

Evention

Mestrado em Engenharia Informática

Pedro Miguel Gomes Martins

Luís Miguel Da Costa Anjo

Diogo Gomes Silva

(nrº 23527, 23528, 23893, regime pós-laboral)

SISTEMAS DE COMPUTAÇÃO EM CLOUD

ESCOLA SUPERIOR DE TECNOLOGIA

INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Conteúdo

1	Introdução	1
1.1	Descrição	1
2	Planeamento	2
2.1	Arquitetura	2
2.2	Endpoints	3
2.2.1	Autenticação	3
2.2.2	Utilizadores	3
2.2.3	Eventos	3
2.2.4	Feedbacks	4
2.2.5	Pagamentos	4
2.2.6	Bilhetes	4
2.3	Estrutura de persistência da Informação	4
2.4	Tecnologias a utilizar	5
2.5	Testes	6
3	Desenvolvimento	7
3.1	Proposta de solução	7
3.2	Utilização do DockerHub e Github Actions	7
3.3	Utilização do Minikube e Docker	8
3.4	Execução de Testes através de Github Actions	13
3.5	Deployment dos containers no cluster Kubernetes	14
3.6	Comunicação entre Serviços	14
3.6.1	Exemplos de uso	15
3.6.2	Testes de Serviços	15
3.7	Métricas de Otimização de Serviços	16
3.7.1	Métricas utilizadas	17
3.8	Repositório Github	20

4 Conclusão

21

Lista de Figuras

2.1	Arquitetura microsserviços	2
2.2	Base de dados	5
3.1	Repositórios do Dockerhub	7
3.2	Release do repositório	8
3.3	Processo Minikube no Docker	8
3.4	GitHub Action executar tests	13
3.5	Exemplo teste unitário Jest	13
3.6	Containers no Minikube	14
3.7	Testes das rotas em Postman - Criar Evento	15
3.8	Testes das rotas em Postman - Aderir a um Evento	16
3.9	Pods em funcionamento antes do teste	18
3.10	Pod Criado Inicialmente chega ao limite	18
3.11	Aumento de replicas	19
3.12	Visualização das réplicas no dashboard	19
3.13	Visualização das réplicas no cmd	19

Lista de Tabelas

2.1	Endpoints da autenticação	3
2.2	Endpoints dos utilizadores	3
2.3	Endpoints dos eventos	3
2.4	Endpoints dos feedbacks	4
2.5	Endpoints dos pagamentos	4
2.6	Endpoints dos bilhetes	4

1. Introdução

No âmbito da unidade curricular de Sistemas de Computação na Cloud foi proposto o desenvolvimento de uma Aplicação Móvel.

A aplicação foi nomeada como Evention e consistirá numa plataforma na qual será possível a criação de eventos e, consequentemente, a disponibilização destes para futuras adesões dos utilizadores.

1.1 Descrição

A plataforma permitirá aos utilizadores a criação de eventos de diferentes tipos, sejam eles gratuitos ou pagos, fornecendo informações detalhadas como a data, hora, localização, descrição e número de participantes.

Ao criar um evento, o utilizador terá a opção de definir para o mesmo, um local ou percurso, que será apresentado num mapa interativo, permitindo aos participantes visualizar a localização ou trajeto sugerido e obter direções detalhadas antes da participação no evento. Para além disso, os utilizadores poderão pesquisar e encontrar eventos disponíveis, quer através de um mapa interativo que mostrará eventos próximos do local onde o utilizador se encontra atualmente, ou pela pesquisa direta de uma localização específica, facilitando assim ao utilizador encontrar eventos relevantes que sejam do seu interesse.

Após a confirmação da adesão de um utilizador em um evento, o criador do mesmo receberá uma notificação automática contendo todas as informações relevantes sobre o novo participante e este receberá um código qr que será usado para confirmar a compra do bilhete no local do evento. Por fim, o utilizador terá a oportunidade de avaliar o evento, proporcionando feedback detalhado sobre a sua experiência.

2. Planeamento

2.1 Arquitetura

Para a nossa aplicação, optámos por implementar a arquitetura de microsserviços. Esta abordagem divide a aplicação em múltiplos serviços pequenos e independentes, cada um responsável por uma funcionalidade específica. Cada API terá uma base de dados independente, permitindo que cada uma aceda às informações das outras APIs conforme necessário.

