Fintech Platform Reference Implementation

Arun Patra

Version 1.1.0, 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.

> The **reloadly** brand name, and logo are copyright of Reloadly. Permission is hereby granted to Reloadly to review the architecture, design, source code and documentation of this project.
>
> Unless required by applicable law or agreed to in writing, this software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

1.1. Source Code

 Head over to the GitHub repository here.

1.2. License

This Software is released under the MIT License

Chapter 2. Code Challenge Summary

This reference implementation embraces modern architectural styles, sound architectural principles and robust engineering practices. The following features have been implemented.

Scalable Microservices
- ✅ Account Microservice
- ✅ Transaction Microservice
- ✅ Notification Microservice

Enterprise Security
- ✅ Authentication Service
- ✅ Supports JWT
- ✅ Supports API Keys

Enterprise Grade Transaction Processor
- ✅ Powered by Kafka
- ✅ Ultra High Scalability
- ✅ Inbuilt Resiliency

Robust Integrations
- ✅ Amazon SES Email
- ✅ SMS Powered by Twilio

Production Ready
- ✅ Kubernetes Deployment (EKS)
- ✅ Monitoring - SBA
- ✅ Prometheus and Grafana
- ✅ Loki Log Aggregation

Robust Architecture
- ✅ Event Driven
- ✅ RESTful Architecture
- ✅ Resilient and Scalable By Design
- ✅ Modern Design Patterns

Modern Tech Stack
- ✅ Java 11
- ✅ Spring Boot
- ✅ Kafka
- ✅ Cloud Native Architecture

Engineering Best Practices
- ✅ Extensive Unit & Integration Tests
- ✅ Fully Documented Code
- ✅ GitHub VCS
- ✅ Full CI/CD

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

Chapter 3. Code Challenge Highlights

- ~ **1500 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 environment. 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 recommended).*

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.

> **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 on your local machine.

```
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 setup 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 the steps below before running your 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 set up Kafka and MySQL 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;

--
```

```sql
-- 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.

```sql
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.

```sql
--
-- 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.

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

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.

Environment Variable	Value	Notes
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` or `~./bashprofile` file or a suitable method of your liking. After adding, you should `source` these files for changes to take effect, e.g. `source ~/.zprofile`.

You may add the following in the script. Supply appropriate values.

```
#
# These properties are used by the platform microservices
#
export DB_USER=
export DB_PASSWORD=
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=

# Change this to false, if you want to send actual emails.
export EMAIL_DRY_RUN=true
export SENDER_EMAIL_ID=

# Change this to false, if you want to send actual SMS. SMS charges may apply.
export TWILIO_DRY_RUN=true
export TWILIO_ACCOUNT_SID=
export TWILIO_AUTH_TOKEN=
export TWILIO_MSG_SVC_ID=

export RELOADLY_SVC_ACCT_API_KEY=
```

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.

> ℹ 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.
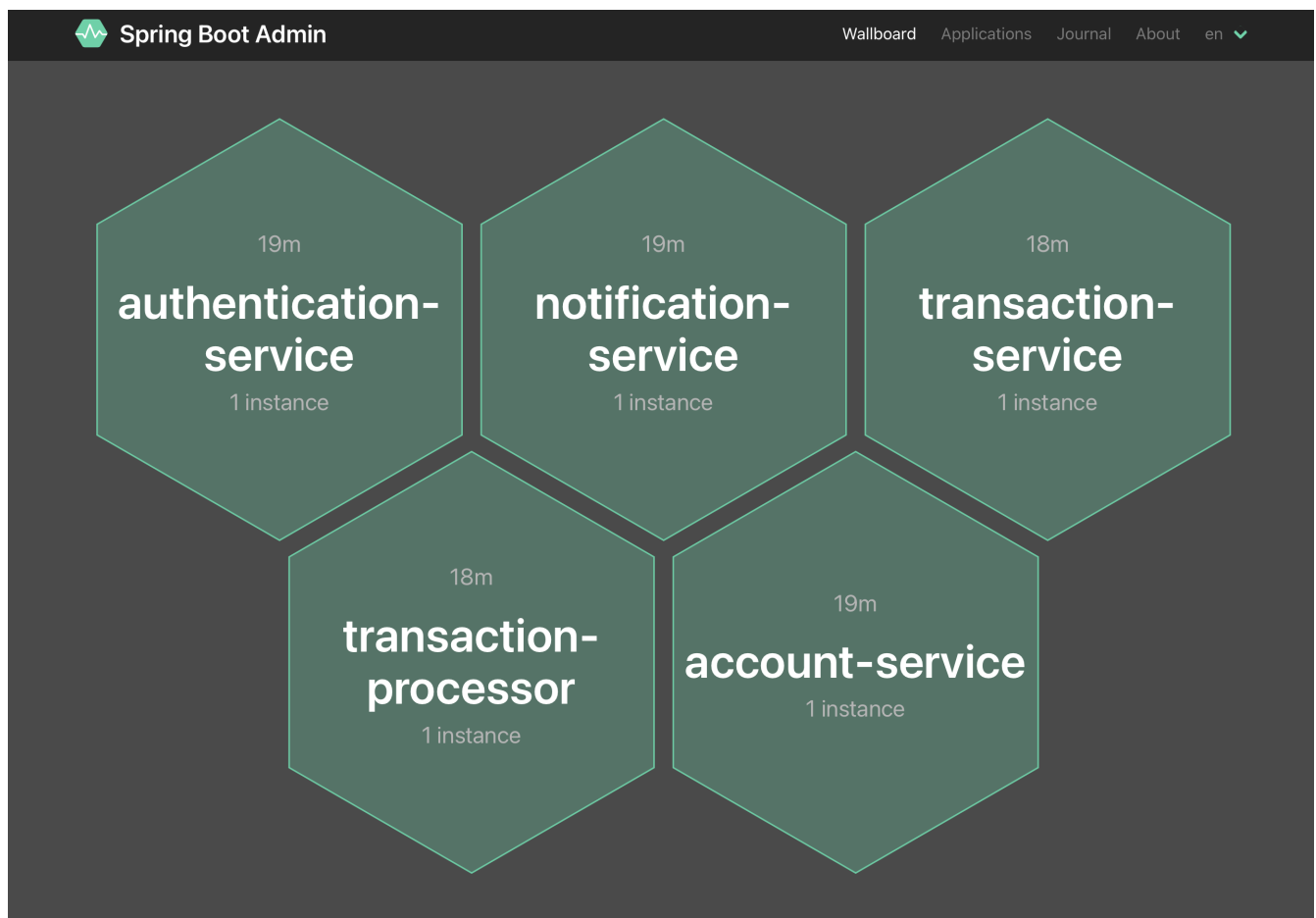
Congratulations! Your services are running.

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

4.3.3. Cleanup

1. Kill all microservices by hitting `CTRL + C` on the terminal windows where you started the services.

2. Shutdown your locally running MySQL instance.

