DESENVOLVIMENTO DE UMA API PARA SERVIR UMA APLICAÇÃO DE ACHADOS E PERDIDOS

Mateus Farinelli Zardo[1], Fabricio Gustavo Henrique[1]

Faculdade de Tecnologia de Ribeirão Preto (FATEC-RP)

Ribeirão Preto, SP – Brasil

mateus.zardo@fatec.sp.gov.br, fabricio.henrique@fatec.sp.gov.br

Resumo. *A perda de um objeto muitas vezes pode ser frustrante e desencadear muitas emoções, além de causar algumas perdas financeiras e atrasos. Para facilitar o desdobramento desses eventos para a comunidade, um sistema de achados e perdidos possibilita a troca de informações quase que imediatamente entre as partes, permitindo a inserção de informações como local, data e hora do evento, contato direto entre as partes, fotos de itens e outras informações. Aliado a isso o desenvolvimento de uma API pública que utiliza padrão REST e princípios S.O.L.I.D pode ser encarado como uma alternativa viável.*

Abstract. *The loss of an object can often be frustrating and trigger a lot of emotions, in addition to causing some financial loss and delays. To facilitate the deployment of these events for the community, a lost and found system enables the exchange of information almost immediately between the parties, allowing the insertion of information such as the location, date and time of the event, direct contact between the parties, photos of items and other information. Allied to this, the development of a public API that uses REST standards and S.O.L.I.D principles can be seen as a viable alternative.*

1. INTRODUÇÃO

Seja em casa, na escola, ou trabalho, perder um objeto por menor que seja pode trazer resultados emocionais e financeiros indesejados, além do tempo demandado na locomoção até o local onde supostamente o objeto pode ter sido perdido, caso o indivíduo não se dê conta do ocorrido de imediato.

Realizando-se uma rápida busca pela internet ou mesmo nas lojas de aplicativos dos aparelhos smartphone, nota-se uma baixa adesão aos sistemas existentes no caso desta última, ou ainda, sistemas antigos e que dependem de um intermediário e espaços físicos como são os sistemas dos Correios (Nacional), Poupatempo (para o estado de São Paulo) e do metrô municipal de São Paulo.

Observa-se nos sistemas mencionados a problemática do intermediário sendo um órgão estatal, e que não consegue manter os objetos em sua guarda por períodos maiores que 90 dias, sendo estes destinados a entidades cadastradas após o período estipulado, conforme evidenciado por meio de informações divulgadas pelos próprios meios de comunicação destes sistemas.

> Os documentos encontrados são mantidos na agência por 60 dias contados a partir da data em que deram entrada. Após esse período são encaminhados aos órgãos emissores.
> No momento da retirada, você deve comprovar que é titular do documento e pagar a taxa. (CORREIOS, 2018)
>
> Os objetos ficam guardados à disposição dos interessados por 60 dias. Os itens não procurados são encaminhados para o Fundo Social de S.O.L.I.Dariedade do Estado de São Paulo, e os documentos para os órgãos emissores. (SÃO PAULO, 2019)
>
> Eles ficam por três meses à disposição dos donos. Após esse período, documentos são encaminhados para os órgãos emissores e os objetos são doados para o Fundo Social de São Paulo (FUSSP). (CPTM 2019)

Sendo assim o presente trabalho torna-se uma alternativa viável a esses sistemas uma vez que a aplicação desenvolvida tem capacidade de receber e disponibilizar dados de forma simples e rápida, podendo diminuir a burocracia se comparada ao que os sistemas citados possuem. Entretanto ele é limitado pela iniciativa dos usuários, ou seja, quanto menor o número de usuários menor a chances de o item perdido ser divulgado.

Não obstante o objetivo do presente trabalho é o desenvolvimento de uma aplicação baseada em padrões estruturados e com aplicação dos princípios que proporcionam uma maior qualidade ao sistema.

Tais conceitos e tecnologias, serão explicados adiante na seção 2, e a implementação deste sistema será elucidada na seção 3 do presente trabalho. Por fim os resultados serão apresentados e discutidos respectivamente nas seções 4 e 5.

