Engenharia e Gestão de Serviços

Relatório: TaskMaster

Eduardo Alves: nºmec 104179, eduardoalves@ua.pt
Rafael Santos: nºmec 98466, rafaelmsantos@ua.pt
Rui Campos: nºmec 103709, ruigabriel2@ua.pt
Gabriel Couto: nºmec 103270, gabrielcouto@ua.pt

Github:
https://github.com/GabrielDCoutoo/TaskMaster

Universidade de Aveiro
2025

Contents

1 Introdução

1.1 Contextualização

O TaskMaster é uma aplicação desenvolvida com o objetivo de implementar um sistema assente numa arquitetura de micro-serviços, orquestrados por um Composer que centraliza a interação entre os diversos componentes do sistema. O projeto visa demonstrar a viabilidade de uma arquitetura modular e escalável, através da integração de micro-serviços independentes, nomeadamente, os serviços de autenticação, notificações e sistema de pontos, desenvolvidos de forma reutilizável e pensados para serem aplicáveis em múltiplos cenários, não estando limitados exclusivamente ao contexto da aplicação TaskMaster.

O Taskmaster visa explorar a gamificação como forma de motivar a participação dos alunos a longo prazo

Este projeto foi feito em colaboração entre estudantes do Departamento de Electrónica, Telecomunicações e Informática (DETI), para o âmbito da disciplina de Engenharia e Gestão de Serviços, responsáveis pelo desenvolvimento do backend, e estudantes do Departamento de Comunicação e Arte da Universidade de Aveiro (DECA), responsáveis pela conceptualização do design do frontend do projeto para o âmbito do projeto Milho a Pardais. Todos os membros participaram de maneira ativa e responsável no projeto.

1.2 Como foi dividida a responsabilidade do projeto?

- **Composer**: Rafael Santos
- **Autenticação**: Gabriel Couto
- **Notificações**: Rui Campos
- **Sistema de Pontos**: Eduardo Alves

1.3 Como está dividido o repositório?

Pasta	Funcionalidade
Composer	Responsável pela gestão de versões do código relativo ao Composer.
Auth	Responsável pela gestão de versões do código relativo ao microserviço da Autenticação.
Notifications	Responsável pela gestão de versões do código relativo ao microserviço de Notificações.
PointSystemAPI	Responsável pela gestão de versões do código relativo ao microserviço de Sistema de Pontos.
Documentos	Pasta responsável pela organização de documentos relevantes ao Projeto.
docker-compose.yaml	Ficheiro responsável pela iniciação e gestão dos diferentes microserviços.

Table 1: Divisão de código

2 Composer

2.1 Aplicação no nosso Projeto

O Composer assume o papel de orquestrador principal da arquitetura de micro-serviços da aplicação *TaskMaster*. Serve como ponto de entrada único para todos os pedidos feitos pelo frontend, analisando cada chamada e encaminhando-a para o micro-serviço correspondente, de acordo com uma lógica de encaminhamento baseada em rotas.

Este módulo abstrai completamente a complexidade da arquitetura interna, permitindo ao cliente interagir com uma interface unificada, sem necessidade de conhecer os detalhes ou localização dos serviços backend.

Para além do encaminhamento, o Composer é responsável por:

- Gerir falhas de comunicação entre serviços, devolvendo mensagens de erro padronizadas em caso de indisponibilidade.
- Centralizar a autenticação e validação de API Keys.
- Registar logs de todas as interações entre serviços, facilitando a monitorização e a deteção de padrões anómalos.

Esta abordagem promove a escalabilidade, manutenção e segurança do sistema, permitindo:

- Adicionar novos micro-serviços sem alterações no frontend.
- Aplicar regras de negócio transversais num único ponto.
- Reduzir o acoplamento entre módulos e facilitar testes.

2.2 Como é que o utilizador interage com o sistema?

O utilizador interage com o sistema através da interface da aplicação (frontend), que comunica com o Composer através de chamadas HTTP. Essas chamadas são processadas pelo Composer, que encaminha a requisição para o serviço correspondente.

Exemplos de Interação:

- Quando o utilizador faz login, a app envia os dados ao Composer, que reencaminha para o serviço de Autenticação.
- Quando o utilizador conclui uma tarefa, o Composer comunica com o sistema de pontos para atualizar a pontuação, e em simultâneo pode ativar uma notificação.
- Quando a app precisa de obter informação consolidada de vários serviços (ex: dados de perfil, progresso, notificações), o Composer orquestra chamadas paralelas e devolve a resposta agregada.

O utilizador nunca comunica diretamente com os micro-serviços individuais – essa responsabilidade pertence unicamente ao Composer, garantindo segurança, organização e escalabilidade.

A presença do Composer no centro da arquitetura permite um elevado grau de controlo e extensibilidade, reduzindo o acoplamento entre serviços e promovendo a reutilização e

segurança. A sua implementação foi essencial para garantir a escalabilidade do nosso sistema, permitindo lidar com diferentes fluxos e comportamentos sem comprometer a integridade do mesmo.

3 Autenticação

3.1 Aplicação no nosso Projeto

O serviço de autenticação da aplicação *TaskMaster* foi construído com base no framework **Flask** e implementa múltiplos mecanismos de autenticação, nomeadamente:

- Autenticação por OAuth2, integrando com provedores externos (ex: WSO2);
- Emissão de tokens **JWT (JSON Web Token)** para controlo de sessão;
- Armazenamento de utilizadores autenticados através da base de dados (usando SQLAlchemy);
- Validação de identidade baseada na presença de uma tag NFC.

A arquitetura garante segurança, escalabilidade e compatibilidade com sistemas modernos, podendo ser facilmente integrada com outros serviços através de headers com JWT.

3.2 Como é que o utilizador interage com o sistema?

O utilizador inicia o processo de autenticação ao clicar num botão de login na aplicação frontend. A partir daí:

1. O frontend redireciona o utilizador para o **servidor OAuth**;
2. Após autenticação, o utilizador é redirecionado de volta para o backend, onde o auth.py:
   - Valida o token de acesso;
   - Gera um token **JWT** assinado com chave secreta;
   - Inicia uma sessão segura com Flask.
3. Este token JWT pode então ser usado para autenticação em pedidos subsequentes.

Trecho de código simplificado do processo:

```
@app.route('/login')
def login():
redirect_uri = url_for('authorize', _external=True)
return oauth.remote_app.authorize_redirect(redirect_uri)

@app.route('/authorize')
def authorize():
token = oauth.remote_app.authorize_access_token()
user_info = oauth.remote_app.parse_id_token(token)

# Gera   o do JWT
jwt_token = jwt.encode({
    "user_email": user_info['email'],
    "exp": datetime.datetime.utcnow() + datetime.timedelta(hours
       =1)
```

```
}, app.secret_key, algorithm="HS256")

return jsonify({"token": jwt_token})
```

Este token JWT é então usado pelos restantes microserviços como prova de identidade do utilizador. A verificação é feita por meio da função decode, utilizando a mesma secret_key.

Além disso, o serviço integra também uma verificação com a base de dados interna de tags NFC (NFCTag), permitindo um mapeamento físico-digital entre cartões e utilizadores.