```
mysql.server stop
```

3. Now kill the Kafka process that you had started earlier by hitting `CTRL + C` in the terminal window where Kafka is running.

Chapter 5. Architecture

The Architecture envisioned has been depicted in the illustration below.
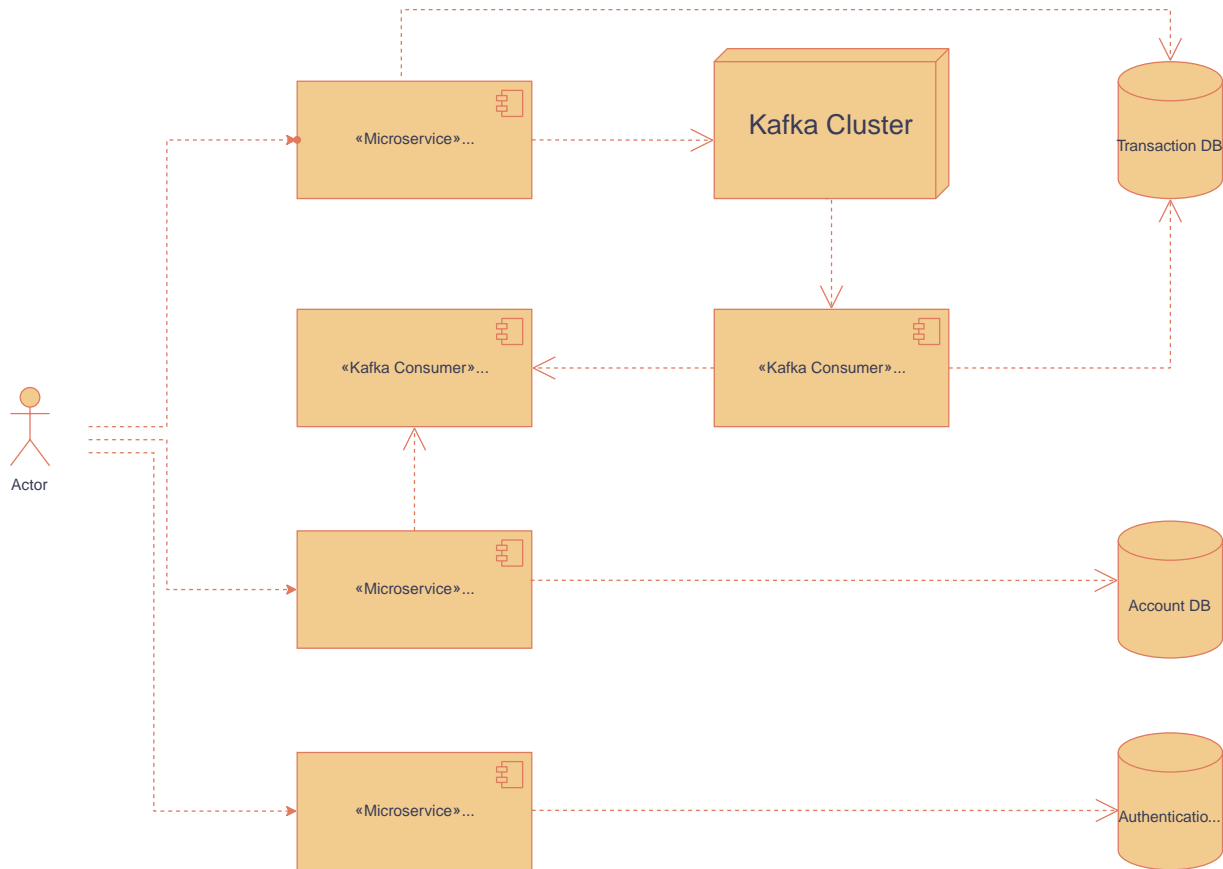
Viewer does not support full SVG 1.1

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 acts as a Kafka Producer.

- The Transaction Processor is a Kafka Consumer and potentially runs a number of instances. All instances of the transaction processor form a Kafka Consumer Group.

- The transaction processor updates transaction status once a transaction is processed. Transactions may be retried.

- Notification microservice is called internally by other microservices and the 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 validates tokens and API keys with the Auth service.

5.2. Design

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

Additionally, for monitoring, A Spring Boot Admin app is utilized (not shown in illustration above); its optional.
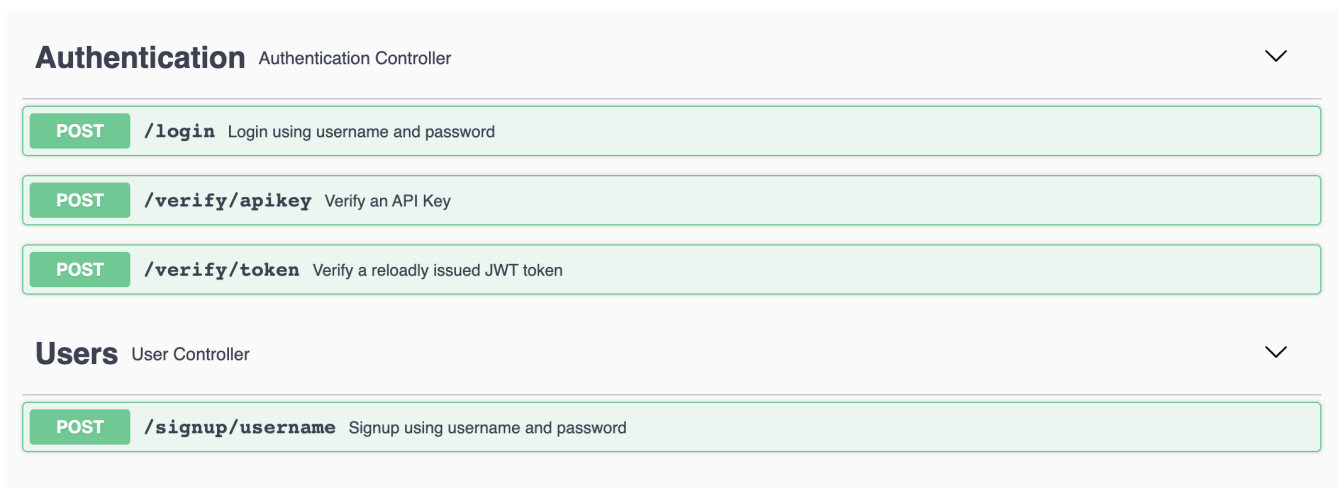
Chapter 6. Reloadly Services

6.1. Authentication Microservice

The Authentication service allows multiple enterprise facilities like:

- User signup : Currently username and password based login is supported. This can be extended to other auth channels. Users are issued JWT tokens.
- Token Verification : JWT tokens are verified at the authentication service.
- API Key verification and issue : 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.
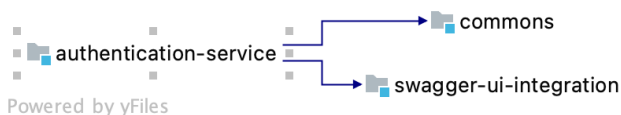
6.1.1. API

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

6.1.2. Module Dependencies


6.2. Account Microservice

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

6.2.1. API

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

