CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO ITAJAÍ - UNIDAVI

PEDRO ARTUR BUZZI PEREIRA

PROTÓTIPO DE GERADOR DE WEBSERVICES POR INTERFACE GRÁFICA

RIO DO SUL
2022

CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO ITAJAÍ - UNIDAVI

PEDRO ARTUR BUZZI PEREIRA

PROTÓTIPO DE GERADOR DE WEBSERVICES POR INTERFACE GRÁFICA

Trabalho de Conclusão de Curso a ser apresentado ao curso de Sistemas da Informação, da Área das Ciências Naturais, da Computação e das Engenharias, do Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí, como condição parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Prof. Orientador: M.e Marciel de Liz Santos

RIO DO SUL
2022

CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO ITAJAÍ - UNIDAVI

PEDRO ARTUR BUZZI PEREIRA

PROTÓTIPO DE GERADOR DE WEBSERVICES POR INTERFACE GRÁFICA

Trabalho de Conclusão de Curso a ser apresentado ao curso de Sistemas da Informação, da Área das Ciências Naturais, da Computação e das Engenharias, do Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí- UNIDAVI, a ser apreciado pela Banca Examinadora, formada por:

_____

Professor Orientador: M.e Marciel de Liz Santos

Banca Examinadora:

_____

Prof. M.e Fernando Andrade Bastos

_____

Prof. M.e Jeancarlo Visentainer

Rio do Sul, 29 de dezembro de 2022

Dedico este trabalho a todos os desenvolvedores que porventura poderão se beneficiar dessa iniciativa originada a partir de um trabalho acadêmico. Também dedico este a instituição UNIDAVI.

AGRADECIMENTOS

Um agradecimento a Deus por me permitir viver esses momentos, um agradecimento mais do que especial aos meus pais que sempre me apoiaram nas minhas decisões e me auxiliaram nos momentos difíceis. E um agradecimento ao meu orientador que sempre me ajudou nos rumos a tomar nesse trabalho quando estava em dúvidas.

RESUMO

Nos dias atuais no mercado de trabalho de desenvolvimento de software, os desenvolvedores são cada vez mais requisitados. São essas pessoas que garantem o bom funcionamento de sistemas utilizados por milhões de pessoas. No entanto a maior parte das vagas nas empresas cobram cada vez mais especialidades das mais variadas possível, o que tem dificultado a contratação e a busca por essas vagas, o que gera cenário dificultoso para os iniciantes. Com essas premissas em mente, foi planejado o desenvolvimento de uma plataforma capaz de automatizar uma pequena parte do trabalho que esses pofissionais realizam e assim conseguir poupar tempo e melhorar o fluxo de trabalho durante as rotinas que já são bem demoradas. Com uma plataforma que automatiza a consulta de uma base de dados usando um dicionário de dados, construída com NodeJS no back-end e React no front-end é possível criar essa fermenta que tem o potencial de ser muito mais conveniente para os programadores nos seus trabalhos ou em projetos pessoais.

Palavras-Chave: Consulta, dicionário, REST, JSON, web servise.

ABSTRACT

Nowadays software development job market, developers are increasingly in demand. It is these people who ensure the proper functioning of systems used by millions of people. However, most vacancies in companies increasingly demand specialties of the most varied possible, which has made it difficult to hire and search for these vacancies. Thus generating a more difficult scenario for beginners. With these objectives in mind, the development of a platform capable of automating a small part of the work that these coders go through was planned, and thus save their time to improve the workflow during the routine that is no longer short. With a platform that automates querying a database using a data dictionary, and made with NodeJS on the backend and React on the frontend, it is possible to create this ferment that has the potential to be much more convenient for programmers. in their work or personal projects.