Exemplo de estrutura do JWT:

```
{
"user_email": "gabrielcouto@ua.pt",
"exp": 1718124000
}
```

O serviço de autenticação é fundamental para garantir que apenas utilizadores válidos acedem ao sistema, sendo responsável por:

- Proteger endpoints sensíveis;
- Validar sessões ativas;
- Identificar os utilizadores em todos os microserviços, através do JWT.

3.3 Deployment do Serviço nos Kubernetes

Abaixo apresenta-se o ficheiro responsável pela orquestração e exposição do serviço de autenticação no cluster Kubernetes:

Ficheiro `auth-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: auth-deployment
namespace: taskmaster
spec:
replicas: 1
selector:
matchLabels:
app: auth
template:
metadata:
labels:
app: auth
spec:
containers:
- name: auth
image: registry.deti/taskmaster/auth:v4
ports:
- containerPort: 5000
env:
```

```
- name: DATABASE_URL
value: postgresql://postgres:password@app-postgres:5432/
   PointSystemEGS
- name: OAUTH_CLIENT_ID
value: u8vpP8lcTQLURxk8itbLvY4pySoa
- name: OAUTH_CLIENT_SECRET
value: REeS1LgzdyZfPmaIXloiNor3cMwa
- name: OAUTH_AUTHORIZE_URL
value: https://wso2-gw.ua.pt/authorize
- name: OAUTH_TOKEN_URL
value: https://wso2-gw.ua.pt/token
- name: OAUTH_USERINFO_URL
value: https://wso2-gw.ua.pt/userinfo
- name: REDIRECT_URI
value: http://auth.taskmaster.deti/callback

apiVersion: v1
kind: Service
metadata:
name: auth-service
namespace: taskmaster
spec:
selector:
app: auth
ports:

    protocol: TCP
    port: 80
    targetPort: 5000
    type: ClusterIP
```

3.4 Comandos para Build, Push e Deploy no Cluster Kubernetes

Para efetuar o deploy do serviço de autenticação, foram utilizados os seguintes comandos:

```
Build da imagem Docker

docker build --network=host -t registry.deti/taskmaster/auth:v4 .
Push da imagem para o registry

docker push registry.deti/taskmaster/auth:v4
Aplicar o deployment no cluster

kubectl apply -f auth-deployment.yaml
Verificar pods e servi os ativos

kubectl get pods -n taskmaster
kubectl get services -n taskmaster
```

Após estes comandos, o serviço passou a estar ativo no namespace `taskmaster`, exposto internamente por um `ClusterIP`, e acessível via DNS interno como `auth-service`.

4 Micro-serviço de NFC

4.1 Aplicação no nosso Projeto

O micro-serviço de NFC foi concebido com o objetivo de registar automaticamente a presença dos alunos nas aulas, utilizando cartões de estudante com tecnologia NFC. Este serviço permite que o docente, na versão da aplicação destinada ao professor, aproxime um leitor NFC ao cartão de um aluno. O ID da tag é então lido e enviado para o backend, onde é tratado por um consumidor Kafka e registado na base de dados.

Este sistema foi desenhado para facilitar a marcação de presenças sem contacto, substituindo os métodos tradicionais de registo manual.

Nota: Por questões de privacidade impostas pela Universidade de Aveiro, não foi possível associar os IDs NFC à informação pessoal dos alunos. Assim, o sistema apenas regista a presença com base no identificador da tag.

4.2 Funcionamento do Sistema

1. O professor acede à aplicação web e aproxima o cartão do aluno ao leitor NFC;
2. O leitor deteta a tag e envia os dados (ex. `tag_id`) via HTTP;
3. O micro-serviço publica os dados no tópico Kafka `nfc_topic`;
4. O consumidor Kafka (`consumer_nfc.py`) regista a tag na base de dados com timestamp;
5. A presença fica registada sem necessidade de autenticação manual.

4.3 Exemplo de Endpoint para Envio de Tag NFC

```
@app.route('/nfc', methods=['POST'])
def receive_nfc_tag():
    data = request.get_json()
    producer.send('nfc_topic', value=data)
    return jsonify({'message': 'Tag␣enviada␣para␣Kafka'}), 200
```

Exemplo de corpo do pedido:

```
{
  "tag_id": "ABCD123456"
}
```

4.4 Registo de Presenças no Consumer

```
consumer = KafkaConsumer(
    'nfc_topic',
    bootstrap_servers=KAFKA_SERVERS,
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    group_id='nfc_group'
)

for message in consumer:
    data = message.value
    tag_id = data["tag_id"]
    nova_presenca = Presenca(tag_id=tag_id, timestamp=datetime.
       utcnow())
    db.add(nova_presenca)
    db.commit()
    print(f"      Presen a registada: {tag_id}")
```

4.5 Deployment em Kubernetes

O sistema foi dividido em dois deployments:

- **nfc-deployment.yaml**: consumidor Kafka e registo na base de dados;
- **server-deployment.yaml**: microservidor Flask que recebe os dados do leitor NFC.

Ficheiro `nfc-deployment.yaml`**:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfc-deployment
  namespace: taskmaster
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nfc
  template:
    metadata:
      labels:
        app: nfc
    spec:
      containers:
      - name: nfc
        image: registry.deti/taskmaster/nfc:v1
        ports:
        - containerPort: 8000
        env:
        - name: DATABASE_URL
          value: postgresql://postgres:password@app-postgres
             :5432/PointSystemEGS
        - name: KAFKA_SERVERS
```

```
          value: kafka:9092
        - name: KAFKA_TOPIC
          value: nfc_topic
```

Ficheiro server-deployment.yaml**:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfc-server
  namespace: taskmaster
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nfc-server
  template:
    metadata:
      labels:
        app: nfc-server
    spec:
      containers:
      - name: nfc-server
        image: registry.deti/taskmaster/nfc-server:v1
        ports:
        - containerPort: 5000
```

4.6 Comandos de Deploy

```
docker build -t registry.deti/taskmaster/nfc:v1 .
docker build -t registry.deti/taskmaster/nfc-server:v1 .

docker push registry.deti/taskmaster/nfc:v1
docker push registry.deti/taskmaster/nfc-server:v1

kubectl apply -f nfc-deployment.yaml
kubectl apply -f server-deployment.yaml
```


```
docker build -t registry.deti/taskmaster/nfc:v1 .
docker build -t registry.deti/taskmaster/nfc-server:v1 .

docker push registry.deti/taskmaster/nfc:v1
docker push registry.deti/taskmaster/nfc-server:v1

kubectl apply -f nfc-deployment.yaml
kubectl apply -f server-deployment.yaml
```

4.7 Conclusão

Este micro-serviço provou ser uma solução eficaz e escalável para registar presenças de forma automatizada, contactless e integrada com o ecossistema da aplicação *TaskMaster*. Apesar das limitações relacionadas com o acesso à informação pessoal dos alunos, o sistema permite manter um registo robusto e auditável, com potencial para expansão futura.

5 Notificações

5.1 Endpoints da API

A aplicação disponibiliza uma API com os seguintes **endpoints**, todos do tipo `POST`:

- `/generate_api_key` – Gera uma nova API Key.
- `/send_notification` – Envia uma notificação push via Expo.
- `/send_email` – Envia um email para o utilizador.
- `/register_token` – Regista o token do dispositivo móvel

5.2 Funcionamento do Serviço de Notificações

5.2.1 Notificações Push via Expo

As notificações push são enviadas para dispositivos móveis Android com recurso à **API Expo**.

Funcionamento:

1. Quando um novo dispositivo é detetado, obtém-se o seu token, que serve como identificador único daquele dispisitivo.
2. Quando um evento importante ocorre (ex: nova tarefa), outro serviço do sistema envia um pedido HTTP ao endpoint `/send_notification`, fornecendo o token, título e corpo da mensagem.
3. O nosso serviço comunica com a API Expo para enviar a notificação.

