International Institute of Information Technology, Bangalore

Software Production Engineering CS-816

Online Banking System

(Under the Guidance of Professor B. Thangaraju
and Jacob Mathew (TA))

Group Members

Kuldip Nivruti Bhatale (MT2023087)
Sanket Shantaram Patil (MT2023051)

Github- Online-Banking-System
Video link- Demo
DockerHub- kb1110, sanketp29

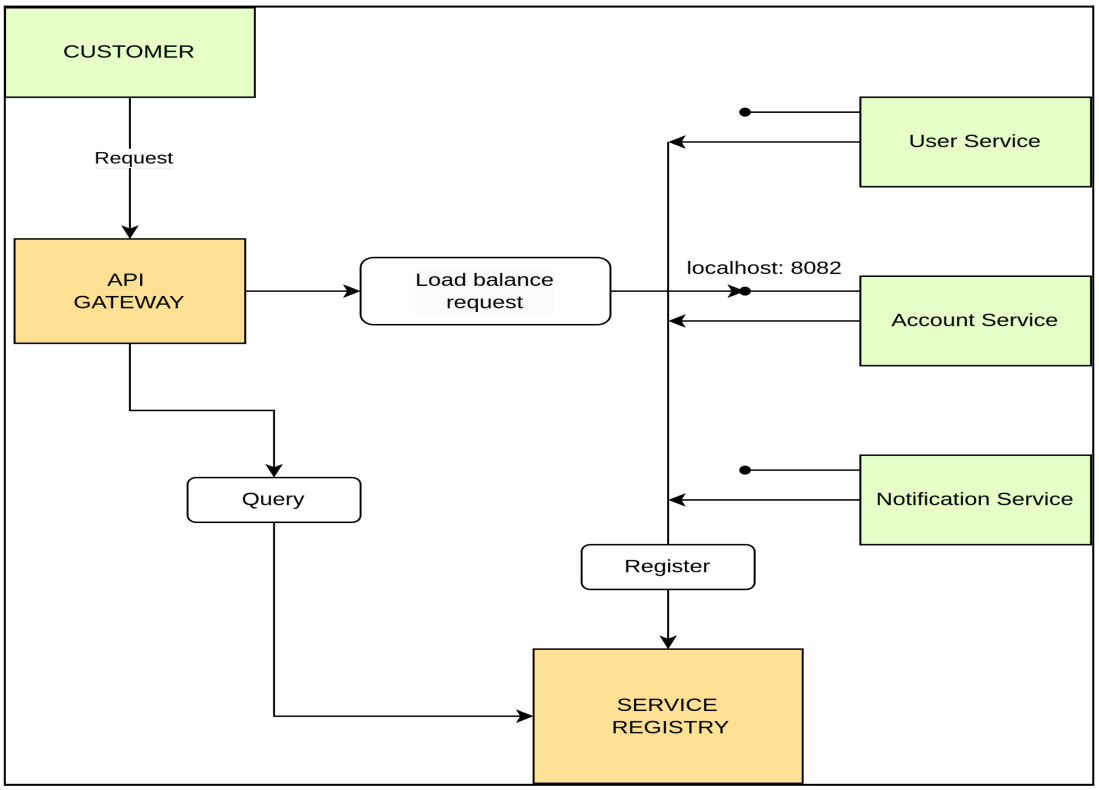
Table of Contents

Introduction

This project is an Online Banking System designed to provide users with a comprehensive and seamless banking experience. Utilizing a microservices architecture, the system ensures scalability, maintainability, and efficient resource management. Key functionalities of the Online Banking System include money transfer between accounts, viewing transaction history and account details, and registering new bank accounts.

Users can manage their profiles, log in, and log out with ease. To enhance the user experience, the system also includes a notification service that sends email alerts for each successful transaction. This feature keeps users informed and adds an extra layer of security.

By leveraging modern technologies and DevOps practices, this project delivers a robust and user-friendly online banking platform that meets the needs of today's digital banking landscape.



Microservice architecture diagram for online banking system

- **Eureka Service Registry -**
  The Eureka Service Registry is a core component of our microservices architecture. It registers all microservices, enabling them to discover and communicate with each other seamlessly. By using Eureka, services can dynamically register themselves and locate other services, which simplifies the process of inter-service communication and ensures high availability and resilience.

- **API Gateway -**
  The API Gateway serves as a centralized entry point for all client requests. It handles authentication and authorization, ensuring that only authenticated and authorized users can access the services. Additionally, the API Gateway performs load balancing, distributing incoming requests across multiple instances of each service to optimize resource usage and enhance performance.

- **User Service -**
  The User Service manages all user-related functionalities. It includes features for user registration, allowing new users to create accounts, and user login, enabling existing users to authenticate and access their accounts. The User Service also handles profile management, allowing users to view and update their personal information.

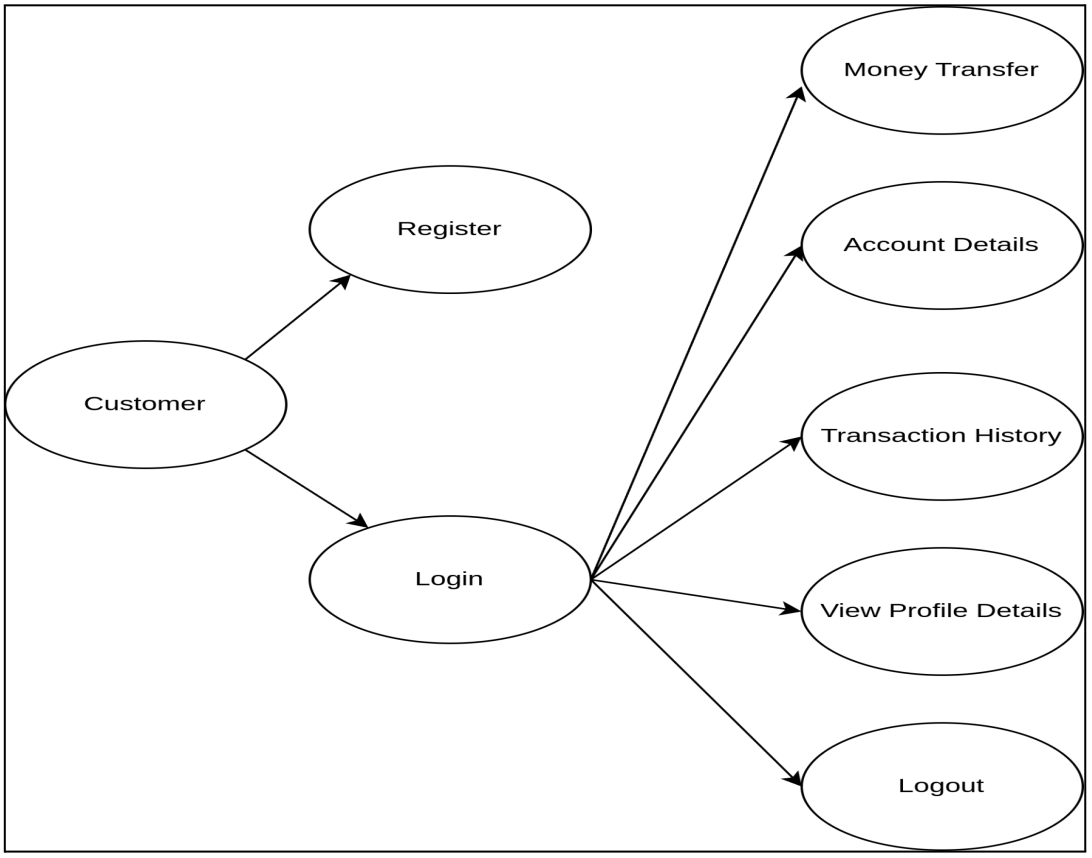
- **Account Service -**
  The Account Service is responsible for managing user accounts and financial transactions. It provides functionalities such as viewing account details, checking balances, viewing transaction history, and transferring money between accounts. This service ensures that all account-related operations are handled securely and efficiently.

- **Notification Service -**
  The Notification Service handles all communication-related tasks, particularly sending email notifications. For each successful transaction, this service sends an email to the user, informing them of the transaction details. This feature enhances user experience by keeping users informed and adding an extra layer of security to the banking operations.

Technology Stack

Frontend	React js Framework
Microservices	Java with Spring Boot
Database	MySQL
Version control	Git, GitHub
CI pipeline	Jenkins
Containerization	Docker
Building/Packaging	Docker and Dockerhub
Deployment	Ansible, Docker Compose
Logging	ELK Stack

Use Case Diagram



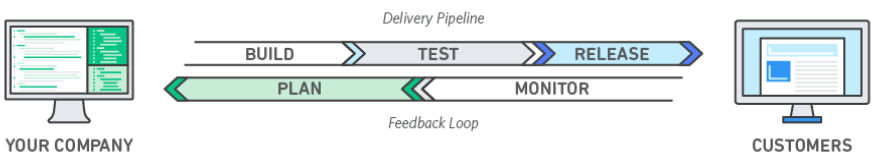
Use Cases

Sr. No	Use Case	Description
1.	Customer Registration	Customers can register to open a bank account.
2.	Login	Customers can login with their credentials to avail the functionalities of the bank.
3.	Money Transfer	Customer can transfer money to any bank account within the system
4.	Account Details	Details(Available account balance, account type, status, etc.) related to all bank accounts linked with the customer id are visible to the customer
5.	Transaction History	Customers can view all the payment transaction details (Transaction date, amount, receiver account no., etc.)
6.	Profile Details	Personal details of the customer
7.	Email Notifications	For each payment transaction the receiver gets email notification with sender's details and amount

What and Why DevOps ?

What is DevOps ?

- DevOps is a software development methodology that fosters collaboration and communication between software developers and IT operations professionals. It aims to enhance the quality and speed of software development and delivery, while simultaneously reducing costs and increasing efficiency.
- Traditional Development vs. DevOps
  In traditional software development processes, developers create code and then pass it to operations teams for deployment and management. This separation can lead to delays and miscommunications, which can slow down the development process and result in less effective software.

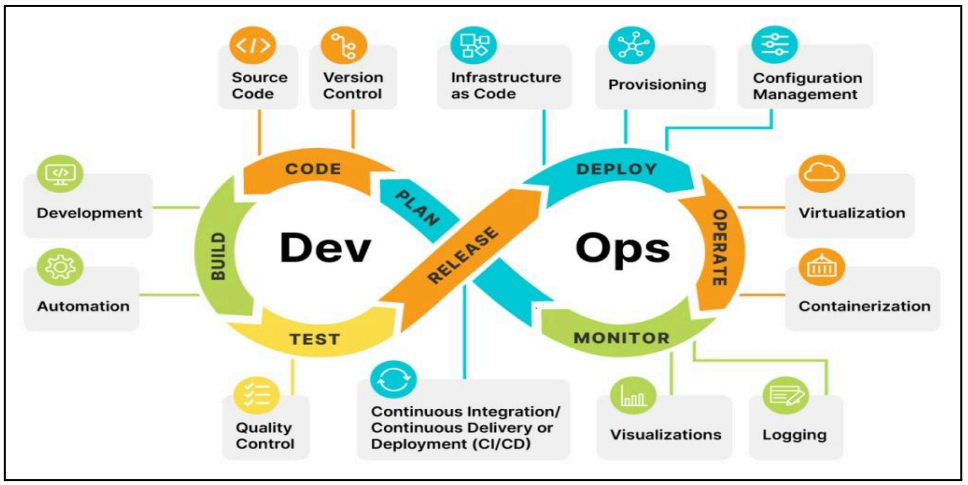


- Goals of DevOps
  DevOps aims to eliminate these barriers by promoting:
    - Collaboration: Encourages continuous interaction and cooperation between development and operations teams.
    - Automation: Uses automated tools and practices to streamline processes and reduce manual intervention.
    - Continuous Feedback: Ensures ongoing feedback loops throughout the software development lifecycle.

- DevOps Practices
  DevOps teams work together throughout the entire software development lifecycle, from planning and development to testing and deployment. This integrated approach allows teams to identify and resolve issues more quickly and efficiently, ensuring that software is delivered on time and meets user needs.

- Key Features
    - Automation and Continuous Delivery: Emphasizes automating repetitive tasks and adopting continuous delivery practices to streamline development and minimize errors and downtime.
    - Efficiency and Speed: Facilitates quick and easy testing, deployment, and updates without manual intervention.
    - Improved Quality and Cost Reduction: Enhances the quality of software development processes, reduces costs, and increases efficiency.

Overall, DevOps is a powerful approach to software development that helps organizations improve the quality and speed of their software development processes while reducing costs and increasing efficiency.



DevOps lifecycle

Why DevOps ?

Transformation of Industries: The widespread use of software and the internet has drastically changed industries such as shopping, entertainment, and banking.

Fundamental Component: Companies no longer view software as a mere support tool but as a fundamental component of their business operations.

Customer Engagement: Businesses use software to connect with their customers through various online services and devices.

Efficiency in Operations: Software is utilized to streamline internal processes, enhancing efficiency and productivity.

Modern Business Needs: Just as physical product companies transformed through industrial automation in the 20th century, modern businesses must now innovate in their approach to software development and delivery.

Tools Used

While DevOps is considered a mindset first, there are several automation tools that can help increase the efficiency of DevOps processes:

Git: Git is a version control system. In DevOps, it's used to keep track of code and is useful for team members to collaborate on projects and update existing ones.

Spring Boot - Spring Boot is a java based framework. It provides a set of tools and conventions to simplify the process of creating, configuring, and deploying standalone, production-grade Spring-based applications that are easy to maintain.

Maven: Maven in DevOps can handle a project's build, reports, and documentation from a vital piece of information.

MySQL - MySQL is an open-source relational database management system (RDBMS) that is widely used for building web applications.

