Instituto Politécnico do Cávado e Ave

Mestrado em Engenharia Informática

Arquitetura e Integração de Sistemas

Carlos Baixo, 16949
Carlos Beiramar, 29988
João Silva, 16951

December 20, 2023

Índice

Lista de figuras

1 Introdução

Neste relatório estará presente uma arquitetura que terá como objetivo viabilizar a implementação de uma aplicação inovadora voltada para a promoção de adoção de animais. Com foco primordial na simplicidade e eficiência, a aplicação visa estabelecer uma ponte digital entre pessoas que desejam adotar animais e animais que aguardam por um lar.

A conceção desta aplicação é motivada pela necessidade de diminuir o número de animais abandonados, oferecendo uma panóplia de funcionalidades que irão contribuir para tornar mais fácil todo o processo de adoção.

Ao proporcionar diversas funcionalidades a aplicação destaca-se pela sua capacidade de conectar os utilizadores a animais disponíveis para adoção, facilitando o contacto direto com abrigos, associações ou até mesmo anúncios particulares.

A simplicidade aliada à eficiência é o ponto fulcral desta arquitetura, e a estrutura que irá ser descrita neste relatório tem o intuito de garantir uma implementação bem sucedida desta aplicação.

1.1 Apresentação do problema

Com a realização deste trabalho prático, inserido na Unidade Curricular de Arquitetura e Integração de Sistemas, é pretendido que seja adquirida experiência no planeamento da arquitetura e respetiva implementação de aplicações distribuídas, bem como na utilização da tecnologia de *webservices* para a integração de vários serviços, seguindo um modelo arquitetural baseado em *microservices*.

De forma a alcançar estes objetivos, este trabalho prático está dividido em várias fases:

1. **Identificar um problema**: a primeira etapa envolve a escolha ou conceção de um problema, real ou imaginário, que deve proporcionar uma base sólida para o projeto;

2. **Análise ao problema**: após a identificação do problema, é necessário realizar uma análise detalhada para compreender a sua natureza, os atores envolvidos e os requisitos necessários para a solução;

3. **Elaborar o processo de negócio associado**: Com base na análise do problema, é necessário elaborar o problema de negócio associado, que servirá como referência para a forma como os vários atores (ou utilizadores) interagem com os vários serviços e como as funcionalidades são executadas;

4. **Desenho do modelo arquitetural baseado em *microservices***: A arquitetura é um elemento crítico deste projeto. Com base na análise de requisitos e no processo de negócio, é necessário desenhar um modelo arquitetural de *microservices* que descreva como os componentes do sistema se encaixam e como a comunicação entre eles é estabelecida.

5. **Implementar a solução**: Utilizando tecnologias como APIs, *message brokers* e *web services* e seguindo o modelo arquitetural previamente concebido, a solução será implementada;

6. **Documentação da API**: Ao longo da implementação, será produzida uma documentação da API dos serviços, seguindo o *standard* OpenAPI, o que garantirá que terceiros possam utilizar de forma eficaz a solução desenvolvida.

1.2 Objetivos

O presente trabalho tem como principal objetivo implementar uma arquitetura inovadora com o intuito de criar uma aplicação que irá promover a adoção dos animais. De realçar que a prioridade será desenvolver algo simples e eficaz.
A motivação para o desenvolvimento desta tecnologia surge da necessidade de reduzir o número de animais abandonados, oferecendo uma panóplia de funcionalidade que irão facilitar todo o processo de adoção. Uma das funcionalidades que se irá destacar será a facilidade com que o utilizar conseguirá entrar em contacto direto com abrigos, associações e até mesmo anúncios particulares.
A arquitetura proposta busca a simplicidade aliada à eficiência como seu ponto focal. A estrutura descrita neste relatório visa garantir uma implementação bem-sucedida da aplicação, proporcionando uma experiência fluida e intuitiva para os usuários.
No âmbito do trabalho prático, inserido na Unidade Curricular de Arquitetura e Integração de Sistemas, procura-se adquirir experiência no planeamento e implementação de arquiteturas distribuídas. Além disso, pretende-se explorar o uso da tecnologia de *web services* para a integração de diversos serviços, seguindo um modelo arquitetural baseado em *microservices*.
O objetivo final é criar uma aplicação eficaz, intuitiva e tecnologicamente robusta, contribuindo assim para a redução do número de animais abandonados e promovendo a adoção responsável.

2 Requisitos e Análise

2.1 Levantamento de requisitos

O levantamento de requisitos permite entender quais são as necessidades dos utilizadores e as funcionalidades essenciais para o sucesso da plataforma.

