MVP - Tech challenge

Lucas Fernando das Neves Carvalho dos Santos

- Email: lfneves.dg@gmail.com
- RM: 350505
- Grupo 53

Repositorys:

https://github.com/lfneves/mvp https://github.com/lfneves/infra-rds-terraform
https://github.com/lfneves/infra-eks-terraform https://github.com/lfneves/infra-vpc-terraform

---

Infra Video explication:

Part 1

https://youtu.be/FCwZ1W9Dc0s

Part 2 https://youtu.be/ZpGls-grP6I

AWS Infra Terraform EKS

Terraform AWS EKS Cluster Deployment

This project uses Terraform to automate the deployment of an Amazon Elastic Kubernetes Service (EKS) cluster on AWS. Amazon EKS is a managed Kubernetes service that simplifies the deployment, scaling, and operation of containerized applications using Kubernetes.

Prerequisites

Before getting started, make sure you have the following prerequisites installed on your machine:

- Terraform (you can use `terraform --version` to check)
- AWS CLI configured with appropriate credentials
- Kubectl for interacting with the cluster
- kubectl-aws-iam-authenticator for authenticating with the EKS cluster
- Internet access

Configuration

1. Clone this repository:

```
git clone https://github.com/lfneves/infra-eks-terraform.git
cd infra-eks-terraform
```

2. Automatically create a `delivery-eks-terraform.tfstate` file and deploy bucket on `delivery-terraform-s3`` and provide the necessary variables:

```
region            = "us-east-1"
cluster_name      = "delivery-cluster"
node_instance_type = "t2.small"
node_max_count     = 2
node_min_count     = 1
```

Replace the values above with the specific settings for your environment.

AWS Infra Terraform RDS

https://github.com/lfneves/infra-rds-terraform

Terraform AWS RDS PostgreSQL Deployment This project uses Terraform to automate the deployment of a single Amazon RDS instance with PostgreSQL on AWS. Amazon RDS (Relational Database Service) is a managed relational database service that makes it easy to deploy, operate, and scale databases.

Application mvp

https://github.com/lfneves/mvp

This is a Spring Boot WebFlux application using Kotlin.

Spring WebFlux utilizes the Reactor library, which is an implementation of Reactive Streams specs for building non-blocking applications.

This project:

- Uses Reactor Netty as the default implementation for testing purposes. To change to Apache Tomcat as the default Web container for Spring WebFlux, follow these steps.
- Utilizes functional endpoints.
- Employs the PostgreSQL database.

💡 Requirements

- Java 17 or later - SDKMAN - Recommendation
- Gradle 7.6.1 or later - Gradle build tool Installation
- Docker 24.0.2 or later - How to install Docker
- Docker Compose 1.29.2 or later - Reference guide
- Minikube v1.31.2 or later - Get Started with Minikube
- Helm v3.10.1 or later - Installing Helm
- The project runs on port 8099 (http://localhost:8099).

Getting Started

```
# Get the latest version

git clone https://github.com/lfneves/mvp.git
```

Project Structure

```
main
├── kotlin
|   └── com
|       └── mvp
|           └── delivery
|               ├── DeliveryApplication.kt
|               ├── application
|               ├── domain
|               ├── infrastruture
|               └── utils
└── resources
    ├── application.yml
    └── database
        ├── 1_create_tables.sql
        └── 2_inserts_category.sql
```

Prerequisites

Check versions:

- Java 17+

  ```
  java --version
  ```

- Docker

  ```
  docker -v
  ```

- Docker Compose

  ```
  docker-compose --version
  ```

Installation

This is an example of how to use the software and how to install it.

Docker

In the main project directory:

Docker build and start applications:

```
$ docker-compose up --build
```

Or use:

```
$ docker-compose up -d --build
```

To recreate the application in case of problems, use the command:

```
$ docker-compose down
```

Kubernetes (k8s)

To initiate Kubernetes applications, execute the commands found within the "k8s" folder.

```
$ kubectl apply -f delivery/k8s/postgres/.
```

```
$ kubectl apply -f delivery/k8s/application/.
```

o access the application URL, use the following command:

```
$ minikube service delivery --url
```

Example output:

```
http://192.168.49.2:32000
```

Inside the "k8s" folder, you will discover ".yaml" files utilized to deploy databases and applications within Kubernetes.

```
/delivery/k8s
├── application
|   ├── 1-deployment.yaml
|   ├── 2-service-load-balancer.yaml
|   ├── 3-hpa.yaml
|   ├── 4-ingress.yaml
|
└── postgres
    ├── 1-db-persistent-volume.yaml
    ├── 2-db-volume-claim.yaml
    ├── 3-db-configmap.yaml
    ├── 4-db-secret.yaml
    ├── 5-db-deployment.yaml
    └── 6-db-service.yaml
```

