**PERSONAL DETAILS**

NAME: NANSUBUGA JOYCE EUZEBIA

Email : [njeuzebia@gmail.com](mailto:njeuzebia@gmail.com)

Phone: 0758720304/0780365538

Repository Url: <https://github.com/euzebia/accounts_manager>

**BRIEF DESCRIPTION OF THE DESIGN AND EVIDENCE IN SCREEN SHOTS**

**Software description**

**Software name: accounts\_manager**

A real-world use case considered for implementation is an off-the-shelf software developed by a third-party software vendor/system owner, that is used in banks to help bank staff (personal banker) create bank accounts, modify accounts based on customer preferences and managing other account details.

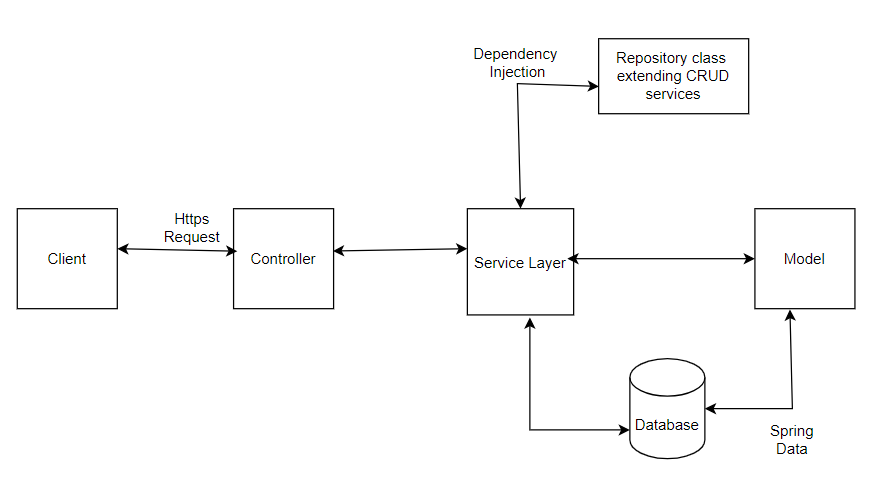
The software assumes that three user types interact with it i.e., bank staff, bank customers and system owners.

The system owners over see all activities taking place in the system, bank staff are responsible for managing customer accounts and for customers they may only interact with the system in scenarios when they want to view their personal information.

The document is broken down into various sections each detailing various software development areas where handled; some of them include but not limited to database design, system architecture, security, application performance, etc..

1. **Api architecture**

The api design follows a layered architecture as illustrated below.



Explanation;

1. A client makes an http request i.e GET,POST,etc.. and is forwarded to the controller.
2. The controller maps the request and calls the service layer logic.
3. In the service layer, data validation, database calls and other business logic take place from there
4. The database acts as a central repository for the data used by the api/application.
5. **Database Design**
   1. Data normalization has been used to eliminate data redundancy as some properties are shared by more than one table.
   2. A primary key has been added to each table to uniquely identify each record in the table.
   3. Stored procedures and views have been to retrieve data from related tables in order to reduce on the traffic between the api and the db. Instead of sending multiple lengthy SQL statements, the api only sends sends the name and parameters of the stored procedure and details are retrieved once found
   4. Use of indices have been on columns like username and email address column in the accounts tables as it is one of the tables on which many search queries are done to retrieve accounts, as data grows, search queries tend to slow down.