2. CONCEITOS E TECNOLOGIAS

Como mencionado anteriormente o objetivo deste trabalho é o desenvolvimento de uma aplicação baseada em padrões estruturados e com aplicação dos princípios que proporcionam uma maior qualidade ao sistema.

Possibilitando ainda, que sistemas terceiros possam ser capazes de acessar essa aplicação apenas seguindo seus padrões sem precisar conhecer de fato as suas implementações, desta forma, independente da plataforma, isto é, qualquer sistema que obedeça os padrões desta aplicação podem fazer a manipulação dos dados independente de onde os sistemas estão sendo rodados, seja um sistema WEB como por exemplo um site, ou ainda, um sistema mobile como por exemplo um aplicativo de smartphone, poderão usufruir deste sistema através dos padrões mencionados.

De forma mais técnica o objetivo deste trabalho é o desenvolvimento de uma *API* baseada no padrão *REST* com aplicação de princípios *S.O.L.I.D*. A seguir nas seções 2.1, 2.2 e 2.3 serão explicados cada um dos termos apresentados.

2.1. APPLICATION PROGRAMMING INTERFACE (API)

Uma API como pode ser considerada “um conjunto de definições e protocolos usado no desenvolvimento e na integração de software de aplicações” (RED HAT, 2017). Portanto, uma API permite que produtos e serviços integrem-se a ela, apenas trocando informações, sem necessidade de conhecer suas implementações. De outra forma (talvez mais simples), uma api pode ser considerada como sendo um “contrato”, ou seja, para que uma aplicação possa consumir uma API ela deve seguir os padrões estabelecidos, sem necessidade alguma de saber como são feitas suas implementações, isto é, a aplicação terceira não precisa conhecer a fundo o código da API, apenas precisa saber quais são os padrões esperados pela API e quais são os padrões devolvidos pela mesma.

Ainda segundo a empresa Red Hat as APIs surgiram quase que no mesmo momento em que a computação, onde naquele período as API eram utilizadas normalmente como bibliotecas de sistemas operacionais, sendo em grande maioria APIs locais, que vez ou outra trocavam mensagens entre mainframes, chegando muito próximo ao que se tem hoje por volta dos anos 2000, onde houve a grande ascensão da internet que se conhece atualmente (RED HAT, 2017).

2.2. REPRESENTATIONAL STATE TRANSFER (REST)

REST é a sigla utilizada para denominar a abstração da arquitetura de software sobre um serviço operando em rede com o uso de web services. Este padrão contribui para que os serviços sejam definidos a partir de uma estrutura e fornecidos para consumo de modo independente, ou seja, aplicações que utilizem o protocolo HTTP (Hypertext Transfer Protocol) podem acessar esses serviços independentemente da sua plataforma (FIELDING, 2000).

Para que se possa realizar ações sobre os recursos disponibilizados, utiliza-se os métodos disponíveis no protocolo HTTP anteriormente mencionado, sendo eles:

- **POST:** utilizado para criar o estado do recurso.
- **GET:** utilizado para recuperar o estado do recurso.
- **PUT/PATCH:** para atualizar o estado do recurso.
- **DELETE:** para deletar o estado do recurso.

Na Figura 2 está a representação da implementação da arquitetura REST.

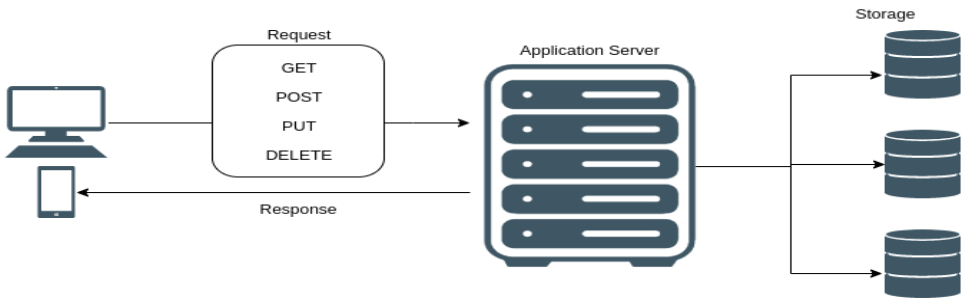

Figura 1. Representação da arquitetura REST
Fonte: (MOODY, M., 2019)

Já os recursos são as informações presentes na API. No caso deste trabalho, pode-se considerar um “item” como sendo um recurso, já que ele é parte da informação do sistema, conforme pode ser visto na figura a seguir, onde obtém-se a listagem de “itens” através de uma requisição HTTP do tipo “GET” na API.

