Desenvolvimento de uma API REST utilizando os princípios SOLID para proporcionar o acesso a dados públicos para aplicações no Brasil

Rafael Londero Cancian[1], Ana Paula Canal[1]

[1]Curso de Bacharelado em Ciência da Computação – Universidade Franciscana (UFN)
CEP 97010-030 – Santa Maria – RS – Brasil

rafael.londero@ufn.edu.br, apc@ufn.edu.br

Abstract. *This Final Graduation Work aims to model an Application Programming Interface (API) emphasizing the use of the Representational State Transfer (REST) architecture and the SOLID principles, having as its theme the access to public data for applications in Brazil. In order to achieve this goal, the agile development methodology Feature Driven Development (FDD), the Typescript programming language and the PostgreSQL database server and manager were chosen. The API is publicly available and makes the data available in a standardized and centralized way.*

Resumo. *Este Trabalho Final de Graduação tem como objetivo modelar uma Application Programming Interface (API) dando ênfase na utilização da arquitetura Representational State Transfer (REST) e dos príncipios SOLID, tendo como tema o acesso a dados públicos para aplicações no Brasil. Visando alcançar este objetivo, foi escolhida a metodologia de desenvolvimento ágil Feature Driven Development (FDD), a linguagem de programação Typescript e o servidor e gerenciador de banco de dados PostgreSQL. A API está disponível publicamente e disponibiliza os dados de forma padronizada e centralizada.*

1. Introdução

Com o grande crescimento no consumo de serviços interconectados pela *web*, surgiu a necessidade de existirem dados disponíveis publicamente para as aplicações terem acesso. Em reflexo desse crescimento as *Application Programming Interfaces* (APIs) também foram melhoradas, porém ainda existia a necessidade desses dados serem acessados de maneira mais restrita e que atendessem certas diretrizes, seja por questões de compatibilidade ou segurança.

Visando servir esses dados de forma mais elaborada para os vários clientes em um sistema distribuído, o cientista da computação Roy Fielding elaborou o modelo de arquitetura *Representational State Transfer* (REST) [Fielding 2000]. Atualmente o REST é descrito como um princípio arquitetural chave para aplicações *web* e tem recebido grande atenção dos programadores.

Boas aplicações começam com um código limpo, isto é onde entram os princípios SOLID. O objetivo dos princípios é a criação de estruturas de software que possibilitem tolerar mudanças, sejam fáceis de entender e sirvam de base para componentes que podem ser usados em muitas outras aplicações [Martin 2018].

Foi escolhido como tema o acesso a dados públicos para aplicações no Brasil, levando em consideração a falta de APIs públicas que disponibilizem esses dados de forma padronizada e centralizada. Pode-se citar alguns desses dados: Códigos de

Endereçamento Postal (CEP), Discagem Direta à Distância (DDD), Cadastro Nacional da Pessoa Jurídica (CNPJ), entre outros.

No desenvolvimento da API, foi utilizada a arquitetura REST e os princípios SOLID para que a API seja segura, sua documentação e seu código sejam fáceis de entender e tenha compatibilidade com o maior número de aplicações possível.

1.1. Objetivo geral

O objetivo deste trabalho consistiu em desenvolver uma *Application Programming Interface* (API) utilizando a arquitetura *Representational State Transfer* (REST) e os princípios SOLID para proporcionar o acesso a dados públicos para aplicações no Brasil de forma padronizada e centralizada.

1.2. Objetivos específicos

Os objetivos específicos deste trabalho foram:

- Modelar a API de acordo com a arquitetura REST, os princípios SOLID e o sistema *fallback*.
- Utilizar o *Feature Driven Developement* (FDD) como metodologia de desenvolvimento ágil.
- Desenvolver a API utilizando a linguagem *Typescript* (superconjunto do *Javascript*) e o Sistema de Gerenciamento de Bancos de Dados (SGBD) PostgreSQL.

1.3. Justificativa

Foi escolhido usar a arquitetura *Representational State Transfer* (REST) pois ela fornece um conjunto de restrições utilizadas para que as requisições HTTP atendam as diretrizes definidas na arquitetura.

Nem todos os componentes de uma arquitetura *web* implantados obedecem a todas as restrições presentes em seu projeto arquitetônico. O REST tem sido usado para definir melhorias e identificar incompatibilidades arquiteturais [Fielding 2000].

Já os princípios SOLID (um acrônimo para cinco princípios de projeto de software orientados a objetos, que serão abordados no referencial teórico) foram escolhidos por terem como finalidade: reduzir a complexidade do código orientado a objetos, o acoplamento entre classes, separar responsabilidades e definir muito bem as relações entre elas, como forma de melhorar a qualidade interna do código-fonte.

Os princípios SOLID orientam à organização de métodos e estruturas de dados em classes, e como essas classes devem ser interconectadas. O uso da palavra “classe” não implica que esses princípios sejam aplicáveis apenas a programas orientados a objetos. Uma classe é um agrupamento acoplado de métodos e dados. Todo sistema de software possui tais agrupamentos, sejam eles chamados de classes ou não. Os princípios SOLID se aplicam a esses agrupamentos [Martin 2018].

A importância desta API para o desenvolvimento *web* se deve pelo fato que ela disponibiliza dados públicos de forma padronizada e centralizada, e utiliza como plano de contingência o *fallback* para alguns dos dados públicos. Com ele a aplicação funciona de maneira a utilizar mais de uma API pública para um mesmo dado, para que aumente a chance desse dado estar disponível, oferecendo assim uma maior compatibilidade e facilidade aos desenvolvedores que irão utilizá-la.

Por fim, a API e a sua documentação estão disponíveis publicamente em um repositório no GitHub [Cancian 2022].

2. Referencial Teórico

Nesta seção serão abordados os conceitos necessários a este trabalho, relacionados à Arquitetura REST, os Princípios SOLID, o plano de contingência *fallback*, as tecnologias utilizadas, a metodologia FDD e o acesso a dados públicos para aplicações no Brasil.

