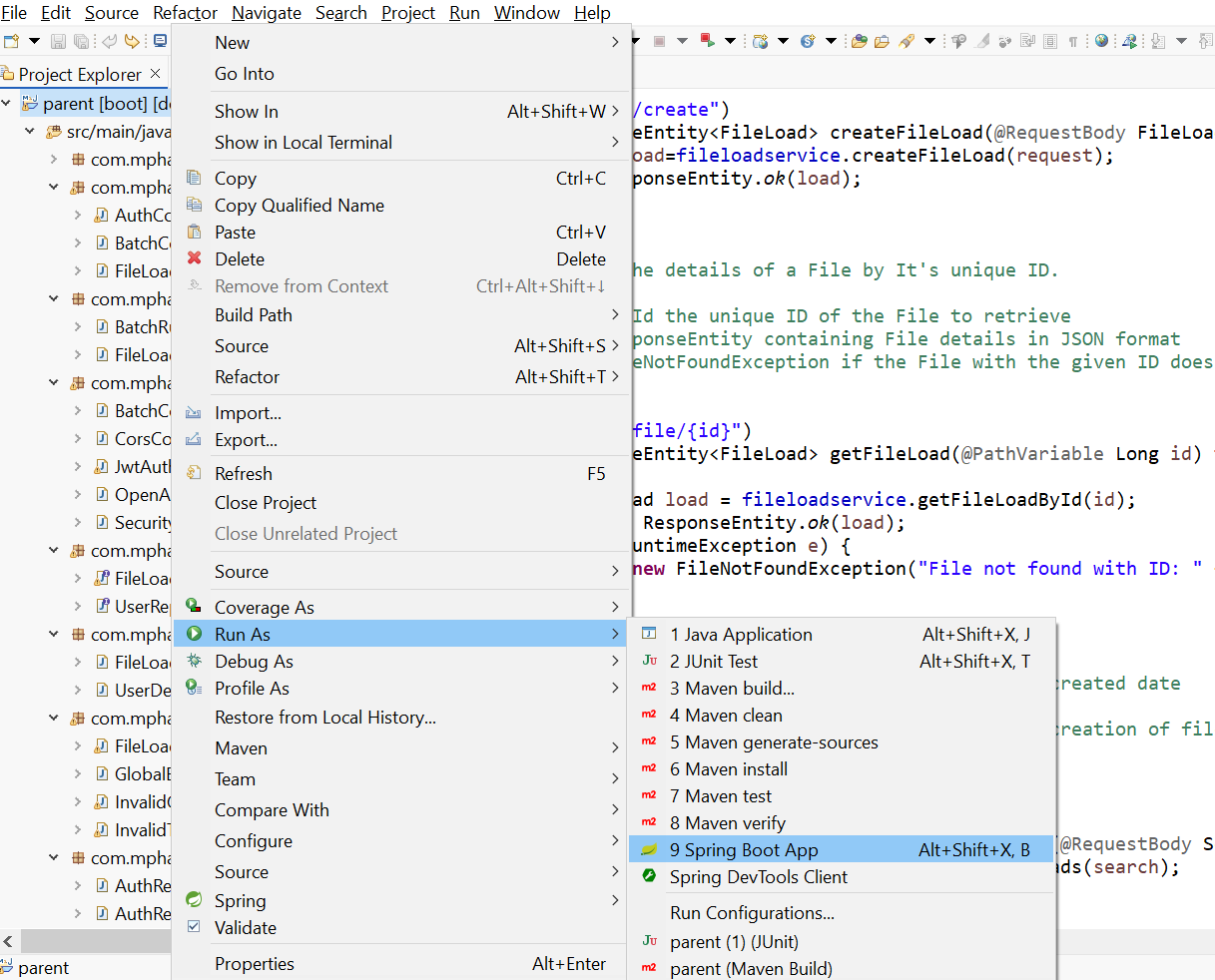


**Fig:** *application.properties to connect to database*

These endpoints were tested in postman whether they were serving the purpose they were created to do, here no authentication and authorization were implemented on our end points, so that they were free to access and can be used by anyone.

Before hitting the end point in the postman, we need to run the application as spring boot application for which we should have spring tool suite(STS), which has to be installed from Eclipse marketplace. The path to download Spring Tool Suite is- **Help > Eclipse Marketplace > STS.**

The application is run as Spring Boot App and It looks something like this:



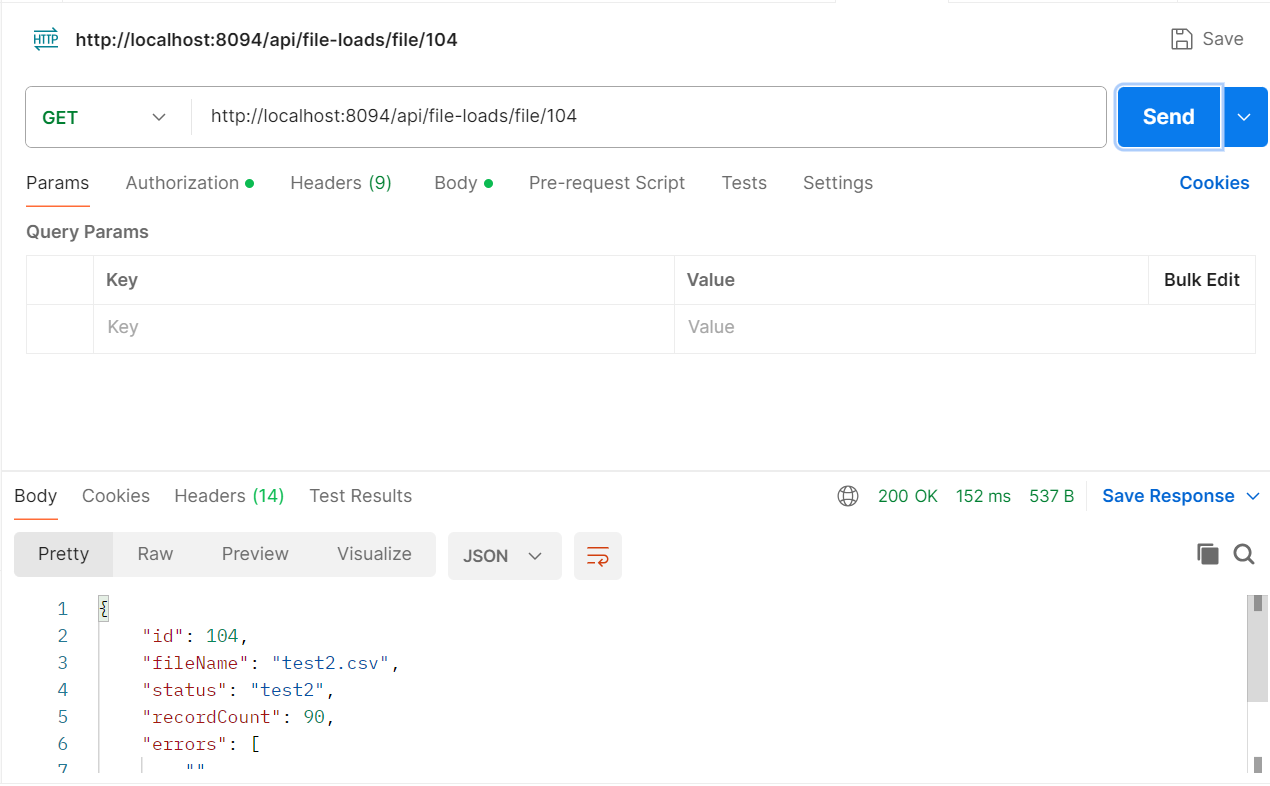
**Fig:** *Way to run application as Spring Boot App.*

The console of the application looks after running the application as this:



**Fig:** *The console of the application after running.*

We can ensure that the application is really started when you see: **Tomcat started on port(s): 8094 (http)** and **Started ParentApplication in 17.614 seconds** on your console. Then you can test your endpoints on postman.

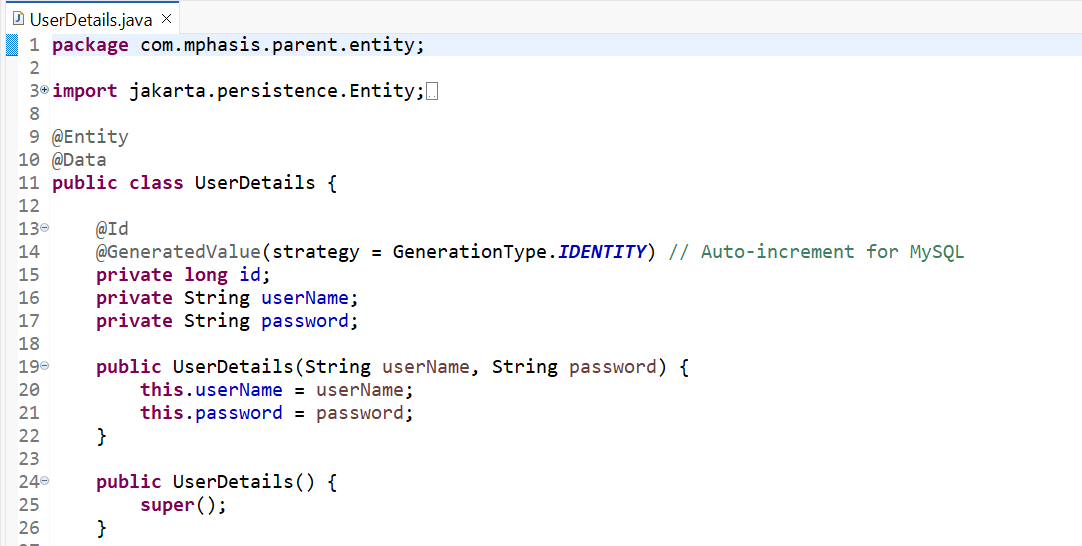


**Fig:** *Triggering an Endpoint from postman.*

Another important aspect in my project is to implement security to the endpoints, so that only authorized users can access the end points and connect to database.