```
@app.post("/send_notification")
def send_expo_notification(notification: NotificationRequest,
   api_key: str = Depends(validar_api_key)):
    expo_api_url = "https://exp.host/--/api/v2/push/send"
    headers = {
        "Accept": "application/json",
        "Accept-Encoding": "gzip, deflate",
        "Content-Type": "application/json"
    }
    payload = {
        "to": notification.user_token,
        "title": notification.title,
        "body": notification.message
    }

    response = requests.post(expo_api_url, json=payload, headers=
       headers)

    if response.status_code == 200:
        return {"message": "Notificacao Expo enviada com sucesso
           "}
```

5.2.2 Envio de Notificações via Email

Os emails são enviados usando o protocolo **SMTP**, autenticando uma conta Gmail com uma *password*.

Funcionamento:

1. Outro micro-serviço (por exemplo, o sistema de pontos ou tarefas) invoca o endpoint: `/send_email`, enviando no corpo do pedido o endereço de destino, o assunto e a mensagem.
2. O serviço de notificações cria uma mensagem MIME com base nos dados recebidos:
   - Define o remetente e o destinatário.
   - Define o assunto.
   - Anexa o corpo em formato de texto simples.
3. Estabelece uma ligação segura com o servidor SMTP da Google (smtp.gmail.com), utilizando a porta 587 e ativando TLS.
4. Autentica-se com as credenciais fornecidas.
5. Envia o email através do método *send_mail()* da biblioteca `smtplib`.
6. Fecha a ligação com o servidor.

```
@app.post("/send_email")
def send_email(email_request: EmailRequest, api_key: str =
   Depends(validar_api_key)):
    try:
        msg = MIMEMultipart()
        msg["From"] = EMAIL_USER
        msg["To"] = email_request.to_email
        msg["Subject"] = email_request.subject
        msg.attach(MIMEText(email_request.body, "plain"))

        server = smtplib.SMTP(EMAIL_SMTP, EMAIL_PORT)
        server.starttls()
        server.login(EMAIL_USER, EMAIL_PASSWORD)
        server.sendmail(EMAIL_USER, email_request.to_email, msg.
           as_string())
        server.quit()

        return {"message": f"Email enviado para {email_request.
           to_email} com sucesso!"}

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Erro ao
           enviar email: {str(e)}")
```

5.3 Execução Local em Ambiente de Desenvolvimento (Windows)

Para testar localmente o serviço:

```
python app_notifications.py
```

5.4 Deployment do Serviço nos Kubernets

Ficheiro `deployment.yaml`**:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-notifications
  namespace: taskmaster
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app-notifications
  template:
    metadata:
      labels:
        app: app-notifications
    spec:
      containers:
      - name: app-notifications
        image: registry.deti/taskmaster/app-notifications:v1
        resources:
          requests:
            memory: "32Mi"
            cpu: "10m"
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: app-notifications-service
  namespace: taskmaster
spec:
  selector:
    app: app-notifications
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8000
\end{verbatim}
```

```
---

\subsection{Comandos para Build, Push e Deploy no Cluster
   Kubernetes}

\begin{verbatim}
docker build --network=host -t registry.deti/taskmaster/app-
   notifications:v1 .

docker push registry.deti/taskmaster/app-notifications:v1

kubectl apply -f deployment.yaml

kubectl get pods -n taskmaster

kubectl get services -n taskmaster
```

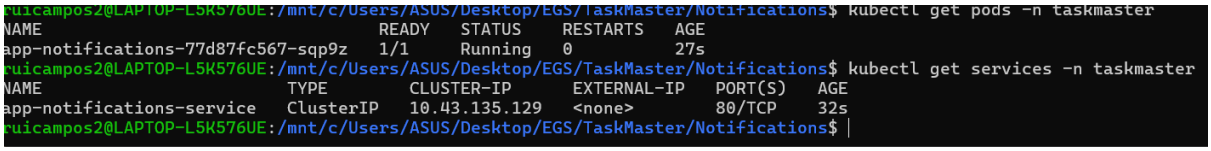

Figura 1: Deploy bem-sucedido no cluster Kubernetes.

Tal como mostra a imagem acima, o serviço foi corretamente implementado, estando o pod ativo e o serviço disponível no namespace `taskmaster`.

5.5 Aplicação no Projeto

No contexto do nosso projeto, o módulo de notificações é utilizado para alertar os utilizadores sobre eventos relevantes. Este módulo funciona de forma independente, podendo ser utilizado por outros serviços como o sistema de tarefas, pontos ou níveis.

Exemplos de utilização:

- Quando uma nova tarefa é atribuída, o sistema chama o endpoint `/send_notification` para alertar o utilizador via push notification.
- Quando um nível é atingido, o sistema envia um email através do endpoint `/send_email`.
- Quando o utilizador instala ou inicia a aplicação, o token do dispositivo é registado no sistema através do endpoint `/register_token`.
- Para fins de segurança e comunicação entre módulos, pode ser gerada uma nova chave de autenticação usando o endpoint `/generate_api_key`.

6 Sistema de Pontos

6.1 Endpoints da API

A aplicação disponibiliza uma API com os seguintes **endpoints**:

- `GET /v1/users/Get Users` – Obtém todas as informações dos utilizadores do sistema.
- `POST /v1/users/Create User` – Permite a criação de um utilizador no sistema.
- `PATCH /v1/users/{user_id}/Update User` – Permite atualizar as informações de um utilizador previamente criado.
- `DELETE /v1/users/{user_id}/Remove User` – Permite remover um utilizador do sistema.
- `POST /v1/users/{user_id}/Points/Add Points` - Permite atribuir pontos a um dado utilizador do sistema.
- `DELETE /v1/users/{user_id}/Points/Remove Points` - Permite remover pontos a um dado utilizador do sistema.
- `GET /v1/points/history/{user_id}/Get User Point History` - Permite ver o histórico de adições e remoções de pontos de um dado utilizador, juntamente com a data e uma mensagem personalizada atribuida na hora da alteração dos pontos.
- `POST /v1/generate-api-key/Generate Api Key` - Permite a criação de uma chave da API.
- `GET /v1/validate-api-key/Validate Api Key` - Permite verificar a validade de uma chave da API.
- `POST /v1/badge/Create Badge` - Permite criar um Badge.
- `POST /v1/badge/assign/Assign Badges` - Permite atribuir Badges aos utilizadores que cumpram as pré-condições.
- `DELETE /v1/badges/{badge_id}/Remove Badge` - Permite remover um Badge.
- `GET /v1/badges/List Badges` - Permite obter a informação de todos os badges disponíveis para atribuição.
- `POST /v1/quests/Create Quest` - Permite a criação de uma Quest.
- `POST /v1/quests/{quest_id}/complete/Complete Quest` - Permite concluir uma tarefa previamente criada.
- `GET /v1/quests/user/{user_id}/Get Quests By User` - Permite obter a informação de todos os quests atribuidos a um dado utilizador.