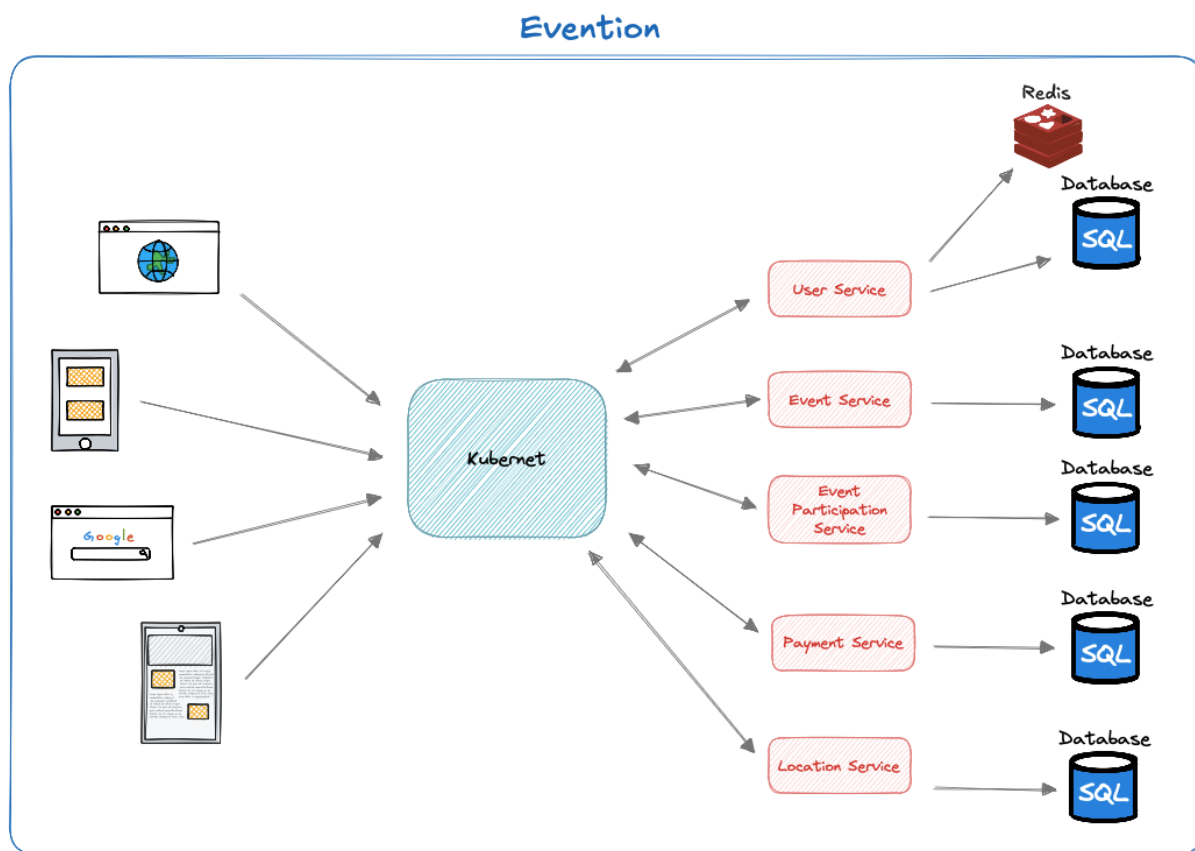


Figura 2.1: Arquitetura microsserviços

2.2 Endpoints

Nas seguintes tabelas, constam os endpoints desenvolvidos para os Microserviços.

2.2.1 Autenticação

Request	Rota	Descrição
POST	api/auth/login	Login do utilizador.
POST	api/auth/logout	Logout do utilizador.
POST	api/auth/loginG	Login do utilizador pelo Google.

Tabela 2.1: Endpoints da autenticação

2.2.2 Utilizadores

Request	Rota	Descrição
POST	api/users/create	Registar um utilizador.
POST	api/users/createG	Registar um utilizador pelo Google.
POST	api/users/validateEmail	Verifica se Email já existe.
UPDATE	api/users/userID	Desativar um utilizador.
DELETE	api/users/userID	Apagar um utilizador.
GET	api/users	Listar todos os utilizadores.
GET	api/users/profile	Retorna Dados do Perfil.

Tabela 2.2: Endpoints dos utilizadores

2.2.3 Eventos

Request	Rota	Descrição
POST	api/events	Criar um evento.
UPDATE	api/events/eventID	Editar um evento.
DELETE	api/eventseventID	Apagar um evento.
GET	api/events	Listar todos os eventos.
GET	api/events/my	Listar todos os meus eventos.
GET	api/events/eventID	Mostrar detalhes de um evento específico.
GET	api/events/eventID/feedbacks	Mostrar feedbacks de um evento.

Tabela 2.3: Endpoints dos eventos

2.2.4 Feedbacks

Request	Rota	Descrição
POST	api/feedbacks	Criar um feedback.
UPDATE	api/feedbacks/feedbackID	Editar um feedback.
DELETE	api/feedbacks/feedbackID	Apagar um feedback.
GET	api/feedbacks	Listar todos os feedbacks.
GET	api/feedbacks/eventID	Listar feedbacks de um evento.

Tabela 2.4: Endpoints dos feedbacks

2.2.5 Pagamentos

Request	Rota	Descrição
POST	api/payments	Criar um pagamento.
GET	api/payments/my	Listar os meus pagamentos.
GET	api/payments	Listar todos os pagamentos.

Tabela 2.5: Endpoints dos pagamentos

2.2.6 Bilhetes

Request	Rota	Descrição
POST	api/tickets	Criar um bilhete.
DELETE	api/tickets/ticketID	Apagar um bilhete.
GET	api/tickets	Listar todos os bilhetes.
GET	api/tickets/my	Devolve os tickets do utilizador.
POST	api/tickets/read	Ler QR Code do bilhete.

Tabela 2.6: Endpoints dos bilhetes

2.3 Estrutura de persistência da Informação

Para implementar a estrutura de persistência de dados da aplicação, será utilizada uma base de dados relacional em PostgreSQL, garantindo robustez, consistência e conformidade, que são fundamentais para a integridade dos dados.

Usamos imagens PostGresSql-Alpine para não ser muito pesada as bases de dados, visto que teremos 5 bases de dados.

O seguinte Diagrama Entidade-Relação (ER) desenvolvido representa o modelo de dados e estabelece a estrutura da base de dados, detalhando as tabelas, os atributos de cada tabela e as relações entre elas.

