Projeto Laboratorial – Arquitetura e Desenvolvimento de Microsserviços

Beatriz Santos - 50473, Manoela Azevedo - 50034, Rodrigo Paiva - 49442 e Tiago Fonseca - 49760

Abstract—No âmbito da unidade curricular "Arquitetura e Desenvolvimento de Microsserviços", este relatório descreve o desenvolvimento de um sistema de gestão de tarefas utilizando uma arquitetura de microsserviços. O sistema compreende serviços independentes para autenticação, gestão de utilizadores e gestão de tarefas, orquestrados por um *gateway Nginx* e que comunicam via *REST APIs*, sob um modelo *Request/Response*, onde cada serviço é contido em *Docker* e implementado num ambiente *Kubernetes* local através do *Minikube*. O projeto visa demonstrar a aplicação prática destes conceitos, incluindo a implementação de *JSON Web Tokens* para autenticação e uma interface gráfica web, cumprindo as funcionalidades mínimas e explorando funcionalidades adicionais.

Index Terms—Microsserviços, Docker, Containers, Minikube, Orquestração, REST API .

1 Introdução

Este projeto tem como objetivo principal o desenvolvimento de um sistema para gestão de tarefas, composto por três microsserviços responsáveis por funcionalidades como a autenticação, gestão de utilizadores e tarefas.

O projeto segue requisitos técnicos específicos: a comunicação entre serviços é síncrona; o modelo de colaboração é o *Request/Response*; o estilo arquitetural é *Representational State Transfer (REST)*; a arquitetura de integração baseia-se em orquestração centralizada; é obrigatória a utilização de *Docker* e *Kubernetes* (via *Minikube*) para a implementação individual de cada serviço; e, por fim, o uso de *MariaDB* como ferramenta para a base de dados.

Este relatório técnico também detalha a arquitetura implementada, os métodos e protocolos utilizados, a configuração experimental do ambiente e as evidências do cumprimento das funcionalidades mínimas obrigatórias. O sistema cumpre essas funcionalidades, que incluem um Serviço Orquestrador, um Serviço de Autenticação, um Serviço de Utilizadores e um Serviço de Tarefas. Adicionalmente, foi implementada uma funcionalidade extra: a Interface Gráfica do Utilizador.

2 Estrutura do Software

O sistema desenvolvido segue uma arquitetura baseada em microsserviços para oferecer a funcionalidade de gestão de tarefas. Para este projeto, foram criados serviços para autenticação, gestão de utilizadores e gestão de tarefas.

O componente central da arquitetura é o serviço orquestrador, responsável por receber todas as requisições dos utilizadores através de uma interface gráfica e encaminhá-las para o serviço apropriado. Desta forma, cada microsserviço comunica-se unicamente com o orquestrador, e não diretamente entre si.

Na arquitetura deste sistema, relativamente ao armazenamento de dados, existem duas bases de dados: uma para o armazenamento dos utilizadores e outra para o armazenamento das tarefas. Os seus respetivos serviços — serviço de utilizadores e serviço de tarefas — possuem uma *API* responsável pela comunicação entre essas diferentes áreas.

O *serviço de autenticação* é um caso à parte e comunica-se, através do orquestrador, com o serviço de utilizadores, que valida as credenciais e garante o acesso apenas a utilizadores válidos.

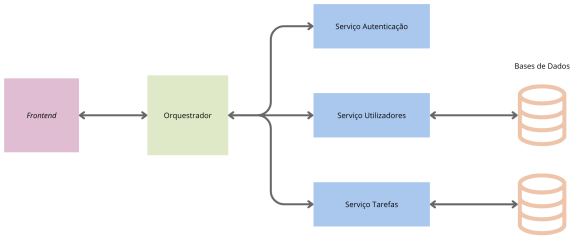

Fig. 1. *Estrutura do Sistema*

2.1 Arquitetura

Para a realização deste projeto, foi adotada a estrutura utilizada no projeto anterior, "Trabalho de Síntese", na qual foi definida uma arquitetura baseada em *Docker*, *Kubernetes* e *Minikube*.