2.1. Arquitetura REST

O *Representational State Transfer* (REST), que em português significa Transferência de Estado Representacional, é um estilo de arquitetura de software desenvolvido para servir aplicações *web* [Marques 2018].

O REST oferece um conjunto de restrições necessárias para se desenvolver um serviço coeso, escalável e com elevada performance, independentemente da plataforma e da linguagem [Marques 2018].

2.1.1. REST ou RESTful?

Para uma *Application Programming Interface* (API) ser considerada RESTful, esta deve obrigatoriamente implementar algumas restrições definidas na arquitetura REST, sendo elas [Marques 2018]:

- Cliente-Servidor.
- Sem estado (*stateless*).
- *Cache*.
- Interface uniforme.
- Sistema em camadas.

Como a API desenvolvida implementa apenas três restrições que serão abordadas nos tópicos abaixo, ela não pode ser considerada uma API RESTful, portanto ela é somente uma API REST.

2.1.2. Cliente-Servidor

A primeira restrição que uma aplicação REST deve ter é separar a arquitetura e as responsabilidades em dois ambientes (cliente e servidor), tornando-se independente e consequentemente escalável [Fielding 2000].

Na situação ilustrada na Figura 1, o consumidor do serviço (cliente ou *front-end*) preocupa-se apenas com a interface enquanto o fornecedor de serviço (servidor ou *back-end*) é responsável por devolver uma resposta ao cliente através da execução de um pedido enviado pelo cliente [Marques 2018].

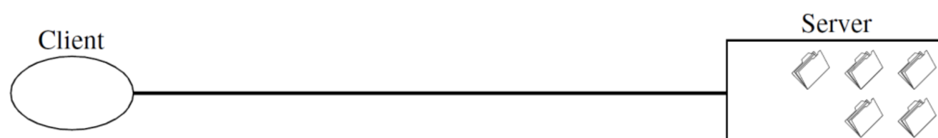

Figura 1. Cliente-Servidor [Fielding 2000, p.78].

2.1.3. Sem estado (*stateless*)

Em seguida, adiciona-se uma restrição à interação cliente-servidor: a comunicação deve ser sem estado, ou seja, o servidor não guarda nenhuma informação relativa ao estado do

cliente. Essa informação é armazenada localmente e um cliente pode fazer vários pedidos ao servidor [Fielding 2000].

Na situação ilustrada na Figura 2, cada pedido ao servidor é feito de forma independente e padronizada passando apenas a informação necessária ao servidor para que ele possa processá-la de forma mais adequada e correta. A desvantagem é que pode diminuir o desempenho da rede aumentando os dados repetitivos (sobrecarga por interação), pois esses dados não podem ser deixados no servidor em um contexto compartilhado [Marques 2018].

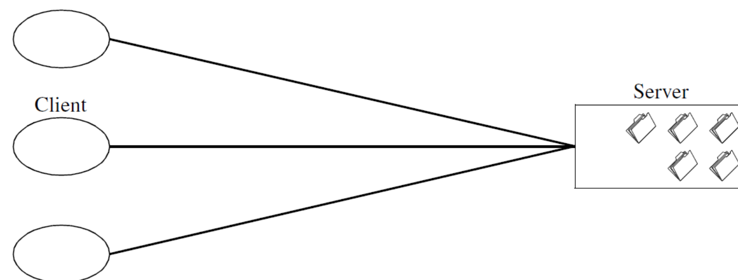

Figura 2. Cliente-Servidor sem estado [Fielding 2000, p.78].

2.1.4. *Cache*

Para melhorar a eficiência da rede, adiciona-se a restrição de *cache*, de modo a eliminar parcialmente ou até completamente algumas requisições. Quando um cliente faz um pedido ao servidor, a resposta fica armazenada temporariamente em *cache* [Fielding 2000].

Na situação ilustrada na Figura 3, se vários clientes fizerem o mesmo pedido ao servidor, o cliente utiliza o que está em seu *cache* local sem ter que gastar uma nova requisição para o servidor processá-la, melhorando a eficiência, escalabilidade e desempenho pelo utilizador. No entanto, a informação em *cache* quando muito utilizada pode diminuir a confiabilidade dos dados tornando-os obsoletos [Marques 2018].

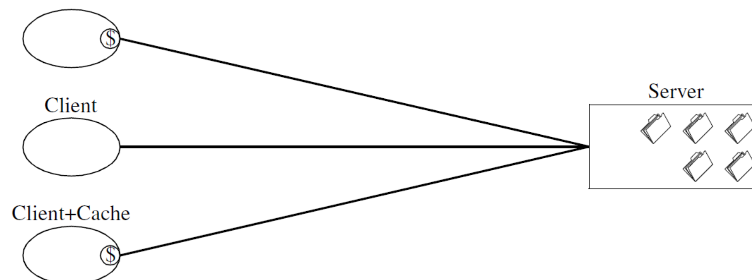

Figura 3. Cliente-Servidor sem estado e com *cache* [Fielding 2000, p.80].

2.2. Princípios SOLID

Os conceitos da orientação a objetos começaram a ser reunidos no final dos anos 1980, enquanto Robert Martin debatia sobre princípios de design de software com colegas de trabalho na rede USENET. Ao longo dos anos, os princípios sofreram diversas modificações. Alguns foram adicionados, outros deletados e ainda alguns agrupados. O conjunto final dos conceitos se estabilizou no início dos anos 2000, embora estivesse em uma ordem diferente da atual [Martin 2018].

Por volta de 2004, Michael Feathers enviou um e-mail para Robert Martin dizendo que se ele reorganizasse os princípios, suas letras iniciais formariam o acrônimo S.O.L.I.D. onde estão contidos os princípios [Martin 2018]:

- *Single Responsiblity Principle* (Princípio da responsabilidade única).
- *Open-Closed Principle* (Princípio Aberto-Fechado).
- *Liskov Substitution Principle* (Princípio da substituição de Liskov).
- *Interface Segregation Principle* (Princípio da Segregação da Interface).
- *Dependency Inversion Principle* (Princípio da inversão da dependência).

