1. Tech challenge

- RM: 350505
- Grupo 53

1.1. Repository:

1.1.1. Documentation

- https://github.com/lfneves/tech-challenge-doc

1.1.2. Projects

- https://github.com/lfneves/tech-challenge-order
- https://github.com/lfneves/tech-challenge-payment
- https://github.com/lfneves/tech-challenge-status

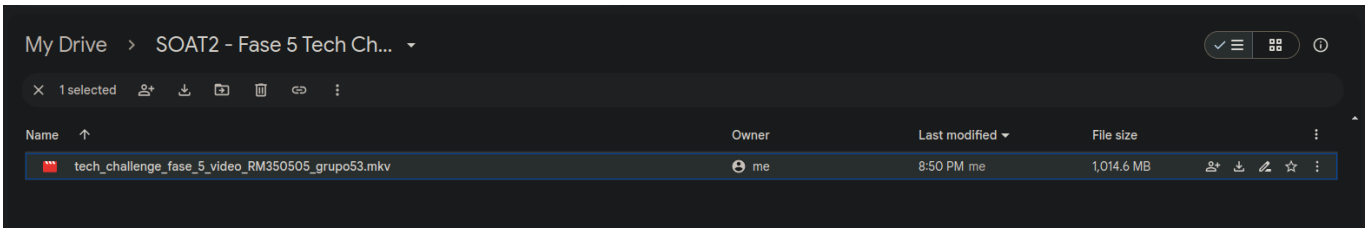
1.2. Video

1.2.1. Topics

- Explanation of the chosen SAGA pattern and its justification;
- Architecture design;
- Cloud structure architecture and how the SAGA communication is set up.
- Links with the OWASP ZAP processing reports (before and after correction);

1.2.2. Link:

https://drive.google.com/file/d/1H8AvE49yABdQwNxchkkKhKOG8pV-Rlkl/view?usp=sharing



1.3. Summary

- 1. Tech challenge
  - 1.1. Repository:
    - 1.1.1. Documentation
    - 1.1.2. Projects
  - 1.2. Video
    - 1.2.1. Topics
    - 1.2.2. Link:

    - 1.3. Summary
- 2. Project Architecture
    - 2.1. Projects Saga Pattern Architecture
        - 2.1.1. Justification for Choosing the SAGA Pattern
            - 2.1.1.1. Distributed Transactions and Data Consistency
            - 2.1.1.2. Decoupling and Flexibility
            - 2.1.1.3. Resilience and Failure Recovery
            - 2.1.1.4. Visibility and Monitoring
            - 2.1.1.5. Scalability
    - 2.2. Projects Cloud Architecture
    - 2.3. OWASP ZAP Report
        - 2.3.1. OWASP ZAP processing reports before correction:
        - 2.3.2. OWASP ZAP processing reports after correction:
    - 2.4. LGPD Endpoint
        - 2.4.1. Route/API where the client can request the deletion of their personal data from the database used.
        - 2.4.2. API have the following fields:
    - 2.5. Projects Automated Deployment/Test
        - 2.5.1. Github Actions Workflow
    - 2.6. Projects Coverage
    - 2.7. Jacoco report
    - 2.8. order-ws
    - 2.9. payment-ws
    - 2.10. status-ws
- 3. Projects structures:
    - 3.1. Terraform AWS EKS Cluster Deployment
        - 3.1.1. AWS Infra Terraform EKS
        - 3.1.2. Prerequisites
        - 3.1.3. Repository
    - 3.2. Terraform AWS RDS Database Deployment
        - 3.2.1. AWS Infra Terraform RDS
    - 3.3. Justification for Choosing PostgreSQL for a Restaurant System
    - 3.4. Database Diagram
        - 3.4.1. Application mvp
            - 3.4.1.1. This is a Spring Boot WebFlux application using Kotlin.
    - 3.5. 💡 Requirements
    - 3.6. Getting Started
    - 3.7. Project Structure
        - 3.7.1. Prerequisites
    - 3.8. Installation
        - 3.8.1. Docker
        - 3.8.2. Kubernetes (k8s)
            - 3.8.2.1. To initiate Kubernetes applications, execute the commands found within the "k8s" folder.
            - 3.8.2.2. o access the application URL, use the following command:
            - 3.8.2.3. Example output:

        - 3.8.2.4. Metric Server
      - 3.8.3. Kubernetes (k8s) - Install with Helm
        - 3.8.3.1. Helm uninstall
  - 3.9. Deploy Github-actions
    - 3.9.1. Integration Mercado Pago
    - 3.9.2. This project uses CommandLineRunner
  - 3.10. The best way to use it as a suggestion is by using Postman
    - 3.10.0.1. A collection is available preconfigured in the project root
    - 3.10.0.2. Create new user example:
    - 3.10.0.3. Login - Use the username (cpf) and password, then copy the token and use it in authenticated endpoints.
    - 3.10.1. 💡To make it easier use environment variables
      - 3.10.1.1. Place the command in the test tab on /api/auth/login-token
      - 3.10.1.2. Example:
  - 3.11.
    - 3.11.1. This project also uses OpenAPI Specification (Swagger).
      - 3.11.1.1. To access swagger use the URL:
  - 3.12. Roadmap
  - 3.13. License

2. Project Architecture

- Three microservices were developed: order-ws, payment-ws, and status-ws, which communicate via AWS SNS and SQS.
- When an order is created, it is saved in a PostgreSQL database and a message is sent to the ORDER_TOPIC, which then delivers the message to both the STATUS_QUEUE and PAYMENT_QUEUE.
- Upon receipt, these queues store the messages in MongoDB. Whenever a payment is made or there is a change in the order status, messages are sent to the queues to process the updated status.
- In this phase of the project, transitioned to AWS ECS Fargate as a strategic move for economic efficiency and knowledge enhancement.

2.1. Projects Saga Pattern Architecture

2.1.1. Justification for Choosing the SAGA Pattern

The SAGA pattern with choreography was used in the snack bar project involving three separate microservices - order-ws (Order Service), payment-ws (Payment Service), and status-ws (Status Service) - offering a robust and scalable solution for managing distributed transactions and ensuring data consistency in a microservices environment. Below, I detail the reasons and justifications for adopting this pattern in this specific context.

2.1.1.1. Distributed Transactions and Data Consistency

In the snack bar system, the process of placing an order is complex and involves several critical steps, such as order creation, payment processing, and order status updating. Each of these steps is the responsibility of a different microservice. The challenge is to ensure that, even when operating independently, all microservices maintain the system's overall consistency. The SAGA pattern, through a series of local transactions where each step knows how to compensate itself in case of failures, ensures this consistency without the need for a blocking distributed transaction.

2.1.1.2. Decoupling and Flexibility

Using choreography, each microservice operates independently, publishing domain events when their operations are completed or when failures occur. Other services listen to these events and react accordingly, without the need for a central coordinator. This model promotes decoupling and flexibility, allowing each service to evolve, scale, and be maintained independently.