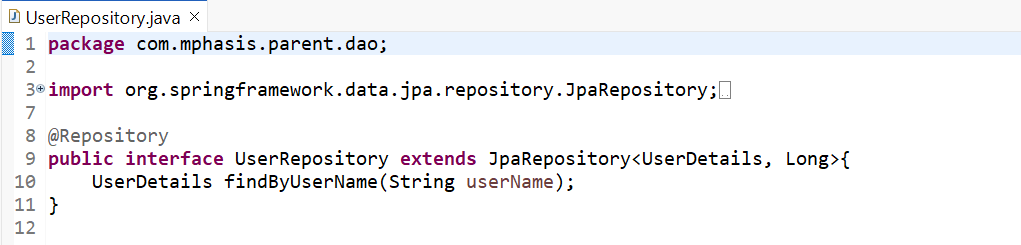
To implement security, spring provides spring security where we can use Jwt authentication, session authentication, 2 factor authentication and may more.

I have chosen Jwt authentication to provide security to my endpoints. First, I created an entity class named “UserDetails” to store the username and password of the users, who got registered and want to use endpoints.



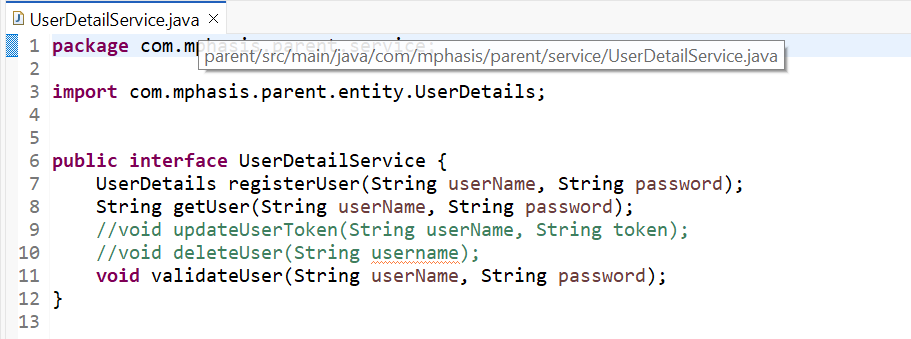
**Fig:** *UserDetails entity class*

A table named “user\_details” will be created in my sql database as it got already connected to the database and a repository class named “UserRepository” which is extended to the JPA repository is created and some methods like “findByUsername” are declared in the repository class and the class looks something like this:



**Fig:** *UserRepository class to connect to database*

Then the UserDetailService interface and UserDetailServiceImpl class that implements the interface are created and all the methods are overridden in the class. And the interface and the class looks like this:



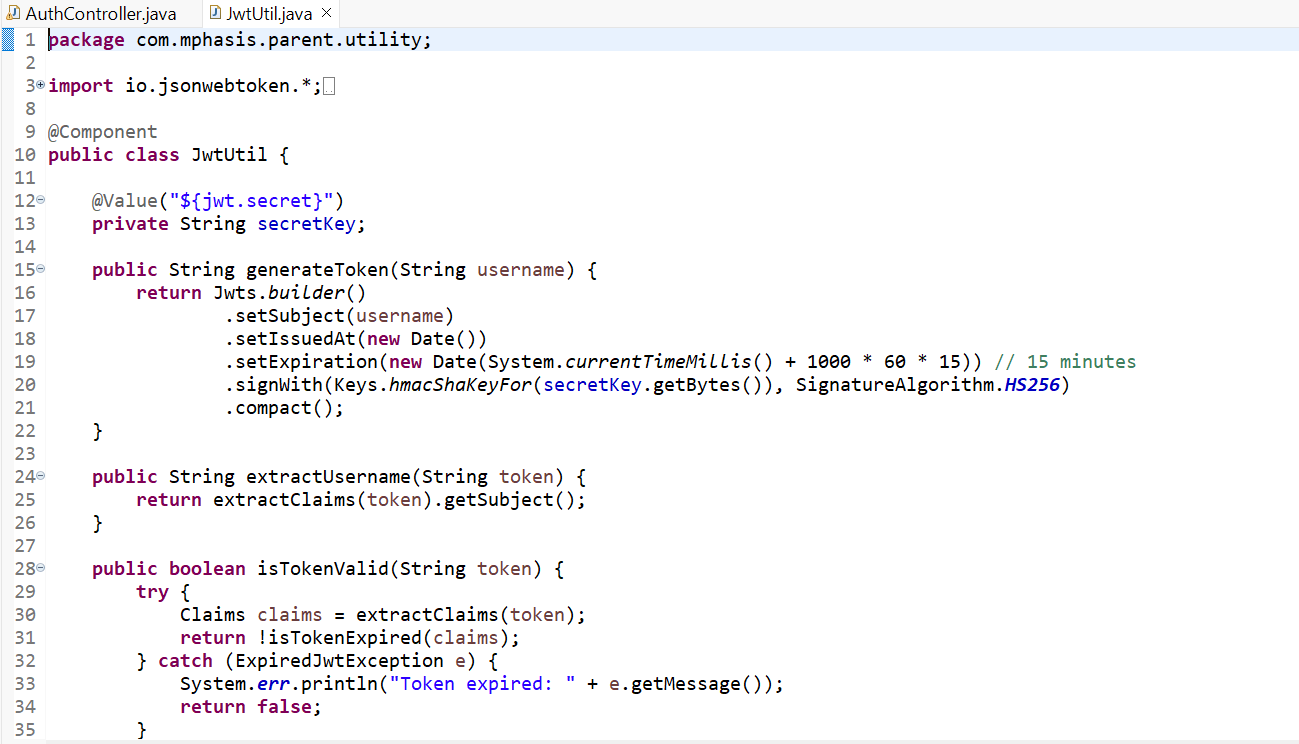
**Fig:** *UserDetailService Interface*

**

**Fig:** *UserDetailServiceImpl class*

The password is encoded using PasswordEncoder to make the original password hidden. I have used “BCryptPasswordEncoder” to encode the passwords in hash form and passwords saved may look like*- “$2a$10$95PY1T6LMiZd3HwvIdbPsetpkiXYYRN6DKdz6dH4znj/CO22mlKyS*”.

Now, the most important part in providing security is the generation of Jason Web Token (JWT). To generate Jwt, I have created a class named “JWTUtil” in another package called “utility”. The Jwt generation requires a secret key that is provided in application.properties and called using $value in the util class. In the util class only, the methods for generation of token, extracting claims from the token and validation of token are written, I made sure that the token is validated for 15 min only to prevent the misuse of the token.



**Fig:** *JWTUtil class and the methods present in it*

The UserDetailService and the JWTUtil classes are used in another class called JwtAuthenticationFilter class to extract username from the token and authenticate the user from the token. As the token is bearer token, it starts with bearer keyword, the substring after removing bearer keyword is the actual token which is used to extract the claims.



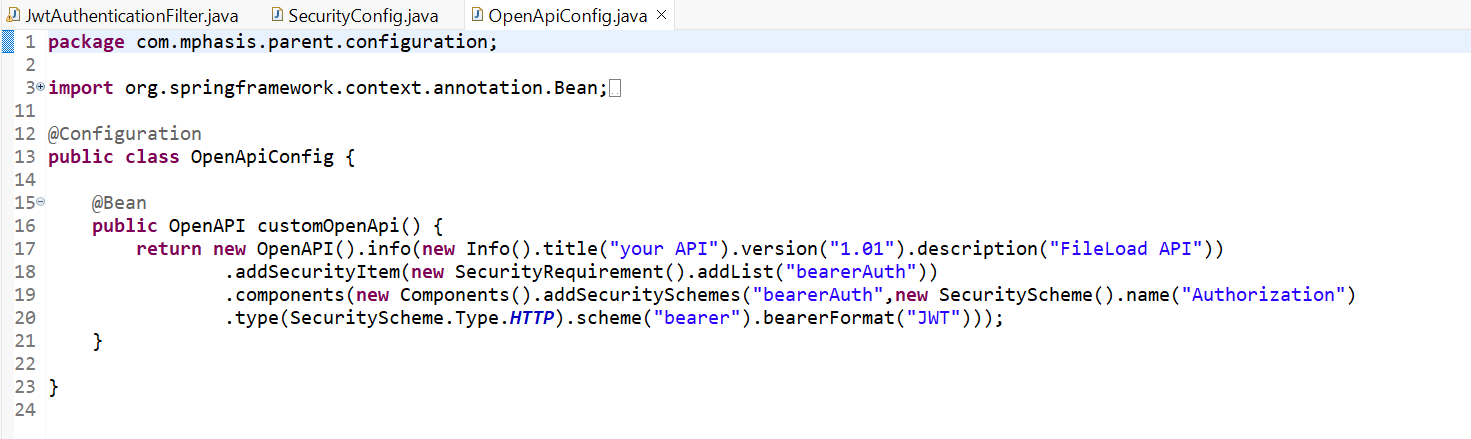
**Fig:** *Filter class to authenticate the users.*

Another Important class is the SecurityConfig class where all the configuration regarding security issues takes place, i.e., disabling of CORS(Cross Origin Resource Sharing) and authenticating request matchers that makes the particular APIs available for authenticate users only.

In this configuration class only, Integration of Swagger takes place, Swagger is an online testing tool for the create Apis and this is also used as Java doc. Required online Apis are added in the SecurityConfig class in the RequestMatchers method that is shown in below figure. To make the Apis available, I have added an **OpenApi** dependency in my pom.xml file. To make the swagger authenticated, I have written a configuration class called “OpenApiConfig”, in where the version, description, type, scheme and bearer Format of the Jwt token are declared.

