Problem Statement:

Creating a banking application that consists of multiple

microservices to handle different banking operations. The application will be designed to handle

testing, scalability, and observability.

Requirements:

The banking application should have the following simple features:

1. Account Creation: Users should be able to create a new account by providing their name,

email, phone number and password.

2. Balance Enquiry: Users should be able to check their account balance.

3. Deposit: Users should be able to deposit money into their account.

4. Withdrawal: Users should be able to withdraw money from their account.

5. Transaction History: Users should be able to view their transaction history.

6. Notifications: Users should receive notifications via email for every transaction made on

their account.

Solution:-

The Banking application consists of 3 services

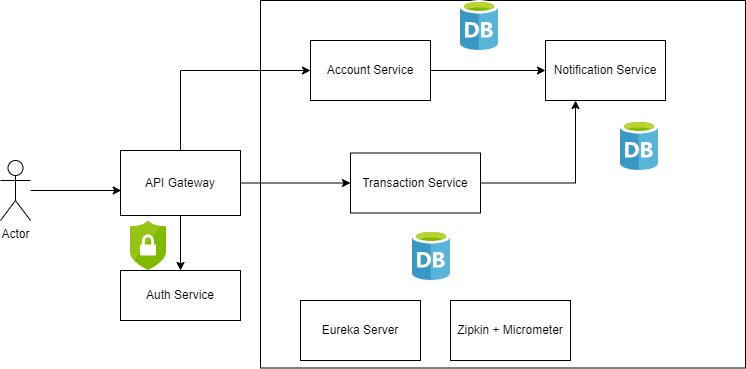
1. Account Service: This service will handle all account-related operations such as account

creation, deposit, withdrawal, and balance inquiry.

2. Transaction Service: This service will handle all transaction-related operations such as

transferring money between accounts and viewing transaction history.

3. Notification Service: This service will handle all email notifications.



Endpoints exposed by different services:-

1. Account Service **/api/accounts/**
2. @PostMapping(@RequestBody AccountRequest accountRequest)

Used for account creation which has all the user details like accountHoldername, username, email id, phone number

1. @GetMapping – used to retrieve details of all accounts stored in the database.
2. @GetMapping("/{accountNumber}")

This endpoint helps to retrieve account details of a specific account number

1. @PostMapping(“/withdraw/{accountNumber}) with withdrawAmount as RequestParam

Used to withdraw amount from the given account number

1. @PostMapping(“/deposit/{accountNumber}) with depositAmount as RequestParam

Used to deposit amount to the given account number.

1. @GetMapping(“/balance/{accountNumber})

Used for balance enquiry for a given account number.

1. Transaction Service **/api/transactions**
2. @PostMapping(@RequestBody TransactionRequest transactionRequest)

Used to create a transaction which consist of fromAccountNumber, toAccountNumber, accountNumber, timestamp.

1. @GetMapping

This endpoint helps us to retrieve all the transaction details

1. @GetMapping(/{txnId})

This endpoint is used to get transaction information for a specific transaction id.

1. @GetMapping(“/history/{accountNo})

This endpoint helps us to view the transaction history for a given account number.

1. Notification Service

Template Creation and Management related service endpoints (/api/templates):

These are meant to be used by admins to manage templates which hold the content for different types of notifications based on locale. The content of the notification will also hold placeholder for parameters to be replaced in the message.

1. @PostMapping(@RequestBody TemplateRequest templateRequest)

Used to create a template for a particular templateName and locale. The content of the template will consist of a formatted message containing multiple placeholders (parameters) which can be replaced with real values while sending out messages.

1. @GetMapping

This endpoint helps us to retrieve all the template details

1. @GetMapping(/{templateName})

(@PathVariable String templateName, @RequestParam String locale)

This endpoint is used to get template details for a specific template name and locale.

Notification related service endpoints (/api/notifications):

These endpoints are related to the notifications themselves.

1. @PostMapping(@RequestBody NotificationRequest notificationRequest)

This endpoint is meant to be used by other services to send out specific notifications based on a template name and locale. The actual values to be replaced into the placeholder parameters of the template’s contents will be passed in as a map while invoking this endpoint.

1. @GetMapping

This endpoint helps us to retrieve all the notifications that were sent out

1. @GetMapping("/{Id}")

This endpoint helps us to retrieve the notification that was sent out based on a specific notification ID.

