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

O GERENCIAMENTO CADASTRAL DE UMA REDE DE FIBRAS ÓPTICAS UTILIZANDO UMA API REST: MODELAGEM E ARQUITETURA

Andre G. Damaceno

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

Orientador
Prof. Dr. André Drummond

Brasília
2020

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

O GERENCIAMENTO CADASTRAL DE UMA REDE DE FIBRAS ÓPTICAS UTILIZANDO UMA API REST: MODELAGEM E ARQUITETURA

Andre G. Damaceno

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

Prof. Dr. André 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

A todos meus amigos e familiares.

Agradecimentos

A todas as pessoas que me apoiaram durante a vida e o curso, em especial Gabrielly, minha namorada, que me apoiou em todos os passos, Gizele e Vagner, meus pais, Guilherme e Juliana, meus irmãos, e também a todos meus amigos da UnB, em especial João Marcelo, João Victor, Lucas, Patrick e Rafael, que me acompanharam em todos os semestres e me ajudaram nessa jornada.

Ao Professor e Orientador André Drummond por me direcionar e prover suporte no desenvolvimento do projeto.

Aos meus animais de estimação, que sempre estiveram ao meu lado fazendo companhia e alegria.

Resumo

O assunto tratado nessa monografia foi a determinação de um método de modelagem e o uso da arquitetura API REST para o desenvolvimento de um sistema de gerenciamento de redes de fibras ópticas. Uma rede de fibras ópticas possui muitos elementos distintos, que se relacionam, sendo um desafio e um benefício na representação deles em um sistema unificado. Para verificar a modelagem e a arquitetura proposta, foi desenvolvido um *web service* com as características descritas no trabalho. Melhorias futuras incluem o uso de autenticação na comunicação com a API, e a comparação da arquitetura atual com a arquitetura de microsserviços.

Palavras-chave: API, REST, Serviços Web

Abstract

The main topic of this work is related to the use of the API REST architecture and a modeling technique to develop a fiber optics distribution grid management system. A great challenge and benefit of the solution acquired by this work is the representation of all elements in a unified system, since a fiber optic distribution grid has many items and a lot of relation between them. Thus, this work aims at verifying the architecture and modeling techniques proposed by developing a web service according to the characteristics described. Future improvements suggested include the use of authentication on the API communication, and a comparison between the architecture used and the microservices architecture.

Keywords: API, REST, Web Services

Sumário

1 Introdução **1**
1.1 Motivação 1
1.2 O Problema Cadastro 3
1.3 Objetivo Geral 5
1.4 Objetivos Específicos 5
1.5 Visão Geral da Solução 6
1.6 Organização do Trabalho 7

2 Referenciais Teóricos **8**
2.1 Web Service 8
2.2 Application Programming Interface (API) 8
2.3 Hypertext Transfer Protocol (HTTP) 9
2.4 Representational State Transfer (REST) 10
2.5 Object Relational Mapper (ORM) 12
2.6 Modelagem 13
2.7 SCRUM 15
2.8 Elementos da Rede de Fibras Ópticas 16
2.9 Resumo Conclusivo 20

3 Modelagem **21**
3.1 Abordagem Inicial 21
3.2 Modelagem do Banco de Dados 22
3.3 Resumo Conclusivo 25

4 Arquitetura **26**
4.1 Arquitetura do Servidor 26
4.2 Linguagem de Programação 29
4.3 Estrutura e Organização do Código 29
4.4 Arquitetura do Banco de Dados 30
4.5 Resumo Conclusivo 34

5 Prova de Conceito **35**
5.1 Fluxo dos Dados 35
5.2 Regras de Negócio 38
5.3 Banco de dados 39
5.4 Visão Geral 40
5.5 Dimensão da Aplicação 41
5.6 Funcionalidades Adicionais 41
5.7 Vantagens e Desvantagens 43
5.8 Resumo Conclusivo 44

6 Conclusão **45**

Referências **47**

Lista de Figuras

1.1 Esquemático do Sistema. 3
1.2 Exemplo de Arquivos CAD. 4
1.3 Exemplo de Arquivo do Google Earth. 4
1.4 Exemplo de Arquivo do Excel. 4

2.1 Relações do Diagrama de Classes. 14
2.2 Exemplo de um Cabo de Fibras Òpticas. 16
2.3 Exemplo de *Loose Tube* e Fibras Coloridas. 17
2.4 Exemplo de uma Caixa de Emenda Subterrânea. 17
2.5 Exemplo de uma Caixa de Emenda Aérea com Reserva Técnica. 18
2.6 Exemplo de uma Reserva Técnica. 18
2.7 Exemplos de um DGO. 19
2.8 Exemplo de um Jumper. 19

3.1 Grupo de Elementos Principais da Rede de Fibras ópticas. 22
3.2 Modelo Relacional do BD. 24

4.1 Exemplo de Adição de um Poste com Sucesso. 27
4.2 Exemplo de Tentativa de Adição de um Poste com Erro. 27
4.3 Arquitetura da API REST. 28

5.1 Representação do Fluxo do Dado nos Pacotes Implementados. 40
5.2 Ferramenta de Administração do Banco de Dados *pgAdmin*. 42
5.3 Representação das Imagens do Docker Usadas. 43

Lista de Tabelas

2.1 Métodos de Requisições HTTP. 10
2.2 Métodos de Requisições REST e Suas Ações. 11
2.3 Código de Status REST.. 12

4.1 Métodos de Requisições, Ações e Endpoint Implementados Pela API. 28

Lista de Algoritmos

4.1 Exemplo de Etiquetas na Interface de Objetos Go 30
4.2 Exemplo de Tipo Customizado para Manipulação do GORM 31
4.3 Exemplo do Modelo de Localização Utilizado 32
4.4 Exemplo do Tipo Customizado *POINT* 32
5.1 Exemplo de Implementação do Servidor HTTP 36
5.2 Exemplo de Roteamento 37
5.3 Exemplo de Todas Etiquetas Usadas na Interface de Objetos Go 38
5.4 Exemplo de Uso das Funções do GORM 39

Lista de Abreviaturas e Siglas

API Application Programming Interface.

BD banco de dados.

CORS Cross-Origin Resource Sharing.

CRUD Create Read Update Delete.

CSS Cascading Style Sheets.

DGO Distribuidor Geral Óptico.

EWKB Extended Well Known Binary.

GBIC Gigabit Interface Converter.

GORM Grails Object Relational Mapping.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

JS JavaScript.

JSON JavaScript Object Notation.

JWT JSON Web Token.

ORM Object Relational Mapper.

REST Representational State Transfer.

SOAP Simple Object Access Protocol.

SQL Structured Query Language.

UML Unified Modeling Language.

UoD Universe of Discourse.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

XML Extensible Markup Language.

Capítulo 1

Introdução

A GigaCandanga [1], que gerencia e distribui redes de fibras ópticas, possui atualmente o armazenamento e a utilização de seus dados cadastrados da rede em diversos meios, comprometendo a manipulação e a visualização completa da rede.

Para solucionar esse problema, é proposta a criação de uma aplicação *web* separada em dois serviços. O foco deste trabalho foi na criação do serviço *back-end*, que faz o papel de um servidor API REST que fornece, armazena e modifica os dados enviados a ele de acordo com suas rotas de comunicação.

Essa aplicação busca a unificação de todos os dados em apenas um sistema, com o intuito de ser mais flexível e de fácil utilização para o usuário final, garantindo assim uma maior eficiência e segurança no cadastro e manipulação da rede.

Assim, neste capítulo será abordado a respeito da associação GigaCandanga e o problema a ser solucionado, que motivou este trabalho. Também é mostrado os objetivos a serem alcançados, uma breve visão da solução proposta e a forma em que o trabalho foi separado.

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.

O gerenciamento de redes de fibras ópticas exige a manipulação de vários elementos, que podem se inter-relacionar, como cabos de fibras ópticas com caixas de emenda, que após a relação passam a fazer parte da rede como um conjunto. As análises feitas pelos

gerenciadores da rede de fibras ópticas e as operações sobre esses dados para a visualização da rede existente, cadastros de elementos ou modificações nos dados, podem ser divididas e feitas de diversas maneiras. Desde a separação em formas manuais em papeis ou planilhas, como o uso de meios digitais com um sistema integrando todos os elementos.

Contudo, o método escolhido para a manipulação e o armazenamento dos dados da rede de fibras ópticas é essencial para a manutenibilidade e a organização estrutural desse tipo de sistema, visto que isso irá impactar também as possibilidades de visualização, distribuição e operação dos dados para outros meios e aplicações.

Especificamente nesse contexto de cadastro da estrutura e de elementos físicos necessários para a ligação de várias instituições por meio da utilização de fibras ópticas, há dois principais desafios para a escolha do melhor método de se gerenciar as redes. O primeiro é a modelagem dos dados de todos os elementos presentes em uma rede de fibras ópticas, tanto devido às inter-relações existentes, quanto na quantidade de elementos que serão gerenciados. O segundo é a arquitetura do sistema, que deve ser planejada para ser escalável juntamente à rede, de fácil manipulação para os usuários e adaptável a diferentes dispositivos (ex: Smartfone, Notebook etc).

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 de *back-end*. Para uma visão completa da solução, é sugerida a leitura de todos os trabalhos: Chaves [2](2020), Chianca [3](2020) e Jorge [4](2020).

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.

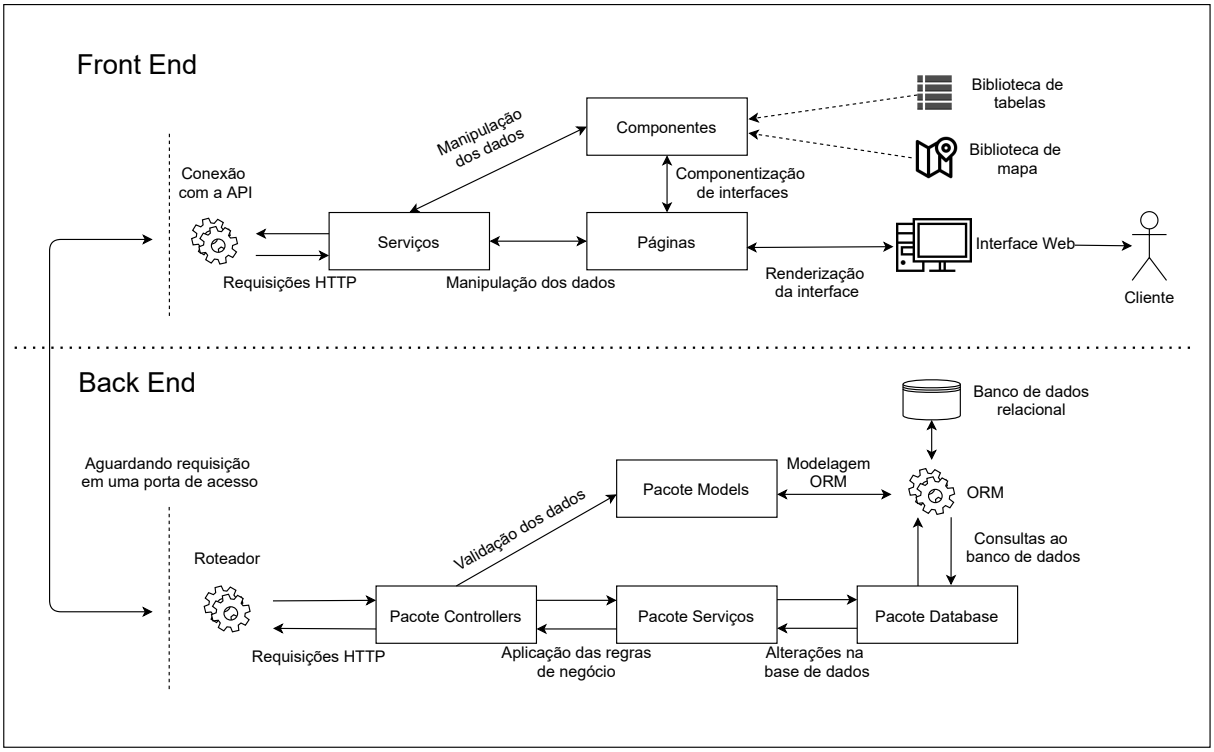


Figura 1.1: Esquemático do Sistema.

No caso do *back-end*, o desenvolvimento seguiu em grande parte o *pair programming*, que é uma técnica do desenvolvimento ágil de trabalhar em conjunto, sendo todas as etapas implementadas de uma forma geral pela dupla. Porém, individualmente, o maior foco foi a implementação da configuração do ORM, a criação das relações dos modelos e na criação de algumas rotas do CRUD dos elementos, como visto o fluxo na Figura 1.1.

1.2 O Problema Cadastro

A associação GigaCandanga possui atualmente seus registros da rede de fibras ópticas distribuídos em ferramentas de projetos CAD (Figura 1.2), que são usados no planejamento para a implantação da rede, arquivos do Google Earth (KML/KMZ) (Figura 1.3), que representam visualmente a rede física implantada, e diversas planilhas de Excel [5] (Figura 1.4), que armazenam principalmente a especificação de todas as relações que as fibras fazem entre si e com outros dispositivos da rede, além de possuírem algumas informações de cadastro de outros elementos.