**Illustration**

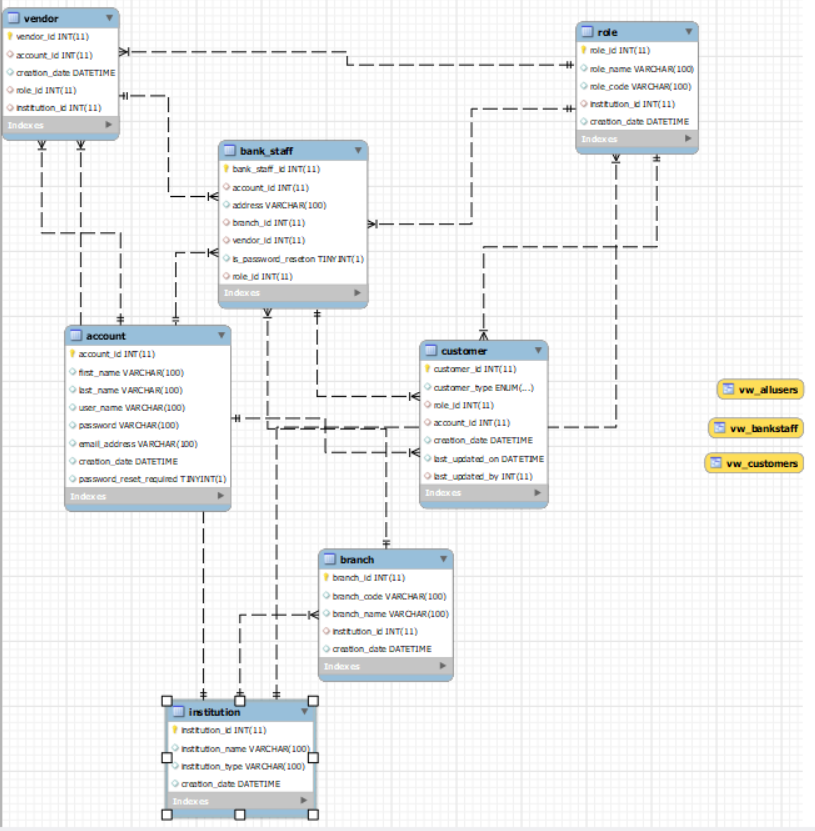
A screenshot of a computer

Description automatically generated

* 1. User\_name and email are unique as they part of details used to uniquely identify a user in the system.
  2. Foreign keys have been used to link tables that are related to each other.
  3. Different types of users ie Type A,Type B,Type C have been categoried as bank staff,Customer and Vendor respectively.
  4. The institution in which these users belong have been categorized by type ie the institution in which bank staff and Customer belong has been categorized as bank, type C as Third party software vendor (vendor of the software).
  5. Type C user(vendor) creates a bank staff and assigns them a under the institution bank.
  6. A bank staff creates customer accounts and modifies customers' info based on customer preferences.
  7. Customers have been categorized according to customer category ie corporate and ordinary user.
  8. Institution bank has a branch attached to it.
  9. A vendor creates a bank staff who is responsible for creating customer accounts.
  10. Database views have been used to group data related to the various user types

The entity relationship diagram showing how different entities talk to each other can be seen below;

**ENTITY RELATIONSHIP DIAGRAM**



1. **Security**

* Api security

The api has basic authentication there by unauthorized are prevented from accessing the endpoint;

Authentication type : Basic

**Username**: api\_user

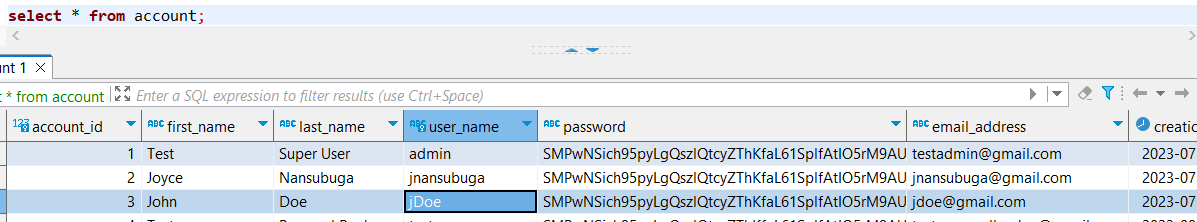
**Password**: t5p4krGSJyil2hWB

* Validating post requests to the endpoint

Measures have been put to handle xss and csrf vulnerabilities as seen below;

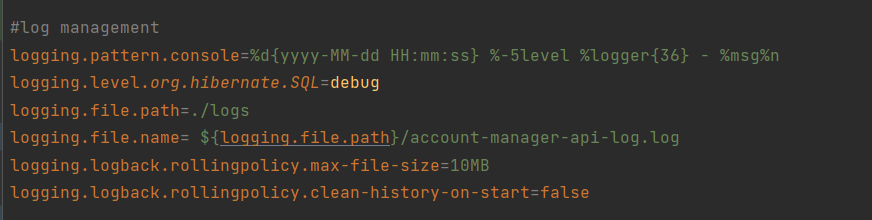


* Encyption of user passwords while data is at rest,this prevents malicious users from identifying user passwords;



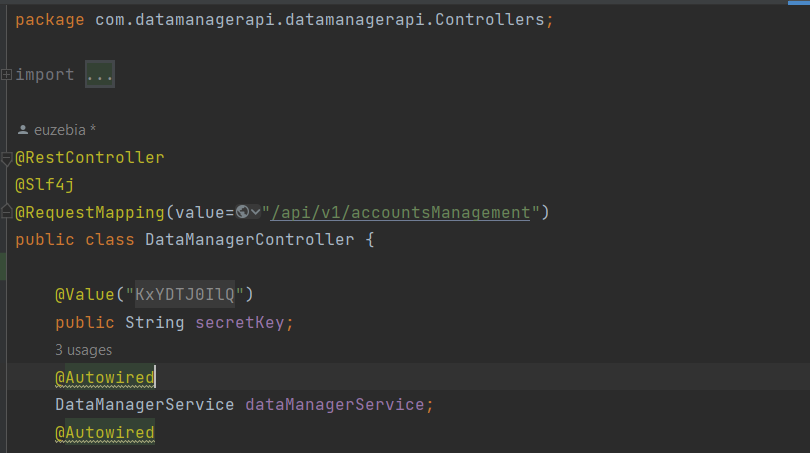
1. Log Management

This helps in easy monitoring of the application performance i.e if the application is catching any errors during execution, errors can easily be identified as they are logged on file during execution. As seen below,the file log reached 10MB,it is zipped,rolled and a new one created basing on configurations below;



1. Api versioning

It helps to ensure stability and reliability,if you don’t properly version apis this can cause a big negative effect on products already in production. With this developers, can easily maintain and make additions to existing software without breaking changes.