2.2.1. Princípio de Responsabilidade Única

O *Single Responsiblity Principle* (SRP), em português Princípio de Responsabilidade Única é o primeiro conceito que compõe o conjunto. O SRP tem por definição que todos os módulos devem ter uma única responsabilidade, em outras palavras, estar relacionado a apenas um requisito funcional e, portanto, ter apenas uma razão para ser alterado. Cada classe ou módulo deve ter uma responsabilidade única, pois classes infladas engessam o projeto [Domingues 2021].

De acordo com Martin, os níveis de modularidade de um sistema podem ser assegurados aplicando esse princípio, de modo que cada módulo de software tenha um, e apenas um, motivo para mudar. Pode ser visto na Figura 6 que a classe Employee tem mais de uma responsabilidade, portanto o princípio foi violado [Martin 2018].

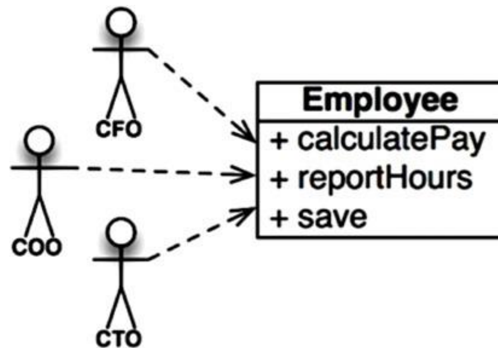

Figura 6. Violação do Princípio de Responsabilidade Única [Martin 2018, p.83].

2.2.2. Princípio Aberto-Fechado

O *Open-Closed Principle* (OCP), em português Princípio Aberto-Fechado foi introduzido por Bertrand Meyer, no ano de 1988, com a seguinte afirmação “Os módulos de um software devem ser abertos para extensões, mas fechados para modificações” [Martin 2018].

Bertrand Meyer tornou esse princípio famoso na década de 1980. A essência é que, para que os sistemas de software sejam fáceis de mudar, eles devem ser projetados para permitir que o comportamento desses sistemas seja alterado pela adição de novo código, em vez de alterar o código existente [Martin 2018].

Essa definição faz referência a duas situações: primeiro um módulo é considerado aberto se ainda estiver disponível para receber extensões e segundo, um módulo é dito fechado caso esteja à disposição para ser utilizado por outros módulos. Isto é, o comportamento de um segmento do software pode ser estendido sem que tenha a necessidade de ser modificado ou adaptado. Esse princípio é implementado por meio de herança ou interfaces [Domingues 2021].

2.2.3. Princípio da Substituição de Liskov

A cientista da computação Barbara Liskov introduziu em 1987 o *Liskov Substitution Principle* (LSP), em português Princípio da Substituição de Liskov. Em suma, esse

princípio diz que para construir sistemas de software a partir de partes intercambiáveis, essas partes devem aderir a um contrato que permita que essas partes sejam substituídas umas pelas outras [Martin 2018].

Em seu artigo Barbara Liskov escreveu o seguinte como uma forma de definir subtipos:

> "O que se deseja aqui é algo como a seguinte propriedade de substituição: Se para cada objeto 01 do tipo S houver um objeto 02 do tipo T de modo que para todos os programas P definidos em termos de T, o comportamento de P é inalterado quando 01 é substituído para 02, então S é um subtipo de T" [Liskov 1987, p. 25, traduzido pelo autor].

Em outras palavras, esse conceito assegura que a utilização de herança e implementação de interface seja feita de forma que não altere o comportamento atual do sistema, por exemplo: um sistema que utiliza PostgreSQL como base de dados deve manter seu estado de funcionamento ao ser migrado para MySQL sem que haja necessidade de tratar exceções ou aplicar correções [Domingues 2021].

2.2.4. Princípio de Segregação de Interface

De todos os princípios SOLID este, aparentemente é o mais simples. No entanto, apesar dessa simplicidade o *Interface Segregation Principle* (ISP), em português Princípio de Segregação de Interface acaba não sendo totalmente compreendido [Domingues 2021].

Na situação ilustrada na Figura 7, existem vários usuários que utilizam os métodos da classe OPS. Supondo que a classe User1 use apenas o método op1, User2 use apenas a op2, e User3 usa apenas a op3 [Martin 2018].

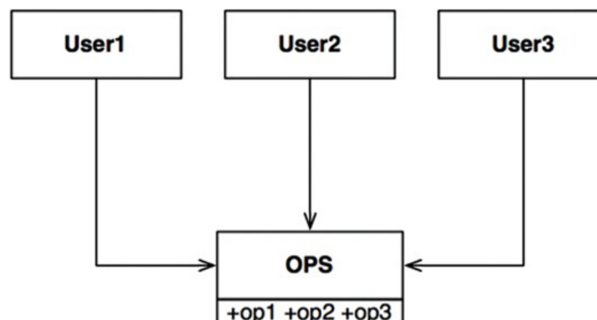

Figura 7. Diagrama antes de aplicar o Princípio de Segregação de Interface [Martin 2018, p.102].

Se OPS é uma classe escrita em uma linguagem como Java, claramente naquele caso, o código fonte da classe User1 dependerá totalmente dos métodos op2 e op3, mesmo embora não os chame. Essa dependência significa que uma mudança no código-fonte de op2 no OPS forçará o User1 a ser recompilado, mesmo que não se tenha mudado nada de importante. Este problema pode ser resolvido separando as funções em interfaces como mostrado na Figura 8 [Martin 2018]:

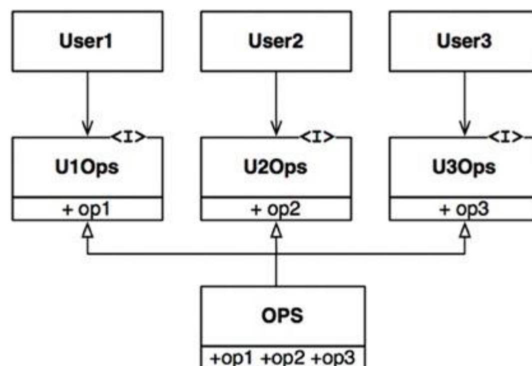

Figura 8. Diagrama depois de aplicar o Princípio de Segregação de Interface [Martin 2018, p.103].

