Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

PROJETO E ARQUITETURA DE API REST PARA SISTEMA DE MONITORAMENTO DE REDES ÓPTICAS

Lucas Campos Jorge

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. André Costa Drummond

Brasília
2020

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

PROJETO E ARQUITETURA DE API REST PARA SISTEMA DE MONITORAMENTO DE REDES ÓPTICAS

Lucas Campos Jorge

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. André Costa Drummond (Orientador)
CIC/UnB

Prof.a Dr.a Fernanda Lima
CIC/UnB

Prof. Dr. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. João José Costa Gondim
Coordenador do Curso de Engenharia da Computação

Brasília, 17 de dezembro de 2020

Dedicatória

À minha família, em especial meus pais, minha irmã e minha namorada.

Agradecimentos

A todos que me apoiaram e me auxiliaram durante a UnB. Principalmente meus pais, Anna Paula e Marcos e minha irmã Marianna que sempre me apoiaram, e minha namorada, Tatyanna, que passou por tudo ao meu lado. Meus melhores amigos Rafael, Pedro e Henrique. Além dos meus colegas de faculdade Andre, João Marcelo, Patrick e João Vitor que fizeram tudo ser mais fácil.

Ao meu orientador André Drummond, que me ofereceu esse projeto e seu suporte desde de seu início, além dos funcionários da GigaCandanga que nos ajudaram na realização do projeto.

Resumo

Esta monografia descreve o projeto e desenvolvimento de um serviço Web disponível através de uma API REST para controle de dados de um sistema de monitoramento de redes ópticas. A solução tem perspectiva de uso real e portanto tem como objetivo entregar uma aplicação confiável e robusta que soluciona o problema proposto. Para a construção da solução, apresentam-se desenvolvimento guiado por boas práticas de desenvolvimento e engenharia de software. Os desafios enfrentados na construção da aplicação ainda permitem futuros refinamentos e melhorias com diferentes infraestruturas e arquiteturas.

Palavras-chave: REST, Redes ópticas, API, Serviços Web

Abstract

This paper presents the project and development of a Web service available through a REST API that aims to control data from an optical network monitoring system. The solution has a real use perspective and therefore aims to deliver a reliable and robust application that solves the proposed problem. The presented development is guided by good development and software engineering practices. The challenges faced in building the application still allow for future refinements and improvements with different infrastructures and architectures.

Keywords: REST, Optical fibers networks, API, Web Services

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

API Application Programming Interface.

GORM Grails Object Relational Mapping.

HATEOAS Hypermedia as the engine of application state.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

ICT Instituição de Ciência e Tecnologia.

IMP Object-relational Impedance Mismatch Problem.

JSON JavaScript Object Notation.

KML Keyhole Markup Language.

KMZ Zipped Keyhole Markup Language Files.

MCT Ministério da Ciência e Tecnologia.

OO Orientado a objetos.

ORM Object Relational Mapper.

REST Representational State Transfer.

RMM Richardson Maturity Model.

RNP Rede Nacional de Ensino e Pesquisa.

SGBD Sistema gerenciador de banco de dados.

SOAP Simple Object Access Protocol.

SQL Standard Query Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

XML Extensible Markup Language.

Capítulo 1

Introdução

1.1 Motivação

A Associação GigaCandanga[1] é responsável pela manutenção, operação e gestão da rede metropolitana do Distrito Federal, Rede GigaCandanga, criada no âmbito da iniciativa intitulada Redecomep, do Ministério da Ciência e Tecnologia (MCT), coordenada pela Rede Nacional de Ensino e Pesquisa (RNP), que tem como objetivo implementar redes de alta velocidade nas regiões metropolitanas do país servidas pelos Pontos de Presença da RNP.

A GigaCandanga gerencia uma rede de fibras ópticas de alta capacidade, integrando diversos institutos de ensino e pesquisa. Atualmente, as informações da rede são organizadas por um sistema descentralizado, utilizando arquivos de diferentes formatos e sem um protocolo de inserção e manutenção bem definido dos dados. Este sistema apresenta diversas vulnerabilidades, entre elas a integridade dos dados e risco de erros humanos nas inserções e manutenção da rede. Além disso, a própria natureza dos dados, sendo parte das informações a respeito de geolocalização, dificulta a manipulação e monitoramento da rede. Logo, a motivação deste trabalho consiste na necessidade de uma solução prática do problema apresentado com a construção de um novo sistema de gerenciamento de uma rede de fibras ópticas.

Dentre os desafios do sistema proposto há a construção de um banco de dados com informações geográficas sobre a localização de elementos físicos da rede óptica, a exemplo de uma caixa de emenda (elemento que envolve emendas das fibras ópticas), e o caminho percorrido por uma fibra. Outro desafio seria interligar esses elementos, construindo assim uma representação lógica próxima da representação no mundo real da rede óptica e suprindo as necessidades de administração da rede.

O trabalho apresentado é um esforço em conjunto com outros três alunos do curso de engenharia de computação, com o propósito final da implementação de um sistema para o

cadastro, gestão e manutenção de uma rede de fibras ópticas da associação GigaCandanga. O trabalho desenvolvido faz parte da frente do *back-end*. Para uma visão completa da solução, é sugerida a leitura de todos os trabalhos:

- Damaceno (2020) [2]
- Chianca (2020) [3]
- Chaves (2020) [4]

No ponto de vista do desenvolvimento, foram criadas duas equipes, uma com o propósito de desenvolver a interface do usuário e outra de desenvolver a aplicação de manipulação dos dados do sistema de gestão da rede, denominadas de *front-end* e *back-end* respectivamente.

Uma esquematização geral do sistema pode ser observada na Figura 1.1. Nela se observa uma visão ampla do funcionamento da aplicação, bem como as conexões dos módulos que resultam no funcionamento total do sistema.

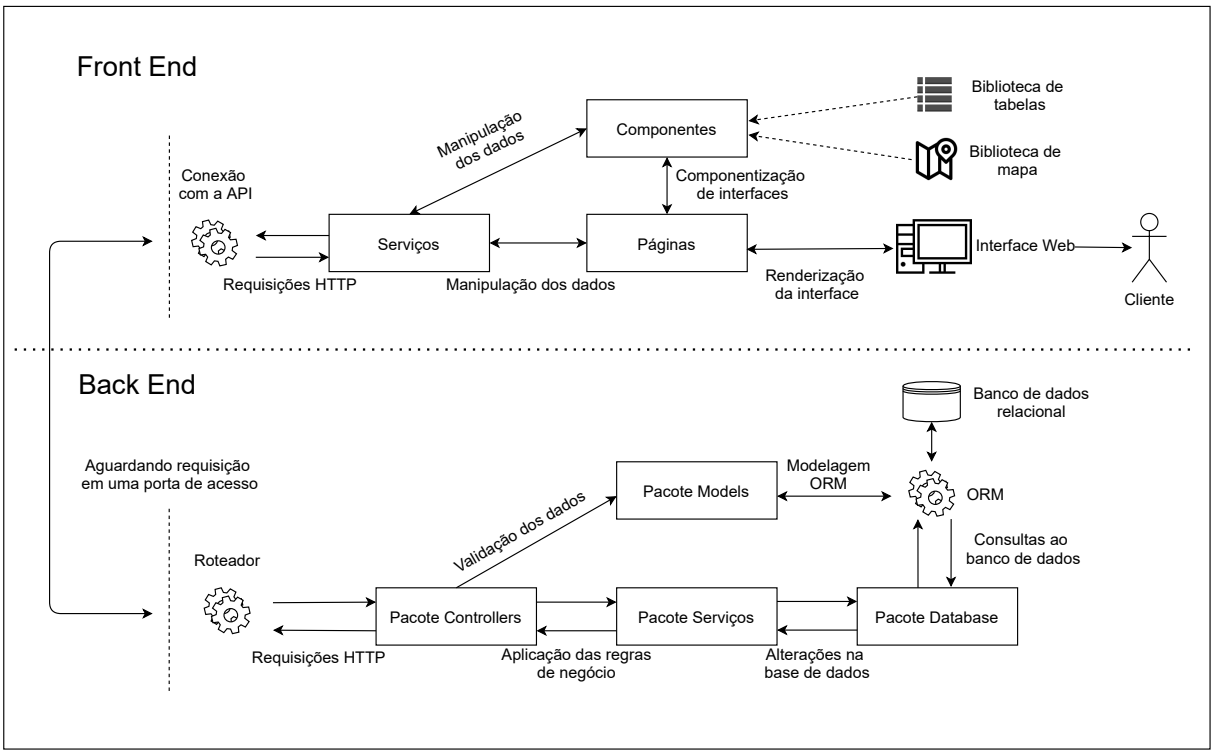

Figura 1.1: Esquemático do sistema.

No caso *back-end*, este trabalho, de modo geral, foca na arquitetura de software desenvolvida e apresenta discussões sobre a implementação de um design REST em uma API, aplicação de padrões de projeto em APIs, além descrever soluções de infraestrutura baseada em contêineres.

1.2 Objetivos

O sistema proposto visa automatizar o armazenamento, controle e monitoramento de dados referentes à administração de uma rede óptica da empresa GigaCandanga. Para tal problema, foi montado um time de desenvolvedores para desenvolver uma aplicação web, no qual me coube resolver a parte do *back-end*. Como solução do setor de *back-end* foi proposto o desenvolvimento de uma *Application Programming Interface (API)* que servirá como interface para os desenvolvedores do *front-end* acessarem as informações da rede e assim desenvolverem a interface visual que o usuário final utilizará.

A API será o principal serviço de comunicação e manipulação dos dados da rede. Logo, o desenvolvimento deve sempre ter como objetivo utilizar boas práticas e convenções de projeto, facilitando o seu uso e provendo as funcionalidades necessárias para quem consumir a API.

1.2.1 Objetivo Geral

O objeto deste projeto é uma solução de arquitetura de serviço web escalável, simples e robusta para um sistema de gestão e monitoramento de redes ópticas. A solução deve envolver padrões de projeto e decisões de engenharia de software compatíveis com o desenvolvimento de uma API REST atestando sua praticidade de desenvolvimento e de implementação.

1.2.2 Objetivos específicos

- Projetar um serviço web simples e prático utilizando uma API REST;
- Verificar a praticidade do design REST para desenvolvimento de APIs;
- Projetar uma arquitetura de software escalável e de fácil manutenibilidade
- Aplicar soluções de conteinerização para criação de uma infraestrutura de fácil implementação;
- Verificar o uso de padrões de projeto que auxiliem na modularidade do software;
- Desenvolver um serviço web robusto e de simples uso para um sistema de monitoramento e gestão de redes ópticas;

1.3 Requisitos obtidos

A solução proposta para o sistema se baseia na construção de uma API com o design REST, a qual utilizará o protocolo HTTP em um servidor e se comunicará com a base de

dados, provendo assim uma interface com rotinas e padrões bem definidos e documentados para manipulação das informações da rede óptica.

1.3.1 Servidor HTTP

O servidor HTTP deve ser responsável por utilizar o protocolo Hypertext Transfer Protocol (HTTP) como método de autenticar e transportar as informações de cada requisição e entrega de dados. Com a utilização do HTTP como protocolo de transmissão dos dados, permite-se enviar vários tipos de dados dentro de um pacote. Como o objetivo é construir um sistema com interface *web*, um dos tipos de dados mais utilizados é JavaScript Object Notation (JSON), que permite com facilidade o envio de objetos *javascript* com definição de dicionários indexados por chave-valor.