- **Autenticação:** O sistema deverá fornecer um mecanismo seguro de autenticação para garantir um acesso controlado aos utilizadores. Será possível realizar o registo de novos utilizadores, incluindo um processo de verificação por email para confirmação da conta.
- **Gestão de utilizadores:** Os utilizadores poderão criar e atualizar os seus perfis. Irá ser possível definir preferências tais como, tipo de animal desejado, localização preferida,...
- **Animais disponíveis:** A aplicação deverá listar todos os animais disponíveis para adoção e os utilizadores terão a capacidade de filtrar os animais com base em algumas características.
- **Associação de animais:** Os animais que estiverem presentes na aplicação, estão associados a associações, abrigos ou anúncios particulares.

2.2 Análise de requisitos

A análise dos requisitos para o desenvolvimento da aplicação revela uma visão abrangente e detalhada das funcionalidades essenciais que irão moldar a arquitetura e o desenvolvimento da aplicação.
Esta análise para além de delinear as necessidades do sistema, irá estabelecer as bases para uma arquitetura flexível, segura e centrada no utilizador. O desafio será a implementação eficaz de todos estes requisitos garantindo que a aplicação corresponde às expectativas técnicas.

3 Arquitetura dos Microsserviços

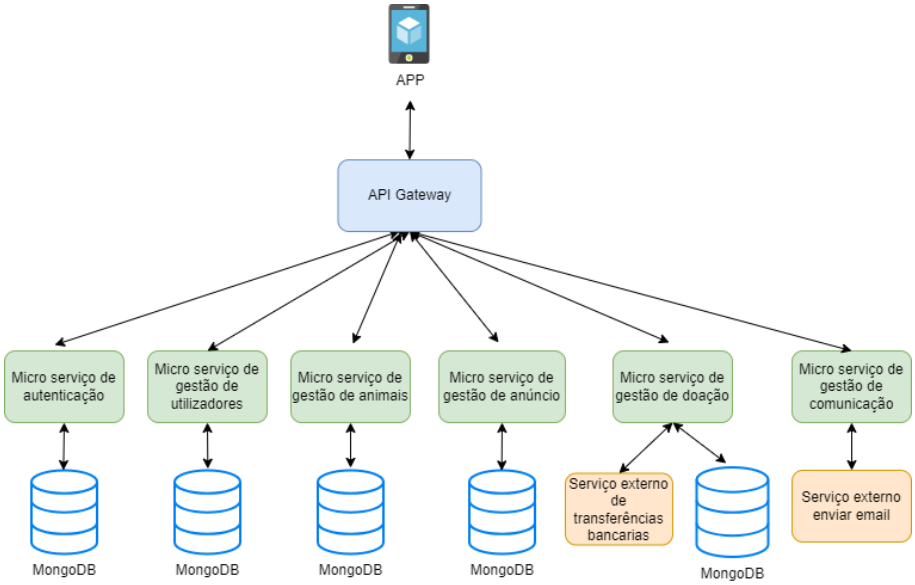

Figure 1: Mapa de arquitetura dos microsserviços

O sistema proposto irá ser estruturado em cinco principais microserviços, cada um responsável por uma funcionalidade específica. Os microsserviços escolhidos são:

- **API Gateway**: Uma API Gateway é um componente necessário para uma arquitetura com sistemas distribuídos. A sua principal função é servir como um ponto único de entrada para todas as solicitações da API, direcionando-as para os respetivos microsserviços.
- **Autenticação**: Este microserviço é responsável pela autenticação e autorização de utilizadores na aplicação. Garante a segurança aos utilizadores pois irá guarda as *passwords* de acordo com o método de encriptação *bcrypt*.
  - **bcrypt:** é um algoritmo de*hashing* que foi criado com o intuito de armazenar *password* de forma segura. Foi desenvolvido para ser lento e resistente a ataques de força bruta e é uma escolha muito utilizada para a proteção de *passwords*.
- **Gestão de Utilizadores:** Os utilizadores poderão criar, visualizar, atualizar e eliminar os seus perfis. Será permitido também aos utilizadores adicionarem algumas informações pessoais de forma a contribuir para uma melhor e eficaz experiência da aplicação Os utilizadores poderão adicionar informações tais como a sua localidade e a sua preferência em relação a algumas características comuns presentes nos animais.
- **Gestão de Animais:** A aplicação permitirá listar todos os animais presentes nas associações, abrigos e até anúncios particulares. Aqui irá ser permitido aos utilizadores aplicar filtros específicos na lista dos animais.

- **Gestão de Anúncio:** A aplicação permitirá a criação de anúncios e a listagem dos mesmos. Aqui irá ser permitido aos utilizadores aplicar filtros específicos na lista dos anúncios.
- **Gestão de Doação:** A aplicação permitirá aos utilizadores realizar doações a associações e abrigos.
- **Gestão de Comunicação:** A aplicação permitirá a comunicação entre o utilizador e associações, abrigos e até utilizadores de anúncios particulares. Numa fase inicial, esta comunicação poderá ser estabelecida através dos contactos fornecidos por cada utilizador no registo da sua conta. Numa fase mais avançada do projeto, será implementado um *live chat* para facilitar a comunicação entre os utilizadores.