Nesta estruturação, os serviços de microsserviços foram divididos em diferentes *containers*.

O *back-end* do sistema foi desenvolvido em *Node.js* com o framework *Express*, sendo responsável pela implementação

dos serviços e das suas *APIs*. A função de orquestrador é realizada pelo *NGINX*, que é um servidor web utilizado aqui como *proxy reverso* para encaminhar as requisições aos serviços.

Para o armazenamento de dados, optou-se pelo *MariaDB* como sistema de gestão de bases de dados. Na Figura 2, é apresentado um esquema visual dessa arquitetura:

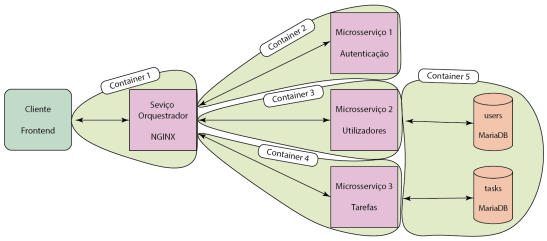

Fig. 2. *Arquitetura do Sistema*

A divisão dos *containers* apresenta-se da seguinte maneira:

- *Container 1*: Orquestrador *NGINX*.
- *Container 2*: Microsserviço de autenticação.
- *Container 3*: Microsserviço de utilizadores.
- *Container 4*: Microsserviço de tarefas.
- *Container 5*: Base de dados *MariaDB*.

O *frontend* é a interface gráfica do utilizador, que comunica-se com o serviço orquestrador. Esta interface foi desenvolvida em *HTML* com recurso à *JavaScript*.

O serviço da base de dados em *MariaDB* é dividido em duas bases de dados, chamadas de *users* e *tasks*. Cada microsserviço que necessita de persistência de dados comunica-se diretamente com a base correspondente, garantindo isolamento e maior organização das informações. Esta separação facilita a manutenção, escalabilidade e segurança dos dados em ambientes distribuídos.

Para garantir que cada microsserviço seja isolado e independente, todos os serviços foram implementados em *Docker containers*. O *Docker* é uma plataforma que permite empacotar o código, as dependências e as configurações necessárias para correr um serviço, garantindo que ele seja executado de forma consistente em qualquer ambiente.

Por exemplo, o serviço de autenticação e o serviço de gestão de tarefas, ambos isolados nos seus respetivos *containers*, foram desenvolvidos, testados e escalados de forma independente. Foi possível a criação de uma rede local para comunicação entre os *containers*, garantindo uma integração eficaz entre os microsserviços.

Já a orquestração dos *containers* foi realizada utilizando *Kubernetes*, com *Minikube* como a ferramenta para criar um *cluster* local, o que permitiu simular um ambiente de produção em um *cluster Kubernetes* local, facilitando a realização de testes e ajustes antes de qualquer *deploy* para produção.

Por exemplo, ao testar a escalabilidade de um serviço específico, como a gestão de tarefas, o *Kubernetes* gerencia automaticamente o número de réplicas do *container* conforme a carga, garantindo alta disponibilidade e desempenho, facilitando assim a implementação de um sistema robusto e pronto para ser escalado conforme necessário, sem complicar o processo de implantação e manutenção.

O *NGINX* desempenha um papel fundamental na gestão do tráfego entre os componentes da aplicação. Para além de encaminhar as requisições, permite aplicar configurações de segurança, cache e balanceamento, contribuindo para uma maior robustez e desempenho do sistema.

A arquitetura é composta por três *microsserviços* principais, cada um encapsulado no seu próprio *container*, com responsabilidades específicas.