1. Dependency injection

It helps to ensure loose coupling between classes. @Autowired annotation injects the DataAccess class into the ManagerDataService as shown below;

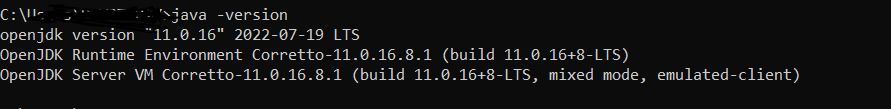
A screen shot of a computer program

Description automatically generated

**Running the developed application;**

**Pre-requisites**

* Mysql and php installed (mysql version 5.1.3,PHP Version: 7.4.28)
* Java installed,to confirm if all is set run the command below;



* After you have successfully closed the application, navigate to the \accounts\_manager\build\libs to locate the .jar file to run;
* On your machine(windows) for our example,navigate to where the jdk was installed and enter the commands below to start the application;

java -jar “the file path to .jar on your machine”



Here the application has started running listening for traffic on port 8006 as seen above,we can start the tests.

**Choosing a client to test with**

* For this case HTTPie is acting as our client. It can downloaded from <https://httpie.io/download>.

**Test scenarios**

* Login: All users access the application from one central point

User Type;

1. Vendor ( Username: admin, Password: admin2020)

A screenshot of a computer

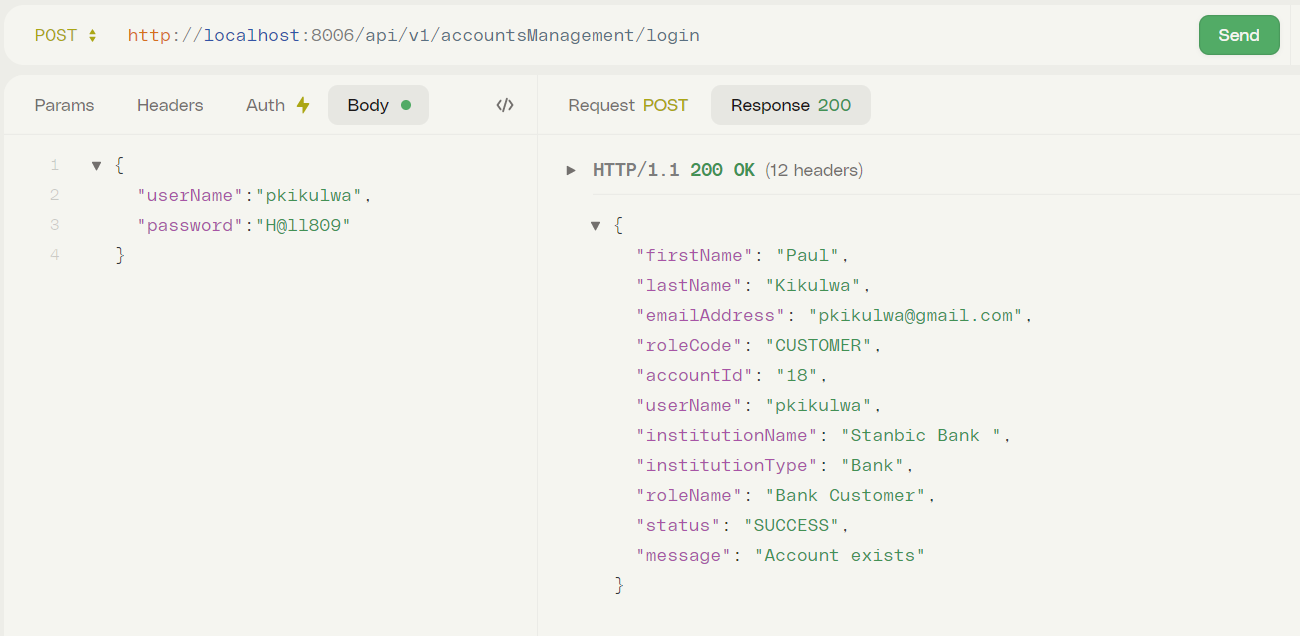
Description automatically generated

1. Bank staff (username:test111,password: cat2873)

A screenshot of a computer

Description automatically generated

1. Customer



All the three user types have been able to logon from one central point.

