Fintech Platform Reference Implementation

Arun Patra

Version 1.0.0, 2021.04.27

Table of Contents

Chapter 1. Introduction

This project implements a hypothetical Fintech Platform. It demonstrates Microservice architecture, design, best practices and production ready features. It was developed as part of a code challenge from Reloadly.

1.1. Source Code

 Head over to the GitHub repository here.

Chapter 2. Code Challenge Summary

The solution developed for the code challenge, embraces modern architectural styles, battle tested architectural principles and robust engineering practices. The following features have been implemented.

Items grayed out, are proposed. Please see open issues in GitHub here.

Chapter 3. Code Challenge Highlights

- **~ 12000 lines of code** covered by JUnit tests.
- Modern tech stack, event driven architecture and solid design principles.
- Fully leverages Spring Boot Technology.
- Advanced Spring Boot features including auto-configuration.
- Ultra scalable and resilient transaction processor built on Kafka technology.
- End to end enterprise security using dedicated Authentication Service designed to be used across the enterprise.
- Production ready features, e.g. Spring Boot Admin.
- Extensive documentation.

Chapter 4. Getting Started

This section provides documentation on setting up a local installation of the Reloadly Microservices. Create an issue on GitHub here if you have questions or need help.

4.1. Pre-requisites

Following pre-requisites need to be fulfilled before you can run a fully functional instance of the platform on your local development environments. The platform has been tested on macOS Big Sur.

1. **Hardware**: Either physical or virtual machines with minimum 8 GB RAM and 4 CPU Cores *(16 GB RAM, 8 CPU Cores if you want to do performance testing)*.
2. **Operating System**: Windows(version 10+), macOS(version 10.15+), Linux (CentOS, RHEL, Ubuntu)
3. **Java** - JDK 8, or JDK 11.
4. **Build Tool** - Maven 3.6 or above
5. **Database** - MySQL, locally running or remotely hosted.
6. **Kafka** - Local instance with the broker server listening on port 9092. A remotely hosted Kafka cluster is acceptable as well.

Installation on macOS

- Install JDK by following the instructions here. Please install JDK 8 or 11 only. Other JDK versions have not been tested.
- Install Apache Maven by following the instructions here.
- Install Apache Kafka by following the instructions here.
- Install MySQL by following the instructions here.

4.2. Build From Source

Clone the git repository from GitHub into a directory of your choice.

```
git clone git@github.com:arunkpatra/reloadly-services.git
```

Build code and run tests. Build should pass, for you to proceed further with the rest of the chapter. From the directory where you cloned the repository, issue the following commands.

```
cd reloadly-services
mvn clean install
```

The above commands will build code and run integration tests. The build artifacts are installed to your local maven repository. For building and running tests, you do NOT need a locally running Kafka and MySQL instance. For running tests, in-memory instances of these resources are run.

4.3. Development Environment Setup

This is useful, when you want to set a development environment and run all the microservices on a single local machine. Kubernetes deployment support is coming up. See this GitHub issue.

These instructions are for macOS. Please tailor them to run in other environments like Windows and Linux.

4.3.1. Starting Local Services

Following services need to be started before you run microservices.

MySQL Setup

Issue the following commands. You should have installed MySQL locally and verified it to work correctly as a pre-requisite.

```
mysql.server start
```

You must secure your local MySQL installation and create a user with database wide access. You can use the admin user for now, on your local machine. See MySQL docs for more information. Take a note of the username and password of the admin user; you will need it in a later step.

All database tables used by the microservices need to be created upfront in your local MySQL database instance. As noted earlier, you do NOT need to do this for running integration tests since tests use in-memory Derby database.

Run the following scripts in order.