2.2 Microsserviços

Microsserviço 1: Autenticação (*Container* 2)

- Responsável pela autenticação dos utilizadores
- Comunica-se com o orquestrador através da API *auth.js.*

Microsserviço 2: Utilizadores (*Container* 3)

- Gere as informações dos utilizadores.
- Comunica-se com o orquestrador através da API *users.js.*

Microsserviço 3: Tarefas (*Container* 4)

- Responsável pela gestão de tarefas atribuídas aos utilizadores.
- Comunica-se com o orquestrador através da API *tasks.js.*

2.3 Protocolos

- *HTTP* para a comunicação entre serviços e com o *frontend*.
- *REST* como estilo arquitetural padrão.
- *JWT* (JSON Web Token) para autenticação segura e sem estado.
- *Docker* como protocolo de empacotamento.
- *Kubernetes* (*Minikube*) como ferramenta de orquestração.

2.4 Algorítmos

2.4.1 Algorítmo de roteamento do NGINX

Nesta arquitetura, o *NGINX* assume o papel de orquestrador central, sendo o ponto único de entrada, orquestrando o fluxo de pedidos entre os serviços.

- O cliente interage com o *frontend*.
- As requisições são encaminhadas para o *NGINX*.
- O *NGINX* redireciona-as para os respetivos serviços.

Desta forma garante-se a existência de uma separação clara de responsabilidades entre os componentes do sistema.

2.4.2 Algorítmo de Autenticação

Nesta arquitetura, o sistema de autenticação garante o controlo de acesso às funcionalidades apenas por utilizadores credenciados. Este processo é baseado em *JSON Web Tokens (JWT)* que permitem a validação de forma simples e segura de cada pedido. O fluxo opera da seguinte forma:

- O cliente envia as credenciais para o endpoint de autenticação.
- O microsserviço de autenticação (*Container 2*) valida as credenciais.
- Após validação bem-sucedida, gera um token *JWT* assinado.
- Este token é devolvido ao cliente para uso em requisições subsequentes.

2.4.3 Algorítmo de Consulta à Base de Dados

No contexto da arquitetura proposta, o sistema de gestão de base de dados *MariaDB* opera como repositório central de informação, seguindo um fluxo otimizado para processamento de queries: Fluxo de Operações:

- O microsserviço correspondente recebe a requisição validada.
- Processamento de Dados.
- Retorno dos Resultados: formatação dos dados em *JSON* pelo microsserviço, acessado através das operações *CRUD* (Create, Read, Update, Delete).

2.5 Métodos

2.5.1 Método de Comunicação entre Microsserviços

API *REST*
Para garantir uma comunicação eficiente entre os microsserviços através de uma *API*, foi adotado o modelo *REST*, que é baseado numa comunicação stateless entre cliente e servidor, onde os recursos são representados por URLs e manipulados utilizando os verbos *HTTP* padrão (*GET*, *POST*, *PUT*, *DELETE*).

Formato
Na arquitetura implementada, a troca de informações entre os diversos componentes do sistema é realizada através do formato *JSON* (*JavaScript Object Notation*), adotado pela sua eficiência e legibilidade.

2.5.2 Método de Serviço

Criptografia de Dados
As palavras-passe dos utilizadores são encriptadas com o auxílio do método de criptografia do tipo hash, *bcrypt*.

Validação de Sessão
Para o controlo da sessão dos utilizadores, utiliza-se a validação de tokens, que permite verificar a autenticidade e validade da sessão em cada pedido.

3 Configuração Experimental

3.1 Estrutura dos Ficheiros do Projeto

```
PROJETO-FINAL-MICROSSERVICOS/
├── Front-End/              # Interface Web
│   ├── login/, register/, tasks/
│
├── MariaDB/                # Base de dados
│   └── mariadb-*.yaml
│
├── Nginx/                  # API Gateway
│   ├── nginx/ (conf)
│   ├── frontend/
│   └── nginx-*.yaml
│
├── NodeJS/                 # Microsserviços
│   ├── Auth/
│   ├── Tasks/
│   └── Users/
│
├── Forward-*.ps1           # Scripts de redirecionamento (port-forward)
└── Start-Everything.ps1    # Script para iniciar tudo
```

Fig. 3. *Estrutura de Ficheiros*

3.2 Docker

Nesta secção, serão apresentados os scripts *PowerShell* utilizados para verificar o estado dos serviços no *Minikube* e estabelecer o encaminhamento de portas entre o ambiente local e os serviços definidos no cluster *Kubernetes*. Estes scripts garantem que os serviços estão a correr e automatizam o acesso local às suas portas internas.