GET	`/account/{uid}` Get Account details	🔓
POST	`/account/{uid}` Create a new account	🔓
PUT	`/account/{uid}` Update Account	🔓
GET	`/account/balance/{uid}` Get Account Balance	🔓
POST	`/account/credit/{uid}` Credit Account	🔓
POST	`/account/debit/{uid}` Debit Account	🔓
GET	`/account/info/{uid}` Get Account Info	🔓

6.2.2. Module Dependencies

Powered by yFiles
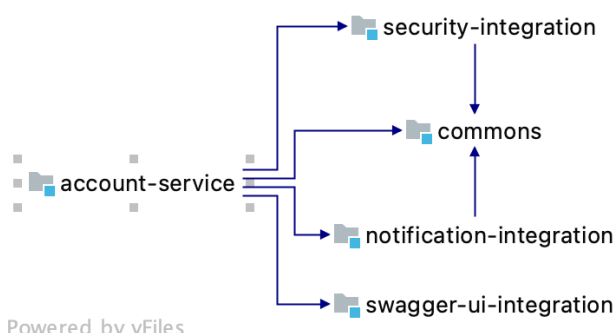
6.3. Transaction Microservice

The Transaction service allows users to initiate user transactions, like money reload and airtime send operations. Note that, the transaction service, accepts user requests and posts to a reliable and resilient Kafka cluster. It also persists the user request in persistent storage so that its status can be queried. User requests return immediately. The transactions posted to Kafka are picked up later asynchronously by the **Transaction Processor** component.

6.3.1. API

See Swagger UI for API documentation. It's available at localhost:8081/swagger-ui/ when you run your services locally.

6.3.2. Module Dependencies

Powered by yFiles
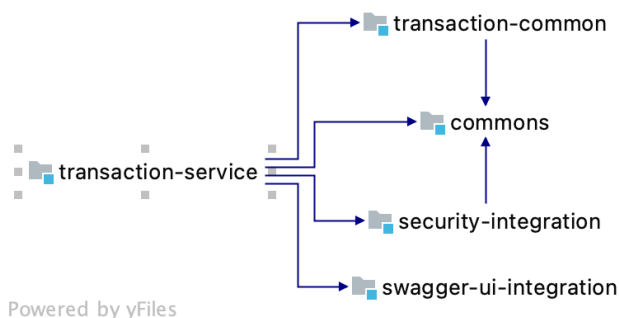
6.4. Transaction Processor

The Transaction Processor is at the heart of the platform and has the responsibility of processing inbound user transactions. The transactions could be related to any business functionality, like sending airtime or reloading money into your own account. Business operations that are ideally suited to be routed to the transaction processor are those that are potentially expensive and have strict consistency guarantees to fulfill. Also, such operations may be retried e.g. an airtime send operation may be retried.