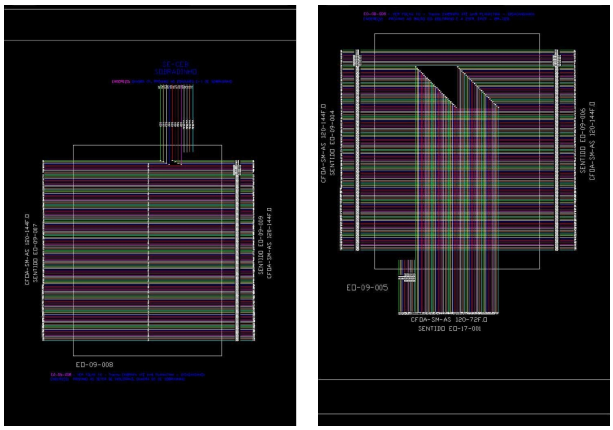


Figura 1.2: Exemplo de Arquivos CAD.

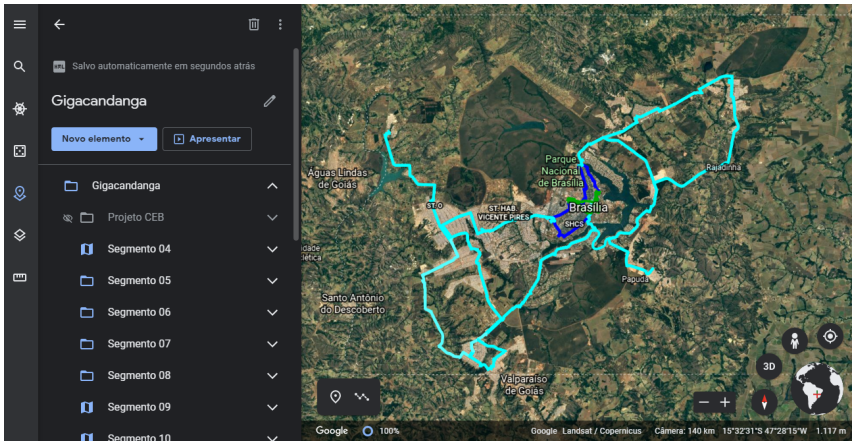
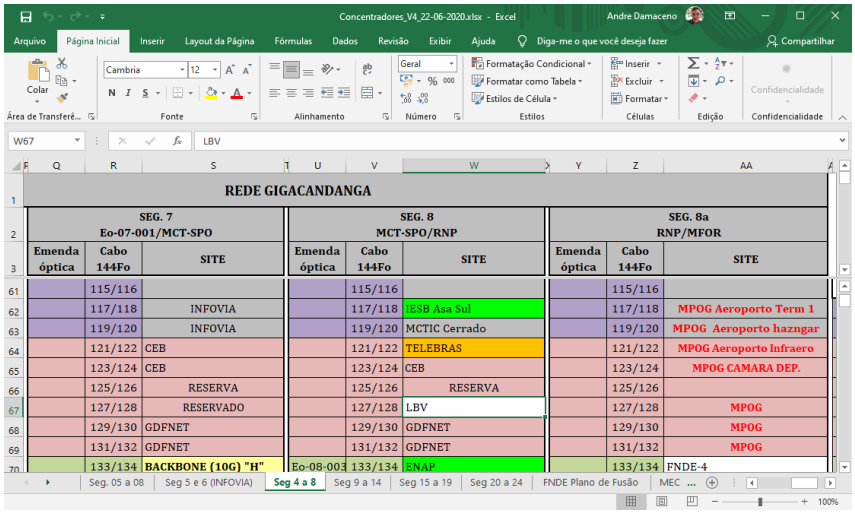


Figura 1.3: Exemplo de Arquivo do Google Earth.



REDE GIGACANDANGA

SEG. 7 Eo-07-001/MCT-SPO			SEG. 8 MCT-SPO/RNP			SEG. 8a RNP/MFOR		
Emenda óptica	Cabo 144Fo	SITE	Emenda óptica	Cabo 144Fo	SITE	Emenda óptica	Cabo 144Fo	SITE
	115/116			115/116			115/116	
	117/118	INFOVIA		117/118	IESB Asa Sul		117/118	MPOG Aeroporto Term 1
	119/120	INFOVIA		119/120	MCTIC Cerrado		119/120	MPOG Aeroporto hazngar
	121/122	CEB		121/122	TELEBRAS		121/122	MPOG Aeroporto Infraero
	123/124	CEB		123/124	CEB		123/124	MPOG CAMARA DEP.
	125/126	RESERVA		125/126	RESERVA		125/126	
	127/128	RESERVADO		127/128	LBV		127/128	MPOG
	129/130	GDFNET		129/130	GDFNET		129/130	MPOG
	131/132	GDFNET		131/132	GDFNET		131/132	MPOG
	133/134	BACKBONE (10G) "H"	Eo-08-003	133/134	ENAP		133/134	FNDE-4

Figura 1.4: Exemplo de Arquivo do Excel.

Devido à forma em que está sendo feito o armazenamento dos dados da rede, descritos anteriormente, há uma grande dificuldade atualmente na visualização completa da rede, na busca de informações de cada elemento e na manipulação de cada dado. Essa complexidade se dá principalmente devido à necessidade de se utilizar várias ferramentas, tornando o processo ineficiente e com um alto risco de perda de dados, inconsistências, além de estar sujeito a maiores erros, visto que é possível que uma mesma informação tenha que ser inserida várias vezes em mais de um meio, podendo ocasionar uma maior chance de um equívoco.

Dessa forma, a unificação de todas as informações em apenas uma aplicação desenvolvida especificamente para manipulação e disposição de todos os dados é imprescindível não só para a escalabilidade futura do cadastro como também para a manutenção da rede.

Os desafios dessa unificação envolvem a análise de uma arquitetura adequada, o mapeamento e a modelagem de todos os itens usados na rede, de forma que eles mantenham suas correlações, a transposição desse modelo para um tipo de armazenamento lógico das informações, e que a aplicação seja feita para ser flexível, que possa ser usada em diferentes dispositivos e serviços.

1.3 Objetivo Geral

O principal objetivo a ser atingido por esse trabalho é o desenvolvimento de uma camada de manipulação do cadastro para unificar toda a rede de fibras ópticas em apenas um banco de dados. O sistema deve ser simples e intuitivo de se utilizar, deve permitir o avanço futuro para diferentes dispositivos, evitar falhas humanas e ter uma forma de armazenamento dos dados segura.

1.4 Objetivos Específicos

Os objetivos específicos do trabalho são:

- Tornar o sistema escalável juntamente à rede física de fibras ópticas, podendo expandir além dos registros atuais.
- Identificar todos os elementos e relações existentes em uma rede de fibras ópticas.
- Realizar uma modelagem de todos os dados da rede de fibras ópticas.
- Identificar como os dados serão armazenados e também a melhor arquitetura a ser usada para a manipulação dos elementos.
- Implementar a prevenção de erros não intencionais.

- Desenvolver uma verificação automática das relações entre os elementos, assim como os campos de cadastro de cada item.

1.5 Visão Geral da Solução

Em conjunto com a associação GigaCandanga REDECOMEP-DF, em um grupo que totaliza quatro alunos de graduação de engenharia de computação, foi proposto o desenvolvimento de um serviço *web* para o cadastro, visualização e manipulação dos dados da rede física de fibras ópticas da associação. A escolha de se utilizar um serviço *web* traz a flexibilização e diversificação necessária para a aplicação.

O projeto é separado em duas frentes, denominadas *front-end* e *back-end*. O *front-end* é constituído pelas duplas Chaves [2](2020) e Chianca [3](2020), responsáveis pelo desenvolvimento da parte de interface do usuário, e o *back-end* constituído por Damaceno e Jorge [4](2020), que ficaram responsáveis pelo desenvolvimento da distribuição, armazenamento e tratamento dos dados.

O desenvolvimento foi feito seguindo as metodologias ágil *SCRUM*, em que o *SCRUM master* se comunicava com o cliente (GigaCandanga), formulava e nos informava o escopo das tarefas que devem ser feitas, para a entrega de maior valor entre as *sprints*.

O projeto segue as características de um servidor API REST, sendo este um serviço que recebe requisições HTTP *POST, GET, PUT, PATCH* e *DELETE*, com um interfaceamento para a comunicação entre outras aplicações, e troca de mensagens por meio de objeto JSON, que representa os dados pela estrutura de chave/valor, que é utilizado pela equipe do *front-end*.

A produção de uma API centraliza toda lógica e manipulação dos dados em um servidor, que por sua vez, pode disponibilizar as rotas que fornecem os comandos de disposição dos dados a diversos outros serviços, tornando a possibilidade de manipulação dos elementos de forma bastante simples.

O desenvolvimento do *back-end* é feito utilizando a linguagem de programação *Go* [6], da Google [7], em conjunto com o pacote de *framework web Gin* [8] para a criação da API.

Para o armazenamento e a manipulação dos dados de rede, foi escolhido o banco de dados relacional POSTGRESQL [9], com suas características de relação entre as entidades e tabelas criadas com o pacote GORM [10] do *Go*, que é uma biblioteca de Object Relational Mapper (ORM), ou seja, é uma forma de manipular requisições de um banco de dados utilizando métodos e conceitos do paradigma de orientação a objetos.

Todos os dados necessários para o mapeamento de forma digital da rede de fibras ópticas é representado na aplicação em forma de interfaces, com as interligações entre os

elementos representados como o relacionamento de objetos, e para cada elemento, há uma definição lógica de relação e propriedades SQL para a definição de cada coluna, tabela e chave estrangeira do banco de dados.

A manipulação de qualquer item da rede é feita pelas requisições HTTP ao serviço *back-end*, que valida a requisição e os campos, pedidos ou a serem armazenados, retornando sempre uma resposta de estado HTTP para a aplicação que fez a requisição, informando sucesso ou falha.

1.6 Organização do Trabalho

A monografia foi separada em seis capítulos. No primeiro é apresentada a motivação, os objetivos e a visão geral da solução desenvolvida. No segundo capítulo são abordados os referenciais teóricos, com os conceitos, arquitetura e modelos usados na confecção, e essenciais para o entendimento do trabalho. No terceiro capítulo é abordada a modelagem feita, detalhando a organização e técnicas usadas. No quarto capítulo é descrita a arquitetura usada, juntamente com a escolha das ferramentas, linguagens de programação e o banco de dados. No quinto capítulo é feita a prova de conceitos, descrevendo o processo de implementação, detalhes adicionais, vantagens e desvantagens. Por fim, o sexto capítulo contém a conclusão da monografia.

Capítulo 2

Referenciais Teóricos

Neste Capítulo, são abordados os conceitos, arquitetura e modelos essenciais que foram usados no desenvolvimento do trabalho. Cada subseção explica e define as características necessárias para o entendimento, além de apontar os principais benefícios de cada escolha.

2.1 Web Service

A solução encontrada para o cadastro da rede de fibras ópticas foi o desenvolvimento de um serviço *web*, que traz a disponibilidade e flexibilidade de uso em diferentes dispositivos, além de manter todos os dados sincronizados para todas as pessoas que irão utilizar o sistema.

Segundo Kalin (2009, p. 1 - 2) [11], o termo *web service* é definido como sendo um serviço que possua clientes, que também podem ser chamados de consumidores, no qual a comunicação da aplicação seja feita tipicamente pela utilização do protocolo HTTP ou HTTPS. É uma aplicação distribuída em que os componentes podem ser implantados e executados em diferentes dispositivos e feitos em diversas linguagens de programação, sendo os dois serviços mais conhecidos o REST e o SOAP.

2.2 Application Programming Interface (API)

A disponibilização da aplicação é feita por uma interface definida em rotas, que proporciona ações e recursos para cada operação de gerenciamento do cadastro da rede. Devido a essa característica, o uso de uma API é essencial para a expansão futura do serviço a outros meios.

De acordo com Reddy (2011, p. 1 - 8) [12], *Application Programming Interface (API)*[1] é uma interface bem definida que fornece um serviço específico para outras peças de

[1]Traduzido para português como Interface de Programação de Aplicação

software, ou seja, um componente de abstração de um problema, que é disponibilizado para a interação padronizada aos elementos de software que fornecem uma solução simples.

Uma API pode ser feita por qualquer desenvolvedor, e utilizada por qualquer tipo de aplicação, das mais simples a complexas, além de também poder possuir interdependências com outras APIs. De um modo geral, é distribuída como uma biblioteca de software, na qual há uma modularização das funcionalidades, que são fornecidas para as aplicações dos usuários finais em componentes disponíveis por uma interface lógica (como um *web service* por exemplo), que consequentemente oculta todos os detalhes da implementação.

As características e benefícios principais de uma API são:

- Compartilhamento da mesma API por várias aplicações;
- Retrocompatibilidade, para a manutenção de aplicações antigas;
- Grande durabilidade de utilização das APIs;
- Detalhes da implementação são invisíveis;
- Modularização;
- Menor quantidade de código duplicado para as aplicações;
- Maior simplicidade na otimização do código da API;
- Reutilização de código;

2.3 Hypertext Transfer Protocol (HTTP)

Para a comunicação do *back-end* com os clientes, é utilizado o HTTP, que define as formas possíveis de comunicação pelo cabeçalho e corpo das mensagens, além de ser adaptável na criação de novos recursos.