2.1.1.3. Resilience and Failure Recovery

Failures are inevitable in distributed systems. The SAGA pattern, with its compensation-based approach, allows the system to recover gracefully. If a payment fails after an order has been created, the system can automatically cancel the order or attempt a new form of payment, maintaining data integrity. Choreography facilitates this recovery by allowing services to autonomously react to failure events.

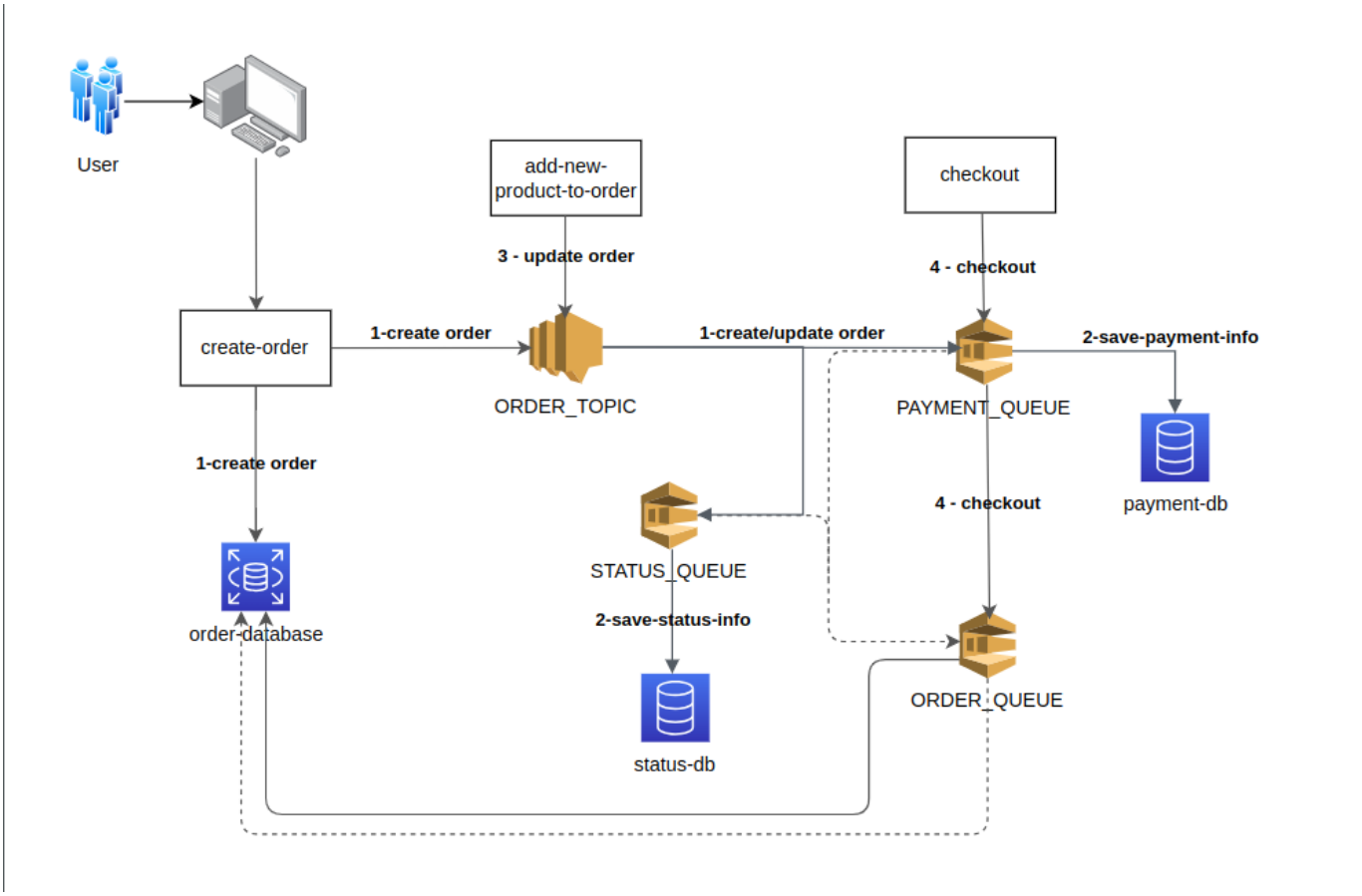
2.1.1.4. Visibility and Monitoring

Although choreography may introduce complexities in tracking the transaction flow, it also offers opportunities to implement effective monitoring and logging at each interaction point. This is crucial in a snack bar environment, where the speed and accuracy of the order are essential. Tracking order, payment, and status events in a unified dashboard can provide valuable operational insights and improve customer experience.