1.3.2 API REST

A API utilizada deve definir o protocolo de requisições e respostas que o servidor HTTP deverá entregar e receber. Neste caso, há diversos padrões de definição de APIs. Dentre eles há a API REST que, entre seus princípios de regras de interface, faz uso dos métodos HTTP para definir ações que o usuário que a utiliza deseja realizar, como: GET, POST, PUT, PATCH e DELETE. Cada método exerce ações correspondentes sobre uma entidade do sistema. A criação de uma API utilizando o protocolo HTTP facilita a independência dos componentes dos sistemas e consequentemente a proteção dos dados. Além disso, facilita a comunicação entre diferentes tecnologias, tornando invisível ao usuário da API qual tipo de servidor, *frameworks* ou banco de dados estão sendo utilizados no *back-end*.

1.3.3 Base de Dados

O banco de dados definido para o sistema deve possuir a capacidade de armazenar a hierarquia e relacionamento dos elementos, tanto para elementos triviais quanto para aqueles com localização geográfica. Bancos relacionais possuem a capacidade de relacionar tabelas entre si através de índices, possibilitando a configuração da rede. Isto, em conjunto com a capacidade de inserir tipos de dados de geolocalização como pontos e caminhos com latitudes e longitudes, satisfaz o objetivo proposto.

1.4 Organização do trabalho

Esta monografia está dividida em 6 capítulos. O segundo capítulo contempla o referencial teórico e os conceitos aplicados no desenvolvimento da arquitetura apresentada. O

terceiro capítulo comenta sobre os requisitos e a modelagem do sistema. O quarto capítulo aborda a arquitetura proposta, com a descrição de seu funcionamento e conceitos do capítulo anterior sendo aplicados. O quinto capítulo apresenta o sistema construído, com detalhamento das ferramentas utilizadas e decisões de implementação. O sexto capítulo conclui a monografia, incluindo considerações finais sobre a solução desenvolvida e propostas de melhorias futuras.

Capítulo 2

Conceitos

Este capítulo descreve os principais conceitos explorados durante o trabalho, tratando elementos presentes no desenvolvimento do software, sua arquitetura e tecnologias utilizadas.

2.1 Servidores HTTP

Servidores HTTP consistem em máquinas que provêm algum serviço utilizando como protocolo de transferência de informação o protocolo de rede HTTP. O Hypertext Transfer Protocol (HTTP) é um protocolo de rede da camada de aplicação amplamente utilizado em serviços na Web.

Além do seu objetivo mais comum de ser utilizado para *websites*, o protocolo também é utilizado para outros propósitos, tirando proveito principalmente da sua compatibilidade e uso já consolidados em navegadores de internet. Portanto, surgem assim mais servidores HTTP com o propósito de prover serviços para aplicações *web*. Estas, por sua vez, possuem a intenção de reproduzir o que aplicações presentes em um sistema operacional são capazes, dentro de navegadores na *web*. Logo, os servidores HTTP podem prover acesso e controle de banco de dados, autenticação, dentre outros serviços.

2.1.1 Docker

Há diversas formas de executar um servidor HTTP: executar nativamente em uma máquina com um sistema operacional compatível, executar em uma máquina virtual ou ainda contratar um serviço em nuvem. Uma alternativa comum é a de se executar uma máquina virtual em um sistema operacional leve em uma máquina de alta performance. Deve-se observar, no entanto, que há a possibilidade de ocorrer problemas de dependências e uso excessivo de recursos para executar um sistema operacional em cima de outro.

Partindo desta premissa, uma alternativa é a utilização de serviços como o Docker. O Docker é uma plataforma que permite a virtualização com menos uso de recursos, pois reaproveita o kernel do sistema operacional da máquina hospedeira e executa somente os recursos necessários para executar a máquina virtual. Ou seja, realiza a virtualização a nível do sistema operacional compartilhando recursos entre aplicações. Sendo assim, cada virtualização é contida em componentes chamados de contêineres que podem compartilhar recursos entre si e com a máquina hospedeira, ilustrados na Figura 2.1.

Cada contêiner é criado a partir de uma Imagem Docker e mantêm as características e aplicações instaladas definidas na imagem e evitando problemas de dependências de uso excessivo de recursos, pois o ambiente se torna mais controlável. Por exemplo, se existe a demanda de rodar dois servidores com versões diferentes de algum *framework*, basta executar um novo contêiner com uma imagem mais recente.

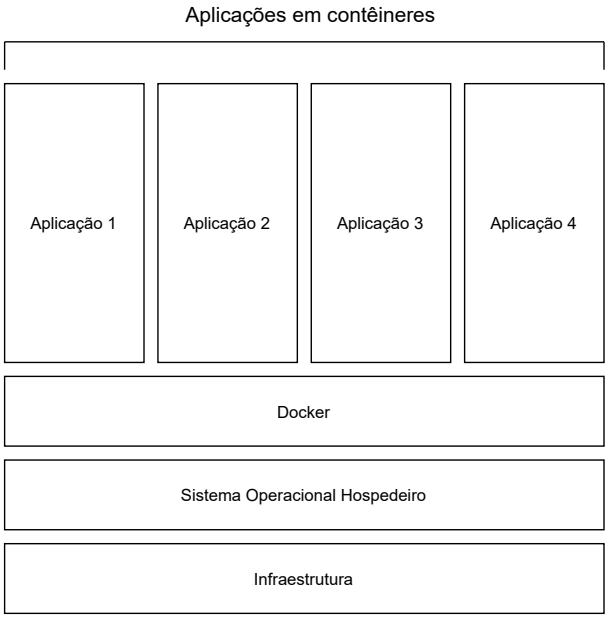

Figura 2.1: Representação do Docker (Fonte: [5]).

2.2 Representational State Transfer

Fielding (2000) [6] foi o primeiro a citar Representational State Transfer (REST) em sua tese de Doutorado, onde apresenta os desafios e problemas de escalabilidade na infraestrutura da *Web* e propõe arquiteturas de software para criação de interfaces com maior escalabilidade. Portanto, a partir de outras arquiteturas de software, Fielding (2000) [6], no capítulo 5 de sua tese, define seis restrições que governam a escalabilidade da

Web e consequentemente compõem o REST, um modelo de design de arquiteturas. Tais restrições foram agrupados em 6 grupos:

1. Client-server;
2. Uniform Interface;
3. Layered System;
4. Cache;
5. Stateless;
6. Code-on-demand;

2.2.1 Client-Server

Esta restrição define a separação de responsabilidades entre cliente e servidor. De acordo com Fielding (2000) [6], a separação entre a interface e a base de dados provê escalabilidade dos sistemas e a portabilidade da interface. Massé (2012) [7] acrescenta que esta separação deve ser realizada em conjunto com a definição de uma interface uniforme de comunicação da *Web*.

2.2.2 Uniform Interface

Como se trata de uma arquitetura dividida em componentes da *Web*, há a necessidade de que tais componentes se comuniquem de maneira padronizada. Fielding (2000) [6] afirma que um dos diferenciais da arquitetura REST está em prover uma interface uniforme entre componentes, na qual generaliza a interface e permite que cada componente seja desacoplado de acordo com o serviço que provê. Logo, o sistema como um todo torna-se altamente escalável. Porém Fielding (2000) [6] ressalva que esta interface uniforme pode em alguns casos limitar a eficiência da comunicação por não permitir outros formatos. Uma aplicação pode, por exemplo, necessitar de outro formato compatível ou mais eficiente.

A interface uniforme de comunicação entre os componentes possui quatro restrições, de acordo com Fielding (2000) [6]. Sendo eles:

1. *Identification of resources*;
2. *Manipulation of resources through representations*;
3. *Self-descriptive messages*;
4. *Hypermedia as the engine of application state (HATEOAS)*;

A Figura 2.2 ilustra os diferentes componentes do design de arquitetura REST em um diagrama. O diagrama posiciona os diferentes componentes em relação às requisições dos clientes, elucidando as etapas que cada requisição deve percorrer para a entrega de um recurso.

Identification of resources

Cada componente ou serviço distinto na Web é definido como recurso e pode ser endereçado por um Uniform Resource Identifier (URI), ou, identificador único de recurso. Cada recurso deve ser acessível pela *internet* através de um URI para cada informação que se deseja transmitir. Um exemplo simples e atual seria um URL como *https://google.com* que, a partir desse endereço, permite o acesso ao Google pela internet.

Manipulation of resources through representations

Esta restrição ratifica a necessidade de representações para manipulação de objetos, ou seja, um recurso pode ser apresentado de diferentes formas. Massé (2012) [7] afirma que este conceito permite diferentes interações com diferentes clientes, mas sem alterar o recurso ou seu identificador. Como exemplos de representações temos o HTML, JSON ou XML.

Self-descriptive messages

Cada requisição feita por um cliente pode conter em si uma informação sobre o estado do recurso que deseja obter ou alterar. Estas informações podem definir o próximo passo que o servidor irá realizar com a requisição. Tais informações podem ser o tamanho de um arquivo, a linguagem de um texto ou formato dos dados.

Hypermedia as the engine of application state (HATEOAS)

Cada recurso pode ter dentro de si endereços de outros recursos. Assim, os próprios dados ou mídias que representam um estado de um recurso podem referenciar outros recursos, guiando o uso pela interface. Um exemplo simples e prático são links presentes em páginas da web.

2.2.3 *Layered System*

Esta restrição propõe uma arquitetura de camadas com hierarquias, onde cada camada serve como intermediadora entre a comunicação do cliente e do servidor. Fielding (2000) [6] define que estas camadas devem ter comportamento de componentes do sistema e não

"enxergarem" além das camadas que as intermedeiam, restringindo a complexidade do sistema e promovendo independência entre os componentes. Tais camadas interceptam as requisições e proveem diversos serviços como segurança, validação e cache, por exemplo.

2.2.4 Cache

A utilização de cache em arquiteturas da *web* pode melhorar a eficiência do sistema, pois permite que certas requisições nem sejam processadas pelo servidor. Fielding (2000) [6] define que cada recurso será identificado como *cacheável* ou não *cacheável*. Um recurso presente na cache pode ser retornado diretamente pela cache para os clientes que realizam requisições semelhantes, evitando interações entre os componentes do sistema.

2.2.5 *Stateless*

Stateless determina que as requisições feitas por um cliente devem contemplar na mensagem todo o contexto necessário para manipular o recurso desejado, pois o servidor não deve armazenar o estado ou contexto dos clientes. Massé (2012) [7] conclui que esta restrição transfere a complexidade das requisições para os clientes, permitindo que o servidor lide com uma quantidade maior de clientes.

2.2.6 *Code-on-demand*

Code-on-demand refere-se a definição de tecnologias que permitem a transferência de código executável entre o cliente e o servidor. Na *internet* atualmente, o exemplo mais comum são *scripts* de Javascript presentes em diversos *sites* e antigamente o Flash.

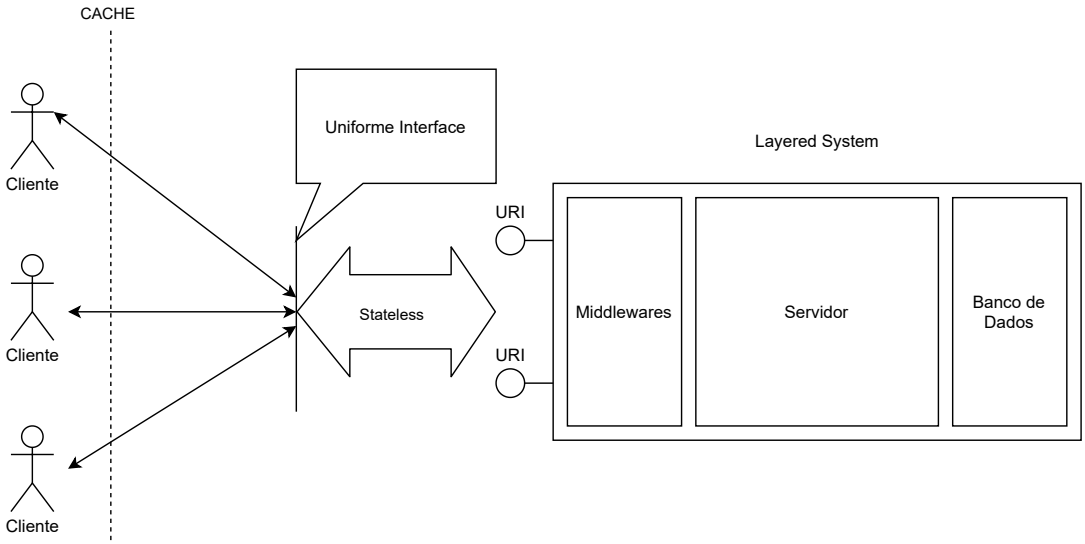

Figura 2.2: Design de arquitetura REST (Fonte: [8]).