Segundo Saudate (2013, p. 13 - 27) [13], *Hypertext Transfer Protocol (HTTP)*[2] é um protocolo de camada de aplicação criado por Tim Berners-Lee, Roy Fielding e Henrik Frystyk Nielsen em 1996, que foi feito para ser flexível e suportar diversas necessidades diferentes das aplicações.

O protocolo define a forma em que é feita a comunicação entre um cliente e um servidor por meio de requisições, que são resumidamente separadas em cabeçalho e corpo, e enviadas pela utilização de um método, que define uma ação.

Os cabeçalhos[3] são de uso opcional no protocolo, porém são utilizados para trafegar meta informações a respeito das requisições. Alguns são padronizados, porém no geral

[2]Traduzido para português como Protocolo de Transferência de Hipertexto
[3]Chamado de *Header* em inglês.

são extensíveis, podendo ajustar de acordo com cada particularidade, deixando sempre o cliente e o servidor livres para negociar o conteúdo da melhor forma possível.

O corpo[4] da mensagem, que também é de uso opcional no protocolo, é utilizado para transmitir informações mais densas tanto pelo servidor quanto pelo cliente, geralmente usado em conjunto da meta informação de tipo de mídia no cabeçalho, para simplificar o processamento do dado enviado, podendo ser por exemplo um JSON ou um XML.

Os métodos do HTTP são usados para indicar qual é a ação que a requisição deve executar. O protocolo define que os métodos não são fixos e podem ser estendidos, porém a versão 1.1 define oficialmente oito padrões, como visto na Tabela 2.1:

Tabela 2.1: Métodos de Requisições HTTP.

Método	**Descrição**
GET	Requisita a representação de um recurso específico.
POST	Submete uma entidade a um recurso específico.
PUT	Requisita a substituição do recurso de destino pelos dados da requisição.
DELETE	Remove um recurso específico.
OPTIONS	Requisita as opções de comunicação com o recurso de destino.
HEAD	Requisita um recurso específico, sem o corpo da mensagem.
TRACE	Retorna a mesma requisição feita.
CONNECT	Estabelece um túnel TCP/IP para o servidor identificado pelo recurso de destino.

Por fim, o protocolo define cinco faixas de códigos de status, que são números de resposta enviados pelo servidor para o cliente em todas as requisições feitas, representando uma informação sobre o pedido feito. Genericamente no HTTP, a faixa 1xx é informacional, 2xx de sucesso, 3xx redirecionamentos, 4xx erros do cliente e 5xx erros no servidor.

2.4 Representational State Transfer (REST)

O padrão de arquitetura REST foi escolhido para ser usado na comunicação da aplicação por definir bem a utilização de um conjunto de métodos HTTP para a manipulação e disponibilização de recursos, além de suas características de indicação de erros, definição das mensagens e simplicidade.

Segundo Yellavula (2020, p. 7 - 12) [14], *Representational State Transfer (REST)*[5] é um tipo de *web service* criado por Roy Fielding, que possui uma flexibilidade na criação dos serviços, com a liberdade de escolher a plataforma e tecnologia de desenvolvimento, contanto que sejam seguidos os seus princípios.

[4]Chamado de *Body* em inglês.

[5]Traduzido para português como Representação por Transferência de Estado

REST é definido como um padrão de arquitetura que permite a comunicação de diferentes sistemas de uma forma simples, através de um método de requisição HTTP e um caminho de URL definido, que é mapeado pela aplicação para a operação desejada, que faz as operações de recuperar, atualizar ou excluir os dados, podendo ser usado como uma API.

Yellavula (2020, p. 10) [14] afirma:

> Desempenho, escalabilidade, simplicidade, portabilidade e modificabilidade são os princípios básicos por trás do design REST. (Tradução livre)[6].

Devido a utilização do HTTP para a comunicação, REST é uma arquitetura baseada em cliente-servidor. A sua comunicação possui algumas características, dentre as mais importantes estão o não armazenamento do estado das requisições e a utilização de cache, ou seja, todas as requisições feitas por um cliente independem de requisições anteriores e são respondidas pelo servidor de forma otimizada (através da aceleração na busca de informações muito requisitadas).

A interface de comunicação REST é definida pelo *Uniform Resource Identifier (URI)*[7], que é uma forma de representação uniformizada de recursos, que mapeia através de uma interface as requisições com os dados lógicos, ou seja, define *endpoints* específicos para cada recurso, e ainda pode especificar o tipo dos dados na resposta esperada, pela identificação no cabeçalho da requisição do tipo de mídia requerido. Similarmente ao HTTP, REST envia as informações por mensagens chamadas de requisições. No geral, a requisição do cliente deve conter um método de requisição, os campos de cabeçalho[8] e um corpo[9].

Para decodificar o recurso que vai ser manipulado, é usado o URI. Os métodos e ações mais usadas são descritas abaixo na Tabela 2.2, juntamente com suas ações esperadas:

Tabela 2.2: Métodos de Requisições REST e Suas Ações.

Método de Requisição REST	**Ação**
GET	Busca um registro ou um conjunto de recursos do servidor
OPTIONS	Busca todas as operações REST disponíveis
POST	Cria um recurso ou um novo conjunto de recursos
PUT	Atualiza ou substitui o registro fornecido
PATCH	Modifica o registro fornecido
DELETE	Exclui o recurso fornecido

Fonte: Yellavula (2020, p. 11) [14]

O código de status HTTP é usado para indicar o resultado de uma operação REST feita por um cliente (sucesso ou falha), visto que o REST não possui estado. São definidos

[6]Performance, scalability, simplicity, portability, and modifiability are the main principles behind the REST design.

[7]Traduzido para português como Identificador Uniforme de Recurso

[8]Componente HTTP *Header*, do inglês

[9]Componente HTTP *Body*, do inglês

pelo REST alguns códigos padrões, podendo ser separados em três tipos de status (sucesso, erro ou redirecionamento), que devem ser usados para garantir o uso correto do padrão REST e a comunicação entre cliente-servidor. A Tabela 2.3 a seguir mostra cada um desses status, sendo o erro descrito separadamente para o cliente e o servidor.

Tabela 2.3: Código de Status REST..

Requisição REST	**Faixa dos Números**	**Ação**
Sucesso	200 - 226	Respostas de sucesso
Redirecionar	300 - 308	Redirecionamento da URL
Erro (cliente)	400 - 499	Indica erros do cliente
Erro (servidor)	500 - 599	Indica falhas de processo do servidor

Fonte: Yellavula (2020, p. 12) [14]

2.5 Object Relational Mapper (ORM)

Para a comunicação do servidor *back-end* com o banco de dados (BD), foi utilizado um pacote de ORM. Essa escolha foi feita pela simplicidade do uso de um ORM, a realização de consultas SQL de forma invisível, otimizada e correta, a facilidade de reutilização e manutenção do código, e também devido à complexidade dos dados da rede de fibras ópticas.

De acordo com Mehta (2008, p. 3 - 5) [15], *Object Relational Mapper (ORM)*[10] é a utilização de representações conceituais dos dados com objetos da programação orientada a objetos, pelo uso de metadados como descritor, para se definir e manusear um banco de dados relacional, fazendo com que a comunicação seja simples e invisível entre o programa e o banco de dados.

Outra definição feita por Raiturkar (2018, p. 294) [16] é:

> ORM é uma forma especial de camada de acesso a dados, que traduz as entidades do banco de dados em objetos. (tradução livre)[11]

Também, na mesma página, Raiturkar cita uma importante observação sobre o uso de banco de dados (BD):

> Embora o código da aplicação possa certamente interagir com o BD usando instruções SQL, você precisa ter cuidado para garantir que as interações do BD não fiquem espalhadas pelas camadas da aplicação. A camada de acesso a dados é uma camada

[10] Traduzido para português como Mapeamento objeto-relacional

[11] Objects Relational Mappers (ORMs) are a special form of DAL, which translate DB entities into objects.

> que deve ser responsável por manipulação de entidades e suas interações com o BD. O resto da aplicação é abstraído dos detalhes de interação do BD. (tradução livre)[12].

Mehta (2008, p. 5 - 8) [15] menciona benefícios e características de um bom ORM como sendo uma ferramenta que possui um mapeamento de um objeto da aplicação para o BD, ou seja, os próprios objetos da programação orientada a objetos são os descritores dos dados do BD.

Outras características importantes são o uso de cache em objetos para melhorar a performance da aplicação, com o intuito de evitar chamadas recorrentes ao BD, suporte a múltiplos BDs e às consultas dinâmicas baseadas na entrada do usuário, sendo realizadas diretamente por funções e usos de objetos sem a necessidade da escrita de *Structured Query Language (SQL)*.

Além disso, um grande benefício é a geração de código, que inicializa as tabelas e características de relações do BD de acordo com o modelo dos metadados inseridos nos objetos, sem a necessidade de escrita de relações com o SQL. Também, a separação da comunicação com o BD apenas em seus módulos específicos, que trás uma maior organização do código e facilita a manutenção.

2.6 Modelagem

Devido a grande complexidade dos dados e das relações existentes no cadastro da rede de fibras ópticas, é essencial seguir um método para a modelagem dos dados, principalmente para a definição dos objetos na programação e do banco de dados.

Segundo Halpin (2001, p. 2 - 18) [17], para se realizar o modelo de uma aplicação, é necessário a compreensão de todos os dados e relações feitas, e a dimensão da aplicação, que quando modelado é chamado de *Universe of Discourse (UoD)*[13]. A modelagem é resumidamente separada em três etapas: A separação e compreensão de todos os dados para a construção do UoD, criação da representação do modelo e a implementação.

Para a construção do UoD, são utilizados elementos representativos da aplicação e formas simples de comunicação, com o uso de diagramas intuitivos, legendas e esquemáticos. Por esse motivo, o modelador, que é a pessoa que cria o UoD, necessita estar familiarizado com o universo da aplicação, caso contrário, é essencial que seja buscado auxílio com especialistas da área para consultas, começando pela análise das menores partes de informação possíveis, para simplificar o trabalho.

[12] While the application code can certainly interact with the database using SQL statements, you needs to be careful to ensure that the DB interactions are not strewn across the application layers. The Data Access Layer (DAL) is a layer that should be responsible for handling entities and their interactions with the database. The rest of the application is abstracted from the DB interaction details.

[13] Traduzido para português como Universo de Discurso.

A representação do modelo é inicialmente especificada em um rascunho do projeto conceitual, sem a preocupação com a implementação, fazendo as adaptações necessárias para a representação dos elementos e suas interligações.

Uma vez que o projeto conceitual inicial tenha sido representado, ele pode ser mapeado para um design lógico em qualquer modelo de dados, o que traz a flexibilidade de migrar a aplicação para diferentes sistemas.

A linguagem de elaboração de modelagem mais promissora é a *Unified Modeling Language (UML)*[14], que especifica suas estruturas de dados por diagramas e identifica as relações entre as estruturas por conexões, podendo ser usada para visualização, construção e documentação do sistema.

Uma das estruturas mais importantes do UML é o diagrama de classes, que é uma boa estrutura para o design lógico e físico de código orientado a objetos, devido ao diagrama também representar as interações e operações possíveis entre os elementos em suas conexões. Um exemplo típico de representação das relações é visto abaixo na Figura 2.1.

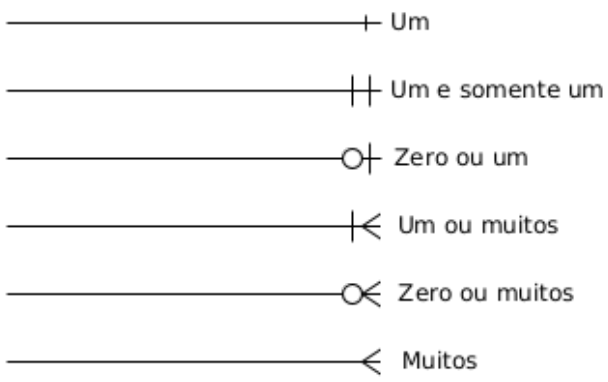


Figura 2.1: Relações do Diagrama de Classes.

Além disso, também é importante a realização da normalização dos dados, para que o modelo seja refinado e melhor representado. A normalização é um processo de análise, que verifica as relações dos dados em uma tabela. No geral, ao se identificar anomalias, são feitas tabelas secundárias para o relacionamento com a original, a fim de se evitar grandes redundâncias na representação dos dados.

[14]Traduzido para português como Linguagem de Modelagem Unificada.

2.7 SCRUM

Como o projeto foi desenvolvido em um grupo de quatro integrantes, foi necessário a escolha de algum método para a organização e separação das tarefas, também para melhorar o foco nas funcionalidades principais. A metodologia que melhor se enquadra nas nossas necessidades, principalmente por termos um acompanhamento feito por um funcionário da GigaCandanga para gerenciamento de projeto, foi o SCRUM.