6.2 Aplicação do PointSystemAPI no Taskmaster

A `PointSystemAPI` foi desenvolvida para servir como o motor de gamificação do nosso projeto **Taskmaster**, permitindo a atribuição, gestão e monitorização de pontos, badges e tarefas (Quests) aos utilizadores do sistema. Esta API é uma componente independente e reutilizável, que comunica com o restante ecossistema do Taskmaster através de chamadas REST e eventos (em versões futuras, via Kafka).

No contexto do **Taskmaster**, a API é utilizada para promover a participação ativa dos alunos e recompensar o seu desempenho com base em critérios definidos pelos docentes, como:

- Presença em aula (confirmada via NFC),
- Entrega atempada de trabalhos,
- Realização de quizzes ou avaliações contínuas,
- Participação em atividades em sala,
- Conclusão de tarefas extra-curriculares.

Cada uma destas ações pode gerar um evento que, ao ser processado, resulta numa chamada à API para atribuição de pontos ao utilizador correspondente. A API permite também:

- Registar novos utilizadores (alunos ou docentes),
- Atualizar e consultar o progresso individual de cada aluno,
- Atribuir **badges** como forma de reconhecimento simbólico por conquistas específicas (ex: "Participante Frequente", "Top 5 da Turma", etc.),
- Criar e concluir **Quests**, ou seja, tarefas que os alunos podem realizar para obter recompensas adicionais,
- Manter um histórico detalhado de alterações nos pontos de cada utilizador, com data e motivo da alteração.

A utilização da API permite automatizar e centralizar a lógica de gamificação do sistema, separando-a da camada de autenticação, das notificações e de diversas outas funcionalidades. Assim, a `PointSystemAPI` oferece uma arquitetura modular e escalável, que pode facilmente ser reutilizada ou adaptada noutros contextos de gamificação como cartões de fidelização de supermercados, empresas de jogos para gestões de recompensas, etc...

A API pode também ser usada como sistema de penalização, uma vez que permite não só adicionar como remover pontos. Caso um aluno não participe nas aulas, o professor poderá escolher remover uma quantidade de pontos à sua escolha a tal aluno. Podendo também justificar a razão pelo qual foi feita a remoçáo de pontos.

Exemplo de Fluxo de Utilização

O aluno autentica-se presencialmente numa aula via NFC;

O sistema principal deteta a sua presença e faz uma chamada à API para adição de um ponto (caso cada presença valha 1 ponto) e atribui esse ponto ao aluno autenticado usando `POST /v1/users/{user_id}/Points/Add Points`;

Caso o aluno atinja determinado número de pontos, um evento é disparado e a API atribui um badge via `POST /v1/badge/assign/Assign Badges`;

O progresso e o histórico podem ser visualizados pelos alunos e professores via `GET /v1/points/history/{user_id}/Get User Point History` e `GET /v1/quests/user/{user_id}/Get Quests By User`.

6.3 Ferramentas tecnológicas utilizadas

O desenvolvimento da `PointSystemAPI` baseou-se no uso de diversas ferramentas modernas, eficientes e orientadas à criação de serviços web leves, rápidos e escaláveis. Abaixo segue a explicação das principais tecnologias utilizadas:

PostgreSQL – Sistema de Gestão de Base de Dados

Para a persistência de dados, foi utilizado o **PostgreSQL**, um sistema de gestão de base de dados relacional (SGBD) robusto, open-source e amplamente utilizado em aplicações de produção. A base de dados é responsável por armazenar duas tabelas, uma responsável pelos utilizadores e outra responsável pelos pontos, e fazer as respectivas relações entre as mesmas.

FastAPI – Framework Web

A API foi desenvolvida usando a **FastAPI**, um framework web moderna e rápida para criação de APIs RESTful em Python. A FastAPI permite definir endpoints com validações claras, tratamento de erros de forma simples e eficiente.

Uvicorn – Servidor ASGI

O servidor utilizado para correr a aplicação é o **Uvicorn**, um servidor leve e de alto desempenho baseado no padrão `ASGI` (Asynchronous Server Gateway Interface). O Uvicorn foi responsável por receber as requisições HTTP, repassá-las à aplicação FastAPI, e devolver as respostas ao cliente de forma eficiente.

6.4 Containerização e Deployment

Para garantir portabilidade, escalabilidade e facilidade de deployment, a PointSystemAPI foi containerizada com Docker, uma ferramenta que permite criar ambientes isolados e replicáveis através de contentores, e implementada num cluster Kubernetes.

6.5 Deployment do Serviço nos Kubernets

Ficheiro `deployment.yaml`**:**

```
apiVersion: v1
kind: Secret
metadata:
  name: app-points-db-secret
  namespace: taskmaster
type: Opaque
stringData:
  DATABASE_URL: postgresql+psycopg2://app_user:app_pass@app-
     postgres:5432/app_db
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-points
  namespace: taskmaster
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app-points
  template:
    metadata:
      labels:
        app: app-points
    spec:
      containers:
      - name: app-points
        image: registry.deti/taskmaster/app-points:v1
        resources:
          requests:
            memory: "32Mi"
            cpu: "10m"
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 8000
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: app-points-db-secret
              key: DATABASE_URL
---
apiVersion: v1
kind: Service
metadata:
  name: app-points-service
  namespace: taskmaster
spec:
```

```
  selector:
    app: app-points
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-postgres
  namespace: taskmaster
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app-postgres
  template:
    metadata:
      labels:
        app: app-postgres
    spec:
      containers:
      - name: postgres
        image: postgres:15
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_DB
          value: app_db
        - name: POSTGRES_USER
          value: app_user
        - name: POSTGRES_PASSWORD
          value: app_pass
        resources:
          requests:
            memory: "64Mi"
            cpu: "50m"
          limits:
            memory: "256Mi"
            cpu: "500m"
        volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgres-storage
      volumes:
      - name: postgres-storage
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: app-postgres
  namespace: taskmaster
spec:
  selector:
    app: app-postgres
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432

-----------------------------------------------------
```

Para isto, foi usado os seguintes comandos:

```
docker build --network=host -t registry.deti/taskmaster/app-
   points:v1 .
docker push registry.deti/taskmaster/app-points:v1
kubectl apply -f deployment.yaml -n taskmaster
kubectl get pods -n taskmaster
kubectl get services -n taskmaster
```