The transaction processor listens to transaction requests that are posted to a Kafka topic. Part of the objective here is to scale the transaction processing horizontally. Kafka listeners(like the transaction processor) can form Consumer Groups(see Kafka documentation for more details). Essentially, when the transaction processor is deployed for example to a Kubernetes cluster, one could scale very easily by increasing/decreasing the number of replicas. All other production grade features of Kafka are available by default to the transaction processor.

Once the Transaction Processor processes a transaction, it ensures that notifications are sent out to the user who initiated the transaction.

6.4.1. Module Dependencies
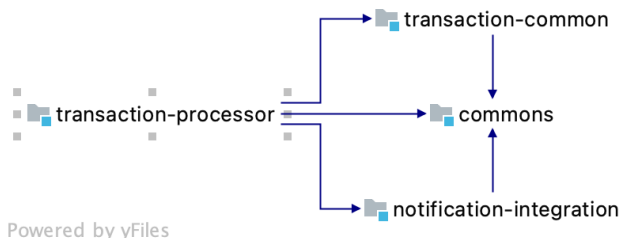
6.5. Notification Microservice

The Notification microservice allows sending emails and SMS messages. Endpoints are provided for callers to invoke these facilities as REST API calls.

For Email, Amazon SES is used. For SMS, Twilio is used. Other email or SMS providers could be plugged in transparently since the required abstractions exist to easily swap out implementations.

6.5.1. API

See Swagger UI for API documentation. It's available at localhost:8082/swagger-ui/ when you run your services locally.
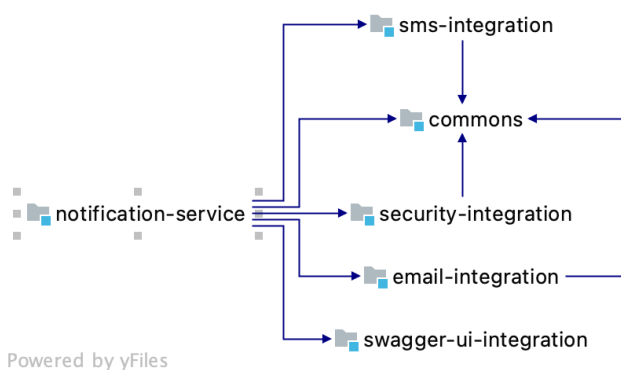

6.5.2. Module Dependencies

Chapter 7. Reloadly Modules

Foundational modules are reusable pieces of software that can be easily integrated into other modules. The following modules are used in the rest of the platform. Note that, in a Microservices architecture, individual microservices should share reusable code only if it makes sense and if sharable code is well managed by a team which keeps pace with others.

7.1. Email

This module supports email sending facilities. This is a drop-in solution. Add the following dependency in any Spring Boot application and Spring Boot Auto-configuration would wire the necessary beans.