Mockito - is a popular Java testing framework that allows developers to create mock objects of interfaces and classes for testing purposes. It's commonly used in conjunction with JUnit, a widely-used unit testing framework for Java.

Docker: Docker is used for containerizing applications—the process of turning an application into a single package of software.

Jenkins: Jenkins is a tool used to build CI/CD pipelines, where developers can build, test, and deploy software.

Ansible: used primarily for configuration management and infrastructure orchestration. Ansible can be used to automate Docker and build and deploy Docker containers.

Npm - package manager for the Node JavaScript platform. It puts modules in place so that node can find them, and manages dependency conflicts intelligently.

Docker Compose - Docker Compose is a tool for defining and running multi-container Docker applications. It allows us to use a YAML file to configure your application's services, making it easier to manage and orchestrate complex applications that involve multiple containers.

ELK Stack - it is a powerful suite of tools for searching, analyzing, and visualizing data in real-time. ELK stands for Elasticsearch, Logstash, and Kibana, three open-source projects

Steps

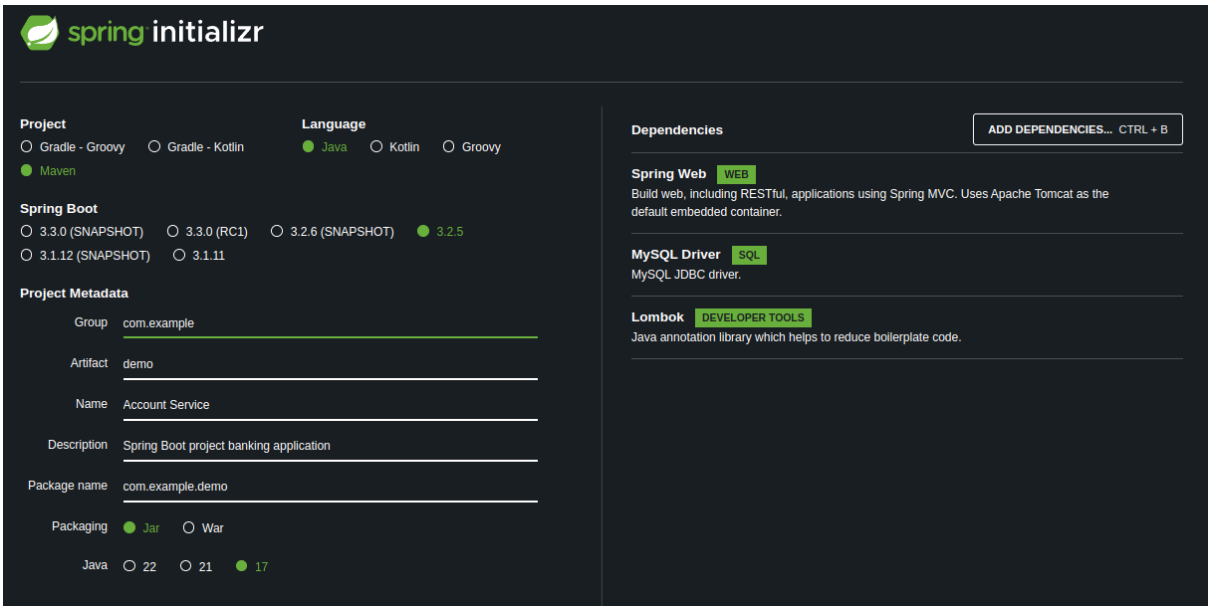
1. Microservices

The command to install Java Development Kit(JDK):

```
sudo apt-get install openjdk-17-jdk
```

Spring Initializer project-

Create a new Spring Initializer project by going to start.spring.io website and create a maven based project and add all the dependencies. Like this we create for each of the microservices. In our project we have 5 Spring Boot based microservices. We are using java 17.

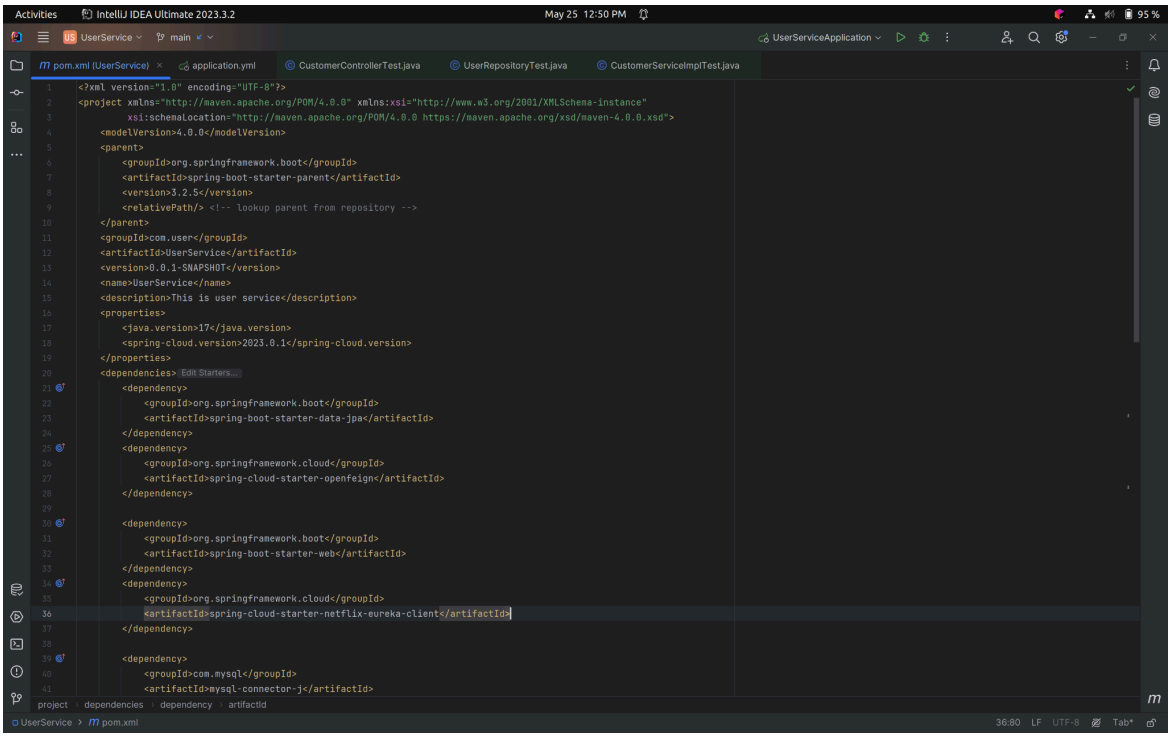


start.spring.io website

Spring Initializr generates a valid project structure with the following files:

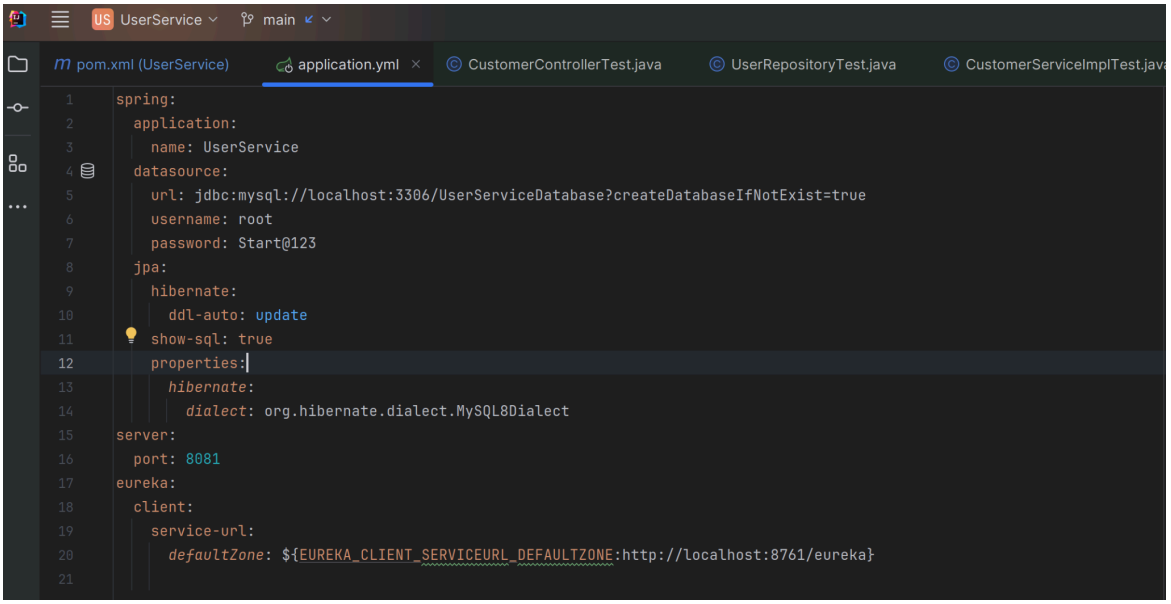
- A build configuration file, for example, build.gradle for Gradle or pom.xml for Maven.
- A class with the main() method to bootstrap the application.
- An empty JUnit test class.
- An empty Spring application configuration file: application.properties , we will be using application.yml file instead of this.

Here is a sample of the following files:



POM.XML

The below application.yml file configures an User Service, specifying the application name, database connection details, Hibernate settings, server port, and Eureka client configuration for service discovery.



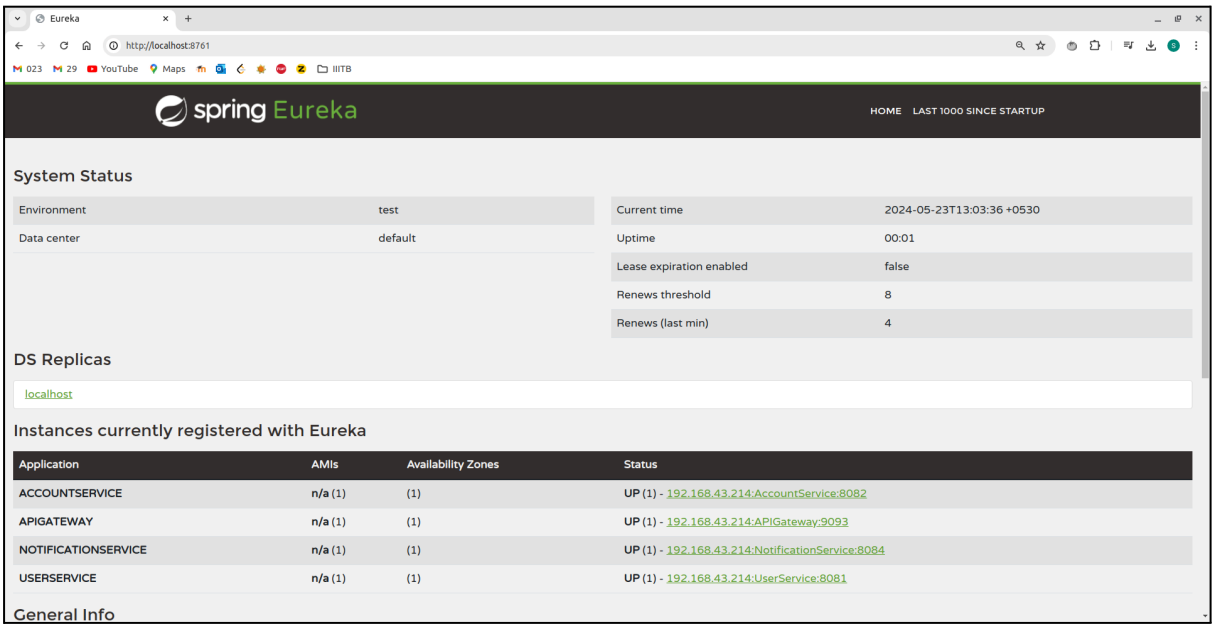
Application.yml

```java
kb111-98
@Test
void getCustomerById_ValidId_ReturnsCustomer() throws Exception {
    // Arrange
    Long customerId = 1L;
    Customer customer = new Customer();
    customer.setCustomer_id(customerId);
    customer.setFirstname("John");
    customer.setLastname("Doe");
    // Set other fields as needed

    when(customerService.getCustomerById(customerId)).thenReturn(customer);

    // Act & Assert using MockMvc
    mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/customer/{c_id}", customerId))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.customer_id").exists())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.customer_id").value( expectedValue: 1L))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.firstname").value( expectedValue: "John"))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.lastname").value( expectedValue: "Doe"));
}
```

JUnit Testing

2. Eureka Service Registry

It centralizes and organizes these settings in a human-readable format, facilitating easy management and maintenance.



Each microservice registers itself with Eureka upon startup. When a microservice needs to communicate with another, it queries Eureka for the location of that service. Eureka responds with the necessary information, allowing seamless communication between microservices. This process simplifies inter-service communication and ensures high availability and resilience within the microservices architecture.

In addition to facilitating microservices registration and discovery, The Eureka server provides a web interface. This interface allows us to visualize the registered microservices, their status, and other relevant metadata. It offers a convenient way to monitor the health and performance of the microservices ecosystem.

3. MySQL Database:

STEP 1: Install the package of 'mysql-server'.