2.2.5. Princípio de Inversão de Dependência

O *Dependency Inversion Principle* (DIP), em português Princípio de Inversão de Dependência nos diz que os sistemas mais flexíveis são aqueles nos quais as dependências do código-fonte se referem apenas a abstrações, não a concreções [Martin 2018].

O código que implementa a política de alto nível não deve depender do código que implementa os módulos de baixo nível. Em vez disso, os módulos devem depender das políticas [Martin 2018].

Em uma linguagem em que a tipagem é estática, como Java, isso significa que o uso de instruções *use*, *import* e *include* devem se referir apenas a módulos contendo interfaces, classes abstratas ou algum outro tipo de declaração abstrata. Nada que é concreto deve ser dependente [Martin 2018].

2.3. Sistema *fallback*

Quando uma API utiliza o sistema *fallback*, os envios dessa API são estruturados em passos e caso um desses passos falhe, automaticamente será executado outro passo [Wavy 2021].

Por exemplo, uma API *fallback* para consultar Códigos de Endereçamento Postal (CEP) pode pegar os dados para consulta utilizando outras APIs, e se uma delas falhar, automaticamente outra API fornecerá os dados no lugar dela.

2.4. Tecnologias

Quando o *Javascript* foi criado, ele foi introduzido como uma linguagem de programação direcionada mais para o lado do cliente (*front-end*). Mas com o passar do tempo, os desenvolvedores perceberam que o *Javascript* poderia ser usado também como uma linguagem de programação para servidores (*back-end*). No entanto, à medida que a linguagem foi crescendo, o código tornou-se complexo e pesado, por não ser capaz de cumprir o requisito de uma linguagem de programação orientada a objetos. Levando em consideração esse problema o *Typescript* foi desenvolvido pela Microsoft em 2012 com o intuito de preencher essa lacuna [Microsoft 2022].

O *Typescript* é um superconjunto do *Javascript*, ou seja, ele possui todos os recursos do *Javascript* e adiciona tipagem estática opcional à linguagem. Ao usar o compilador nativo para converter (transpilar) o arquivo *Typescript* (.ts) em arquivo *Javascript* (.js), ele facilita a integração com projetos *Javascript*. Além disso, ao fornecer verificação para tipos estáticos, permite que o programador verifique e atribua variáveis

ou tipos à uma função, tornando o código mais fácil de ler e sinalizando potenciais erros [Microsoft 2022].

O PostgreSQL é um servidor e gerenciador de banco de dados (SGBD) relacional de código aberto que usa e estende a linguagem *Structured Query Language* (SQL) combinada com muitos recursos que armazenam e dimensionam com segurança as cargas de trabalho de dados mais complicadas [PostgresSQL 2022].

O servidor do PostgreSQL pode ser executado em todos os principais sistemas operacionais e é compatível com o ACID (acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade) desde 2001. Acabou conquistando uma forte reputação por sua arquitetura comprovada, confiabilidade, integridade de dados, conjunto robusto de recursos, extensibilidade e dedicação da comunidade em manter o código aberto por trás do software para fornecer soluções inovadoras e de desempenho consistentes [PostgreSQL 2022].

2.5. Metodologia *Feature Driven Development* (FDD)

O *Feature Driven Development* (FDD) é uma metodologia de desenvolvimento ágil que foi concebido originalmente por Peter Coad e seus colegas como um modelo de processos prático para a engenharia de software orientada a objetos.

Dessa forma o FDD oferece maior ênfase às diretrizes e técnicas de gerenciamento de projeto do que muitos outros métodos ágeis. Existem cinco atividades metodológicas “colaborativas” que no FDD são denominados “processos” conforme mostra a Figura 9 [Pressman 2011].

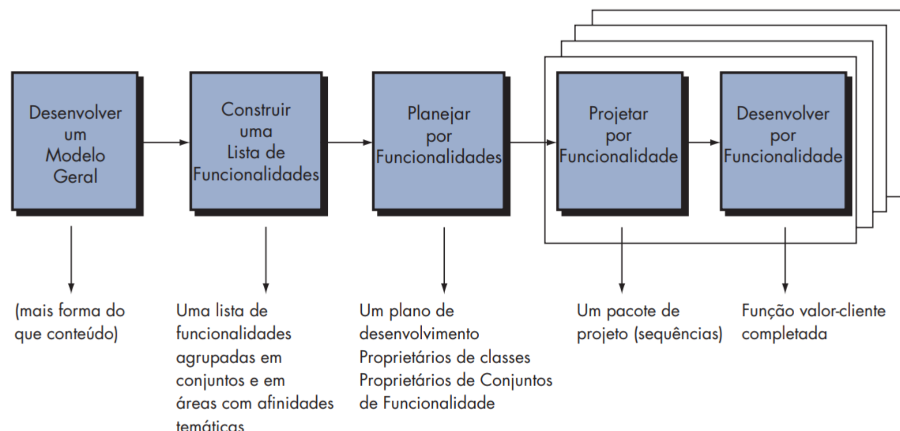

Figura 9. Os cinco processos dentro do FDD [Pressman 2011, p.99].

2.6. Acesso a dados públicos para aplicações no Brasil

O acesso programado a informações é algo fundamental para uma boa comunicação entre sistemas, mas uma informação tão útil e pública quanto um Código de Endereçamento Postal (CEP) não consegue ser acessada diretamente por um navegador por conta da API dos Correios não possuir *Cross-Origin Resource Sharing*[1] (CORS) habilitado [Deschamps e Franca 2020].