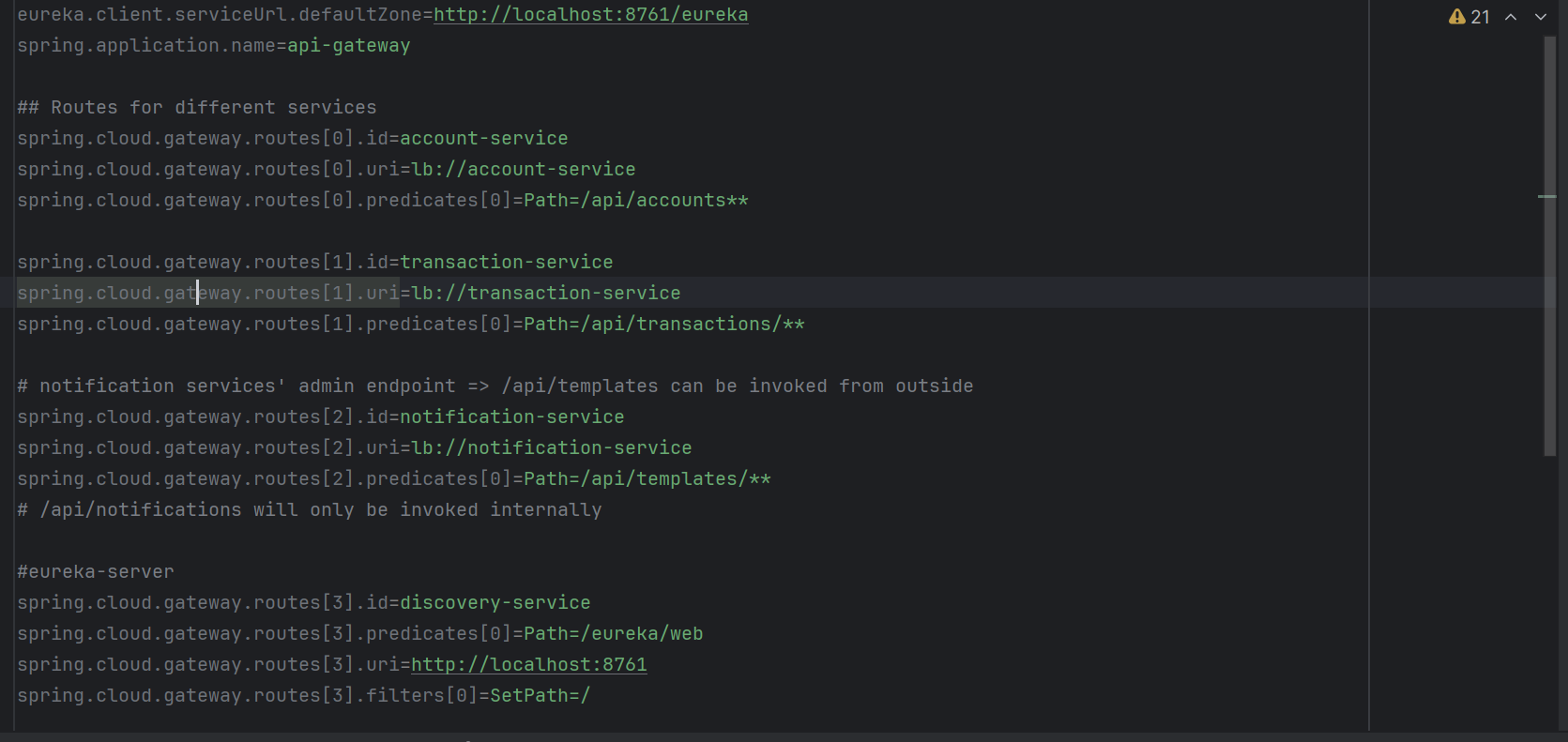
Similarly, we can have other endpoints implemented as needed to show the notifications in some UI (for example, as a “Notification History” tab against an account).

**Additional Components:**

API Gateway:-

The user communicates with either the Account or the transaction Service through the API Gateway. The API Gateway intercepts every request before forwarding it to various services. It has API routes, predicates and filters defined, based on which it forwards the request to the corresponding services.

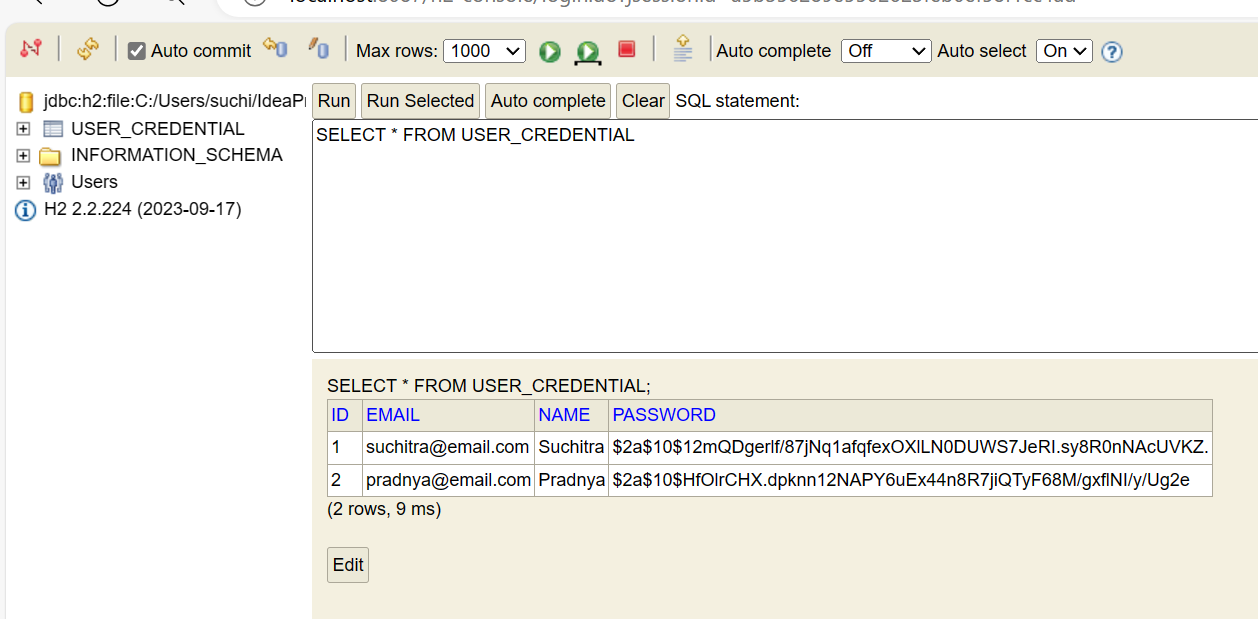
This way, all the services are accessible through a common <url:port> combination and centralized rules for security, rate limiting, client side load balancing, etc, can be implemented here.