Metric Server

```
$ minikube addons enable metrics-server
```

To monitor the Horizontal Pod Autoscaler, employ the following command:

```
$ kubectl get hpa
```

```
NAME                                         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/delivery   Deployment/delivery   55%/80%   2         4         2          51m
```

Kubernetes (k8s) - Install with Helm

[BETA] Because this hasn't been implemented following best practices.

```
$ helm install deliveryhelm deliveryhelm/
```

Helm uninstall

```
$ helm uninstall deliveryhelm deliveryhelm/
```

Integration Mercado Pago

For the webhook checkout process, generate a QR code.

For testing full process with Mercado Pago webhook, use hookdeck.com with CLI to change the order status in the localhost application.

Apllication path **/api/v1/mp-order/qr-code-checkout** creates a checkout with Mercado Pago.

Example:

```
{
    "in_store_order_id": "75ca8fe9-3b1a-4053-8f3e-49a62e91f8e8",
    "qr_data": "00020101021243650016COM.MERCADOLIBRE02013063675ca8fe9-3b1a-
4053-8f3e-49a62e91f8e8520400005303986580 2BR5908delivery6009SAO
PAULO62070503***63042BFA"
}
```

This project uses CommandLineRunner

- CommandLineRunner is used to create a default user, products and categories on start application startup.
- Default login :

/api/auth/login-token

```
{
  "username": "99999999999",
  "password": "123"
}
```

The best way to use it as a suggestion is by using Postman

A collection is available preconfigured in the project root

MVP - Pos tech delivery application.postman_collection.json

- This project uses user and session control for access
- Endpoints without control access ***"/api/auth/*"***, ***"/api/v1/users/signup"***

Create new user example:

http://localhost:8099/api/v1/users/signup

Body:

```
{
    "name": "Admin",
    "email": "admin@email.com",
```

```
    "cpf": "99999999999",
    "password": "admin",
    "address": {
        "street": "rua 1",
        "city": "sp",
        "state": "sp",
        "postalCode": "1234"
    }
}
```

Login - Use the username (cpf) and password, then copy the token and use it in authenticated endpoints.

http://localhost:8099/api/auth/login-token

```
{
  "username": "99999999999",
  "password": "admin"
}
```

Response:

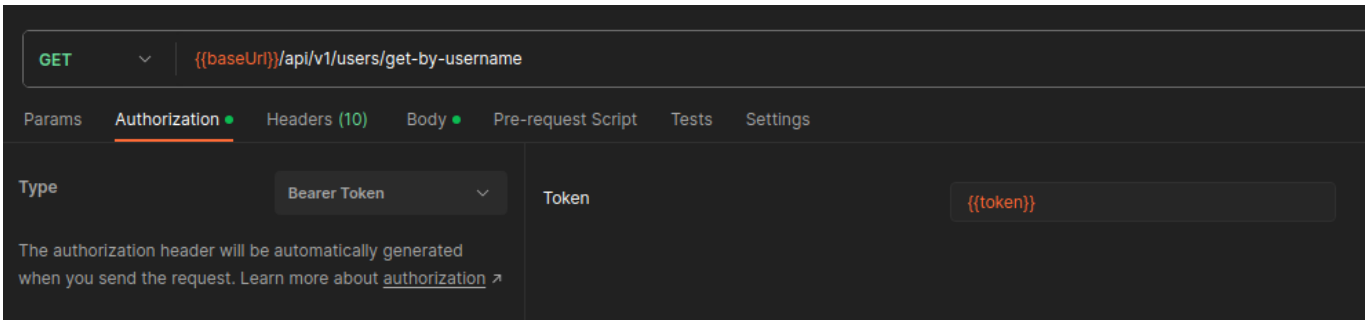
```
{
"token":
"eyJhbGciOiJIUzI1NiJ9.eyJpZENsaWVudCI6IjAiLCJ1c2VybmFtZSI6IjEyMzQ1Njc4OTEyIiwic3ViIjoiMTIzNDU2Nzg5MTIiLCJpYXQiOjE2ODgwOTI1NTAsImF1ZCI6Im5vLWFwcGxpY2F0aW9uLW5hbWUiLCJleHAiOjE2ODgwOTQwMDB9.HagYPqukwOML3OYad8sRjlnE0Gsy-5tGUSC72S-xyfU"
}
```