Sendo assim, surgiu a ideia de criar uma API que disponibilize esses dados de maneira padronizada e centralizada. Padronizar a forma como se acessa a API ajuda as

[1] *Cross-Origin Resource Sharing* (Compartilhamento de recursos com origens diferentes) é um mecanismo que usa cabeçalhos Hypertext Transfer Protocol (HTTP) adicionais para permitir que um servidor indique quaisquer origens (domínio ou porta) além da sua própria, a partir da qual um navegador deve permitir o carregamento de recursos [Mozilla 2021].

aplicações a terem uma maior consistência, pois ao invés do desenvolvedor ter que se preocupar com a documentação de três APIs por exemplo, ele terá que se preocupar somente com uma, justamente por nela conter de maneira centralizada alguns dos dados públicos listados abaixo:

- Informações sobre o sistema bancário brasileiro.
- Código de Endereçamento Postal (CEP).
- Cadastro Nacional da Pessoa Jurídica (CNPJ).
- Discagem Direta à Distância (DDD).
- Feriados Nacionais.
- Geolocalização por IP.
- Instituto Brasileiro de Geografia e Estatística (IBGE).

3. Trabalhos Correlatos

Nesta seção serão apresentados os trabalhos correlatos que utilizaram algumas das tecnologias citadas anteriormente e contribuíram para o desenvolvimento deste projeto.

O artigo de Sousa, Silva e Bandeira (2017) aborda a criação de um sistema de gestão de eventos distribuído aplicando as principais restrições da arquitetura REST, mostrando como cada restrição busca criar uma estrutura de organização do servidor para otimização e organização de serviços distribuídos. O artigo teve como conclusão que o uso da arquitetura REST na construção de aplicação para *web* é uma ótima opção, visto as vantagens de escalabilidade, separação de componentes e sistema de hipermídias distribuídos.

O projeto proposto por Marques (2018) teve como maior objetivo desenvolver uma API REST que permitiu integrar a uma aplicação *web* que estava paralelamente sendo desenvolvida. Como foi utilizado a arquitetura REST, a API foi desenvolvida de forma simples e rápida, através de uma solução onde as diferentes responsabilidades são claramente delimitadas pelos diferentes elementos envolvidos.

O trabalho de Domingues (2021) descreve a reformulação de um sistema de gerenciamento de cartões de crédito buscando atender os princípios SOLID e alguns elementos da arquitetura limpa. Nesse processo, a estrutura do sistema foi organizada em camadas que separam entidades, casos de uso, interfaces, *frameworks* e drivers. O trabalho concluiu que apesar de ser um projeto de pequeno porte, a forma como os princípios SOLID foram implementados proporcionou um produto escalável e compatível com manutenções futuras.

Como visto nos trabalhos correlatos apresentados acima, o uso da arquitetura REST e dos princípios SOLID foram fundamentais para o desenvolvimento de um sistema organizado e eficaz. Os trabalhos foram escolhidos pensando nas similaridades com as tecnologias utilizadas neste projeto.

4. Desenvolvimento da API

O desenvolvimento da API seguiu os cinco processos da metodologia FDD, sendo eles: Modelo Geral, Lista de Funcionalidades, Planejar por Funcionalidades, Projetar por Funcionalidades e Desenvolver por Funcionalidades.

Pode ser visto na Figura 10, um diagrama onde o usuário acessa os dados por meio de uma aplicação que estará consumindo a API ou diretamente pelo navegador. Após o usuário solicitar os dados, é feita uma requisição para a API por meio das rotas que estão

disponíveis publicamente e também é armazenado no banco de dados um registro dessa requisição, que pode servir para resolver possíveis problemas no futuro.

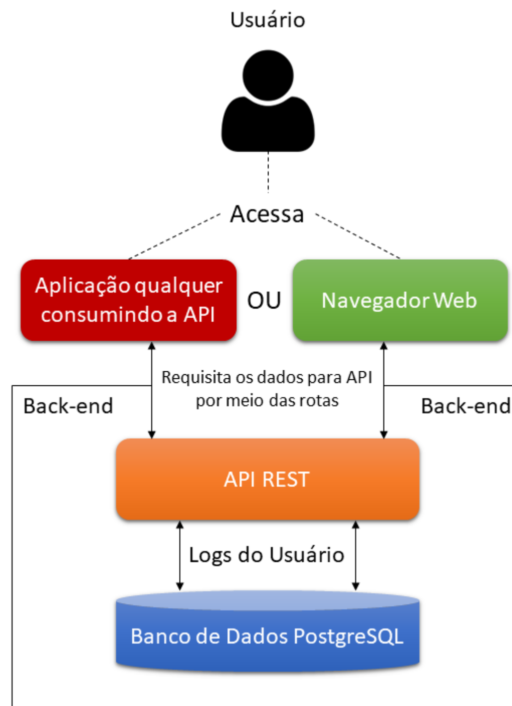

Figura 10. Diagrama de como o usuário irá acessar a API e serão feitas as requisições.

4.1. Modelo Geral

O primeiro processo da metodologia FDD é a definição do Modelo Geral, que serve para conhecer e analisar o sistema no contexto em que ele está inserido. A partir disso como ilustrado na Figura 11 é desenvolvido o Diagrama de Domínio, onde é possível visualizar os conceitos e os relacionamentos do sistema.

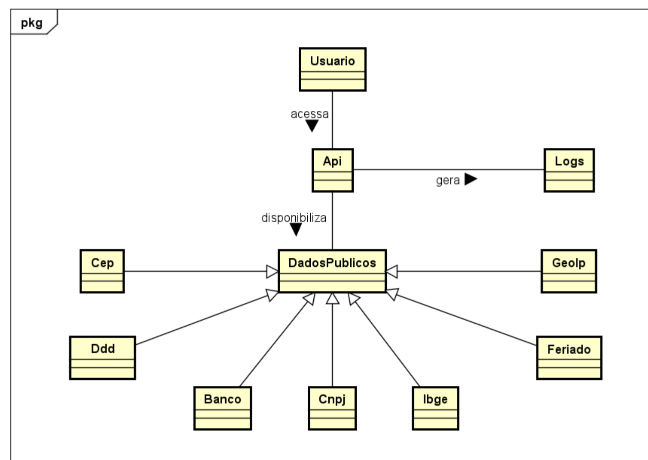

Figura 11. Diagrama de Domínio.