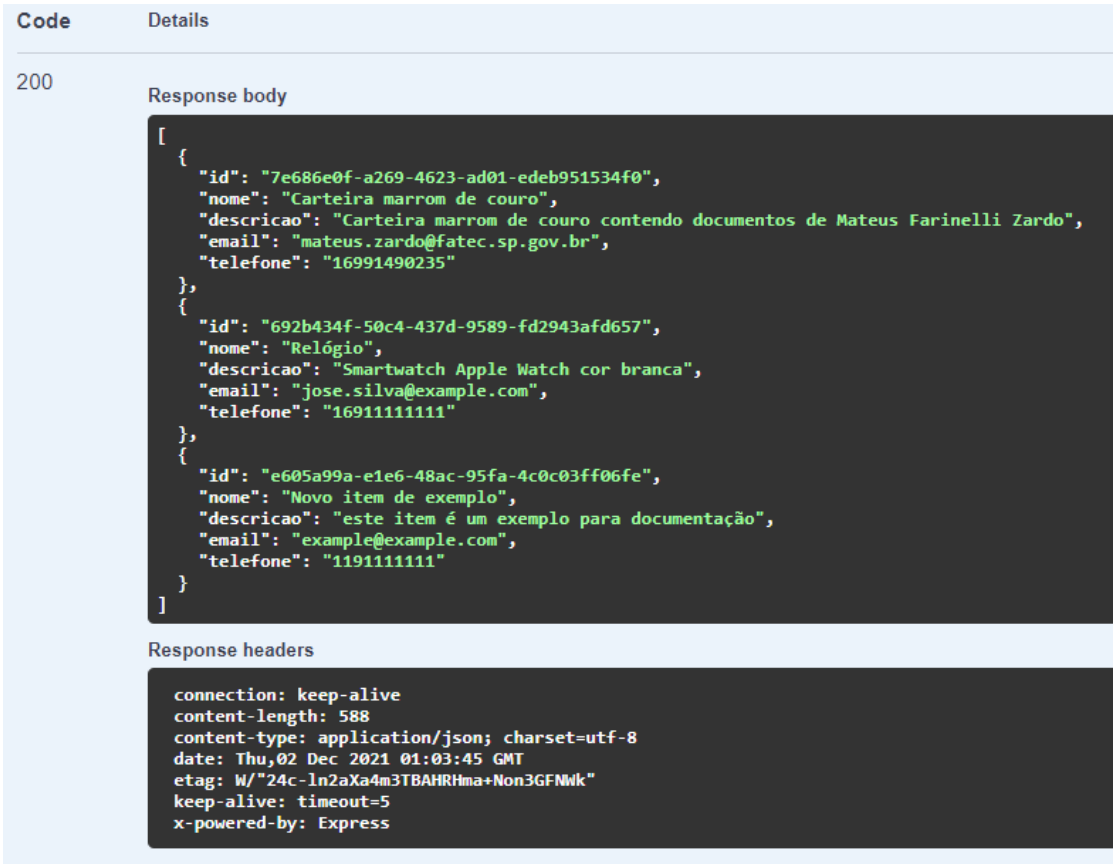

Figura 2. Resposta da API a uma requisição HTTP do tipo GET
Fonte: Autoria própria, 2021.

Para que uma API seja considerada RESTful, ou seja, API que implementa o padrão REST, a mesma deve atender as seis restrições sendo elas (RED HAT, 2017):

- **Client-Server Architecture (Arquitetura Cliente-Servidor):** composta por clientes, servidores e recursos, onde as interações devem ser realizadas através de requisições HTTP.
- **Stateless (Sem estados):** nenhum dado do cliente deve ser mantido no servidor entre as requisições, esses dados devem ser mantidos pelo próprio cliente.
- **Cacheable (Armazenável):** capacidade de manter informações em cache evitando acessos repetidos demasiados aumentando a performance da API.
- **Layered system (Sistemas em camadas):** o cliente não deve fazer uma requisição diretamente ao servidor da aplicação, ele deve antes passar por alguma camada, ou camadas, que fazem a interface com o servidor.