Segundo Satpathy (2017, p. 2 - 5) [18], SCRUM é uma metodologia ágil que utiliza técnicas para o aprimoramento do trabalho em equipe, com um grande foco em transparência, organização e comunicação. Existem três papeis principais no *SCRUM*: O *SCRUM master*, que possui o domínio das regras do guia do *SCRUM* e é responsável pelo cumprimento dos princípios no time e na organização inteira, o *Product owner*, que é responsável pelo planejamento e organização do produto, sem entrar em aspectos tecnológicos, mas definindo uma orientação do que deve ser priorizado e quais as funcionalidades são as mais importantes de serem focadas primeiro, e por fim o time de desenvolvimento, que segue a priorização definida pelo dono do projeto e desenvolve o produto.

No SCRUM, o projeto é dividido em atividades, que são separadas em histórias de acordo com as necessidades do cliente, que são comunicadas ao *SCRUM master* pelo dono do projeto. Então, são separadas pela equipe de desenvolvimento, em conjunto do *SCRUM master*, as histórias de maior valor, e estabelecido um período para a execução total delas chamado de *sprint*. *Sprint* é um período pré determinado pela equipe de desenvolvimento de estimativa de tempo para a execução de todas as tarefas estabelecidas. As histórias são sempre separadas em entregas de maior relevância para o cliente, ou seja, que produzem continuamente o maior valor possível.

A própria equipe do SCRUM faz a quebra das histórias em atividades menores, e a organização de quem irá executar cada tarefa, sendo realizadas reuniões diárias chamadas de *Daily Stand-Up Meeting*[15], que é uma reunião breve para o esclarecimento sobre os avanços e dificuldades encontradas, para que a equipe esteja sempre ciente de todas as coisas que estão ocorrendo, e para ajudar a identificar e acudir os membros que estão com impedimentos.

No geral, ocorrem apenas duas reuniões grandes no SCRUM, a de inicio de *sprint*, para a definição das histórias, e a de fim de *sprint*, que é quando ocorrem as entregas dos trabalhos e a análise de aceitação pelo cliente das tarefas executadas.

Dessa forma, o processo do SCRUM completo se dá pela análise das necessidades do cliente pelo *SCRUM master*, seguido da reunião de início de *sprint*, onde são separadas as histórias e atividades com os membros da equipe. Então são realizadas as tarefas

[15]Traduzido para português como reunião diária em pé.

diariamente no período pré determinado do tamanho da *sprint*, fazendo as reuniões diárias de acompanhamento. E no final da *sprint*, a reunião de retrospectiva, divulgando tudo que foi feito ao cliente, que analisa as entregas de acordo com seus critérios de aceitação.

2.8 Elementos da Rede de Fibras Ópticas

A descrição de cada elemento da rede, suas associações e necessidades, foram vistas e analisadas por todos os membros da equipe de desenvolvimento, juntamente com o uso do site da GigaCandanga [19] e um especialista da GigaCandanga que nos orientou.

Para um melhor entendimento, a explicação será separada em partes que irão se relacionar, começando com uma das mais essenciais da rede, os cabos de fibra óptica (Figura 2.2). Um cabo de fibra óptica é composto geralmente por um conjunto de 36, 72 ou 144 fibras ópticas, que são separadas internamente no cabo em grupos de 12 fibras coloridas, envolvidas em um tubo colorido chamado de *loose tube* (Figura 2.3). O cabo possui um fio rígido central interno e isolamentos de plástico, para dar rigidez e proteção da fibra. A separação interna dos cabos por grupos de fibras e tubos coloridos é feita para a identificação e manutenção correta de cada fibra. As cores das fibras e dos tubos seguem um padrão, que é identificado em cada cabo.

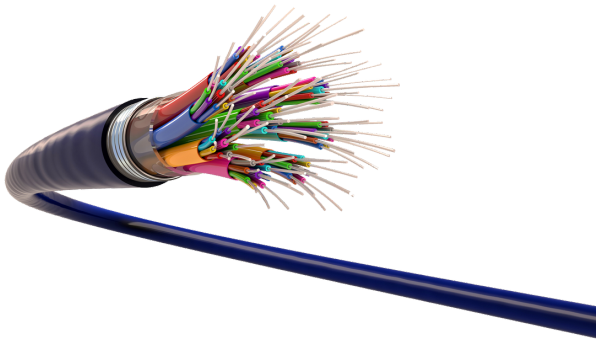


Figura 2.2: Exemplo de um Cabo de Fibras Òpticas (Fonte: [20]).

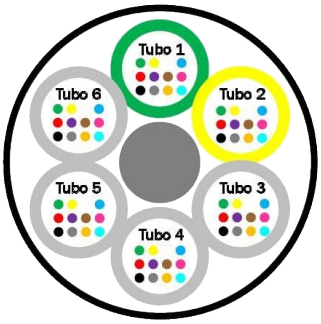


Figura 2.3: Exemplo de *Loose Tube* e Fibras Coloridas (Fonte: [21]).

As fibras são geralmente feitas por uma mistura de sílica e vidro ou de plástico, e são usadas para a transmissão de informação pelo uso de pulsos de luz. Devido ao material, no geral não são muito flexíveis e podem eventualmente sofrer rompimentos, ou terem a necessidade de ser emendadas em outro cabo, sendo essa manipulação de junção de fibras chamadas de fusão.

Para a realização de uma fusão, é necessário utilizar uma caixa de emenda. Nela chegam todos os cabos onde irão ser feitas manutenções ou associação de fibras, com o objetivo de ser um ponto protegido e de fácil acesso às fibras dos cabos. Uma caixa de emenda pode ser subterrânea (Figura 2.4), com uma tampa de acesso, ou aérea, instalada em um poste (Figura 2.5). Além disso, as caixas de emenda possuem o armazenamento de alguns metros extras de cabo para ajudar na manutenção da rede, chamados de reserva técnica (Figura 2.6).

Figura 2.4: Exemplo de uma Caixa de Emenda Subterrânea (Fonte: [21]).

Figura 2.5: Exemplo de uma Caixa de Emenda Aérea com Reserva Técnica (Fonte: [21]).

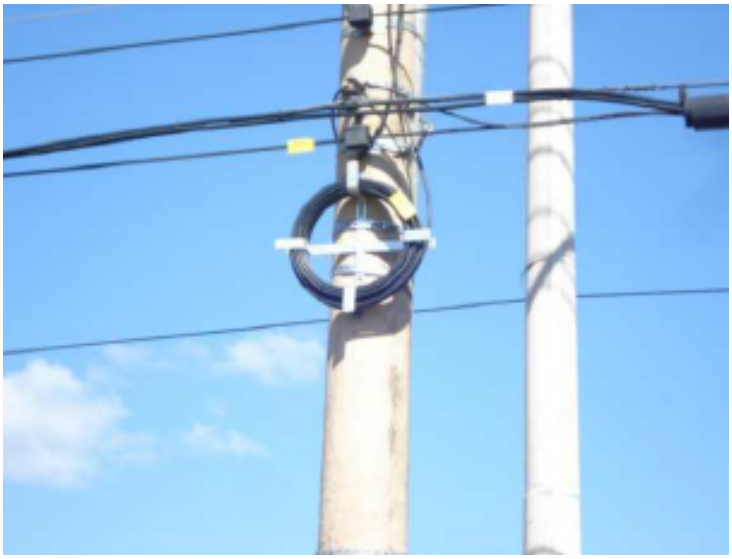


Figura 2.6: Exemplo de uma Reserva Técnica (Fonte: [21]).

Outra parte importante na visão da rede da GigaCandanga são as instituições. As instituições são os centros de ensino e pesquisa que estão integradas na rede de fibras ópticas, com uma alta capacidade de conexão entre outras instituições e também com a internet. Elas recebem o acesso da rede da GigaCandanga em um local interno chamado de *site*. Em um *site*, encontra-se fisicamente vários componentes para a utilização da rede, dentre eles o Distribuidor Geral Óptico (DGO), *Gigabit Interface Converter (GBIC)*[16] e o *switch*.

[16]Traduzido para português como Conversor de Interface Gigabit

O DGO (Figura 2.7) é um equipamento usado para conectar fibras diretamente, sem a necessidade da realização de fusões, sendo apenas um painel de conexões, em que podem ser colocados cabos de fibras menores para a conexão entre equipamentos internos do *site* e *jumpers* (Figura 2.8) para associar fibras. É usado principalmente para evitar a manipulação direta do cabo de fibra que chega na instituição.

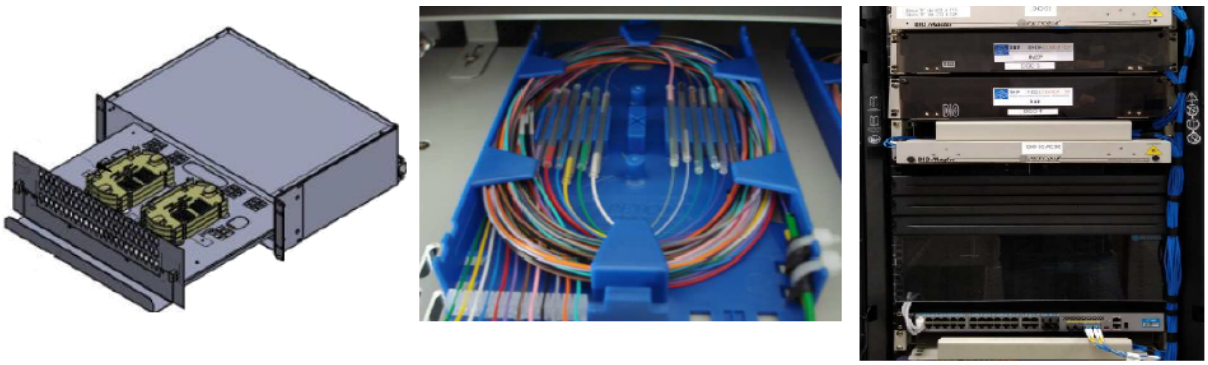


Figura 2.7: Exemplos de um DGO (Fonte: [21]).

Figura 2.8: Exemplo de um Jumper (Fonte: [21]).

O *GBIC* é usado para a conversão entre sinais elétricos e luminosos da fibra óptica. É um equipamento de extrema importância para a conexão entre dispositivos com interface convencional elétrica, para a comunicação na rede.

O *switch* é um equipamento de rede usado especialmente para a distribuição da conexão para todos os outros dispositivos da rede interna da instituição, podendo ser por um sinal elétrico ou luminoso direto da fibra. Porém, pode possuir também outras funções, dependendo de cada instituição.

Por fim, existem outras três abstrações muito usadas pela GigaCandanga, que fazem parte do sistema de gerenciamento da rede, sendo eles: trecho de cabo, cabo de acesso e segmento.

Trecho de cabo é usado para representar todo cabo de fibra óptica contínuo e sem emendas, com exceção dos cabos que conectam uma instituição. Então por consequência, uma caixa de emenda define o início e o fim de um trecho de cabo, sendo os trechos sempre separados justamente por não possuir emendas em nenhuma de suas fibras.

Cabo de acesso é todo cabo de fibra óptica que não possui emendas e faz a conexão entre uma instituição e um ponto de acesso, que pode ser provido por uma caixa de emenda por exemplo.

Segmento é todo o conjunto de elementos que estão na rede entre dois concentradores, ou seja, é todo o caminho que a rede faz de um ponto de concentração de várias conexões da rede com outro ponto.

2.9 Resumo Conclusivo

O sistema desenvolvido é um *web service* que implementa o padrão de arquitetura API REST, utilizando o protocolo HTTP para comunicação. Na realização da aplicação, foram utilizadas as práticas de desenvolvimento ágil *SCRUM*, em que o *SCRUM master* define o escopo das tarefas para a realização de entregas de maior valor.

A definição da modelagem do sistema foi feita seguindo a representação dos elementos físicos da rede de fibras ópticas, transpondo, então, esse modelo para os objetos do *Go* [6], que foram utilizados para a representação do ORM, para a criação e manipulação do banco de dados.

Capítulo 3

Modelagem

Neste Capítulo, é descrito como foi feita a abordagem inicial para a realização da modelagem dos dados. Então é mostrado o modelo relacional do banco de dados.

3.1 Abordagem Inicial

A modelagem, por ser uma parte crucial na definição da implementação da aplicação e também por necessitar de um grande esforço devido a quantidade de dados e relações existentes, foi um processo feito por todos os integrantes da equipe com o auxílio de alguns funcionários da associação GigaCandanga para a identificação dos elementos da rede de fibras ópticas, além de nos mostrar quais são as possíveis relações feitas por cada elemento.

Considerando a explicação básica sobre os elementos de uma rede de fibras ópticas descrita na Seção 2.8, inicialmente foi feito um esforço para identificar qual seria a melhor abordagem de modelagem que seria utilizada. Então, foi separado os dados em grupos de elementos principais.