Como pode ser visto ainda na Figura 11, o usuário irá acessar a API que estará disponibilizando cada um desses dados, onde estarão herdando as propriedades dos dados públicos. Quando for feita qualquer requisição, não importando se ela foi bem sucedida ou apresentou erros, é armazenado no banco de dados os *logs* gerados pela API.

4.2. Lista de Funcionalidades

No segundo processo da metodologia FDD é criada uma lista de funcionalidades do sistema, descrevendo cada uma delas. O Quadro 1 mostra os requisitos funcionais (RF), onde são representadas as tarefas e o comportamento que o sistema deve ter.

Quadro 1. Requisitos funcionais.

ID	Descrição
RF01	O sistema deve disponibilizar rotas que estarão disponíveis publicamente.
RF02	O sistema deve permitir que o usuário acesse os dados por meio de uma aplicação que estará consumindo a API ou diretamente pelo navegador.
RF03	O sistema deve retornar um código de status HTTP (*Hypertext Transfer Protocol*) e os dados em formato JSON (*JavaScript Object Notation*).
RF04	O sistema deve ser capaz de consultar dados do CEP em outras APIs públicas, caso seja requisitado.
RF05	O sistema deve ser capaz de consultar dados do CNPJ em outras APIs públicas, caso seja requisitado.
RF06	O sistema deve ser capaz de consultar dados do DDD em outras APIs públicas, caso seja requisitado.
RF07	O sistema deve ser capaz de consultar dados dos Feriados Nacionais em outras APIs públicas, caso seja requisitado.
RF08	O sistema deve ser capaz de consultar dados sobre o sistema bancário brasileiro em outras APIs públicas, caso seja requisitado.
RF09	O sistema deve ser capaz de consultar dados do IBGE em outras APIs públicas, caso seja requisitado.
RF10	O sistema deve ser capaz de consultar dados da Geolocalização por IP em outras APIs públicas, caso seja requisitado.
RF11	O sistema deve armazenar no banco de dados um registro de todas as requisições.

A seguir, é apresentado o Quadro 2 que mostra os requisitos não funcionais (RNF), onde consta a descrição da parte técnica que não constitui as funcionalidades do sistema.

Quadro 2. Requisitos não funcionais.

ID	Descrição
RNF01	O sistema foi modelado de acordo com a arquitetura REST, os princípios SOLID e o sistema *fallback*.
RNF02	O sistema foi desenvolvido utilizando a linguagem *Typescript* (superconjunto do *Javascript*).
RNF03	O sistema utiliza o sistema de gerenciamento de bancos de dados PostgreSQL para armazenar os registros das requisições.
RNF04	O sistema deve estar com o CORS (*Cross-Origin Resource Sharing*) devidamente configurado para permitir o acesso de qualquer origem e somente requisições do tipo GET.
RNF05	O sistema deve ser devidamente documentado para que o usuário consiga acessar a API.

4.3. Projetar por Funcionalidades

O quarto processo da metodologia FDD consiste em projetar cada uma das funcionalidades possibilitando ter uma visão estática do sistema, que pode ser

representado como ilustrado na Figura 12 pelo diagrama de classes. O diagrama mostra todas as classes que serão implementadas futuramente na aplicação com seus respectivos atributos e métodos, assim como as suas relações com as demais classes.

O diagrama de classes foi inteiramente modelado baseado nos princípios SOLID, onde pode ser visto o uso principalmente de três princípios, sendo eles: o Princípio de Responsabilidade Única, o Princípio da Substituição de Liskov e o Princípio de Inversão de Dependência. Como pode ser visto ainda na Figura 12, cada dado que é uma entidade possui o seu repositório, o seu controlador e o seu caso de uso, onde cada caso de uso chama a interface de cada repositório e não diretamente ele, assegurando o Princípio de Responsabilidade Única e o Princípio de Inversão de Dependência.

Como também o sistema está bem desacoplado e dependendo apenas das interfaces, qualquer mudança que for realizada em uma classe não quebrará o restante da aplicação, dessa forma garantindo o Princípio da Substituição de Liskov.

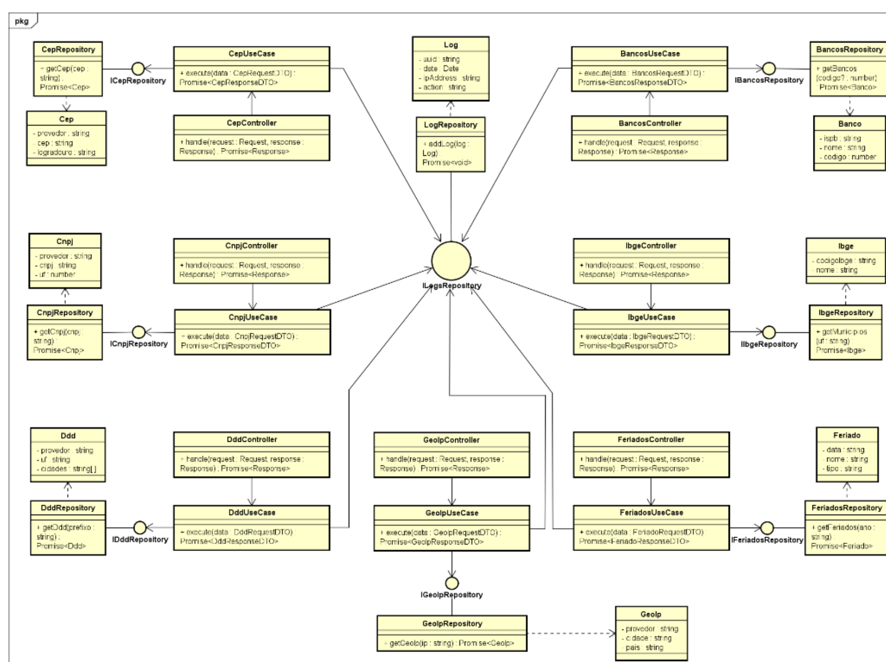

Figura 12. Diagrama de Classe.