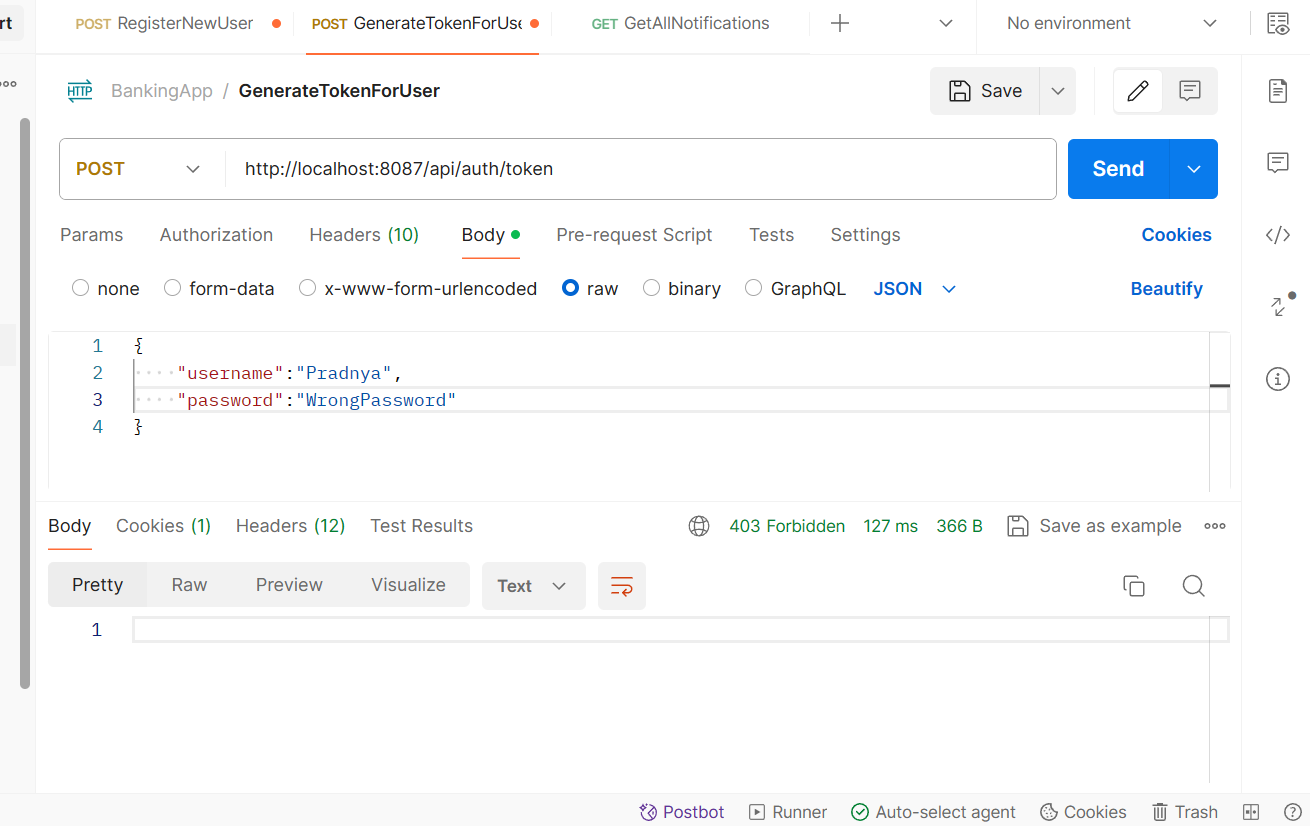
Auth Service (identity-service):

The API Gateway talks to the Auth Service for Authentication. The Auth service generates tokens for all Authenticated users and validates the tokens before forwarding the requests.