- **Uniform interface (Interfaces uniformes):** esta restrição diz mais respeito ao design dos recursos disponibilizados, e também é composta por quatro itens que modelam as interfaces, sendo:
    - **Resource identification (Identificação do recurso):** cada recurso deve ter uma URI específica e coesa.
    - **Manipulation through representations (Representação do recurso):** é como o recurso será devolvido para o cliente.
    - **Self-descriptive messages (Mensagens autodescritivas):** retorna informações em forma de metadata para o cliente.
    - **Hypermedia (Hipermídia):** retorna todas as informações necessárias para que o cliente consiga navegar sobre determinado recurso da aplicação.

2.3. PRINCÍPIOS S.O.L.I.D

S.O.L.I.D é o acrônimo para designar cinco princípios de desenvolvimento de software aplicados a orientação a objetos. Princípios esses introduzidos por Robert “Uncle Bob” Martin no artigo “Design Principles and Design Patterns” (MARTIN, 2000). Sendo eles:

S - Single Responsibility (Responsabilidade Única): este é o princípio mais simples e talvez o com maior compatibilidade com seu título, nele Uncle Bob defende que uma classe deve apresentar uma única responsabilidade.

Seu uso é justificado pois quanto mais responsabilidades uma classe tem maiores a chances de aparecimento de bugs, já que acaba se tornando extensa e complexa.

O – Open-Closed (Aberto-Fechado): neste princípio o autor define que uma classe deve estar aberta para extensão, mas fechada para alterações.

A justificativa deste princípio se dá pela alteração das classes quanto suas funcionalidades, ou seja, uma classe nunca deve ter sua funcionalidade alterada pois isso pode gerar bugs e comprometer todo o funcionamento do sistema, em vez disso, a classe deve poder estender funcionalidades de outras classes para que ela possa então operar corretamente.

L - Liskov Substitution (Substituição de Liskov): “objetos em um programa devem ser substituíveis por instâncias de seus subtipos sem alterar a corretude desse programa” (MARTIN, 2000).

A implementação desse princípio garante que se uma classe for estendida, a mesma deve herdar o comportamento da classe pai sem perdas, caso contrário a classe herdeira (filha) foi completamente violada e já não atende este princípio.

I - Interface Segregation (Segregação de Interfaces): este princípio diz que os clientes não devem ser forçados a depender de métodos que não usam, ou ainda, é melhor ter interfaces específicas a ter interfaces genéricas.

Portanto é melhor que uma interface genérica seja segregada em interfaces mais específicas, pois assim conseguem realizar apenas as funcionalidades que lhe são necessárias.

D - Dependency Inversion (Inversão de Dependências): “Este princípio visa reduzir a dependência de uma classe de alto nível na classe de baixo nível, introduzindo uma interface.” (UGONNA, 2020).

Ou seja, uma classe de alto nível não deve conhecer a ferramenta que realiza uma determinada ação, em contrapartida essa classe conhece somente a interface que implementa essa ferramenta. Do mesmo modo a interface também não deve conhecer minúcias da ferramenta, entretanto é preciso que a ferramenta atenda as especificações dessa interface.

3. DESENVOLVIMENTO

3.1. MODELOS CONCEITUAIS

Para que se possa entender todo o desenvolvimento da API, parte-se da construção de modelos conceituais como diagrama de classes e de casos de uso da UML, sigla que em portugues significa Linguagem de Modelagem Unificada. uma linguagem de modelagem que permite representar um sistema de forma padronizada (WIKIPEDIA, 2020).