4.4. Desenvolver por Funcionalidades

O quinto e último passo da metodologia FDD consiste em dar início a implementação do projeto, que buscou seguir à risca o diagrama de classes, como mostrado na Figura 12. O primeiro dado público implementado foi o CEP, baseado na ordem da lista de funcionalidades que foi apresentada no tópico 4.2. Sendo assim, a primeira classe da funcionalidade RF04 (CEP) implementada foi a sua entidade, como pode ser visto na Figura 13.

```
1  export class Cep {
2    public provedor: string;
3    public cep: string;
4    public logradouro: string;
5    public complemento?: string;
6    public bairro: string;
7    public localidade: string;
8    public uf: string;
9
10   constructor(props: Cep) {
11     Object.assign(this, props);
12   }
13 }
```

Figura 13. Implementação da entidade do CEP.

Após ter sido realizada a implementação da entidade `Cep` foi feita a interface `ICepRepository` (Figura 14) e o repositório `CepRepository`, onde esse repositório implementa a interface. O repositório faz a requisição para a API do provedor ViaCEP[2] e verifica se houve erros, caso não houver erros é criado um novo objeto `Cep` baseado na sua entidade e ao fim retornado, como pode ser visto na Figura 15.

```
4  export interface ICepRepository {
5    getCepFromViaCep(cep: string): Promise<Cep | Erro>;
6    getCepFromWidenet(cep: string): Promise<Cep | Erro>;
7    getCepFromInverTexto(cep: string): Promise<Cep | Erro>;
8  }
```

Figura 14. Implementação da interface do CEP.

```
7   export class CepRepository implements ICepRepository {
8     async getCepFromViaCep(cep: string): Promise<Cep | Erro> {
9       let erro: Erro = new Erro({
10        estado: false
11      });
12
13      const result: AxiosResponse | void = await axios
14      .get(`https://viacep.com.br/ws/${cep}/json/`)
15      .catch(function (error) {
16        console.log('Erro do Axios: ' + error.message);
17        erro.estado = true;
18        erro.mensagem = error.response ? error.response.data : error.message;
19      });
20
21      // Tratamento de erro genérico
22      if (!result || erro.estado == true) return erro;
23
24      const cepResponse = new Cep({
25        provedor: 'ViaCEP',
26        cep: result.data.cep,
27        logradouro: result.data.logradouro,
28        complemento: result.data.complemento,
29        bairro: result.data.bairro,
30        localidade: result.data.localidade,
31        uf: result.data.uf
32      });
33
34      return cepResponse;
35    }
```

Figura 15. Implementação do repositório do CEP.

Seguindo a ordem da implementação foi feito o controlador `CepController`, onde são armazenadas em variáveis os parâmetros *path* e *query*. Após isso é feita a implementação da funcionalidade na classe `CepUseCase` como pode ser visto na Figura 16.

```
4   export class CepController {
5     constructor(private cepUseCase: CepUseCase) {}
6
7     async handle(request: Request, response: Response): Promise<Response> {
8       const { cep } = request.params;
9       const { provedor } = request.query;
10
11      const ip = request.socket.remoteAddress;
12
13      const result = await this.cepUseCase.execute({
14        cep,
15        provedor: provedor?.toString(),
16        ip
17      });
18
19      return response.status(200).send(result);
20    }
21  }
```

Figura 16. Implementação do controlador do CEP.

---

[2] Como o CEP está utilizando o sistema *fallback* ele possui múltiplos provedores, porém neste artigo é mostrado apenas um dos provedores que nesse caso é o ViaCEP.

Após ter sido realizada a implementação do controlador é feita a chamada do `CepUseCase` onde foi implementado as funcionalidades do CEP. Nesta classe são chamadas no construtor as interfaces `ICepRepository` e `ILogsRepository` como pode ser visto na Figura 17, assegurando assim os princípios de Inversão de Dependência e de Substituição de Liskov, pois a classe não está dependendo diretamente dos repositórios e sim das suas interfaces, portanto pode ser desacoplada sem quebrar o restante da aplicação.

```
11  export class CepUseCase {
12    constructor(
13      private cepRepository: ICepRepository,
14      private logsRepository: ILogsRepository
15    ) {}
16
17    async execute(data: CepRequestDTO): Promise<CepResponseDTO> {
18      let erros: any[] = [];
19
20      // Provedor ViaCEP
21      const viacep = await this.cepRepository.getCepFromViaCep(data.cep);
22      if (viacep instanceof Erro) {
23        erros.push({
24          provedor: 'ViaCEP',
25          erro: viacep.mensagem
26        });
27      } else {
28        await this.logsRepository.addLog(new Log({
29          ipAddress: data.ip,
30          action: 'Cep',
31          provider: 'ViaCEP',
32          status: 200
33        }));
34        return viacep;
35      }
36
37      // Outros provedores
38      // ...
39
40      await this.logsRepository.addLog(new Log({
41        ipAddress: data.ip,
42        action: 'Cep',
43        provider: 'Todos',
44        status: 500,
45        errors: erros
46      }));
47      throw new ThrowMessageRequest().internalServerErrorObject('Todos os serviços de CEP retornaram erro.', erros);
48    }
49  }
```

Figura 17. Implementação da funcionalidade do CEP.

Seguindo ainda na Figura 17 ocorre a chamada de múltiplos provedores assegurando assim o sistema *fallback*, onde cada provedor chama a sua respectiva função contida no repositório `CepRepository`. Caso ocorra um erro por parte do provedor, é passado para o próximo e o erro gerado é adicionado a um *array* de erros que posteriormente será exibido caso todos os provedores falhem na requisição.

Caso a requisição seja bem sucedida ou de algum erro será feito um registro no banco de dados PostgreSQL, onde é chamada a função `addLog` que está contida no repositório `LogsRepository`, passando por parâmetro uma entidade `Log`.

Por fim, tem-se a implementação da rota do CEP do tipo *get* onde são chamados os *middlewares*, que são funções executadas entre a chamada da rota e o controlador, como pode ser visto na Figura 18.

Ao chamar a rota passando como parâmetro um CEP válido, utilizando o navegador ou programas como Postman e Insomnia[3] a API irá retornar um código de status HTTP (*Hypertext Transfer Protocol*) e os dados em formato JSON (*JavaScript Object Notation*) como pode ser visto na Figura 19, onde foi utilizado o CEP da instituição como referência.