Ao separar todos os elementos da rede de fibras ópticas, foi observado que é possível juntar todos os dados em quatro grupos fundamentais. O primeiro grupo engloba o cabo de fibra óptica, que é um elemento distribuidor que provém toda conexão da rede e conecta com todos os outros componentes. No segundo grupo existe a caixa de emenda, que pode ser vista como um elemento intermediário, que recebe os cabos de fibra óptica para emendar ou reparar as fibras, provém acesso a instituições e pode fazer parte de segmentos. No terceiro grupo temos as instituições, que são os elementos fins. Recebem a conexão pelas fibras dos cabos e fazem parte de um segmento. Por fim, o segmento, que é um elemento de análise macro de todas as conexões, recebendo os detalhes dos dados e separando as informações de acordo com cada percurso que o cabo de fibra óptica faz, de

um concentrador a outro. Dessa forma, com relação a uma visão macro da rede, temos os seguintes grupos de elementos principais e seus dados, visto abaixo na Figura 3.1.

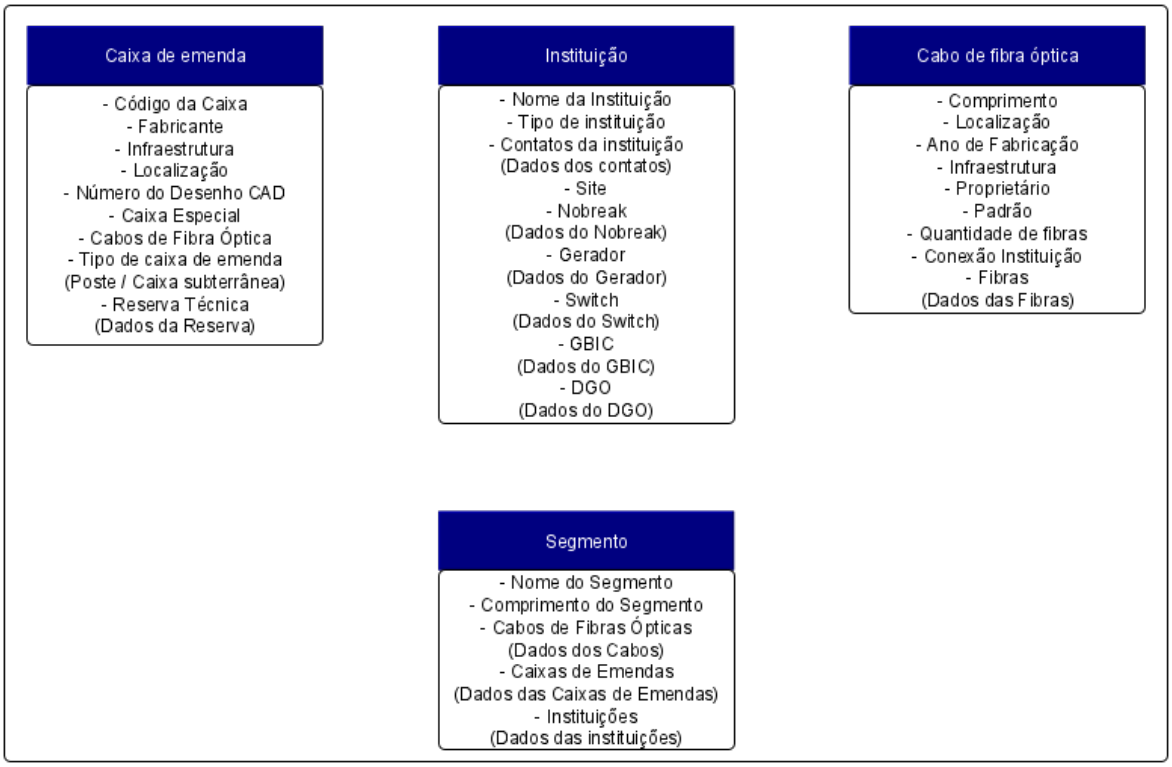


Figura 3.1: Grupo de Elementos Principais da Rede de Fibras ópticas.

3.2 Modelagem do Banco de Dados

A visão criada pela separação dos elementos nos grupos, gerou uma discussão sobre a utilização de um banco de dados relacional ou não relacional, além da escolha da estratégia utilizada na modelagem. A equipe após verificar como seria possível implementar cada abordagem, optou pela utilização de um banco de dados relacional e na aplicação de uma representação que segue de forma igual ao funcionamento e na descrição das relações existentes na rede de fibras ópticas física.

Com a análise da descrição real da rede de fibras ópticas e da separação feita anteriormente nos grupos de elementos principais, existem muitos dados entre os grupos que se interconectam. Além de existirem muitas relações internas dos dados de cada grupo. Dessa forma, para uma melhor visualização e interpretação dos modelos, foi usado o diagrama de classes com as relações entre as tabelas descritas segundo a Figura 2.1.

A realização da modelagem do banco de dados foi iniciada a partir da separação dos elementos principais descritos anteriormente, sendo feita por partes, expandindo cada grupo principal, analisando quais seriam as tabelas existentes e as relações internas necessárias. A maior intenção nessa etapa foi encontrar um equilíbrio entre redundância dos dados e a agilidade nas buscas, como por exemplo: os dados de fibras, que devem ser separados da tabela de cabos a fim de se evitar redundâncias.

Além disso, o modelo do banco de dados realizado serviu também como um direcionamento para a implementação da arquitetura. Definindo que seria usado um banco de dados relacional, além de, direcionar a escolha da linguagem de programação devido a organização dos dados. Então, a relação final completa do modelo do banco de dados (BD) pode ser vista abaixo na Figura 3.2.

A abordagem escolhida faz com que exista uma ordem na inserção dos dados. É necessário que seja realizado, primeiro, a inserção de elementos principais, como por exemplo um trecho de cabo, para que em seguida seja adicionada uma característica ou relação no elemento, como os dados de fibras ou a inserção em um segmento.

Neste sentido, devem ser inseridos primeiro os elementos que não possuam dados com chaves primárias estrangeiras, ou que possua chave estrangeira no geral. Elementos que não exigem a relação para a criação, podem seguir uma ordem diferente na inserção. Porém necessitarão de modificações futuras para a adição de outras relações.

Figura 3.2: Modelo Relacional do BD.

3.3 Resumo Conclusivo

A modelagem foi realizada por todos os integrantes do projeto, com auxílio de um especialista da GigaCandanga, que ajudou na descrição da rede de fibras ópticas. A compreensão de todos os itens da rede e como se relacionam é um passo importante para a modelagem, para melhor representar e utilizar os dados de forma mais eficiente.

A realização da identificação dos elementos principais e da modelagem do banco de dados foi um passo crucial para a redução de redundâncias e na definição dos objetos *Go* [6], que ao final representam a construção do banco de dados pelo ORM.

Capítulo 4

Arquitetura

Neste Capítulo, é descrita a arquitetura usada no servidor *back-end*, juntamente com a escolha da linguagem de programação, estrutura, organização do código e a arquitetura de implementação do banco de dados.

4.1 Arquitetura do Servidor

O servidor *back-end* foi desenvolvido de acordo com o padrão de arquitetura REST, sendo implementada uma API com um padrão da interface de comunicação que define os métodos e dados da comunicação.

Nessa abordagem, a aplicação é separada em duas partes. O *front-end*, que é um serviço utilizado para a visualização do sistema, comunicando diretamente com o navegador do usuário, retornando o HTML, CSS e JS necessários, implementado pela dupla Chaves [2](2020) e Chianca [3](2020). E o *back-end*, implementado por Damaceno e Jorge [4](2020), que é responsável pelo processamento, armazenamento e validação dos dados, assim como a distribuição deles por algum formato, no caso o JSON.

A comunicação entre os dois serviços é feita pelo uso dos *endpoints* disponibilizados pela API do *back-end*, com o *front-end* realizando requisições de obtenção, modificação, ou deleção de dados, com o uso dos métodos REST disponibilizados, juntamente com o envio de dados pelo uso do JSON quando necessário. O *back-end* recebe essas requisições, realiza a operação requisitada, caso seja válida, e retorna o dado na forma de JSON para o *front-end* juntamente com um código de *status* de resposta de sucesso (Figura 4.1) ou falha (Figura 4.2), seguindo os códigos de *status* REST.

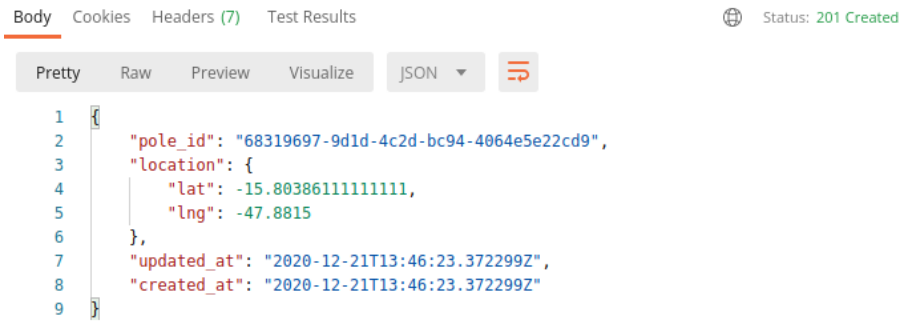


Figura 4.1: Exemplo de Adição de um Poste com Sucesso.

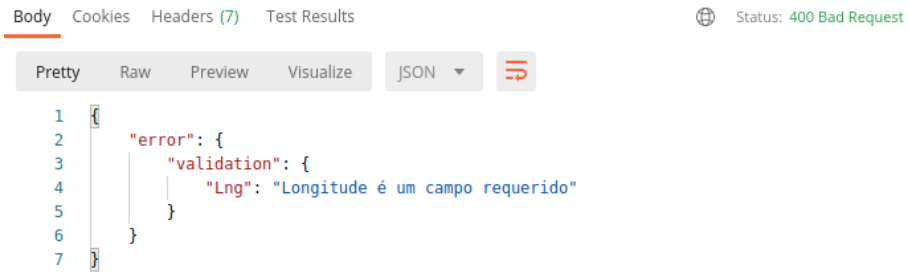


Figura 4.2: Exemplo de Tentativa de Adição de um Poste com Erro.

A escolha do uso dessa arquitetura e a separação entre o *front-end* e *back-end* em diferentes serviços traz a vantagem de uma maior flexibilidade e facilidade em escalar a aplicação para outros meios, como o *mobile* por exemplo. Além disso, essa separação remove a necessidade de um recarregamento completo da página *web* do cliente em todas as requisições. Também traz benefícios no desenvolvimento, que devido ao desacoplamento, permite a possibilidade do uso de tecnologias diferentes para cada servidor. A Figura 4.3 mostra essa separação, e a comunicação entre os serviços de forma visual.

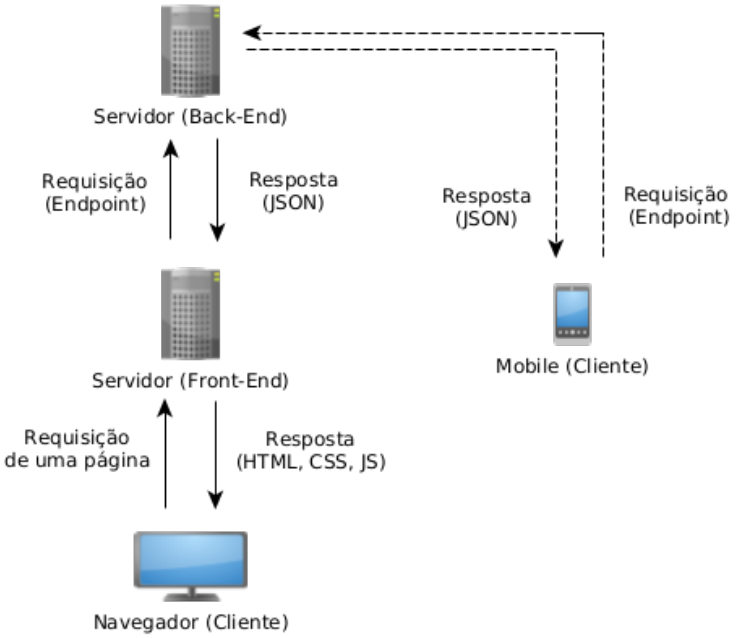


Figura 4.3: Arquitetura da API REST.

As rotas disponibilizadas pelo *back-end* foram criadas de forma similar às tabelas do modelo relacional feito anteriormente, sendo fornecido para todas as rotas os métodos *GET*, *PUT*, *POST*, *DELETE*, respectivamente para obtenção, edição, envio e deleção dos dados.

Os *endpoints* escolhidos seguem o nome dos elementos em inglês, sendo os dados trafegados de duas formas, pela URL e pelo corpo da mensagem. Os dados enviados pela URL seguem o padrão *URL/elemento/dado*. Já o dado usado no corpo da requisição é um JSON, que é enviado pelo cliente com todas as informações do elemento. Sendo enviado como resposta ao cliente as informações requisitadas, modificadas, detalhes dos erros cometidos pelo usuário ou erros internos ocorridos na operação. A Tabela 4.1 abaixo descreve o método, ação e *endpoint*, que foi implementado para todos os elementos da API.

Tabela 4.1: Métodos de Requisições, Ações e Endpoint Implementados Pela API.