> A UML é adequada para a modelagem de sistemas, cuja abrangência poderá incluir desde sistemas de informação corporativos a serem distribuídos a aplicações baseadas na Web e até sistemas complexos embutidos de tempo real. É uma linguagem muito expressiva, abrangendo todas as visões necessárias ao desenvolvimento e implantação desses sistemas. (WIKIPEDIA, 2020)

Não se discutirá sobre elicitação de requisitos, nem temas relacionados, pois este não é o foco do presente trabalho, entretanto esses modelos podem ser vistos como boas ferramentas para entendimento do desenvolvimento e do funcionamento da API.

Inicia-se então com o diagrama de classe onde será apresentada a interface “Item”, sendo esta interface a responsável por ser o “modelo” dos objetos que serão trabalhados na API.

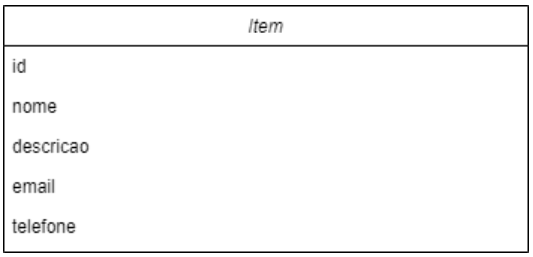

Figura 3. Representação do modelo Item.
Fonte: Autoria própria, 2021.

O próximo diagrama apresentado se refere aos casos de uso que estão presentes na API. Para construí-los utilizou-se os métodos HTTP “POST” e “GET” como visto na seção 2.2 .

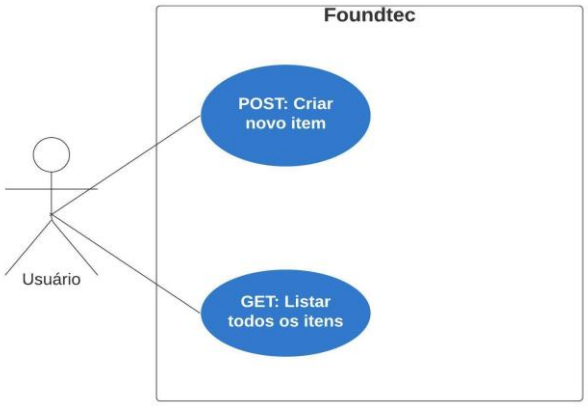

Figura 4. Representação dos casos de uso da API.
Fonte: Autoria própria, 2021.

3.2. INICIANDO O DESENVOLVIMENTO

Antes de dar início ao desenvolvimento, faz-se necessário discutir acerca das tecnologias utilizadas, começando pela linguagem de programação que no presente trabalho optou-se pela utilização de JavaScript, sendo esta uma linguagem leve, interpretada (MDN, 2021) e que ganhou ainda mais espaço com o desenvolvimento do runtime Node.js, extraído a partir da engine presente no navegador “Google Chrome” (OPENJS).

Além destes, ainda foram utilizadas importantes bibliotecas e frameworks como:

- TypeScript: superset desenvolvido pela Microsoft para JavaScript que adiciona a ela sintaxes do paradigma de orientação a objetos (MICROSOFT).
- Express.js: framework que proporciona uma série de módulos para criação de web-services como rotas, tratamento de exceções e outros (OPENJS, 2017).
- TypeORM: biblioteca responsável por abstrair conceitos de SQL e banco de dados relacionais em orientação a objetos, permitindo manipulação de tabelas e entidades através de classes e linguagem JavaScript (TYPEORM).
- TSyringe: biblioteca da Microsoft que auxilia na automatização de instâncias e injeção de dependências (MICROSOFT, 2017).
- Swagger: importante biblioteca para construção de documentação de APIs, disponibilizando uma interface que permite interação direta aos métodos, além de trazer informações necessárias para utilização da API (SMARTBEAR, 2021).

Também optou-se pela utilização do PostgreSQL como sistema gerenciador de banco de dados deste projeto, e a escolha pela sua utilização se dá pela grande fama sendo utilizado por grandes empresas como Skype por exemplo, além de ser gratuito e de código aberto (POSTGRESQL, 1996).

Com as tecnologias e conceitos apresentados, pode-se dar início ao desenvolvimento do código, aqui disponibiliza-se todo o código fonte desenvolvido para este trabalho a partir do repositório remoto hospedado no site GitHub: https://github.com/mateusfarinelli/foundtec.