* Save Bank Staff Details



* Save Customer Details

A screenshot of a computer

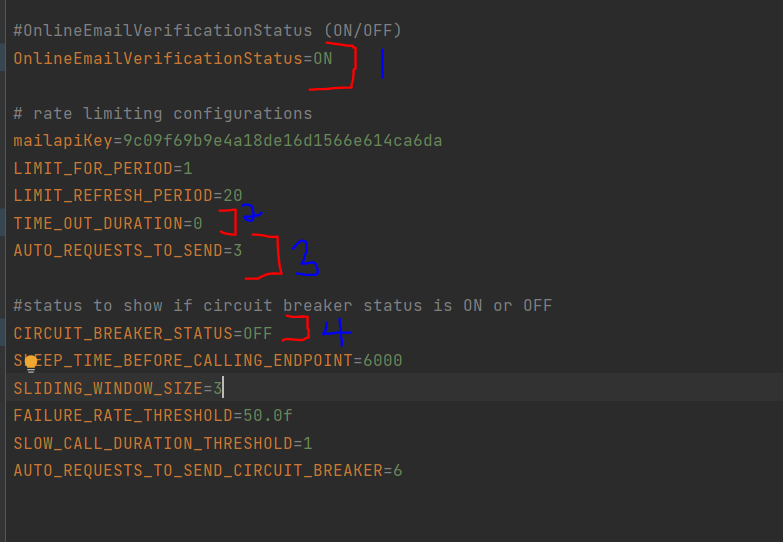
Description automatically generated

* Save Bank Staff Details with Rate Limiting Enabled

To illustrate how rate limiting works, a configuration was done in the service on validating user input we check to confirm if the supplied email is valid, for this case the service can either choose to validate the email locally using the regex expression or online using a third party endpoint that provides online email verification. This implementation focuses on client side rate limiting as the emails are issued basing on a given service plan. This example illustrates how to limit the number of requests sent to the third party service provider or saving the cost that would be incurred if the service is elasting. Here when a give request limit is exceeded,the requests are throttled thus preventing them from being sent out.

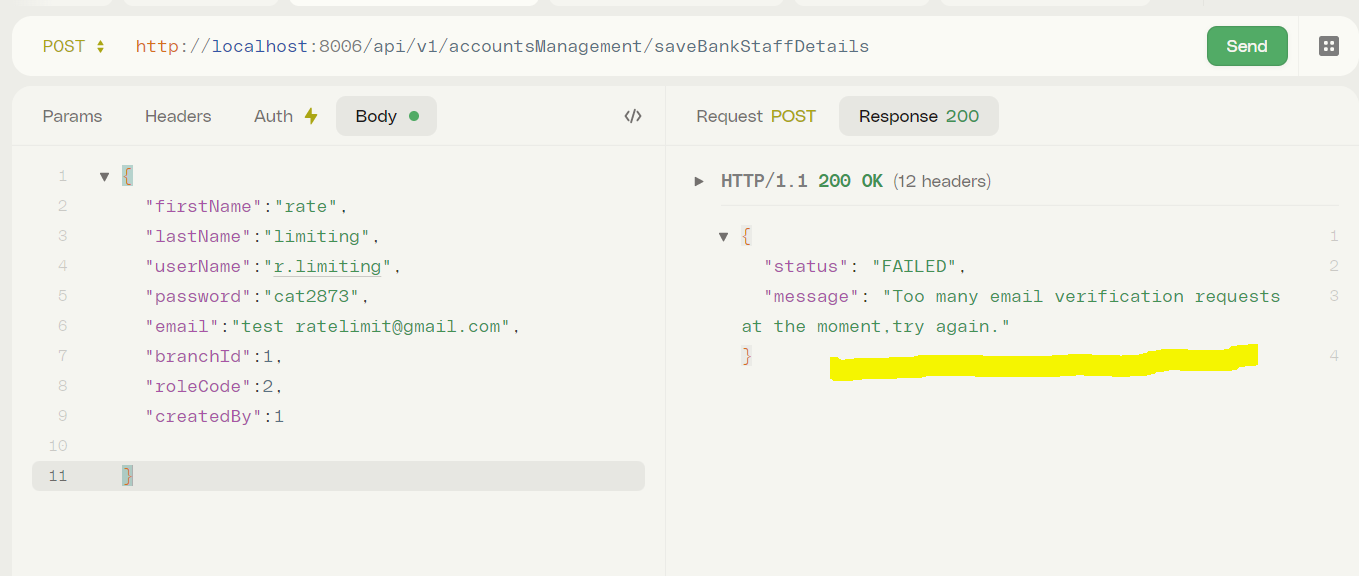
**Configs to enable rate limiting to take place.**