Serviço de Autenticação:

```
kubectl port-forward service/auth-node
    31000:2500 -n auth-node
```

Serviço Utilizadores:

```
kubectl port-forward service/users-node
    32000:3000 -n users-node
```

Serviço Tarefas:

```
kubectl port-forward service/tasks-node
    31500:2750 -n tasks-node
```

Serviço NGINX:

```
kubectl port-forward service/nginx 80:80
     -n nginx
```

Serviço MariaDB:

```
kubectl port-forward service/maria-db
    30000:3306 -n maria-db
```

Automação Completa:

```
# Inicia o Minikube
minikube start

# Obtém o IP
minikube ip

# Executa todos os scripts paralelamente
Start-Process -NoNewWindow -FilePath "
    powershell.exe" -ArgumentList "-File
    ", "./Forward-Database.ps1"
Start-Process -NoNewWindow -FilePath "
    powershell.exe" -ArgumentList "-File
    ", "./Forward-Nginx.ps1"
```

```
Start-Process -NoNewWindow -FilePath "
    powershell.exe" -ArgumentList "-File
    ", "./Forward-Tasks-Node.ps1"
Start-Process -NoNewWindow -FilePath "
    powershell.exe" -ArgumentList "-File
    ", "./Forward-Users-Node.ps1"

# Regista evento de encerramento
$exitingEvent = Register-EngineEvent
    PowerShell.Exiting -Action {
   Write-Host "PowerShell fechado.
       Parando o Minikube..."
   minikube stop
}

# Abre o painel gráfico do Minikube
minikube dashboard

# Impede que o terminal feche
while ($true) {
   Start-Sleep -Seconds 1
}
```

3.3 Serviços

Autenticação
O servidor Node.js foi configurado com Express para lidar com rotas de autenticação. Utiliza-se a biblioteca *dotenv* para variáveis de ambiente, bcrypt para hashing de passwords, e JWT para gestão de sessões.

- **Validação de Sessão (*/isAuthorized*)**: Extrai o *token JWT* do cabeçalho *Authorization*, verifica se é válido através da função *isAuthorized*, e retorna os dados do utilizador se a sessão for válida.
- **Registo de Utilizador (*/register*)**: Permite o registro de novos utilizadores, desde que seja fornecido um *token* de administrador válido na *query string*. Verifica se o nome de utilizador já existe, encripta a senha com *bcrypt*, e cria o novo utilizador na *API* externa.
- **Autenticação (*/login*)**: Recebe *username* e *password*, valida as credenciais através da *API* externa e do *bcrypt*, e gera um *token JWT* que é enviado ao cliente via *cookie* (*clientToken*).
- **Logout (*/logout*)**: Limpa o *cookie clientToken*, encerrando a sessão do utilizador autenticado.

As variáveis de ambiente necessárias estão num ficheiro *.env* e incluem *ADMIN_TOKEN*.

Tarefas

O serviço de tarefas foi desenvolvido com **Node.js** e utiliza o **Express** para gerir rotas e o **MySQL** como base de dados. A biblioteca *util.promisify* é usada para converter as operações de consulta SQL em promessas, facilitando o uso com *async/await*.

- **Listagem de Tarefas (*/getTasks*)**: Verifica a autorização do utilizador e, se válida, retorna todas as tarefas associadas ao *userid* recebido por *query string*.
- **Criação de Tarefa (*/createTask*)**: Recebe *userid*, *title* e *description* no corpo do pedido, valida o utilizador através de uma chamada ao serviço de utilizadores, e insere a nova tarefa na base de dados com *done = 0*.
- **Atualização de Tarefa (*/updateTask*)**: Permite editar os campos *title*, *description* e *done* de uma tarefa existente, identificada por *taskid*.
- **Remoção de Tarefa (*/deleteTask*)**: Apaga a tarefa identificada por *taskid*, desde que a autorização seja válida.