2.3 Application Programming Interface

A construção de um sistema Web passa por diversas etapas e escopos diferentes, sendo a separação desses escopos mais comumente nomeada de *Front-End* e *Back-End*. O *Front-End* é responsável pela interface e controle dos dados de entrada e saída e o *Back-End* é responsável por manipulação, armazenamento, consistência e segurança dos dados manipulados no sistema. Esta separação pode inclusive determinar a organização de um ambiente de trabalho e alocação de desenvolvedores em equipes.

Sendo organizadas desta forma, as equipes podem tomar decisões diferentes de tecnologias e soluções de arquitetura. As escolhas podem ser tão diversas que podem inclusive ser escolhidas tecnologias incompatíveis ou que exigem um esforço grande de adaptação. Estas decisões podem impactar no grau manutenibilidade do sistema e até restringir a implementação de futuras funcionalidades.

Uma solução para a problemática acima seria o desenvolvimento de uma aplicação intermediária, uma Application Programming Interface (API), que realiza a comunicação entre as duas entidades - tecnologias ou sistemas - abstraindo a necessidade de algum grau de compatibilidade. Uma API exerce esse papel de ponte e permite que o desenvolvimento do sistema avance sem que as equipes tenham conhecimento das ferramentas aplicadas em seus sistemas.

O desenvolvimento de uma API consiste em grande parte na construção de um protocolo de comunicação abstrato que deve ser fácil de utilizar e replicável para melhorias futuras da aplicação. Em se tratando de um sistema *Web* onde podemos utilizar o proto-

colo HTTP como protocolo de transporte das informações entre os elementos do sistema, exitem diversas arquiteturas já utilizadas no mercado. Dentre elas, temos como exemplo APIs REST e SOAP.

2.3.1 APIs REST

A arquitetura Representational State Transfer (REST), descrita na Seção 2.2, pode ser vista como um abstração da arquitetura *web* utilizada na construção de serviços *web*. A arquitetura REST é comumente utilizada para construção de APIs. Assim, APIs REST definem uma forma de comunicação simples e eficiente entre os componentes da web. Como REST não é padrão fechado, existem diversas forma de implementá-lo, entretanto ele possui o conjunto de princípios de design descritos na Seção 2.2.

APIs REST permitem uma interface simples que utiliza os métodos presentes no protocolo HTTP (GET, PUT, POST, DELETE, dentre outros) como meio manipulação de recursos. Sendo assim, permite um conjunto completo de operações CRUD *(Create, Read, Update e Delete*), por exemplo. Portanto, com este conjunto de operações, um serviço *web* RESTful consiste em um serviço web que pode ser acessado por uma API REST, como descrito em Massé (2012) [7].

2.3.2 Operações

Podem ser executadas diversas operações em recursos, sendo um recurso, de acordo com Fielding (2000) [6]:

> Um recurso é qualquer coisa que é importante o suficiente para ser referenciada como algo como si próprio. (Tradução Livre)[1]

A tabela 2.1 lista as operações mais comuns utilizando o HTTP no contexto de uma API REST e seus códigos de status de resposta do servidor. Esta tabela é uma versão estendida de uma tabela presente em Yellavula (2017)[9]. Os códigos de retorno são necessários para informar o cliente do status da sua requisição, pondo em prática o conceito de *stateless* apresentado por Fielding (2000) [6].

2.3.3 Padrões de projeto

Em Biehl (2007) [10], se afirma que REST não é uma arquitetura, mas sim uma forma de julgar arquiteturas, onde inclusive se compara com termo "orientado a objeto". Esta afirmação reforça a falta de regras rígidas de design de uma arquitetura considerada REST

[1] "a resource is anything that is important enough to be referenced as a thing in itself", Fielding (2000) [6]

Tabela 2.1: Comandos REST e suas ações.

Comando REST e URI	**Ação**	**Sucesso**	**Falha**
GET /recurso/id	Busca um recurso	200	404
GET /recurso	Busca um conjunto de recursos	200	404
POST /recurso	Cria um novo recurso ou conjunto de recursos	201	404, 409
PUT /recurso/id	Atualiza ou sobrescreve um recurso por completo	200, 204	404
PATCH /recurso/id	Modifica um recurso	200, 204	404
DELETE /recurso/id	Remove um recurso	200	404

além das definições descritas por Fielding (2000) [6]. Logo, ao se criar uma API REST, há uma certa liberdade de implementação em termos de engenharia de software. Neste caso, abre-se a oportunidade de utilizar padrões de projeto já presentes em outras soluções de software.

Padrões de projeto de acordo com Luecke (2018) [11]:

> Padrões de projeto são boas práticas formalizadas que o programador pode usar para resolver problemas comuns ao desenvolver uma aplicação ou sistema. (Tradução livre) [2]

Um dos padrões de projeto que provê uma interface aplicável em um serviço web é a camada de serviços.

Camada de Serviços

Camada de Serviços, citada em Fowler (2009) [12], consiste em uma interface capaz de realizar serviços para múltiplos usuários com diferentes interfaces. Segundo Fowler (2009) [12], a camada de serviços:

> Define a fronteira de uma aplicação com uma camada de serviços que estabelecem um conjunto de operações disponíveis e coordena a resposta de cada operação. (Tradução livre) [3]

[2] Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system., Luecke (2018) [11]

[3] Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation., Fowler (2009) [12]

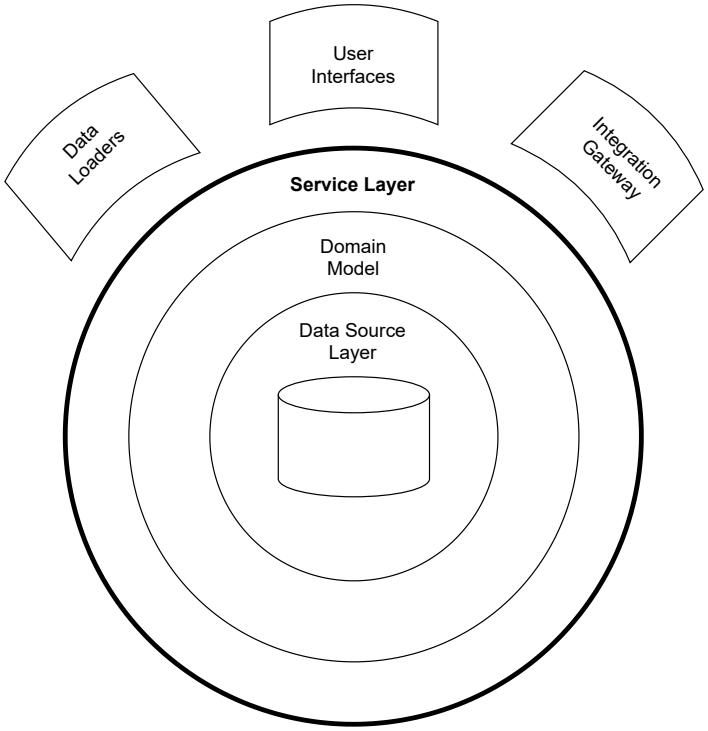

Figura 2.3: Camada de serviços (Fonte: [12]).

Nesse sentido, a camada de serviços permite que diferentes interfaces acessem e manipulem os dados de uma aplicação, porém requisitando de um conjunto finito de operações disponibilizados pela camada de serviço. Ou seja, a camada de serviços acessa a base de dados e os modelos da aplicação, trata os dados, aplica as lógicas de negócio da aplicação e retorna a resposta adequada para as interfaces que a requisitaram.

A Figura 2.3, ilustra a camada de serviços e sua característica de fronteira entre os modelos e fonte de dados de um sistema, e as diferentes interfaces que necessitam acessar a fonte de dados.

Segundo Luecke (2018) [11], a camada de serviços exerce no contexto de APIs diversas vantagens, dentre elas:

- **Separação de responsabilidades e testabilidade** - Separa a lógica da aplicação dos controladores e da interface de roteamento, permitindo assim testes somente da lógica da aplicação e não somente de integração.
- **Modularidade da camada de transporte** - Como a lógica da aplicação fica separada da lógica de transporte dos dados, permite-se a troca de protocolos de comunicação. Por exemplo, caso haja necessidade, pode haver a utilização de outros protocolos diferentes do HTTP.

2.4 Object-relational mapping

Object Relational Mapper (ORM) é uma solução de software que tem como objetivo reduzir a complexidade ao desenvolver uma aplicação de manipulação de um banco de dados relacional utilizando uma linguagem orientada a objetos. Como descrito em Torres (2017) [13], programação orientada a objetos e banco de dados relacional é baseada em paradigmas distintos e contém incompatibilidades técnicas, conceituais e culturais, onde tais discrepâncias são citadas como Object-relational Impedance Mismatch Problem (IMP). Uma das primeiras discussões a respeito dessa problemática foi apresentada em Copeland e Maier (1984) [14], onde foram discutidos os problemas sobre os bancos de dados orientados a objetos da época e as possíveis dificuldades de se implementar conceitos como polimorfismo, herança e associações, Torres (2017) [13].

O mapeamento mais comum realizado em ferramentas de ORM ocorre de forma que cada classe representa uma tabela e os registros são representados por instâncias destas classes. Com esta facilidade de representação, o ORM auxilia na criação de consultas ao banco de dados, pois cada consulta SQL é gerada a partir do framework ou biblioteca ORM utilizada. Logo, o banco de dados relacional é manipulado e construído na linguagem de programação orientada a objetos escolhida, sem a necessidade, na maioria dos casos, de se declarar comandos SQL. Dentre as dificuldades de se implementar uma solução de ORM em uma aplicação, podem ser observadas as diferentes sintaxes e nomenclaturas que cada *framework* ou biblioteca ORM possuem.

Vale ressaltar a problemática citada em Torres (2017) [13], na qual, ao utilizar mapeamento ORM, exige-se dos desenvolvedores, em certos momentos, tomadas de decisão que trocam problemas de design do banco de dados (normalização, chaves primárias e relacionamentos) por problemas de design dos modelos OO.

2.5 Modelo de maturidade de Richardson

O modelo de maturidade de Richardson, desenvolvido por Leonard Richardson e explicado em Richardson (2013) [15], apresenta uma classificação de maturidade de implementações do design REST. A proposta desta classificação é explicar como utilizar serviços web RESTful para lidar com problemas de integração geralmente encontrados no mercado [16]. Essa classificação, portanto, permite definir o nível de maturidade de uma API REST, por exemplo.

O modelo apresentado por Richardson define quatro níveis de maturidade o nível 0, 1, 2 e 3.

2.5.1 Nível 0

O nível 0 define a utilização do protocolo HTTP como método de transporte das requisições. A utilização do HTTP será feita para interagir com o sistema disponibilizando um *endpoint* em uma URI para manipular dados de algum recurso. Logo, um cliente que requisitar um recurso em um *endpoint* receberá do servidor uma resposta com os dados requisitados ou uma resposta de erro, caso não seja possível realizar a ação. Entretanto, é valido ressaltar que o design REST em si não define a utilização específica do HTTP, havendo a possibilidade de se utilizar outros protocolos de aplicação.