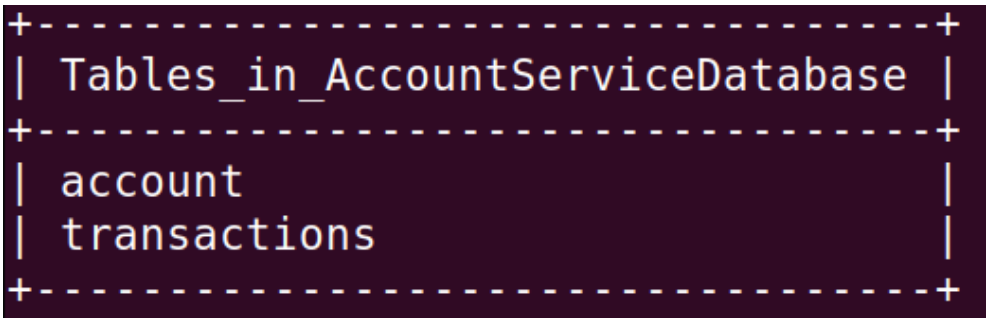
```
$ sudo apt install mysql server
```

STEP 2: Verify MySQL service status

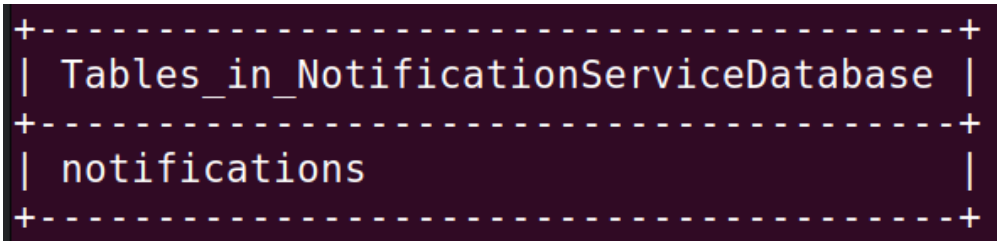
```
$ sudo systemctl status mysql
```

STEP 3: Creating a Dedicated MySQL User and Granting Privileges

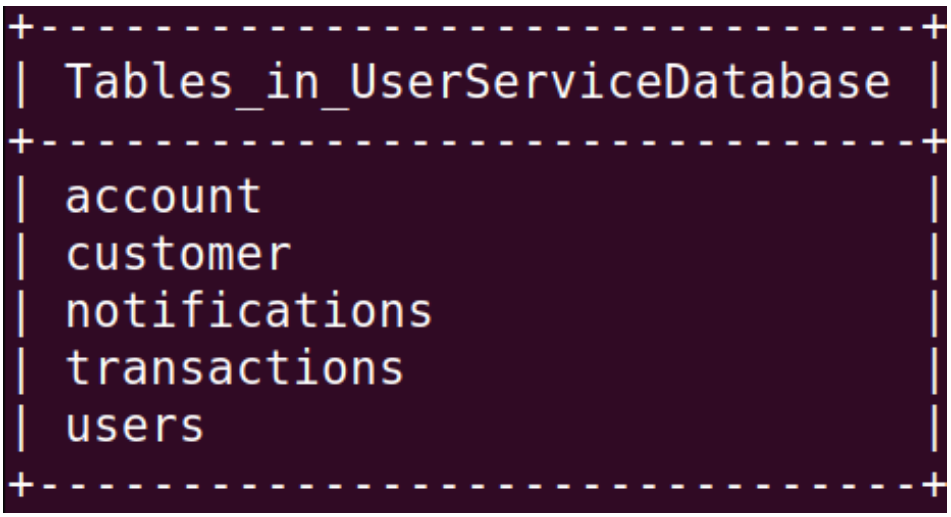
```
$ sudo mysql -u root -p
```



Account Service Database



Notification Service Database



Tables_in_UserServiceDatabase
account
customer
notifications
transactions
users

User Service Database

4. Source Code Management-

A Source Code Management (SCM) tool is essential for managing source code in software development projects. In our project, we use GitHub for SCM. Here's our workflow:

1. Cloning the Repository:
   Command: `**git clone <repository url>**`
   Description: Copies the entire data from the GitHub repository to the local system.

2. Creating a Branch:
   Command: `**git checkout -b <branch_name**>`
   Description: Creates a new branch locally with the specified name for isolated development.

3. Adding Changes:
   Command: `**git add <changed files>**`
   Description: Adds changes in the working directory to the staging area, preparing them for commit.

4. Committing Changes:
   Command: `**git commit -m "commit messag**e"`
   Description: Saves changes to the local repository with a concise message describing the changes.

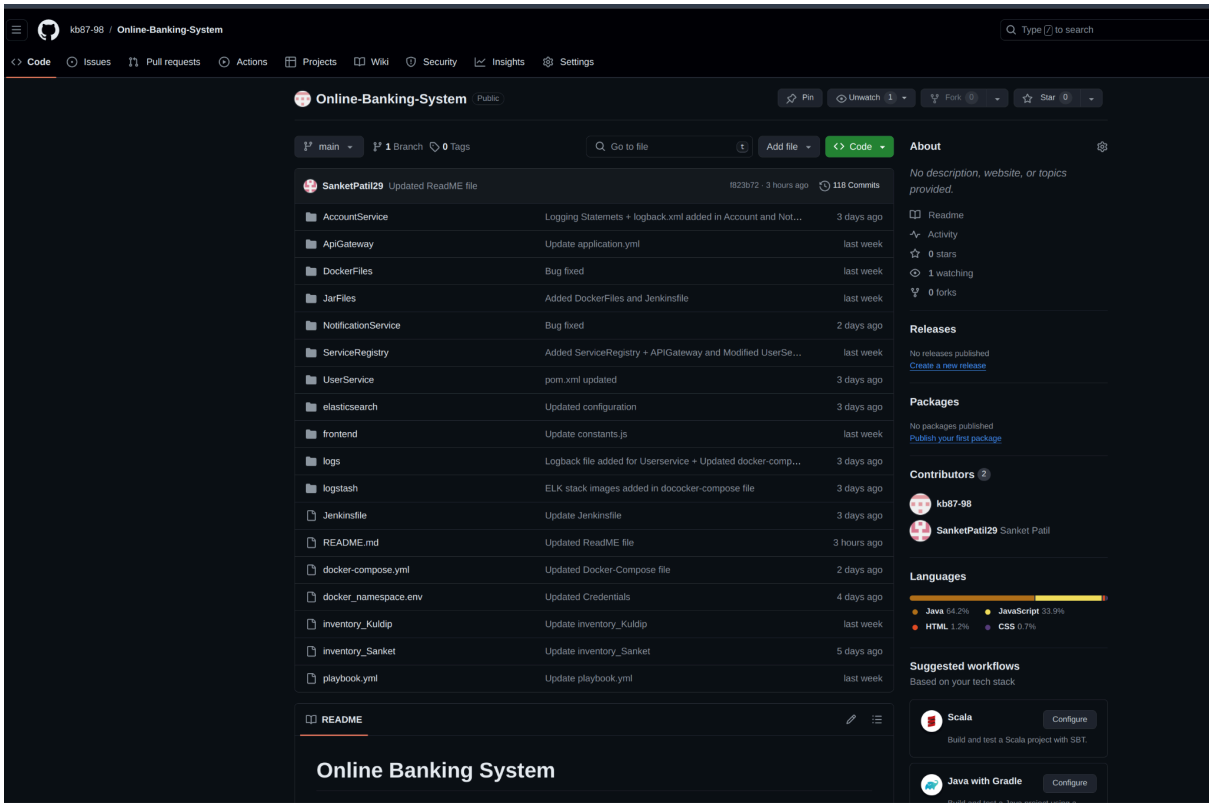
5. Switching to Master Branch:
   Command: `**git checkout master**`
   Description: Switches the working directory to the master branch.

6. Pulling Latest Code:
   Command: `**git pull**`
   Description: Updates the local version of the repository with the latest changes from the remote master branch.

7. Merging Changes:
   Command: `**git merge <branch_name**>`
   Description: Integrates changes from the specified branch into the master branch.

8. Pushing Changes:
   Command: `**git push**`
   Description: Pushes all committed changes from the local repository to the remote repository on GitHub.

This workflow ensures that every team member can work independently on their tasks and then integrate their changes smoothly into the main codebase, maintaining consistency and minimizing conflicts.



5. Maven Build

sudo apt install maven command is used for installation of Maven. After installation is complete, the maven version can be checked using below command.

mvn –version is used for checking the version of the maven

6. Unit Testing in the Project

In our Online Banking System project, unit testing is implemented to ensure the reliability and correctness of individual components. The testing framework used is JUnit in conjunction with Mockito for mocking dependencies. Here's a brief overview of the testing structure:

1. Mockito (JUnit Testing)

Mockito is used alongside JUnit to create mock objects and define their behavior. This allows for isolated testing of components by simulating the behavior of dependencies.

2. Service Implementation Testing

Purpose: To validate the business logic within service classes.

Approach: Mock dependencies (such as repositories or external services) to test service methods in isolation. Ensure that the service methods behave as expected under different conditions and inputs.

Example: Testing a money transfer service method to verify that it correctly updates account balances and records transactions.

```
class CustomerServiceImplTest {

    @Mock
    private CustomerRepository customerRepository;

    @Mock
    private UserRepository userRepository;

    @Mock
    private AccountClient accountClient;

    @InjectMocks
    private CustomerServiceImpl customerService;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.initMocks(this);
    }


    @Test
    void add_ValidInput_ReturnsCustomer() {
        // Arrange
        UserInfoDto userInfoDto = new UserInfoDto();
        userInfoDto.setUsername("testUser");
        userInfoDto.setPassword("testPassword");
        userInfoDto.setRole("testRole");
```

```
        userInfoDto.setFirstname("John");
        userInfoDto.setLastname("Doe");
        userInfoDto.setType("Personal");
        userInfoDto.setGender("Male");
        userInfoDto.setDob(Date.valueOf("1990-01-01"));
        userInfoDto.setAddress("123 Main St");
        userInfoDto.setMobile(1234567890L);
        userInfoDto.setEmail("test@example.com");

        User user = UserMapper.mapToUser(userInfoDto);
        when(userRepository.save(any(User.class))).thenReturn(user);


when(customerRepository.save(any(Customer.class))).thenAnswer(invocation ->
{
            Customer savedCustomer = invocation.getArgument(0);
            savedCustomer.setCustomer_id(1L); // Set a dummy customer ID
            return savedCustomer;
        });

        Account mockAccount = new Account();
        mockAccount.setAccount_id(1L);

when(accountClient.createAccount(any(Account.class))).thenReturn(mockAccount
);

        // Act
        Customer result = customerService.add(userInfoDto);

        // Assert
        assertNotNull(result);
        assertEquals("John", result.getFirstname());
        assertEquals("Doe", result.getLastname());
        assertEquals("Personal", result.getType());
    }
```

3. Controller Testing

Purpose: To ensure that the endpoints in the controllers handle HTTP requests correctly and return appropriate responses.

Approach: Use mock MVC (Model-View-Controller) frameworks to simulate HTTP requests and verify the responses. Test various scenarios, including successful requests and error handling.

Example: Testing the user registration endpoint to check that it processes valid input correctly and handles invalid input gracefully.

```
@WebMvcTest(CustomerController.class)
class CustomerControllerTest {

    @Autowired
```

```
    private MockMvc mockMvc;

    @MockBean
    private CustomerService customerService;

    @Test
    void addCustomer_ValidInput_ReturnsCreated() throws Exception {
        // Arrange
        UserInfoDto userInfoDto = new UserInfoDto();
        userInfoDto.setUsername("testUser");
        userInfoDto.setPassword("testPassword");
        userInfoDto.setRole("testRole");
        userInfoDto.setFirstname("John");
        userInfoDto.setLastname("Doe");
        userInfoDto.setType("Personal");
        userInfoDto.setGender("Male");
        userInfoDto.setDob(Date.valueOf("1990-01-01"));
        userInfoDto.setAddress("123 Main St");
        userInfoDto.setMobile(1234567890L);
        userInfoDto.setEmail("test@example.com");

        Customer addedCustomer = new Customer();
        addedCustomer.setCustomer_id(1L);
        addedCustomer.setFirstname("John");
        addedCustomer.setLastname("Doe");
        // Set other fields as needed

        when(customerService.add(userInfoDto)).thenReturn(addedCustomer);

        // Act & Assert using MockMvc
        mockMvc.perform(MockMvcRequestBuilders.post("/api/customer")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(asJsonString(userInfoDto)))
                .andExpect(MockMvcResultMatchers.status().isCreated())

.andExpect(MockMvcResultMatchers.jsonPath("$.customer_id").exists())

.andExpect(MockMvcResultMatchers.jsonPath("$.customer_id").value(1L))

.andExpect(MockMvcResultMatchers.jsonPath("$.firstname").value("John"))

.andExpect(MockMvcResultMatchers.jsonPath("$.lastname").value("Doe"));
    }
```

4. Repository Testing

Purpose: To verify the correctness of data access operations performed by repository classes.

Approach: Use an in-memory database or mock database to test CRUD (Create, Read, Update, Delete) operations. Ensure that queries return the correct data and that the database state is managed correctly.

Example: Testing the user repository to ensure that it correctly saves user details, retrieves user information, and handles query conditions properly.

```
class CustomerRepositoryTest {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private UserRepository userRepository;

    @Test
    void saveCustomer_ReturnsSavedCustomer() {
        // Arrange
        User user = new User("testUser", "testPassword", "testRole");
        userRepository.save(user);
        Customer customer = new Customer();
        customer.setFirstname("John");
        customer.setLastname("Doe");
        customer.setGender("Male");
        customer.setDob(Date.valueOf("1990-01-01"));
        customer.setAddress("123 Main St");
        customer.setType("Personal");
        customer.setMobile(1234567890L);
        customer.setEmail("test@example.com");
        customer.setUser(user);

        // Act
        Customer savedCustomer = customerRepository.save(customer);

        // Assert
        assertEquals(customer.getFirstname(), savedCustomer.getFirstname());
        assertEquals(customer.getLastname(), savedCustomer.getLastname());
        // Add more assertions as needed
    }
```

Key Benefits

Reliability: Ensures that each component functions correctly in isolation.
Early Bug Detection: Identifies issues early in the development cycle, reducing the cost and effort of fixing bugs later.
Refactoring Confidence: Provides confidence to developers when refactoring code, knowing that existing functionality is safeguarded by tests.
Documentation: Acts as documentation for the expected behavior of the code.
By implementing thorough unit testing with Mockito and JUnit, the project ensures that each component is robust, reliable, and works correctly, contributing to the overall quality and maintainability of the system.