Keywords: Query, dictionary, REST, JSON, web service.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CLI	Command-Line Interface
CMS	Content Management System
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheet
ERP	Enterprise Resource Planning
HTM	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
LSP	Linguagem Senior de Programação
MA	Modelo de Abreviatura
NPM	Noda Package Manager
ORM	Object Relational Mapping
OSI	Open System Interconnection
REST	Representational State Transfer
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator
WEB	World Wide Web

SUMÁRIO

1. I NTRODUÇÃO

O processo de criação de webservices por muitas vezes pode demandar um trabalho muito extenso. Apesar de muitas vezes ser trabalhoso, é uma das tarefas mais recorrentes para os desenvolvedores. Com um padrão estrutural bem estabelecido, será possível reduzir muito tempo de desenvolvimento necessário para o cumprimento dessa tarefa. Assim aumentando a produtividade no desenvolvimento e focar mais nos requisitos do sistema. A ferramenta proposta, será uma forma de abstrair e disponibilizar recursos de uma aplicação existente para um WebService REST, para ser utilizados da forma que o desenvolvedor requerer.

Com a tecnologia ficando cada vez mais moderna e principalmente prática se se utilizar, faz sentido a automatização de partes do trabalho dos desenvolvedores. Esse trabalho se foca principalmente no ato de criar um webservice RESTFUL.

Com o intuito de facilitar a vida do programador, desenvolveu-se uma plataforma fácil de se utilizar cuja sua principal funcionalidade é automatizar a criação de webservices para a consulta dinâmica de uma base dados, basicamente escolhendo os campos desejado e consumindo esses mesmos dados com a finalidade que se deseja.

1.1 PROBLEMA DE PESQUISA

É possível criar uma ferramenta que auxilie na geração de um Web Service pronto para ser utilizado?

1.2 OBJETIVOS

1.2.1 Geral

- Criar uma aplicação de interface gráfica que auxilie desenvolvedores na criação de Web Services.

1.2.2 Específicos

- Levantar requisitos do sistema.
- Criar interface gráfica com React para a construção dos serviços.
- Criar Back-End em NodeJS para as requisições HTTP.

- Criar um dicionário de dados da base de consulta.
- Transformar objetos JSON em comandos SQL para consulta.
- Centralizar o acesso aos WebServices mediante a um código identificador.

1.3 JUSTIFICATIVA

O tempo gasto por empresas de pequeno porte, sem uma estrutura e fluxo de trabalho bem estabelecidas, tornam o desenvolvimento de WebServices para integrações ou para uso interno mais complicados e demorados.

Será que essas empresas gostariam mais de focar no negócio em vez de desenvolver uma estrutura complexa que leva tempo para ser elaborada? Essas mesmas empresas na verdade precisam gastar muito tempo de trabalho para resolver problemas e desafios que já poderiam ser resolvidos. Assim podendo focar somente no produto.

Com o objetivo de auxiliar na resolução rápida desses problemas, esse trabalho propõe a criação de uma ferramenta auxiliar, com o objetivo de criar e publicar webservices REST para serem consumidos por outras aplicações de forma rápida e dinâmica.

Tendo em vista essa necessidade de uma maior automação nos processos rotineiros dos programadores nas empresas, a intenção de criar uma aplicação para aumentar a produtividade e flexibilidades das aplicações desenvolvidas por esses desenvolvedores, percebesse como poderia ser vantajoso um sistema que atenda exatamente essa demanda.

O protótipo de aplicativo que foi desenvolvido com essa finalidade, vai atuar como um software para automatizar essa tarefa especifica que tem a pretensão de impactar positivamente a vida dos profissionais do ramo de software.

2. REFERENCIAL TEÓRICO

Esse capítulo trata-se de uma compilação dos principais componentes de softwares que serão utilizados no desenvolvimento do trabalho. As tecnologias foram escolhidas pensando na maior produtividade e aproveitamento de conceitos já pré-estabelecidos. As principais tecnologias abordadas são Node e React, com os quais as telas e funcionalidades do sistema serão construídas.