2.5.2 Nível 1 - Recursos

O nível 1 define a utilização de múltiplos *endpoints* onde através de cada *endpoint* é possível extrair individualmente um recurso. Em resumo, o nível 1 adiciona a capacidade de se passar a identidade de um recurso no *endpoint* e manipulá-lo diretamente. Por exemplo, o nível 1 permitiria obter um recurso específico através de um identificador (id) com o *endpoint* /recurso/id. Neste caso, os parâmetros enviados a mais na requisição manipulariam somente o recurso atribuído ao identificador escolhido.

2.5.3 Nível 2 - Verbos HTTP

O nível 2 define a utilização dos verbos HTTP para manipulação dos recursos. Logo, a interação do cliente com o servidor já é pré-definida pelo método HTTP da requisição, seja ele *GET, POST, PUT* ou *DELETE*, entre outros. A representação e papéis destes métodos podem ser vistos na Tabela 2.1. Um exemplo de benefício do nível 2 está no uso do método *GET*, que já impõe que a requisição não irá modificar ou apagar recursos, somente ler o mesmo.

2.5.4 Nível 3 - Controle de Hipermídia

O nível 3 define o uso da utilização do *Hypermedia as the engine of application state (HATEOAS)*, citado na Seção 2.2.2. O HATEOAS prevê o uso de *links* como guia da API. Neste sentido, cada recurso que possui referências ou conteúdo de outro recursos deve prover o *endpoint* que dá acesso a este outro recurso, desta forma adicionando um caráter de auto documentação da API e aplicando os conceitos da arquitetura REST descritas por Roy Fielding. A implementação do HATEOAS pode inclusive permitir que a API altere, ao longo do seu desenvolvimento e uso, as URIs e *endpoints* dos recursos sem prejudicar os usuários que utilizam os *links* para navegar pela API.

2.5.5 Rede de fibra óptica

O objeto de estudo de projeto é uma rede de fibras ópticas. Portanto, é revelante a descrição de alguns elementos físicos principais de uma rede como esta. Uma rede de fibras ópticas se caracteriza principalmente pelo uso de fibra ópticas e sua capacidade de conduzir luz, tal característica permite a transmissão de informação. Fibras ópticas em geral são reunidas em conjuntos de 36, 72 ou 144 formando um trecho de cabo (Figura 2.4). Como meio de travessia destas fibras são utilizados postes (Figura 2.6) ou caixas subterrâneas (Figura 2.5) que permitem uma passagem dos cabos tanto aérea quanto terrestre. Além destes componentes, um elemento primordial em rede de fibras ópticas são as caixas de emenda que são o local onde se realiza fusão de fibras ópticas. Outro elemento presente na infraestrutura deste tipo são as reservas técnicas que são cabos extras utilizados para a manutenção da rede.

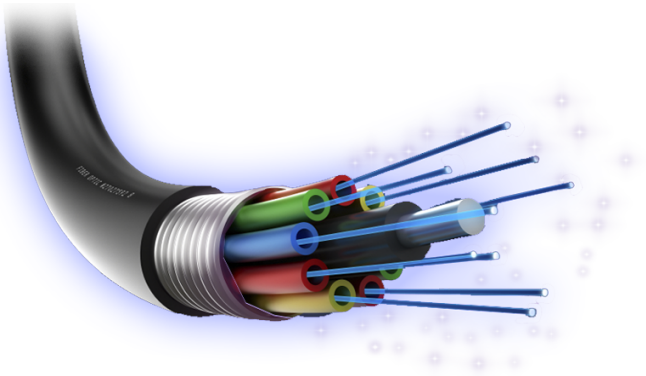

Figura 2.4: Exemplo de um cabo de fibra óptica (Fonte: [17]).

Figura 2.5: Exemplo de caixa subterrânea com uma caixa de emenda instalada (Fonte: [18]).

Figura 2.6: Exemplo de poste com uma caixa de emenda e reserva técnica instaladas (Fonte: [18]).

2.6 Resumo conclusivo

Neste capítulo foram apresentados diversos conceitos importantes para entendimento completo do trabalho, como o design REST, APIs e a camada de serviços. O design de arqui-

tetura REST apresenta uma forma simples de construir aplicações escaláveis e robustas em um contexto *web*. Além disto, APIs são aplicações que permitem a comunicação de diferentes sistemas e aplicações, sendo comumente aplicadas em sistemas *web*, onde é comum a separação do sistema em *front-end* e *back-end*. Por fim, a camada de serviços, que apresenta um padrão de projeto, o qual acrescenta modularidade em um projeto criando uma camada de fronteira em a base de dados e regras de negócio, das interfaces que desejam acessar a base de dados.

Tais conceitos serão revistos ao longo do trabalho e serão aplicados em uma solução de software. Esta solução apresentada foi desenvolvida após etapas importantes do projeto como a definição dos requisitos e a modelagem do sistema, tema do próximo capítulo.

Capítulo 3

Requisitos e Modelagem

3.1 Definição do problema

A Associação GigaCandanga é uma Instituição de Ciência e Tecnologia (ICT) que, dentre os seus objetivos, realiza a gestão da Redecomep. A Redecomep integra diversos institutos de ensino superior e pesquisa do Distrito Federal. A Redecomep possui uma infraestrutura própria de cabos ópticos com mais de 550 quilômetros de extensão, além de conter infraestrutura compartilhada com a GDF Net, a rede do Governo do Distrito Federal e a Infovia Brasília, a rede que conecta instituições do Governo Federal em Brasília. [19]

O sistema utilizado pela GigaCandaga para gerir a rede de fibras ópticas em sua responsabilidade não é automatizado e necessita de uma manutenção complexa de arquivos com diferentes formatos para armazenar e atualizar os dados da rede. Dentre estes formatos, é utilizado uma Keyhole Markup Language (KML) para armazenar os dados geográficos da rede, como localização de elementos físicos, cabos de fibras ópticas, postes e caixas de emendas. Os arquivos KML exigem uma manutenção constante e complexa dos dados, sem conexão lógica entre os elementos, somente suas localizações. No ponto de vista de armazenamento e conexão da rede em si, a gestão é feita a partir do armazenamento de arquivos Excel, por que se utilizam diversas tabelas para gerir as conexões da rede e de elementos como fibras e *switches*.

Essa organização não garante escalabilidade e consistência dos dados, sendo mais suscetível a erros humanos na manipulação dos dados. Além disso, dificulta a análise dos dados e exige um nível alto de familiaridade do usuário, ou gestor, com o sistema implementado nestes arquivos.

O objetivo do trabalho é desenvolver uma arquitetura de *back-end* escalável e que provê consistência dos dados para um sistema novo que irá substituir a gestão da Redecomep da GigaCandanga.

3.2 Entidades do sistema

O objeto de estudo principal do sistema proposto é uma rede de fibra óptica. A rede em si possui diversos elementos físicos, portanto o sistema deve representar todas as informações relevantes de cada elemento. Ao se tratar de uma rede, tais elementos se interconectam de diversas formas fisicamente, e estes relacionamentos também devem estar presentes no sistema. Além disto, a proposta inicial de interface seria a manipulação destes elementos por um mapa, onde deve ser possível a visualização e edição dos elementos, logo, necessita-se da manipulação de dados que representem a localização geográfica de cada um destes.

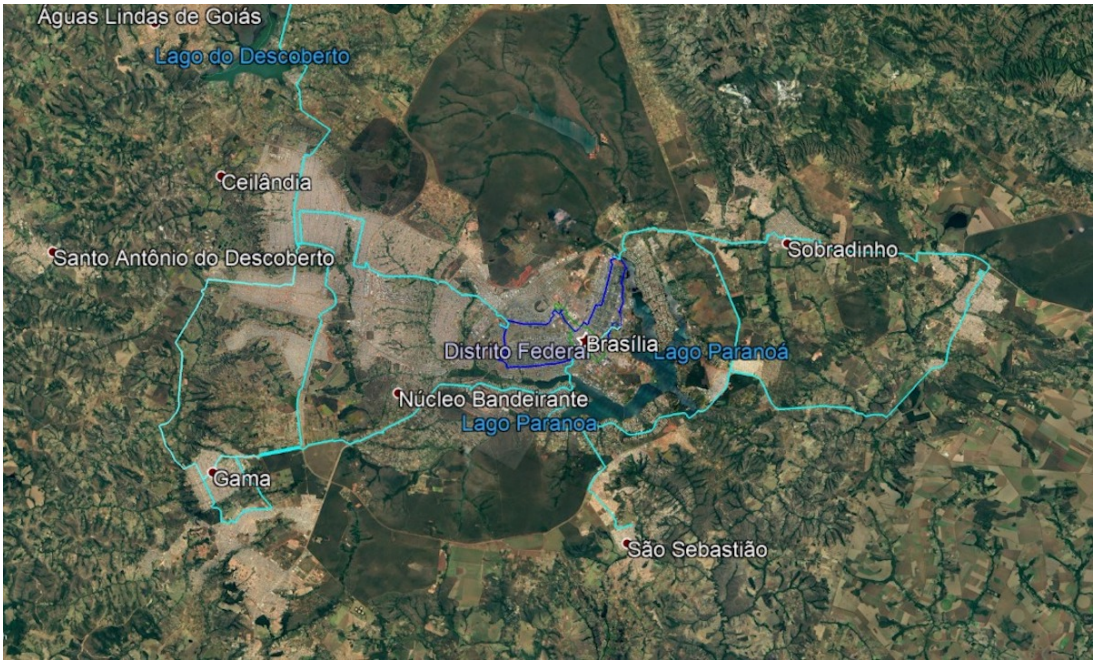

Figura 3.1: Mapa de satélite da Redecomep Gigacandanga (Fonte: [19]).

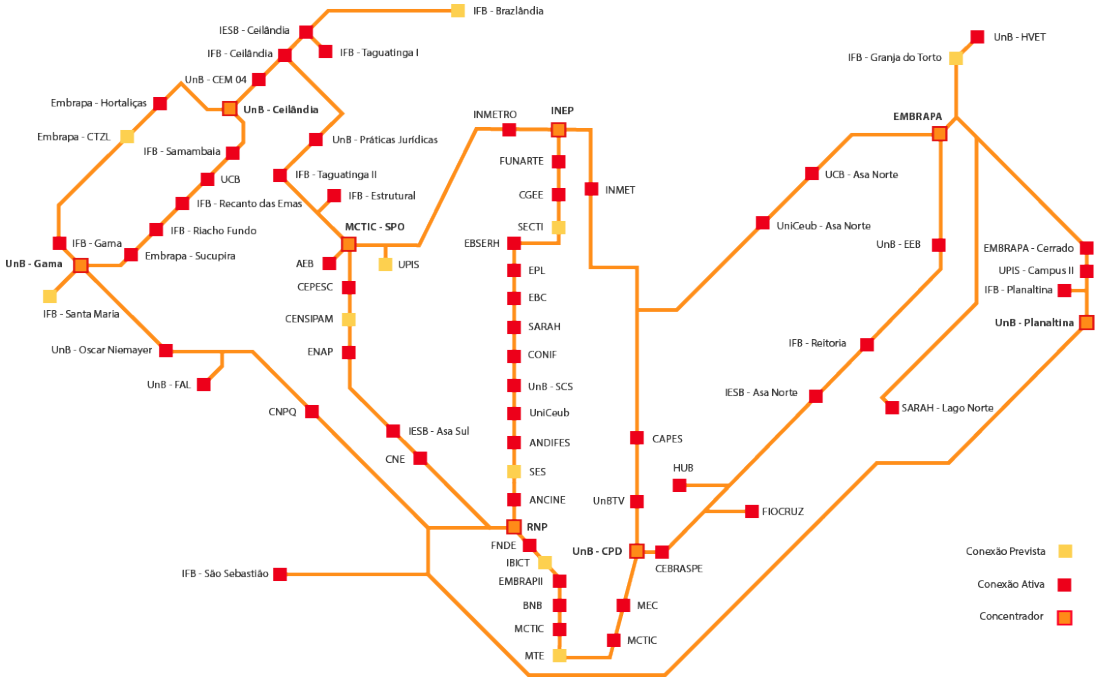

Figura 3.2: Esquemático da Redecomep Gigacandanga (Fonte: [19]).