Iniciou-se o desenvolvimento utilizando a biblioteca TypeORM pela entidade “Item” que é a principal entidade da presente aplicação, pois é ela que vai definir os campos da tabela, conforme desenvolve-se o arquivo “Item.ts”.

Em seguida foram desenvolvidos os arquivos responsáveis por representar os “Casos de Uso” representados pela figura 4. Neste ponto basicamente o que se tem é o desenvolvimento de interfaces, repositories e controllers, que são as representações das classes que o sistema irá acessar para funcionar.

Como visto na seção 2.3 nos princípios S.O.L.I.D, separa-se essas classes para alcançar-se responsabilidade única, aplica-se também a segregação de interfaces, fazendo com que o repository não seja chamado diretamente pelo controller e sim por uma interface, onde nem a interface e nem o controller conhecem a implementação do repository.

```
import { Item } from "../entities/Item";

interface CreateItemDTOInterface {
  nome: string;
  descricao: string;
  email: string;
  telefone: string;
}

interface ItemRepositoryInterface {
  list(): Promise<Item[]>;
  create({ nome, descricao, email, telefone }: CreateItemDTOInterface): Promise<void>;
}

export { ItemRepositoryInterface, CreateItemDTOInterface };
```

Figura 5. Implementação do “ItemRepositoryInterface.ts”.
Fonte: Autoria própria, 2021.

```typescript
import { getRepository, Repository } from "typeorm";
import { Item } from "../../entities/Item";
import { CreateItemDTOInterface, ItemRepositoryInterface } from "../ItemRepositoryInterface";

class ItemRepository implements ItemRepositoryInterface {
  private repository: Repository<Item>;

  constructor() {
    this.repository = getRepository(Item);
  }

  async create({nome, descricao, email, telefone}: CreateItemDTOInterface): Promise<void> {
    const category = this.repository.create({
      nome,
      descricao,
      email,
      telefone
    });
    await this.repository.save(category);
  }

  async list(): Promise<Item[]> {
    const Item = await this.repository.find();

    return Item;
  }
}
export { ItemRepository };
```

Figura 6. Implementação do "ItemRepository.ts".
Fonte: Autoria própria, 2021.

```typescript
import { inject, injectable } from "tsyringe";
import { ItemRepositoryInterface } from "../../repositories/ItemRepositoryInterface";

interface RequestInterface {
  nome: string;
  descricao: string;
  email: string;
  telefone: string;
}

@injectable()
class CreateItemUseCase {
  constructor(
    @inject("ItemRepository")
    private itemRepository: ItemRepositoryInterface
  ) {}

  async execute({ nome, descricao, email, telefone }: RequestInterface): Promise<void> {
    this.itemRepository.create({ nome, descricao, email, telefone });
  }
}

export { CreateItemUseCase };
```

Figura 7. Implementação do "CreateItemUseCase.ts".
Fonte: Autoria própria, 2021.

```typescript
import { Response, Request } from "express";
import { container } from "tsyringe";
import { CreateItemUseCase } from "./CreateItemUseCase";

class CreateItemController {

  async handle(request: Request, response: Response): Promise<Response> {
    const { nome, descricao, email, telefone } = request.body;
    const createItemUseCase = container.resolve(CreateItemUseCase);

    await createItemUseCase.execute({ nome, descricao, email, telefone });

    return response.status(201).send();
  }
}

export { CreateItemController };
```

Figura 8. Implementação do "CreateItemController.ts".
Fonte: Autoria própria, 2021.

As imagens de 5 a 8 são os códigos desenvolvidos para o caso de uso de criação de itens utilizando uma requisição com método POST a API. Neste contexto, o usuário/cliente irá enviar um objeto no corpo da requisição semelhante ao observado na figura 3.

Por sua vez a rota irá chamar o método “handle” da classe “CreateItemController”, que por sua vez irá processar os dados e enviá-los para a o método “execute” presente na classe “CreateItemUseCase”, essa por sua vez irá acessar o método “create” da classe “ItemRepositoryInterface” que por fim irá chamar o método “create” da classe “ItemRepository” salvando as informações no banco de dados.