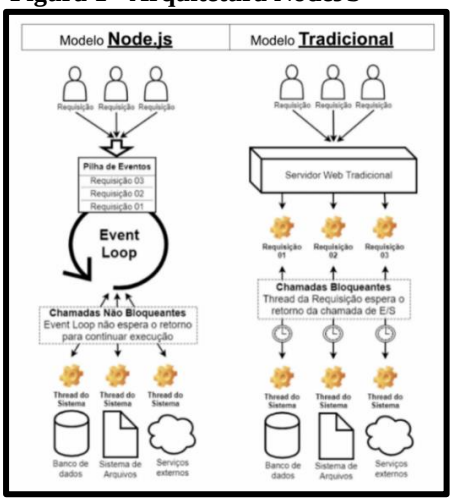
2.1 NODE

Segundo Pereira (2013, p.1), “Os sistemas para web desenvolvidos sobre plataforma .NET, Java, PHP, Ruby ou Python possuem uma característica em comum: eles paralisam um processamento enquanto utilizam I/O do servidor. Essa paralização é conhecida como modelo bloqueante (Blocking Thread)”.

Caio Ribeiro Pereira (2013, p.1) também diz, “Tarefas como enviar e-mail, consultar o banco de dados, leitura em disco, são exemplos de tarefas que gastam uma grande fatia desse tempo, bloqueando o sistema inteiro enquanto não são finalizadas.”. Como mostra na Figura 1.

> Esta tecnologia possui um modelo inovador, sua arquitetura é totalmente non-blocking thread (não-bloqueante), apresentando uma boa performance com consumo de memória e utilizando ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta carga de processamento. Usuários de sistemas Node estão livres de aguardarem por muito tempo o resultado de seus processos, e principalmente não sofrerão de dead-locks no sistema, porque nada bloqueia em sua plataforma e desenvolver sistemas nesse paradigma é simples e prático. (PEREIRA, 2013, p.3)

Figura 1 - Arquitetura NodeJS



Fonte: https://www.opus-software.com.br/node-js/

Caio (2013, p.2) também aponta que, “Suas aplicações serão single-thread, ou seja, cada aplicação terá instancia de um único processo. Se você está acostumado a trabalhar com programação concorrente em plataforma multi-thread, infelizmente não será possível com Node.”. Mas ainda assim é possível criar sistemas concorrentes.

Caio (2013, p.3) diz, “[...] existem outras maneiras de se criar um sistema concorrente, como por exemplo, utilizando clusters (assunto a ser explicado no capítulo 9.2), que é um módulo nativo do Node.js e é super fácil de implementá-lo. Outra maneira é utilizar ao máximo a programação assíncrona.”.

2.2 BANCO DE DADOS

Segundo Elmasri e Navathe (2006, p.3), “Os bancos de dados e os sistemas de bancos de dados se tornaram componentes essenciais no cotidiano da sociedade moderna.”. E Esses bancos de dados são essenciais pois, como diz Navathe (2006, p.3), “No decorrer do dia, a maioria de nós se depara com atividades que envolvem alguma interação com os bancos de dados. Por exemplo, se formos ao banco para efetuarmos um depósito ou retirar dinheiro, se fizermos reservas em um hotel ou para a compra de passagens aéreas, se acessarmos o catálogo de uma biblioteca informatizada para consultar uma bibliografia, ou se comprarmos produtos — como livros, brinquedos ou computadores — de um fornecedor por intermédio de sua página Web, muito provavelmente essas atividades envolverão uma pessoa ou um programa de computador que acessará um banco de dados.”

Para Elmasri e Shamkant (2006, p.4), “A construção de um banco de dados é o processo de armazenar os dados em alguma mídia apropriada controlada pelo SGBD.”. E segundo ELMASRI; NAVATHE, 2006, p.4, “A manipulação inclui algumas funções, como pesquisas em banco de dados para recuperar um dado específico, atualização do banco para refletir as mudanças no minimundo e gerar os relatórios dos dados.”.

Segundo Para Ramez Elmasri e Shamkant (2006, p.4), “Outras funções importantes do SGBD são a proteção e a manutenção do banco de dados por longos períodos.” Além disso:

> A proteção inclui a proteção do sistema contra o mau funcionamento ou falhas (crashes) no hardware ou software, e segurança contra acessos não autorizados ou maliciosos. Um banco de dados típico pode ter um ciclo de vida de muitos anos, então, os SGBD devem ser capazes de manter um sistema de banco de dados que permita a evolução dos requisitos que se alteram ao longo do tempo. (ELMASRI; NAVATHE, 2006, p.5)