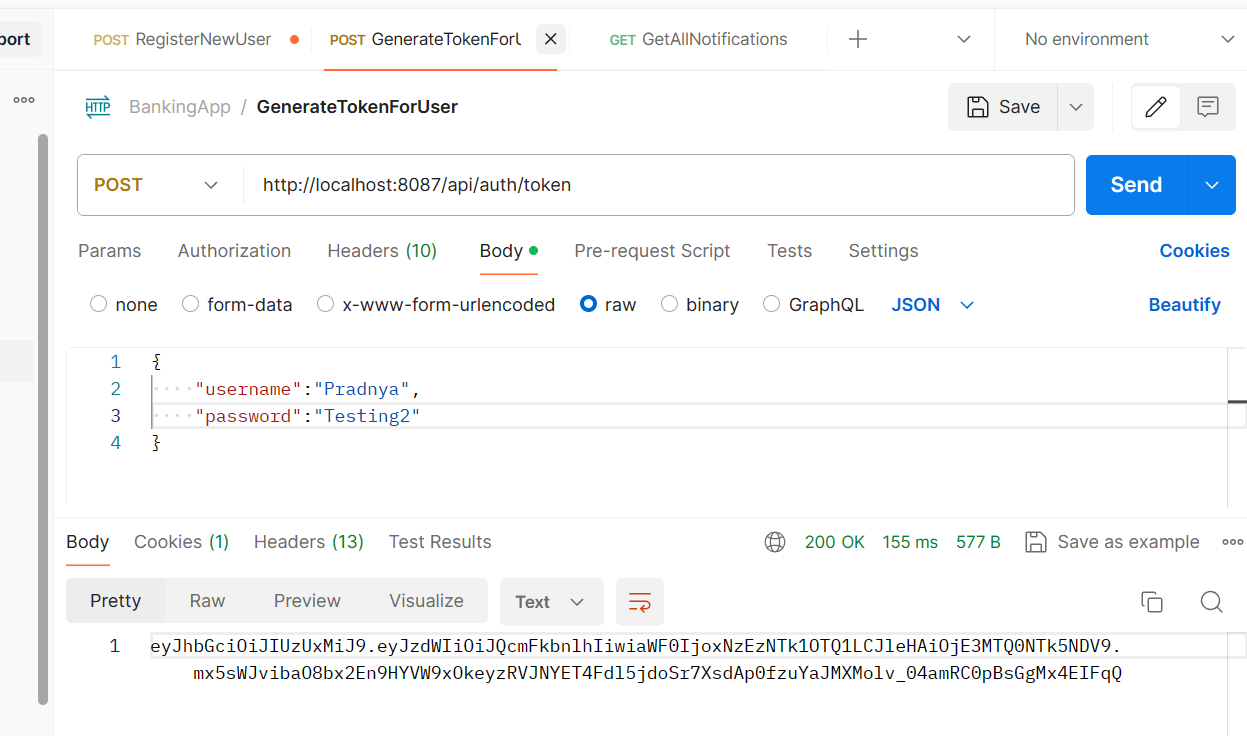
The user passwords are encrypted while storing in the database. Used by the identity-service microservice (which implements auth using JWT)

The users created are stored in USER\_CREDENTIAL table in the identity database.

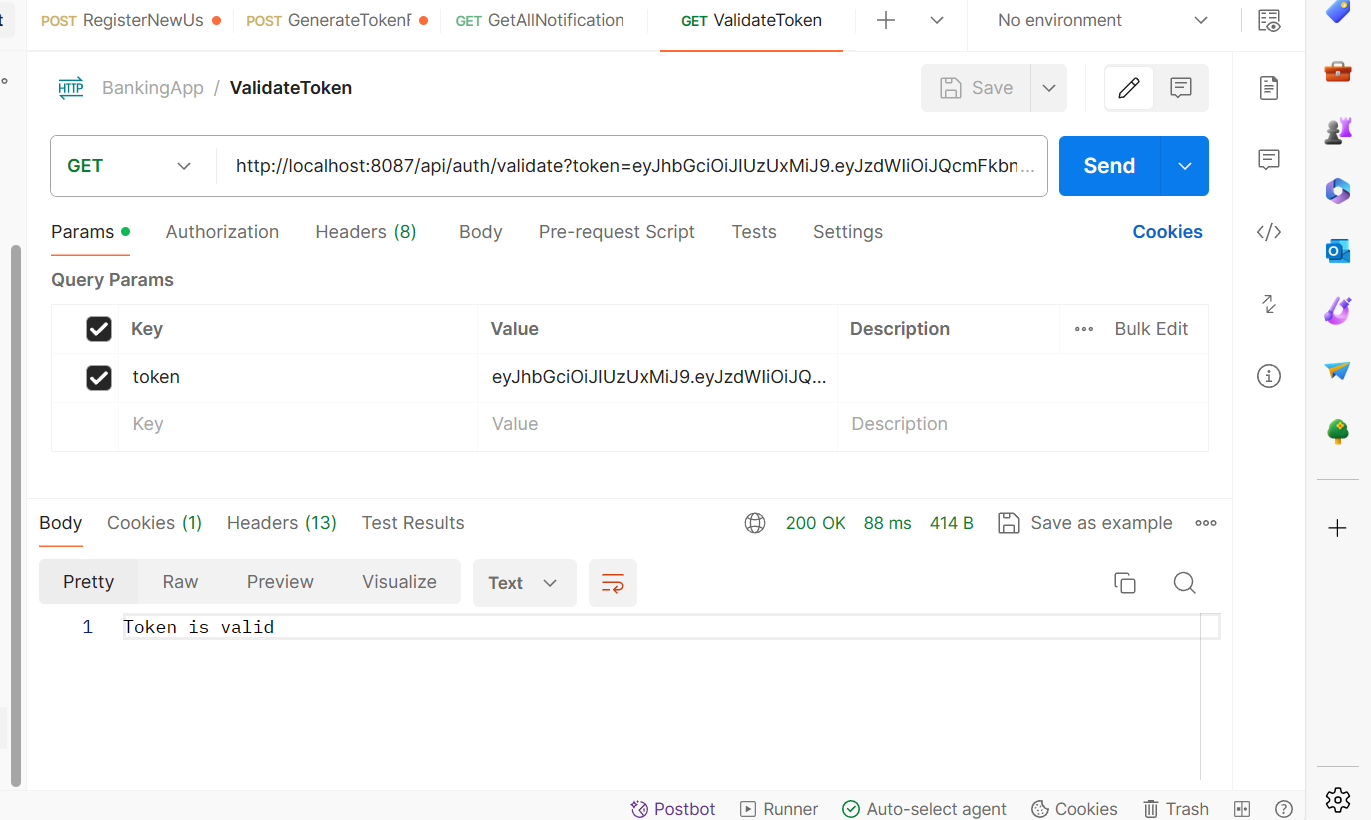
If GenerateTokenForUser endpoint is invoked with an invalid user, or with valid user but wrong password, a 403 Forbidden error gets thrown:



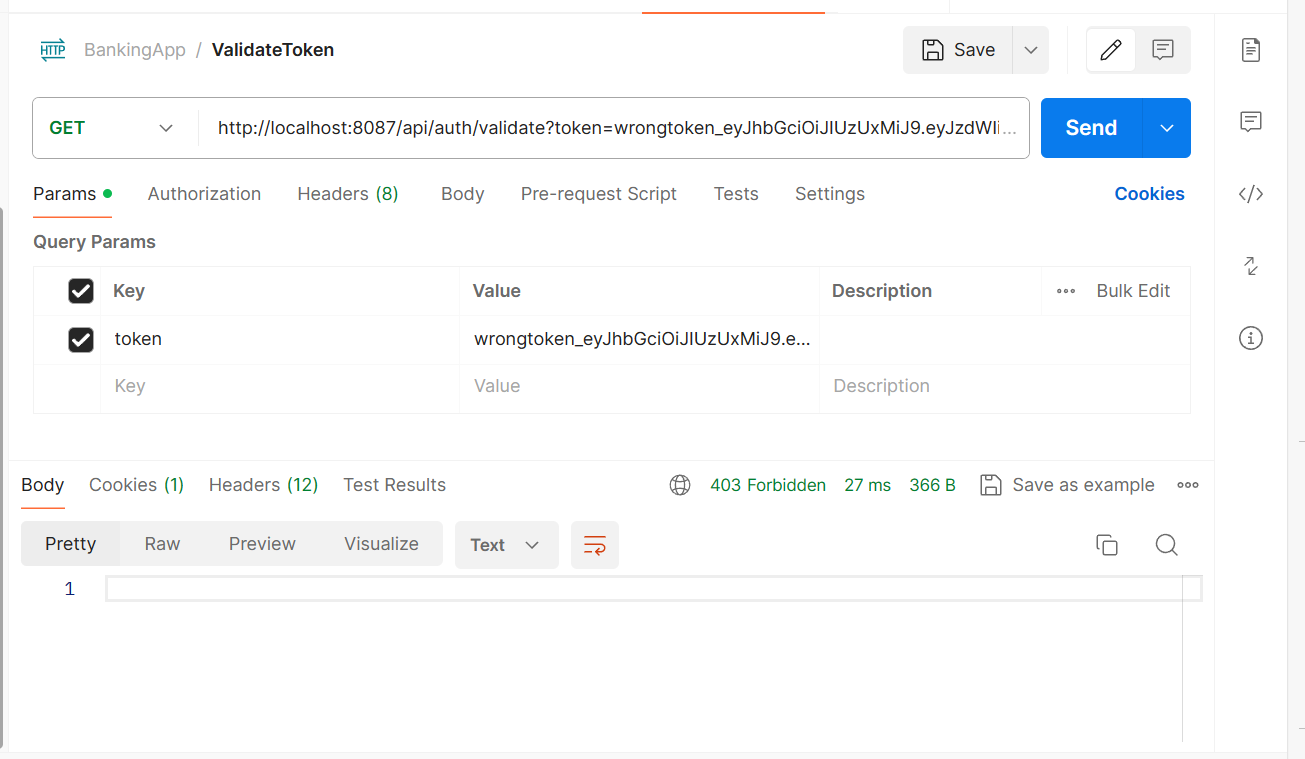
Valid token is generated when the GenerateTokenForUser endpoint is invoked with a valid user and password:



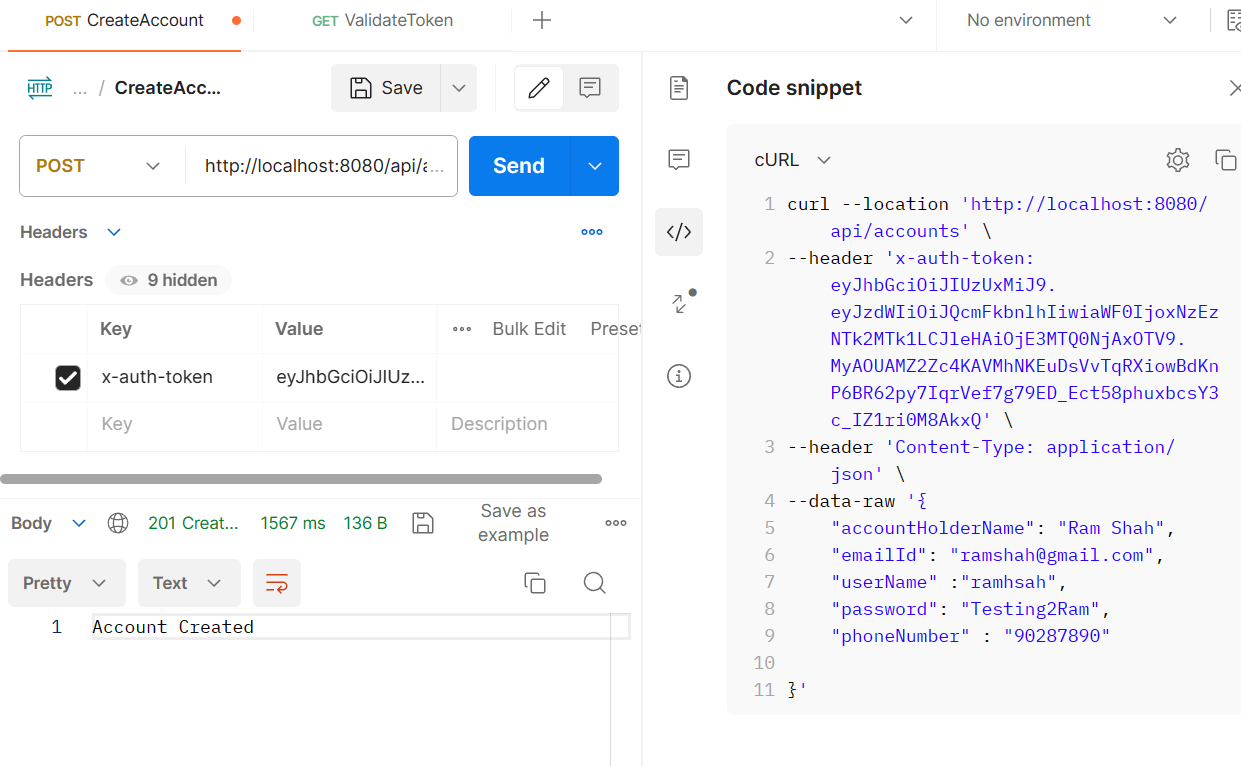
The ValidateToken endpoint returns a successful response with 200 status when the token is valid.