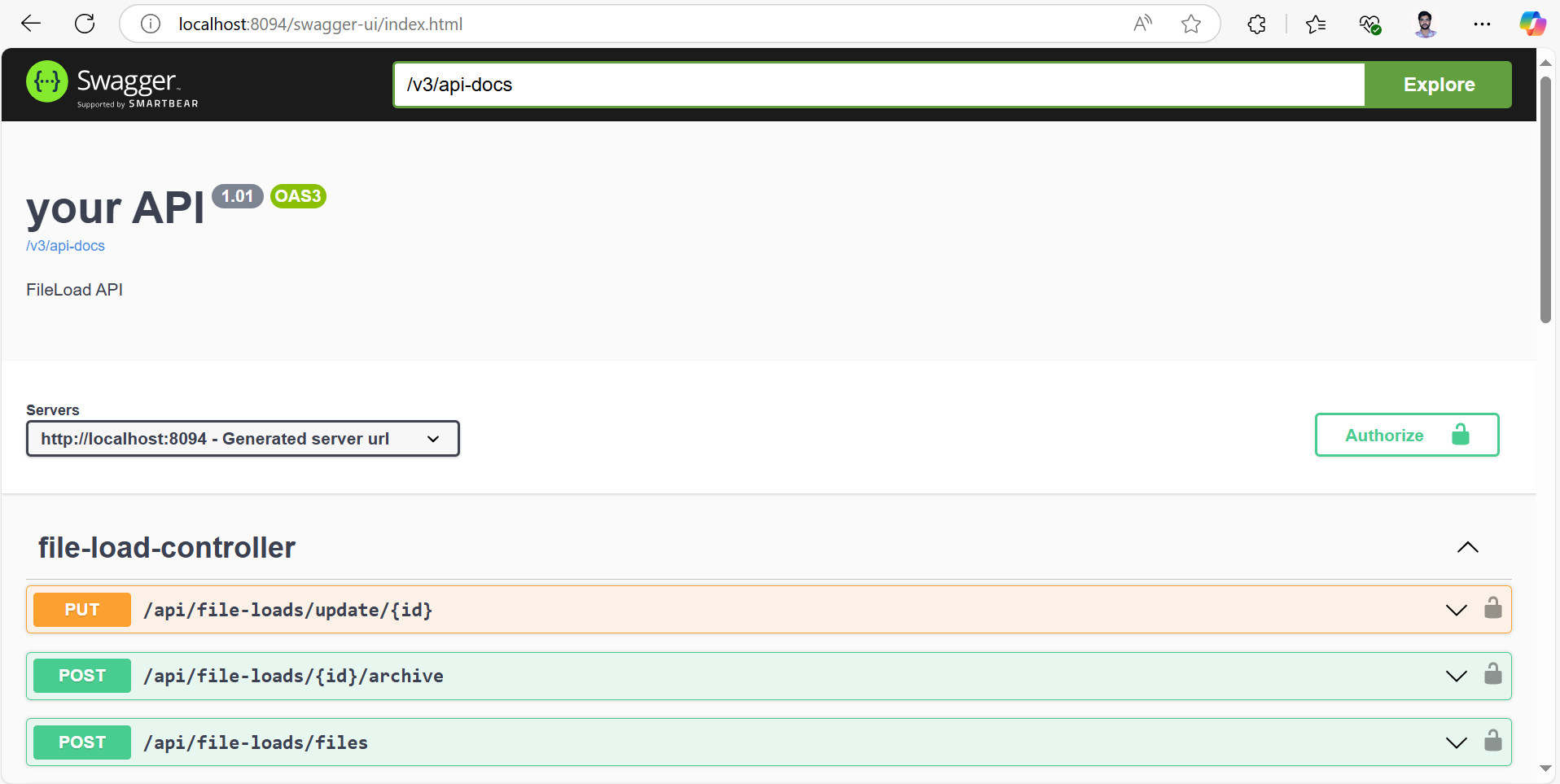


**Fig:** *SecurityConfig class to Authenticate users.*



**Fig:** *Integration of swagger to the application.*

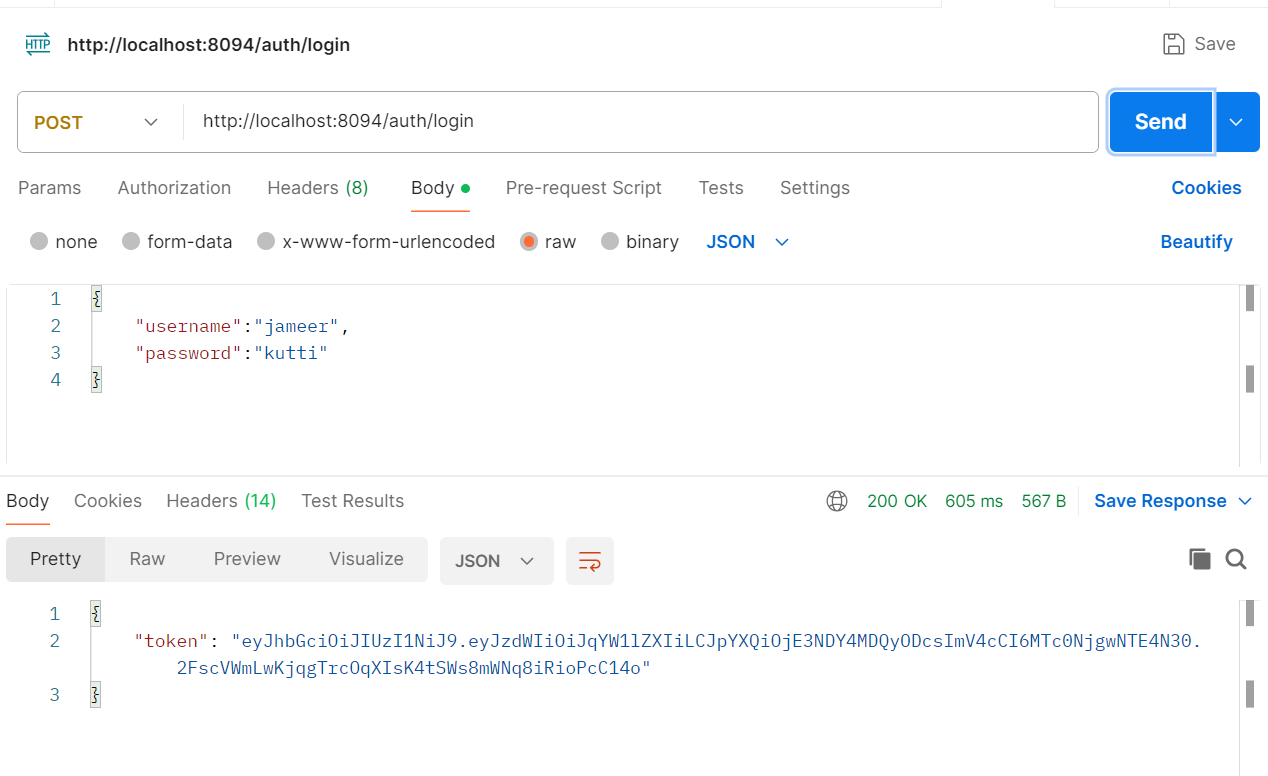
To open the swagger application, you can browse the url: <http://localhost:8094//swagger-ui/index.html> where 8094 is the port number where our application runs.



**Fig:** *Swagger Interface.*

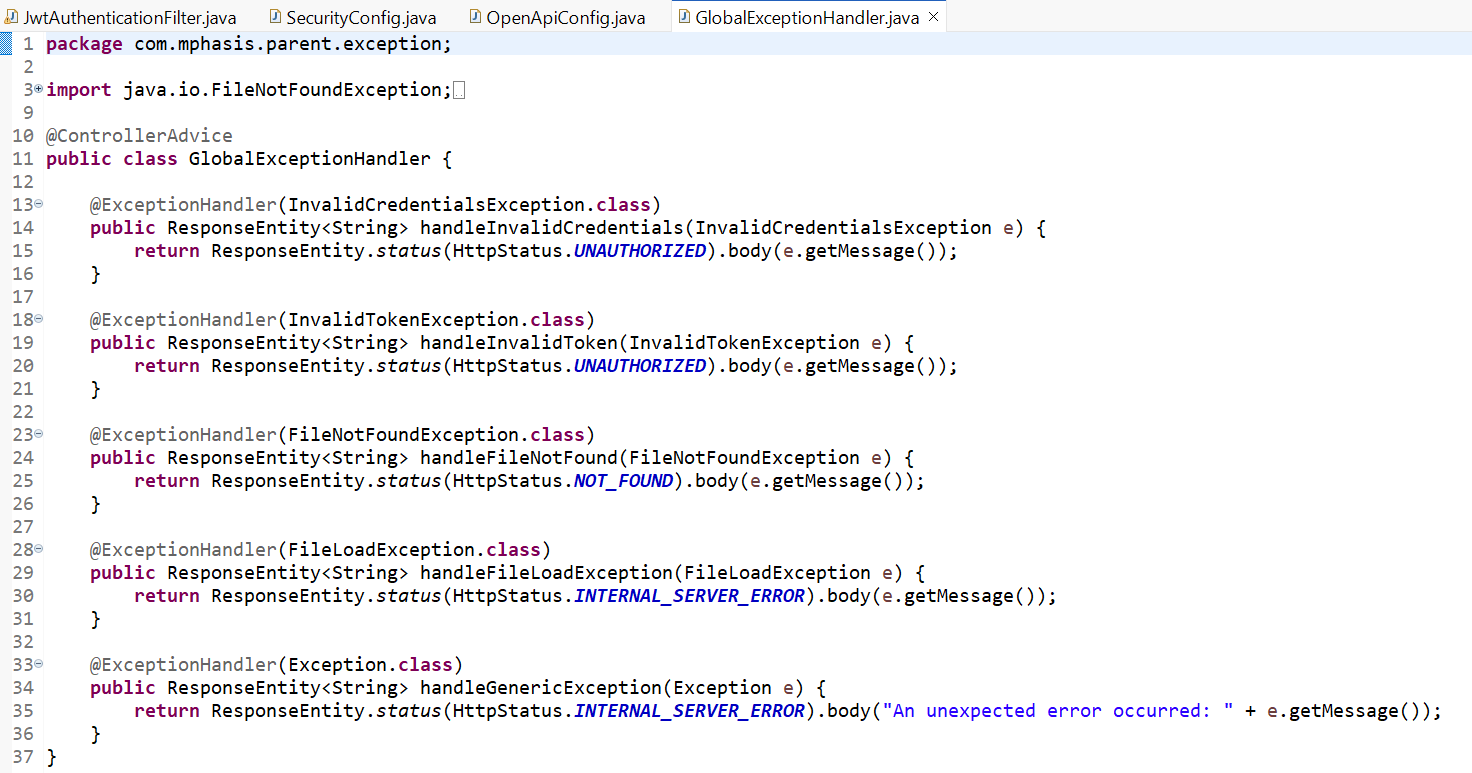
That’s how security is provided to our APIs and to check in postman, the authenticated user has to

Login with his/her credentials from where he receives a token as response that can be pasted in Authorization part of the Api to use/trigger the end point or else the user may get error.