4 Serviços externos

A integração de serviços externos é uma estratégia necessária para implementar algumas das funcionalidades que consideramos essenciais na nossa aplicação. Com a sua utilização, pretendemos enriquecer a experiência final do utilizador e manter um nível elevado de desempenho

- **Gmail API**: Este tipo de serviço será utilizado para estabelecer comunicação com clientes. Ao utilizar este serviço, podemos implementar facilmente o envio de e-mails aos utilizadores, como no processo de reposição de palavra-passe, envio de notificações e até *newsletters*. Além disso, este serviço oferece um elevado serviço de segurança e autenticação, garantindo a proteção de dados dos clientes durante o processo de envio de e-mails.
- **Google Maps API**: Com a integração dos serviços de geolocalização da Google na nossa API para mostrar a localização de centros de adoções de animais, assim como aumentar a possibilidade de adoção de animais pelos utilizadores, na medida em que permitirá apresentar melhores resultados, filtrados tendo em conta a localização do próprio utilizador.

5 Implementação

Neste capítulo irão ser detalhados todos os passos que foram utilizados para o desenvolvimento deste projeto, projeto este que abrange seis microsserviços e uma *gateway* que facilita a comunicação entra cada um dos microsserviços.

5.1 Gateway

Neste projeto, a *gateway* é vista como uma *gateway* de comunicação que desempenha um papel fundamental para estabelecer e otimizar a comunicação entre os diversos microsserviços que estão presentes na aplicação. O principal objetivo será então, orientar os pedidos dos utilizadores para os microsserviços correspondentes.
A configuração da *gateway* utiliza o módulo **http-proxy-middleware** para criar *proxies* e direccionar os pedidos de forma inteligente. Cada rota é definida com algumas propriedades e, dentro dessas, é possível encontrar o **URL** da rota e, o **URL** para onde essa mesma rota será direcionada.
A porta atribuída à *gateway* foi a porta **3000**

5.2 Microsserviço de autenticação

Este microserviço desempenha um papel crucial na gestão completa da autenticação na aplicação. É encarregado do registo, login, recuperação de palavra-passe, entre outros pontos essenciais para a utilização da nossa aplicação.

5.2.1 Endpoints

- **signup** - Este endpoint possibilita a inscrição de um utilizador. Requer a receção de um corpo na chamada HTTP contendo o nome de utilizador, palavra-passe e endereço de email (não duplicado).
- **signout** - Este endPoint destrói o token de utilizador.
- **login** - Este endpoint possibilita a autenticação de um utilizador. Requer um corpo contendo o nome de utilizador e a palavra-passe, devolvendo um token de autenticação temporário (válido por 48 horas).
- **requestPasswordReset** - A função deste endpoint é solicitar a redefinição da palavra-passe. Ele envia um email contendo o token de reset para o endereço de email fornecido no corpo da chamada.
- **confirmPasswordReset** - Este endpoint tem como finalidade a modificação da palavra-passe. Recebe o token previamente enviado para o email do utilizador, juntamente com o nome de utilizador e a nova palavra-passe.
- **checkToken** - Este endpoint tem como objetivo verificar se o token ainda está ativo.
- **isAdmin** - Este endpoint responde se o utilizador possui ou não permissões de administrador.
- **getUserEmail** - Este endpoint responde com o email do utilizador, ele recebe o nome do utilizador.
- **getRole** - Este endPoint responde com a função (role) do utilizador que está autenticado.

5.2.2 Modelo para a base de dados

Este microserviço possui dois modelos de dados, um para o utilizador em si e outro para a sua função (role):

```
const mongoose = require("mongoose");
const messages = require("../assets/i18n/validationErrors");
const i18n = require("../services/i18n/translationService");
const Schema = mongoose.Schema;

const UserSchema = new mongoose.Schema(
  {
    username: { type: String, required: true, unique: true },
    password: { type: String, required: true, unique: true },
    email: {
      type: String,
      required: true,
      unique: true,
      trim: true,
      lowercase: true,
      match: [
        /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/,
        i18n.__(messages.invalidEmail),
      ],
    },
    created_at: { type: Date, default: Date.now },
    resetPasswordToken: { type: String },
    resetPasswordExpires: { type: Date },
    role: { type: Schema.Types.ObjectId, ref: "Role" },
  },
  { collection: "users" }
);

module.exports = mongoose.model("User", UserSchema);
```

```
const mongoose = require("mongoose");
const RoleSchema = new mongoose.Schema(
  {
    name: { type: String, required: true, unique: true },
    description: { type: String, required: false, unique: false },
  },
  { collection: "roles" }
);

module.exports = mongoose.model("Role", RoleSchema);
```

A porta utilizada para este microsserviço foi a porta **3001**.