A cerca das transações, Elmasri e Navathe (2006, p.404) dizem que, “As transações devem possuir algumas propriedades, chamadas propriedades ACID, e elas devem ser impostas pelo controle de concorrência e métodos de restauração do SGBD. As propriedades ACID são as seguintes:”. Conforme o Quadro 1

Quadro 1 – Características ACID

Atomicidade	Uma transação é uma unidade atômica de processamento; ou ela será executada em sua totalidade ou não será de modo nenhum
Preservação de consistência	Uma transação será preservadora de consistência se sua execução completa fizer o banco de dados passar de um estado consistente para outro.
Isolamento	Uma transação deve ser executada como se estivesse isolada das demais. Isto é, a execução de uma transação não deve sofrer interferência de quaisquer outras transações concorrentes.
Durabilidade ou permanência	As mudanças aplicadas ao banco de dados por uma transação efetivada devem persistir no banco de dados. Essas mudanças não devem ser perdidas em razão de uma falha.

Fonte: Elaborado a partir de Elmasri e Navathe (2006)

2.2.1 PostgreSQL

Para Carvalho (2017, p.1), “Tecnologias de banco de dados dão suporte diário para operações e tomadas de decisões nos mais diversos níveis da empresa, da operação à gerência.”

Segundo Carvalho (2017, p.4), “O PostgreSQL é um poderoso sistema gerenciador de banco de dados objeto-relacional de código aberto.”

> Por muito tempo, foi descriminado no mundo dos bancos de dados, e o seu recente aumento de popularidade veio de usuários de outros bancos de dados em busca de um sistema com melhores garantias de confiabilidade, melhores recursos de consulta, mais operação previsível, ou simplesmente querendo algo mais fácil de aprender, entender e usar. Você encontrará no PostgreSQL todas essas coisas citadas e muito mais. (CARVALHO, 2017, p.4)

Carvalho (2017, p4), “Com mais de 15 anos de desenvolvimento ativo e uma arquitetura que comprovadamente ganhou forte reputação de confiabilidade, integridade de dados e conformidade a padrões [...]”. As características do PostgreSQL são apresentadas no Quadro 2.

Quadro 2 – Características do PostgreSQL

Fácil de usar	“comandos SQL do PostgreSQL são consistentes entre si e por padrão. As ferramentas de linha de comando aceitam os mesmos argumentos. Os tipos de dados não têm truncamento silencioso ou outro comportamento estranho. Surpresas são raras, e essa facilidade de utilização”
Seguro	“PostgreSQL é totalmente transacional, incluindo mudanças estruturais destrutivas. Isto significa que você pode tentar qualquer coisa com segurança dentro de uma transação, mesmo a exclusão de dados ou

	alterar estruturas de tabela, com a certeza de que, se você reverter a transação, cada mudança que você fez será revertida. Fácil backup e restauração tornam trivial clonar um banco de dados.”
Poderoso	“PostgreSQL suporta muitos tipos de dados sofisticados, incluindo JSON, XML, objetos geométricos, hierarquias, tags e matrizes.”
Confiável	“O layout de pasta padrão torna mais fácil de controlar onde os dados são armazenados para que você possa fazer o uso máximo do seu particionamento. Ele usa as facilidades de inicialização do sistema operacional em todas as plataformas.”
Rápido	“PostgreSQL faz uso estratégico de indexação e consulta de otimização para trabalhar com o menor esforço possível.”

Fonte: Elaborado a partir de Carvalho (2017)

Além disso, o PostgreSQL possui várias funcionalidades interessantes como:

- O controle de concorrência multiversionado (MVCC, em inglês);
- Recuperação em um ponto no tempo (PITR, em inglês), tablespaces;
- É altamente escalável, tanto na quantidade enorme de dados que pode gerenciar quanto o número de usuários concorrentes que pode acomodar.