* **Set the email verification service to be online i.e validate at the third party**

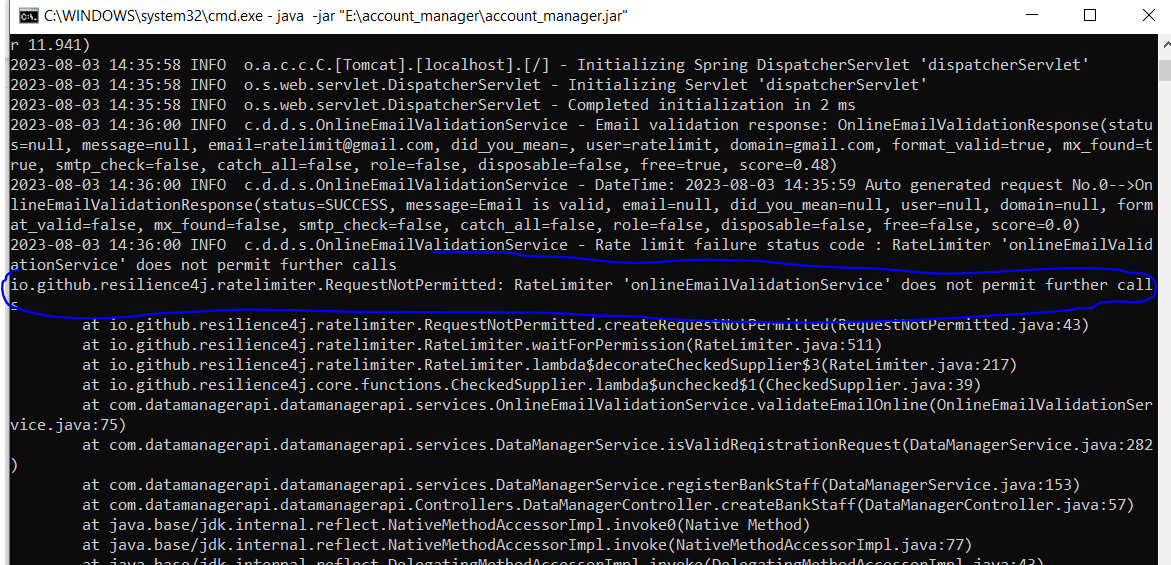


Settings labelled 1,2,3 and 4 should be as shown above, so when a user accesses the client(httpie) and attempts to register a new bank staff, the service checks if email verification is being done online or offline. In this case it is online,we have limited to number of requests acceptable period to 1,2 says if a request comes and a thread is not able to acquire permission don’t wait just stop execution and return error back to client,internally the service automatically generates 3 more requests to send along with this initiated from the client. Here the all together 4 requests will be sent and since configuration 2 is set to 0, an exception of request not permitted will be thrown and a failure message will be sent to the client. If configuration 2 is not 0, the thread will wait and until it gets permission to process the request;