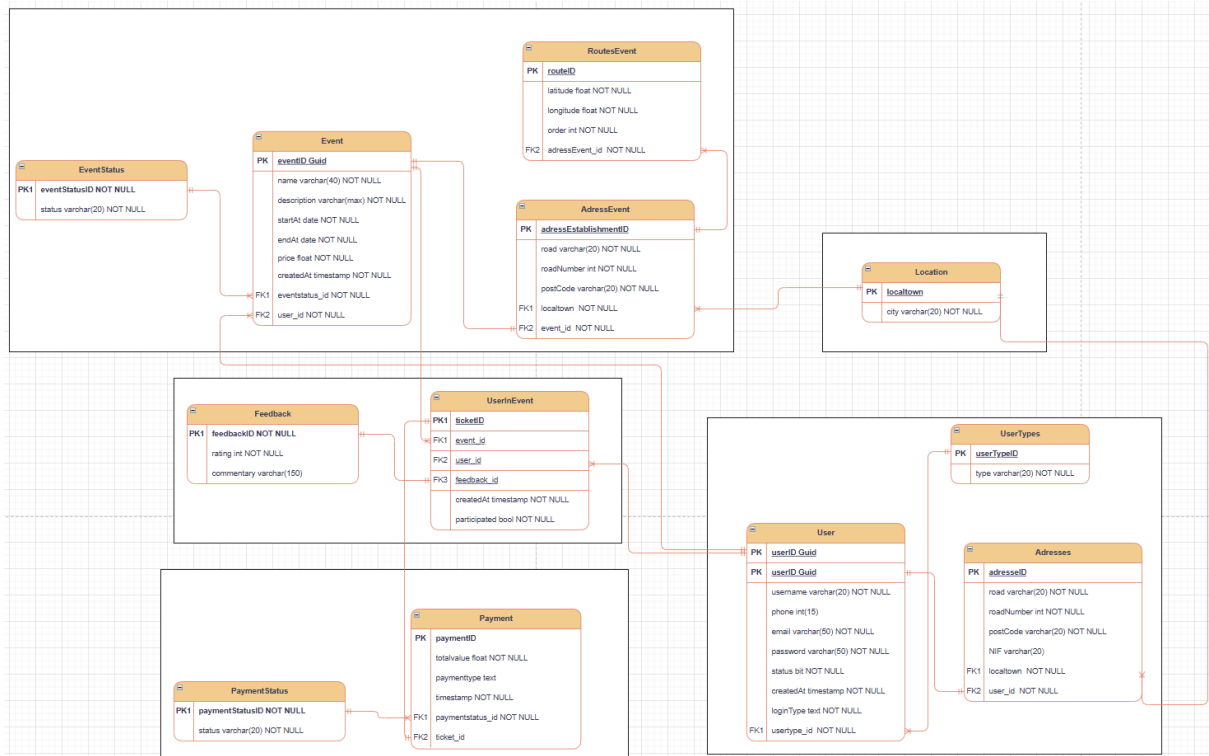


Figura 2.2: Base de dados

2.4 Tecnologias a utilizar

- **Mockups:** Utilizaremos a plataforma Figma para criar os *mockups*.
- **API:** A *API* da plataforma será desenvolvida na linguagem Node.js.
- **Base de Dados:** Os dados da plataforma serão armazenados em uma base de dados no *PostgreSQL*.
- **Docker:** Utilizaremos o Docker para dividir a aplicação em containers, garantindo a consistência do ambiente de desenvolvimento. Também será utilizado o *DockerHub* para guardar e publicar as imagens geradas.
- **GitHub:** Utilizaremos o Github para o devido controlo de versões do código desenvolvido e utilizaremos as Github Actions de forma a automatizar a publicação das imagens e dos testes.
- **MiniKube:** Utilizaremos a plataforma *Minikube* para montar um ambiente de *cluster kubernetes* local na nossa máquina.
- **ORM:** Utilizaremos o Prisma como ORM para gerir a comunicação entre a aplicação e a base de dados.
- **Testes Unitários:** Para realizar os testes unitários utilizaremos a framework Jest.
- **Testes de Integração:** Para realizar os testes de integração utilizaremos o *Postman*.

2.5 Testes

Para garantir a qualidade e fiabilidade da aplicação, realizaremos dois tipos principais de testes: testes unitários e testes de integração.

- **Testes unitários:** Com os testes unitários, vamos validar partes individuais do código, como funções e métodos. Utilizaremos a framework Jest para automatizar estes testes, assegurando que cada função desempenha o seu papel corretamente e que erros são detetados cedo.
- **Testes de integração:** Nos testes de integração, vamos verificar se os diferentes módulos da aplicação funcionam bem em conjunto. Para isso, utilizaremos o Postman, que permitirá simular requisições à API e testar a comunicação entre a aplicação e a base de dados (PostgreSQL).

3. Desenvolvimento

Neste capítulo, será detalhada a proposta de solução e será descrito todo o processo de desenvolvimento da mesma.

3.1 Proposta de solução

De acordo com o enunciado, será criado um cluster local de *Kubernetes* utilizando o *Minikube*, de forma a alojar os serviços da arquitetura de microsserviços apresentada anteriormente, contendo as *API's* definidas e as respectivas bases de dados independentes de cada serviço.

De forma a alcançar este objetivo, foi pretendido desenvolver a arquitetura definida com os serviços descritos, automatizando a publicação dos mesmos através de *GitHub Actions*, e fazer o seu devido *deployment* no *Cluster* de *Kubernetes* utilizando o *Minikube*. Posteriormente, serão executados também testes tanto aos serviços como ao *cluster* construído, utilizando métricas de utilização da memória e de *CPU*.

Todo este processo será descrito em detalhe nos pontos seguintes.

3.2 Utilização do DockerHub e Github Actions

No nosso projeto, utilizamos o DockerHub e o GitHub Actions para automatizar o processo de criação e publicação das imagens Docker de cada microsserviço. Cada API terá a sua respetiva imagem armazenada no DockerHub, facilitando a distribuição e o uso em diferentes ambientes.

No caso dos microsserviços "Event" e "Location", criamos os seguintes repositórios no Dockerhub:

Name	Size	Last Pushed ↑	Contains	Visibility	Scout
pedromartins70/eventservice	0 Bytes	10 minutes ago	IMAGE	Public	Inactive
pedromartins70/locationservice	0 Bytes	12 minutes ago	IMAGE	Public	Inactive

Figura 3.1: Repositórios do Dockerhub

Após a criação dos repositórios, criamos uma action para a imagem do docker e usamos o ficheiro "docker-publish" para automatizar o processo de construção e publicação das imagens Docker no DockerHub, especificando o caminho do repositório onde as imagens serão armazenadas.