Método	**Ação**	**Endpoint**
GET	Busca Todos registros	/elemento
GET	Busca o registro com 'id'	/elemento/id
POST	Registro do novo elemento enviado no JSON	/elemento
PUT	Atualiza o elemento enviado no JSON, com 'id'	/elemento/id
DELETE	Exclui todos os elementos por 'id', enviados no JSON	/elemento
DELETE	Exclui o elemento com 'id'	/elemento/id

4.2 Linguagem de Programação

Para a implementação do servidor *back-end*, foi escolhida a linguagem de programação *Go* [22]. *Golang* é uma linguagem de propósito geral, com tipagem e suporte a programação concorrente. Sua sintaxe é simples, e facilmente analisada por ferramentas de análise de código estáticas.

Os programas feitos em *Go* são construídos a partir de pacotes. Um pacote nada mais é que um diretório com alguns arquivos de código, que expõe diferentes recursos a partir de um único ponto de referência, cujas propriedades permitem um gerenciamento eficiente das dependências.

Os principais motivos da escolha do *Golang* foram sua similaridade na programação com relação a outras linguagens de programação orientada a objetos, simplicidade, suporte a programação concorrente, pacotes disponíveis e principalmente por ser uma linguagem moderna em ascensão. Além disso, foi escolhida por ter sido indicada por um analista da associação GigaCandanga, que já utilizou o *Go* na implementação de soluções similares em outros projetos, e mencionou ser uma boa opção.

4.3 Estrutura e Organização do Código

Para a melhor organização possível e separação do código em seções lógicas, foram feitos cinco pacotes principais: o *server*, *controller*, *services*, *models* e o *database*.

O pacote *server* foi feito para lidar com a criação das rotas e a separação de cada método REST que cada rota irá ter, identificando quais necessitam do parâmetro de *id* na URL, e a associação entre cada método e a função que irá lidar com a ação do método.

O pacote *controller* lida com toda a comunicação que deve ser feita, interpretando e validando os dados recebidos tanto pela URL quanto pelo *body* no formato JSON, redirecionando-os para os serviços adequados de acordo com cada método e por fim respondendo as requisições com os códigos de status REST de cada operação e com os dados JSON no *body*.

O pacote *services* executa a ação requisitada pelo *controller*, lidando com toda lógica de manipulação de dados, executa as funções do BD e operações necessárias para a ação, sendo responsável pela identificação de erros no processo, e o retorno dos dados requisitados para o *controller*.

O pacote *database* contém as conexões do BD, definições de criação de tabelas, extensões, *triggers* e chaves estrangeiras do ORM, também as funções de acesso ao BD de cada modelo, pré definindo como é feita a obtenção, atualização e deleção dos dados pelo ORM.

Por fim, o pacote *models* contém a definição da interface de todos os tipos usados e elementos da rede de fibras ópticas, também com a ligação de relação entre os objetos, sendo sua separação e relações feita seguindo o modelo da Figura 3.2.

Essa separação é ideal para um desacoplamento do código, mantendo a lógica de cada seção contida nos pacotes relacionados, facilitando assim a manutenção e expansão futura do sistema.

4.4 Arquitetura do Banco de Dados

Para a manipulação do banco de dados (BD) no programa *Go*, foi utilizado o ORM Grails Object Relational Mapping (GORM) [23], que é distribuído por um pacote público.

Para o sistema de gerenciamento de BD, foi usado o PostgreSQL [24], sendo ele um sistema *open-source* que pode ser usado livremente por qualquer aplicação, com suporte ao padrão SQL. Além disso, também foi utilizada a extensão *PostGIS* [25], que adiciona suporte de coordenadas geográficas nas consultas SQL do Postgres.

GORM utiliza as interfaces de objetos do Go como referências para a construção das tabelas do BD. Dessa forma, para uma melhor definição dos elementos que devem ser transpostos para o BD, é necessário definir juntamente ao modelo de interfaces as características de cada elemento. Essa descrição é feita pelo uso de etiquetas, definindo as características que cada elemento terá na tabela do BD, como visto no Algorítimo 4.1:

```
1  package models
2
3  import (
4    "time"
5  )
6
7  // Tipo da Tabela Exemplo
8  type TabelaExemplo struct {
9    IDTabela        VARCHAR     `gorm:"column:id_tabela; primary_key; type
       :varchar(15); not null"`
10   DadoFloat       float64     `gorm:"column:dado_float; type:real;
       default:0"`
11   DadoOculto      int         `gorm:"-"`
12   PontoGeografico POINT      `gorm:"column:point; type:GEOGRAPHY(point);
       default:null"`
13   UpdatedAt       *time.Time `gorm:"column:updated_at"`
14   CreatedAt       *time.Time `gorm:"column:created_at"`
15   DeletedAt       *time.Time `gorm:"column:deleted_at"`
16 }
17
```

```
18 // TableName : Usado para nomear no banco de dados a tabela como ‘
       tabela_exemplo‘
19 func (TabelaExemplo) TableName() string {
20   return "tabela_exemplo"
21 }
```

Algoritmo 4.1: Exemplo de Etiquetas na Interface de Objetos Go

No exemplo, é possível ver as etiquetas em roxo, sendo usadas logo após a declaração de cada dado da interface. A etiqueta define todos os atributos necessários para a transposição do BD, como o nome e o tipo da variável. Também é possível ver a função *TableName* que o GORM utiliza para a definição do nome que a tabela terá no BD. Devido a essas características da forma como o ORM funciona, a modelagem das interfaces do pacote *models* segue identicamente ao modelo do BD descrito no Capítulo 3.

Também é possível ver no exemplo a utilização de tipos customizados, como o *POINT* e o *VARCHAR*. O GORM pode utilizar esses tipos customizados sob condição de que sejam implementados os métodos *Scan* e *Value*, para a definição de como será feita a interpretação do objeto para inserção e obtenção dos dados, como visto no Algorítmo 4.2 para o tipo *VARCHAR*:

```
1  package models
2
3  import (
4    "database/sql/driver"
5    "errors"
6  )
7
8  // VARCHAR : Tipo usado para inserir strings ’NULL’ no bd quando vazias,
       e o valor quando preenchidas
9  type VARCHAR string
10
11 // Value VARCHAR : Usado para inserir o valor do tipo ’VARCHAR’ no bd
12 func (vc VARCHAR) Value() (driver.Value, error) {
13   if vc == "" {
14     return nil, nil
15   }
16   return string(vc), nil
17 }
18
19 // Scan VARCHAR : Usado para a leitura do valor no bd, convertendo
       corretamente para o tipo ’VARCHAR’
20 func (vc *VARCHAR) Scan(v interface{}) error {
21   switch vt := v.(type) {
22   case string:
23     // Converte o valor lido
```

```
24      *vc = VARCHAR(vt)
25    default:
26      return errors.New("Type handling error")
27
28    }
29    return nil
30  }
```

Algoritmo 4.2: Exemplo de Tipo Customizado para Manipulação do GORM

Essa utilização de tipos customizados é de extrema importância, principalmente devido ao comportamento padrão do GORM para alguns tipos nativos do *Go*. GORM possui a política de atualizar os tipos com seus valores iniciais, caso não tenham sido preenchidos, então um *int* é preenchido com o valor zero, *string* como *string* vazia, por exemplo. Devido a esse comportamento, caso uma tabela possua relações com chave estrangeira, iriam ocorrer erros ao atualizar a tabela, visto que GORM preencheria um valor, como *string* vazia, ao invés de *NULL*, mesmo que não exista uma relação.

Dessa forma, a utilização de tipos customizados remedia esse problema, traz mais flexibilidade para a aplicação e também torna a manipulação de alguns dados mais simples, como por exemplo na conversão mútua na leitura ou escrita do tipo EWKB para valores *float64*. EWKB é um formato utilizado pelo *PostGIS* para o armazenamento das coordenadas geográficas, e no *back-end*, a localização é manipulada em um objeto que possui variáveis do tipo *float64*, como visto abaixo no Algorítmo 4.3. Com esses tipos customizados, foi feito um novo tipo em que a conversão entre EWKB e *float64* ocorre de forma direta e automática devido aos métodos implementados na leitura e escrita do BD, mantendo essa comutação invisível para o restante da aplicação, como visto no exemplo de conversão de escrita e leitura do banco de dados para o tipo *POINT* no Algorítmo 4.4.

```
1 package models
2
3 // O tipo Location descreve a estrutura de coordenadas geograficas Lat (
    latitude) e Lng (Longitude)
4 type Location struct {
5   Lat float64 `gorm:"-" json:"lat,omitempty" validate:"required,latitude
    " pt_BR:"Latitude"`
6   Lng float64 `gorm:"-" json:"lng,omitempty" validate:"required,
    longitude"  pt_BR:"Longitude"`
7 }
```

Algoritmo 4.3: Exemplo do Modelo de Localização Utilizado

```
1 ...
2 // POINT : tipo usado para converter de um ponto EWKB para o tipo do
    modelo Location
```

```
type POINT Location

// Value POINT : funcao usada para converter o modelo Location e inserir
    valores do tipo 'EWKB' no bd
func (p POINT) Value() (driver.Value, error) {
  point := fmt.Sprintf("POINT(%g %g)", p.Lng, p.Lat)
  return point, nil
}

// Scan POINT : funcao usada para conversao do tipo 'EWKB' para o modelo
    Location na leitura do bd
func (p *POINT) Scan(v interface{}) error {
  switch vt := v.(type) {
  case []uint8:
    // caso o ponto esteja vazio
    if len(vt) == 0 {
      *&p.Lat = 0
      *&p.Lng = 0
    } else {
      // converte cada item uint8 para string
      var itemList string
      for i := 0; i < len(vt); i++ {
        itemList += string(vt[i])
      }

      // Transforma o tipo 'POINT' para Location
      item, err := decodePoint(itemList)

      if err != nil {
        return err
      }

      *p = POINT(item)
    }
  default:
    return errors.New("Type handling error")

  }
  return nil
}
...
```

Algoritmo 4.4: Exemplo do Tipo Customizado *POINT*

4.5 Resumo Conclusivo

O servidor *back-end* foi desenvolvido de acordo com o padrão de arquitetura API REST, possuindo uma interface bem definida para a comunicação com outros serviços pelos *endpoints* disponíveis.

Para a implementação do servidor, foi utilizada a linguagem de programação *Go* [6], e feita uma organização na estrutura do código de forma que os pacotes fiquem desacoplados, para facilitar na manutenção, expansão e mudanças futuras do sistema.

Por fim, o banco de dados foi implementado com a utilização da biblioteca GORM [23] do *Go*, o sistema de gerenciamento PostgreSQL [24] e a extensão PostGIS [25], para o uso do padrão SQL na manipulação de coordenadas geográficas.

Capítulo 5

Prova de Conceito

Neste Capítulo é descrito como foi feito o desenvolvimento da API REST do sistema de cadastro de redes de fibras ópticas. Também são apresentadas algumas funcionalidades adicionais implementadas e as vantagens e desvantagens.

5.1 Fluxo dos Dados

A implementação da parte de comunicação HTTP foi feita com a utilização do pacote *Gin* [26] do *Go* [6]. O *Gin* é um framework utilizado para a criação de aplicações *web* e microsserviços, contendo nativamente suporte a roteamento, adição de *middlewares* e renderização de dados. Sua utilização traz uma maior simplicidade na manipulação das requisições e na modularização da aplicação. Além disso, possui a vantagem de criar uma nova instância de rotina no *Go* para cada requisição, dinamicamente, tornando possível a recepção de requisições simultâneas.

O pacote *server* é o ponto de entrada dos dados no *back-end*. Nele, foram configurados dois *middlewares*: um para lidar com a recuperação do servidor caso ocorra um erro não tratado, já retornando como resposta *Internal Server Error*, e outro para a configuração do CORS, que é um mecanismo de segurança de navegadores de internet que impede requisições para um domínio diferente do de origem, que deve ser configurado devido a separação do servidor *back-end* do *front-end*.

Na recuperação do servidor foi utilizado um método nativo do *Gin*, já para o CORS foram adicionados alguns cabeçalhos como o *Access-Control-Allow-Origin*, *Access-Control-Allow-Headers* e o *Access-Control-Allow-Methods*, que são inseridos em todas as requisições para identificar os recursos permitidos e liberar o acesso de domínios diferentes, para a utilização do *front-end*.