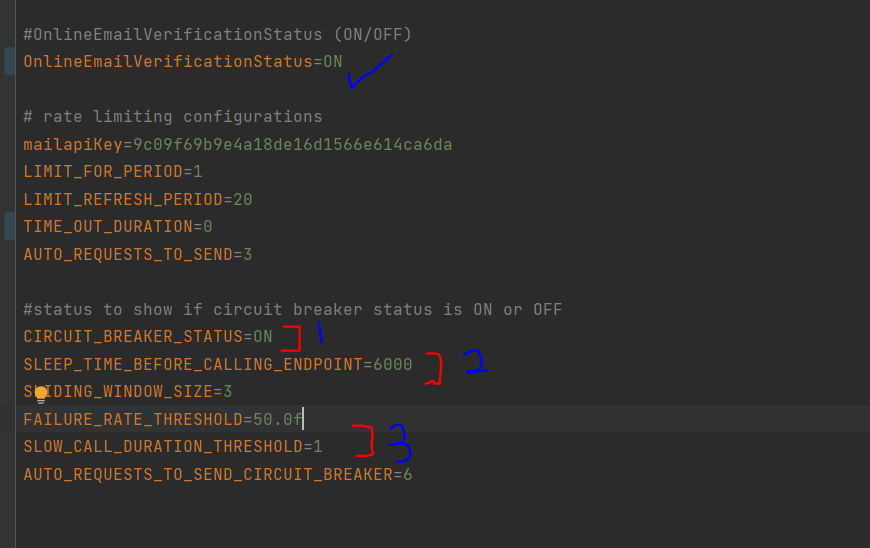
**Error returned on client side;**



Exception caught in the application

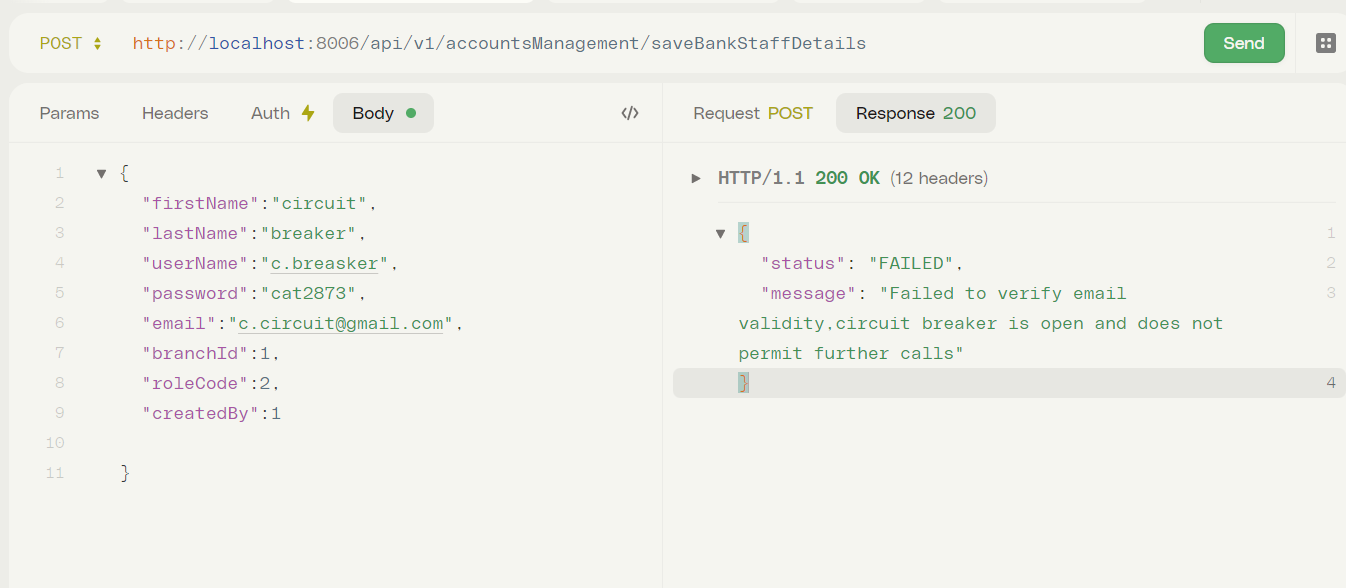


* Save Bank Staff Details with Circuit Breaker Enabled

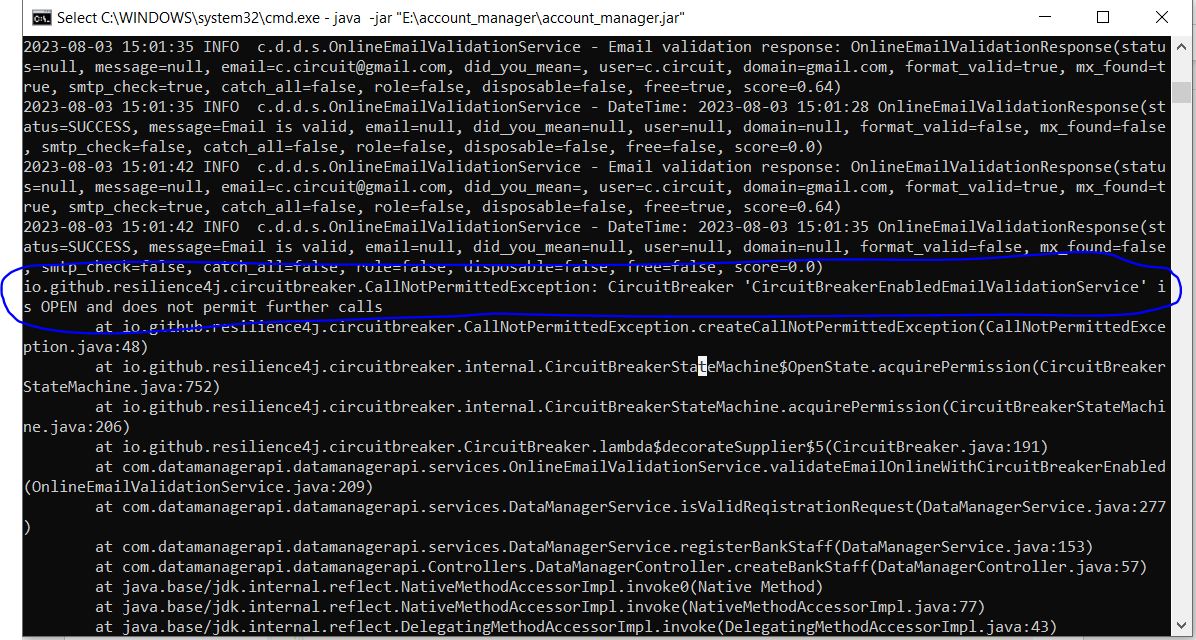


If the email verification status is set to on, the service checks if circuit breaker status is set to on, if so then, a delay of 6 seconds is introduced (for each request a thread first sleeps for 6 seconds before calling the third party api)as seen labelled (2) in the image,