Após a criação da action, fizemos um release no repositório do GitHub para acionar o fluxo de trabalho do GitHub Actions, que automaticamente constrói e publica a imagem Docker no DockerHub:

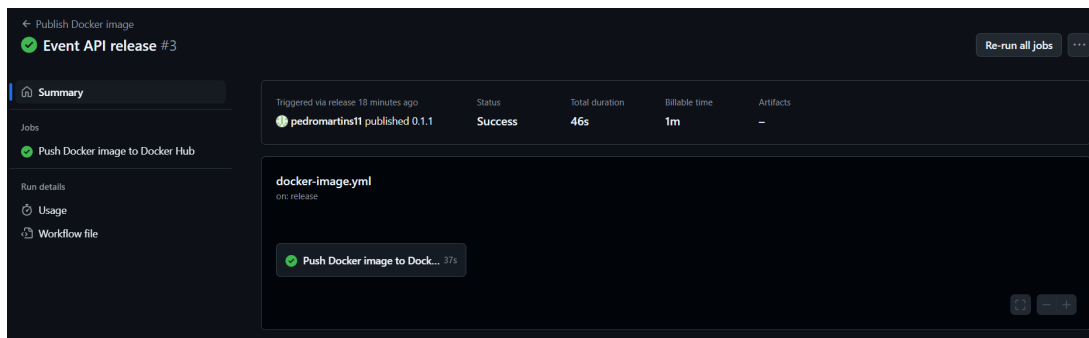


Figura 3.2: Release do repositório

3.3 Utilização do Minikube e Docker

No nosso projeto, utilizamos o Minikube e o Docker para criar um ambiente local de desenvolvimento e testes para os microserviços.

Para utilizar o Minikube em ambiente local, após a sua devida instalação é necessário utilizar o comando "Minikube Start" para iniciar o processo. Este comando irá iniciar o *cluster de kubernetes* local, utilizando no nosso caso o DockerDesktop.

Após a sua execução, o *cluster de kubernetes* já se encontra disponível e podemos visualizar a instancia do minikube no DockerDesktop.

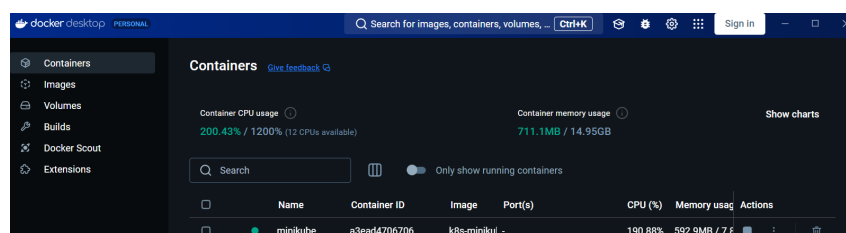


Figura 3.3: Processo Minikube no Docker

Com isto, podemos então proceder à execução dos comandos de forma a controlar o nosso *cluster*, criando serviços e *deployments* de forma a construir os diferentes pods para os serviços que iremos utilizar, que no caso, serão as *API's* e as bases de dados definidas para cada microsserviço.

Para cada microsserviço, utilizamos os ficheiros Deployment e Service para expor os serviços dentro de um cluster *Kubernetes*. O *Deployment* é responsável por definir como o microsserviço será implantado e gerido e o *Service* permite que os microsserviços se comuniquem entre si, assim como expõe portas para acesso externo.

No caso do microsserviço dos "Eventos", criamos o ficheiro "eventservice-deployment". Através do Deployment, podemos garantir que a aplicação esteja sempre em execução e disponível, configurando o número de réplicas desejado que neste caso será 1. Este também especifica a imagem Docker a ser utilizada (pedromartins70/eventservice:latest) e define as variáveis de ambiente necessárias para a conexão com a base de dados, como o host, a porta, o utilizador e a palavra-passe:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: userservice-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: userservice
  template:
    metadata:
      labels:
        app: userservice
    spec:
      containers:
        - name: userservice
          image: a23893/userservicee:main
          ports:
            - containerPort: 5001
          env:
            - name: DB_HOST
              value: "userservice-db-service"
            - name: DB_PORT
              value: "5432"
            - name: DB_USER
              value: "userservice_user"
            - name: DB_PASSWORD
              value: "userservice_pass"
            - name: DB_NAME
              value: "userservice_db"
      resources:
        requests:
          memory: "256Mi"
```

```
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Após a criação do ficheiro de Deployment, criámos o ficheiro "eventservice-service" para expor o microsserviço de forma acessível dentro e fora do cluster Kubernetes.

A criação deste serviço é essencial para garantir que o microsserviço seja acessível e possa comunicar com outros microsserviços ou com sistemas externos. Ao utilizar o tipo LoadBalancer, também conseguimos expor o serviço para acesso externo:

```
apiVersion: v1
kind: Service
metadata:
  name: userservice
  labels:
    app: userservice
spec:
  selector:
    app: userservice
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 5001          #Porta Interna Serviço
      targetPort: 5001    #Porta Pod
```

Foi criado também o ficheiro "postgres-storage" responsável por configurar o armazenamento persistente necessário para o funcionamento da base de dados utilizada pelo microsserviço.

No caso deste microsserviço, configurou-se um PV com uma capacidade de 2Gi, armazenado localmente no diretório /mnt/data/eventservice-db do nó.