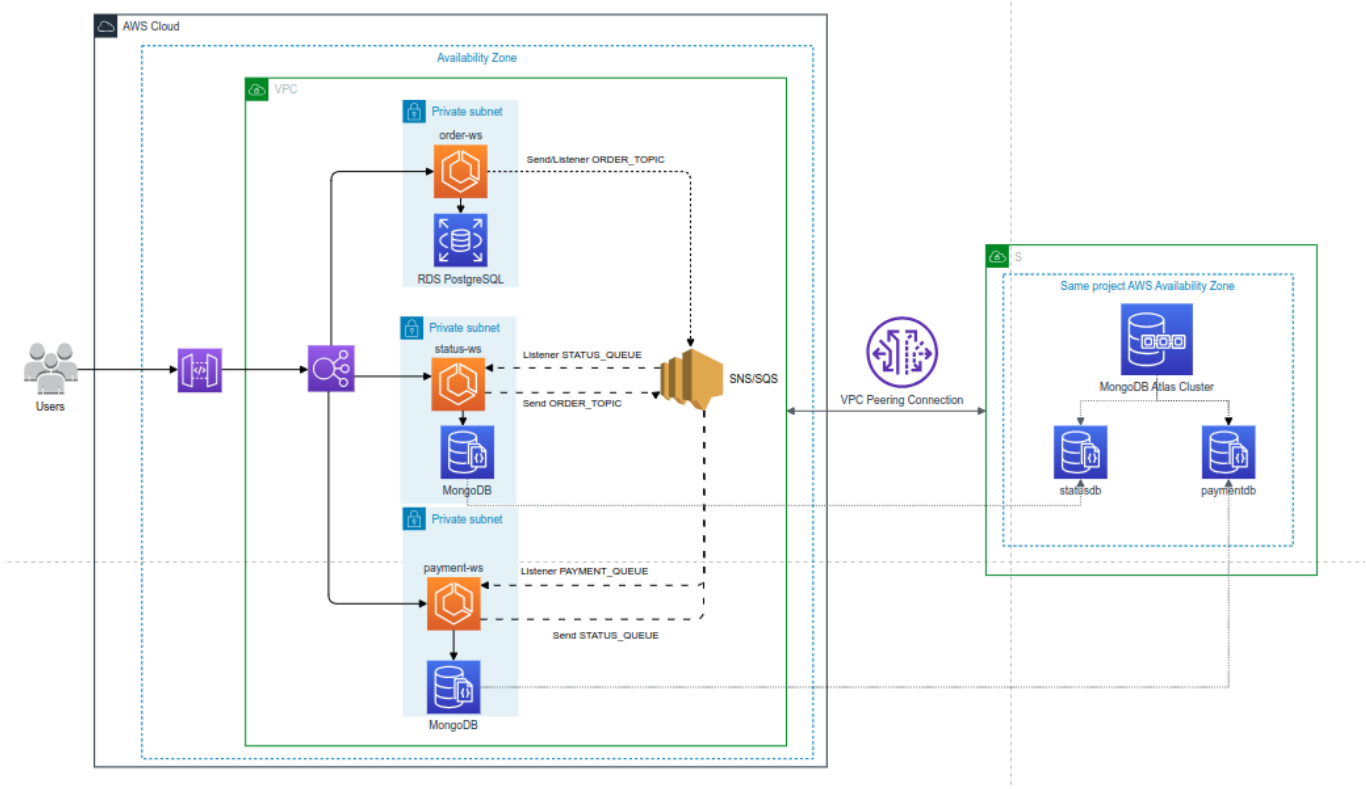
2.1.1.5. Scalability

The SAGA pattern with choreography is naturally scalable. As services operate independently, responding to events asynchronously, the system can easily scale horizontally. This is especially beneficial during periods of high demand in a snack bar, such as lunches or special events, where the number of orders and payments can significantly increase.

Adopting the SAGA pattern with choreography in the snack bar project involving microservices order-ws, payment-ws, and status-ws offers a strategic solution to manage complex and distributed transactions, promoting decoupling, flexibility, resilience, and scalability. Although it presents challenges in terms of traceability and may require a learning curve for effective implementation, the benefits in terms of data consistency maintenance and failure recovery justify its use in this context.



2.2. Projects Cloud Architecture



2.3. OWASP ZAP Report

2.3.1. OWASP ZAP processing reports before correction:

Alert counts by risk and confidence

This table shows the number of alerts for each level of risk and confidence included in the report.

(The percentages in brackets represent the count as a percentage of the total number of alerts included in the report, rounded to one decimal place.)

		Confidence				
		User Confirmed	High	Medium	Low	Total
Risk	High	0 (0.0%)	1 (12.5%)	0 (0.0%)	1 (12.5%)	2 (25.0%)
	Medium	0 (0.0%)	0 (0.0%)	1 (12.5%)	0 (0.0%)	1 (12.5%)
	Low	0 (0.0%)	0 (0.0%)	1 (12.5%)	0 (0.0%)	1 (12.5%)
	Informational	0 (0.0%)	2 (25.0%)	2 (25.0%)	0 (0.0%)	4 (50.0%)
	Total	0 (0.0%)	3 (37.5%)	4 (50.0%)	1 (12.5%)	8 (100%)

2.3.2. OWASP ZAP processing reports after correction:

Alert counts by risk and confidence

This table shows the number of alerts for each level of risk and confidence included in the report.

(The percentages in brackets represent the count as a percentage of the total number of alerts included in the report, rounded to one decimal place.)

		Confidence				
		User Confirmed	High	Medium	Low	Total
Risk	High	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	Medium	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	Low	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	Informational	0 (0.0%)	0 (0.0%)	1 (100.0%)	0 (0.0%)	1 (100.0%)
	Total	0 (0.0%)	0 (0.0%)	1 (100.0%)	0 (0.0%)	1 (100%)

2.4. LGPD Endpoint

2.4.1. Route/API where the client can request the deletion of their personal data from the database used.

2.4.2. API have the following fields:

- Name
- Address;
- Username(CPF or Email)