7 Composer- anexos

7.1 App Aluno

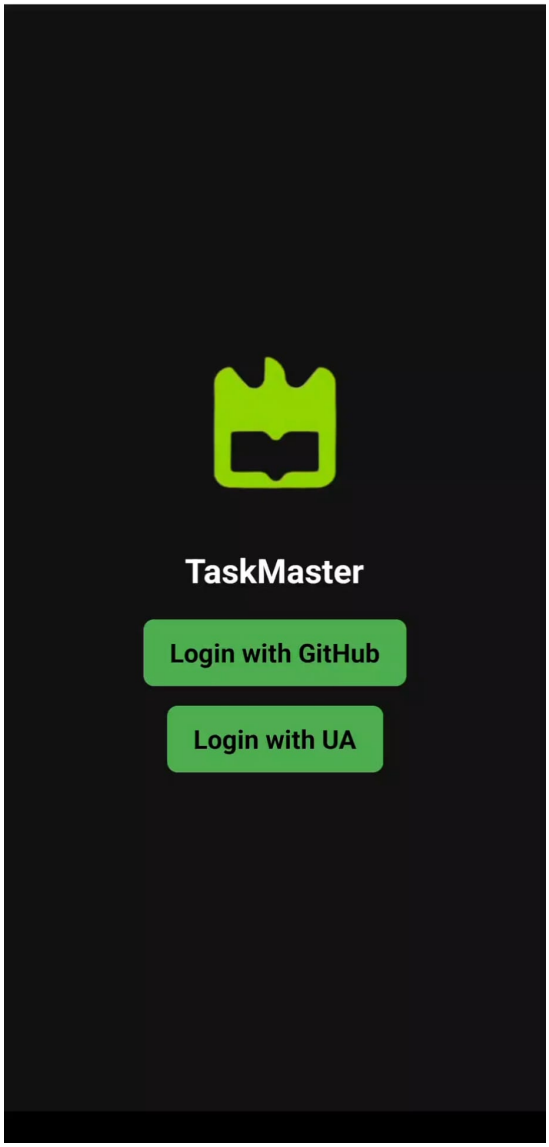

Figure 1: Login

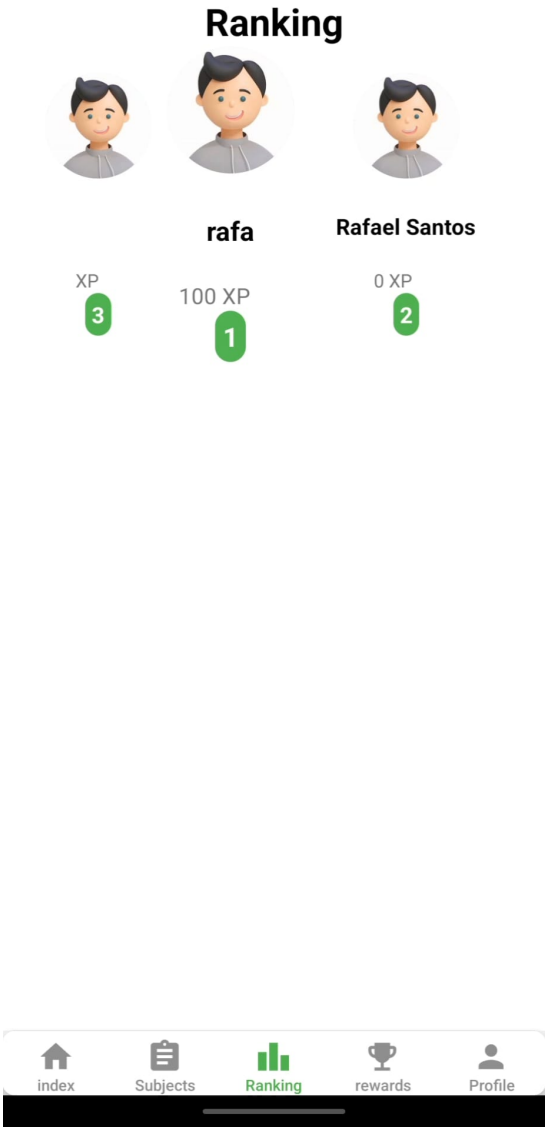

Figure 2: Tabela de pontos

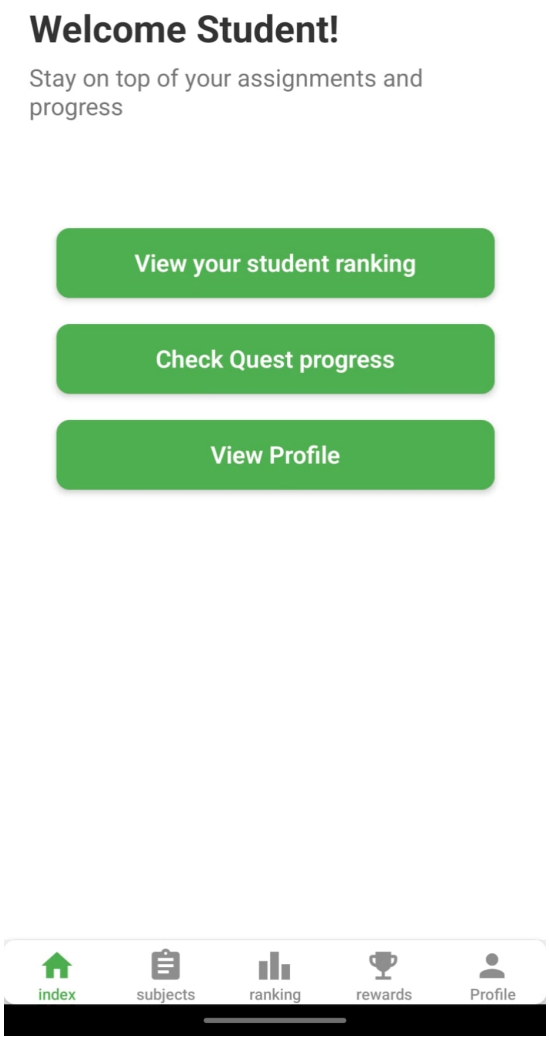

Figure 3: Index