Authentication Microservice Schema

```
CREATE DATABASE IF NOT EXISTS rlauthdb;
USE rlauthdb;

DROP TABLE IF EXISTS api_key_table;
DROP TABLE IF EXISTS username_password_table;
DROP TABLE IF EXISTS authority_table;
DROP TABLE IF EXISTS role_table;
DROP TABLE IF EXISTS user_table;

--
```

```
-- User table. Contains unique UIDs.
--
CREATE TABLE user_table
(
    id     BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    uid    VARCHAR(40)           NOT NULL UNIQUE,
    active BOOLEAN DEFAULT true
);

--
-- Valid roles that may be assigned to a user.
--
create table role_table
(
    id        BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    role_name VARCHAR(32)           NOT NULL UNIQUE
);

--
-- Authorities assigned to a user. Authorities are equivalent to roles in the current
context.
--
CREATE TABLE authority_table
(
    id        BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    uid       VARCHAR(40)           NOT NULL,
    authority VARCHAR(32)           NOT NULL,
    CONSTRAINT unique_user_id_authority UNIQUE (uid, authority),
    CONSTRAINT fk_igc_authority_user_name_igc_user_user_name
        FOREIGN KEY (uid)
            REFERENCES user_table (uid),
    CONSTRAINT fk_igc_authority_authority_igc_role_role_name
        FOREIGN KEY (authority)
            REFERENCES role_table (role_name)
);

--
-- Table supporting a username/password based authentication. Passwords are encrypted.
--
CREATE TABLE username_password_table
(
    id        BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    user_name VARCHAR(64)           NOT NULL UNIQUE,
    password  VARCHAR(128)          NOT NULL,
    uid       VARCHAR(40)           NOT NULL,
    CONSTRAINT fk_username_password_table_user_id_user_table
        FOREIGN KEY (uid)
            REFERENCES user_table (uid)
);

create TABLE api_key_table
```

```
(
    id      BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    uid     VARCHAR(40)           NOT NULL,
    api_key VARCHAR(40)           NOT NULL UNIQUE,
    active  BOOLEAN DEFAULT true,
    CONSTRAINT fk_api_key_table_uid_user_table
        FOREIGN KEY (uid)
            REFERENCES user_table (uid)
)
```

Account Microservice Schema

```
CREATE DATABASE IF NOT EXISTS rlacctdb;
USE rlacctdb;

DROP TABLE IF EXISTS address_table;
DROP TABLE IF EXISTS account_balance_table;
DROP TABLE IF EXISTS account_table;

--
-- Accounts table.
--
CREATE TABLE account_table
(
    id           BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    uid          VARCHAR(40)           NOT NULL UNIQUE, -- Only one account per user
ID
    account_id   VARCHAR(40)           NOT NULL UNIQUE,
    name         VARCHAR(128)          NOT NULL,
    email        VARCHAR(128)          NOT NULL UNIQUE,
    phone_number VARCHAR(20)           NOT NULL UNIQUE,
    currency_cd  VARCHAR(3)            NOT NULL DEFAULT 'USD',
    active       BOOLEAN                         DEFAULT TRUE
);

--
-- Account Balance table.
--
CREATE TABLE account_balance_table
(
    id              BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    account_id      VARCHAR(40)           NOT NULL UNIQUE,
    account_balance FLOAT                 NOT NULL DEFAULT 0.0,
    CONSTRAINT fk_acct_balance_table_acct_id_acct_table
        FOREIGN KEY (account_id)
            REFERENCES account_table (account_id)
);

--
-- Address table.
```

```
--
CREATE TABLE address_table
(
    id             BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    account_id     VARCHAR(40)           NOT NULL,
    address_type   VARCHAR(40)           NOT NULL DEFAULT 'BILLING',
    address_line_1 VARCHAR(40)           NOT NULL,
    address_line_2 VARCHAR(40),
    city           VARCHAR(64)           NOT NULL,
    state          VARCHAR(64)           NOT NULL,
    postal_code    VARCHAR(20)           NOT NULL,
    country        VARCHAR(64),
    CONSTRAINT fk_address_table_acct_id_acct_table
        FOREIGN KEY (account_id)
            REFERENCES account_table (account_id)
);
```

Transaction Microservice Schema

The following script, sets up the tables needed for both the `transaction-processor` and the `transaction-service`. See later sections for details.