DELETE {{aws-order-ws}}/api/v1/admin/users/delete-by-username

Params Authorization Headers (9) Body Pre-request Script Tests Settings

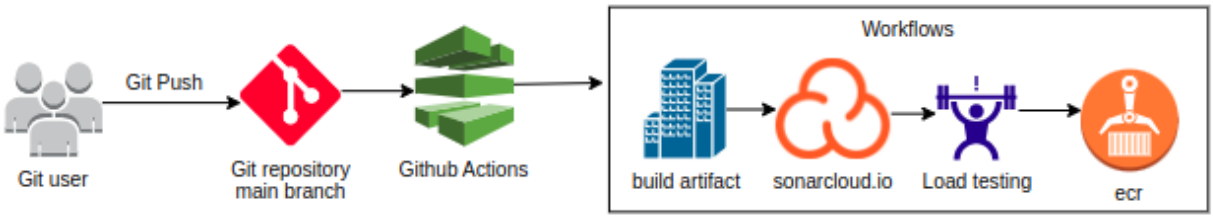
none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "name": "Lucas",
    "username": "lucas@email.com",
    "addressEntity": {
        "street": "rua 1",
        "city": "sp",
        "state": "sp",
        "postalCode": "1234"
    }
}
```

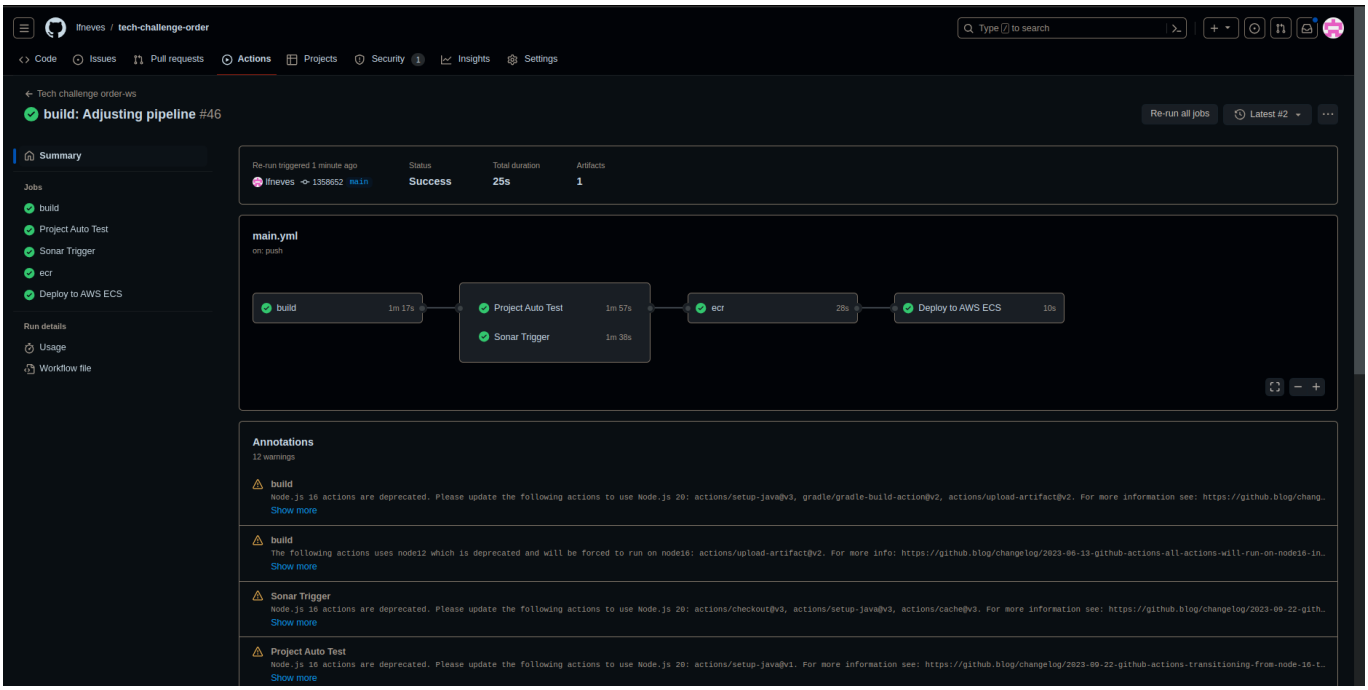
```
@DeleteMapping("/delete-by-username")
@Operation(
    summary = "Deleta usuários por username",
    description = "Deleta usuários by id não valida usuárioa admin neste momento",
    tags = ["Administrador de Usuários"]
)
@ResponseStatus(HttpStatus.NO_CONTENT)
fun deleteById(@RequestBody request: RemoveUserDTO): ResponseEntity<Unit> {
    logger.info("Admin - delete-by-username")
    return ResponseEntity.ok(userAdminService.deleteByUsername(request))
}
```

2.5. Projects Automated Deployment/Test

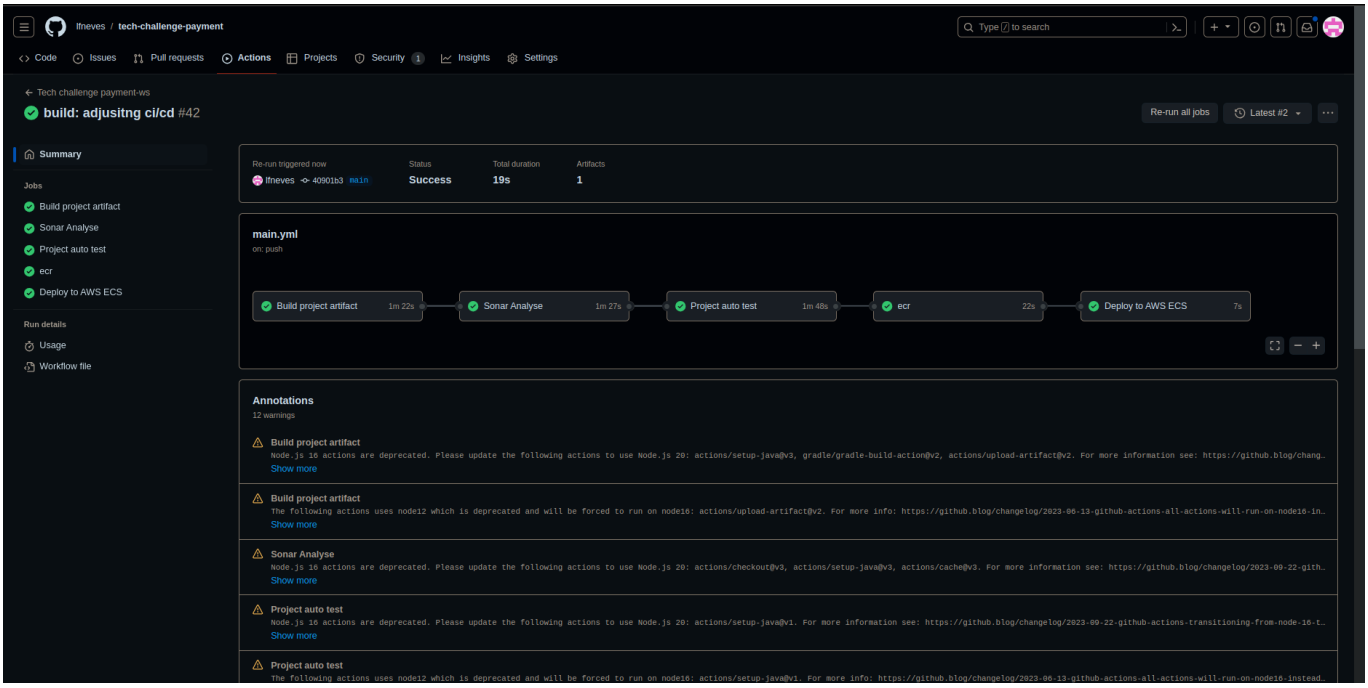
2.5.1. Github Actions Workflow



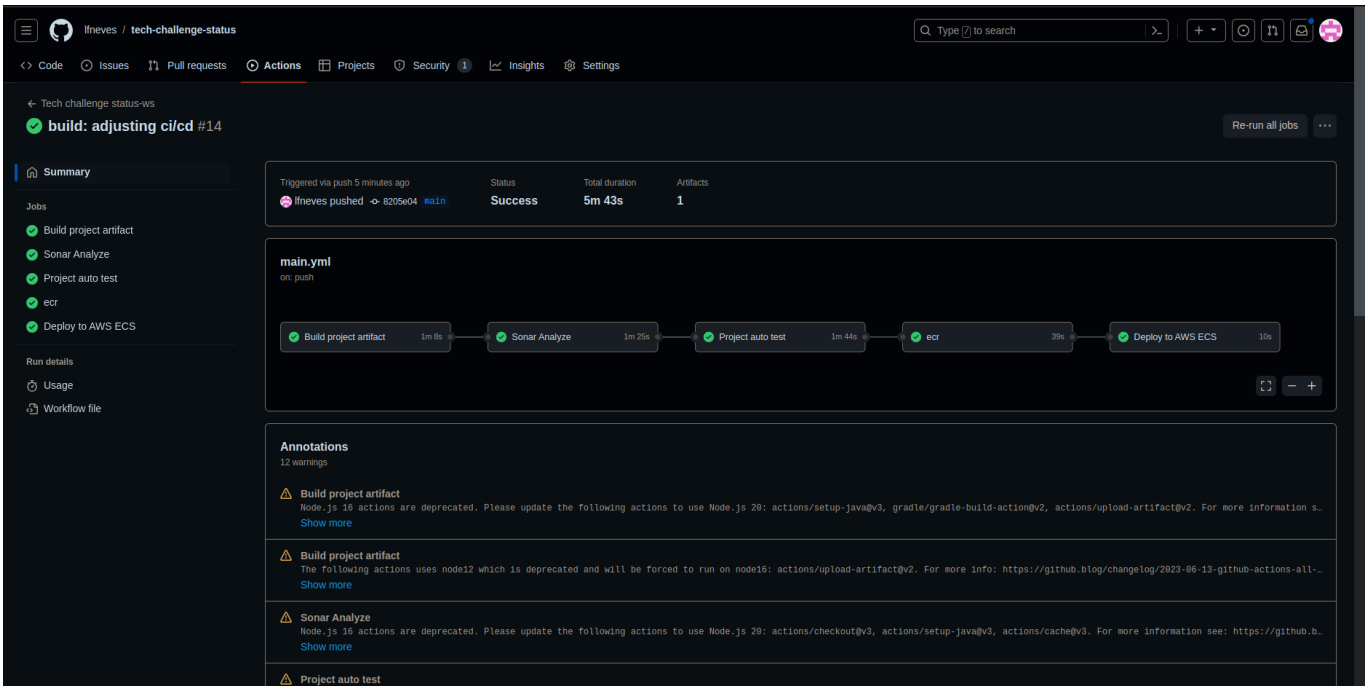
- order-ws



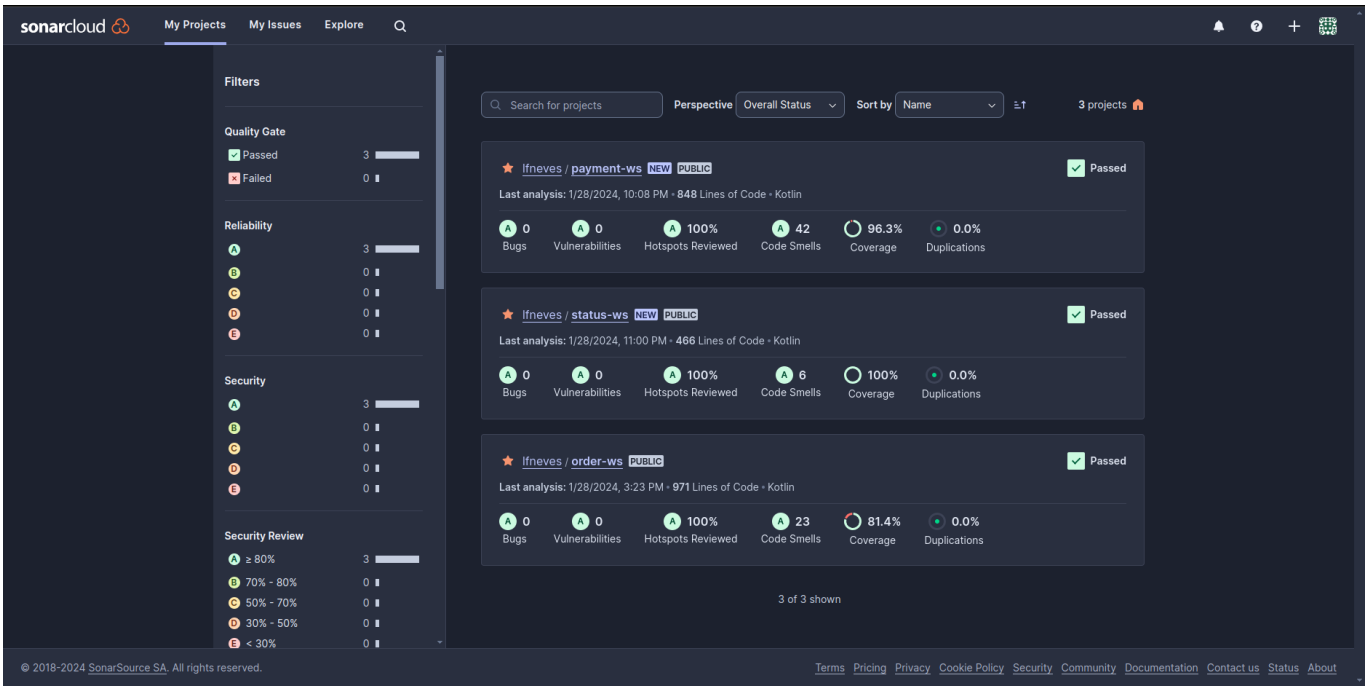
- payment-ws



- status-ws



2.6. Projects Coverage



2.7. Jacoco report

- NOTE: The variance in test coverage between Jacoco and Sonar can be attributed to variations in behavior exhibited by certain filters and integration tests, leading to different or unexpected outcomes.
- The test files are located in the respective folders

2.8. order-ws

- Report file download folder and open in browser: Link order-ws report

order Sessions

order