5.3 Microsserviço de gestão de utilizadores

Este microsserviço é utilizado para gerir todos os utilizadores presentes na aplicação. É utilizada uma base de dados em *MongoDB* para guardar toda a informação referente a cada um dos utilizadores.

5.3.1 Endpoints

- **getAllUsers** - Este endpoint irá retornar todos os utilizadores presentes na aplicação.
- **createUserByID** - Permite criar um utilizador através do seu ID, no entanto só os utilizadores considerados **admin** é que têm permissão para executar este endpoint.
- **getUserByID** - Caso o utilizador seja admin, permite obter a informação de qualquer um dos utilizadores existentes. Caso o user nao seja admin só consegue ter acesso à sua própria informação.
- **updateUser** - Permite a qualquer utilizador atualizar a sua informação.
- **deleteUser** - permite elimintar um utilizador presente na aplicação. Os utilizadores considerados admins podem eliminar qualquer utilizador e o utilizadores comuns só podem eliminar o seu próprio perfil. Quando um utilizador é eliminado deste microsservico, irá também ser eliminado do microsserviço de autenticação.
- **getUserIDByEmail** - Permite obter o ID de um utilizador através do seu email.

5.3.2 Modelo para a base de dados

O modelo utilizado para a implementação deste microsserviço foi:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userSchema = new Schema(
  {
    name: String,
    email: {
      type: String,
      unique: true,
    },
    birth_date: {
      type: Date,
      default:  null,
    },
    phoneNumber: {
      type: String,
      default: null,
    },
    updatedAt: {
      type: Date,
      default: Date.now,
    },
    createdAt: {
      type: Date,
      default: Date.now,
    },
    city: {
      type: String,
      default: null,
    },
  },
  {
    collection: "users",
  }
);

module.exports = mongoose.model("Users", userSchema);
```

A porta utilizada para este microsserviço foi a porta **3002**.

5.4 Microsserviço de gestão de animais

Este microsserviço foi implementado para garantir uma gestão prática e intuitiva de todos os animais que estão presentes na aplicação. Cada animal tem que estar associado a um *shelter* e/ou um *user*.

5.4.1 Endpoints

- **getAllAnimals** - *Endpoint* para obter todos os animais presentes na base de dados.
- **getAnimalsByShelterId** - *Endpoint* para obter todos os animais que estão associados a um *helter*.
- **createAnimal** - *Endpoint* para inserir um novo animal na aplicação.
- **getAnimalById** - *Endpoint* para obter todas as informções de um animal específico.
- **updateAnimal** - *Endpoint* para atualizar as informações de um animal em particular.
- **deleteAnimal** - *Endpoint* para eliminar um animal específico da base de dados.

5.4.2 Modelo para a base de dados

O modelo utilizado para a implementação deste microsserviço foi:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const GenderEnum = Object.freeze({
    MALE: 'male',
    FEMALE: 'female',
    OTHER: 'other'
  });


const animalSchema = new Schema({
    name: String,
    birth_date: Date,
    description: String,
    gender:{
        type: String,
        enum: Object.values(GenderEnum)
    },
    user_id: {
        type: Schema.Types.ObjectId,
        ref: "User",
        required: true,
    },
    shelter_id: {
      type: Schema.Types.ObjectId,
      ref: "Shelter",
      required: true,
    },
    size: String,
    animal_type: String,
    breed: String,
    createdAt: {
        type: Date,
        default: Date.now,
    },
```

```
});

module.exports = mongoose.model("Animal", animalSchema);
```

A porta utilizada para este microsserviço foi a porta **3003**.

5.5 Microsserviço de gestão de anúnicos

Este microserviço foi implementado para permitir gerir todos os anúncios particulares de adoções associados a um utilizador.

5.5.1 Endpoints

- **getAllAdvertisements** - *Endpoint* utilizado para obter todos os anúncios existentes na base de dados.
- **createAdvertisement** - *Endpoint* utilizado para criar um anúncio de adoção.
- **getAdvertisementByID** - *Endpoint* criado para obter o ID de um anúncio específico.
- **updateAdvertisement** - *Endpoint* criado para permitir atualizador os dados de um anúncio.
- **deleteAdvertisement** - *Endpoint* para eliminar um anúncio da aplicação.
- **getAdvertisementByUserID** - *Endpoint* utilizado para encontrar anúncios de um User específico.