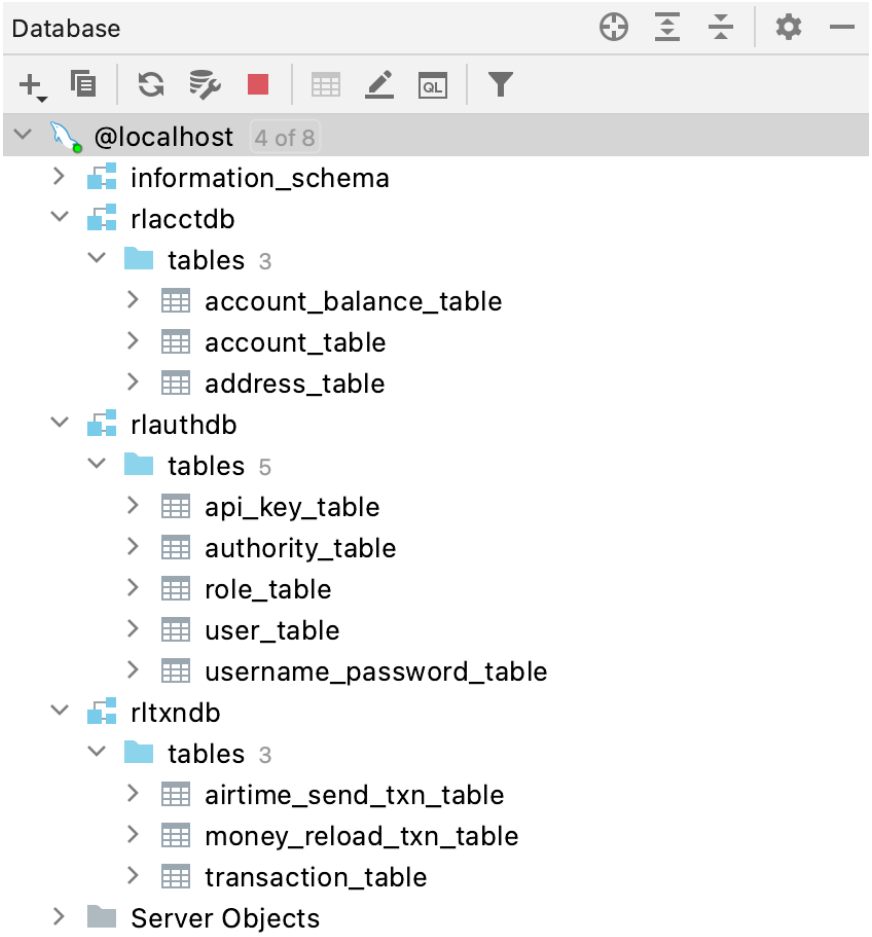
```
CREATE DATABASE IF NOT EXISTS rltxndb;
USE rltxndb;

DROP TABLE IF EXISTS money_reload_txn_table;
DROP TABLE IF EXISTS airtime_send_txn_table;
DROP TABLE IF EXISTS transaction_table;
--
-- Transaction table.
--
CREATE TABLE transaction_table
(
    id         BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    uid        VARCHAR(40)                          NOT NULL,
    txn_id     VARCHAR(40)                          NOT NULL UNIQUE,
    txn_type   VARCHAR(16)                          NOT NULL,
    txn_status VARCHAR(16)                          NOT NULL
);

CREATE TABLE money_reload_txn_table
(
    id     BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    txn_id VARCHAR(40)                           NOT NULL UNIQUE,
    amount FLOAT                                 NOT NULL,
    CONSTRAINT fk_money_reload_txn_table_txn_id
        FOREIGN KEY (txn_id)
            REFERENCES transaction_table (txn_id)
);

CREATE TABLE airtime_send_txn_table
(
    id           BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    txn_id       VARCHAR(40)                           NOT NULL UNIQUE,
    amount       FLOAT                                 NOT NULL,
    phone_number VARCHAR(16)                           NOT NULL,
    CONSTRAINT fk_airtime_send_txn_table_txn_id
        FOREIGN KEY (txn_id)
            REFERENCES transaction_table (txn_id)
);
```