Após a configuração do PersistentVolume (PV), foi necessário criar um PersistentVolumeClaim (PVC), que serve como uma interface para que os pods do Kubernetes possam requisitar e utilizar o armazenamento definido no PV.

```
# PersistentVolume para armazenar os dados do PostgreSQL
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv-volume
  labels:
    type: local
    app: userservice-db
spec:
  storageClassName: manual
  capacity:
    storage: 2Gi
```

```
accessModes:
  - ReadWriteMany
hostPath:
  path: "/mnt/data/userservice-db"

---
# PersistentVolumeClaim para utilizar o PV acima
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
  labels:
    app: userservice-db
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
```

Por fim, foi criado o ficheiro "postgres-deployment". Este ficheiro é responsável por gerir o ciclo de vida da base de dados PostgreSQL no ambiente Kubernetes. Utiliza um StatefulSet para garantir que a base de dados tenha uma identidade estável e persistente. Com a configuração de replicas: 1, é definida uma única instância da base de dados, mas o design do StatefulSet permite fácil escalabilidade caso seja necessário. Além disso, o ficheiro configura um Service do tipo LoadBalancer, o que permite que o PostgreSQL seja acessado tanto dentro como fora do cluster.

```
# StatefulSet para gerenciar o PostgreSQL
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: userservice-db
  labels:
    app: userservice-db
spec:
  serviceName: "userservice-db-service"
  replicas: 1
  selector:
    matchLabels:
      app: userservice-db
  template:
    metadata:
      labels:
        app: userservice-db
    spec:
      containers:
```



```
- name: userservice-db
  image: postgres:alpine
  env:
    - name: POSTGRES_USER
      value: "userservice_user"
    - name: POSTGRES_PASSWORD
      value: "userservice_pass"
    - name: POSTGRES_DB
      value: "userservice_db"      # Nome DB
  ports:
    - containerPort: 5432
  volumeMounts:
    - name: userservice-db-data
      mountPath: /var/lib/postgresql/data # Diretório de dados
volumeClaimTemplates:
  - metadata:
      name: userservice-db-data
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 2Gi

---
# Serviço para expor o PostgreSQL dentro e fora do cluster
apiVersion: v1
kind: Service
metadata:
  name: userservice-db-service
  labels:
    app: userservice-db
spec:
  type: LoadBalancer
  ports:
    - port: 5432          # Porta do serviço
      targetPort: 5432    # Porta no container
  selector:
    app: userservice-db
```

3.4 Execução de Testes através de Github Actions

Colocamos na action que publica a imagem da api no docker hub o código que está na figura abaixo para poder automaticamente serem testados os testes unitários existentes na API.

```
jobs:
  push_to_registry:
    name: Push Docker image to Docker Hub
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '23'

      - name: Install dependencies
        run: |
          npm install

      - name: Run tests
        run: |
          npm test

      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${ secrets.DOCKER_USERNAME }
          password: ${ secrets.DOCKER_PASSWORD }
```

Figura 3.4: GitHub Action executar tests

Foram feitos testes unitários ao endpoint e usado o "mock" para simular as ações na base de dados para que desta forma consiga passar nos testes no github.

```
import { app, server } from '../src/server';
const request = require('supertest');
const { PrismaClient } = require('@prisma/client');

jest.mock('@prisma/client', () => {
  const mockPrisma = {
    user: {
      findUnique: jest.fn(),
      findMany: jest.fn(),
      update: jest.fn(),
      delete: jest.fn(),
    },
  };
  return { PrismaClient: jest.fn(() => mockPrisma) };
});

const prisma = new PrismaClient();

describe('User Controller Tests', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  describe('GET /api/users/:id', () => {
    it('should return 200 and a user by ID', async () => {
      const mockUser = { userID: '1', email: 'test@example.com', username: 'user1' };
      prisma.user.findUnique.mockResolvedValue(mockUser);

      const res = await request(app).get('/api/users/1');

      expect(res.status).toBe(200);
      expect(res.body).toEqual(mockUser);
    });
  });
});
```

Figura 3.5: Exemplo teste unitário Jest

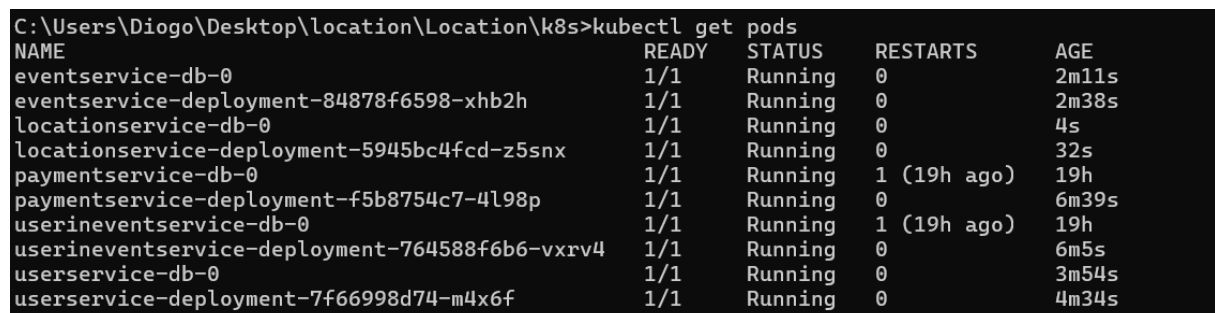
3.5 Deployment dos containers no cluster Kubernetes

De forma a testar a devida comunicação entre os microsserviços da nossa plataforma, foi feito o deployment de todos os serviços e a suas respectivas bases de dados, no cluster de kubernetes no minikube, utilizando os ficheiros descritos anteriormente, de forma a separar corretamente os componentes stateless e statefull.