5.5.2 Modelo para a base de dados

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const advertisementSchema = new Schema(
  {
    description: String,
    city: String,
    rating_avg: Number,
    createdAt: {
      type: Date,
      default: Date.now,
    },
    user_id: {
      type: Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
  },
  {
    collection: "advertisements",
  }
);

module.exports = mongoose.model("Advertisement", advertisementSchema);
```

A porta utilizada para este microserviço foi a porta: **3004**.

5.6 Microserviço de gestão de doações

Este microservico foi implementado para gerir todas as doações que poderão ser feitas através da aplicação. Cada doação tem que estar associada a um user, que será o doador, e a um shelter, que será a entidade que irá receber a doação.

5.6.1 Endpoints

Os *endpoints* implementados para este microserviço foram:

- **getAllDonations** - Permite obter todas as doações feitas através da aplicação
- **insertDonation** - Permite a qualquer utilizador fazer um doação para um shelter específico.
- **getDonationByID** - *Endpoint* utilizado para obter um doação específica.
- **deleteDonation** - *Endpoint* que permite aos administradores eliminarem doações.
- **getDonationsByUserID** - *Enpoint* que permite obter todas as doações feitas por um utilizador específico.
- **getDonationsByShelterID** - *Enpoint* que permite obter todas as doações feitas para um shelter específico.

5.6.2 Modelo para a base de dados

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const donationsSchema = new Schema(
  {
    amount: Number,
    createdAt: {
      type: Date,
      default: Date.now,
    },
    user_id: {
      type: Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
    shelter_id: {
      type: Schema.Types.ObjectId,
      ref: "Shelter",
      required: true,
    }
  },
  {
    collection: "donations",
  }
);

module.exports = mongoose.model("Donations", donationsSchema);
```

A porta utilizada para este microserviço foi a porta **3005**.

5.7 Microserviço de gestão de shelters

Este microserviço permite gerir todos os *shelters* presentes na aplicação. Nos *shelters* será possível encontrar a maior parte dos animais que poderão ser adotados. Apesar de ser possível para todos os utilizadores criar um *shelter*, após a criação dos mesmos, os administradores terão que aprovar o shelter para este se tornar visível para a aplicação.

5.7.1 Endpoints

Os *endpoints* utilizados para implementação deste microserviço foram:

- **getAllShelters** - *Enpoint* que permite obter todos os *shelters* presentes na aplicação.
- **getNoneVerifiedShelters** - *Enpoint* para obter todos os *shelters* que ainda não foram aprovados pelos administradores.
- **verifyShelterByID** - *Endpoint* que permite aos administradores aprovar um *shelter* específico.
- **createShelter** - *Enpoint* que permite aos utilizadores criar um *shelter*. Este *shelter* irá ficar associado ao utilizador que o criou.
- **getShelterByID** -*Endpoint* que permite procurar um *shelter* através de um ID específico.
- **updateShelter** - Atualiza as informações de um *shelter* específico.
- **deleteShelter** - *Enpo8int* que permite aos administradores eliminar um *shelter* específico.

5.7.2 Modelo para a base de dados

O modelo de base dados utilizado para este microserivço foi o seguinte:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const shelterSchema = new Schema({
    name: String,
    description: String,
    email: String,
    birth_date: Date,
    phone_number: String,
    country: String,
    city: String,
    postal_code: String,
    address: String,
    isVerified: {
        type: Boolean,
        default: false,
    },
    createdAt: {
        type: Date,
        default: Date.now,
    },
    user_id: {
        type: Schema.Types.ObjectId,
        ref: "User",
        required: true,
    },
});
```