2.3 GIT

Segundo Aquiles e Ferreira (2014, p.1), “Ao mexermos em um código existente é importante tomarmos cuidado para não quebrar o que já funciona.”. Tendo isso em vista o que diz Ferreira (2014, p.1), “Por isso, queremos mexer o mínimo possível no código. Temos medo de remover código obsoleto, não utilizado ou até mesmo comentado, mesmo que mantê-lo já nem faça sentido. Não é incomum no mercado vermos código funcional acompanhado de centenas de linhas de código comentado. Sem dúvida, é interessante manter o histórico do código dos projetos, para entendermos como chegamos até ali. Mas manter esse histórico junto ao código atual, com o decorrer do tempo, deixa nossos projetos confusos, Trabalhando em equipe Casa do Código poluídos com trechos e comentários que poderiam ser excluídos sem afetar o funcionamento do sistema.”.

Alexandre Aquiles e Ferreira (2014, p.3), “O Git é um sistema de controle de versão que, pela sua estrutura interna, é uma máquina do tempo extremamente rápida e é um robô de integração bem competente.”. E segundo Aquiles e Ferreira, 2014, p.3, “Foi criado em 2005 por Linus Torvalds, o mesmo criador do Linux, que estava descontente como BitKeeper, o sistema de controle de versão utilizado no desenvolvimento do kernel do Linux.”. Alguns comandos básicos de GIT são descritas no Quadro 3.

Quadro 3 – Comandos Básico GIT

git init	Isso cria um subdiretório chamado. git que contém todos os arquivos necessários de seu repositório — um esqueleto de repositório
git clone	Você clona um repositório com git clone [url].
git add	Monitora Novos Arquivos
git status	A principal ferramenta utilizada para determinar quais arquivos estão em quais estados é o comando.
git diff	Você quer saber exatamente o que você alterou, não apenas quais arquivos foram alterados.
git commit	Armazena o conteúdo atual do índice em um novo commit, juntamente com uma mensagem de registro do usuário que descreve as mudanças.
git fetch	Esse comando vai até o projeto remoto e pega todos os dados que você ainda não tem.
git pull	Incorpora as alterações de um repositório remoto no branch atual. Em seu modo padrão, git pull é uma abreviação para git fetch seguido de git merge FETCH_HEAD.
git push	O git push é o comando em que você transfere commits a partir do seu repositório local para um repositório remoto.

Fonte: Elaborado a partir de Gomes (2016)

2.4 HTML

Como diz Duckett (2016, p.20), “O código HTML (em azul) é composto de personagens que vivem dentro de colchetes angulares - estes são chamados HTML elementos. Os elementos são normalmente feitos de duas Tag”.

Segundo Duckett (2016, p.21), “Html utiliza elementos para descrever a estrutura de páginas”. Cada tag HTML tem seus os atributos, visto a afirmação de Jon Duckett (2016, p26) “Atributos fornecem informações adicionais sobre o conteúdo de um elemento. Eles aparecem na tag de abertura do elemento e são compostos de duas partes: a nome e um valor, separados por um sinal de igual.”.

Em resumo a estrutura de um arquivo HTML é:

- Páginas HTML são documentos de texto.
- HTML usa tags (personagens que ficam dentro anguladas parênteses) para dar a informação que cercam significado especial.
- Etiquetas geralmente vêm em pares. Os denota tag abertura o início de uma parte do conteúdo; o tag de fechamento indica o final.
- Abertura tags podem transportar atributos, que nos dizem mais sobre o conteúdo desse elemento. Atributos
- Para saber HTML que você precisa saber o que tags são disponíveis para você usar, o que eles fazem, e onde podem ir.

2.5 CSS

Segundo Gonçalves (2022, n.p), “CSS é a sigla para o termo em inglês Cascading Style Sheets que, traduzido para o português, significa Folha de Estilo em Cascatas. O CSS é fácil de aprender e entender e é facilmente utilizado com as linguagens de marcação HTML”.

> CSS é chamado de linguagem Cascading Style Sheet e é usado para estilizar elementos escritos em uma linguagem de marcação como HTML. O CSS separa o conteúdo da representação visual do site. Pense na decoração da sua página. Utilizando o CSS é possível alterar a cor do texto e do fundo, fonte e espaçamento entre parágrafos. Também pode criar tabelas, usar variações de layouts, ajustar imagens para suas respectivas telas e assim por diante. (GONÇALVES, 2022, n.p.)