```xml
<dependency>
    <groupId>com.reloadly</groupId>
    <artifactId>email-integration</artifactId>
    <version>1.1.0</version>
</dependency>
```

This component is configurable. See the configuration properties of this module.

7.2. SMS

This module supports SMS sending facilities. This is a drop-in solution. Add the following dependency in any Spring Boot application and Spring Boot Auto-configuration would wire the necessary beans.

```xml
<dependency>
    <groupId>com.reloadly</groupId>
    <artifactId>sms-integration</artifactId>
    <version>1.1.0</version>
</dependency>
```

This component is configurable. See the configuration properties of this module.

7.3. Security Integration

This module automatically secures all Web API endpoints with Auth Service security via Spring Security. This is a drop-in solution. Add the following dependency in any Spring Boot application and Spring Boot Auto-configuration would wire the necessary beans.

```
<dependency>
    <groupId>com.reloadly</groupId>
    <artifactId>security-integration</artifactId>
    <version>1.1.0</version>
</dependency>
```

This component is configurable. See the configuration properties of this module.

7.4. Notification Integration

This module integrates Notification Service via a Java API. The Java API wraps the Notification Service REST API calls. This is a drop-in solution. Add the following dependency in any Spring Boot application and Spring Boot Auto-configuration would wire the necessary beans.

```
<dependency>
    <groupId>com.reloadly</groupId>
    <artifactId>notification-integration</artifactId>
    <version>1.1.0</version>
</dependency>
```

This component is configurable. See the configuration properties of this module.

7.5. Swagger UI Integration

This module adds Swagger UI integration to any Spring Boot app. This is a drop-in solution. Add the following dependency in any Spring Boot application and Spring Boot Auto-configuration would wire the necessary beans.

```
<dependency>
    <groupId>com.reloadly</groupId>
    <artifactId>swagger-ui-integration</artifactId>
    <version>1.1.0</version>
</dependency>
```

This component is configurable. See the configuration properties of this module.

Chapter 8. Production Deployment

The individual microservices may be run as standalone spring boot applications in lower environments. However, for production like environments, where HA, failover and scalability needs exist, a suitable approach may be taken. We discuss some aspects and approaches below.

8.1. Kubernetes Deployment (EKS)

See this GitHub issue.

See the Kubernetes Deployment chapter for more details.

Kubernetes would be the ideal way to run and scale the microservices of this platform. Of special interest is the transaction processor which necessarily needs to scale based on transaction volumes. Some aspects to note are:

- It's best to keep databases outside the scope of Kubernetes. Consider having a managed database service from a cloud provider like AWS or GCP.

- Microservices including the transaction processor should be Dockerized and docker images should be created in the build pipeline. Store images in an organizational Docker Registry or a hosted one.

- Consider a managed Kubernetes Service like Amazon EKS of Google GKE.

8.1.1. Docker Images

To create Docker images, follow these steps. First install Docker Desktop for macOS.

```
cd account-service
docker build --tag reloadly/account-service .
```

To spin up a container, issue the following command. First start your MySQL instance on your local machine and ensure that the tables are created.

```
docker run -it -p 8080:8080 --rm --env DB_USER=root --env DB_PASSWORD=mysqlpass123
--env DB_URL=jdbc:mysql://host.docker.internal:3306/rlacctdb reloadly/account-
service:latest env
```

To launch in the background:

```
docker run -d -p 8080:8080 --rm --env DB_USER=root --env DB_PASSWORD=mysqlpass123
--env DB_URL=jdbc:mysql://host.docker.internal:3306/rlacctdb reloadly/account-
service:latest env
```

8.1.2. Helm Charts

As a best practice, consider using Helm Charts to easily manage deployments and deployment environments. It's beneficial to use Helm Charts since it allows for easy rollback.

8.1.3. Service Mesh

Consider using Istio or Linkerd as a service mesh. This would allow you to handle load balancing, retries and TLS offloading.

8.1.4. Distributed Tracing

Consider using Jaeger for distributed tracing. This should serve as a very useful tool in a microservices context.

8.2. AWS Beanstalk or EC2 Containers

While deployment to AWS Beanstalk or EC2 containers is possible, it should probably make sense only for small deployments. Managing these environments is fraught with errors and do not support automation very well. For modern microservice deployment strategies, consider using Helm Charts and Kubernetes.

Chapter 9. Production Monitoring

Production monitoring is a necessary part of operations, and the right tools need to be utilized to get the job done.

9.1. Spring Boot Admin

This is a great tool to have full visibility into the runtime of Spring Boot Apps over HTTP(S). This is currently supported. One can easily integrate logfile viewers. SBA supports discovery in Kubernetes environments as well.

9.2. Grafana

See this GitHub issue.

Grafana provides powerful dashboards that can display any application metrics and other performance parameters sent to it from applications. It has excellent integration with Spring and plays well in Kubernetes deployments. See grafana.com/ for more details.

9.3. Prometheus

Prometheus is a leading tool for gathering metrics from applications and route them to Grafana for powerful visualizations. Excellent support exists in the Spring ecosystem, and it plays well in Kubernetes deployments. See prometheus.io/ for more details.

9.4. Loki

Loki is a popular choice for log aggregation and integrates well with Grafana. For more details see grafana.com/oss/loki/.

Chapter 10. Kubernetes Deployment

This chapter is dedicated to containerized deployments. We will need a number of components. We will start with simpler aspects and will finally achieve full automation.

This chapter is targeted to developers. Some familiarity with Containerization technology, Docker and Kubernetes is helpful, but not mandatory.

10.1. Docker Desktop

You can always run the microservices developed in this project as standalone Spring Boot applications. Alternatively, you can create Docker images for each microservice, like the Account microservice, Transaction Processor etc. Once you create a docker image, you can launch a container that will run your docker image. First, you need to install Docker Desktop on your local machine.

10.1.1. Install

On macOS systems, install Docker Desktop by following the instructions here.

10.1.2. Building Docker Images and Running

After you install Docker Desktop, start it. The following example shows how to build a Docker image for a microservice and run the service.