7. Frontend Setup

NPM

NPM (Node Package Manager) is a package manager for the JavaScript programming language. It is used primarily for managing Node.js modules, but it can also be used for managing front-end packages and dependencies.
NPM consists of two main parts:

1. a CLI (command-line interface) tool for publishing and downloading packages, and
2. an online repository that hosts JavaScript packages

It allows developers to easily install and manage third-party packages and libraries for their projects. Packages can be installed locally or globally, and can be specified as dependencies in a project's package.json file. NPM also provides a range of commands for updating, publishing, and removing packages.

```
sudo apt install npm
```

```
npm -version
```

```
sudo apt install nodejs
```

```
node -version
```

```
npm install <package_name>
```

React:

React is an open-source JavaScript library that is used for building user interfaces. React is based on the concept of components, which are self-contained modules that define a specific piece of UI. Components can be nested and combined to create complex user interfaces. React provides a declarative way of defining components using JSX (a syntax extension to JavaScript), making it easy to write and read code. Its primary goal is to make it easy to reason about an interface and its state at any point in time. It does this by dividing the UI into a collection of components.

Step 1: Install Create React App
Install create-react-app helps to set all tools required to create React Applications.
Run the following npm command to install the create-react-app utility:

```
sudo npm -g install create-react-app
```

```
create-react-app --version
```

```
npx create-react-app BankingApp
```

```
cd BankingApp
```

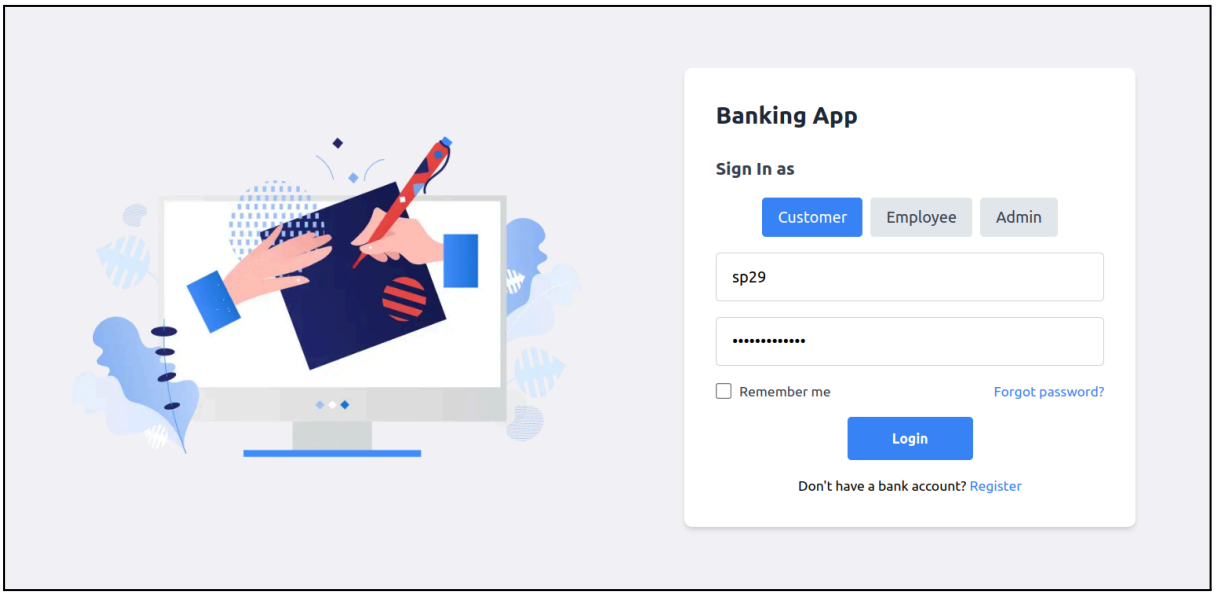
to run react application:

```
npm start
```

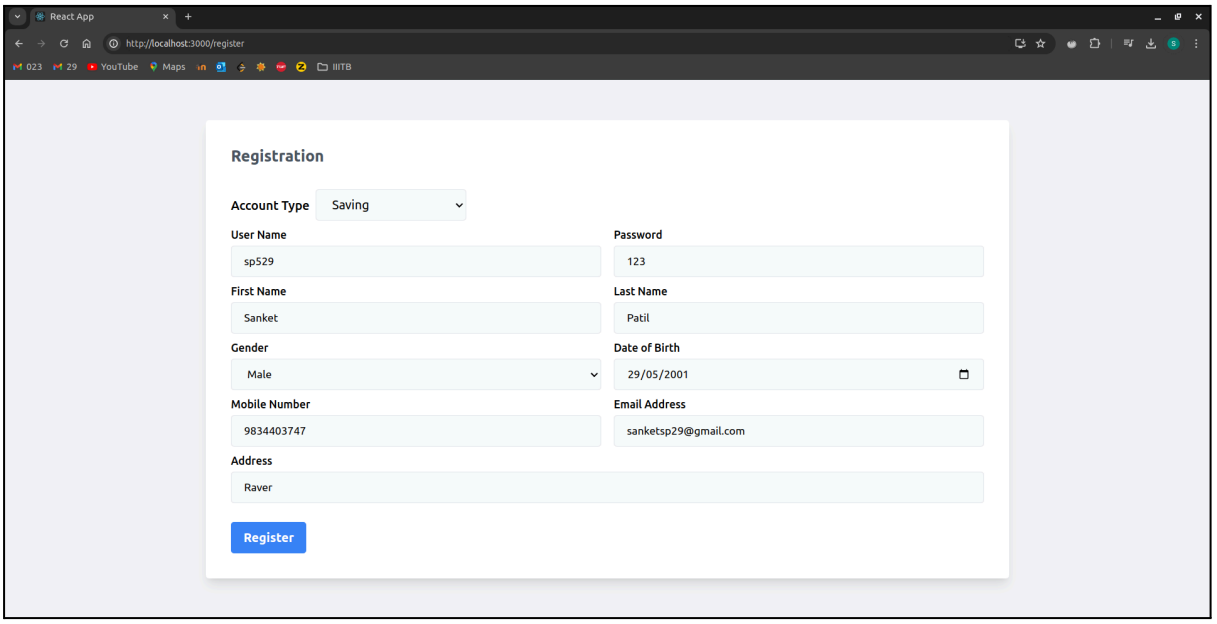
Browser will open-up and show that the app is up and running on http://localhost:3000.

8. UI of Online Banking System:

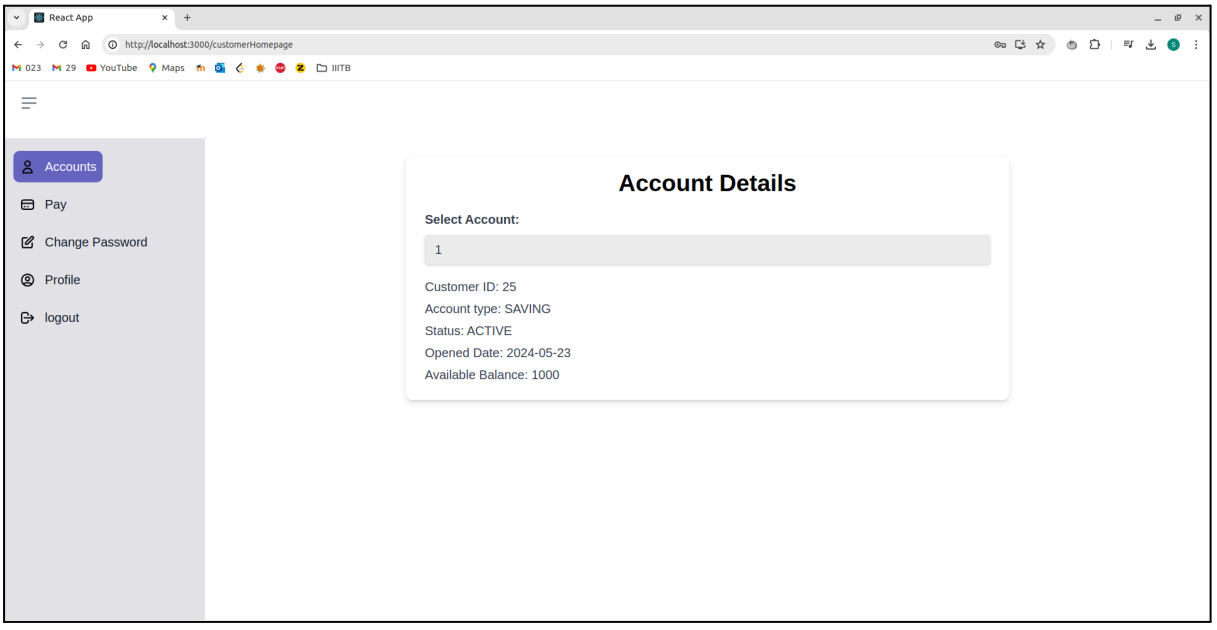
Login Page



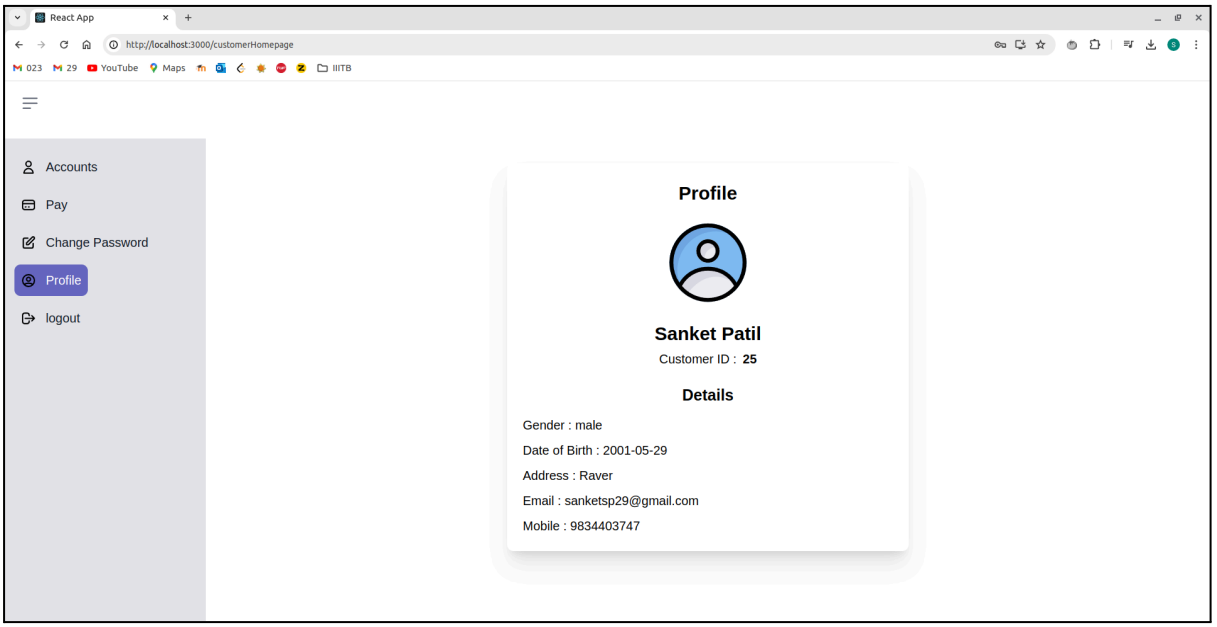
Registration Page



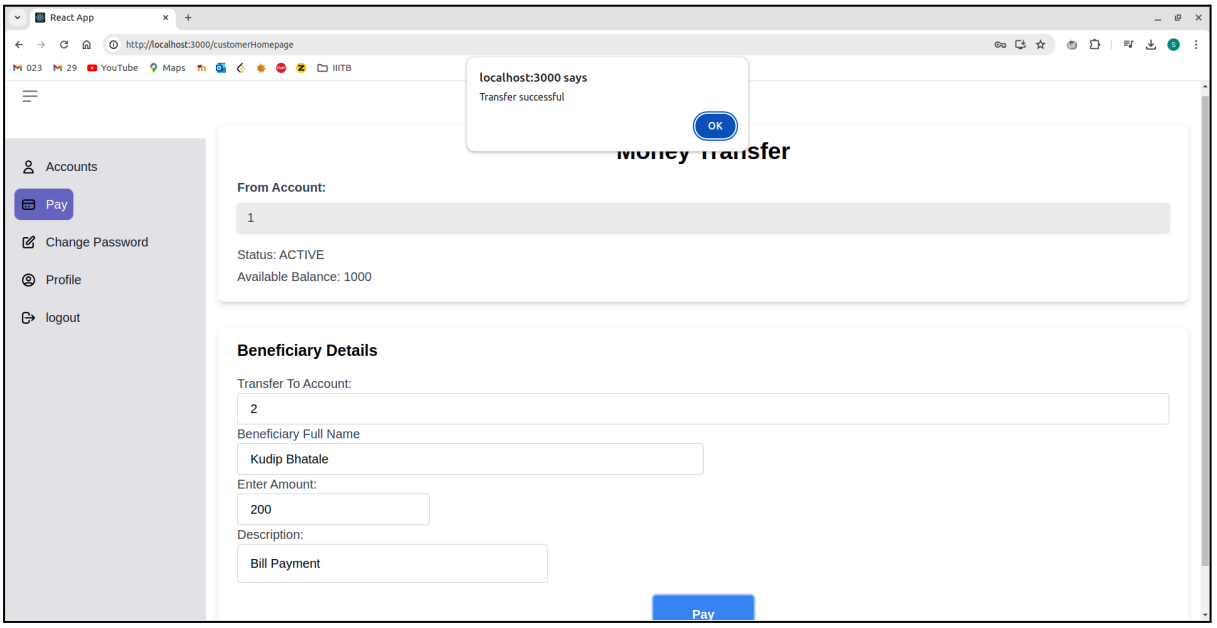
Customer Homepage after logged in with above registered credentials