Circuit breaker status is set to On as seen in (1),the time at which the service considers to be getting a slow response is set to 1 as seen in (3) but as explained earlier each requests first waits for 6 seconds before being sent out, The service also supplements on the request with 6 more request (AUTO\_REQUESTS\_TO\_SEND\_CIRCUIT\_BREAKER=6) here all requests are bound to fail and if the circuit breaker sees that the 3 requests (SLIDING\_WINDOW\_SIZE=3) previous calls were not successful, the failure rate threshold will be reached therefore the circuit will open and an error will be returned to the client as seen;

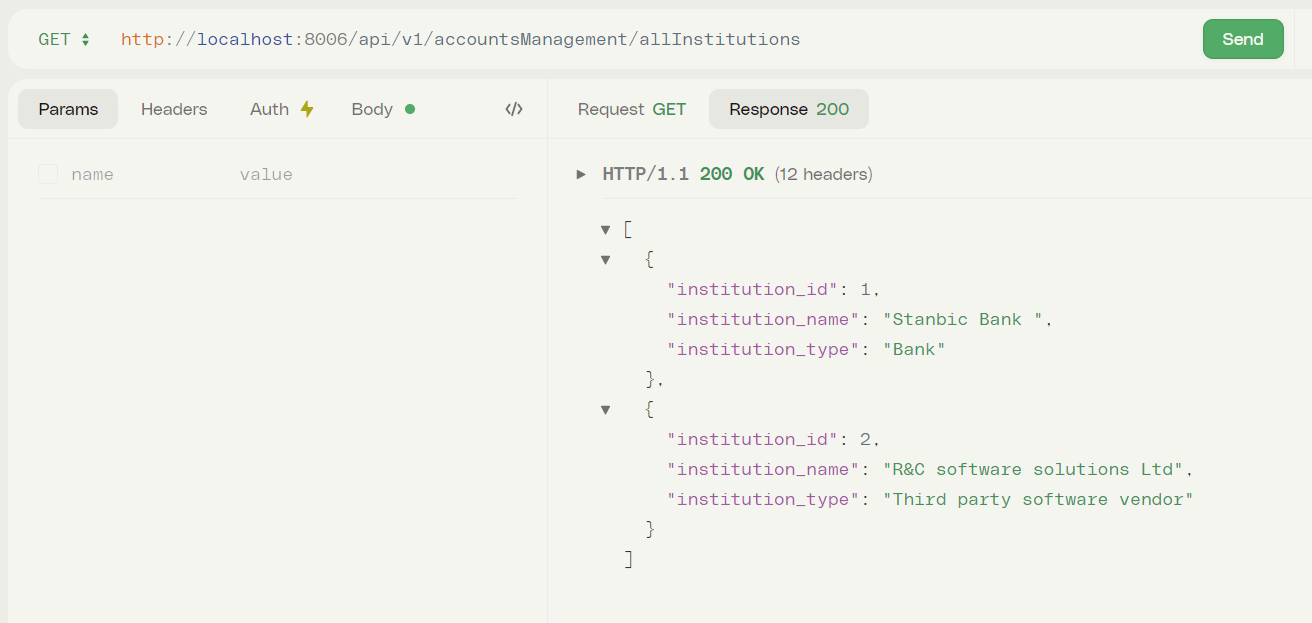


**Error in application**



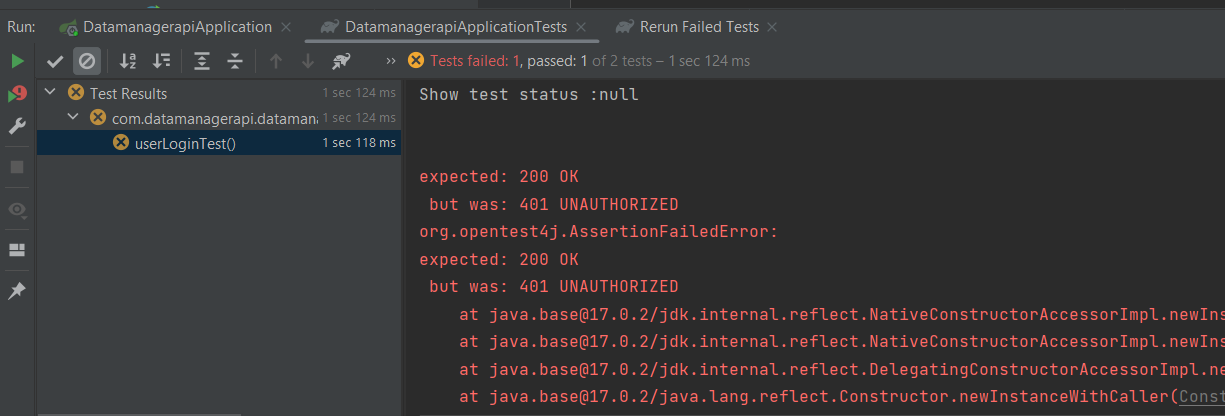
* Retrieval of cached institution details