Veja que só neste exemplo é possível verificar o princípio de responsabilidade única em ação, já que nenhuma das classes faz mais de que uma única operação, ou operação fora daquela em que a classe foi feita para atender. Também é possível verificar o princípio de segregação de interfaces e o princípio de inversão de dependência, onde, as classes “CreateItemUseCase” e “ItemRepositoryInterface” são as respectivas interfaces para “CreaiteItemController” e “ItemRepository” fechando os contratos entre o controller e o repository, dessa forma o mais alto nível de abstração que no caso é o controller não dependa da implementação da classe de mais baixo nível o repository e sim de suas interfaces.

4. RESULTADOS

Chega-se então no resultado apresentado nas imagens a seguir, onde a documentação da API pode ser vista através de uma interface utilizando a biblioteca “Swagger”.

Neste ponto o cliente consegue realizar as requisições através dos métodos disponibilizados, inserindo e listando itens conforme previsto pelos casos de uso propostos conforme figura 4.

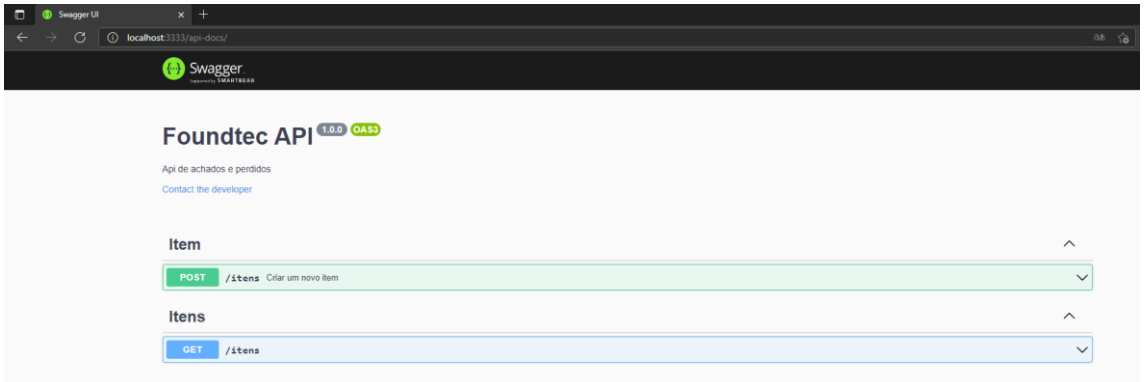

Figura 9. Acessando a documentação desenvolvida com swagger via navegador. Fonte: Autoria própria, 2021.

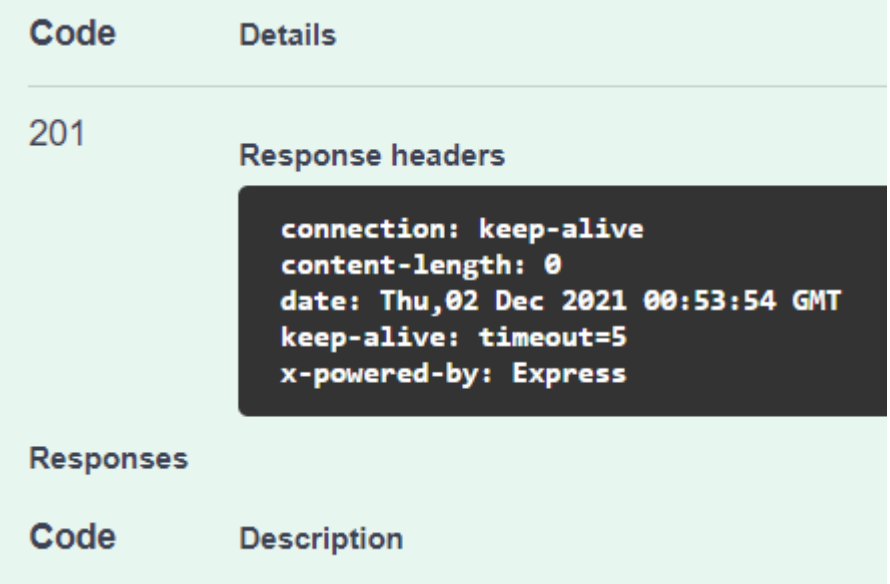

Figura 10. Resposta da API a uma requisição HTTP do tipo POST
Fonte: Autoria própria, 2021.