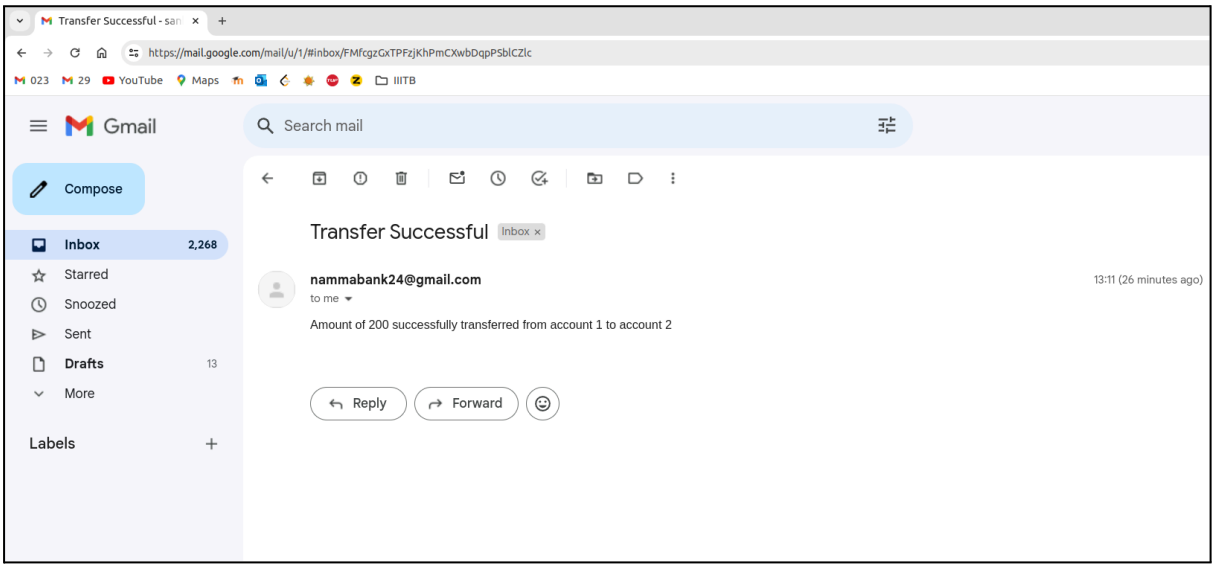
Profile Page



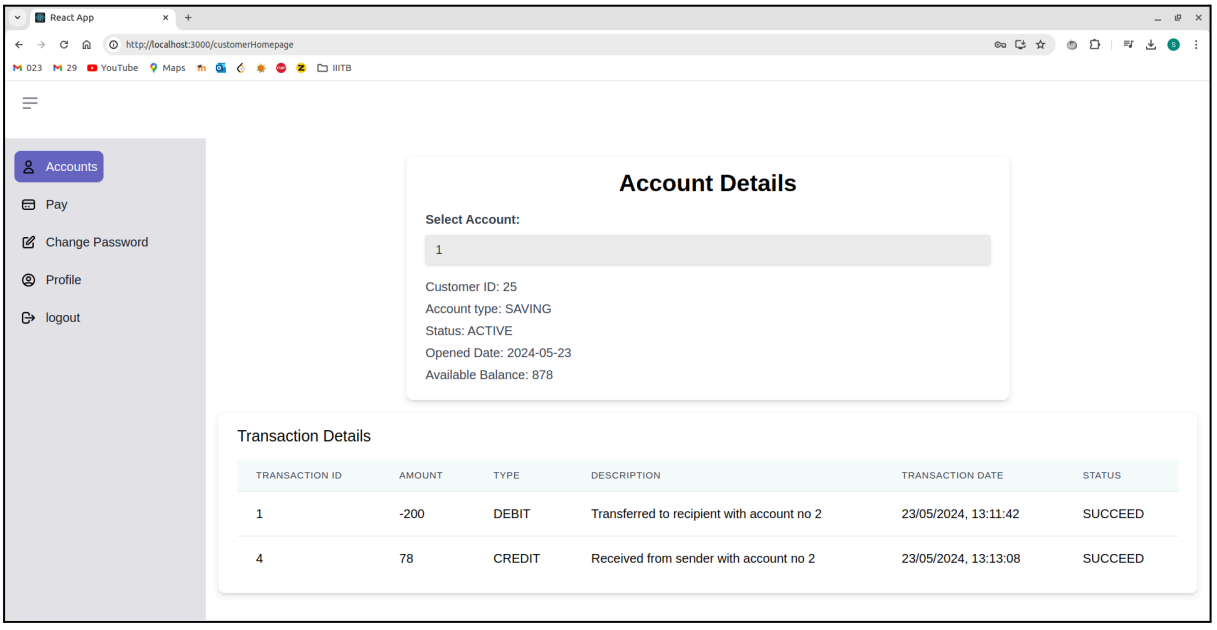
Money Transfer



Email notification for the above payment transaction



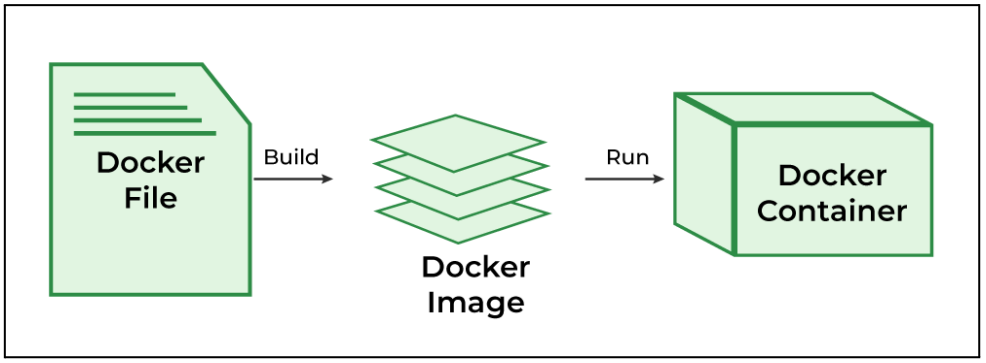
Transaction History



9. Containerization and Packaging

Containerization is a modern virtualization method that allows developers to package an application and its dependencies into a single container that can be run anywhere.
Docker is one of the most popular tools for containerization, and it provides a range of features and tools for building, deploying, and managing containerized applications.

In a containerized environment, applications run independently of the underlying operating system, allowing for greater portability and flexibility. Containers are lightweight and efficient, using fewer resources than traditional virtual machines, making them easier to deploy and manage at scale.



The Dockerfile is a text file that contains instructions for building a Docker image. The image contains the application and its dependencies, as well as any custom configuration or settings. Once the image is built, it can be pushed to a container registry, such as Docker Hub, where it can be easily accessed and deployed.

To install docker on the system we need to follow below commands:

```
sudo apt install docker.io
```

```
sudo systemctl start docker
```

```
sudo systemctl status docker
```

To display all the docker images we use command:

```
docker ps -a
```

The Docker version can be checked using command.

```
docker --version
```

To run it as a user you have to enter a user to the group of docker which can be done using

```
sudo usermod -aG docker
```

Run the docker on to terminal via command

```
sudo systemctl start docker
```

To pull the docker image use command

```
docker pull
```

To push the docker image use command

```
docker push /:tagname
```

The docker image can be run using command

```
docker run -i -t
```

Following is the code snippet for frontend dockerfile:

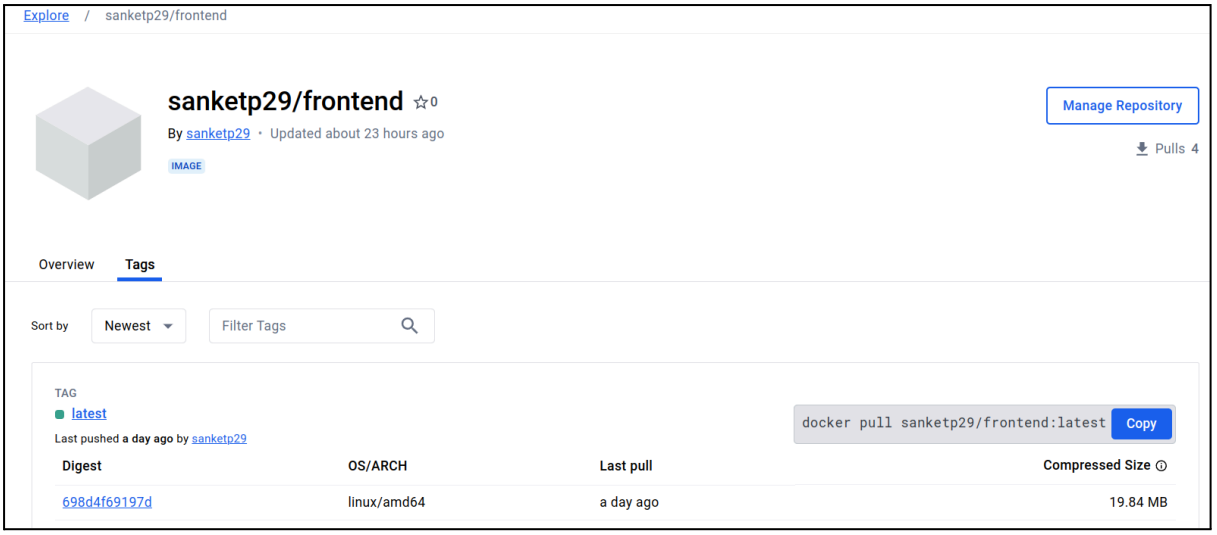
```
# Step 1: Build the React application
FROM node:18 AS build

# Set the working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY frontend/package.json ../frontend/package-lock.json ./

# Install dependencies
RUN npm install
```

```
# Copy the rest of the application code and configuration files
COPY frontend/public ./public
COPY frontend/src ./src
COPY frontend/tailwind.config.js ./


# Build the React application
RUN npm run build

# Step 2: Serve the React application using Nginx
FROM nginx:alpine

# Copy the build output to the Nginx HTML directory
COPY --from=build /app/build /usr/share/nginx/html

# Copy custom Nginx configuration file to the correct directory
COPY frontend/nginx.conf /etc/nginx/conf.d/default.conf

# Expose port 80
EXPOSE 80

# Start Nginx
CMD ["nginx", "-g", "daemon off;"]
```

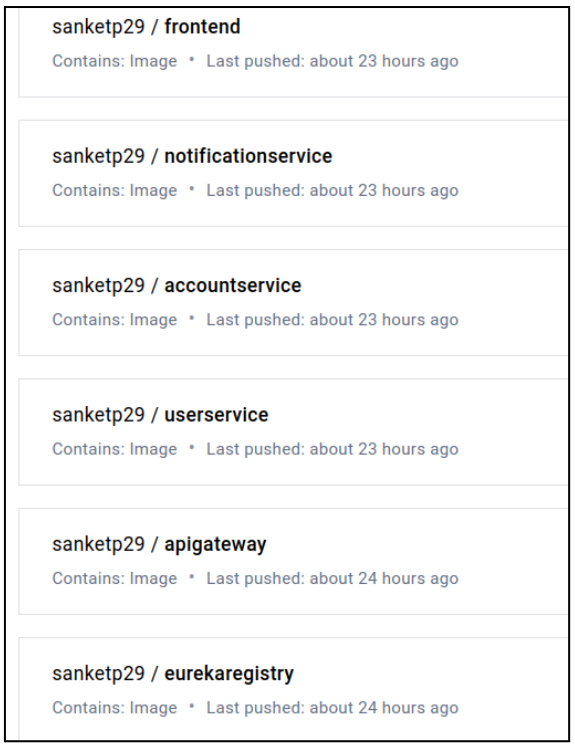
Dockerfile for frontend

Following is the dockerfile for Userservice:

```
# Use a smaller base image to run the application
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY JarFiles/UserService-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "app.jar"]
```



Frontend Docker Image



Docker Images pushed on DockerHub

10. Docker Compose

- Docker Compose is a tool for defining and running multi-container Docker applications.
- It allows you to define all of the services that your application needs to run in a single file, called a "docker-compose.yml" file. With Docker Compose, we can manage multiple containers that make up our application and run them all with a single command.
- The "docker-compose.yml" file defines the services that make up your application, including the Docker images, environment variables, network settings, and other configuration options. Each service can be defined with its own set of options, such as the port mappings, volume mounts, and resource limits.