💡To make it easier use environment variables

Place the command in the test tab on /api/auth/login-token

```
pm.environment.set("token", pm.response.json().token);
```

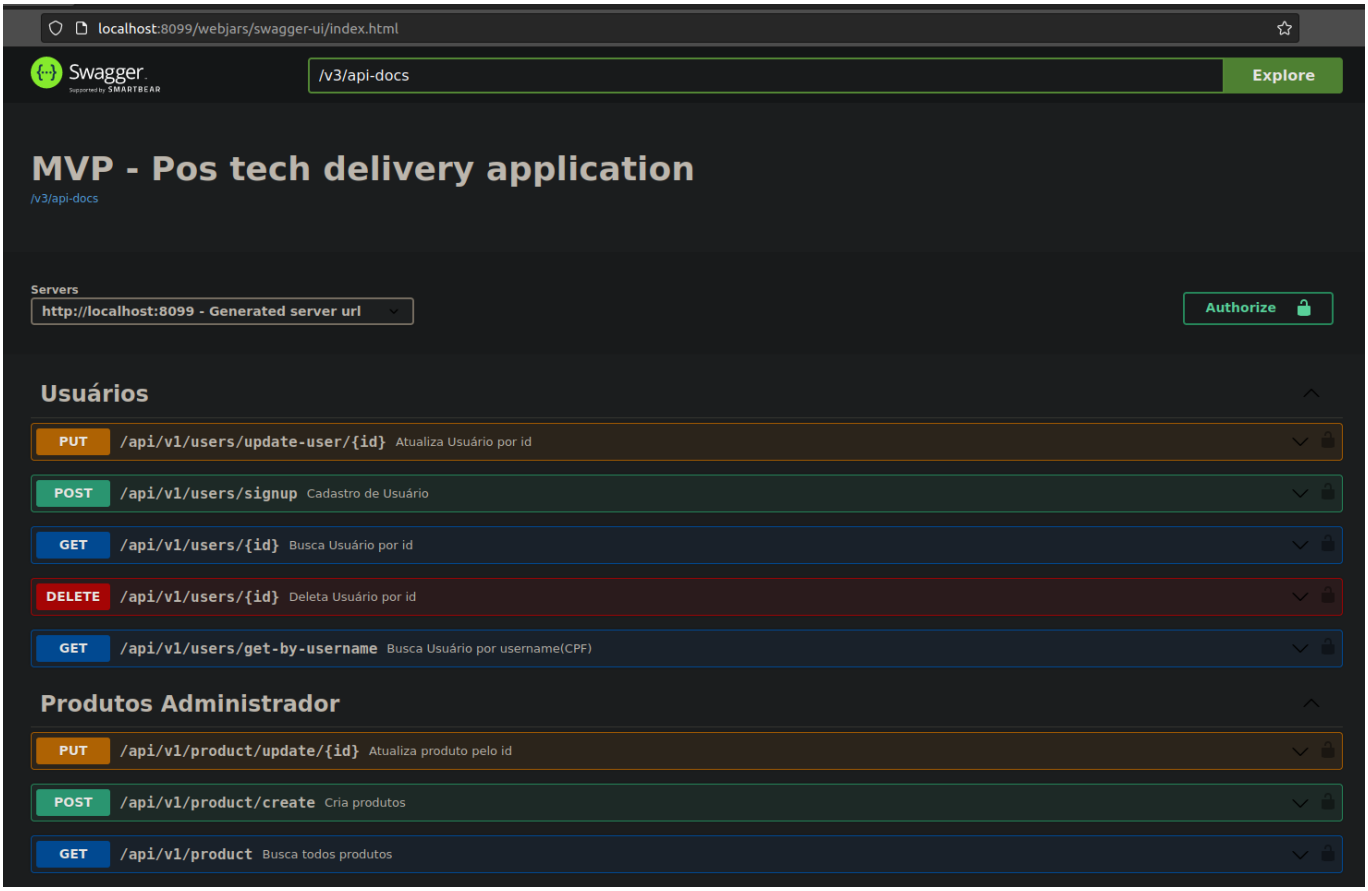
Example:



This project also uses OpenAPI Specification (Swagger).

To access swagger use the URL:

http://localhost:8099/swagger-ui.html or http://localhost:8099/webjars/swagger-ui/index.html



Roadmap

- [x] Improve README.md
- [x] Update order add paid status and adjusting service
- [x] Implementation Helm

- Improvements

- [x] Refactor admin services and repository to new package
- [x] Fix create order exceptions
- [x] Mercado Pago Qr code checkout
- [x] Refactor scripts database

License