Esse módulo está desenhado para funcionar integrado com o sistema de autenticação e verificação de utilizadores externos, garantindo que apenas utilizadores válidos e autenticados podem criar, ler, atualizar ou apagar tarefas.

Utilizadores

Este ficheiro contém o serviço responsável pela gestão de utilizadores do sistema. Implementado com *Express*, permite consultar, criar, atualizar e apagar utilizadores da base de dados, com verificação de sessão/autorização quando necessário.

- **Listagem de Utilizadores (*/getUsers*)**: Retorna todos os utilizadores presentes na base de dados sem necessidade de autenticação. Executa uma query *SELECT * FROM users* e devolve os resultados.
- **Listagem de um utilizador (*/getUser*)**: Recebe *username* via query string e retorna os dados do utilizador correspondente. Caso o *username* não seja fornecido, retorna erro 400.
- **Buscar um utilizador por ID (*/getUserByID*)**: Recebe *id* via query string, valida a autorização através de uma chamada ao serviço de autenticação (*/api/auth/isAuthorized*) e, se válida, retorna os dados do utilizador correspondente.
- **Criação de novo utilizador (*/createUser*)**: Recebe *username*, *fullname*, *password* e *type* no corpo do pedido, valida se todos os campos estão preenchidos e insere o novo utilizador na base de dados.
- **Atualização de Utilizador (*/updateUser*)**: Permite atualizar os campos *username*, *fullname*, *password* e *type* de um utilizador existente, identificado por *id*. Valida a autorização via serviço externo e só permite a atualização se o tipo de utilizador for *"admin"*.
- **Remoção de utilizador (*/deleteUser*)**: Remove o utilizador identificado por *username*, validando primeiro a autorização via serviço externo e garantindo que o utilizador autenticado é do tipo *"admin"*.

3.4 NGINX

Configuração do *Dockerfile* para criação da imagem do orquestrador adaptado, neste caso, com o ficheiro de configuração para as rotas associado e o *front-end*.

```
FROM nginx:alpine
COPY frontend/ /usr/share/nginx/html/
COPY nginx/routes.conf /etc/nginx/conf.d
   /default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Configuração de rotas do serviço orquestrador, onde indica a porta em que comunica, os caminhos (*URLs*) existentes, o caminho original com o *front-end* e as rotas para cada uma das *API*s por proxy.

```
server {
    listen 80;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }

    location /api/auth/ {
        proxy_pass http://10.96.18.3:2500/
            api/v1/auth/;
    }

    location /api/tasks/ {
        proxy_pass http://10.96.18.4:2750/
            api/v1/tasks/;
    }