Seed Data

This data must be seeded into the DB for system to work correctly. It does nothing other than ading a service account and API key which is needed for inter-microservice communication.

```
--
-- Data goes into the Auth Database
--
USE rlauthdb;

--
-- These roles must exist in the system, for application to work.
--
INSERT INTO role_table(role_name)
VALUES ('ROLE_USER'),
       ('ROLE_ADMIN');

--
-- This is for a Reloadly Service Account, which the system needs.
--
INSERT INTO user_table(uid, active)
VALUES ('c1fe6f0d-420e-4161-a134-9c2342e36c95', true);

--
-- Service Account authorities.
--
INSERT INTO authority_table(uid, authority)
VALUES ('c1fe6f0d-420e-4161-a134-9c2342e36c95', 'ROLE_USER'),
       ('c1fe6f0d-420e-4161-a134-9c2342e36c95', 'ROLE_ADMIN');

--
-- Service Account credentials.
--
INSERT INTO username_password_table(user_name, password, uid)
VALUES ('reloadly_svc_acct',
'$2a$10$OwuE74o7m2qCj7yTVgleGOhwuqrvbJZT2qwYCGtyUi6ITSjSqHEZy',
        'c1fe6f0d-420e-4161-a134-9c2342e36c95');

--
-- API Key issued to 'reloadly_svc_acct' service account.
--
INSERT INTO api_key_table(uid, api_key, active)
VALUES ('c1fe6f0d-420e-4161-a134-9c2342e36c95', 'd3fe6f0d-120e-4161-a134-8c2342e36ca6
', true);
```

After running these scripts, you must have the schemas correctly created and seed data populated. Please verify. The following image shows the tables in IntelliJ Idea Data Explorer (you can use any tool of your choice).



Kafka Setup

Issue the following commands. You should have installed Kafka locally and verified it to work correctly as a pre-requisite.

```
zookeeper-server-start -daemon /usr/local/etc/kafka/zookeeper.properties & kafka-server-start /usr/local/etc/kafka/server.properties
```

A topic named `com.reloadly.inbound.txn.topic` is needed by the `transaction-processor` (Kafka consumer) and `transaction-service`(Kafka producer) components. This topic is created on the fly by the transaction-service` if it does not exist. While this is acceptable in non-production environments, it is discouraged in higher environments. The `transaction-processor` listens to this topic and won't process messages if this topic does not exist.

In your local environment, Kafka uses PLAINTEXT by default. In other environments, you must use strong encryption. Please refer Kafka documentation for securing a Kafka installation.

4.3.2. Running Microservices Locally

Follow the steps listed below in order to run all components of the platform locally. You should folow these steps sequentially.

Environment Variable Configuration

You would need to set the following Environment variables. Use the method that's appropriate for your operating system. On macOS, you can export environment variables on the command line as follows.

```
export YOUR_ENV_VARIABLE=env_var_value
```

Environment Variable	Value	Notes
DB_USER	The MySQL DB username	Used by all microservices that access the database.
DB_PASSWORD	The MySQL DB password	Used by all microservices that access the database.
AWS_ACCESS_KEY_ID	Your AWS account Access key ID	Needed to send email via Amazon SES
AWS_SECRET_ACCESS_KEY	Your AWS account Secret access key.	Needed to send email via Amazon SES.
EMAIL_DRY_RUN	Set to `true` or `false`	Setting `false` suppresses email sending. `true` will send actual emails. You need to configure SES correctly in your AWS account. See Amazon SES documentation.
SENDER_EMAIL_ID	The sender email ID verified by AWS SES to send emails from.	Needed to send email via Amazon SES.
TWILIO_DRY_RUN	Set to `true` or `false`	Setting `false` suppresses SMS sending. `true` will send actual SMS messages.
TWILIO_ACCOUNT_SID	Your Twilio account SID. Get it from your Twilio account.	Needed to send SMS via Twilio (SMS charges will be applied by Twilio).
TWILIO_AUTH_TOKEN	Your Twilio account auth token. Get it from your Twilio account.	Needed to send SMS via Twilio.
TWILIO_MSG_SVC_ID	Your Twilio account Message Service ID. Get it from your Twilio account.	Needed to send SMS via Twilio.
RELOADLY_SVC_ACCT_API_KEY	The reloadly service account API key. Use the value you inserted in the Seed Data section.	Needed for the platform to work correctly. Use the `api-key` value from the final `INSERT` statement in that script.

Consider writing a small shell script, or add these in your `~/.zprofile` of `~./bashprofile` file or a suitable method of your liking.