```
28
29 module.exports = mongoose.model("Shelter", shelterSchema);
```

A porta utilizada para este microserviço foi a porta **3006**.

5.8 Swagger

A documentação da API desempenha um papel crucial no desenvolvimento e manutenção dos microsserviços. Assim, para desenvolver esta documentação foi utilizado um framework para simplificar o processo de design, **Swagger**.
De forma a adaptar o **Swagger** na arquitetura de microsserviços desenvolvida para este projeto, foram utilizados dois pacotes fundamentais, **swagger-ui-express** para renderizar o **Swagger UI** e **swagger-jsdoc** para gerar o **Swagger** com base nos comentários **JSDoc** presentes no código.

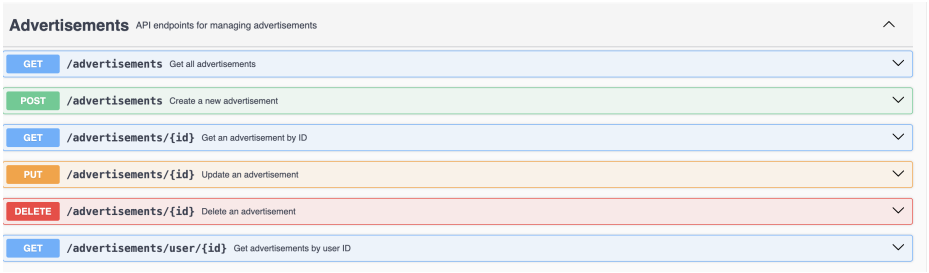

Figure 2: Swagger gerado para o microsserviço de anúncios

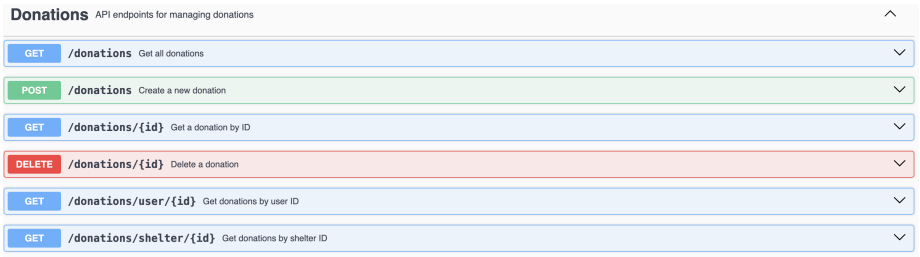

Figure 3: Swagger gerado para o microsserviço de doações

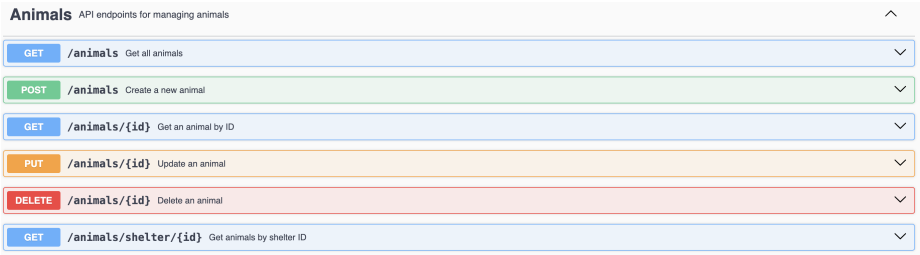

Figure 4: Swagger gerado para o microsserviço de animais

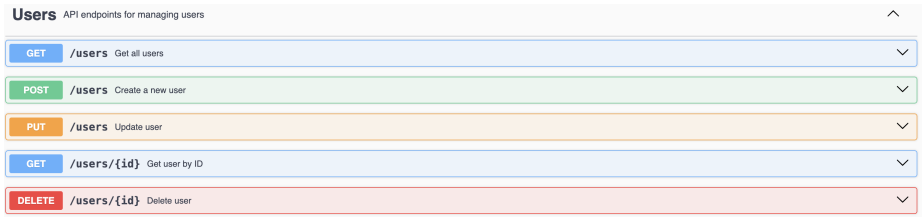

Figure 5: Swagger gerado para o microsserviço de utilizadores

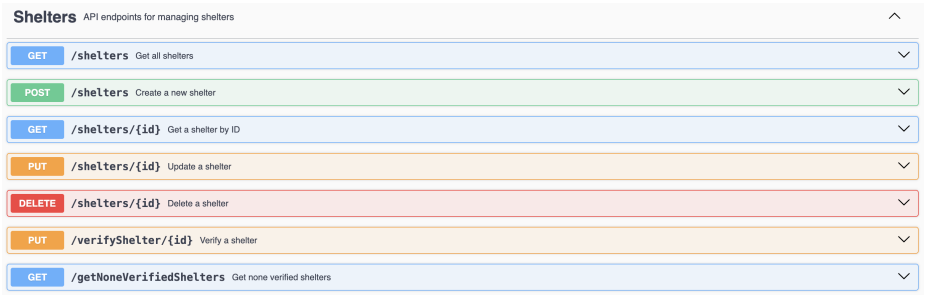

Figure 6: Swagger gerado para o microsserviço de abrigos

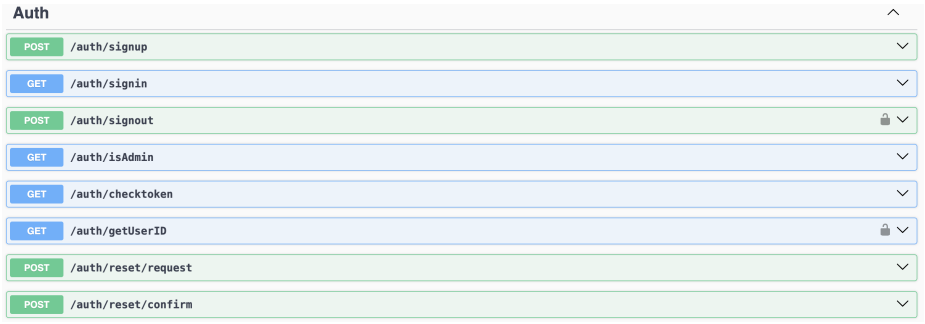

Figure 7: Swagger gerado para o microsserviço de autenticação

A porta escolhida para o swagger foi a porta **3007**.

5.9 Testes implementados

De forma a garantir que toda a aplicação funcione de forma esperada e que a integração de todos os microsserviços seja robusta, foram implementados dois tipos de testes: Testes **unitários** e Testes no **Postman**.

5.9.1 Testes unitários

Os testes unitários foram criados de foram a ser possível obter uma validação do código. Assim, no contexto do presente projeto, estes testes servem para garantir que cada *endpoint* de cada microsserviço tenha o comportamento esperado. Para além de tudo isso, a implementação destes testes também facilitou a deteção de erros e a correção dos mesmos. Os testes unitários presentes neste projeto são os seguintes:

- Microsserviço dos anúncios
  - createAdvertisement.test.js
  - deleteAdvertisement.test.js