**Fig:** *Login request and token response in postman.*

Spring provides a lot of things in which “Global Exception Handler” is one of the frequent used features, where one global class where all the exceptions are handled there itself in the class. Exceptions from various classes are thrown to the Global Exception class only.



**Fig:** *Global Exception Handler class.*

The Global Exception Handler class handles various exceptions including FileNotFound, InvalidCredentials, InvalidToken, FileLoadException which are custom Exceptions.

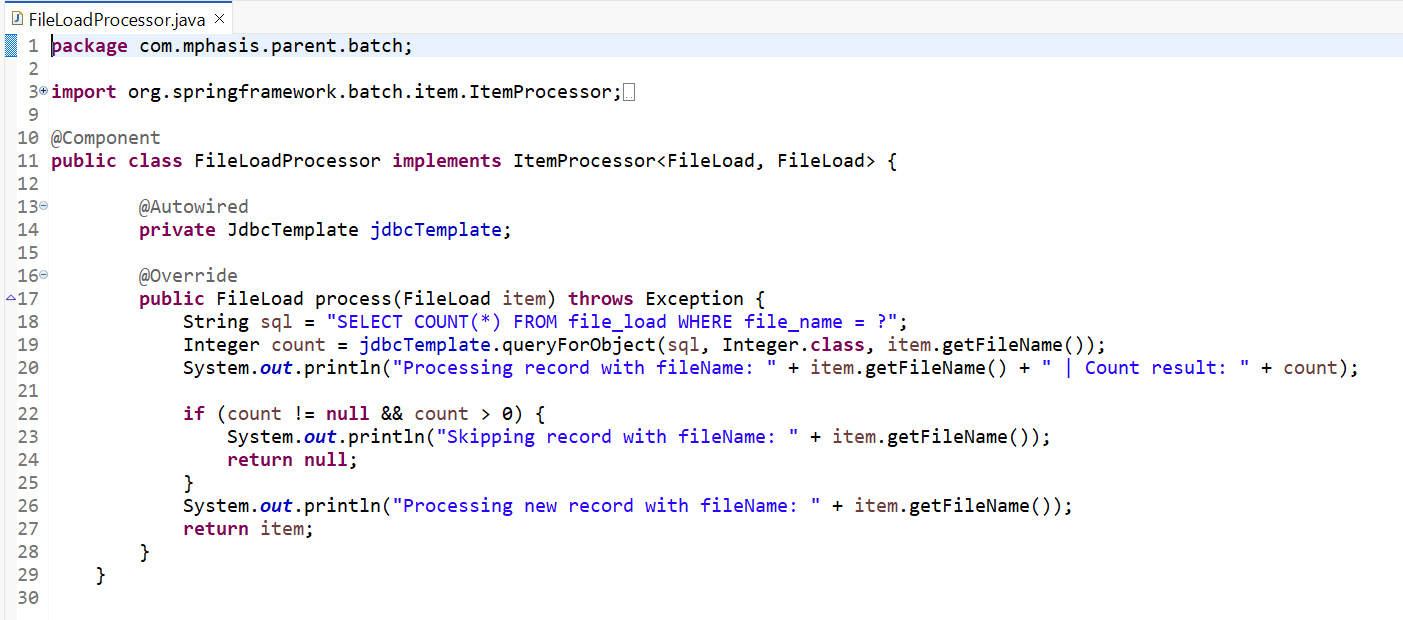
Another important feature of spring is “Batch Jobs”. To insert n number of records into a database manually is a time-consuming task, it can be done in fraction of seconds by triggering a batch job. The Batch job can be done in two ways, i.e., using a scheduler(to trigger a batch job within a given interval of time) and triggering an Api. I have implemented the batch job in the second way, i.e., when batch endpoint is triggered by an authenticated user, the batch job works and the files from a csv file will be moved into the database within a few seconds.

To implement batch job, we need to write the JobLauncher which contains JobRepository and Jobs, where the jobs consist of multiple steps and each step consists of Reader, Processor and writer which are used to read data from csv, perform some operations and write the data into the database respectively. A BatchConfig class that consists of the reader, processor and writer.



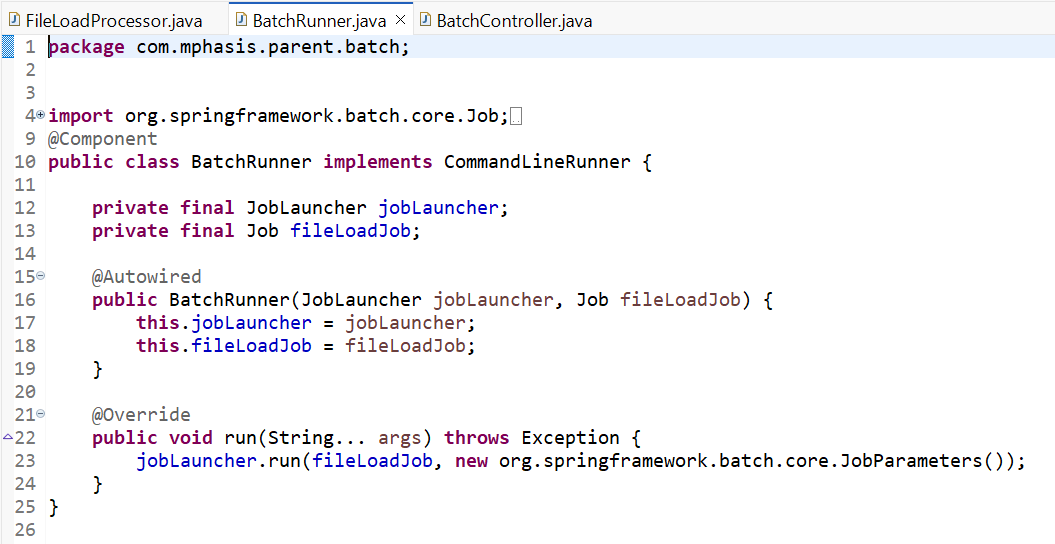
**Fig:** *BatchConfig class with reader, writer*

The processor is written separately but the bean is created in the batch config class only. The processor class looks like:



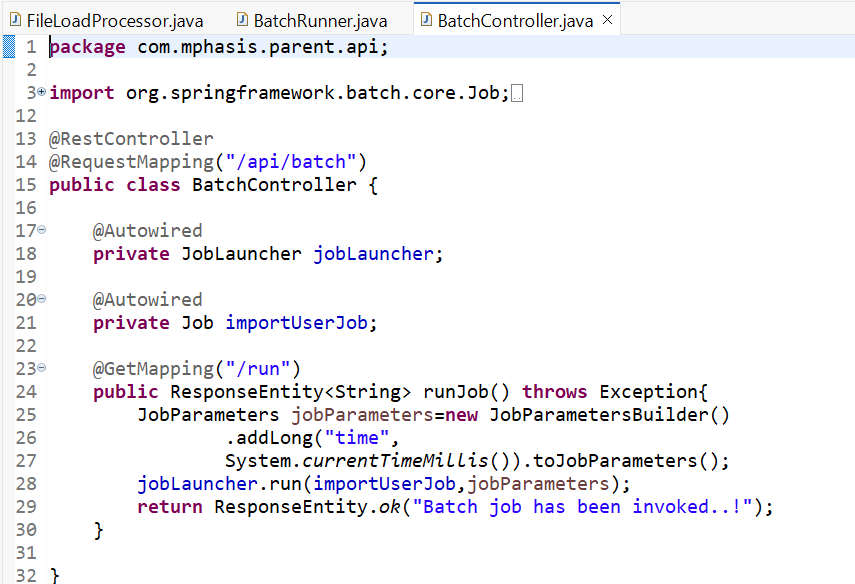
**Fig:** *FileLoadProcessor class*

A BatchRunner class that implements CommandLineRunner interface is created to run the batch job by overriding the run method of the interface that requires the Job parameters and fileLoadJob.



**Fig:** *BatchRunner class to run BatchJob*

To make the batch job trigger from an endpoint, BatchController class is created that provides an endpoint to run the batch job by triggering the endpoint.