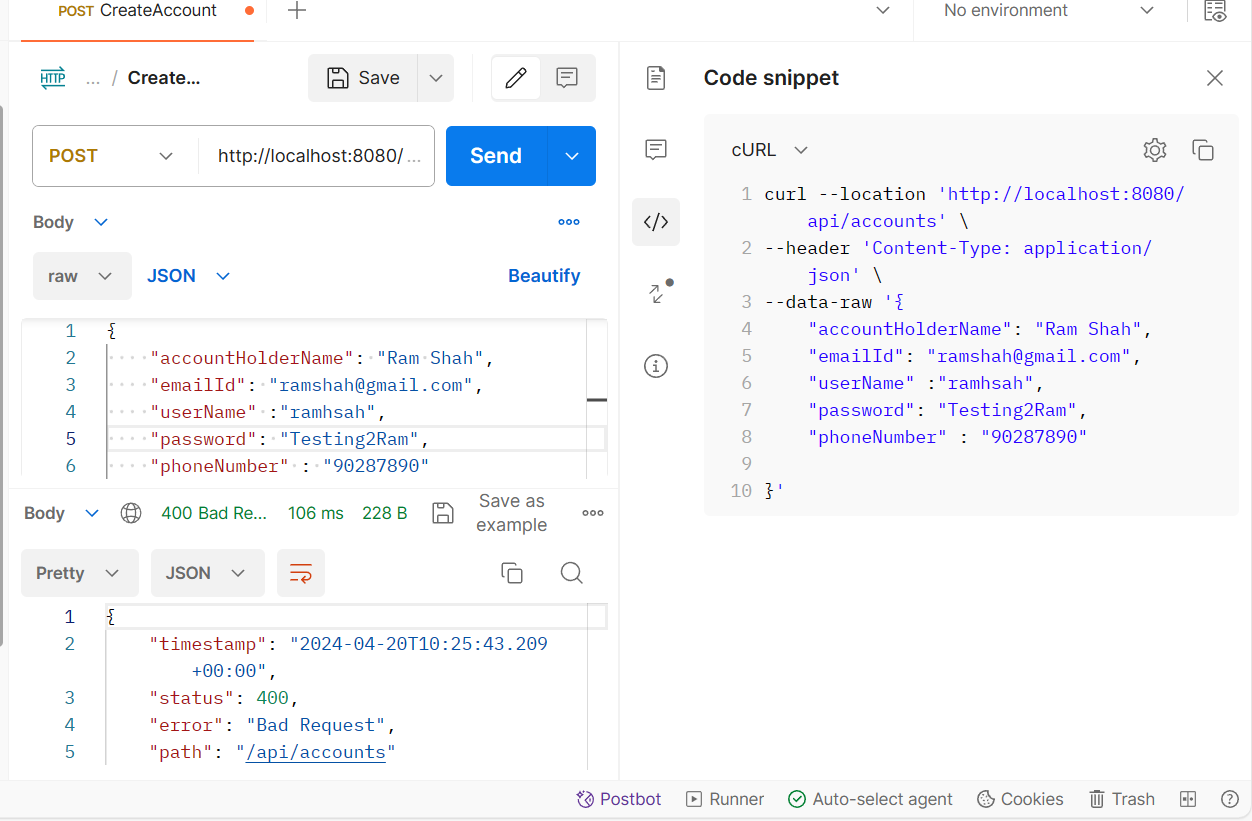
The ValidateToken endpoint returns a 403 Forbidden error when invoked with an invalid token.



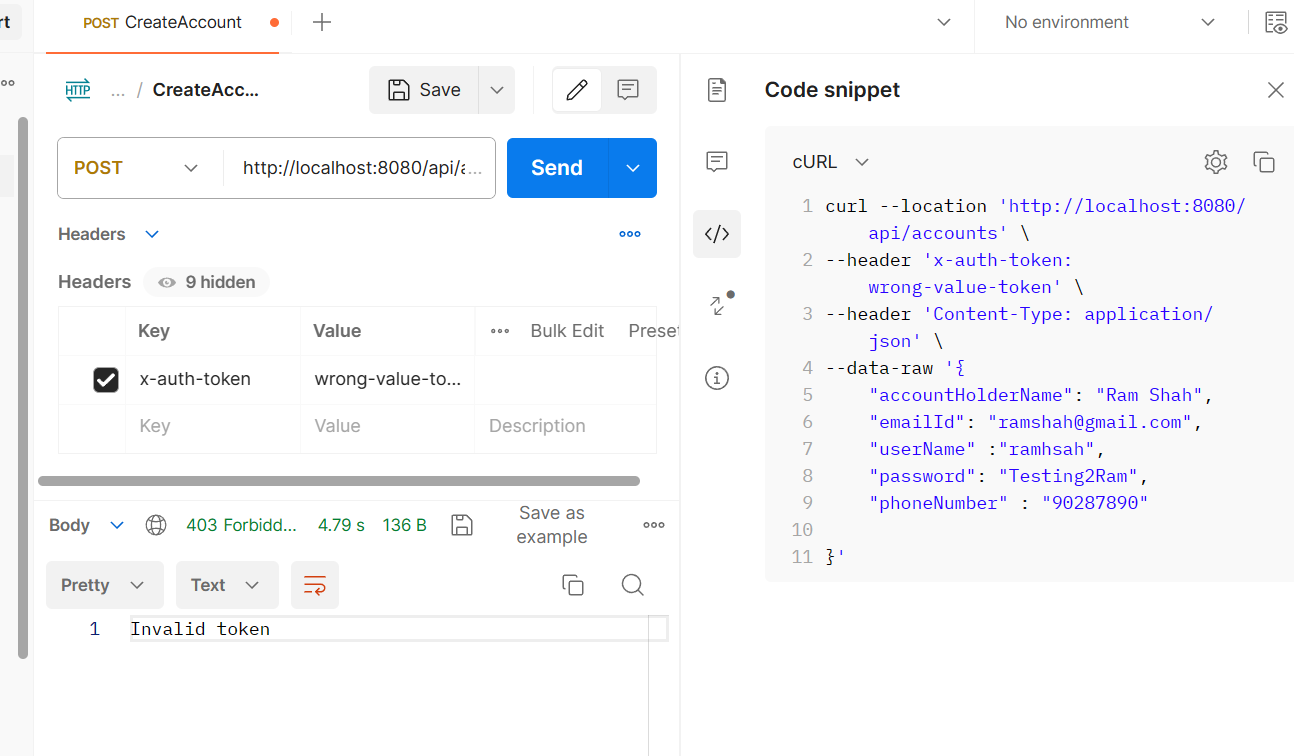
AccountService internally invokes the validateToken endpoint from Identity Service before proceeding with the business logic to create the account. It is successful when invoked with a valid token (and valid request).