```
cd account-service
docker build --tag reloadly/account-service .
```

To spin up a container, issue the following command. First start your MySQL instance on your local machine and ensure that the tables are created.

```
docker run -it -p 8080:8080 --rm --env DB_USER=root --env DB_PASSWORD=mysqlpass123
--env DB_URL=jdbc:mysql://host.docker.internal:3306/rlacctdb reloadly/account-
service:latest env
```
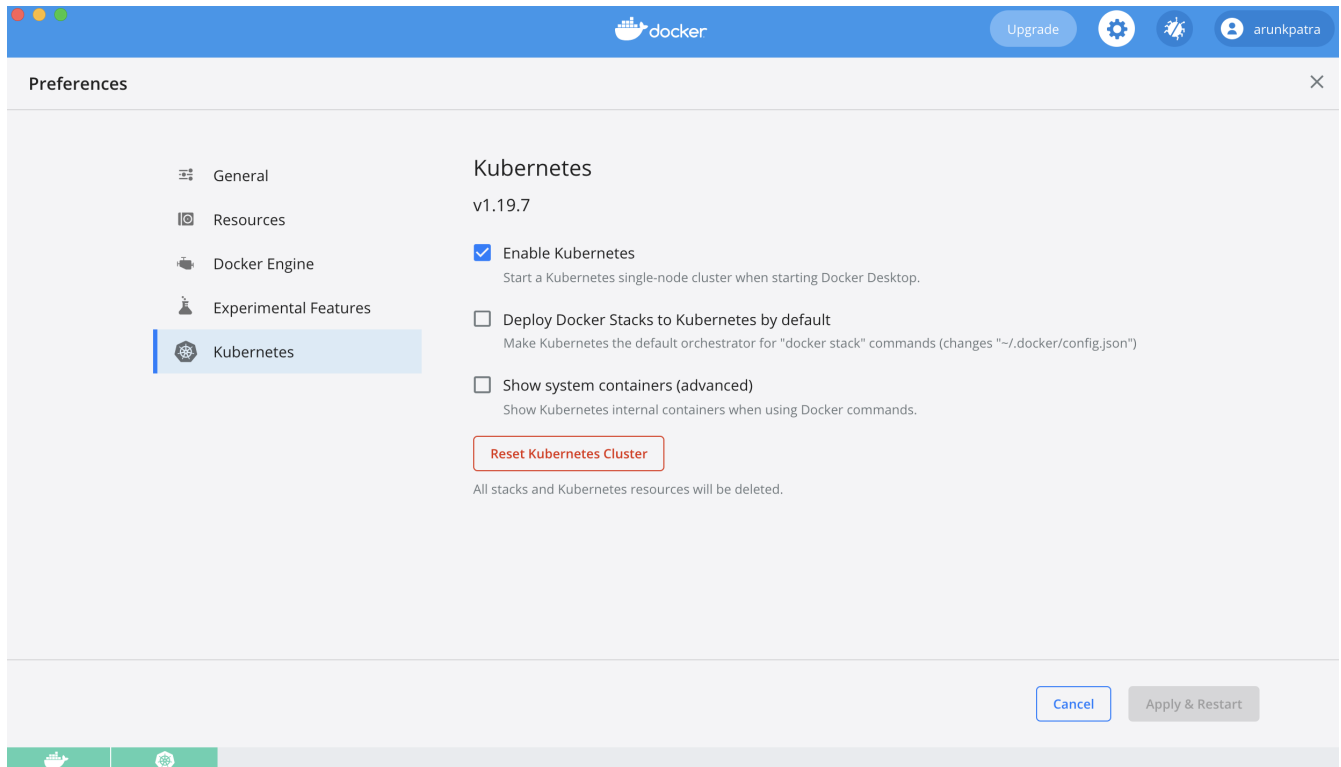
To launch in the background:

```
docker run -d -p 8080:8080 --rm --env DB_USER=root --env DB_PASSWORD=mysqlpass123
--env DB_URL=jdbc:mysql://host.docker.internal:3306/rlacctdb reloadly/account-
service:latest env
```

10.2. Kubernetes

10.2.1. Install

When you install Docker Desktop, it also brings in Kubernetes, but is not enabled yet. Enable Kubernetes from the Docker Desktop Settings.

10.3. Kubernetes Dashboard

The first thing you might want to do is to install Kubernetes Dashboard.

10.3.1. Install

To deploy Dashboard, execute following command:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.2.0/aio/deploy/recommended.y
aml
```

10.3.2. Access

To access Dashboard from your local workstation you must create a secure channel to your Kubernetes cluster. Run the following command:

```
kubectl proxy
```

Now access Dashboard at:

localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/

proxy/

10.3.3. Login

To login to the dashboard, you would need a token. See instructions here

Start Proxy

```
kubectl proxy
```

Get Token

```
kubectl -n kubernetes-dashboard get secret $(kubectl -n kubernetes-dashboard get
sa/admin-user -o jsonpath="{.secrets[0].name}") -o go-template="{{.data.token |
base64decode}}"
```


Here's a helpful tutorial.

10.4. Linkerd

10.4.1. Install

Install Linkerd using the instruction here.

10.4.2. Run

Start Linkerd Dashboard.

```
linkerd viz dashboard &
```

Access Linkerd dashboard at localhost:50750/

10.5. Helm Charts 3

You can use Helm Charts to install software on your K8s cluster.

10.5.1. MySQL Install

See the chart documentation here.

- Install MySQL in your K8s Cluster, by issuing the following command:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install mysql --set
auth.rootPassword=mysqlpass123,primary.service.type=NodePort,primary.service.nodePo
rt=30306 bitnami/mysql --namespace mysql --create-namespace --dry-run
```

> 🛈 The usage of the `--dry-run` parameter, just lets you know what actions will be performed by the Chart. You need to execute the chart again without the `--dry-run` parameter to perform the actual installation.

- Now go ahead and execute for real.

```
helm install mysql --set
auth.rootPassword=mysqlpass123,primary.service.type=NodePort,primary.service.nodePo
rt=30306 bitnami/mysql --namespace mysql --create-namespace
```

- To know the status of the deployment, issue the following command:

```
helm status mysql --namespace mysql
```

- To see the pods created by the chart deployment:

```
kubectl get pods  --namespace mysql
```

- Startup a container for your mySQL client that will be used to connect to the MySQL server that you just provisioned on yuur K8s cluster. It will be deleted when you log out.

```
kubectl run mysql-client --rm --tty -i --restart='Never' --image
docker.io/bitnami/mysql:8.0.24-debian-10-r0 --namespace mysql --command -- bash
```

- Now login into your instance. You will be prompted for the root password. Use the one that you selected while deploying the chart.

```
mysql -h mysql.mysql.svc.cluster.local -uroot -p my_database
```

- Now execute all the database scripts mentioned in the Getting Started section.
- Issue the `SHOW DATABASES` command to list the databases that got created.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| my_database        |
| mysql              |
| performance_schema |
| rlacctdb           |
| rlauthdb           |
| rltxndb            |
| sys                |
+--------------------+
8 rows in set (0.00 sec)
```

- Quit out of the client by issuing a `quit;` and the `exit` out of your mySQL client container.

At this point you have a fully functional MySQL Server containing all the application databases(or Schemas) that you need. Your seed data is up there as well. Here are some salient points about the MySQL Server that got provisioned.

1. Your mySQL server will be accessible in your K8s cluster with the `mysql.mysql.svc.cluster.local`. Port os `3306`.

2. Root user is `root`. Root password is `mysqlpass123`.

3. You could access this database from your local machine at `localhost:30306` using any MySQL client of your choice. This is possible since you chose `NodePort` to be the K8s `ServiceType` while deploying your Helm chart.

10.5.2. MySQL Cleanup

When you no longer need this MySQL server, you can delete the resources just created:

```
helm uninstall mysql -n mysql
```

The above deletes all resources except the persistent volumes.

10.5.3. Kafka Install

See the chart documentation here.

- Install kafka in your K8s Cluster, by issuing the following command:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install kafka bitnami/kafka --namespace kafka --create-namespace --dry-run
```

> **ℹ** The usage of the `--dry-run` parameter, just lets you know what actions will be performed by the Chart. You need to execute the chart again without the `--dry-run` parameter to perform the actual installation.

- Now go ahead and execute for real.

```
helm install kafka bitnami/kafka --namespace kafka --create-namespace
```

- To know the status of the deployment, issue the following command:

```
helm status kafka --namespace kafka
```

- To see the pods created by the chart deployment:

```
kubectl get pods  --namespace kafka
```

- Now start a client container

  1. To create a pod that you can use as a Kafka client run the following commands:

  ```
  # Create client container
  $ kubectl run kafka-client --restart='Never' --image
  docker.io/bitnami/kafka:2.8.0-debian-10-r0 --namespace kafka --command -- sleep
  infinity
  # Connect to client container
  kubectl exec --tty -i kafka-client --namespace kafka -- bash
  ```

  2. Now put a message. Type some text when you see the `>` prompt and then hit the `ENTER` key. After that hit `CTRL + C` button.