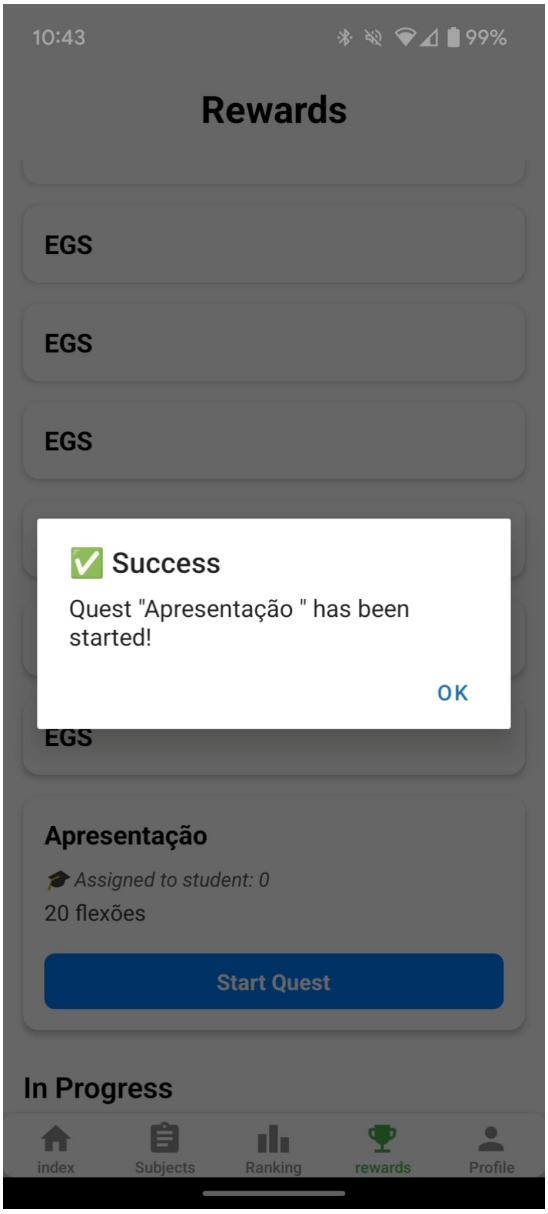

Figure 4: comelar quest

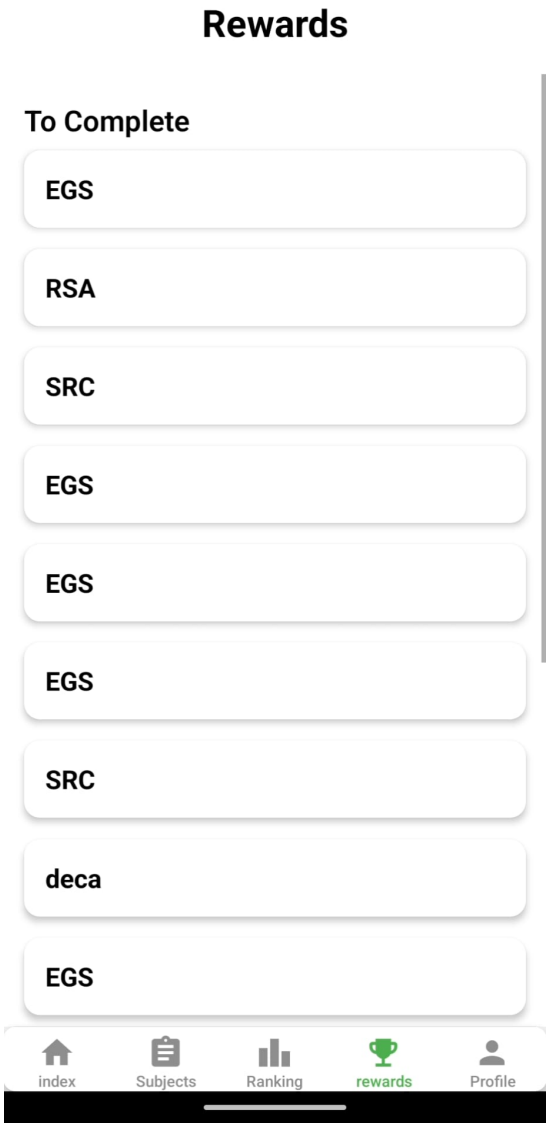

Figure 5: Rewards

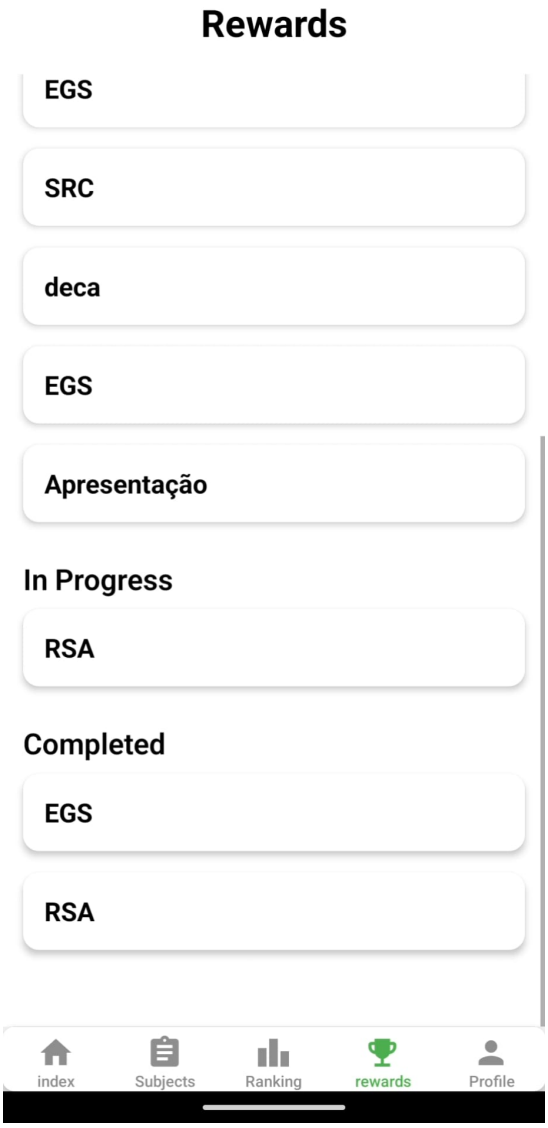

Figure 6: Rewards

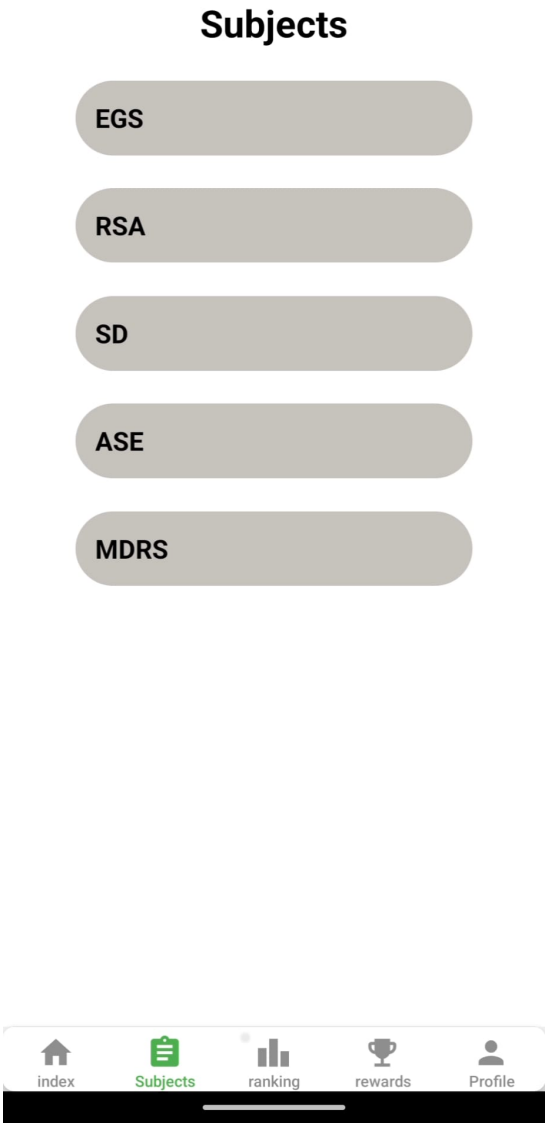

Figure 7: Login