Running Microservices

Run these services in sequence (recommended). You must have done a `mvn clean install` from the root of your cloned repository. That builds all microservices.

Ensure that, your MySQL server and Kafka instances are up and running before you proceed with the following steps.

Authentication Microservice

Issue these commands from the root of your cloned repository.

```
cd authentication-service
mvn spring-boot:run
```

Access Swagger UI of Authentication Microservice at localhost:9090/swagger-ui/

Account Microservice

Issue these commands from the root of your cloned repository.

```
cd account-service
mvn spring-boot:run
```

Access Swagger UI of Authentication Microservice at localhost:8080/swagger-ui/

Transaction Microservice

Issue these commands from the root of your cloned repository.

```
cd transaction-service
mvn spring-boot:run
```

Access Swagger UI of Authentication Microservice at localhost:8081/swagger-ui/

Notification Microservice

Issue these commands from the root of your cloned repository.

```
cd notification-service
mvn spring-boot:run
```

Access Swagger UI of Authentication Microservice at localhost:8082/swagger-ui/

Transaction Processor

Issue these commands from the root of your cloned repository.

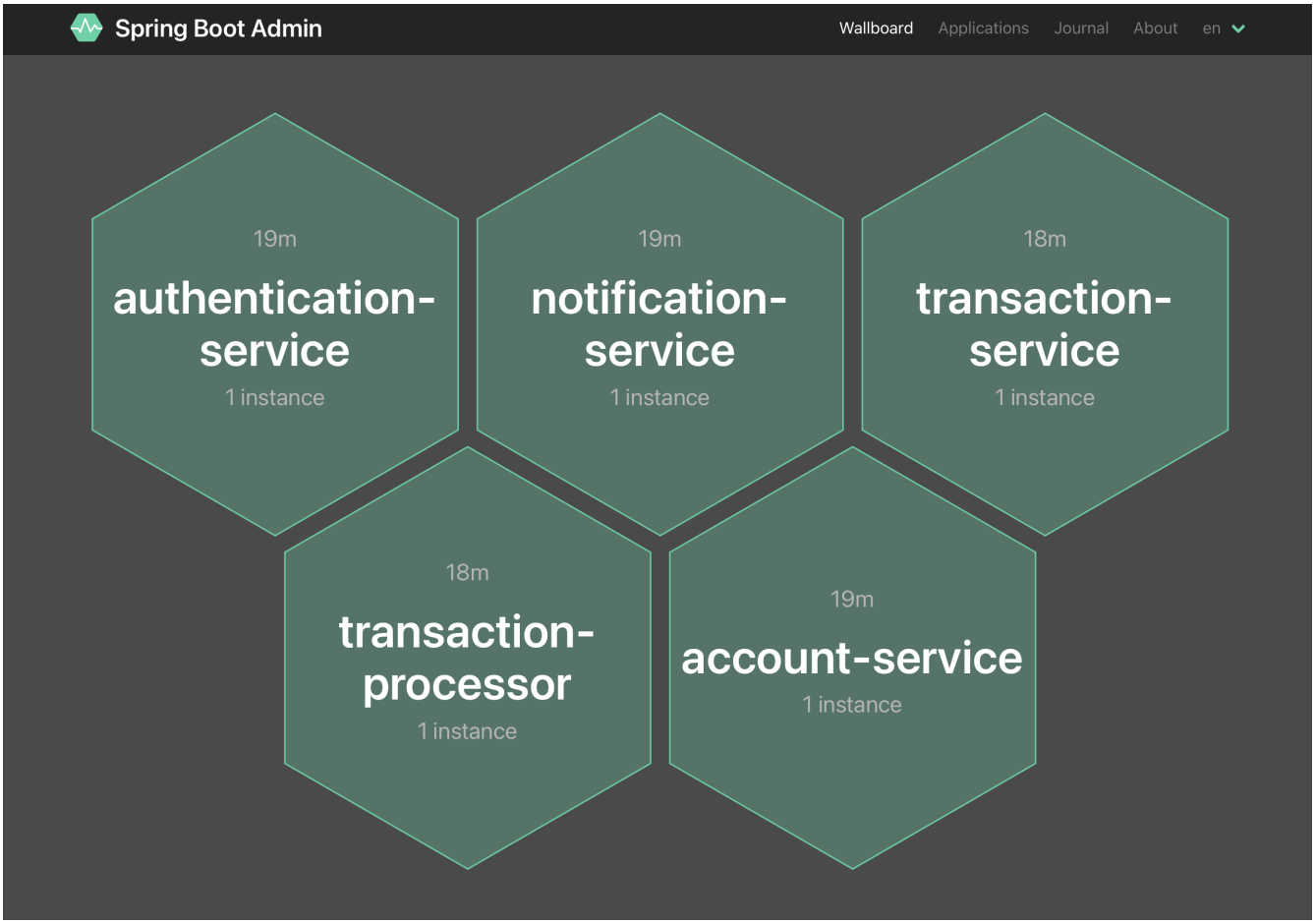
```
cd transaction-processor
mvn spring-boot:run
```