**Fig:** *BatchController class with endpoint*

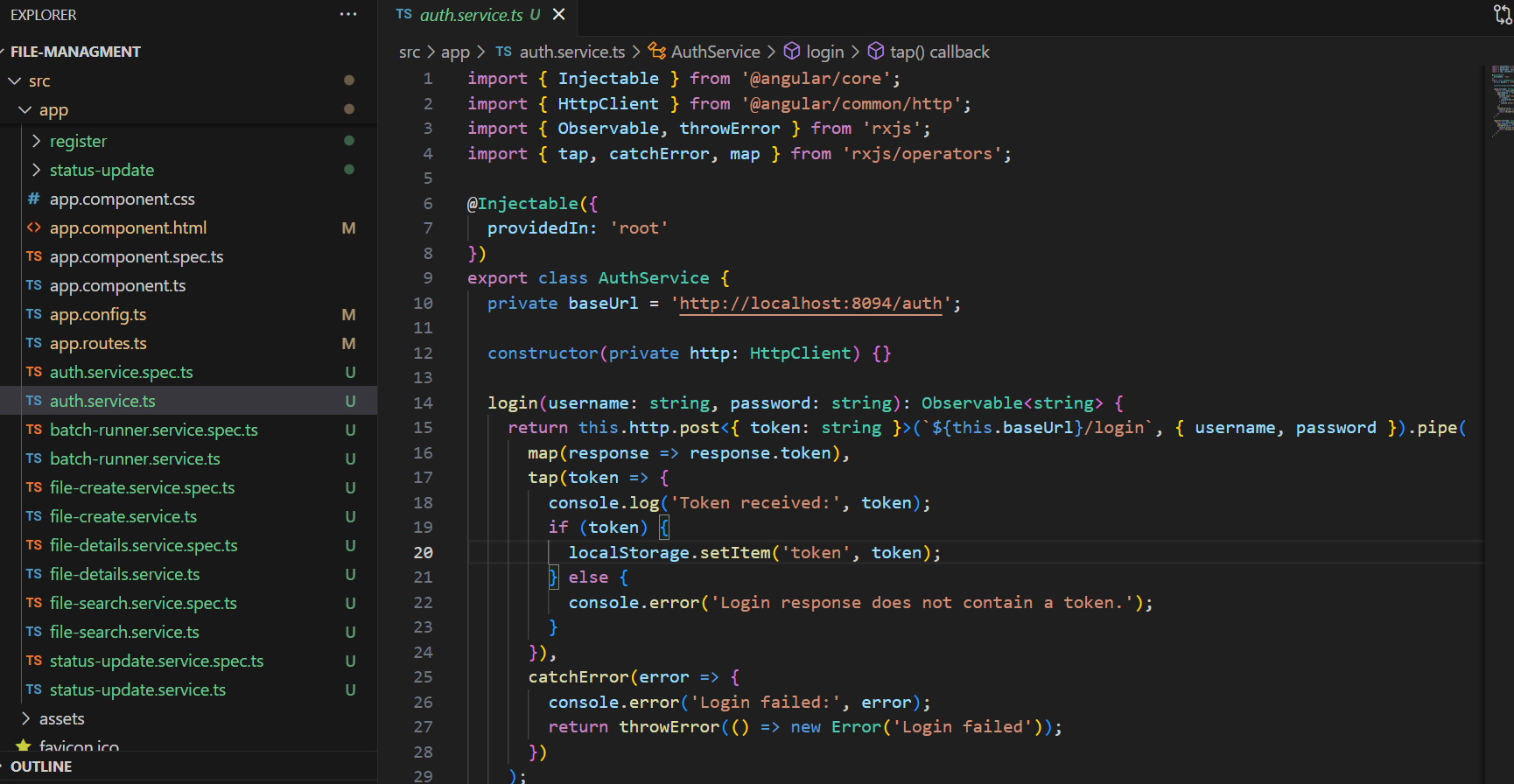
That’s how the backend part of the project is completed up to the user stories provided and all key concepts are integrated in the spring boot application created.

**Front-End Development:**

To develop the User Interface or the front-end part of the application, I have used:

* **Angular CLI**: 19.2.10
* **Node** : 23.6.0
* **Npm**: 11.3.0
* **Tool**: Vs code

I started the front-end development with the creation of login and register pages. To create those pages, I created a service named auth using the command- “**ng generate service service-name**”. It gave me AuthService and its test class.



**Fig:** *AuthService service class*

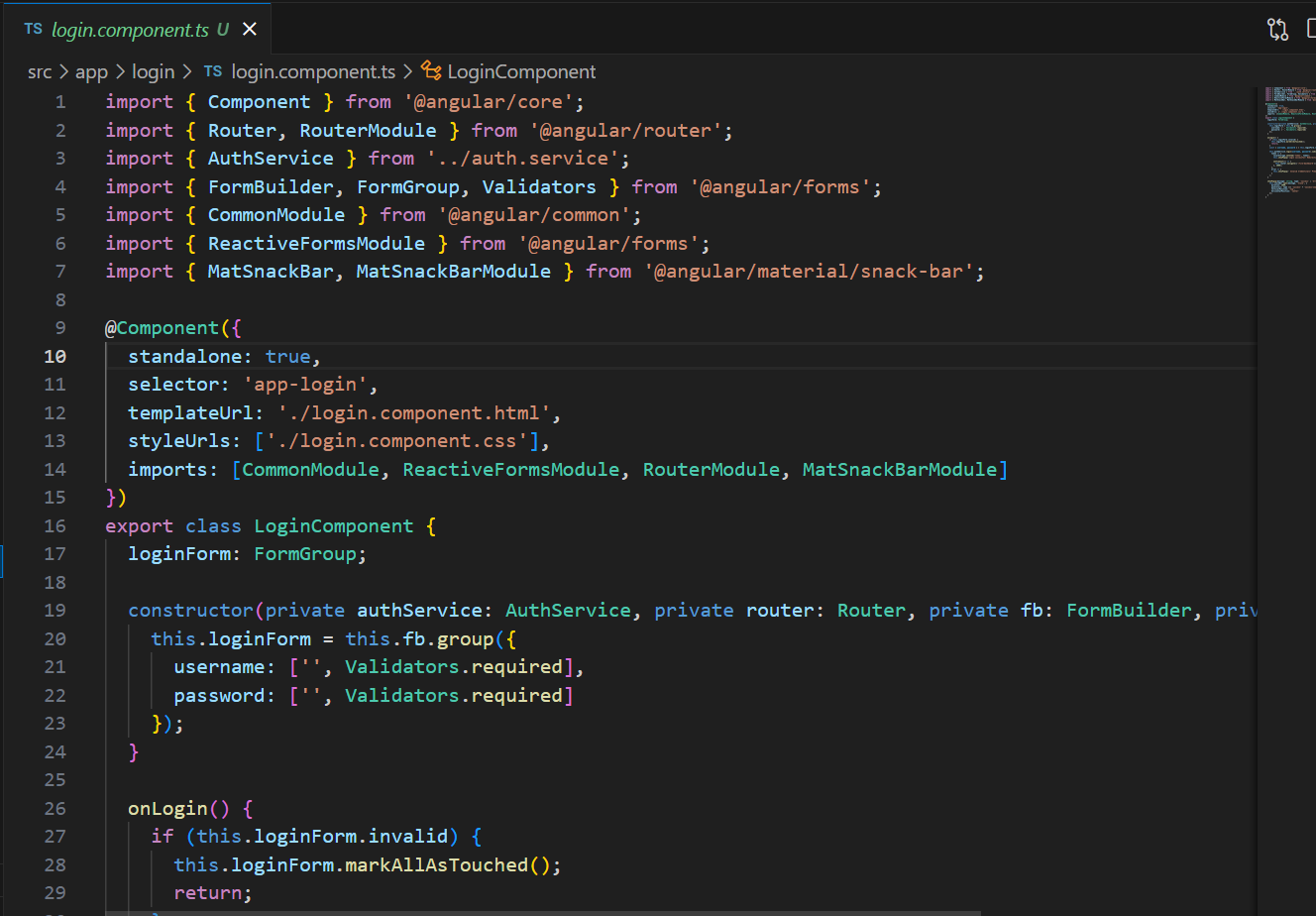
It connects with the backend url and validates the user credentials provided by user with the details from the database and is used to validate and print the token in the developer console.

Then two components “login” and “register” were created using the command “**ng generate component component-name** Every component gives us four classes named:

* component-name.component.css
* component-name.component.html
* component-name.component.spec.ts
* component-name.component.ts

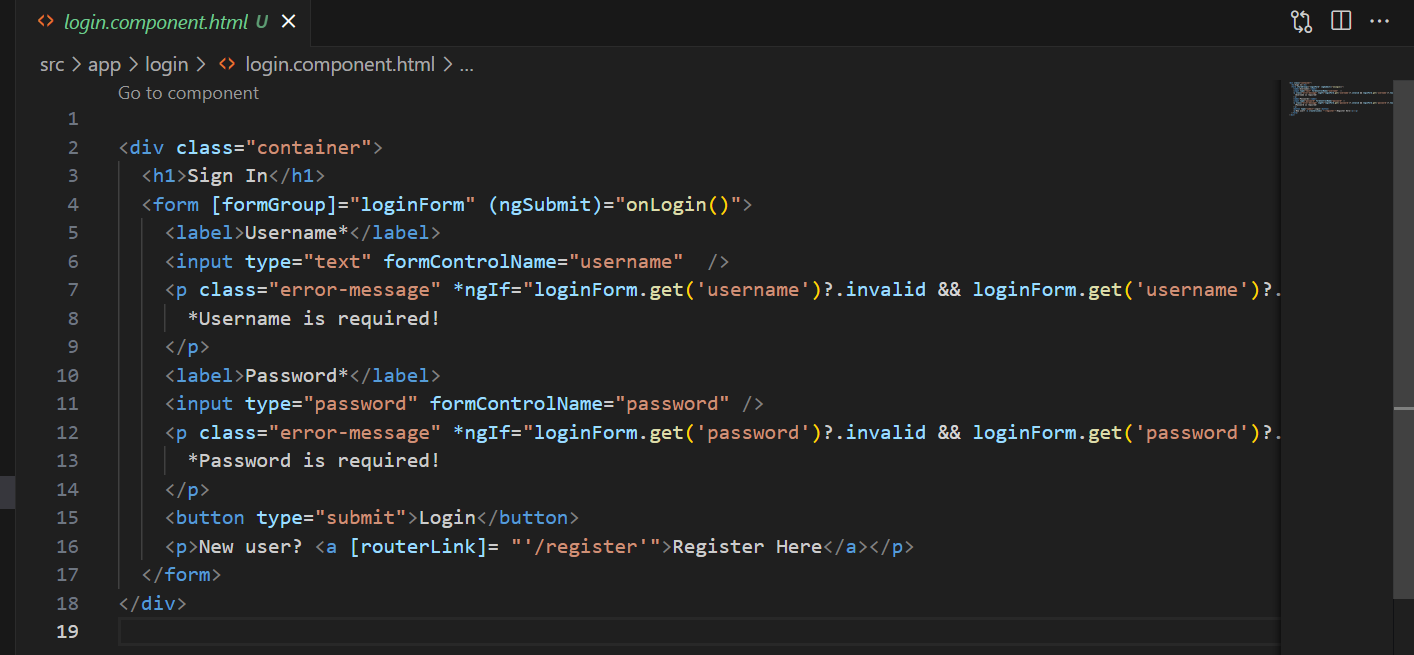
The component-name.component.ts is used to validate the logic from the front end. i.e., if any field is not touched and it is required filed then we can write to throw errors in the component.ts file.