    location /api/users/ {
        proxy_pass http://10.96.18.5:3000/
            api/v1/users/;
    }
}
```

3.5 *MariaDB*

Base de Dados *users*
Tabela *users*

Campo	Tipo	Definição
id	*int*	*unique, auto_increment*
username	*longtext*	—
fullname	*longtext*	—
password	*longtext*	—
type	*varchar*	—

Base de Dados *tasks*
Tabela *tasks*

Campo	Tipo	Definição
id	*int*	*unique*
userid	*int*	—
title	*longtext*	—
description	*longtext*	—
done	*int*	—

3.6 *Frontend*

O *frontend* da aplicação foi desenvolvido com *HTML*, *CSS* e *JavaScript*, sendo composto por três páginas principais: autenticação (login), gestão de tarefas e gestão de utilizadores. A interação com o *backend* é realizada por meio de chamadas síncronas, utilizando o *fetch*, à *API*.

4 Resultados Experimentais

Nesta secção, descrevemos o funcionamento do sistema após a sua devida configuração e implementação. O objetivo é demonstrar como os diferentes componentes interagem entre si em tempo de execução, bem como analisar o comportamento do sistema em termos de desempenho, nomeadamente no que diz respeito à latência e ao consumo de memória.

Para realizar a monitorização e validação das operações do sistema, recorremos à ferramenta *Insomnia*, que permitiu simular e observar os pedidos feitos à *API*. Através desta ferramenta, foi possível verificar as respostas devolvidas pelo sistema, os códigos de estado *HTTP*, os tempos de resposta e o conteúdo das mensagens trocadas entre os serviços.

Adicionalmente, acompanhámos a execução dos *containers* responsáveis por cada serviço utilizando comandos de linha como *kubectl get all*. Estes recursos possibilitaram uma análise em tempo real do consumo de *CPU*, memória e rede por parte de cada *container*.

4.1 Rotas

API register
Este processo deve ser o primeiro a ser executado para a criação do primeiro utilizador da plataforma.

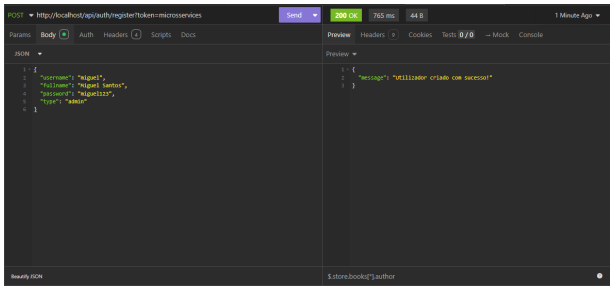

Fig. 4. *Rota Registo*

Estatística:

- Tempo de Resposta: 765 ms
- Utilização de CPU e memória: 44 B

API getUsers
API responsável por buscar todos os utilizadores existentes.

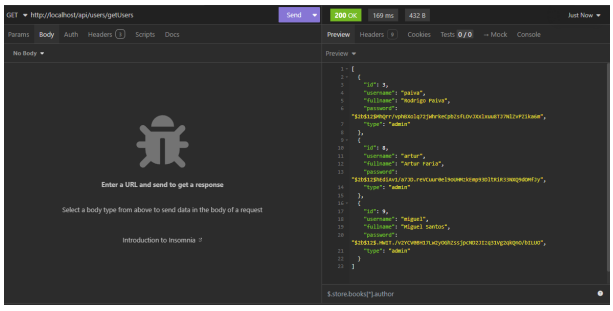

Fig. 5. *Rota Utilizadores*

Estatística:

- Tempo de Resposta: 169 ms
- Utilização de CPU e memória: 432 B

API getUserByID
API responsável por buscar o utilizador através de um id.

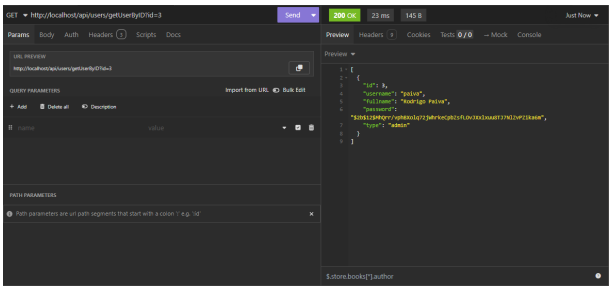

Fig. 6. *Rota Utilizador por ID*

Estatística:

- Tempo de Resposta: 23 ms
- Utilização de CPU e memória: 145 B