Spring Boot Admin App

Issue these commands from the root of your cloned repository.

```
cd admin-service
mvn spring-boot:run
```

Access the Admin Service Dashboard at localhost:9595/. You should see the following dashboard once all your services register with the Spring Boot Admin app. You should navigate to the `Wallboard` tab of the Spring Boot Admin UI.

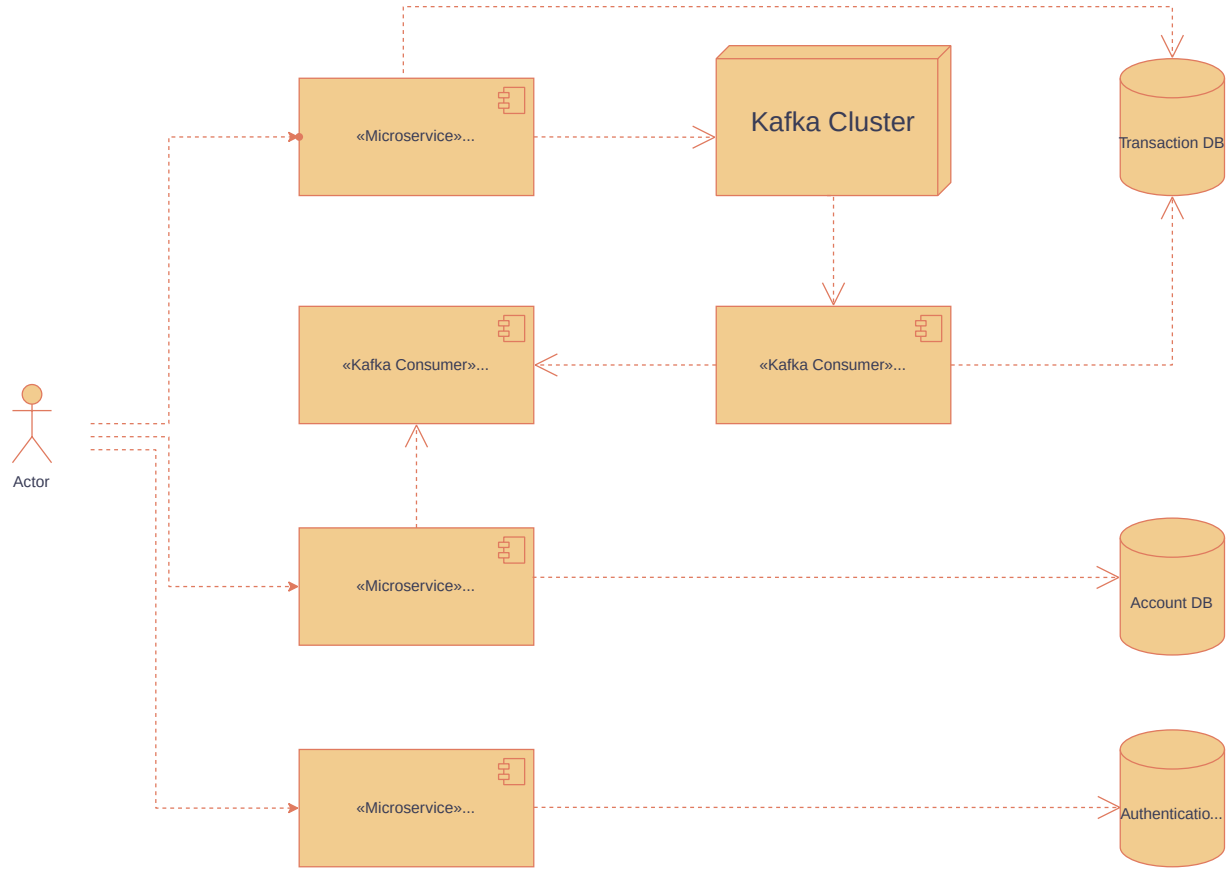


Congratulations! Your services are running.