Então, utilizando o roteamento do *Gin*, foram definidos os *endpoints* de todos os elementos, incluindo os parâmetros de *id* existentes na URL, juntamente com a definição

dos métodos HTTP que cada rota terá, e a função interna do pacote *controller* que será executada. Abaixo é mostrado no Algoritmo 5.1 um exemplo de implementação de um servidor HTTP com o *Gin*, e no Algoritmo 5.2 um exemplo de roteamento de um elemento genérico.

```
...
    // Utiliza o Gin no modo de depuracao
    gin.SetMode(gin.DebugMode)

    // Cria uma nova instancia principal do Gin
    router := gin.New()

    // Habilita o depurador do Gin
    router.Use(gin.Logger())

    // Middleware de recuperacao, que recupera o servidor e retorna 500 se ocorrer um panico
    router.Use(gin.Recovery())

    // Middleware utilizado para o ajuste de CORS
    router.Use(CORSMiddleware())

    // Criacao do roteamento
    createHandler(router)

    // Execucao do servidor na pota 3333
    router.Run(":3333")
...
    // CORSMiddleware : middleware usado para habilitar os cabecalhos do CORS
    func CORSMiddleware() gin.HandlerFunc {
      return func(c *gin.Context) {
        c.Writer.Header().Set("Access-Control-Allow-Origin", "*")
        c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-Type")
        c.Writer.Header().Set("Access-Control-Allow-Methods", "POST, OPTIONS, GET, PUT, DELETE")

        if c.Request.Method == "OPTIONS" {
          // Status 'No Content', por nao retornar nenhum dado (metodo nao implementado)
          c.AbortWithStatus(204)
          return
        }

```

```
36          c.Next()
37    }
38 ...
39 }
```

Algoritmo 5.1: Exemplo de Implementação do Servidor HTTP

```
1  ...
2  func createHandler(router *gin.Engine) {
3  ...
4      // Adicionar rotas do 'element'
5      elementRoutes(router)
6  ...
7  }
8  ...
9  func elementRoutes(router *gin.Engine) {
10   // Rota de obtencao de todos elementos
11   router.GET("/elementos", controller.GetAllElements)
12
13   // Rota de obtencao de um elemento por 'id'
14   router.GET("/elementos/:id", controller.GetElement)
15
16   // Rota de registrar um elemento
17   router.POST("/elementos", controller.RegisterElement)
18
19   // Rota de editar um elemento por 'id'
20   router.PUT("/elementos/:id", controller.EditElement)
21
22   // Rota de deletar multiplos elementos
23   router.DELETE("/elementos", controller.DeleteElementsBulk)
24
25   // Rota de deletar um elemento por 'id'
26   router.DELETE("/elementos/:id", controller.DeleteElement)
27 }
28 ...
29 }
```

Algoritmo 5.2: Exemplo de Roteamento

O pacote *controller* é responsável pela validação dos dados recebidos na requisição pelo metadado JSON, a transposição desses dados para os objetos *Go*, e por fim a chamada correspondente da função do pacote *services* que irá executar a ação do método HTTP recebido.

A validação dos campos do JSON é feita utilizando o pacote *validator* [27] do *Go*. Para a identificação do que deve ser validado, em todas as interfaces dos modelos são colocadas a etiqueta de validação *validate*, juntamente aos itens que devem ser validados. São

preenchidos nos modelos item a item por exemplo a verificação do tamanho de *strings*, tipo da variável e várias outras checagens, que foram usadas de acordo com a documentação [28].

A transposição do JSON para os objetos do *Go* é feita com o auxílio da etiqueta *json* nas variáveis. Caso ocorra algum erro em alguma etapa, o *controller* retorna para a instância do *Gin* um *Bad Request* juntamente com um JSON preenchido com a descrição do erro em português, devido ao uso da etiqueta de tradução *pt_BR* para as variáveis, implementado com a extensão de tradução *translations* do pacote *validator*. Segue abaixo no Algoritmo 5.3 um exemplo com o uso de todas as etiquetas em conjunto:

```
1 ...
2   IDTabela         VARCHAR     `gorm:"column:id_tabela; primary_key; type
     :varchar(15); not null" json:"id_tabela,omitempty" validate:"required
     ,max=15" pt_BR:"ID da Tabela"`
3   DadoFloat        float64     `gorm:"column:dado_float; json:"dado_float,
     omitempty" validate:"omitempty,numeric" pt_BR:"Dado Float"`
4 ...
```

Algoritmo 5.3: Exemplo de Todas Etiquetas Usadas na Interface de Objetos Go

Após a validação, o *controller* executa o serviço do pacote *services* adequado ao método chamado, que lida com as regras de negócio da aplicação e retorna os dados requisitados, editados, inseridos ou se ocorreu um erro interno ou do usuário na operação. Então o *controller* retorna para a instância do *Gin* o código de *status* e o JSON adequado a cada situação, com o erro ou com os dados da operação.

5.2 Regras de Negócio

No pacote *services*, foram implementadas todas as regras de negócio das ações requisitadas pelo cliente. Para cada método do *controller* foi feita uma função de serviço, que se encarrega de todos os detalhes, desde uma análise mais aprofundada dos dados à chamada das funções do banco de dados.

De uma forma geral, a maior complexidade está na adição ou edição de um objeto, que deve checar previamente no banco de dados se o elemento já existe, atualizar as relações dele com outros itens da rede de fibras, como por exemplo a relação de um trecho de cabo com uma caixa de emenda, e só depois fazer a adição ou edição no banco de dados.

A necessidade da criação de um serviço para a adição de relações surgiu devido a um pedido feito pela equipe do *front-end*, que preferia um JSON simplificado com apenas o envio do objeto e do *id* dos itens que se relacionavam com ele, ao invés do envio completo de todos os dados dos itens de relacionamento. Devido a forma que o GORM faz os

relacionamentos no banco de dados, é necessário buscar antes esses elementos por *id*, para então preenchê-los com todos seus dados no objeto que os relaciona.

O processo de obtenção de todos os itens ou de um dado por *id* é feito com apenas uma chamada da função respectiva do banco de dados, já a parte de deleção por *id* ou deleção múltipla é feita inicialmente por uma verificação da existência dos itens, para apenas depois efetivamente deletar.

Por fim, o pacote *services* também lida com todos os tipos de erros que podem ocorrer durante as checagens e operações, desde erros internos quanto a erros do usuário. Retornando para o *controller* o tipo de erro ocorrido com mensagens customizadas para os erros de usuário e genéricas para os erros internos, sem revelar a lógica interna.

5.3 Banco de dados

Como visto no Capítulo 4, foi utilizado o pacote GORM do *Go*, e configurado no pacote *models* as características de cada variável no banco de dados, além da criação de tipos customizados. Já no pacote *database*, foram implementadas as funções de manipulação do banco de dados e também parte das configurações.

Para cada operação de leitura, escrita, edição ou deleção, foi feita uma função para lidar com a comunicação do banco de dados. Essas funções utilizam os métodos nativos do GORM, que recebem o objeto completo do *Go* para a manipulação, e todas retornam um erro, caso ocorra.

Na obtenção dos elementos, utiliza-se o método *Preload* para carregar os dados relacionados a aquele elemento, e em conjunto, são usados os métodos *Where* e *First* ou *Find*, dependendo se a busca é por um *id* específico ou de todos os dados.

A criação, atualização e deleção utilizam os métodos respectivos *Create*, *Save* e *Delete*, utilizando também em conjunto o *Where* para especificar o item por *id* na atualização e deleção.

Por fim, foi colocado dentro do pacote *database* um arquivo de configurações, para a realização da conexão com o banco de dados, também a especificação manual de chaves estrangeiras de itens específicos e de tabelas de relacionamentos que são geradas pelo GORM. No Algoritmo 5.4 é mostrado um exemplo de uso do GORM para a obtenção de um elemento por *id*.

```
1 ...
2     // Leitura de um dado por 'id'
3     // db - variavel global com a instancia de conexao com o postgres
4   err = db.Preload("relacao-X").Where("id_tabela = ?", id).First(&
    element).Error
```

```
5 ...
```

Algoritmo 5.4: Exemplo de Uso das Funções do GORM

5.4 Visão Geral

A Figura 5.1 exemplifica de uma forma simples o fluxo do dado na aplicação. Recapitulando o que foi apresentado, os *endpoints* de um *request* que chega na aplicação é analisado pelo pacote *server* e são utilizados os *middlewares*, então é executado no *controller* uma análise dos dados e a transposição do JSON para os objetos *Go*. Em seguida é chamada a função correspondente do pacote *services*, que executa as regras de negócio e a manipulação do banco de dados pelas funções do pacote *database*. Por fim, o pacote *models* é uma definição de como é feita a transposição do JSON para os objetos *Go*, a validação e a utilização do ORM, utilizados respectivamente pelos pacotes *controller, services* e *database*. Em todas as etapas existem a checagem de erros e é apenas passado para a próxima etapa caso nenhum erro tenha ocorrido. No evento de um erro, é identificado pela etapa em que ele ocorreu e retornado ao cliente um JSON contendo os detalhes, caso seja um *bad request*, ou uma mensagem customizada sobre o erro ocorrido.

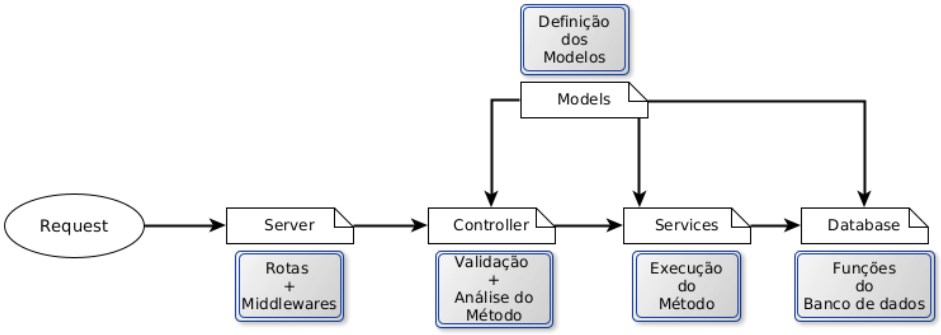


Figura 5.1: Representação do Fluxo do Dado nos Pacotes Implementados.

Dessa forma, todo *request* que é feito, seja para o cadastro de um novo elemento, manipulação dos dados ou apenas para a obtenção dos elementos, segue esse fluxo. A verificação que ocorre em todas as etapas é extremamente importante devido a quantidade de relações existentes entre os elementos, e por esse motivo, não podem ser adicionados em qualquer ordem. Caso o cliente tente fazer uma requisição sem respeitar a ordem dos modelos, que é feito primeiro com a adição dos elementos principais, para apenas em seguida a adição das relações com aquele elemento, como por exemplo adicionar primeiro um cabo de acesso ou uma fibra sem antes adicionar um trecho de cabo, é retornado um

bad request e na mensagem de erro será mencionado que a relação nos dados do JSON enviado está incorreta.

5.5 Dimensão da Aplicação

Na solução desenvolvida, foram implementados todos os modelos e as relações existentes, como visto na Figura 3.2. Pela quantidade de *endpoints* existentes em cada elemento, foram feitos apenas o CRUD completo de alguns deles. Segue abaixo uma visão da dimensão da aplicação implementada pela equipe do *back-end*:

- 27 modelos de objetos;
- 19 modelos com CRUDs completos;
- 57 relações entre os objetos;
- 98 *endpoints* criados;

5.6 Funcionalidades Adicionais

Como mencionado no Capítulo 2, uma das principais características de uma API é a possibilidade de uso por mais de uma aplicação. Devido a uma particularidade da associação GigaCandanga de registrar parte dos dados de uma instituição em seu sistema administrativo, foi feita uma conexão com a API de seu sistema para a obtenção desses dados, organizada e implementada em um pacote criado chamado de *external*. Dessa forma, os dados cadastrados no sistema administrativo são fornecidos também pelo sistema de gerenciamento, mostrando maiores informações nos objetos. Além disso, ficaram mais simples novas conexões com outras APIs se necessário no futuro, devido a implementação do pacote *external*. Essa utilização exemplifica de forma prática a flexibilidade de uma API, podendo ocorrer futuramente também no sistema de gerenciamento, para outros sistemas.

Também foi implementada uma rota para importar os dados dos arquivos do Google Earth (*KML/KMZ*) existentes, que são organizados seguindo o padrão XML. Essa importação foi feita no pacote criado chamado de *imports*, possuindo um fluxo similar ao de uma rota comum, sendo feita primeiramente uma análise dos dados do arquivo enviado, verificando se são válidos e se não são duplicados no banco de dados, e caso tudo esteja correto, o ultimo passo é a inserção, adicionando também a relação existente entre os objetos. Os erros são tratados de forma similar às rotas, analisando o arquivo por completo e indicando se existe algum dado inválido ou fora do padrão do arquivo, por uma mensagem de JSON, com os elementos incorretos.

Para a verificação dos dados salvos pelo servidor do *back-end*, foi utilizada a plataforma de administração de banco de dados *pdAdmin*. Nela é possível navegar por todas as tabelas, verificar atributos e manipular os dados. Além de possuir uma visualização nativa de mapa para os tipos geográficos, como visto na Figura 5.2.