The backend validation logic from the AuthService was included in the login and register components and the login component.ts looks like:



**Fig:** *login.component.ts*

Then the raw page containing the header, paragraphs, button, fields to display in the browser are created in the html page. The html tags like <h1>, <P>, <label> etc. are used to generate a raw html page and html page for login looks like the below:



**Fig:** *html page for login.*

The same has been done with the registration page. The same AuthService is used in registration page also to create a similar registration page as login page.

Now, five different components along with services named “file-create”, “file-details”, “file-search” ,“batch-runner” and “status-update” are created to perform different tasks of backend having different paths.

The file-create component is used to create a file and save it in the database, it consists of filename, status(a dialog) and record count as input fields and button to submit the user responses.

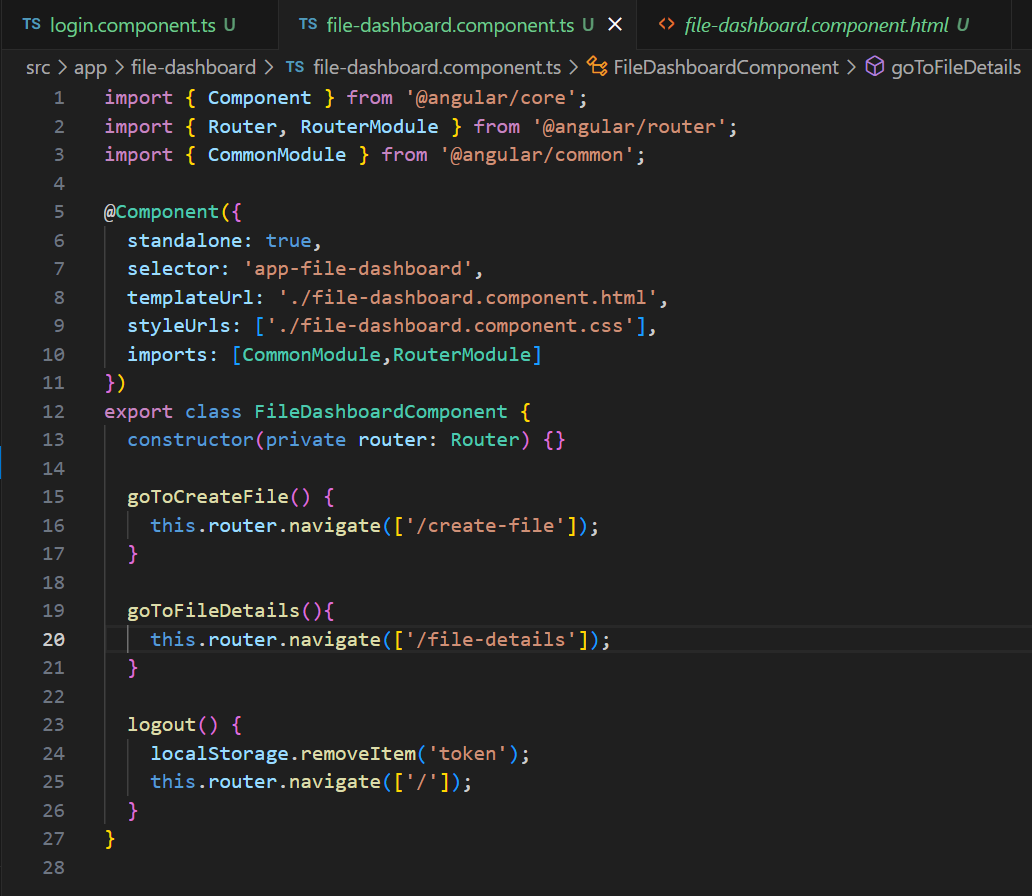
The file-details component is used to get the file details when the user/admin gives the file id. The details provided are File id, name, status, load date etc.

The file-search component is used to represent the Search criteria of the backend where the user provides name and status of the file then the details of file will be visible for the user.

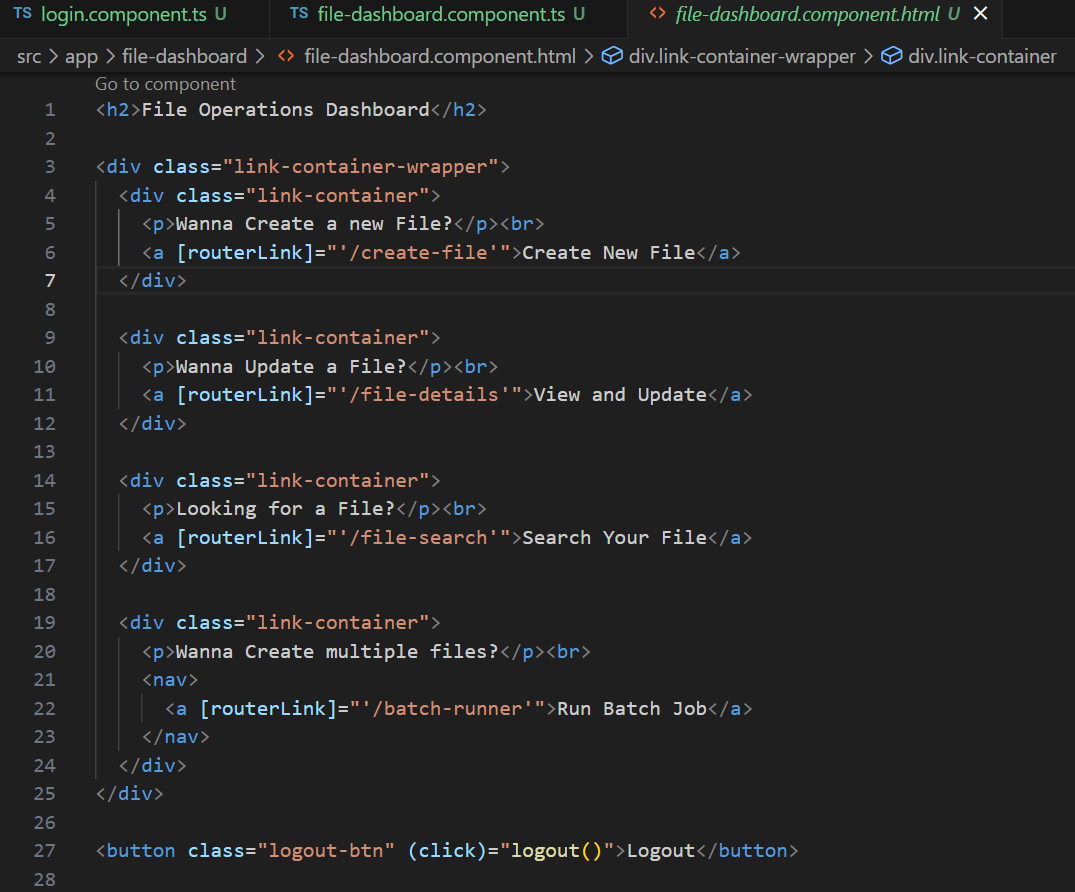
The batch-runner component consists of only one button to trigger the batch job and the data present in the file i.e. in application.properties will be uploaded into the database.

The status-update component is used to update the status of existing file, the option for updating the status is included in the file-details component. It consists of a field as dialog to get the new status from user/admin and a button to submit.