Now head over to the documentation of individual microservices for more details.

Chapter 5. Architecture

The Architecture envisioned has been depicted in the illustration below.



Viewer does not support full SVG 1.1

5.1. Salient Points

- The client facing microservices are Authentication, Account and Transaction.
- The Transaction microservice, posts inbound transactions to Kafka and also maintains transaction status in its DB. Transaction microservice is this a Kafka Producer.
- The Transaction Processor is a Kafka Consumer and potentially runs a number of instances. All instances are part of a Kafka consumer group.
- Transaction processor updates transaction status once processed. Transactions can be retried.
- Notification microservice is called internally by other microservices and transaction processor.
- Authentication microservice is used to issue JWT tokens, User signup, token verification, API key verification etc.
- All microservices are secured using Spring Security. Inbound HTTP requests into any microservice are expected to carry either an Auth service issued JWT token, or a valid API key. Security infrastructure will validate tokens with Auth service.

5.2. Design

The Authentication, Account, Transaction and Notification microservices are Spring Boot applications. Each microservice embrases Domain Driven Design (DDD). Each microservice is responsible for managing data for the domain model it manages.

Additionally, for monitoring, A Spring Boot Admin app is utilized (not shown in illustration above); its optional.

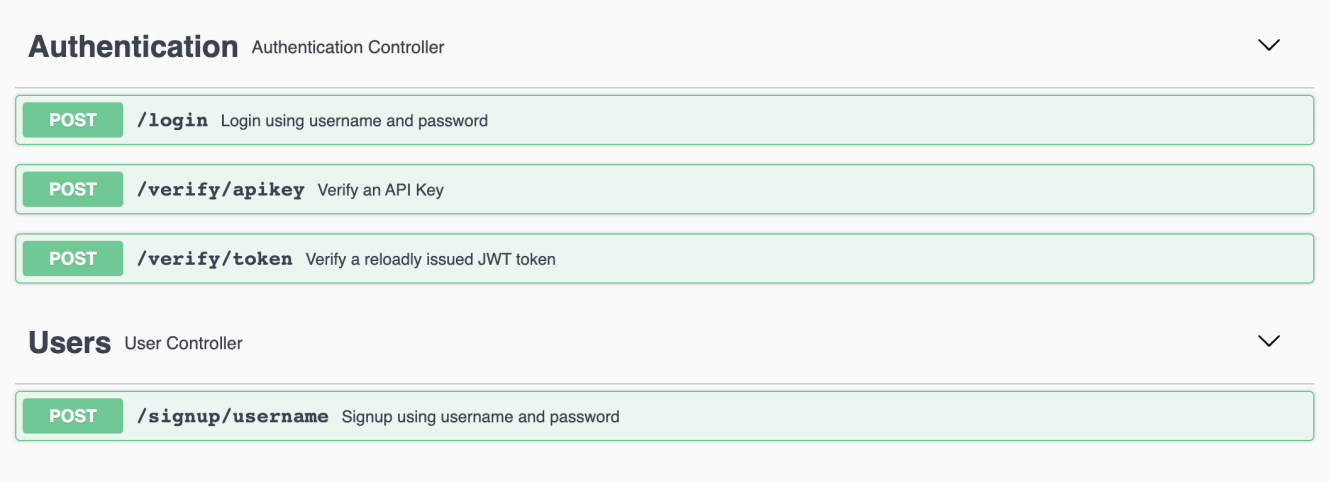
Chapter 6. Reloadly Services

6.1. Authentication Microservice

The Authentication service allows multiple enterprise facilities like:

- User signup - Currently username and password based login is supported. Can be extended to other auth channels. Users are issued JWT tokens.
- Signup - Users can signup by choosing a username and password.
- Token Verification - Verify JWT tokens.
- API Key verification and issue functions. API keys are issued to users and service accounts.
- All other microservices call into the Auth Service to validate tokens and API keys. It is a foundational service.