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mvp.order.domain.service.order		87%		62%	7	20	12	85	1	12	0	1
com.mvp.order.domain.service.product		82%		50%	8	15	3	34	2	9	0	1
com.mvp.order		18%		n/a	1	2	2	4	1	2	1	2
com.mvp.order.application.v1		89%		n/a	1	12	1	16	1	12	0	2
com.mvp.order.domain.service.message		100%		n/a	0	2	0	11	0	2	0	1
com.mvp.order.domain.service.user		100%		100%	0	3	0	8	0	2	0	1
Total	125 of 908	86%	12 of 30	60%	17	54	18	158	5	39	1	8

Created with JaCoCo 0.8.11.202310140853

2.9. payment-ws

- Report file download folder and open in browser: Link payment-ws report

payment-ws Sessions

payment-ws

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mvp.payment.domain.service.payment		80%		56%	7	19	24	114	2	11	0	2
com.mvp.payment.application.v1		77%		n/a	1	5	1	8	1	5	0	1
com.mvp.payment		18%		n/a	1	2	2	4	1	2	1	2
com.mvp.payment.domain.service.listener		88%		50%	5	9	0	17	0	4	0	2
com.mvp.payment.domain.service.message		100%		n/a	0	2	0	11	0	2	0	1
Total	163 of 855	80%	12 of 26	53%	14	37	27	154	4	24	1	8

Created with JaCoCo 0.8.11.202310140853

2.10. status-ws

- Report file download folder and open in browser: Link status-ws report

status-ws Sessions

status-ws

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mvp.status		18%		n/a	1	2	2	4	1	2	1	2
com.mvp.status.domain.service.listener		90%		50%	2	7	0	15	0	5	0	3
com.mvp.status.domain.service.payment		100%		100%	0	5	0	17	0	4	0	1
com.mvp.status.domain.service.message		100%		n/a	0	2	0	11	0	2	0	1
com.mvp.status.application.v1		100%		n/a	0	2	0	4	0	2	0	1
Total	21 of 239	91%	2 of 6	66%	3	18	2	51	1	15	1	8

Created with JaCoCo 0.8.11.202310140853

3. Projects structures:

- order-ws folders