For Hazelcast’s support for data caching,it stores frequently data in memory such that it is easily available to services that readily need the data.

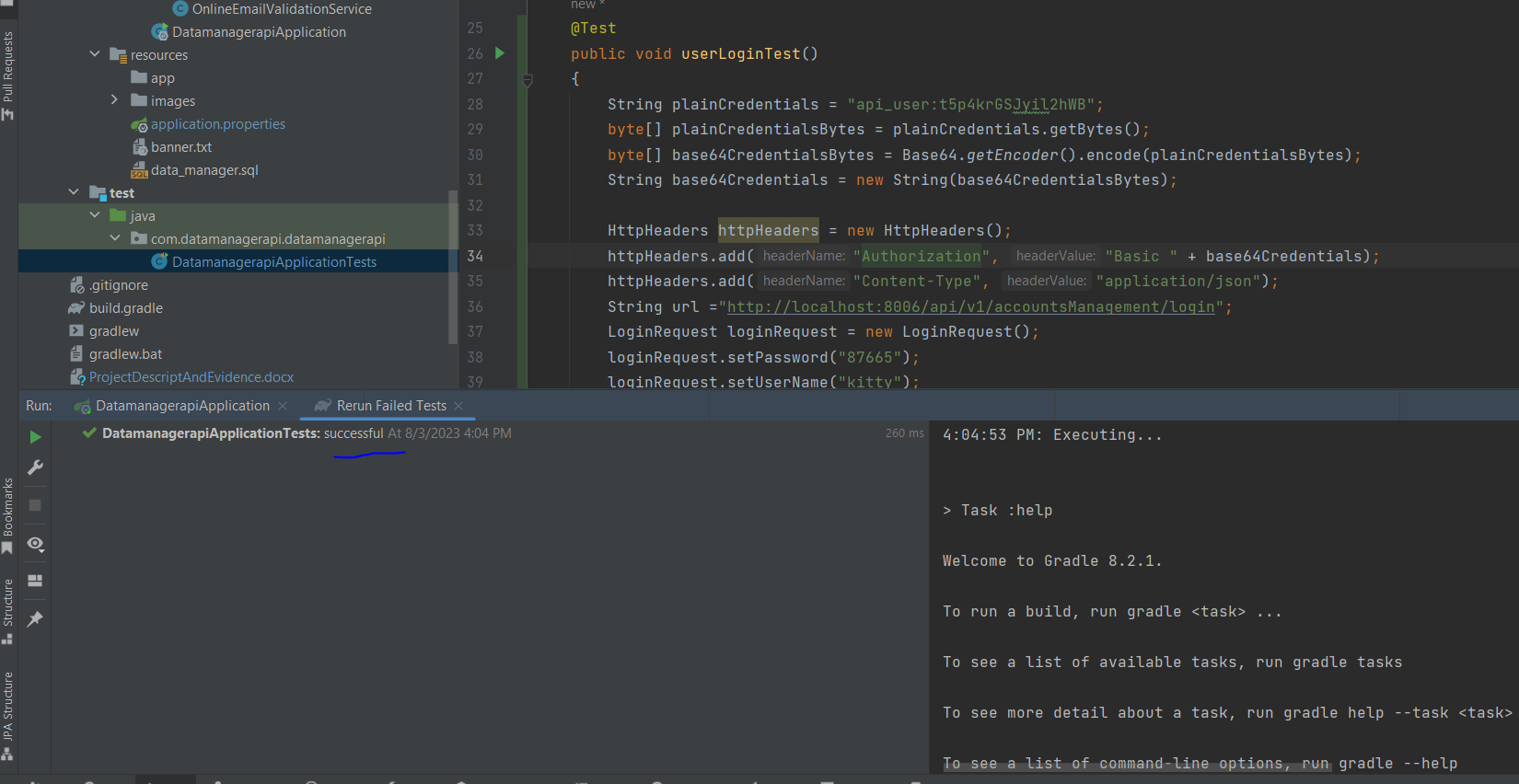


Configuration in code

* Test driven Development
* User login test without api credentials being passed, fails with 401 status code



* After supplied the api credentials in the header, the test passes



END