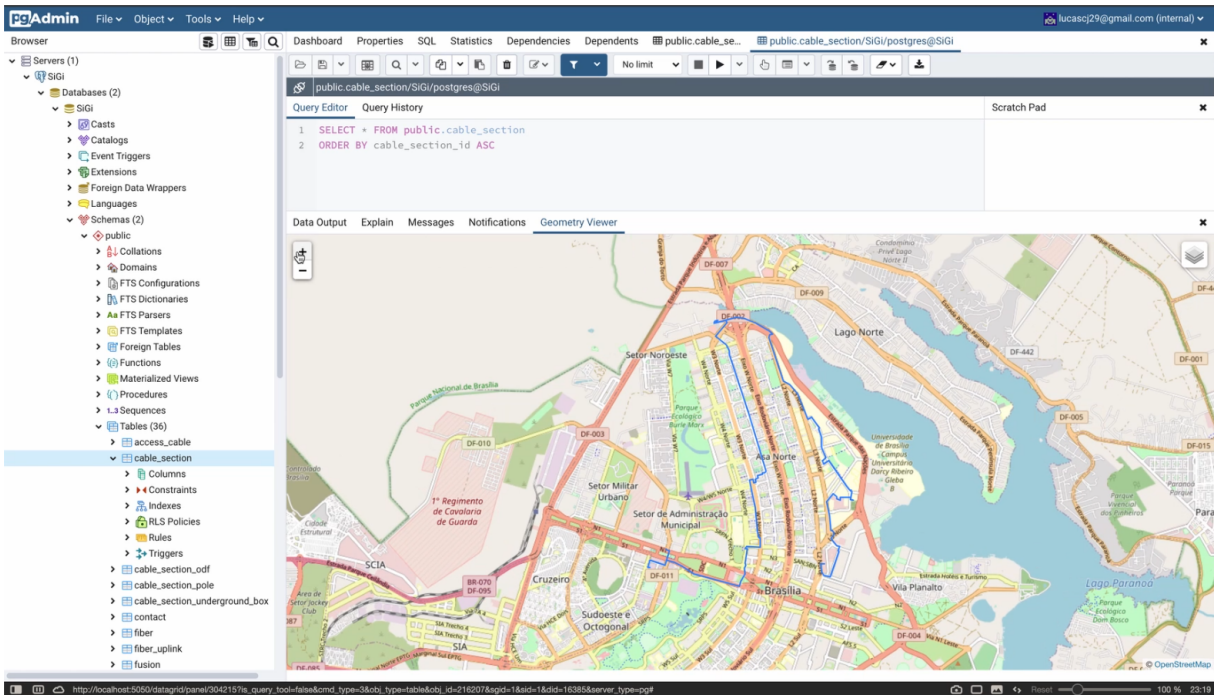


Figura 5.2: Ferramenta de Administração do Banco de Dados *pgAdmin*.

Por fim, para uma melhor organização e garantia de implantação do servidor em qualquer máquina, foi desenvolvido um arquivo de *environment* e utilizado o *Docker* [29]. O arquivo de *environment* contém dados básicos para a execução do servidor, como as portas de implantação do banco de dados ou da comunicação HTTP, credenciais e outros dados importantes de configuração que são lidos no início da execução do programa. Já o *Docker* é usado para garantir um ambiente isolado para a aplicação, contendo apenas os recursos necessários para a execução.

No *Docker* foi configurado um *container* contendo as imagens da aplicação em *Go*, *pgAdmin*, *PostgreSQL* e um volume de dados separado para o armazenamento do banco de dados, de forma a facilitar *backups* e recuperação dos dados. A Figura 5.3 mostra as três imagens do *container* em execução.

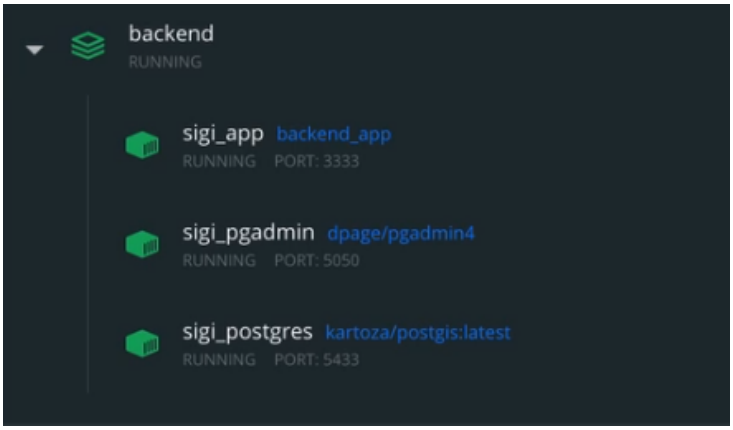


Figura 5.3: Representação das Imagens do Docker Usadas.

5.7 Vantagens e Desvantagens

API REST é um padrão de arquitetura bastante usado e ideal para a criação de aplicações distribuídas. Como é um padrão, sua implementação pode ser feita de forma livre, sendo escolhida a linguagem de programação *Go* na implementação deste trabalho.

Go possui a vantagem da detecção de erros em tempo de compilação, velocidade na execução das rotinas, que pode ser visto em vários testes de performance relatados em livros ou no próprio site de distribuição dos pacotes, como visto a comparação do *Gin* em sua documentação [26]. Além disso, a simplicidade e facilidade da criação da API foi uma grande vantagem.

Também há a vantagem no uso de um ORM para a manipulação dos dados no banco de dados, que facilita a criação das tabelas, simplifica as requisições e trás uma maior organização no código por manter contida a lógica de acesso a dados. Ainda que as requisições possam não ser tão otimizadas em comparação da criação de SQL por uma pessoa, em projetos muito grandes com equipes pequenas, compensa a perda de performance.

Embora a arquitetura usada do padrão API REST seja bastante difundida, há novas tecnologias, como os microsserviços [30], que segmentam as regras de negócio em vários serviços independentes que comunicam entre si usando APIs, tornando a aplicação mais flexível à mudanças de arquitetura interna de cada serviço, também trazendo uma facilidade na escalabilidade, que pode ser feita apenas na parte da aplicação em que ocorre o maior pico.

Apesar dos benefícios da linguagem de programação *Go*, sua escolha trouxe vários empecilhos, principalmente, com relação a busca de cursos ou livros de assuntos específicos online. Por ser uma linguagem nova, e mesmo que possua bastante documentação, nem sempre são detalhadas o suficiente para uma compreensão completa.

5.8 Resumo Conclusivo

A comunicação HTTP do servidor *back-end* foi implementada com a utilização do pacote *Gin* [26], utilizando seus métodos nativos para o uso de middlewares, roteamento e renderização de dados.

O fluxo de processamento de uma requisição possui quatro etapas. A primeira é a análise do método de requisição e da rota pelo *Gin*, o segundo é uma análise dos dados pelo pacote *controller*, o terceiro é a execução da ação pelo pacote *services* e por fim é a manipulação do banco de dados pelo pacote *database*, utilizando as funções do ORM GORM. Em todas as etapas há a checagem de erros, sejam internos ou de requisição, retornando ao final o status correspondente da execução, os dados ou a mensagem de erro ocorrida, sem revelar a lógica interna implementada.

Capítulo 6

Conclusão

O objetivo principal desse trabalho foi a criação de uma camada de manipulação do cadastro da rede de fibras ópticas da associação GigaCandanga, com a finalidade de unificar toda a rede em apenas um banco de dados. O sistema deve ser simples e intuitivo de se utilizar, possuir a possibilidade de expansão para diferentes dispositivos, evitar falhas humanas e que tenha uma forma de armazenamento dos dados segura. Além disso, o sistema deve prevenir a inserção de dados com erros e adicionar a relação entre os elementos.

Para alcançar todos os objetivos estabelecidos, foi feito um *web service* com o padrão de arquitetura API REST desenvolvido em *Go* [6] que centralizou a lógica do sistema em apenas um servidor, possuindo rotas bem definidas para a manipulação de cada elemento da rede individualmente. Além disso, devido a escolha das tecnologias e também da organização feita, a adição de novas funcionalidades e a mudança do uso de pacotes ou da lógica de alguma etapa são facilmente implementadas.

A necessidade de unificação dos dados em apenas um sistema foi suprida com essa abordagem. O sistema tem a capacidade de armazenar e manipular os dados da rede de fibras ópticas no banco de dados PostgreSQL [24] utilizando o ORM GORM [23] e a extensão PostGIS [25], além de relacionar os elementos e também importar os dados já existentes de *KML* ou *KMZ*. Dessa forma, evita-se possíveis perdas relacionadas à forma antiga de distribuição dos dados em múltiplas ferramentas.

O sistema possui etapas no fluxo da requisição bem definidas, separando as camadas de comunicação, verificação e manipulação dos dados. São feitos para todos os elementos a verificação do tipo de variável e outras particularidades, evitando erros não intencionais. Ademais, a utilização de uma API flexibiliza a distribuição da aplicação para diferentes meios, podendo facilmente ser expandida para o *mobile*, por exemplo.

Este trabalho mostrou que a arquitetura e modelagem escolhidas foram adequadas para o sistema de gerenciamento de redes de fibras ópticas, sendo alcançados os objetivos

estabelecidos, embora não tenha sido possível a conclusão de implementação de todas as rotas dos elementos, devido a quantidade de elementos da rede e do tempo limitado, foram feitos os itens principais, sendo possível uma utilização inicial do sistema.

Como perspectivas futuras, deve ser considerado o uso de autenticação na comunicação das rotas, como o JSON Web Token (JWT) por exemplo, para que a comunicação com a API seja restrita apenas às pessoas autorizadas. Também deve ser comparada a implementação desenvolvida neste trabalho com a utilização de microsserviços [30], que podem trazer vantagens no futuro do sistema, principalmente por ser possível escalar as funcionalidades de forma separada.

Referências

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

[2] 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, 6, 26

[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, 6, 26

[4] JORGE, Lucas. PROJETO E ARQUITETURA DE API REST PARA SISTEMA DE MONITORAMENTO DE REDES ÓPTICAS. Orientador: André Drummond. 2020. 43 f. Monografia (Graduação) – Engenharia de Computação, CIC, UnB, Brasília. 2020. 2, 6, 26

[5] *Microsoft excel*. `https://www.microsoft.com/pt-br/microsoft-365/excel`. Acesso em: 19/12/2020. 3

[6] *The go programming language*. `https://golang.org/`. Acesso em: 3/11/2020. 6, 20, 25, 34, 35, 45

[7] *Google*. `https://about.google/`. Acesso em: 3/11/2020. 6

[8] *Gin web framework*. `https://gin-gonic.com/`. Acesso em: 3/11/2020. 6

[9] *Postgresql: The world's most advanced open source relational database*. `https://www.postgresql.org/`. Acesso em: 3/11/2020. 6

[10] *Gorm: The fantastic orm library for golang*. `https://gorm.io/`. Acesso em: 3/11/2020. 6

[11] Kalin, Martin: *Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction*. O'Reilly Media., 1ª edição, 2009, ISBN 978-0-596-52112-7. 8

[12] Reddy, Martin: *API Design for C++*. Morgan Kaufmann Publishers., 1ª edição, 2011, ISBN 978-0-12-385003-4. 8

[13] Saudate, Alexandre: *REST: Construa API's inteligentes de maneira simples*. Casa do Código., 1ª edição, 2013, ISBN 9788566250374. 9

[14] Yellavula, Naren: *Hands-On RESTful Web Services with Go*. Packt Publishing., 2ª edição, 2020, ISBN 978-1-83864-357-7. 10, 11, 12

[15] Mehta, Vijay: *Pro LINQ Object Relational Mapping with C# 2008*. Apress; 1st Edition (July 7, 2008)., 1ª edição, 2008, ISBN 978-1-59059-965-5. 12, 13

[16] Raiturkar, Jyotiswarup: *Hands-On Software Architecture with Golang: Design and architect highly scalable and robust applications using Go*. Packt Publishing Ltd., 1ª edição, 2018, ISBN 978-1-78862-259-2. 12

[17] Halpin, Terry: *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufmann Publishers., 1ª edição, 2001, ISBN 1-55860-672-6. 13

[18] Satpathy, Tridibesh: *A Guide to the SCRUM BODY OF KNOWLEDGE*. SCRUMstudy™, a brand of VMEdu, Inc., 3ª edição, 2017, ISBN 978-0-9899252-0-4. 15

[19] *Gigacandanga: Rede comep*. `https://gigacandanga.net.br/redecomep/`. Acesso em: 22/10/2020. 16

[20] Cxhzxhyis: *Cabo de fibra óptica elétrica de cabo de rede do computador - fibra ótica*. `https://www.gratispng.com/png-jfshq3/`, [Online; Acesso em: 20/12/2020]. 16

[21] GigaCandanga. [Imagem de arquivo da GigaCandanga]. 17, 18, 19

[22] *The go programming language: Specification*. `https://golang.org/ref/spec`. Acesso em: 22/10/2020. 29

[23] *Gorm: Guides*. `https://gorm.io/docs/`. Acesso em: 26/10/2020. 30, 34, 45

[24] *What is postgresql?* `https://www.postgresql.org/docs/13/intro-whatis.html`. Acesso em: 26/10/2020. 30, 34, 45

[25] *Postgis*. `https://postgis.net/`. Acesso em: 26/10/2020. 30, 34, 45

[26] *Gin: Guides*. `https://github.com/gin-gonic/gin`. Acesso em: 9/11/2020. 35, 43, 44

[27] *Package validator*. `https://github.com/go-playground/validator`. Acesso em: 10/11/2020. 37

[28] *Validator docs*. `https://godoc.org/gopkg.in/go-playground/validator.v9`. Acesso em: 10/11/2020. 38

[29] *Why docker*. `https://www.docker.com/why-docker`. Acesso em: 11/11/2020. 42

[30] *O que são microsserviços?* `https://aws.amazon.com/pt/microservices/`. Acesso em: 12/11/2020. 43, 46