```
└── src
    ├── main
    |   ├── kotlin
    |   |   └── com
    |   |       └── mvp
    |   |           └── order
    |   |               ├── application
    |   |               |   └── v1
    |   |               ├── domain
    |   |               |   ├── configuration
```

```
    |   |           |   ├── model
    |   |           |   |   ├── exception
    |   |           |   |   ├── order
    |   |           |   |   |   └── enums
    |   |           |   |   ├── product
    |   |           |   |   └── user
    |   |           |   └── service
    |   |           |       ├── message
    |   |           |       ├── order
    |   |           |       ├── product
    |   |           |       └── user
    |   |           ├── infrastruture
    |   |           |   ├── entity
    |   |           |   |   ├── order
    |   |           |   |   ├── product
    |   |           |   |   └── user
    |   |           |   └── repository
    |   |           |       ├── order
    |   |           |       ├── product
    |   |           |       └── user
    |   |           └── utils
    |   |               └── constants
    |   └── resources
    |       └── database
    └── test
        ├── kotlin
        |   └── com
        |       └── mvp
        |           └── order
        |               ├── application
        |               |   ├── bdd
        |               |   ├── integration
        |               |   |   ├── order
        |               |   |   |   ├── controller
        |               |   |   |   └── service
        |               |   |   └── user
        |               |   ├── message
        |               |   └── unit
        |               |       ├── order
        |               |       |   └── model
        |               |       ├── product
        |               |       └── user
        |               └── helpers
        └── resources
            ├── features
            ├── schemas
            └── sql
```

- payment-ws folders

```
```

```
└── src
    ├── main
    |   ├── kotlin
    |   |   └── com
    |   |       └── mvp
    |   |           └── payment
    |   |               ├── application
    |   |               |   └── v1
    |   |               ├── domain
    |   |               |   ├── configuration
    |   |               |   ├── model
    |   |               |   |   ├── exception
    |   |               |   |   ├── payment
    |   |               |   |   |   ├── enums
    |   |               |   |   |   ├── listener
    |   |               |   |   |   └── store
    |   |               |   |   └── status
    |   |               |   └── service
    |   |               |       ├── listener
    |   |               |       ├── message
    |   |               |       └── payment
    |   |               ├── infrastruture
    |   |               |   ├── entity
    |   |               |   └── repository
    |   |               └── utils
    |   └── resources
    └── test
        ├── kotlin
        |   └── com
        |       └── mvp
        |           └── payment
        |               ├── application
        |               |   ├── bdd
        |               |   ├── integration
        |               |   ├── message
        |               |   ├── model
        |               |   ├── repository
        |               |   └── unit
        |               └── mongodb
        └── resources
            └── features
```

- status-ws folders

```
└── src
    ├── main
    |   ├── kotlin
    |   |   └── com
    |   |       └── mvp
    |   |           └── status
```

```
|   |           ├── application
|   |           |   └── v1
|   |           ├── domain
|   |           |   ├── configuration
|   |           |   ├── model
|   |           |   |   ├── exception
|   |           |   |   ├── payment
|   |           |   |   |   ├── enums
|   |           |   |   |   └── listener
|   |           |   |   └── status
|   |           |   └── service
|   |           |       ├── listener
|   |           |       ├── message
|   |           |       └── status
|   |           ├── infrastruture
|   |           |   ├── entity
|   |           |   └── repository
|   |           └── utils
|   └── resources
└── test
    ├── kotlin
    |   └── com
    |       └── mvp
    |           └── status
    |               ├── application
    |               |   ├── bdd
    |               |   ├── integration
    |               |   ├── message
    |               |   ├── model
    |               |   └── unit
    |               └── mongodb
    └── resources
        └── features
```

3.1. Terraform AWS EKS Cluster Deployment

Repositorys:

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

3.1.1. AWS Infra Terraform EKS

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.

```
infra-eks-terraform
├── LICENSE
├── README.md
└── infra-eks-terraform
    ├── eks-cluster.tf
    ├── outputs.tf
    ├── providers.tf
    ├── variables.tf
    ├── vpc.tf
    └── workstation-external-ip.tf
```

3.1.2. 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

3.1.3. Repository

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     = 1
node_min_count     = 1
```

3.2. Terraform AWS RDS Database Deployment

3.2.1. 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.

```
infra-rds-terraform
├── LICENSE
├── README.md
└── infra-rds-terraform
    ├── main.tf
    ├── outputs.tf
    ├── table_schema.sql
    ├── variables.tf
    └── versions.tf
```

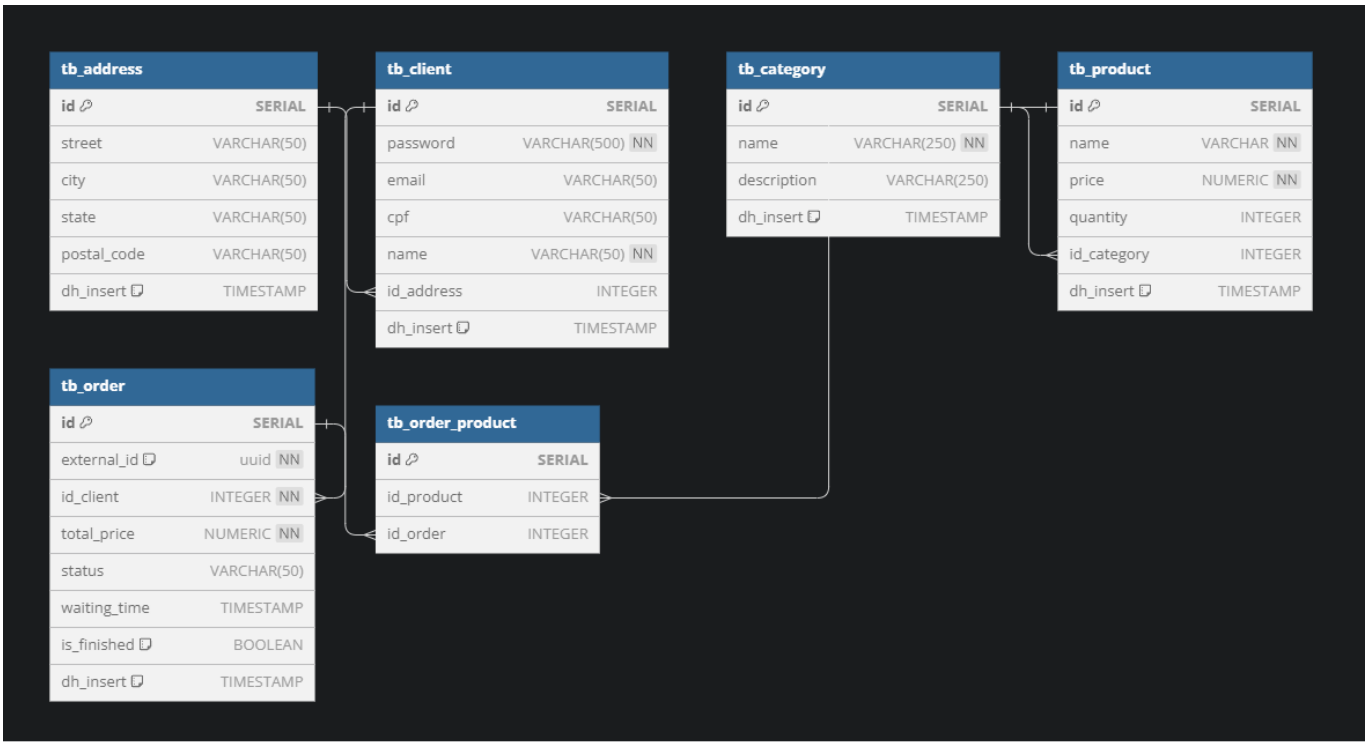
3.3. Justification for Choosing PostgreSQL for a Restaurant System