Dentre alguns elementos encontrados em uma rede de fibra óptica, destacam-se as próprias fibras ópticas e caixas de emenda (locais onde se encontram as fusões das fibras).

3.3 Arquitetura do sistema proposto

O sistema proposto inicialmente deve permitir diversas operações de CRUD. Como proposta de banco de dados, após a decisão da equipe envolvida no projeto, foi definido o uso de um banco de dados relacional para armazenar todas as informações da rede. Devido à grande quantidade de entidades e relacionamentos complexos, houve um esforço nos primeiros meses para definir uma modelagem ideal do banco de dados. Esta modelagem contempla todos os elementos físicos de uma rede de fibra óptica. A modelagem mais atual encontra-se na Figura 3.3.

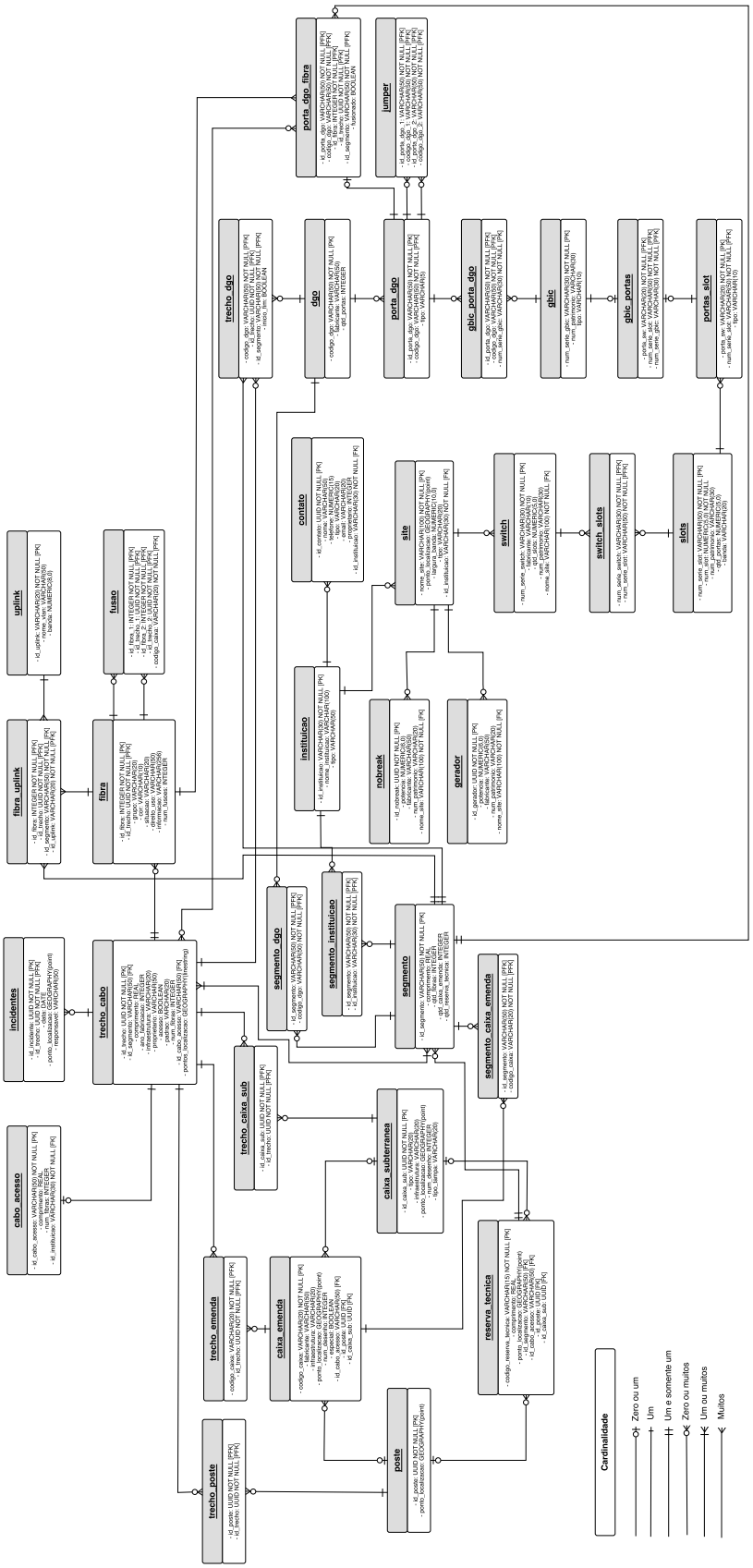

Figura 3.3: Modelagem do Bando de Dados.

A arquitetura do back-end se baseou no modelo de arquitetura REST, citado na seção 2.2. Esse modelo permite a utilização da arquitetura web para o controle dos dados. Tendo em vista que o sistema entregue pela equipe será um sistema web, a utilização do REST, principalmente, foi definida por praticidade de implementação e compatibilidade com o propósito do sistema como um todo. Além disso, a arquitetura REST permite a escalabilidade necessária para este projeto. Tendo REST como base, o *back-end* se comportará como um serviço *RESTful*, aplicando regras de design descritas por Fielding (2000) [6]. O serviço então estará disponível na web e poderá ser acessado por *endpoints* pela equipe de *Front-End* para manipulação dos dados da rede através de uma API.

3.4 Resumo conclusivo

Neste capítulo foram apresentadas de forma geral as etapas iniciais do desenvolvimento da solução de software. Primeiramente houve uma etapa de definição do problema da Associação GigaCandanga, que possui um sistema de gestão e monitoramento da rede de fibras ópticas descentralizado e baseado em arquivos, complexos de se realizar manutenção e propenso a erros e perda de dados. Além disto, foi apresentado os tipos de dados que serão armazenados, principalmente dados que possuem localização geográfica. Por fim, uma visão inicial da arquitetura proposta apresentando a modelagem e a decisão de se construir uma API REST. O próximo capítulo apresenta uma visão mais detalhada a arquitetura de software planejada para o desenvolvimento da solução.

Capítulo 4

Arquitetura

Tendo como objetivo o desenvolvimento de uma API REST escalável, simples e robusta, neste capítulo é apresentada a arquitetura planejada para o sistema, explorando decisões a respeito da base de dados, sintaxe da API e arquitetura do software.

4.1 Base de dados

A escolha de como armazenar a base da dados desse sistema é primordial para o desenvolvimento do software. A equipe em conjunto definiu que inicialmente a modelagem desenvolvida seria aplicada em um banco relacional, tendo em vista o caráter inicial do projeto de armazenamento e controle da rede. Neste caso, o Sistema gerenciador de banco de dados (SGBD) escolhido deve prover armazenamento de dados geográficos e permitir consultas em cima deste formato de dado.

O banco de dados então será acessado e manipulado através de uma API, que estará disponível pela internet e permitirá alterações e consultas dos dados, obviamente, seguindo as regras de negócio da aplicação.

4.2 API REST

A proposta para a API tem como molde o design de arquitetura definido por Fielding (2000) [6], ou seja, se propõe à construção de uma API REST. Desta forma, uma das definições atribuídas à construção da API é a definição dos URIs e utilização dos verbos HTTP, tirando proveito do design REST para facilitar a usabilidade da API.

A API pode ser vista como ponte entre a base de dados e a interface do usuário, entretanto também deve exercer uma camada de validação dos dados de entrada, aplicando tanto regras de negócio quanto verificação da qualidade dos dados. Esta camada de responsabilidade da API evita a construção de uma base de dados incorreta ou até

incompleta, além de evitar erros de consultas ou inserções no banco de dados inesperadas. A validação deve elucidar os erros de utilização da API, como inserção de dados incompletos, incorretos ou já existentes na aplicação, por exemplo.

O formato de dados escolhido para ser utilizado no corpo das requisições para envio e resposta de informações foi o JSON, devido à simplicidade de se manipular esse formato pela a equipe do *front end*, onde será utilizada uma biblioteca Javascript para renderização da telas da interface do usuário final.

4.3 Arquitetura de software

Ao se analisar os componentes que compõe a solução proposta, é necessária a construção de uma arquitetura que realize a comunicação entre eles. Podemos separá-los inicialmente da seguinte forma:

- Controlador de rotas e servidor HTTP,
- Módulo de requisições e respostas HTTP,
- Módulo de comunicação e consultas com o SGBD.

Uma análise inicial permite afirmar que o desenvolvimento destes componentes supre a necessidade em questão e permite a construção de uma API REST. Entretanto, do ponto de vista de modularidade, manutenibilidade e escalabilidade da solução de software há espaço para melhorias. Logo, se propõe uma arquitetura mais complexa com interligação de uma camada de controle, serviços, base de dados e modelos.

4.3.1 Camada de modelos

A camada de modelos deve conter a definição de todos os recursos que poderão ser acessados pela API, acrescentando no projeto uma melhor modularidade e manutenibilidade. Ela deve definir não apenas seus tipos de dados, mas também regras de validação e, como veremos posteriormente, o grau de abstração dos dados em relação à sua representação no banco de dados. Os modelos podem ser acessados tanto pela camada de controle, quanto pela camada de base de dados.

4.3.2 Camada de serviços

A camada de serviços visa implementar o padrão de projeto apresentado na Seção 2.3.3. Na camada de serviços estarão presentes todas as regras de negócio e lógica da aplicação, entretanto ela agirá como um intermédio entre a camada de controle do banco de

dados e a camada de controle, permitindo assim que a lógica seja inerente às tecnologias aplicadas nas outras camadas. Isto permite futuras alterações nas outras camadas sem exigir grandes correções na lógica do sistema. Sendo assim, adicionamos escalabilidade e manutenibilidade à solução, dois critérios importantes para o propósito do projeto.

A Figura 4.1 apresenta a disposição da camada de serviços em uma API, representando a separação de responsabilidades entre a camada de serviços e a camada que lida com as requisições HTTP, neste caso chamada de camada de transporte.

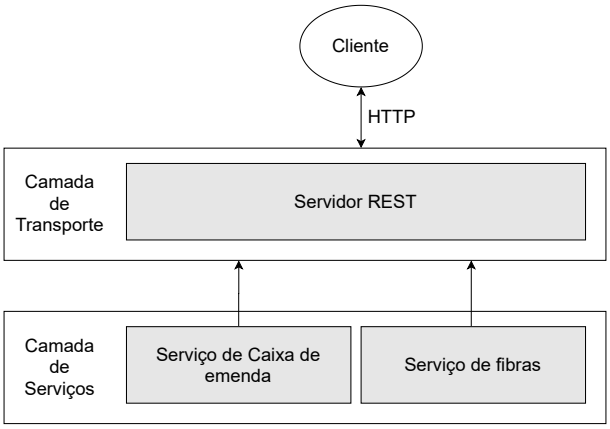

Figura 4.1: Uso da camada de serviços em uma API (Fonte: [11]).