[3] Postman e Insomnia são ferramentas para construir e utilizar APIs.

```
15  cepRouter.get(
16    '/api/cep/v1/:cep',
17    middlewareParamsValidation(cepParamsValidationRequest),
18    middlewareQueryValidation(cepQueryValidationRequest),
19    middlewareAntiFloodPrevention(),
20    async (request, response, next) => {
21      try {
22        await cepController.handle(request, response);
23      } catch (e) {
24        next(e);
25      }
26    }
27  );
```

Figura 18. Implementação da rota do CEP.

```
Status: 200 OK   Time: 539 ms   Size: 572 B
1  {
2      "provedor": "ViaCEP",
3      "cep": "97010-030",
4      "logradouro": "Rua dos Andradas",
5      "complemento": "de 466 a 1304 - lado par",
6      "bairro": "Bonfim",
7      "localidade": "Santa Maria",
8      "uf": "RS"
9  }
```

Figura 19. Resposta para o CEP solicitado utilizando o provedor ViaCEP.

Todos os demais dados (Informações sobre o sistema bancário brasileiro, CNPJ, DDD, Feriados Nacionais, Geolocalização por IP e IBGE) foram implementados de maneira similar a apresentada acima, onde cada dado que é uma entidade possui o seu repositório implementando uma interface, o seu controlador, o seu caso de uso e a sua rota, dando ênfase no Princípio de Responsabilidade Única. O CEP foi escolhido como exemplo pois foi o primeiro a ser implementado e possui mais provedores.

5. Conclusão

Este Trabalho Final de Graduação apresentou a modelagem e o desenvolvimento da API utilizando a arquitetura REST e os princípios SOLID, dando ênfase nas regras da arquitetura e nos princípios. Para esta API ter um propósito e ilustrar bem essas regras e esses princípios foi escolhido como tema o acesso a dados públicos para aplicações no Brasil.

Outro aspecto fundamental foi a escolha das tecnologias a serem utilizadas juntamente com a metodologia ágil FDD que possibilitou ter uma visão mais ampla do projeto, assim permitindo ter um bom planejamento, que foi fundamental para o desenvolvimento do projeto.

Os trabalhos correlatos enfatizam o uso da arquitetura REST e dos princípios SOLID em diferentes sistemas, concluindo que após a utilização de ambos o projeto foi desenvolvido de uma forma rápida e proporcionou como resultado final um produto escalável e compatível com manutenções futuras.

Após o desenvolvimento do trabalho ter sido concluído, foi possível atingir todos os objetivos propostos, tendo como resultado a disponibilização de uma API modelada na arquitetura REST e utilizando os princípios SOLID, juntamente com a sua documentação que está disponível publicamente para qualquer programador poder ler e acessar os dados públicos que foram citados anteriormente neste trabalho. Graças ao sistema *fallback*, a disponibilidade de alguns dos dados que o utilizam acaba aumentando, evitando assim o programador ter que utilizar múltiplos provedores em seu próprio projeto.

Dando continuidade ao desenvolvimento do trabalho visando aperfeiçoá-lo, pretende-se implementar mais dados públicos e também se possível utilizar mais provedores para os dados que ainda não utilizam o sistema *fallback*.

Referências

Cancian, R. L. (2022). *Trabalho Final de Graduação*. Disponível em: https://github.com/rafaellcancian/Trabalho-Final-de-Graduacao. Acesso em: 02 dez. 2022.

Deschamps, F. e Franca, L. P. (2020). *Brasil API*. Disponível em: https://brasilapi.com.br/docs. Acesso em: 01 jun. 2022.

Domingues, T. S. (2021). *Reformulação de um Sistema de Gerenciamento de Cartões de Crédito para atender os princípios SOLID*. Engenharia da Computação. Universidade Federal de Mato Grosso, Várzea Grande. Trabalho de Conclusão de Curso. Disponível em: https://bdm.ufmt.br/handle/1/2004. Acesso em: 17 mar. 2022.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Information and Computer Science. University of California, Irvine. Dissertação. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Acesso em: 06 abr. 2022.

Liskov, B. (1988). *Data Abstraction and Hierarchy*. SIGPLAN Notices 23, 5.

Marques, A. I. A. (2018). *Desenvolvimento de API para aplicação cloud*. Mestrado em Engenharia Informática. Instituto Politécnico de Leiria, Leiria. Dissertação de Mestrado. Disponível em: https://iconline.ipleiria.pt/handle/10400.8/3263. Acesso em: 15 abr. 2022.

Martin, R. C. (2018). *Clean Architecture*: A Craftsman's Guide to Software Structure and Design. Pearson Education, Inc.

Microsoft (2022). *TypeScript is JavaScript with syntax for types*. Disponível em: https://www.typescriptlang.org/. Acesso em: 25 abr. 2022.

Mozilla (2021). *Cross-Origin Resource Sharing (CORS)*. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/CORS. Acesso em: 01 jun. 2022.

Pressman, R. S. (2011). *Engenharia de Software*: Uma Abordagem Profissional. 7ª Edição, AMGH Editora Ltda (AMGH Editora é uma parceria entre Artmed Editora S.A. e McGraw-Hill Education).

PostgreSQL (2022). *About - What is PostgreSQL?* Disponível em: https://www.postgresql.org/about/. Acesso em: 06 out. 2022.

Sousa, H. C., Silva, B. R. A. e Bandeira, J. M. (2017). *Uso de Api Restful para Gestão de Eventos*. In: Encontro Anual de Pesquisa e Iniciação Científica, 2017, Balsas. Encontro Anual de Pesquisa e Iniciação Científica. Disponível em: http://www.unibalsas.edu.br/wp-content/uploads/2017/01/Hitalo.pdf. Acesso em: 19 mar. 2022.

Wavy (2021). *Fallback API*. Disponível em: https://docs.wavy.global/documentacao-tecnica/todas-as-integracoes/fallback-api. Acesso em: Acesso em: 26 abr. 2022.