- **Robust Performance**: PostgreSQL is known for delivering solid performance, even in high transaction volume environments. This is crucial for a restaurant system where it's essential to process orders quickly and efficiently, ensuring a seamless customer experience.
- **Reliability and Stability**: PostgreSQL is renowned for its stability and reliability. Restaurant systems need a database that can handle continuous and critical workloads, minimizing the risk of unexpected failures that could disrupt business operations.
- **Flexible Data Model**: PostgreSQL supports a variety of data types and offers support for advanced features such as foreign keys, indexes, and stored procedures. This allows for flexible data modeling, covering everything from menus and orders to employee and customer information.
- **Advanced Query Capabilities**: PostgreSQL has a powerful SQL query engine that allows for efficient execution of complex queries. This is crucial for generating reports and data analysis that can help restaurant owners and managers make informed decisions.
- **Active Community and Support**: PostgreSQL has an active community of developers and users worldwide. This means you'll have access to ongoing technical support, updates, and security fixes. Additionally, many resources and plugins are available to customize the system to meet the restaurant's specific needs.
- **Cost-Effective**: PostgreSQL is an open-source database, which means it's a cost-effective option compared to many commercial database management systems. This can be especially advantageous for restaurants with budget constraints.
- **Integration and Scalability**: PostgreSQL is highly compatible with many programming languages and can be easily integrated with other parts of the restaurant system, such as online ordering apps, inventory management systems, and more. Furthermore, it is scalable, allowing the database to grow as the restaurant expands its operations.

In summary, PostgreSQL offers a solid combination of performance, reliability, flexibility, and cost-effectiveness, making it a sensible choice for a restaurant system that requires a robust and dependable

database to meet critical business needs.

3.4. Database Diagram



3.4.1. Application mvp

https://github.com/lfneves/mvp

3.4.1.1. 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.

3.5. 💡 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).

3.6. Getting Started

```
# Get the latest version

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

3.7. 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
```

3.7.1. Prerequisites

Check versions:

- Java 17+

  ```
  java --version
  ```

- Docker

  ```
  docker -v
  ```

- Docker Compose

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

3.8. Installation

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

3.8.1. 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
```

3.8.2. Kubernetes (k8s)

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

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

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

3.8.2.2. o access the application URL, use the following command:

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

3.8.2.3. 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
```

3.8.2.4. 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
```

3.8.3. Kubernetes (k8s) - Install with Helm

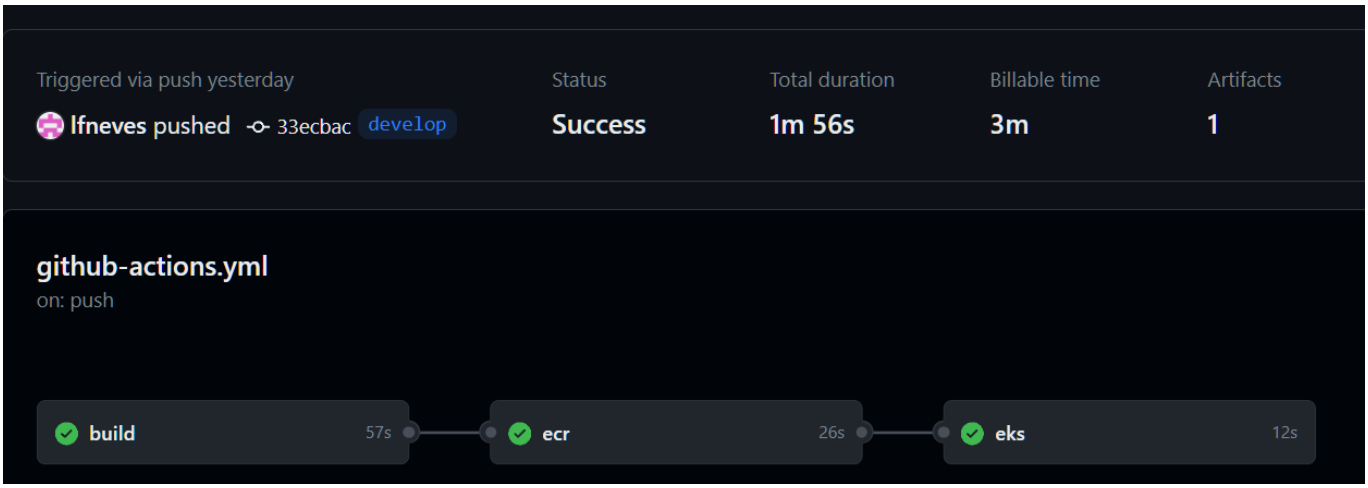
[BETA] Because this hasn't been implemented following best practices.

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

3.8.3.1. Helm uninstall

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

3.9. Deploy Github-actions



3.9.1. 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"
}
```

3.9.2. 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"
}
```

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

3.10.0.1. 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"***

3.10.0.2. 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"
    }
}
```

3.10.0.3. 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:

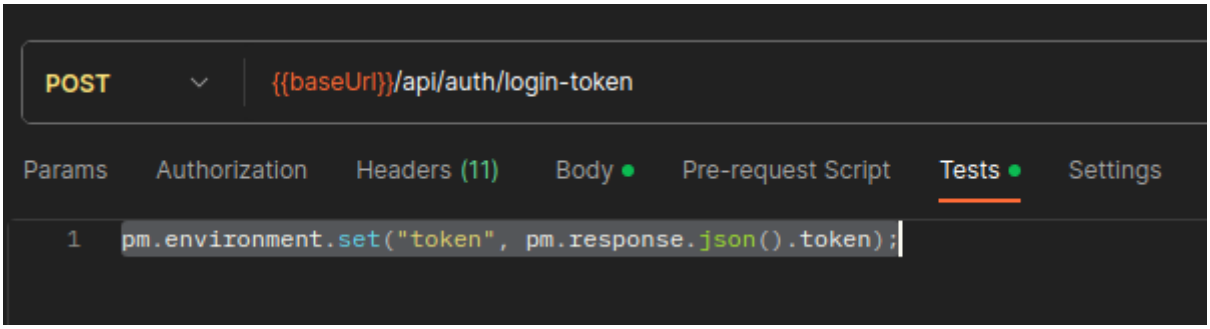
```
{
"token":
"eyJhbGciOiJIUzI1NiJ9.eyJpZENsaWVudCI6IjAiLCJ1c2VybmFtZSI6IjEyMzQ1Njc4OTEyIiwic3ViIjoiMTIzNDU2Nzg5MTIiLCJpYXQiOjE2ODgwOTI1NTAsImF1ZCI6Im5vLWFwcGxpY2F0aW9uLW5hbWUiLCJleHAiOjE2ODgwOTQwMDB9.HagYPqukwOML3OYad8sRjlnE0Gsy-5tGUSC72S-xyfU"
}
```