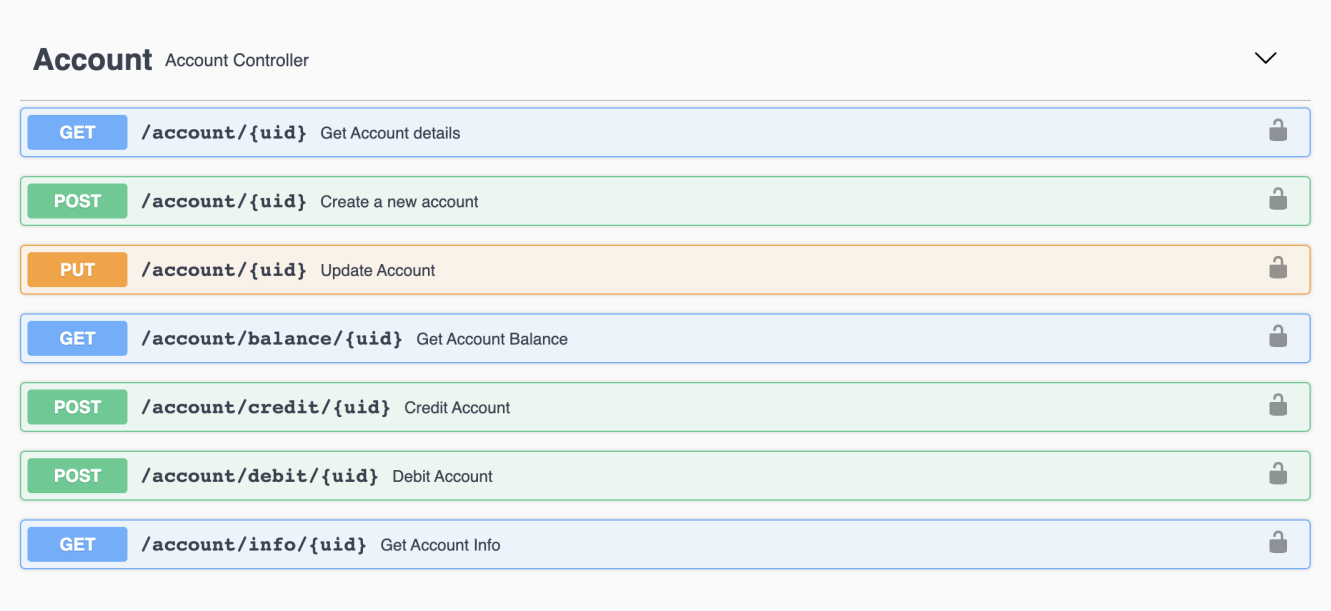
See Swagger UI for API documentation. It's available at localhost:9090/swagger-ui/ when you run your services locally.



6.2. Account Microservice

The Account microservice, allows interacting with Account domain model. Various update and query operations can be performed.

See Swagger UI for API documentation. It's available at localhost:8080/swagger-ui/ when you run your services locally.



6.3. Transaction Microservice

6.4. Transaction Processor

6.5. Notification Microservice

Chapter 7. Reloadly Modules

7.1. Email

7.2. SMS

7.3. Security Integration

7.4. Notification Integration

7.5. Swagger UI Integration

Chapter 8. Production Deployment

8.1. Kubernetes Deployment (EKS)

8.1.1. Docker Images

8.1.2. Helm Charts

8.2. AWS Beanstalk

Chapter 9. Production Monitoring

9.1. Spring Boot Admin

9.2. Grafana

9.3. Prometheus

9.4. Loki

Chapter 10. Product Development Approach

10.1. Processes and Methodology

10.2. Team Culture

10.3. Tools

10.3.1. Team Communication

10.3.2. Defect Management

10.3.3. Release Management

10.3.4. Source Code Management

10.3.5. CI/CD and Automation