docker-compose.yml file-

```
version: "3"

services:
 eurekaregistry:
   image: kb1110/eurekaregistry
   container_name: eurekaregistry
   ports:
     - "8761:8761"
   networks:
     - bank-network

 apigateway:
   image: kb1110/apigateway
   container_name: apigateway
   ports:
     - "9093:9093"
   networks:
     - bank-network
   environment:
     -
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eurekaregistry:8761/eureka/
   depends_on:
     - eurekaregistry

 userservice:
   image: kb1110/userservice
   container_name: userservice
```

```
    ports:
      - "8081:8081"
    environment:
      -
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eurekaregistry:8761/eureka/
      -
SPRING_DATASOURCE_URL=jdbc:mysql://userservice_db/userservice?createDat
abaseIfNotExist=true
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=root
      - LOGGING_LOGSTASH_HOST=logstash
      - LOGGING_LOGSTASH_PORT=5001
    networks:
      - bank-network
    depends_on:
      - eurekaregistry
      - apigateway
      - userservice_db
    volumes:
      - /home:/logs

  accountservice:
    image: kb1110/accountservice
    container_name: accountservice
    ports:
      - "8082:8082"
    environment:
      -
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eurekaregistry:8761/eureka/
      -
SPRING_DATASOURCE_URL=jdbc:mysql://accountservice_db/accountservice?cre
ateDatabaseIfNotExist=true
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=root
      - LOGGING_LOGSTASH_HOST=logstash
      - LOGGING_LOGSTASH_PORT=5001
    networks:
      - bank-network
    depends_on:
      - eurekaregistry
      - apigateway
      - accountservice_db
    volumes:
```

```
    - /home:/logs

 notificationservice:
   image: kb1110/notificationservice
   container_name: notificationservice
   ports:
     - "8084:8084"
   environment:
     -
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eurekaregistry:8761/eureka/
     -
SPRING_DATASOURCE_URL=jdbc:mysql://notificationservice_db/notifications
ervice?createDatabaseIfNotExist=true
     - SPRING_DATASOURCE_USERNAME=root
     - SPRING_DATASOURCE_PASSWORD=root
     - LOGGING_LOGSTASH_HOST=logstash
     - LOGGING_LOGSTASH_PORT=5001
   networks:
     - bank-network
   depends_on:
     - eurekaregistry
     - apigateway
     - notificationservice_db
   volumes:
     - /home:/logs

 userservice_db:
   image: mysql:8
   container_name: userservice_db
   ports:
     - "3307:3306"
   environment:
     - MYSQL_ROOT_PASSWORD=root
     - MYSQL_DATABASE=userservice
   volumes:
     - userservice_db:/var/lib/mysql
   networks:
     - bank-network

 accountservice_db:
   image: mysql:8
   container_name: accountservice_db
   ports:
```

```
    - "3308:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=root
    - MYSQL_DATABASE=accountservice
  volumes:
    - accountservice_db:/var/lib/mysql
  networks:
    - bank-network

notificationservice_db:
  image: mysql:8
  container_name: notificationservice_db
  ports:
    - "3309:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=root
    - MYSQL_DATABASE=notificationservice
  volumes:
    - notificationservice_db:/var/lib/mysql
  networks:
    - bank-network

frontend:
  image: kb1110/frontend
  container_name: frontend
  ports:
    - "3000:80"
  networks:
    - bank-network

elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:8.3.3
  container_name: elasticsearch
  restart: always
  environment:
    - bootstrap.memory_lock=true
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    - xpack.security.enabled=false
    - discovery.type=single-node
    - network.host=0.0.0.0
    - http.port=9200
  ports:
    - "9200:9200"
```

```
    networks:
      - bank-network
    volumes:
      - elastic_data:/usr/share/elasticsearch/data
      - /home/kb/IIITB PROJECTS/SEM2/SPE MAJOR
PROJECT/Online-Banking-System/elasticsearch/elasticsearch.yml:/usr/shar
e/elasticsearch/config/elasticsearch.yml
      - /home/kb/IIITB PROJECTS/SEM2/SPE MAJOR
PROJECT/Online-Banking-System/elasticsearch:/var/log/elasticsearch

  kibana:
    image: docker.elastic.co/kibana/kibana:8.3.3
    container_name: kibana
    restart: always
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - "5601:5601"
    networks:
      - bank-network
    depends_on:
      - elasticsearch

  logstash:
    image: docker.elastic.co/logstash/logstash:8.3.3
    container_name: logstash
    restart: always
    volumes:
      - /home/kb/IIITB PROJECTS/SEM2/SPE MAJOR
PROJECT/Online-Banking-System/logstash/config/logstash.yml:/usr/share/l
ogstash/config/logstash.yml:ro
      - /home/kb/IIITB PROJECTS/SEM2/SPE MAJOR
PROJECT/Online-Banking-System/logstash/pipeline:/usr/share/logstash/pip
eline:ro
    environment:
      - LS_JAVA_OPTS=-Xmx512m -Xms512m
    ports:
      - "5001:5001/tcp"
      - "5001:5001/udp"
      - "9600:9600"
    networks:
      - bank-network
    depends_on:
```

```
    - elasticsearch

networks:
 bank-network:
   driver: bridge

volumes:
 userservice_db:
 accountservice_db:
 notificationservice_db:
 elastic_data:
   driver: local
```

version: '3': Specifies the version of the Docker Compose file format being used.

In Docker Compose, the services, volumes, and networks sections define various aspects of the application's configuration and infrastructure:

Services:

- ➢ Define the individual components or services that compose the application. Each service corresponds to a container.
- ➢ Specifies the Docker images to be used, container names, ports to expose, environment variables, dependencies, and more.

Volumes:

- ➢ Define persistent data storage for containers. Volumes ensure that data persists even if containers are stopped or removed.
- ➢ Allows sharing of data between containers or between a container and the host system.

Networks:

- ➢ Define isolated networks for containers to communicate with each other.
- ➢ Helps in organizing and isolating communication between containers, improving security and performance.
- ➢ Allows defining custom network configurations for different components of the application.

These sections collectively define the infrastructure and configuration of the multi-container Docker application, ensuring proper communication, data persistence, and network isolation.

Services

Each service defined under the services section represents a containerized application component.

1. **eurekaregistry**
   Maps port 8761 on the host to port 8761 in the container, allowing external access to the Eureka server dashboard.
   Connected to the bank-network for inter-service communication and discovery.
2. **apigateway**
   - Acts as a central hub for incoming requests, directing them to the appropriate microservice based on predefined routing rules.
   - Configures the Eureka client to automatically register with the Eureka server for seamless service discovery.
   - Depends on eurekaregistry to ensure the availability of service registration and discovery functionality.

3. **userservice**
   - Handles various user-related operations such as registration, login, and profile management, ensuring user data integrity and security.
   - Interacts with a UserServiceDatabase to persist and retrieve user information, ensuring data consistency and availability.
   - Relies on eurekaregistry, apigateway, and userservice_db for service discovery and database connectivity, ensuring smooth operation within the ecosystem.

4. **accountservice**
   - Manages account-related functionalities including balance inquiries, transaction processing, and account management.
   - Connects to an AccountServiceDatabase to store and retrieve account information, ensuring accurate and up-to-date data handling.
   - Dependent on eurekaregistry, apigateway, and accountservice_db for service registration and database access, facilitating seamless integration with other services.

5. **notificationservice**
    - Handles the generation and dispatch of notifications such as transaction alerts and account updates to users.
    - Utilizes a NotificationServiceDatabase to persist notification records, ensuring delivery reliability and auditability.
    - Relies on eurekaregistry, apigateway, and notificationservice_db for service discovery and database connectivity, ensuring smooth operation and reliable communication.

6. **frontend**
    - Serves as the primary user interface for the online banking system, providing users with an intuitive and interactive platform.
    - Communicates with backend services via the API Gateway to fetch and display user-specific data and perform transactions.

7. **Elasticsearch, Logstash, Kibana**
    - Elasticsearch: Provides powerful search and analytics capabilities, enabling efficient storage, retrieval, and analysis of application logs and data.
    - Logstash: Acts as a data processing pipeline, facilitating the ingestion, transformation, and enrichment of log data before indexing in Elasticsearch.
    - Kibana: Offers a rich visualization dashboard for Elasticsearch data, allowing users to explore, analyze, and visualize application logs and metrics.

The databases include:

1. userservice_db:
    - MySQL database dedicated to storing customer-related data for the UserService.
    - Stores information such as user profiles, authentication credentials, and personal details.
    - Critical for ensuring data integrity and security in user management operations.

2. accountservice_db:
    - MySQL database designed to manage customer account information, transaction history, and related details.
    - Stores account balances, transaction records, and other financial data crucial for account management functionalities.
    - Integral component for maintaining accurate and up-to-date account information within the banking system.

3. notificationservice_db:
    - Database utilized by the NotificationService to store notification records and related data.
    - Stores information about notifications sent to users, including timestamps, recipient details, and notification content.
    - Enables tracking and auditing of notification activities for compliance and monitoring purposes.

4. elastic_data:
    - Volume used for persisting data in Elasticsearch, a distributed search and analytics engine.
    - Stores indexed data from various sources, including application logs, metrics, and other structured or unstructured data.
    - Facilitates efficient search, retrieval, and analysis of data using Elasticsearch and related tools like Kibana and Logstash.

All of the services and databases are connected to a custom Docker network called "**bank-network**".

11. Ansible Playbook

Ansible is a software tool designed for cross-platform automation and orchestration at scale. Written in Python and backed by RedHat and a loyal open-source community, it is a command-line IT automation application widely used for configuration management, infrastructure provisioning, and application deployment use cases.

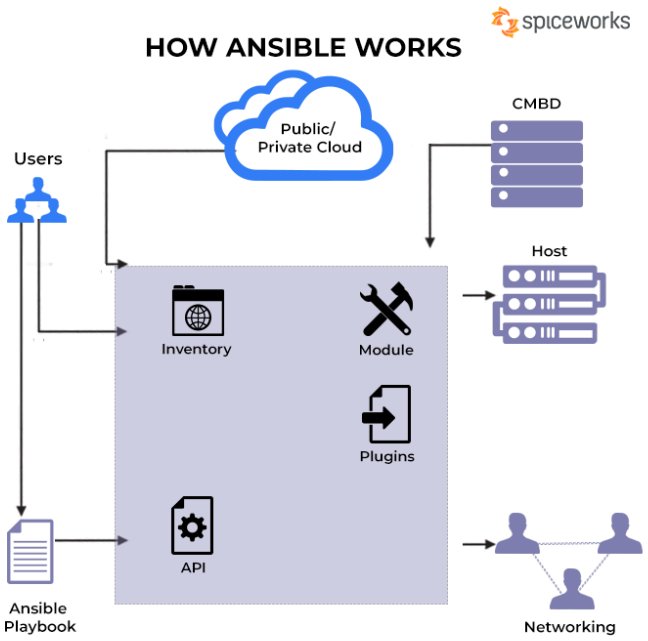


Image source- spiceworks

Playbooks are the simplest way in Ansible to automate repeating tasks in the form of reusable and consistent configuration files. Playbooks are scripts defined in YAML files and contain any ordered set of steps to be executed on our managed nodes.

Tasks in a playbook are executed from top to bottom. At a minimum, a playbook should define the managed nodes to target and some tasks to run against them.

In playbooks, data elements at the same level must share the same indentation while items that are children of other items must be indented more than their parents.

Before installing Ansible we need to install python and ssh server on controller and managed nodes, SSH installation and key generation can be done using below commands.

```
sudo apt install openssh-server
```

```
ssh-keygen -t rsa
```

Installing ansible and checking version.

```
sudo apt install ansible.
```

```
ansible --version
```

Copy ssh key generated on remote serve onto our jenkins server.
This will copy the remote user's ssh keys to the known hosts in our jenkins ssh directory , now jenkins can log-on to the managed node without any password.
We can run our playbooks on the managed node without any password.

Inventory files- The Ansible Inventory File defines all the hosts and groups of hosts that we want to automate with Ansible. Also called the Ansible Hosts File it tells Ansible about the hosts that it can connect to, we can also define custom options per host e.g. different port number or credentials

```
localhost ansible_user=kb ansible_ssh_pass=Welcome@123
ansible_become_pass=Welcome@123
```

inventory_Kuldip

```
localhost ansible_user=sanket-patil ansible_ssh_pass=sanket-patil
ansible_ssh_common_args='-o StrictHostKeyChecking=no'
```

inventory_Sanket

playbook.yml file- The tasks to be performed on the hosts are specified in below playbook.

```
- name: Use docker compose to run all containers
  hosts: all
  tasks:
    - name: Start docker service
      service:
        name: docker
        state: started

    - name: Copy Docker Compose file
      copy:
        src: docker-compose.yml
        dest: ./
```

```
  - name: Copy Env file
    copy:
      src: docker_namespace.env
      dest: ./

  - name: Run docker compose up command
    become: true
    docker_compose:
      project_src: ./
      state: present
      pull: yes
```

playbook.yml

12. Jenkins Pipeline

A Jenkins pipeline is a set of plug-ins to create automated, recurring workflows that constitute CI/CD pipelines ensuring that the changes in deployment are automatically reflected whenever changes are made in source code. It is a sequence of stages, each of which represents a distinct phase of build, test, and deployment process. Each stage can contain one or more steps, which are the individual tasks that need to be executed within that stage.