4.3.3 Camada de controle

A camada de controle deve tratar do protocolo de comunicação utilizado. Neste caso, HTTP. A camada de controle define o comportamento da aplicação ao tratar das rotas e tratamento das requisições e respostas HTTP. Seu papel consiste em validar cada requisição, executar os serviços necessários da camada de serviços após a tratativa dos dados e logo após responder corretamente cada requisição utilizando os códigos de status HTTP corretos, seja de sucesso ou falha da requisição, além do conteúdo da resposta com mensagens de erro descritivas.

4.3.4 Camada da base de dados

A camada de base de dados possui o objetivo de prover funções de controle do banco de dados, ou seja, ela realiza de fato as consultas no SGBD, porém adiciona uma camada mínima de abstração entregando métodos que recebem e retornam elementos da camada de modelos. Desta forma, os métodos de manipulação da base de dados utilizados na camada de serviços se tornam independentes das particularidades do SGBD utilizado ou de como as consultas SQL são executadas.

No caso particular desta arquitetura é possível tirar proveito da camada de modelos e utilizar um ORM, citado na Seção 2.3.3. O ORM permitirá utilizar os modelos construídos como base das tabelas do banco de dados e diminuir a complexidade de implementação das consultas. Além disto, o ORM permitirá uma melhor consistência das informações armazenadas e uma melhor definição de como os dados são armazenados no banco de dados diretamente no código fonte.

4.4 Resumo conclusivo

Neste capítulo foi apresentado a arquitetura de software planejada para o desenvolvimento de uma solução de back-end aplicando em uma API REST. De modo geral, o foram descritas a necessidades para a escolha da base de dados e SGBD. Além disto foi apresentado a arquitetura da API REST que será implementada, citando principalmente as camadas de software implementadas, em especial a camada de serviços que aplica o padrão de projeto apresentado na Seção 2.3.3.

A arquitetura apresentada foi implementada de fato e durante seu desenvolvimento houveram diversas decisões de projeto relevantes, principalmente a escolha das ferramentas e tecnologias que foram utilizadas. O próximo capítulo apresenta a implementação da arquitetura descrevendo as ferramentas utilizadas e como elas se encaixam na solução.

Capítulo 5

Implementação

Este capítulo apresenta a solução de *back-end* implementada até o momento. Durante o capítulo serão discutidas as tecnologias utilizadas e como a arquitetura e infraestrutura apresentadas anteriormente se manifestam na implementação.

5.1 Tecnologias utilizadas

Durante a etapa de planejamento do projeto foram tomadas diversas decisões pelo time de desenvolvedores sobre quais tecnologias utilizaríamos. Uma das decisões inicias foi qual linguagem de programação utilizar. Por diversos fatores, dentre eles a experiência prévia de cada desenvolvedor e facilidade de implementação, a linguagem escolhida foi a linguagem de programação Go[20] (também conhecida como Golang).

O Go é uma linguagem de programação desenvolvida pelo Google em 2009, que propõe soluções simples e robustas, com um dos focos principais em programação concorrente. Uma outra vantagem da linguagem está na simplicidade em criar servidores HTTP com poucas linhas de código.

5.1.1 Framework Gin

O Gin [21] é um *framework web* de alta performance feito em Go que permite a criação de aplicações *web* e microsserviços. As principais vantagens de se utilizar o Gin estão na sua performance e facilidade de uso. O Gin permite a criação de um servidor HTTP e realiza o controle das rotas, recebimento e entrega de requisições, além de permitir a criação de *middlewares*.

Após a criação de um roteador com *framework*, ele executa as seguintes etapas: ao receber uma requisição HTTP na porta escolhida, ele fará a análise da rota acessada, comparará a rota da requisição com as rotas definidas no roteador e executará os *middlewares*

e a função escolhida para lidar com a requisição. Neste caso, as funções escolhidas são as funções da camada de controle que lidarão com cada requisição e sua resposta.

O Algoritmo 5.1 exemplifica a criação de rotas utilizando o *framework* Gin. Cada rota é descrita em roxo no algoritmo em conjunto com o método HTTP utilizado e como segundo argumento se define o método do pacote *controller* que será executado.

```
// Funcao que adiciona rotas de um determinado elemento
func AdicionaRotas(router *gin.Engine) {

  // Adiciona rota para obter todos os elementos
  router.GET("/elemento", controller.GetAll)

  // Adiciona rota para obter um elemento por identificador
  router.GET("/elemento/:id", controller.Get)

  // Adiciona rota para criar um elemento
  router.POST("/elemento", controller.Register)

  // Adiciona rota para editar um elemento por identificador
  router.PUT("/elemento/:id", controller.Edit)

  // Adiciona rota para apagar mais de um elemento
  router.DELETE("/elemento", controller.DeleteBulk)

  // Adiciona rota para apagar um elemento por identificador
  router.DELETE("/elemento/:id", controller.Delete)
}
```

Algoritmo 5.1: Exemplo de definição das rotas de um elemento utilizando o framework Gin

5.1.2 PostgreSQL

O PostgreSQL[22] é um gerenciador de banco de dados relacional *open-source*. O postgreSQL foi escolhido pelo time de desenvolvedores para ser utilizado como banco de dados principal da aplicação. Uma das vantagens de se utilizar este banco de dados é o uso de uma de suas extensões, o PostGIS. O PostGIS[23] define um conjunto de objetos de localização geográfica e possibilita realizar consultas de localização com SQL. Esta extensão auxilia no armazenamento dos dados geográficos da rede e permitiu, por exemplo, a criação do seguinte *trigger* para calcular o comprimento de um cabo a partir de um vetor de coordenadas geográficas armazenadas no banco de dados, apresentado no Algoritmo 5.2.

```
CREATE OR REPLACE FUNCTION path_len() RETURNS trigger AS
    $BODY$BEGIN
```

```
    NEW.length := ST_Length(NEW.line_string);
    RETURN NEW;
    END;
    $BODY$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS cable_section_path_len ON cable_section;

CREATE TRIGGER cable_section_path_len
    BEFORE INSERT OR UPDATE ON cable_section
    FOR EACH ROW EXECUTE PROCEDURE path_len();
```

Algoritmo 5.2: Exemplo de uso das Funções do PostGIS

5.1.3 GORM

O Grails Object Relational Mapping (GORM)[24] é uma biblioteca ORM feita em Go que possibilita relacionar os modelos descritos da aplicação às tabelas do banco de dados relacional. Além disto, o GORM permite realizar consultas no banco de dados SQL utilizando os modelos definidos na camada de modelos. Esta camada de ORM simplifica o desenvolvimento e permite que o código possua mais similaridade com a modelagem do banco de dados.

O GORM foi utilizado na camada de banco de dados para realizar as consultas necessárias solicitadas pela camada de serviços. No exemplo do Algoritmo 5.3, temos uma consulta SQL feita utilizando a biblioteca e no Algoritmo 5.4 temos um modelo definido com as *tags* de configuração do GORM para gerar o mapeamento da estrutura feita em Go com o banco de dados.

```
// InsertSpliceClosure : Insere uma nova caixa de emenda no banco de
   dados
func InsertSpliceClosure(spliceClosure *models.SpliceClosure)(err error)
{
    // db: instancia de uma conexao com o banco de dados
  err = db.Create(spliceClosure).First(spliceClosure).Error
  return
}
```

Algoritmo 5.3: Exemplo de consulta no Banco de Dados com o GORM

```
// Contact type definition
type Contact struct {
  ContactID     UUID         `gorm:"column:contact_id; primary_key; type
   :uuid; not null; default:uuid_generate_v4()"`
  Name          VARCHAR      `gorm:"column:name; type:varchar(50);
   default:null"`
```

```
12   Telephone      int            `gorm:"column:telephone; type:numeric(15,0)
      ; default:null"`
13   Email          VARCHAR        `gorm:"column:email; type:varchar(20);
      default:null"`
14   UpdatedAt      *time.Time     `gorm:"column:updated_at"`
15   CreatedAt      *time.Time     `gorm:"column:created_at"`
16 }
```

Algoritmo 5.4: Exemplo do modelo de um contato utilizando o GORM

5.1.4 Docker

O Docker, como já citado na Seção 2.1.1, cria contêineres para virtualizar máquinas de acordo com uma imagem. No projeto foi implementado um *docker-compose*[25], um arquivo que descreve um conjunto de contêineres que serão criados e como eles devem se comunicar. Esta configuração permitiu que se criasse três imagens: uma imagem para rodar o servidor HTTP com o *framework* Gin que terá acesso a uma porta externa; uma imagem do PostgreSQL para executar o SGBD e armazenar os dados; e por último uma imagem do PGAdmin, uma plataforma de manutenção e administração de bancos de dados PostgreSQL, que pode ser acessada também através de uma porta externa. A Figura 5.2 apresenta uma foto da interface do PGAdmin já com dados geográficos armazenados no banco de dados através da API.

Essas três imagens se comunicam através de portas internas do ambiente Docker, permitindo acesso do back-end desenvolvido ao Banco de Dados, por exemplo. A Figura 5.1 apresenta a arquitetura e disposição das imagens e as portas, internas e externas ao Docker, utilizadas para a comunicação dos contêineres.

No caso do banco de dados foi necessário implementar um volume no Docker. Volumes [26] são ferramentas disponibilizadas pelo Docker que implementam a persistência de dados, armazenando-os no sistema de arquivos da máquina hospedeira. Estes dados, através de um volume, podem ser acessados por um contêiner. O volume criado irá armazenar os dados do banco de dados do sistema e será acessado pela imagem do PostgreSQL. Esta implementação possibilita novas alterações na arquitetura desenhada no *docker-compose* e suas imagens sem a perda dos dados, além de facilitar futuras migrações.

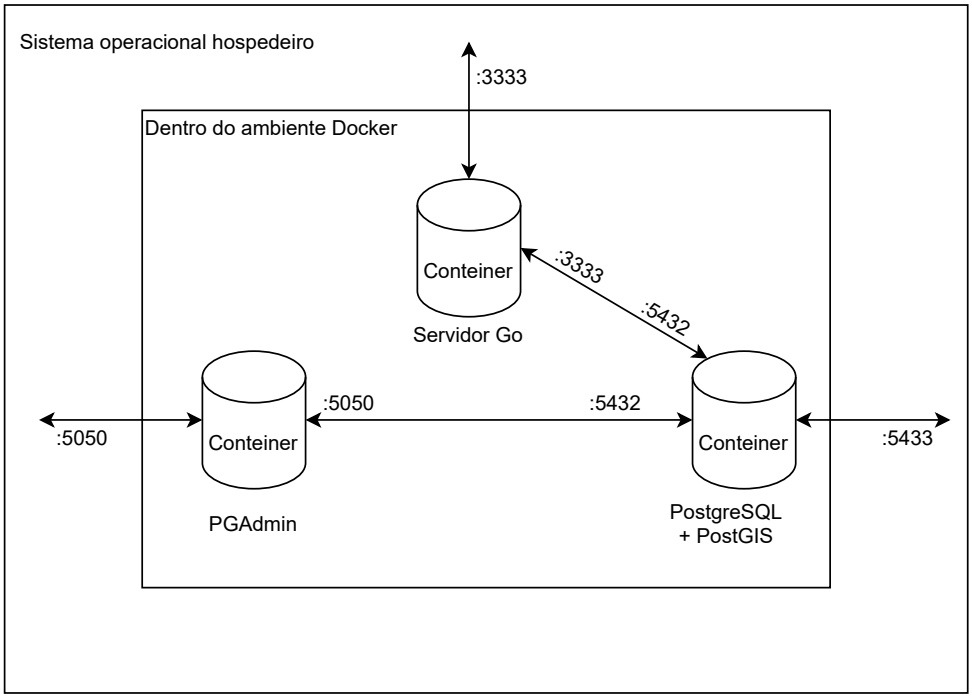

Figura 5.1: Arquitetura do Docker.