  - getAdvertisementbyID.test.js
  - getAdvertisementbyUserID.test.js
  - getAllAdvertisements.test.js
  - updateAdvertisement.test.js
- Microsserviço das doações
  - deleteDonation.test.js
  - getAllDonations.test.js
  - getDonationByID.test.js
  - getDonationsByShelterID.test.js
  - getDonationsByUserID.test.js
  - insertDonation.test.js
- Microsserviço dos animais
  - createAnimal.test.js
  - deleteAnimal.test.js
  - getAllAnimals.test.js
  - getAnimalByID.test.js
  - getAnimalsByShelterID.test.js
  - updateAnimal.test.js
- Microsserviço dos utilizadores
  - createUserByID.test.js
  - deleteUser.test.js
  - getAllUsers.test.js
  - getUserByID.test.js

  - updateUser.test.js

- Microsserviço dos abrigos
  - createShelter.test.js
  - getAllShelters.test.js
  - getShelterById.test.js

5.9.2 Testes no Postman

Os testes de integração no Postman têm como principal objetivo garantir a comunicação entre os diferentes microsserviços. Foi utilizado o Postam para realizar testes automatizado que simulam as interações entre os diferentes microsserviços de forma a garantir uma comunicação eficiente. Estes testes envolvem simulações de HTTP entre os microsserviços. Para cada um destes, foi criada uma *collection* que irá conter os respetivos endpoints.
Assim, será disponibilizado um ficheiro gerado automaticamente pelo Postman contendo tudo que é necessário para testar a aplicação, esse ficheiro estará presente no projeto como **wePet.postman_collection.json**.

5.10 Microsserviços e Docker

Cada um dos microsserviços utilizado é encapsulado no seu próprio ambiente isolado que irá garantir a consistência e a portabilidade em diferentes ambientes de execução. Assim, cada um dos microsserviços possui o seu próprio Dockerfile que define as instruções para construir a imagem do contêiner. Estes conteiners garantem que cada microsserviço seja executado independentemente.
As Dockerfiles referidas anteriormente incluem a configuração do ambiente, a instalação das dependências e a definição de como o microsserviço será executado. Exemplo de uma Dockerfile utilizada:

```
FROM node:18

# Create the working directory inside the contanier
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the app code to the working directory
COPY . .

# Choose the port for this microservice
EXPOSE 3003

# Set environment variables
ENV GATEWAY_PORT=3000
ENV ANIMAL_PORT=3003
ENV SECRET_KEY=wepet2023
ENV ATLAS_URL=mongodb://host.docker.internal:27017/animals?authSource=admin
ENV JWT_SECRET=wepet2023

# Start the application
CMD [ "npm", "start" ]
```

5.10.1 Docker Compose

O Docker Compose é uma ferramenta valiosa para coordernar a execução simultânea de todos estes conteiners, descrevendo os serviços para cada um dos microsserviços implementados para a aplicação e, para além disso, foi adicionado um serviço para a base de dados MongoDB e um serviço dedicado para a documentação Swagger.
Assim, é possível criar todo o ambiente necessário para executar a aplicação e todos os serviços dos quais a mesma depende para um funcionamento eficiente.

5.11 Processos de negocio

Esta aplicação tem vários processos de negócio visto que cada microsservico tem o sem próprio comportamento, no entanto, nas seguintes imagens é apresentado o processo de negócio para fazer o login na aplicação e também para fazer uma doação.

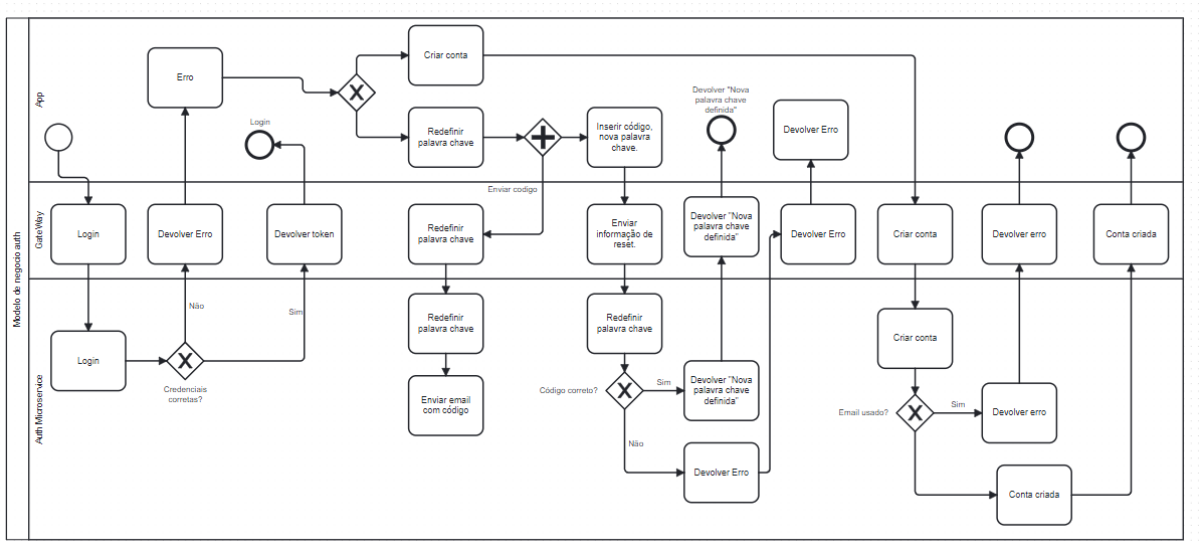

Figure 8: Processo de negocio para entrar na aplicação

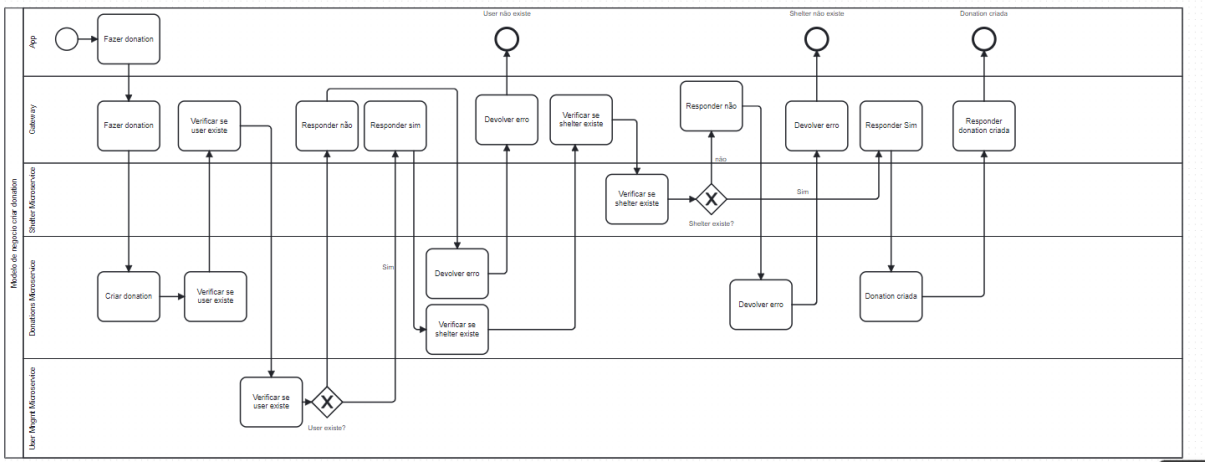

Figure 9: Processo de negocio para criar fazer uma donation

Em suma, o processo de negócio da aplicação é algo não muito complexo o que torna a aplicação muito intuitiva e, o processo de negócio dos restantes microsservicos será muito semelhante ao processo de negócio apresentado anteriormente para fazer uma doação.

6 Conclusão

6.1 Primeira entrega

A arquitetura proposta para além de corresponder às expectativas de simplicidade e eficiência, também incorpora as melhores práticas de segurança e gestão de sistemas distribuídos. À medida que se foi avançando na implementação deste sistema de adoção de animais, é possível perceber que a combinação de microserviços e API Gateway proporcionarão uma aplicação segura e intuitiva, cumprindo a sua missão de facilitar adoções bem sucedidadas e contribuir para o bem estar dos animais.

6.2 Entrega final

Após a conclusão deste trabalho, o grupo sentiu que foi possível expandir conhecimentos e obter uma ideia clara e concreta do que são microsserviços e de que forma é que estes podem comunicar entre si. Permitiu-nos trabalhar com muita tecnologias novas das quais não estávamos muito familiarizados e, achamos que é um excelente trabalho de preparação para o mercado de trabalho. Para trabalho futuro, achamos que haverão pontos que deverão ser melhorados tais como, adicionar mais alguns serviços externos como por exemplo, um serviço externo de pagamento para as doações, fazer deploy dos containers de forma a ter a aplicação a funcionar numa cloud.