Account Service without a token value in the header will fail with a 400 Bad Request



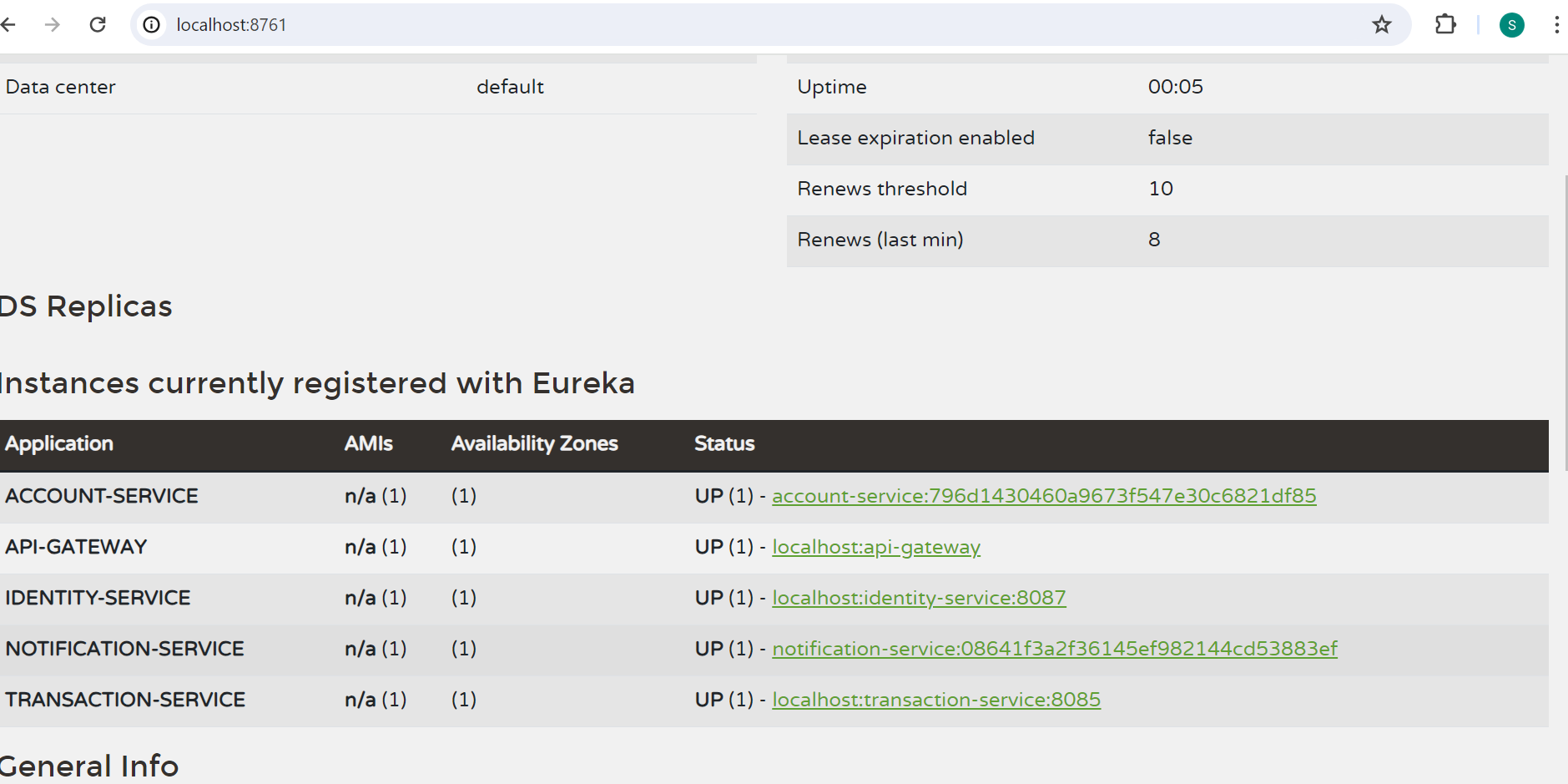
Account request with a wrong value of x-auth-token will throw a 403 Forbidden Error with a message indicating that the token value is invalid.



Eureka Discovery Server

All the services register itself with Eureka Discovery Server. The Eureka Discovery Server and the API Gateway function as a Load balancer. If there are multiple instances of the Account Service, all these instances register with Eureka Server. When the transaction service communicates it with the Account Service, it first sends a request to the Discovery server. The Discovery Server responds with a list of registries of the Account Service. The transaction Service then invokes one of the Account service instance.

Eureka Discovery client and server helps in inter process communication with multiple instances of the service.



Zipkin and Micrometer

This is used for distributed tracing and observability. Distributed Tracing allows us to trace the entire request path from its initiation to completion. Every request is assigned with a trace id that spans across multiple microservices. There is a span id assigned for when the request traverses through each microservice. Span id represents the number of the trips the request takes in our system.

Zipkin is a UI tool which shows the traceability of the request across different microservices.