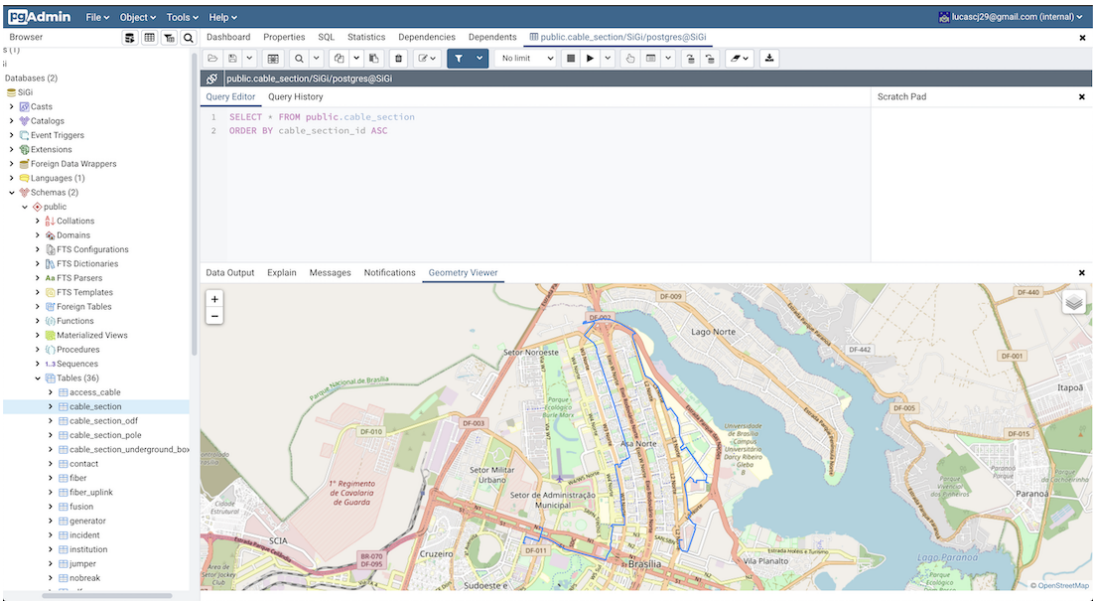

Figura 5.2: Interface do PGAdmin.

5.2 API

5.2.1 Arquitetura do software

O desenvolvimento do software durante o projeto teve como objetivo utilizar as tecnologias apresentadas na Seção 5.1 em conjunto com uma arquitetura de *software* simples, modular, escalável e de fácil manutenção. O software foi desenvolvimento em 4 pacotes escritos em GO: o *Database, Controller, Modals e Services.*

A Figura 5.3 apresenta um diagrama da arquitetura de software aplicada, um resumo das principais funcionalidades dos pacotes e suas respectivas interações com outros pacotes e ferramentas utilizadas.

Database

O pacote *Database* possui os principais métodos de manipulação de cada modelo de dados. O pacote utiliza as funções disponíveis pelo GORM que permitem a execução de consultas no banco de dados PostgreSQL do sistema.

Controller

O pacote *Controller* é responsável por lidar com cada requisição recebida pelo *framework* Gin, definindo as rotinas que serão executadas. Cada método descrito no *Controller* exerce as funções de tradução de JSON para uma estrutura em GO, realiza a validação dos dados, executa o serviço necessário e retorna para o Gin a devida resposta para a requisição.

Models

O pacote *Models* é o pacote onde estão descritos todos recursos acessíveis da aplicação em estruturas do GO. Ou seja, descreve os dados e seus tipos, que serão acessados e manipulados pela API e, consequentemente, devido à utilização do ORM, também define a estrutura dos dados no banco de dados. Esta dupla responsabilidade dos modelos unifica a representação dos dados e simplifica o desenvolvimento da aplicação, com uma representação única dos dados estruturados manipulados pela API e dos dados armazenados no banco de dados relacional.

Services

O pacote *Services* aplica o padrão de projeto descrito na Seção 2.3.3, que define uma camada de serviços. Foi implementado no pacote métodos que define as regras de negócio da aplicação independente das ferramentas utilizadas, como GORM ou Gin. Ou seja, caso

futuramente estas ferramentas sejam modificadas ou substituídas, a lógica da aplicação se mantem inalterada.

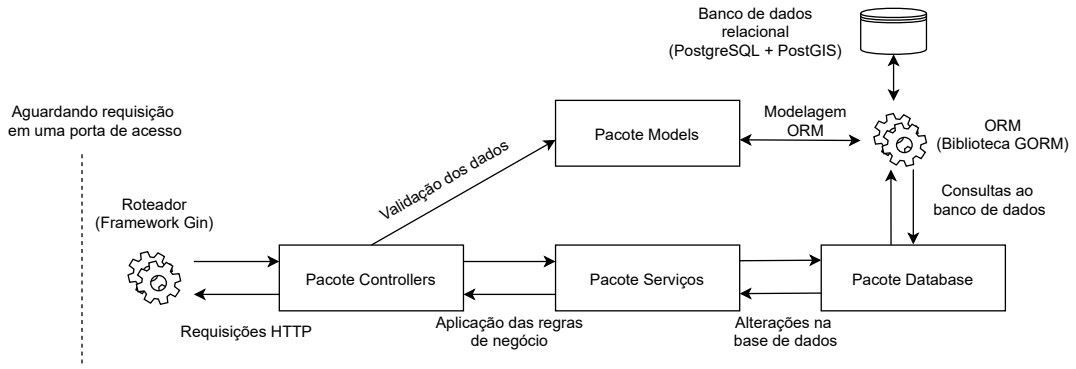

Figura 5.3: Arquitetura do software.

5.2.2 CRUD dos elementos

O monitoramento e gestão da rede exige diversas funcionalidades. O escopo do projeto vai além do tempo de desenvolvimento atual, portanto como prioridade definida pela equipe, o foco inicial está na CRUD dos elementos da rede. Para cada elemento foram utilizados os verbos HTTP e tiveram como guia de design da API o design REST. O desenvolvimento das operações de CRUD seguiu a padronização descrita abaixo, com o nome de cada elemento no *endpoints* em inglês.

Criando um elemento

A criação de um elemento pode ser realizada ao enviar uma requisição HTTP utilizando o método *POST* para o *endpoint* /elemento, com o elemento incluído no corpo da requisição um JSON com todas as informações do elemento a ser adicionado e o seu identificador. A resposta da requisição, em caso de sucesso, retorna o elemento criado em formato JSON.

Obtendo um ou mais elementos

Para se obter um elemento deve-se enviar uma requisição HTTP utilizando o método *GET* para o *endpoint* /elemento/identificador. A resposta da requisição, em caso de sucesso, retorna o elemento criado em formato JSON. Também foi implementada a listagem de todos os elementos, para tal, deve-se realizar uma requisição HTTP utilizando o método *GET* para o *endpoint* /elemento.

Editando um elemento

A edição de um elemento pode ser realizada ao enviar uma requisição HTTP utilizando o método *PUT* para o *endpoint* /elemento/identificador, adicionando no corpo da requisição um JSON com as novas informações do elemento. A resposta da requisição, em caso de sucesso retorna o elemento editado em formato JSON.

Apagando um ou mais elementos

Para apagar um elemento deve-se realizar uma requisição HTTP utilizando o método *DELETE* para o *endpoint* /elemento/identificador. Também foi implementada a ação de apagar diversos elementos, para tal, deve-se realizar uma requisição HTTP utilizando o método *DELETE* para o *endpoint* /elemento com os identificadores em um vetor dentro do corpo da requisição.

5.2.3 Validação das requisições

Uma funcionalidade importante implementada foi a validação dos dados inseridos nas requisições. O papel dessa validação é evitar a entrada de dados incorretos, seja devido ao tipo do dado ou pelo valor do dado inserido. Para implementar a validação foi utilizado o pacote Validator [27], ele permite a utilização de etiquetas nas estruturas do Go para definir diversos critérios de validação. O próprio pacote já retorna mensagens de erros contextuais em português de acordo com o validação aplicada, descrevendo qual campo e em qual critério o dado inserido falhou. As mensagens são importantes pois são inseridas nas respostas das requisições e podem ser utilizadas na interface para comunicar o usuário dos dados inseridos incorretamente.

```
17 type UndergroundBox struct {
18   UndergroundBoxID       UUID              `validate:"uuid4" ptBR:"ID
      da caixa subterranea"`
19   Type                   VARCHAR           `validate:"max=20" ptBR:"
      Tipo"`
20   Infrastructure         VARCHAR           `validate:"max=20" ptBR:"
      Infraestrutura"`
21   DrawNumber             int               `validate:"numeric" ptBR:"
      Numero do desenho"`
22   ManholeCoverType       VARCHAR           `validate:"max=20" ptBR:"
      Tipo da tampa"`
23   UpdatedAt              *time.Time
24   CreatedAt              *time.Time
25 }
```

Algoritmo 5.5: Exemplo de modelo com etiquetas de validação

No Algoritmo 5.5, é possível observar as etiquetas em roxo *validate* e *ptBR* que serão lidas pelas funções de *parser* do pacote. A etiqueta *validate* será interpretada pela biblioteca e determinará os critérios de validação, neste exemplo as etiquetas com os valores "uuid" e *"numeric"* são etiquetas que determinarão a tipagem que os dados devem possuir, já os valores "max=20" determinam que o dado inserido no determinado campo da estrutura deve possuir no máximo 20 caracteres. A etiqueta *ptBR* foi desenvolvida e implementada no método de tradução da biblioteca para indicar o nome traduzido dos campos, pois o código foi desenvolvido em inglês.

5.2.4 Importação de arquivos KML

Dentre as funcionalidades adicionais desenvolvidas, destaca-se a rota que permite a importação de arquivos KML. Um dos objetivos do projeto como um todo é unificar a base dados das redes de fibras ópticas da Associação GigaCandanga, portanto há a necessidade de permitir, sempre que possível e viável, a importação dos dados armazenados pela GigaCandanga pelo sistema antigo (que era descentralizado em arquivos). Estes arquivos KMZ eram utilizados para representar as posições geográficas de diversos elementos das rede de fibras ópticas e possuía uma organização própria do arquivo. Como o arquivo era manipulado diretamente através do Google Earth[28], havia diversas inconsistências na organização do arquivo, dificultando a implementação de uma rotina automatizada que lesse os dados desses arquivos e os armazenasse corretamente no banco de dados. Logo, este problema foi discutido com um técnico da GigaCandanga durante as reuniões com o cliente e foi definido um formato de organização dos dados dos arquivos KML. Com o formato definido, foi desenvolvido um *endpoint* que recebe na requisição o arquivo KML, ou KMZ (versão comprimida de um KML), processa os dados e armazena corretamente no banco de dados os elementos e seus relacionamentos entre si, além de suas posições geográficas.

A interface desenvolvida pela equipe de *front end* possui uma descrição da formatação desejada do arquivo KML para uma importação com sucesso. A Figura 5.4 é uma imagem desta descrição.