To go to all links, I have created a dashboard where links for all components/pages are placed

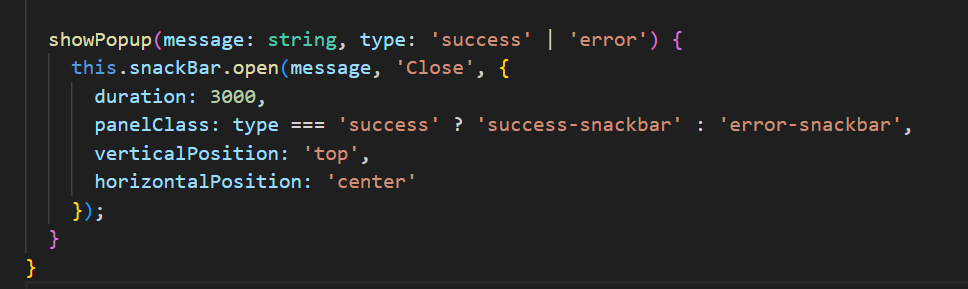


**Fig:** *Dashboard component.ts*



**Fig:** *Dashboard component.html*

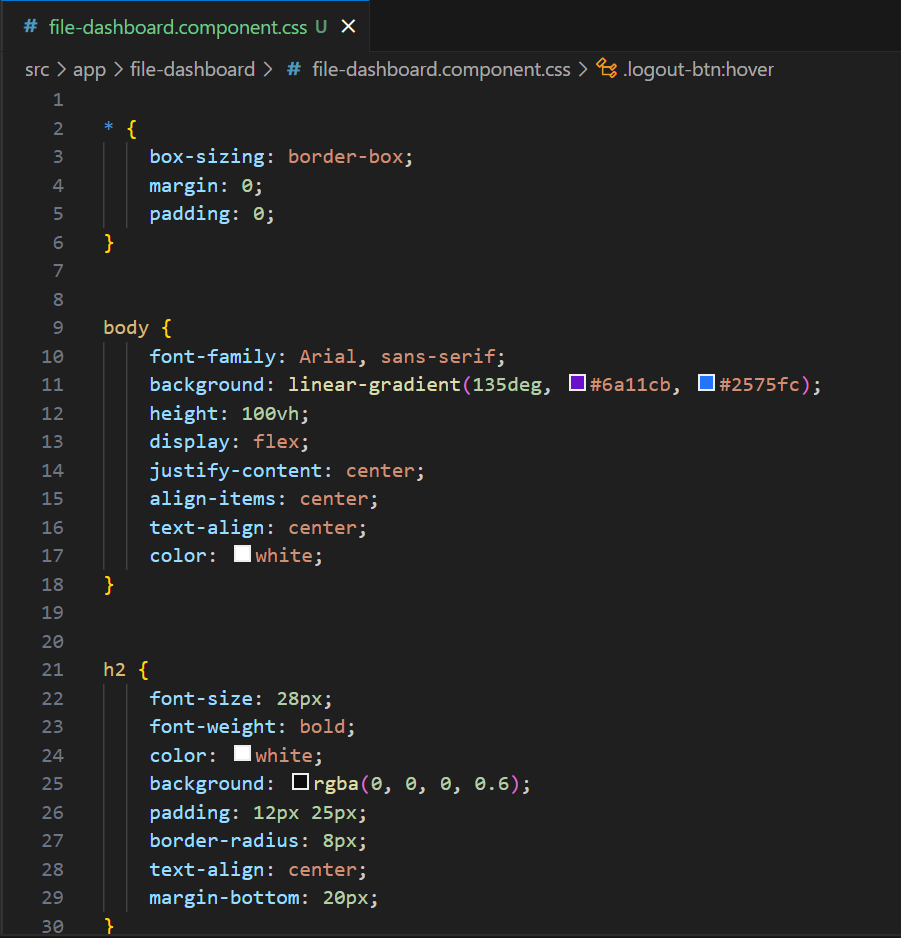
Another important task was to show successful messages and errors. Instead of showing alert messages, I wanted to make them more attractive. So, I have used **MatSnackModule** from @angular/material/snack-bar which has showPopup() method that displays the message as a floating popup. The method looks like:



**Fig:** *showPopup() method.*

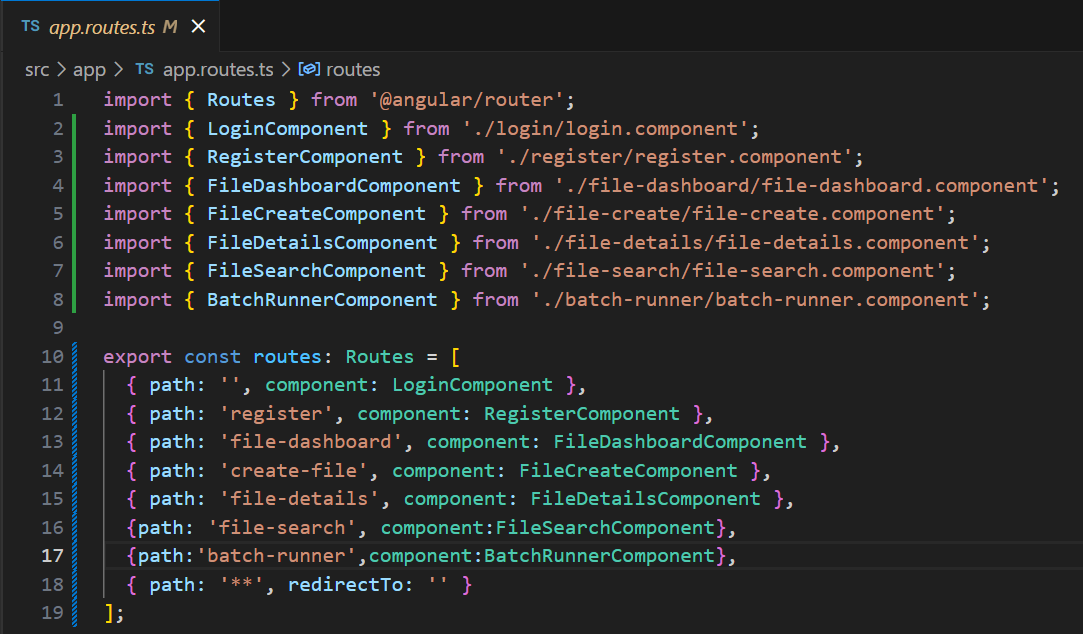
The above code makes the message display for 3 seconds at the top middle of the page in the browser.

To make the browser pages more attractive, styles have been added in the CSS file. One of the css files looks like:



**Fig:** *Css file of Dashboard*

The Front end will be running on localhost itself but on different port number, 4200. To make the pages route correctly, in app.route.ts, all routes have been declared and the app.route.ts look like:



**Fig:** *app.route.ts file to specify paths*

Here path for each component is declared clearly. For example, to open the dashboard, the user needs to navigate to the link- [**http://localhost:4200/dashboard**](http://localhost:4200/dashboard)**.**

To run the frontend in localhost on port number 4200, we need to give the command- **ng serve** in the terminal of vs code and it gives something like this:

A screenshot of a computer

AI-generated content may be incorrect.

**Fig:** *Terminal of the vs code*

To go to the link, click on the link with the ctrl being pressed. When the user gets to the link, it automatically brings the user to the login page which looks like:

A screenshot of a login form

AI-generated content may be incorrect.

**Fig:** *Login page*

If wrong credentials are provided, then a floating popup message will be seen like this:

A screenshot of a login box

AI-generated content may be incorrect.

**Fig:** *floating popup message*

After successful login with correct credentials, the page will be redirected to the dashboard automatically within 2 seconds and the dashboard looks like:

A screenshot of a computer

AI-generated content may be incorrect.

**Fig:** *File Dashboard with different operations*

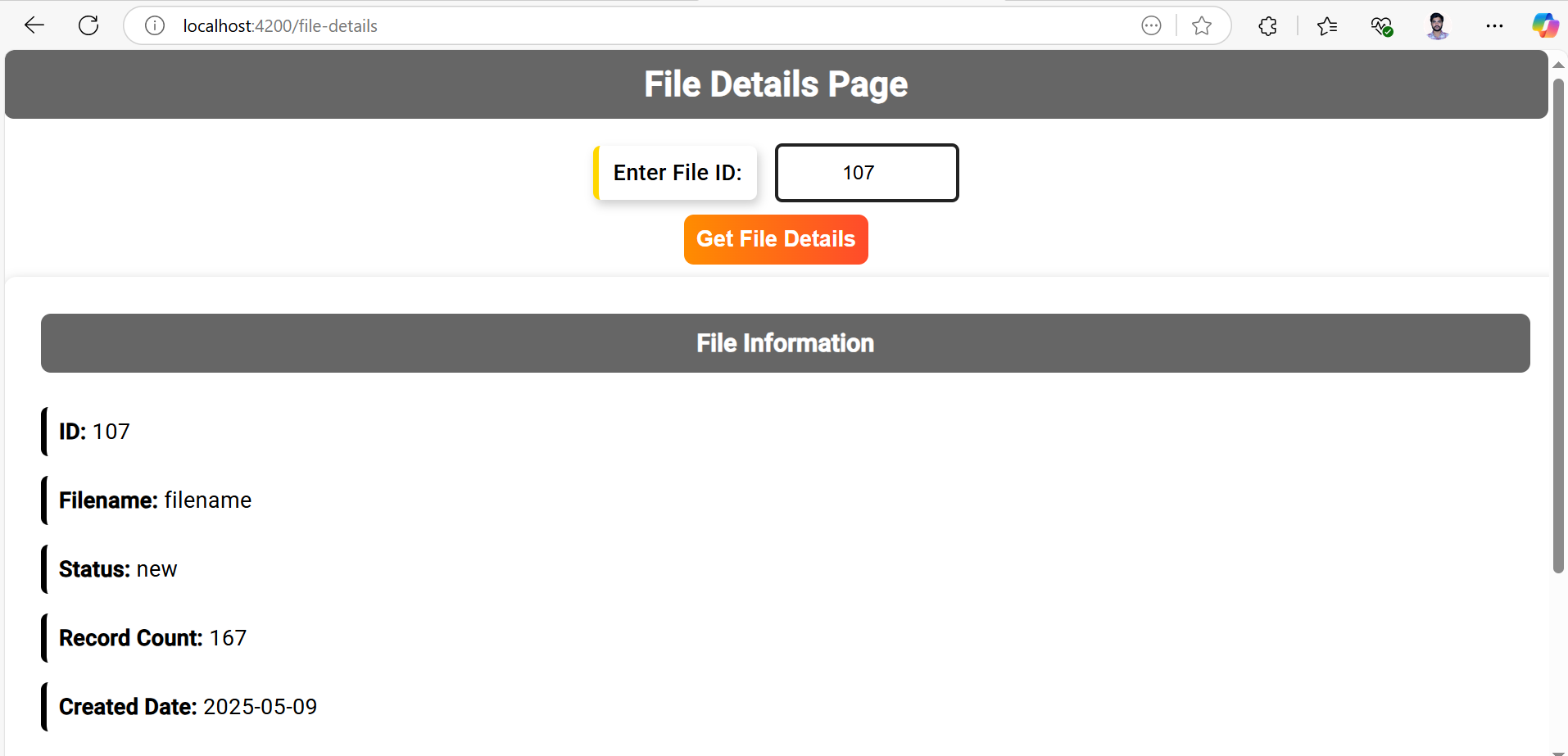
When the user clicks on create new file link, it will get redirected to the page where the user can create a new file by giving the details of file. After successful creation of new file, it looks like:

A screenshot of a computer

AI-generated content may be incorrect.