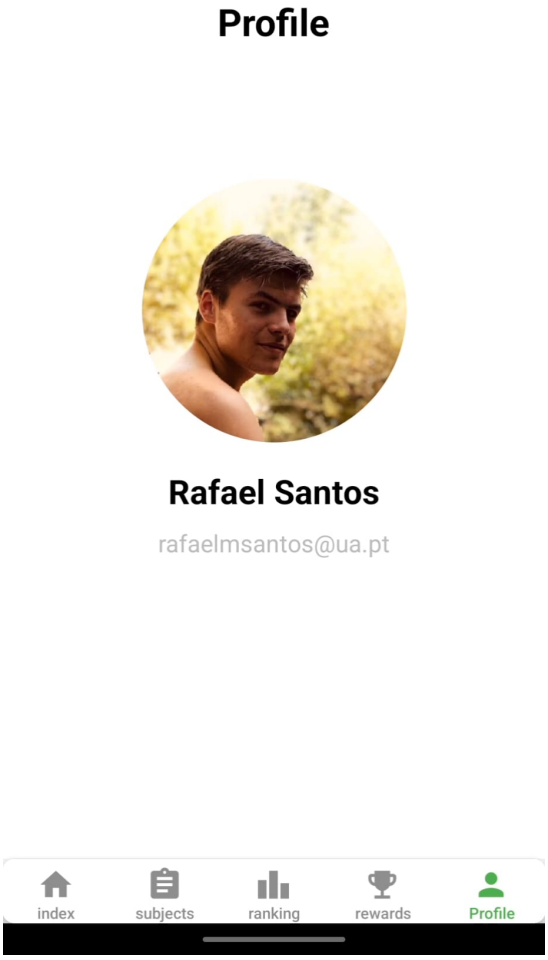

Figure 8: Profile

7.2 App Professor

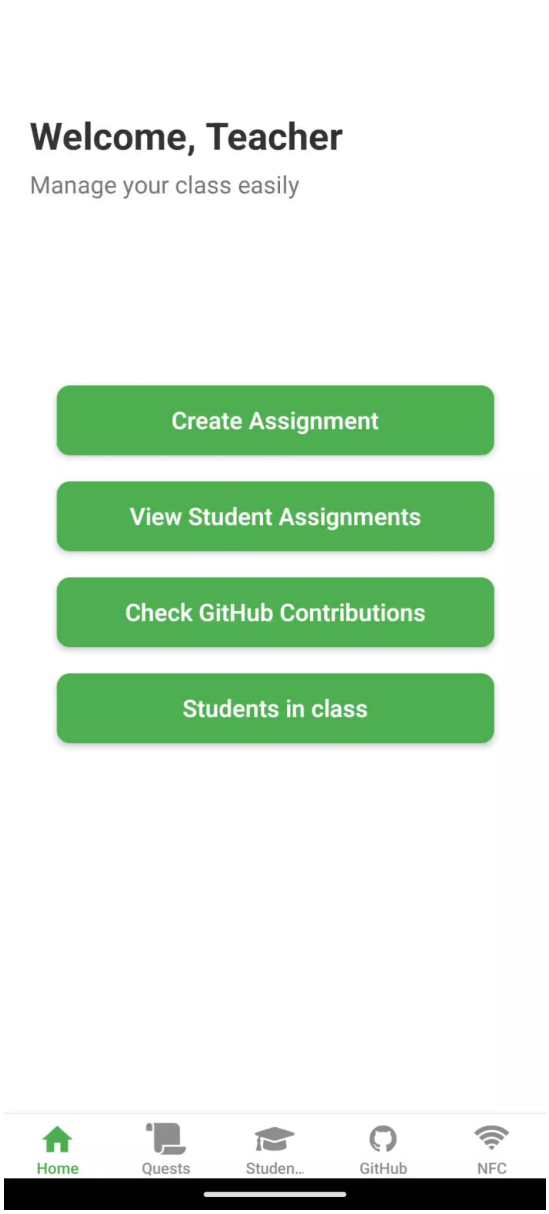

Figure 9: Index

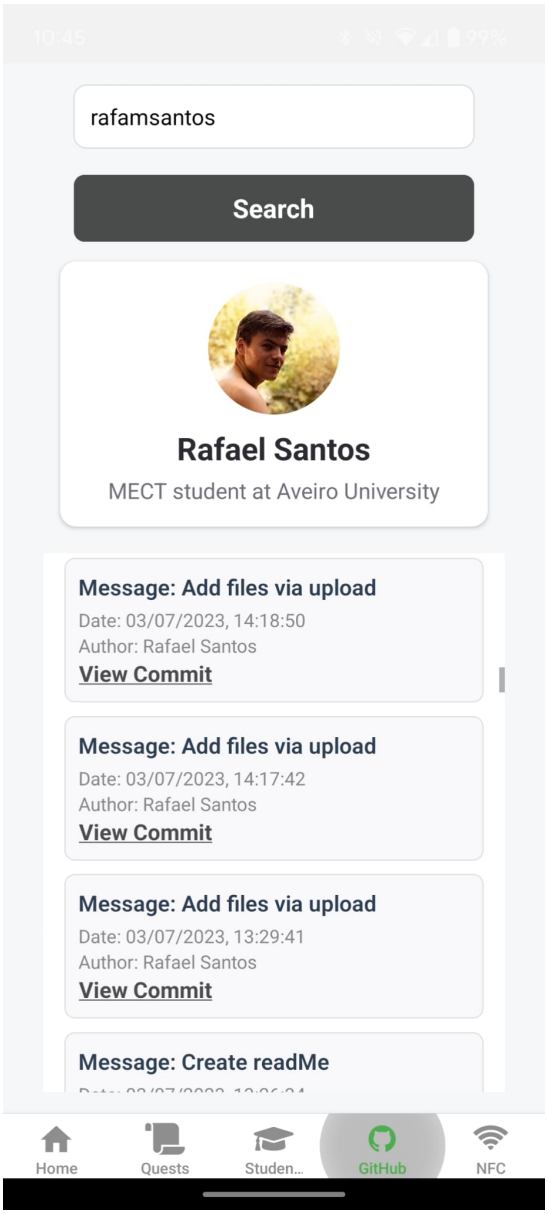

Figure 10: Github

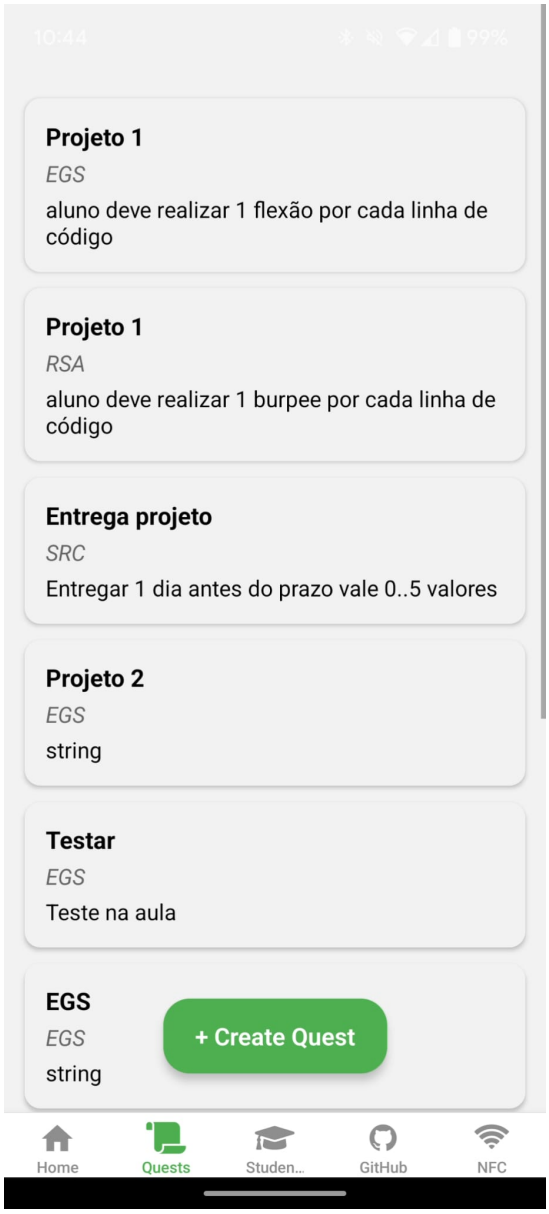