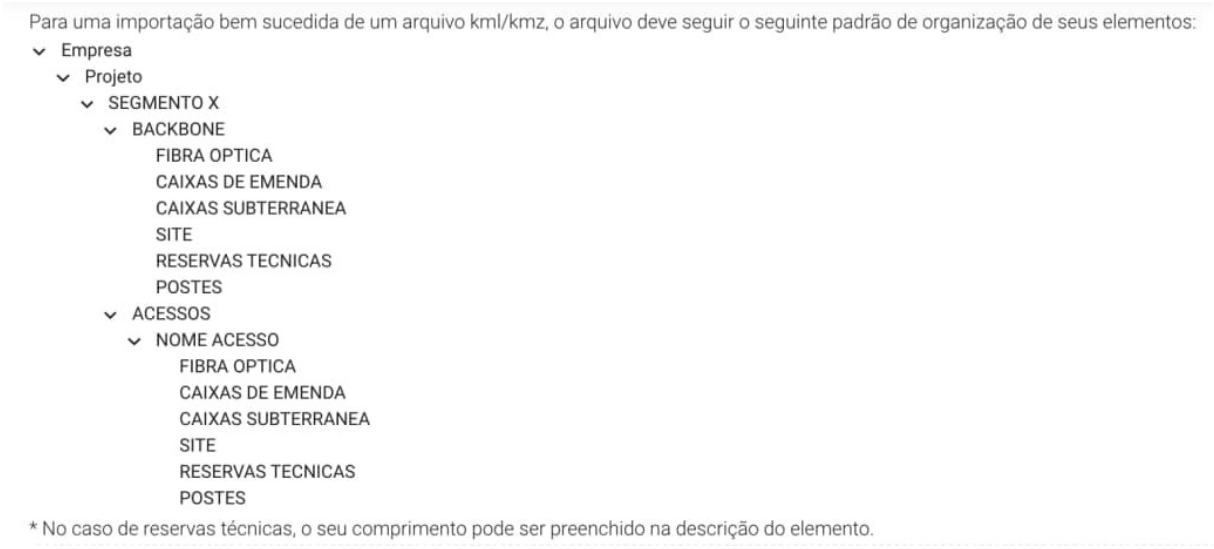

Figura 5.4: Descrição de formatação do arquivo KML.

5.3 Dimensão da aplicação

A respeito do desenvolvimento realizado a partir do trabalho apresentado se faz necessário apresentar uma visão de dimensão da aplicação. Segue abaixo dados sobre o desenvolvimento até a conclusão deste documento.

- 98 rotas desenvolvidas;
- 19 recursos diferentes;
- 27 modelos e seus relacionamentos implementados;
- Tratamento de variáveis de ambiente e linha de comando;
- 167 mensagens de erro diferentes.

5.4 Verificação de maturidade da API REST com RMM

A API implementada tem como uma de suas diretrizes de desenvolvimento o design REST, portanto é válida a sua avaliação utilizando o *Richardson Maturity Model (RMM)*, citado na Seção 2.5. A avaliação pode ser feita verificando cada nível presente no modelo RMM e verificando se a API desenvolvida implementou tais funcionalidades.

De modo geral, a API desenvolvida implementa as funcionalidades dos níveis 0, 1 e 2 do RMM. Ao utilizar o protocolo HTTP como pacote de manipulação dos recursos, a implementação do nível 0 se satisfaz. Os níveis 1 e 2 foram implementados com a distribuição dos recursos em diferentes *endpoints*, manipulação individual de cada recurso e utilização dos verbos HTTP como diretrizes da ação realizada pelo servidor.

Por último, o nível 3 não foi implementado no desenvolvimento, ou seja, a API não desfruta dos benefícios de utilizar o HATEOAS como guia do usuário pela API. A falta deste componente limita a API na modularidade das URIs que não podem ser alteradas sem prejuízos na sua utilização, arriscando a necessidade de se desenvolver uma nova versão da API. Esta funcionalidade deve ser implementada para que se alcance a implementação completa o design REST descrito por Roy T. Fielding, sendo até descrita por ele com essencial em APIs REST, Fielding (2008) [29].

5.5 Resumo conclusivo

O trabalho realizado para a construção da API possui o objetivo principal de unificar e simplificar a organização e manipulação dos dados referentes às redes de fibras ópticas sob responsabilidade da Associação GigaCandanga. A implementação descrita nas seções anteriores teve como diretrizes práticas consolidadas de desenvolvimento de API, como o design REST, além dos cuidados para que o software seja simples de ser utilizado e de ser mantido. A partir da utilização do padrão de projeto da camada de serviços, que adiciona modularidade ao *software*, foram colocados em prática estes cuidados, que permitiram que a solução possua mais longevidade no ponto de vista de sua utilização pela GigaCandanga.

O projeto e implementação apresentados descrevem as decisões realizadas em um ponto de vista macro da aplicação. A grande quantidade de entidades e relacionamentos apresentados pela modelagem e a perspectiva de utilização real do software exigiram ainda mais um cuidado com os aspectos de modularidade, simplicidade e robustez que são mencionados durante a descrição da solução implementada. A API como um todo foi desenvolvida a partir tanto de demandas do cliente diretamente quanto por demandas da equipe de *front-end* para possibilitar ações na interface para o usuário final. Um exemplo que pode ser citado é o *endpoint* que realiza a exclusão de múltiplos elementos. Por fim, a API continuará em desenvolvimento e há diversos aspectos que devem ser revistos futuramente em seu desenvolvimento, incluindo a adaptação da API para satisfazer o nível 3 do *Richardson Maturity Model (RMM).*

Capítulo 6

Conclusão

Este capítulo apresenta as considerações finais e reflexões sobre o desenvolvimento do projeto com um todo, além de apresentar melhorias futuras necessárias ao projeto.

6.1 Considerações finais e Melhorias futuras

O trabalho desenvolvido apresenta o desenvolvimento de um sistema web, discutindo decisões de projeto e, principalmente, como a arquitetura do *software* influencia em sua longevidade e escalabilidade. Em um projeto desta magnitude é primordial garantir desde seu início simplicidade, modularidade e manutenibilidade da solução. A perspectiva é do desenvolvimento do software continuar após realização deste trabalho, portanto as decisões farão impacto no futuro. O design REST adicionou simplicidade à solução e permitiu o desenvolvimento de uma API robusta e escalável em um contexto complexo como é o de uma rede de fibra óptica, porém há possibilidade de melhorar a implementação do design REST adicionando o *Hypermedia as the engine of application state (HATEOAS)* à solução, como descrito no RMM. A adição do Hypermedia as the engine of application state (HATEOAS) adiciona um caráter de auto-documentação do software e permite uma flexibilidade na definição de *endpoints*. O padrão de projeto implementado com a camada de serviços adicionou no projeto modularidade e escalabilidade, condizente com os objetivos do trabalho. Além disso, a utilização do Docker simplificou a infraestrutura necessária para implementação e permitiu a criação de uma ambiente isolado para o execução do serviço.

A utilização de REST nesta aplicação ainda pode ser alterada e revista no futuro do projeto, tendo em vista o surgimento de outras arquiteturas de *back-end* como a de microsserviços que dividem a aplicação em diversos serviços que se comunicam através de APIs. Outra alteração possível pode ser a utilização de GraphQL[30], uma linguagem de consultas de APIs que permite requisições mais complexas com múltiplas ações em uma

única requisição, diferente das APIs REST onde cada requisição realiza somente uma ação.

O desenvolvimento irá se estender após este trabalho, logo é possível a análise de melhorias futuras necessárias e previstas no desenvolvimento. O principal requisito não-funcional a ser implementado é uma solução de cache. Com a cache irá se implementar mais um elemento já previsto no design REST e permitir uma melhora de performance da aplicação. A solução de cache já prevista envolve o uso de um banco de dados não relacional para armazenamento de respostas à requisições. Até o momento da escrita deste trabalho se propõe a utilização de um banco de dados Redis[31]. Além da cache, também se prevê a adição de um banco de dados não-relacional para realização de rotinas de análise da rede óptica com utilização de algoritmos de grafos, onde banco de dados não relacionais são mais apropriados. Dentre os requisitos funcionais, funcionalidades importantes como filtragem e paginação serão de extrema importância com o crescimento da base de dados, facilitando consultas e otimizando tempos de resposta.

Referências

[1] *Gigacandanga*. `https://gigacandanga.net.br/`. Acesso em: 22/10/2020. 1

[2] Damaceno, Andre. O GERENCIAMENTO CADASTRAL DE UMA REDE DE FIBRAS ÓPTICAS UTILIZANDO UMA API REST: MODELAGEM E ARQUITETURA. Orientador: André Drummond. 2020. 48 f. Monografia (Graduação) – Engenharia de Computação, CIC, UnB, Brasília. 2020. 2

[3] Chianca, Rafael. INTERFACE WEB PARA SISTEMA DE GESTÃO DE REDES SEGUINDO PROJETO CENTRADO EM USUÁRIO. Orientador: André Drummond. 2020. 44 f. Monografia (Graduação) – Engenharia de Computação, CIC, UnB, Brasília. 2020. 2

[4] Chaves, João. Sistema de monitoramento de redes de fibra óptica: Desenvolvimento e projeto voltados à experiência do usuário. Orientador: André Drummond. 2020. 49 f. Monografia (Graduação) – Engenharia de Computação, CIC, UnB, Brasília. 2020. 2

[5] *Docker*. `https://www.docker.com/resources/what-container`, acesso em 2020-12-02. 7

[6] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures*, 2000. `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`. 7, 8, 9, 10, 12, 13, 24, 25

[7] Massé, Mark: *REST API Design Rulebook*, volume 2012. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2012. 8, 9, 10, 12

[8] Yadav, Akansha: *A look at rest api design patterns*, 2020. `https://dzone.com/articles/a-look-at-rest-api-design-patterns`. 11

[9] Yellavula, Naren: *Building RESTful Web Services with Go*, volume 2017. Packt Publishing Ltd., 2017. 12

[10] Biehl, Matthias: *RESTful Web Services*, volume 2007. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2007. 12

[11] Luecke, David: *Design patterns for modern web apis*, 2018. `https://blog.feathersjs.com/design-patterns-for-modern-web-apis-1f046635215`. 13, 14, 27

[12] Fowler, M.: *Patterns of Enterprise Application Architecture*. Bookman, 2009. 13, 14

[13] Torres, Alexandre, Renata Galante, Marcelo S Pimenta e Alexandre Jonatan B Martins: *Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design.* information and software technology, 82:1–18, 2017. 15

[14] Copeland, George e David Maier: *Making smalltalk a database system.* ACM Sigmod Record, 14(2):316–325, 1984. 15

[15] Richardson, Leonard: *Justice will take us millions of intricate moves*, 2008. `https://www.crummy.com/writing/speaking/2008-QCon/act3.html`. 15

[16] Fowler, Martin: *Richardson maturity model*, 2010. `https://martinfowler.com/articles/richardsonMaturityModel.html`. 15

[17] Cklcqovjou: *Cabo de fibra óptica.* `https://www.gratispng.com/png-jfshq3/`, acesso em 2020-12-21. 17

[18] GigaCandanga. [Imagem de arquivo da GigaCandanga]. 18

[19] *Associação gigacandanga.* `https://gigacandanga.net.br`, acesso em 2020-12-02. 20, 21, 22

[20] *Golang.* `https://golang.org/`, acesso em 2020-12-02. 29

[21] *Gin framework.* `https://github.com/gin-gonic/gin`, acesso em 2020-12-02. 29

[22] *Postgresql.* `https://www.postgresql.org/`, acesso em 2020-12-02. 30

[23] *Postgis.* `https://postgis.net/`, acesso em 2020-12-02. 30

[24] *Gorm.* `https://gorm.io/index.html`, acesso em 2020-12-02. 31

[25] *Overview of docker compose.* `https://docs.docker.com/compose/`, acesso em 2020-12-11. 32

[26] *Use volumes | docker.* `https://docs.docker.com/storage/volumes/`, acesso em 2020-12-11. 32

[27] *Pacote validator.* https://github.com/go-playground/validator, 2020. 36

[28] *Google earth.* `https://www.google.com.br/intl/pt-BR/earth/`, acesso em 2020-12-12. 37

[29] Fielding, Roy T.: *Rest apis must be hypertext-driven*, 2008. `https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`. 39

[30] *Graphql | a query language for your api.* `https://graphql.org/`, acesso em 2020-12-08. 40

[31] *Redis for dummies.* `https://redislabs.com/redis-for-dummies/`, acesso em 2020-12-08. 41