  ```
  kafka-console-producer.sh --broker-list kafka.kafka.svc.cluster.local:9092
  --topic test
  ```

3. No read messages.

```
kafka-console-consumer.sh --bootstrap-server kafka.kafka.svc.cluster.local:9092
--topic test --from-beginning
```

At this point you have a fully functional Kafka installation. Here are some salient points about the Kafka installation that got provisioned.

1. Kafka can be accessed by consumers via port 9092 on the following DNS name from within your cluster: kafka.kafka.svc.cluster.local. Port is 9092.

2. PLAINTEXT protocol is enabled, so that messages can be sent without security. This is strongly discouraged in any environment other than local testing.

10.5.4. Kafka Cleanup

When you no longer need this Kafka installation, you can delete the resources just created:

```
helm uninstall kafka -n kafka
```

The above deletes all resources except the persistent volumes.

10.6. Running Reloadly Platform Services on Kubernetes

From root of project root, issue the following commands.

1. This will install the reloadly microservices in your locally running K8s cluster.

```
cd deployment
kubectl apply -f ./kubernetes/reloadly-platform.yaml
# OR source the manifest directly from GitHub
kubectl apply -f https://raw.githubusercontent.com/arunkpatra/reloadly-
services/main/deployment/kubernetes/reloadly-platform.yaml
```

2. Check Pod initialization status.

```
kubectl get pods -n reloadly
```

3. You should see output like:

```
NAME                          READY   STATUS    RESTARTS   AGE
admin-service-6d897557d-v6b27 2/2     Running   0          2m29s
```

4. To uninstall all the services, issue the following commands from root of project directory:

```
cd deployment
kubectl delete -f ./kubernetes/reloadly-platform.yaml
# source the manifest directly from GitHub
kubectl delete - f https://raw.githubusercontent.com/arunkpatra/reloadly-
services/main/deployment/kubernetes/reloadly-platform.yaml
```

10.7. Jaeger

10.7.1. Install

Reference: linkerd.io/2.10/tasks/distributed-tracing/

```
linkerd jaeger install | kubectl apply -f -
linkerd jaeger check
linkerd jaeger dashboard
```

Chapter 11. Product Development Approach

Product development in the Microservices world has its benefits, but it needs a certain approach, culture and the appropriate tools.

The Microservices paradigm, is part technology and part organizational culture. It closely ties with the Agile manifesto. Microservices architecture make adequate sense when the size of the business, the organization and the domain justifies it. Eventually Microservice boundaries would align with team boundaries. That reflects in code organization as well. if these nuances are overlooked, it's very easy to get the entire Microservices approach wrong and rack up huge technical debts in the mid to long term. Microservices done the right way, with the right team organization and tools will go a long way towards increasing developer productivity and keep operational costs low.

11.1. Processes and Methodology

Some salient points:

- Consider adopting Agile Methodology for software development and delivery.
- Consider aligning team boundaries with Microservice domain boundaries. As an example, a dedicated team to work on Account Microservices may be formed. This team should own the software development, release planning and deployment for this microservice. As a best practice, deployment of this microservice should not have software "build time" dependencies with any other microservice. Additionally, the database should be exclusively owned and used by this microservice.
- Team size should not exceed 5-7 team members with members being full stack engineers.
- Consider Scrum of Scrums to align all teams well in advance.
- Scrum meetings (daily standup meetings) may be used. However, this is falling out of fashion and may not work very well in Distributed Agile. Also consider effective usage of tools like GitHub Issues, Gitter, Discord Server, GitHib Discussions, Slack etc. These tool go a long way towards increasing productivity.
- Consider fortnightly software release cycles.
- Agile Scrum teams are best led by a Project Lead. The project lead has the responsibility and authority to manage software, reviews, merging and releasing. Each microservice should have a project lead, and a team attached to it.
- A Product Owner or Product Manager is responsible for managing business requirements and planning. A PO can be assigned to multiple logically grouped teams. A PO has no authority over engineering teams but fully owns requirements and what should be delivered in a particular release.
- Consider a culture of merging often and releasing often.
- Consider automating every single manual process that you can afford to.

11.2. Tools

Following tools are recommended for their wide adoption, excellent usability and reliability.

11.2.1. Team Communication

Consider Slack. Consider creating multiple channels as needed and add the right apps, e.g. GitHub, Confluence, Miro etc to Slack.

11.2.2. Defect Management

Consider using Jira for defect management, Integrate with Slack.

11.2.3. Release Management

Consider release planning and management in GitHub.

11.2.4. Source Code Management

GitHub is the leading platform for DVCS and is recommended. It provides a full set of umbrella services suited for modern teams following industry best practices.

11.2.5. CI/CD and Automation

Consider GitHub Actions and/or other suitable tools to facilitate deployment to Kubernetes.