Figure 11: Tarefas criadas

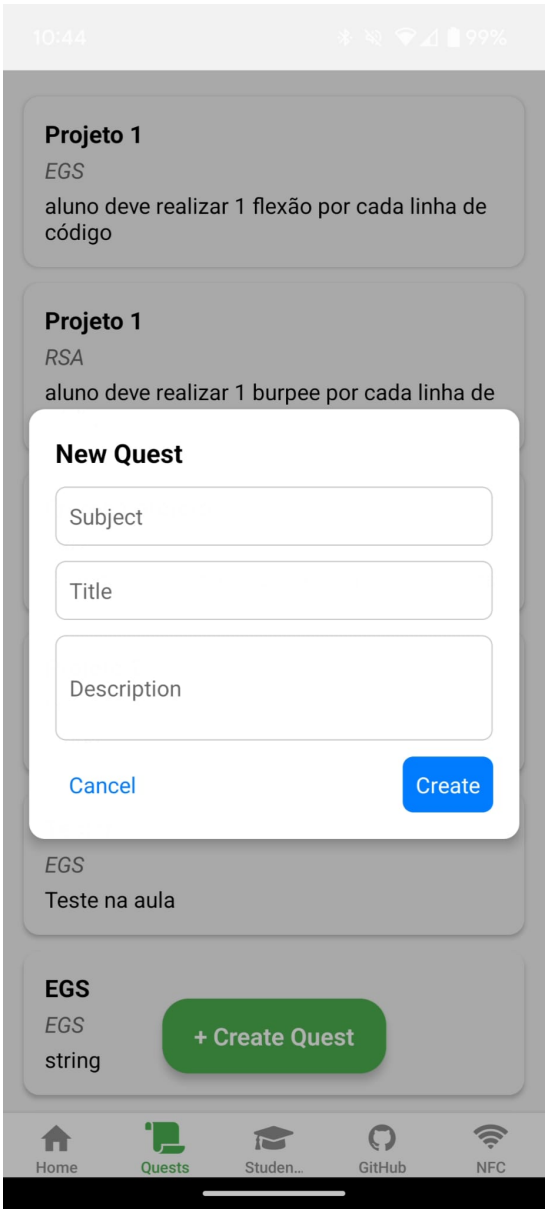

Figure 12: Criação de tarefas

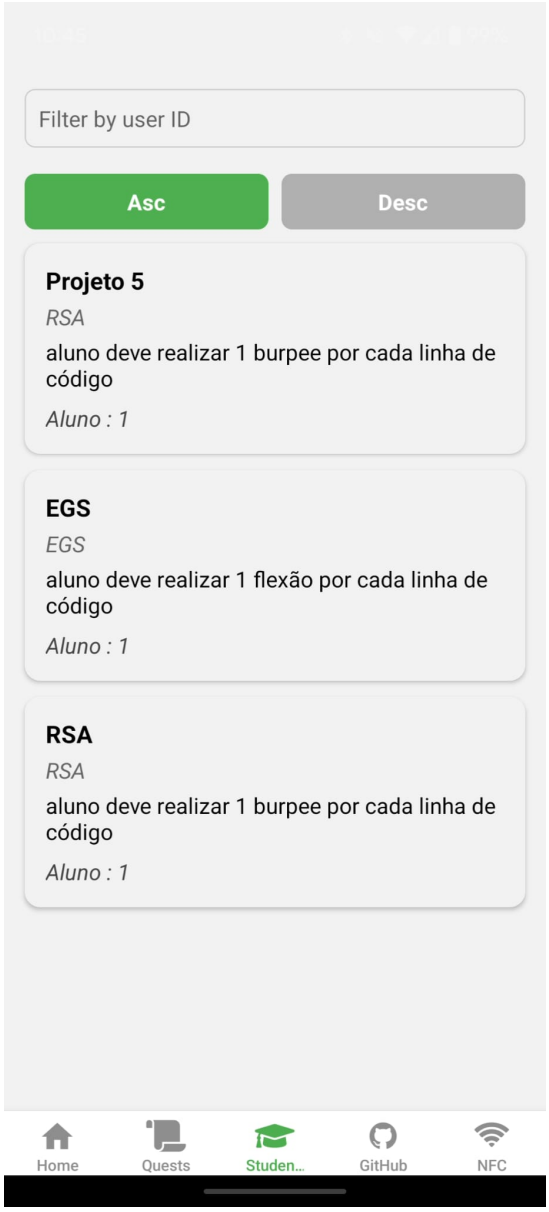

Figure 13: Tarefas aceites pelos alunos

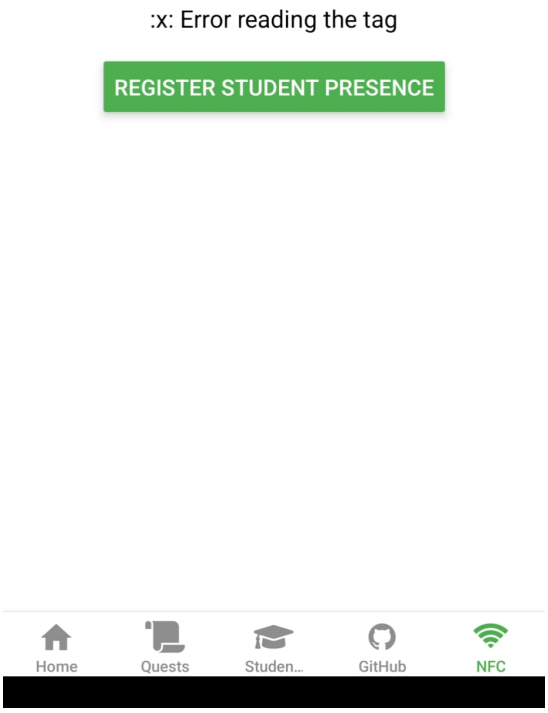

Figure 14: Registar presenças