A ordem de execução dos ficheiros no caso do serviço do User por exemplo, foi a seguinte:

- **userservice-deployment**
- **userservice-service**
- **postgres-storage**
- **postgres-deployment**

Foi efetuado este processo para cada um dos microsserviços e foi executado também o seed das bases de dados, e assim podemos observar os serviços todos no cluster através do minikube, como podemos ver na imagem seguinte:



NAME	READY	STATUS	RESTARTS	AGE
eventservice-db-0	1/1	Running	0	2m11s
eventservice-deployment-84878f6598-xhb2h	1/1	Running	0	2m38s
locationservice-db-0	1/1	Running	0	4s
locationservice-deployment-5945bc4fcd-z5snx	1/1	Running	0	32s
paymentservice-db-0	1/1	Running	1 (19h ago)	19h
paymentservice-deployment-f5b8754c7-4l98p	1/1	Running	0	6m39s
userineventservice-db-0	1/1	Running	1 (19h ago)	19h
userineventservice-deployment-764588f6b6-vxrv4	1/1	Running	0	6m5s
userservice-db-0	1/1	Running	0	3m54s
userservice-deployment-7f66998d74-m4x6f	1/1	Running	0	4m34s

Figura 3.6: Containers no Minikube

3.6 Comunicação entre Serviços

No contexto do nosso sistema, a comunicação entre os diferentes serviços é fundamental para garantir a integração e o funcionamento da aplicação. Para implementar essa comunicação de forma eficiente, optámos por utilizar a biblioteca Axios, uma ferramenta adotada para realizar pedidos HTTP.

3.6.1 Exemplos de uso

No caso do serviço responsável pela criação de eventos, verificamos a existência de um utilizador noutro serviço antes de prosseguir com a operação. Para isso, utilizamos o Axios para fazer um pedido GET ao serviço de utilizadores, conforme o trecho de código abaixo:

```
const userExists = await axios.get
('http://userservice:5001/api/users/${userId}');

if (!userExists) {
  return res.status(404).json({ message: 'User not found' });
}
```

3.6.2 Testes de Serviços

Para garantir o correto funcionamento das chamadas entre os serviços, utilizámos o Postman. Através do Postman, conseguimos simular requisições HTTP, verificar as respostas e validar endpoints.

Para ilustrar o processo de teste realizado com o Postman, apresentamos uma coleção que inclui os principais endpoints do sistema: registo de utilizadores, autenticação (login) e criação de eventos. Esta coleção permitiu validar o fluxo completo de interação entre os serviços:

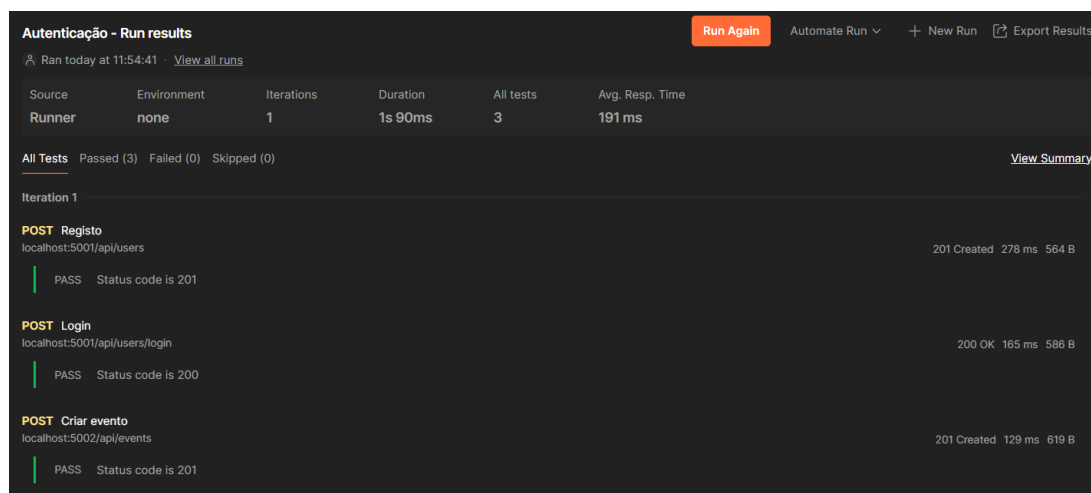


Figura 3.7: Testes das rotas em Postman - Criar Evento

E outra coleção que serve para testar as rotas do processo de adesão a um evento já existente por parte de um utilizador, onde poderá visualizar os seus bilhetes todos e os pagamentos:

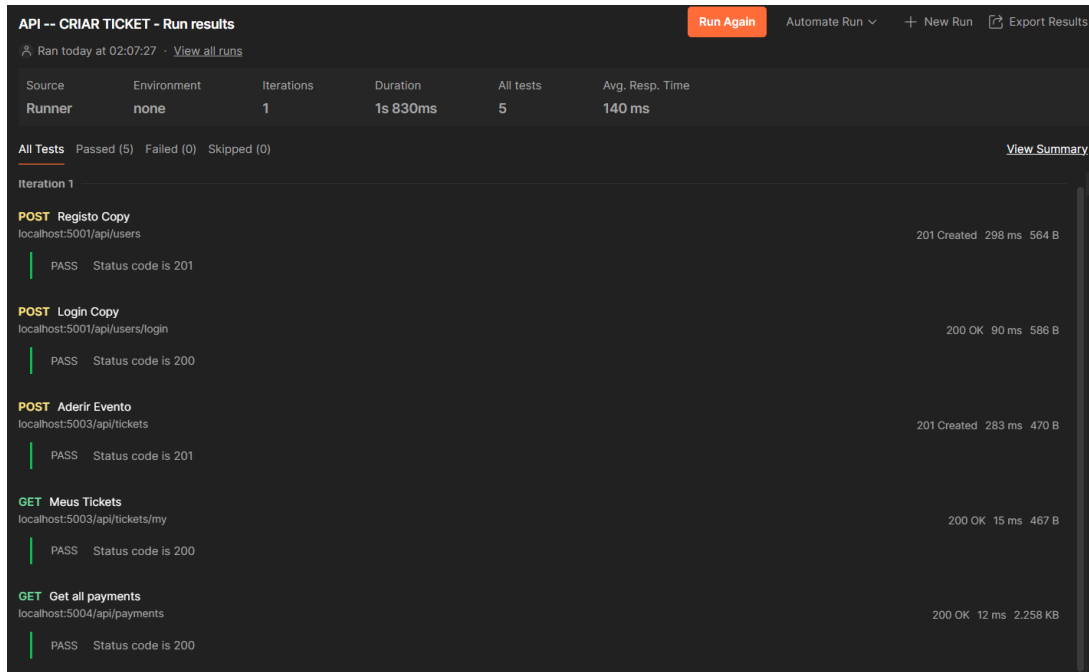


Figura 3.8: Testes das rotas em Postman - Aderir a um Evento

3.7 Métricas de Otimização de Serviços

Para assegurar o desempenho e a escalabilidade dos nossos serviços no ambiente Kubernetes, utilizamos métricas de consumo de CPU e memória como base para avaliar e otimizar a utilização de recursos.

Para visualizar o estado do CPU e da memória de cada pod usamos o seguinte comando:

```
kubect1 top pods
```

Este comando fornece uma visão detalhada do consumo de CPU e memória de cada pod no cluster, permitindo detetar rapidamente padrões de utilização que indicam a necessidade de ajustes, como aumento ou redução do número de réplicas. Esta análise inicial é essencial para identificar serviços que possam beneficiar de escalabilidade automatizada.

3.7.1 Métricas utilizadas

Teve que ser executado o comando abaixo para habilitar o *metrics* server para conseguirmos recolher as métricas de utilização do cpu e memória ram.

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

A partir das métricas observadas, implementámos um Horizontal Pod Autoscaler (HPA) para monitorizar e ajustar automaticamente o número de réplicas do serviço com base em limites predefinidos de CPU e memória. O HPA foi configurado com os seguintes parâmetros:

- Escalonar automaticamente o número de réplicas do pod userservice-deployment quando o consumo de CPU ultrapassar 80%;
- Adicionar uma métrica adicional de memória, acionando o escalonamento quando o consumo de memória ultrapassar 75%;
- Mínimo de 1 réplica e máximo de 10 réplicas.

Definimos também limites de recursos para o container userservice. Os *requests* correspondem à quantidade mínima de memória e CPU que o pod precisa para funcionar corretamente, sendo que, no nosso caso, definimos a memória como 256 Mi e a CPU como 250m (milicores de CPU). Já os *limits* representam o máximo de recursos que o pod pode consumir, que no caso foram definidos como 512 Mi de memória e 500m de CPU.

Inicialmente como colocamos valores muito reduzidos o pod dava o erro "OutOfMemoryKilled (OOMKilled)", pois ultrapassava os valores de limite e estava sempre a reiniciar.

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

A configuração do HPA foi validada utilizando um script para fazer múltiplas requisições a uma rota. Criámos um script (test-load.sh) para fazer requisições simultâneas para os endpoints da API.

O script fez 500 requisições simultâneas, repetidas continuamente para sobrecarregar o sistema.

```
#!/bin/bash

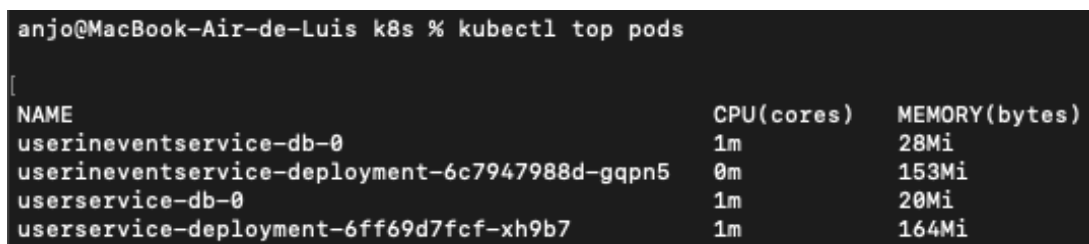
# Número de chamadas simultâneas
CONCURRENT_CALLS=10

# URL da API
API_URL="http://localhost:5001/api/users/afacf97d-e692-43a6-8f79-00dbc30db3b9"

# Função para fazer chamadas à API
make_request() {
    for ((i=0; i<$CONCURRENT_CALLS; i++)); do
        curl -s -o /dev/null -w "%{http_code}\n" -X GET "$API_URL" &
    done
    wait
}

# Executa a função em loop para gerar carga contínua
while true; do
    make_request
    sleep 1
done
```

Como podemos ver na figura abaixo, neste momento ainda não tinha sido executado o script e tal como podemos ver o userservice-deployment só tem 1 pod.