Com imagens 2, 9 e 10, pode-se ver que o desenvolvimento da API foi concluído, entretanto algumas restrições do padrão REST, bem como alguns dos princípios S.O.L.I.D não foram contemplados devido a simplicidade do projeto.

5. CONCLUSÃO

Pode-se concluir que utilizar os princípios S.O.L.I.D auxiliam na estruturação, desenvolvimento e manutenibilidade de um software, pois mantém classes bem desacopladas e com baixa "complexidade" possibilitando assim realizar eventuais modificações com baixo risco de aparição de bugs ou erros. Além disso, a utilização do padrão REST traz também uma grande colaboração para o desenvolvimento da aplicação utilizando o S.O.L.I.D, já que é necessário que nossa aplicação esteja arquitetada em camadas, e também possibilita ao cliente uma estrutura muito mais organizada e de fácil manipulação.

Não obstante, o desenvolvimento de uma API baseada no padrão REST com aplicação de princípios S.O.L.I.D foi atendido.

E como sugestões de trabalhos futuros pode-se citar:

- Implementação de campos de Geolocalização.
- Implementação de rota de upload de imagens dos objetos e disponibilização dessas imagens.
- Desenvolvimento de uma interface/aplicação (WEB ou mobile) para consumo da API.

Referências

CORREIOS. (2018). **Achados e perdidos**. Disponível em: <http://www2.correios.com.br/servicos/achados_perdidos/>. Acesso em 03/03/2021.

CPTM. (2019). **ACHADOS E PERDIDOS**. Disponível em <http://www.metro.sp.gov.br/sua-viagem/achados-perdidos.aspx>. Acesso em 03/03/2021.

FIELDING, R. T. (2000). **Estilos arquitetônicos e o design de arquiteturas de software baseadas em rede.** Irvine.

KENJI, B; ESCODRO, I. (2020). **O que é SOLID?** Disponível em: <https://www.venturus.org.br/o-que-e-solid/> Acesso em 30/10/2021.

MARTIN, R. C. (2000). **Design Principles and Design Patterns**. Disponível em: <http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf>. Acesso em 30/10/2021 .

MDN. (2021). **JavaScript**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 28/09/2021

MICROSOFT. **O que é TypeScript?**. Disponível em <https://www.typescriptlang.org/>. Acesso em: 28/09/2021.

MICROSOFT. (2017). **TSyringe**. Disponível em <https://github.com/Microsoft/tsyringe>. Acesso em 01/10/2021.

MOODY, M. (2019). **GraphQL vs. REST API Architecture**. Disponível em: <https://medium.com/swlh/graphql-vs-rest-api-architecture-3b95a77512f5>. Acesso em 03/03/2021.

OPENJS. (2017). **Express.js**. Disponível em: <https://expressjs.com/>. Acesso em: 01/10/2021.

OPENJS. **Sobre Node.js®**. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 28/09/2021.

POSTGRESQL. (1996). **PostgreSQL: Sobre**. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 28/09/2021.

RED HAT. (2017). **O que é API?**. Disponível em <https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>. Acesso em 25/10/2021.

SAO PAULO. (2019). **'Achados e Perdidos' do Poupatempo pode ser consultado em aplicativo**. Disponível em <https://www.saopaulo.sp.gov.br/ultimas-noticias/achados-e-perdidos-do-poupatempo-pode-ser-consultado-em-aplicativo/>. Acesso em 03/03/2021.

SMARTBEAR. (2021). **Swagger**. Disponível em: <https://swagger.io/>. Acesso em 01/10/2021.

TYPEORM. **TypeORM**. Disponível em: <https://typeorm.io/#/> . Acesso em

01/10/2021.

UGONNA, T. (2020). **The S.O.L.I.D Principles in Pictures.** Disponível em <https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898>. Acesso em: 30/10/2021.

WIKIPEDIA (2020). **UML**. Disponível em <https://pt.wikipedia.org/wiki/UML>. Acesso em: 17/08/2021.