Jenkinsfile - It is a text file that contains the definition of your pipeline.
The Jenkinsfile can be stored in source code repository alongside the application code, allowing us to version control the pipeline and manage changes to it over time.

```
pipeline {
    environment {
        DOCKERHUB_CREDENTIALS = credentials('DockerHubCred')
        MYSQL_CREDENTIALS = credentials('MySqlCred')
        DOCKERHUB_USER = 'kb1110'
        EMAIL_TO = 'Kuldip.Bhatale@iiitb.ac.in'
    }
    agent any
    stages {
        stage('Clone repository') {
            steps {
                git branch: 'main', url: 'https://github.com/kb87-98/Online-Banking-System'
            }
        }
        stage('Maven Build RegistryService') {
            steps {
                echo 'Building RegistryService'
                sh 'cd ServiceRegistry && mvn clean install'
                sh 'mv -f ServiceRegistry/target/ServiceRegistry-0.0.1-SNAPSHOT.jar JarFiles/'
```

```
        }
    }
    stage('Maven Build APIGateway') {
        steps {
            echo 'Building APIGateway'
            sh 'cd ApiGateway && mvn clean install'
            sh 'mv -f ApiGateway/target/ApiGateway-0.0.1-SNAPSHOT.jar JarFiles/'
        }
    }
    stage('Maven Build Services') {
        parallel {
            stage('User Service') {
                steps {
                    echo 'Building User Service'
                    sh "cd UserService && mvn clean install
-DSPRING_DATASOURCE_USERNAME=${MYSQL_CREDENTIALS_USR}
-DSPRING_DATASOURCE_PASSWORD=${MYSQL_CREDENTIALS_PSW}"
                    sh 'mv -f UserService/target/UserService-0.0.1-SNAPSHOT.jar JarFiles/'
                }
            }
            stage('Account Service') {
                steps {
                    echo 'Building Account Service'
                    sh "cd AccountService && mvn clean install
-DSPRING_DATASOURCE_USERNAME=${MYSQL_CREDENTIALS_USR}
-DSPRING_DATASOURCE_PASSWORD=${MYSQL_CREDENTIALS_PSW}"
                    sh 'mv -f AccountService/target/AccountService-0.0.1-SNAPSHOT.jar JarFiles/'
                }
            }
            stage('Notification Service') {
                steps {
                    echo 'Building Notification Service'
                    sh "cd NotificationService && mvn clean install
-DSPRING_DATASOURCE_USERNAME=${MYSQL_CREDENTIALS_USR}
-DSPRING_DATASOURCE_PASSWORD=${MYSQL_CREDENTIALS_PSW}"
                    sh 'mv -f NotificationService/target/NotificationService-0.0.1-SNAPSHOT.jar JarFiles/'
                }
            }
        }
    }

    stage('Build Docker Images') {
        steps {
            echo 'Building Docker Images'
            sh "docker build -t ${DOCKERHUB_USER}/eurekaregistry -f DockerFiles/ServiceRegistryDockerfile ."
            sh "docker build -t ${DOCKERHUB_USER}/apigateway -f DockerFiles/APIGatewayServiceDockerfile ."
            sh "docker build -t ${DOCKERHUB_USER}/userservice -f DockerFiles/UserServiceDockerfile ."
            sh "docker build -t ${DOCKERHUB_USER}/accountservice -f DockerFiles/AccountServiceDockerfile ."
            sh "docker build -t ${DOCKERHUB_USER}/notificationservice -f DockerFiles/NotificationServiceDockerfile ."
            sh "docker build -t ${DOCKERHUB_USER}/frontend -f DockerFiles/FrontendDockerfile ."
        }
```

```
        }

        stage('Login to Docker Hub') {
            steps {
                echo 'Login to Docker Hub'
                withCredentials([usernamePassword(credentialsId: 'DockerHubCred', usernameVariable: 'DOCKERHUB_USER',
passwordVariable: 'DOCKERHUB_PASS')]) {
                    sh "docker login -u ${DOCKERHUB_USER} -p ${DOCKERHUB_PASS}"
                }
            }
        }

        stage('Push Images to Docker Hub') {
            steps {
                echo 'Pushing Images to Docker Hub'
                sh "docker push ${DOCKERHUB_USER}/eurekaregistry"
                sh "docker push ${DOCKERHUB_USER}/apigateway"
                sh "docker push ${DOCKERHUB_USER}/userservice"
                sh "docker push ${DOCKERHUB_USER}/accountservice"
                sh "docker push ${DOCKERHUB_USER}/notificationservice"
                sh "docker push ${DOCKERHUB_USER}/frontend"
            }
        }

        stage('Clean Up Local Images') {
            steps {
                echo 'Cleaning Up Local Docker Images'
                sh "docker rmi ${DOCKERHUB_USER}/eurekaregistry"
                sh "docker rmi ${DOCKERHUB_USER}/apigateway"
                sh "docker rmi ${DOCKERHUB_USER}/userservice"
                sh "docker rmi ${DOCKERHUB_USER}/accountservice"
                sh "docker rmi ${DOCKERHUB_USER}/notificationservice"
                sh "docker rmi ${DOCKERHUB_USER}/frontend"
            }
        }

        stage('Run ansible playbook'){
            steps{
                echo 'Running the ansible playbook yml file'
                sh 'export LC_ALL=en_IN.UTF-8;export LANG=en_US.UTF-8;ansible-playbook -i inventory_Kuldip
playbook.yml'
            }
        }
    }
    post {
        success {
            emailext body: 'Check console output at $BUILD_URL to view the results. \n\n ${CHANGES} \n\n
---------------------------------------------- \n${BUILD_LOG, maxLines=100, escapeHtml=false}',
                to: "${EMAIL_TO}",
                subject: 'Build succeed in Jenkins: $PROJECT_NAME - #$BUILD_NUMBER'
        }
```

```
        failure {
            emailext body: 'Check console output at $BUILD_URL to view the results. \n\n ${CHANGES} \n\n
---------------------------------------------------- \n${BUILD_LOG, maxLines=100, escapeHtml=false}',
                to: "${EMAIL_TO}",
                subject: 'Build failed in Jenkins: $PROJECT_NAME - #$BUILD_NUMBER'
        }


    }
}
```

Jenkinsfile

The environment variables contain the credentials required to login to various accounts used like docker hub and mysql.

Stages-

- **Clone repository**: This stage retrieves the code from the GitHub repository for further processing in the pipeline.

- **Maven Build RegistryService**: compiles and packages the RegistryService module of the application using Maven, generating a JAR file for deployment.

- **Maven Build APIGateway:** Similar to the previous stage, this one builds the APIGateway module of the application using Maven, creating a deployable JAR file.

- **Maven Build Services:** This parallel stage builds multiple service modules of the application concurrently, including User Service, Account Service, and Notification Service, configuring them with MySQL credentials.

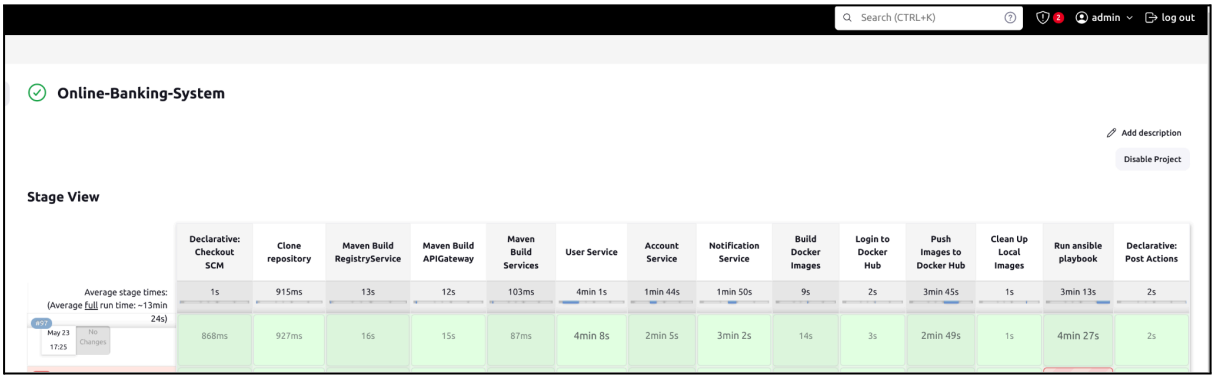
- **Build Docker Images:** This stage constructs Docker images for each service and the frontend component of the application using Dockerfiles provided in the project.

- **Login to Docker Hub:** This stage authenticates with Docker Hub using the credentials for pushing the built Docker images.

- **Push Images to Docker Hub:** Once authenticated, this stage uploads the Docker images to Docker Hub for distribution and deployment.

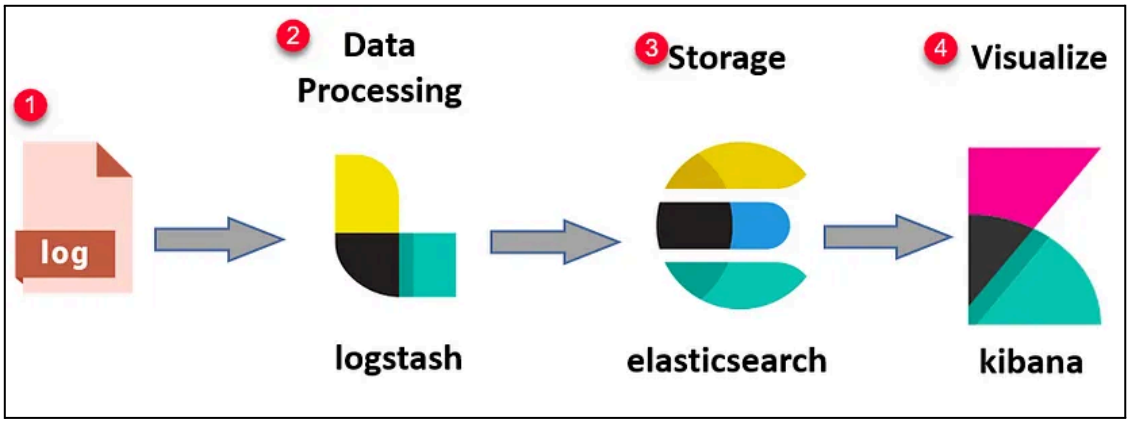
- **Clean Up Local Images:** After pushing images to Docker Hub, this stage removes the locally built Docker images to free up disk space.

- **Run ansible playbook:** Executes an Ansible playbook to automate deployment and configuration tasks on target servers specified in the inventory file.



13. ELK Stack:

The ELK Stack is a collection of three open-source tools—Elasticsearch, Logstash, and Kibana—used for searching, analyzing, and visualizing log data in real-time.



1. Elasticsearch: At the core of the ELK Stack is Elasticsearch, which is a distributed, RESTful search and analytics engine. Elasticsearch is designed for horizontal scalability, allowing it to handle large volumes of data across multiple nodes. It's used for indexing and searching structured and unstructured data, making it ideal for log analysis.
2. Logstash: Logstash is a data processing pipeline that ingests, transforms, and sends log data to Elasticsearch. It's responsible for collecting logs from various sources, parsing them, enriching them, and formatting them into a common format that Elasticsearch can understand. Logstash supports a wide range of inputs, filters, and outputs, making it highly flexible and customizable.

3. Kibana: Kibana is a powerful visualization and analytics platform that sits

on top of Elasticsearch. It provides a user-friendly interface for exploring, analyzing, and visualizing data stored.
For generating logs, add the following dependencies to your pom.xml file:

```
<dependency>
   <groupId>net.logstash.logback</groupId>
   <artifactId>logstash-logback-encoder</artifactId>
   <version>7.0</version>
</dependency>

<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-api</artifactId>
</dependency>
```

Flow of execution:

Spring Application Sends Logs Directly to tcp port 5001
It is configured in logback.xml

```
<!-- New Logstash TCP appender -->
<appender name="LOGSTASH"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">

<destination>${LOGGING_LOGSTASH_HOST:-localhost}:${LOGGING_LOGSTASH_PORT:-500
1}</destination>
   <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
</appender>
```

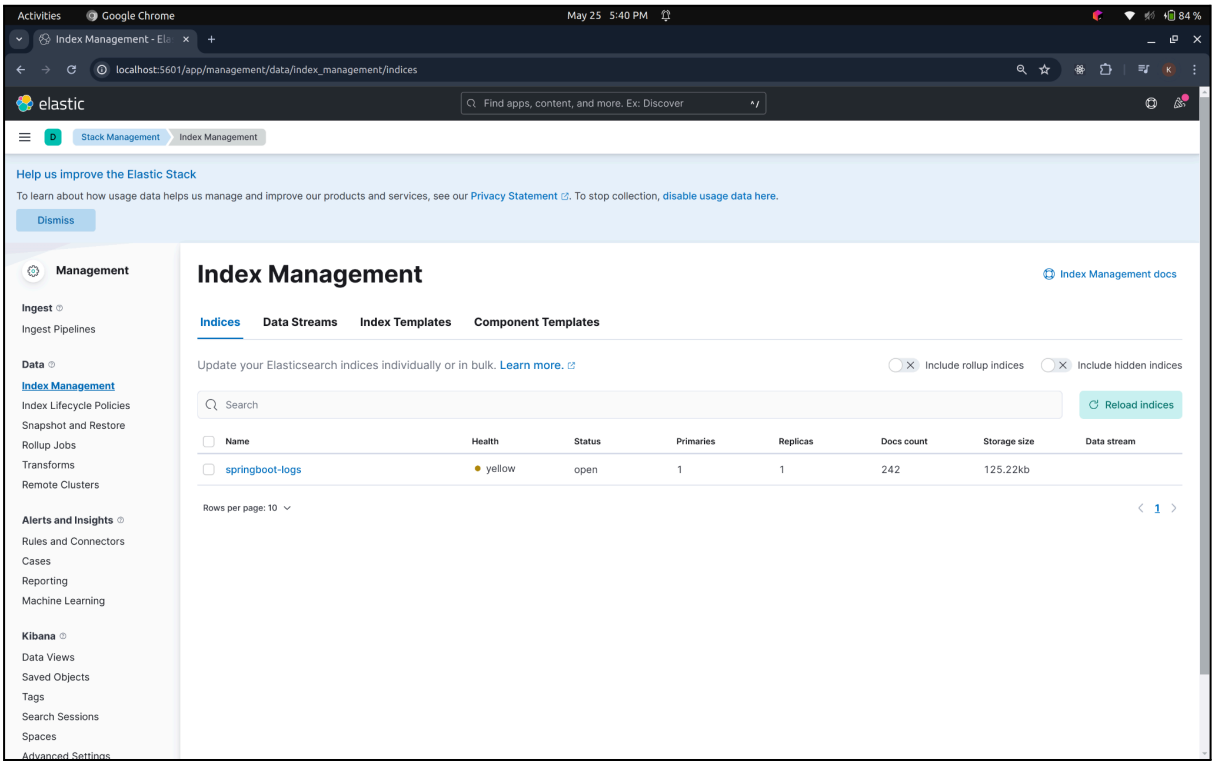
Logstash pipeline configured to listen from port 5001 and forward logs to port 9200