API getTasks do User
API responsável por obter a informação sobre as tarefas dos respetivos utilizadores.

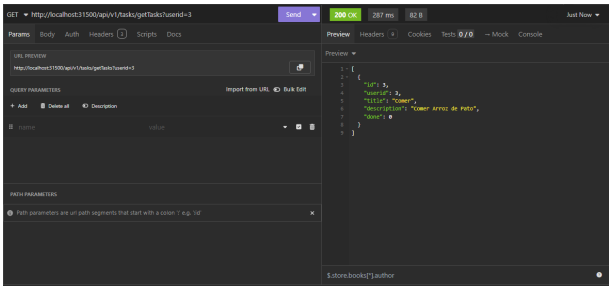

Fig. 7. *Rota Tarefas do Utilizador*

Estatística:

- Tempo de Resposta: 287 ms
- Utilização de CPU e memória: 82 B

API Login
API responsável por autenticar o utilizador.

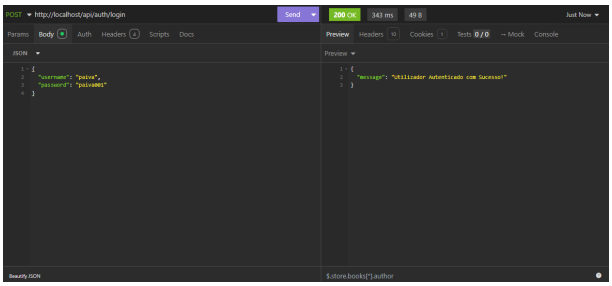

Fig. 8. *Rota Login*

Estatística:

- Tempo de Resposta: 343 ms
- Utilização de CPU e memória: 49 B

4.2 Estado dos *Containers*

Após a execução deste comando é possível observar detalhes da área de trabalho de cada serviço, sendo neste caso possível observar o seu estado (*running*) e o seu ip de comunicação entre serviços.

Autenticação

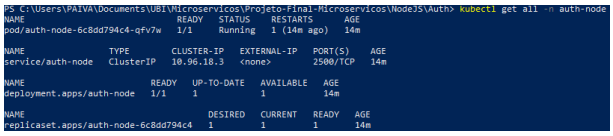

Fig. 9. *Estado do Namespace: Auth*

Utilizadores

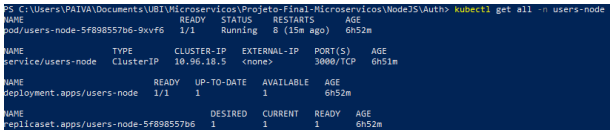

Fig. 10. *Estado do Namespace: Users*

Tarefas

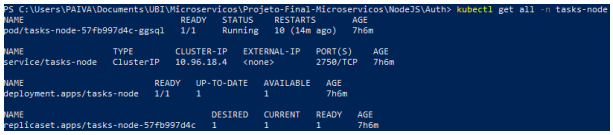

Fig. 11. *Estado do Namespace: Tasks*

Orquestrador

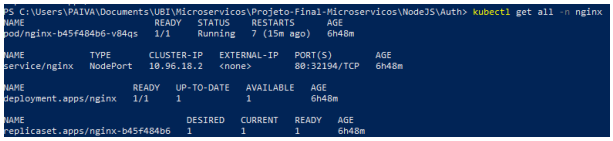

Fig. 12. *Estado do Namespace: NGINX*

Base de Dados

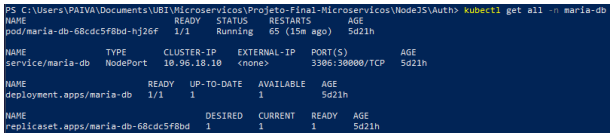

Fig. 13. *Estado do Namespace: MariaDB*

5 Conclusão

No contexto do projeto desenvolvido, foi possivel atingir os objetivos propostos, implementando um sistema de gestão de tarefas baseado em uma arquitetura de microsserviços. Utilizando o *Docker*, *Kubernetes*, *Nginx* e *MariaDB*, conseguimos isolar e orquestrar os serviços de autenticação, gestão de utilizadores e gestão de tarefas de forma eficiente bem como a implementação do sistema foi realizada com sucesso, com foco na escalabilidade e facilidade de manutenção. Todos os requisitos técnicos foram atendidos, e o sistema demonstrou o seu funcionamento adequado, cumprindo as funcionalidades mínimas exigidas e uma

funcionalidade extra (Interface Gráfica do Utilizador).

Para o futuro, uma das principais melhorias seria a implementação de um design mais moderno e uma interface gráfica mais intuitiva, facilitando ainda mais a interação com o utilizador final. Além disso, seria interessante explorar a inclusão de novos serviços, como um sistema de notificações, além de aprimorar a escalabilidade do sistema, permitindo uma gestão de tarefas ainda mais eficiente em ambientes de alta demanda.

6 ANEXOS

- Repositório do projeto no *GitHub*: link
- Video de demonstração: link

REFERENCES

[1] Kubernetes. link. Última vez acedido: 01/06/2025.
[2] minikube - start. link. Última vez acedido: 01/06/2025.