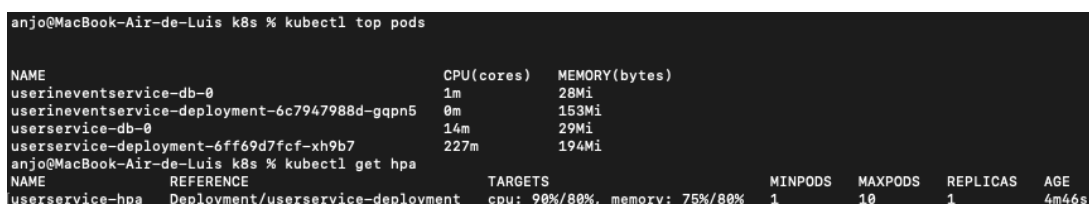
```

anjo@MacBook-Air-de-Luis k8s % kubectl top pods
[
NAME                                CPU(cores)   MEMORY(bytes)
userineventservice-db-0             1m           28Mi
userineventservice-deployment-6c7947988d-gqpn5  0m           153Mi
userservice-db-0                    1m           20Mi
userservice-deployment-6ff69d7fcf-xh9b7       1m           164Mi

```

Figura 3.9: Pods em funcionamento antes do teste

A execução do script desenvolvido anteriormente fez com que aumentasse o número de pods no kubernetes. Na figura abaixo podemos ver o primeiro pod a chegar ao seu limite no CPU começando então a replicação do pod para outros.



```

anjo@MacBook-Air-de-Luis k8s % kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
userineventservice-db-0             1m           28Mi
userineventservice-deployment-6c7947988d-gqpn5  0m           153Mi
userservice-db-0                    14m          29Mi
userservice-deployment-6ff69d7fcf-xh9b7       227m         194Mi
anjo@MacBook-Air-de-Luis k8s % kubectl get hpa
NAME              REFERENCE                                TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
userservice-hpa   Deployment/userservice-deployment        cpu: 90%/80%, memory: 75%/80%   1         10        1          4m46s

```

Figura 3.10: Pod Criado Inicialmente chega ao limite

Após algum tempo de execução e testes, conseguimos alcançar o limite de 6 Pods no nosso cluster Kubernetes.

```

anjo@MacBook-Air-de-Luis k8s % kubectl top nodes
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube      1491m       37%    2318Mi          79%
anjo@MacBook-Air-de-Luis k8s % kubectl get hpa
NAME          REFERENCE                               TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
userservice-hpa  Deployment/userservice-deployment  cpu: 77%/80%, memory: 69%/80%    1         10         6         6m17s

```

Figura 3.11: Aumento de replicas

Além disso, podemos monitorizar o desempenho dos Pods no dashboard do Minikube, onde é possível verificar a utilização de CPU e memória de alguns dos pods replicados.

userservice-deployment-6ff69d7fcf-lcvf7	a23893/userservice-main	app: userservice pod-template-hash: 6ff69d7fcf	minikube	Running	0	1.00m	192.45Mi	26 minutes ago	⋮
userservice-deployment-6ff69d7fcf-dcgdr	a23893/userservice-main	app: userservice pod-template-hash: 6ff69d7fcf	minikube	Running	0	1.00m	198.57Mi	27 minutes ago	⋮
userservice-deployment-6ff69d7fcf-dxw7m	a23893/userservice-main	app: userservice pod-template-hash: 6ff69d7fcf	minikube	Running	0	1.00m	204.91Mi	27 minutes ago	⋮
userservice-deployment-6ff69d7fcf-pttb4	a23893/userservice-main	app: userservice pod-template-hash: 6ff69d7fcf	minikube	Running	0	1.00m	206.81Mi	27 minutes ago	⋮
userservice-deployment-6ff69d7fcf-xh9b7	a23893/userservice-main	app: userservice pod-template-hash: 6ff69d7fcf	minikube	Running	0	0.00m	203.73Mi	32 minutes ago	⋮

Figura 3.12: Visualização das réplicas no dashboard

Por fim, na linha de comandos, conseguimos verificar o número total de 6 réplicas do Pod e as métricas associadas a cada uma delas.

```

anjo@MacBook-Air-de-Luis k8s % kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
userineventservice-db-0             1m           21Mi
userineventservice-deployment-6c7947988d-gqpn5  1m           167Mi
userservice-db-0                     76m          49Mi
userservice-deployment-6ff69d7fcf-5klqd       196m         196Mi
userservice-deployment-6ff69d7fcf-dcgdr       206m         196Mi
userservice-deployment-6ff69d7fcf-dxw7m       210m         201Mi
userservice-deployment-6ff69d7fcf-lcvf7       199m         189Mi
userservice-deployment-6ff69d7fcf-pttb4       228m         204Mi
userservice-deployment-6ff69d7fcf-xh9b7       202m         198Mi
anjo@MacBook-Air-de-Luis k8s % kubectl get hpa
NAME          REFERENCE                               TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
userservice-hpa  Deployment/userservice-deployment  cpu: 82%/80%, memory: 77%/80%    1         10         6         29m
anjo@MacBook-Air-de-Luis k8s % kubectl get hpa
NAME          REFERENCE                               TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
userservice-hpa  Deployment/userservice-deployment  cpu: 79%/80%, memory: 77%/80%    1         10         6         29m

```

Figura 3.13: Visualização das réplicas no cmd

3.8 Repositório Github

Para organizar e facilitar a gestão do código do projeto, criámos uma organização no GitHub dedicada exclusivamente a este propósito.

Dentro da organização, foram criados cinco repositórios, cada um correspondente a uma API do projeto. Cada repositório contém o código-fonte, a documentação técnica e as ferramentas específicas para o desenvolvimento, teste e implementação da respetiva API.

Segue o link da organização: [Evention Team](#).

4. Conclusão

Neste projeto, a utilização do Kubernetes foi fundamental para garantir a escalabilidade, a gestão eficiente e a disponibilidade dos serviços implementados. Através da configuração de Pods, réplicas e serviços, conseguimos criar um ambiente flexível e robusto, capaz de responder de forma eficaz ao aumento de carga e a eventuais falhas no sistema.

Uma das principais mais-valias do Kubernetes foi a sua capacidade de automatizar a gestão de containers, facilitando a administração e a monitorização centralizada de múltiplas instâncias de serviços.

No processo de desenvolvimento deste projeto encontramos vários desafios, na qual foi necessário pesquisar conteúdos no âmbito do projeto e onde conseguimos aplicar os conhecimentos adquiridos na unidade curricular de Computação na cloud.