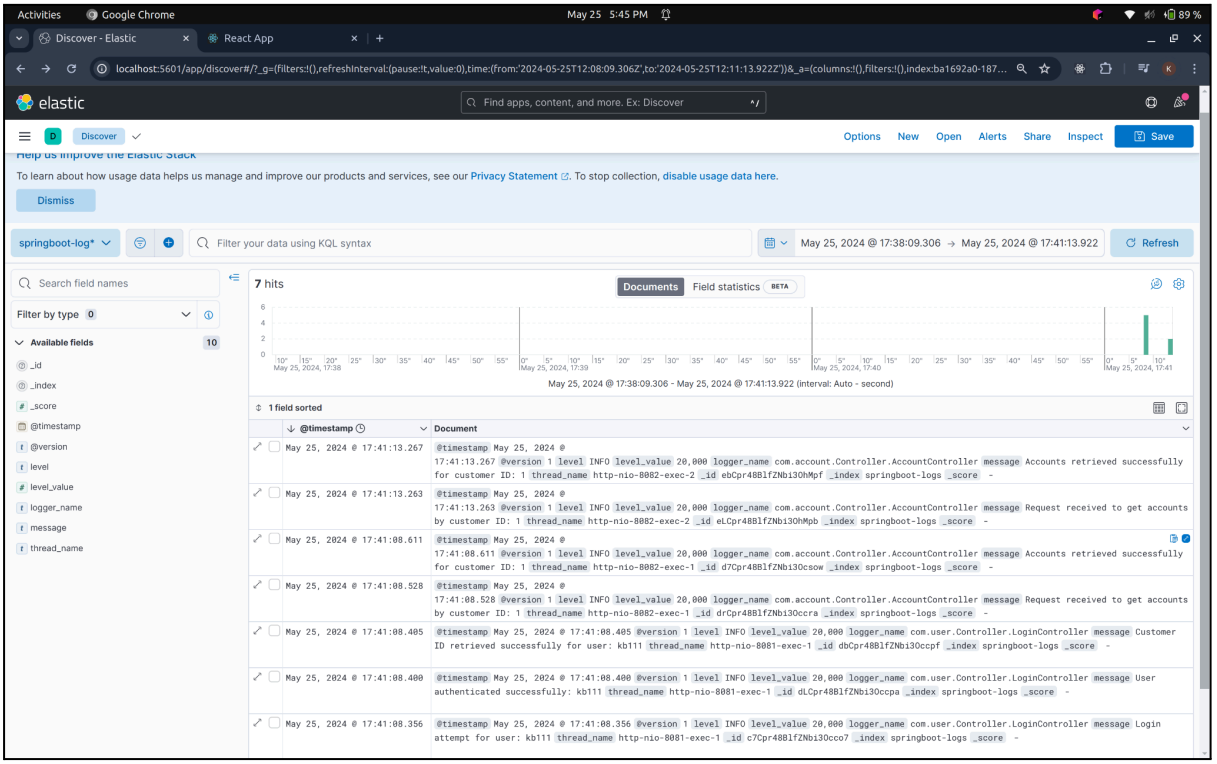
```
input {
 tcp {
   port => 5001
   codec => json
 }
}


output {
 elasticsearch {
   hosts => ["http://elasticsearch:9200"]
```

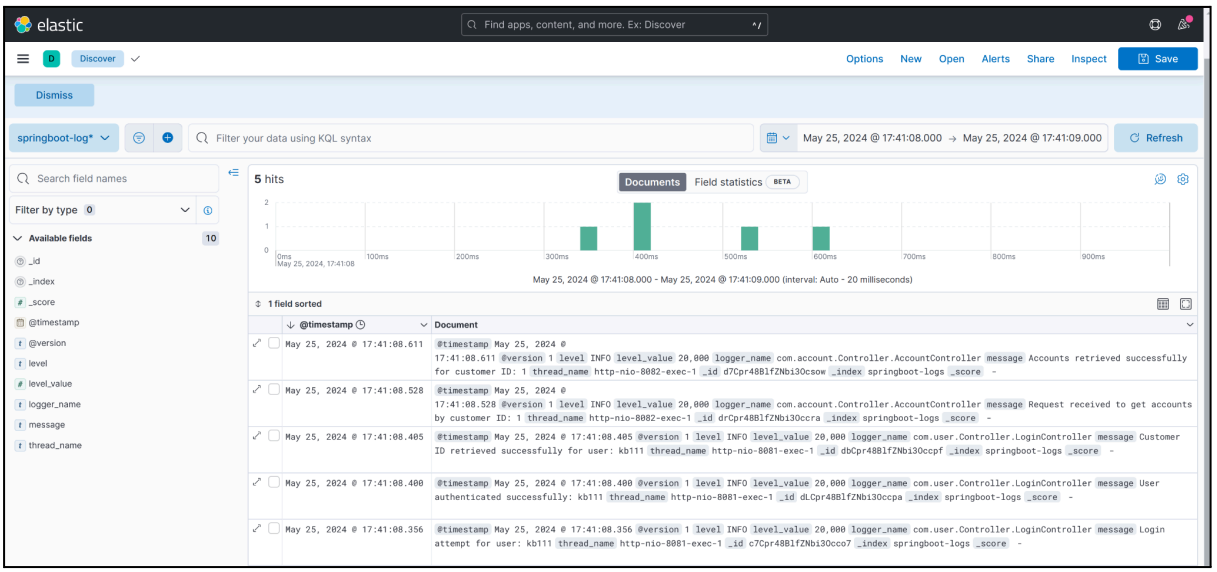
```
    # index =>
"%{[fields][application]}-%{[fields][service]}-%{+YYYY.MM.dd}"
    # This will create index names like
"userservice-complaintservice-2024.05.20"

    index => "springboot-logs"
  }
  stdout { codec => rubydebug }
}
```

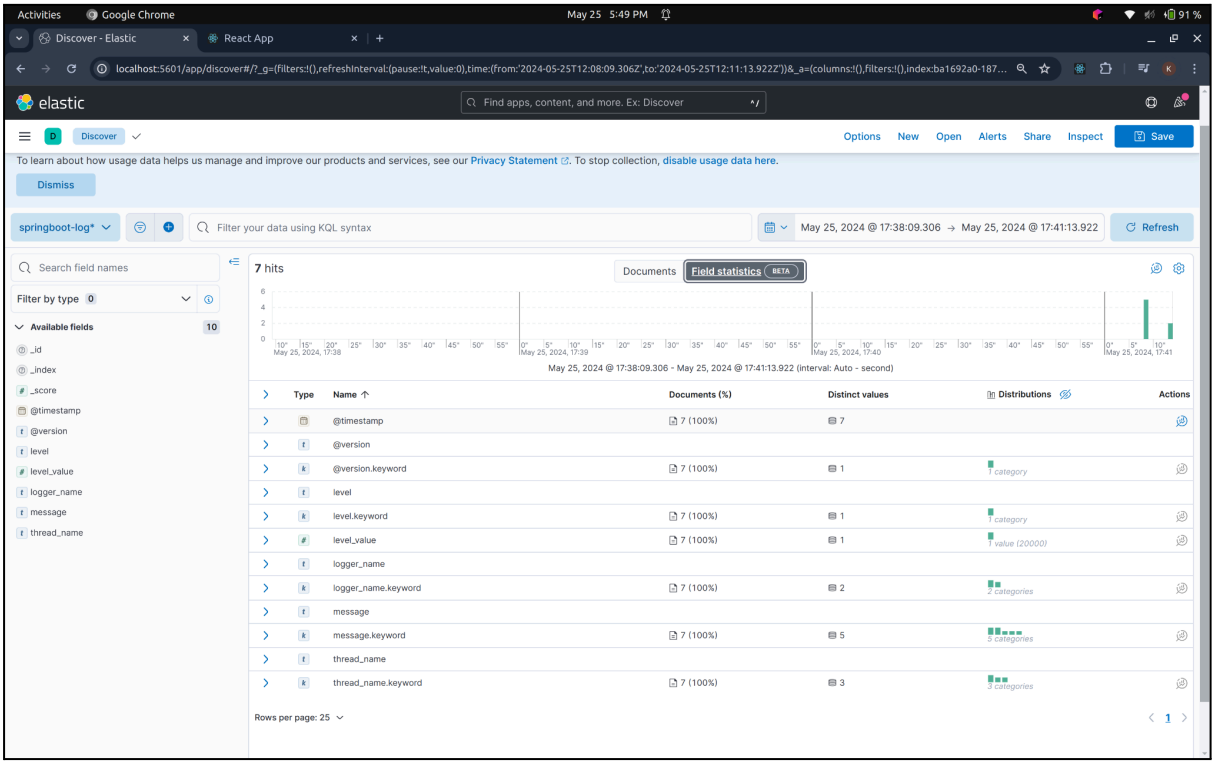
In kibana logs are received by the index named "springboot-logs"

Following is the log representation in Kibana:




Activities
Google Chrome
May 25 5:45 PM
89 %
Discover - Elastic
React App
localhost:5601/app/discover#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:'2024-05-25T12:08:09.306Z',to:'2024-05-25T12:11:13.922Z'))&_a=(columns:!(),filters:!(),index:ba1692a0-187...
elastic
Find apps, content, and more. Ex: Discover
Discover
Options
New
Open
Alerts
Share
Inspect
Save
To learn about how usage data helps us manage and improve our products and services, see our Privacy Statement. To stop collection, disable usage data here.
Dismiss
springboot-log*
Filter your data using KQL syntax
May 25, 2024 @ 17:38:09.306 → May 25, 2024 @ 17:41:13.922
Refresh
Search field names
Filter by type 0
Available fields 10
_id
_index
_score
@timestamp
@version
level
level_value
logger_name
message
thread_name
7 hits
Documents
Field statistics BETA
May 25, 2024 @ 17:38:09.306 - May 25, 2024 @ 17:41:13.922 (interval: Auto - second)
1 field sorted
@timestamp
Document
May 25, 2024 @ 17:41:13.267
May 25, 2024 @ 17:41:13.263
May 25, 2024 @ 17:41:08.611
May 25, 2024 @ 17:41:08.528
May 25, 2024 @ 17:41:08.405
May 25, 2024 @ 17:41:08.400
May 25, 2024 @ 17:41:08.356


elastic
Find apps, content, and more. Ex: Discover
Discover
Options
New
Open
Alerts
Share
Inspect
Save
Dismiss
springboot-log*
Filter your data using KQL syntax
May 25, 2024 @ 17:41:08.000 → May 25, 2024 @ 17:41:09.000
Refresh
Search field names
Filter by type 0
Available fields 10
_id
_index
_score
@timestamp
@version
level
level_value
logger_name
message
thread_name
5 hits
Documents
Field statistics BETA
May 25, 2024 @ 17:41:08.000 - May 25, 2024 @ 17:41:09.000 (interval: Auto - 20 milliseconds)
1 field sorted
@timestamp
Document
May 25, 2024 @ 17:41:08.611
May 25, 2024 @ 17:41:08.528
May 25, 2024 @ 17:41:08.405
May 25, 2024 @ 17:41:08.400
May 25, 2024 @ 17:41:08.356



14. API Documentation

End Point	HTTP Method	Request Body	Description
/auth/login	POST	Customer credentials (Object)	After registration, customer can login to the bank application
/api/customer	POST	Customer details (Object)	Customers can register here.
/api/customer/all	GET	Customers Details (Object)	To get all the customers inside the system along with their details.
/api/customer/{c_id}	GET	Customer Details (Object)	To get a particular customer inside the system along with his details.
/api/accounts	POST	Account details (Object)	Customers can create new bank account.
/api/accounts/all	GET	All accounts in system	To retrieve all bank accounts created from the system

/api/accounts/{accountId}	GET	Account Details (Object)	To get the details of the account belongs to provided accountId.
/api/accounts/customer/{customerId}	GET	Account Details(Object)	To get all the bank accounts belongs to the customerId.
/api/accounts/transfer	POST	Transfer details(Object)	To transfer money between two bank accounts.
/api/notifications/send	POST	Notification Details Body(Object)	To send email notifications for each transaction to the receiver.
/api/notifications/all	GET	Getting all the sent notifications	To get all details about the notifications from the system.

Conclusion and Future Work

Conclusion:

- The Online Banking System project successfully delivered a robust, scalable, and user-friendly platform.
- The project employed a microservices based backend using Java with Spring Boot, ensuring scalability, maintainability, and efficient resource management.
- For communication between Services **Feign clien**t is used which allows developers to make HTTP requests to RESTful APIs with minimal code and configuration by using annotations and interfaces to define the desired HTTP endpoints and request parameters.
- Key functionalities such as money transfer, transaction history, account management, and user notifications were seamlessly integrated, providing users with a comprehensive banking experience.
- The frontend, developed with **React.js**, offers a responsive and intuitive user interface.
- By implementing a suite of DevOps tools and practices, including **Jenkins** for continuous integration, **Docker and Docker Compose** for containerization, and **Ansible** for configuration management, the project ensured efficient deployment and maintenance processes.
- Centralized logging with the **ELK stack** (Elasticsearch, Logstash, and Kibana) facilitated effective monitoring and troubleshooting. The use of Git for version control enabled effective collaboration and streamlined code management.

Overall, the Online Banking System project achieved its goal of creating a scalable, maintainable, and user-friendly online banking platform.

Future Work:

1. Adding more features to the system.
2. Improving Security and overall transaction management.
3. Scaling with Kubernetes, enhancing security, expanding features, improving user experience, and fostering community engagement, ensuring OnlineBankManagement remains a valuable tool for all the customers.

References

1. https://docs.docker.com/compose/
2. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html
3. https://www.jenkins.io/doc/book/pipeline/
4. https://medium.com/@devops_83824/introduction-to-docker-compose-934238b14c13
5. https://medium.com/yavar/elk-stack-architecture-and-how-does-it-works-bda6382b3c83
6. https://www.youtube.com/watch?v=Nb2PPuqtL2g&t=2556s