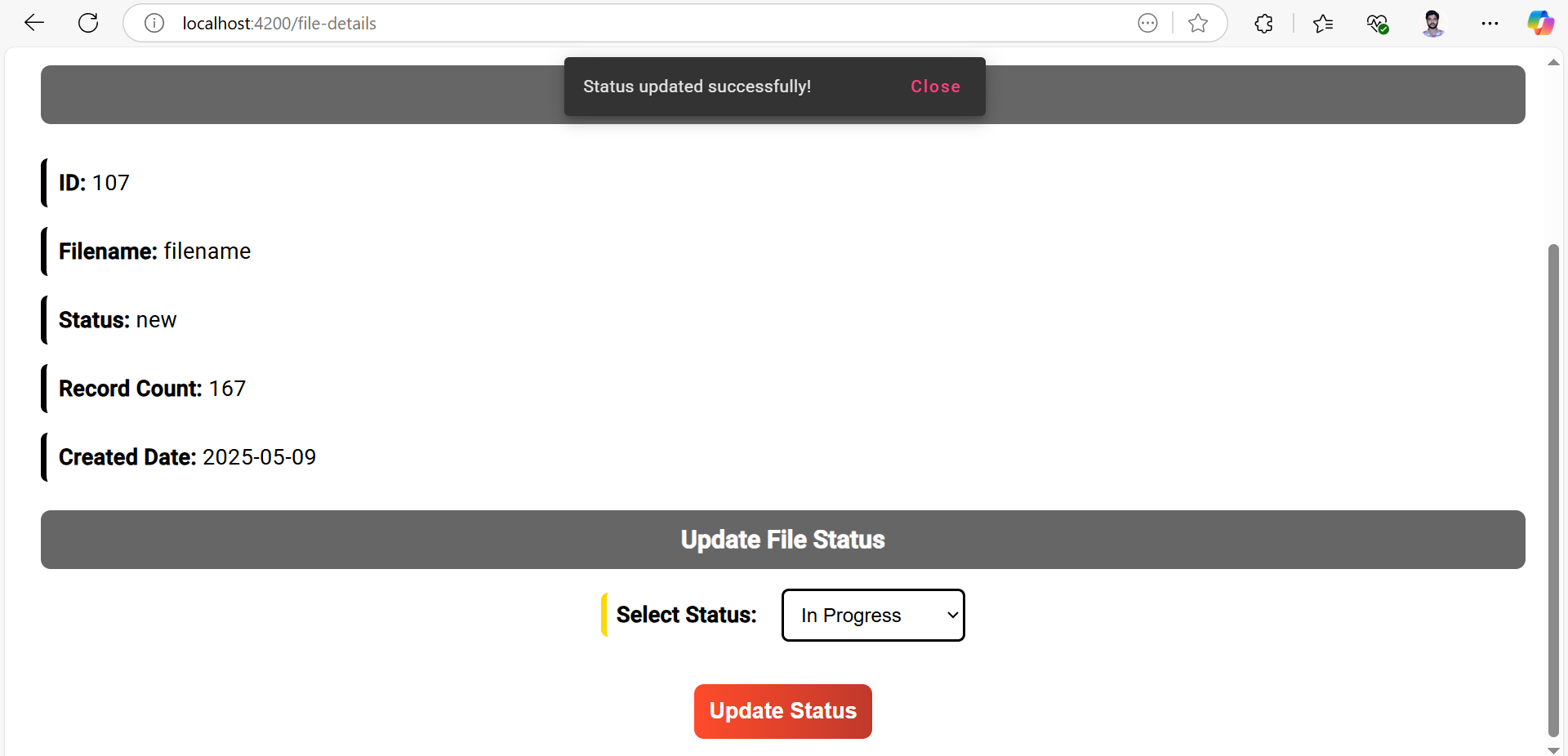
**Fig:** *File Creation page and success popup message*

After the successful creation of a new file then it will again be redirected to the dashboard. Now, the second view and update will get the user to the page where the user can view file details with file id and able to change the status of the file.



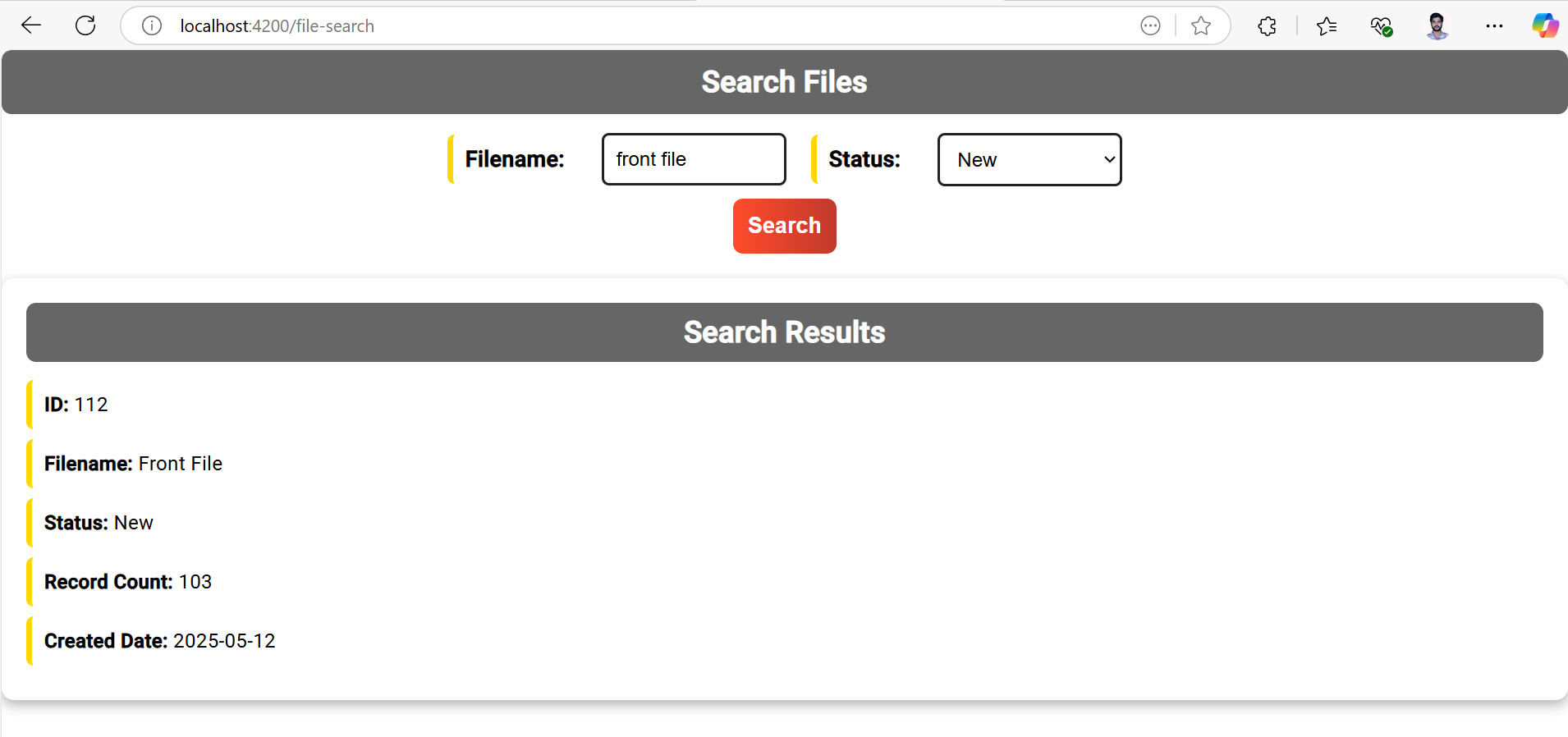
**Fig:** *Viewing details of file with file id 107*

The user can update the status of the file in the same page below the File information. And it looks like:



The file status will be updated successfully after clicking the update status button.

The user who is unaware of file id can search for the file using the file name and status of the file in the search your file page.



**Fig:** *Search by File name and status.*

This is how the user can search for details of a particular file by giving the name and status of the file.

The Run Batch Job brings the user to a page from where the user can run a batch job and can insert multiple files at a time by just clicking the Run Batch Job button and the interface looks like:

A screenshot of a computer

AI-generated content may be incorrect.

**Fig:** *Batch Job Runner Interface*

The application created may not be that helpful as it doesn’t store any original files but can keep track of the files stored somewhere else.

It can be modified to keep tracking the books in a library/bookstore with parameters like book name, row, copies, price etc.

**Thanks & Regards**

**Mr. Miriyala RaviBabu** sir and Team

|  |  |
| --- | --- |
| **Name of the Session head** | **List of contents** |
| Shyam | Basics of UI designing and Live scrum session |
| Chaitanya and Bhargav | Orientation session |
| Geeth Kiran | Basics of Backend development |
| Devendra | Implementation of Spring Security |
| Padmaja Rani | Basics of Java |
| Sai Kumar | Agile and Waterfall methodology |
| Bhargav and Sree Keerthi | Overview on testing |
| Nagarjuna | Modifications in the project |
| Sushmita | Testing tools |
| Suresh Babu | Introduction |