3.10.1. 💡To make it easier use environment variables

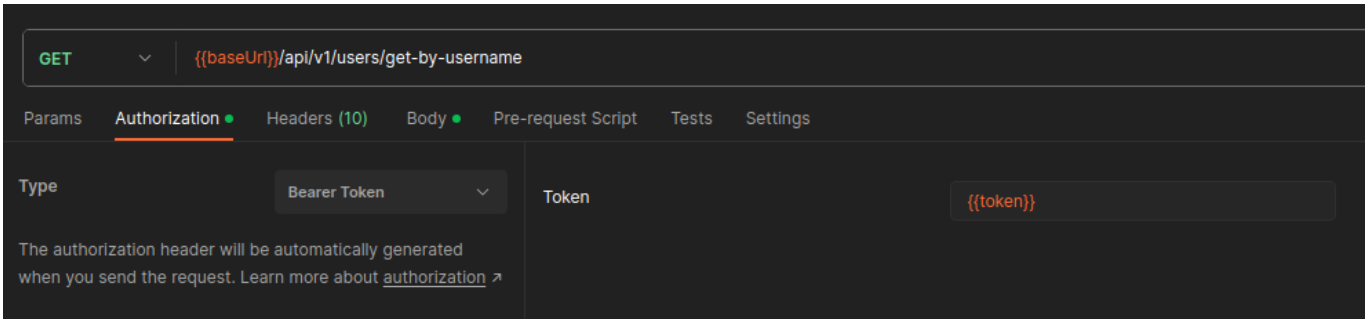
3.10.1.1. Place the command in the test tab on /api/auth/login-token

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

3.10.1.2. Example:



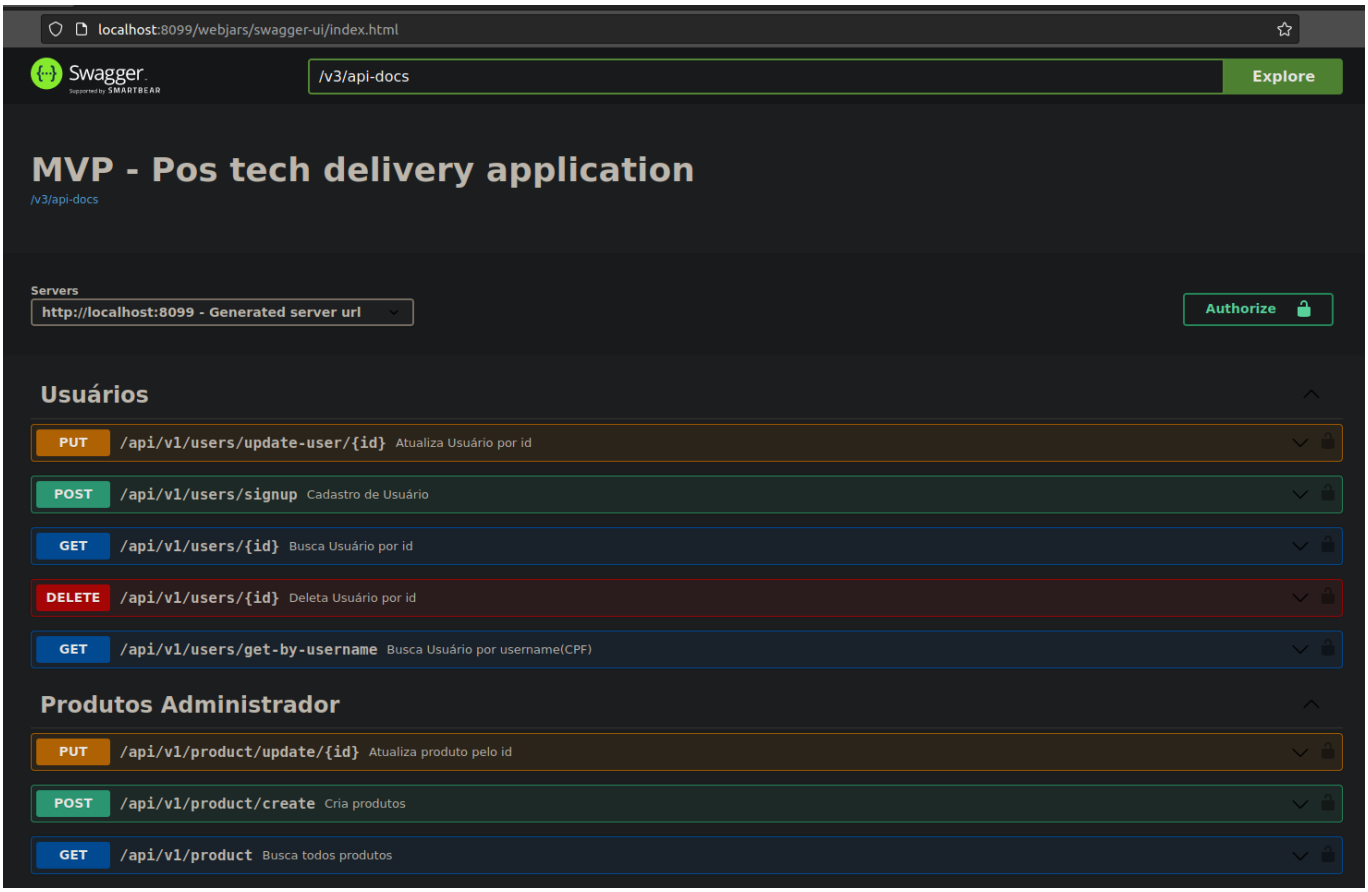
3.11.



3.11.1. This project also uses OpenAPI Specification (Swagger).

3.11.1.1. To access swagger use the URL:

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


localhost:8099/webjars/swagger-ui/index.html
Swagger
/v3/api-docs
Explore
MVP - Pos tech delivery application
/v3/api-docs
Servers
http://localhost:8099 - Generated server url
Authorize
Usuários
PUT /api/v1/users/update-user/{id} Atualiza Usuário por id
POST /api/v1/users/signup Cadastro de Usuário
GET /api/v1/users/{id} Busca Usuário por id
DELETE /api/v1/users/{id} Deleta Usuário por id
GET /api/v1/users/get-by-username Busca Usuário por username(CPF)
Produtos Administrador
PUT /api/v1/product/update/{id} Atualiza produto pelo id
POST /api/v1/product/create Cria produtos
GET /api/v1/product Busca todos produtos

3.12. 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

3.13. License

Distributed under the MIT License. See LICENSE.txt for more information.