Como diz Gonçalves (2022, n.p), “CSS foi desenvolvido pelo W3C (World Wide Web Consortium) em 1996, por uma razão bem simples. O HTML não foi projetado para ter tags que ajudariam a formatar a página. Você deveria apenas escrever a marcação para o site.”.

Segundo Gonçalves (2022, n.p), “A estrutura da sintaxe CSS é bem simples. Tem um seletor e um bloco de declaração. Você seleciona um elemento e depois declara o que deseja fazer com ele.”. Ainda segundo Zemel (2015, p.12), “Seletores CSS não são novidades para você. Os seletores mais básicos são de tipo (exemplo: div), ID (exemplo: #header) e classe (exemplo: .tweet), respectivamente. Os mais incomuns incluem pseudoclasses comuns (exemplo: :hover) e seletores CSS3 mais complexos, incluindo :first-child ou [class^="grid-"]. Seletores têm uma eficiência inerente.”.

Exemplo de seletor CSS na Figura 2:

Figura 2 - Seletor CSS

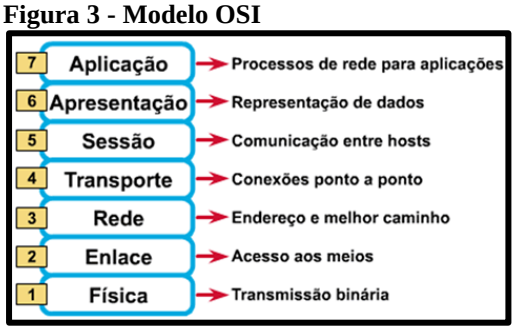
Fonte: https://www.devmedia.com.br/css-seletores/40729

2.6 HTTP

Segundo Saudate (2013, p.13), “Trata-se de um protocolo de camada de aplicação (segundo o modelo OSI) e, portanto, de relativa facilidade de manipulação em aplicações.”. Pode-se ver as camadas do modelo OSI na Figura 3.

> O protocolo HTTP (HyperText Transfer Protocol – Protocolo de Transferência de Hipertexto) data de 1996, época em que os trabalhos conjuntos de Tim Berners-Lee, Roy Fielding e Henrik Frystyk Nielsen levaram à publicação de uma RFC (Request for Comments) descrevendo este protocolo. (SAUDATE, 2013, p.12)

Figura 3 - Modelo OSI



Fonte: https://www.dltec.com.br/blog/cisco/entendendo-o-modelo-osi-para-melhorar-sua-capacidade-de-resolver-problemas-em-uma-rede-cisco/

Saudate (2013, p.15) diz que “A versão corrente do HTTP, 1.1, define oficialmente oito métodos - embora o protocolo seja extensível em relação a estes métodos. Hoje, [...]”. Os métodos HTTP são apresentados no Quadro 4.

Quadro 4 – Métodos HTTP

GET	Recupera os dados identificados pela URL
POST	Cria um novo recurso
PUT	Atualiza um recurso
DELETE	Apaga um recurso
HEAD	Para obter apenas o código de status da validação

Fonte: Elaborado a partir de Saudate (2013)

Saudate (2013, p.23) “Toda requisição que é enviada para o servidor retorna um código de status. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, [...]”. Sendo apresentados no Quadro 5.

Quadro 5 – Categorias de códigos HTTP

1xx	Informacionais
2xx	Códigos de sucesso
3xx	Códigos de redirecionamento
4xx	Erros causados pelo cliente

5xx	Erros originados no servidor

Fonte: Elaborado a partir de Saudate (2013)

Os códigos mais utilizados são vistos no Quadro 6:

Quadro 6 – Códigos HTTP mais usados

200	Indica que a operação indicada teve sucesso.
202	Indica que a solicitação foi recebida e será processada em outro momento. É tipicamente utilizada em requisições assíncronas, que não serão processadas em tempo real.
400	É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.
401	Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação.
403	Utilizado quando o cliente está tentando realizar uma operação sem ter a devida autorização.
404	Utilizado quando o recurso solicitado não existe.
500	É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica. Normalmente erro do lado do serviço.

Fonte: Elaborado a partir de Saudate (2013)

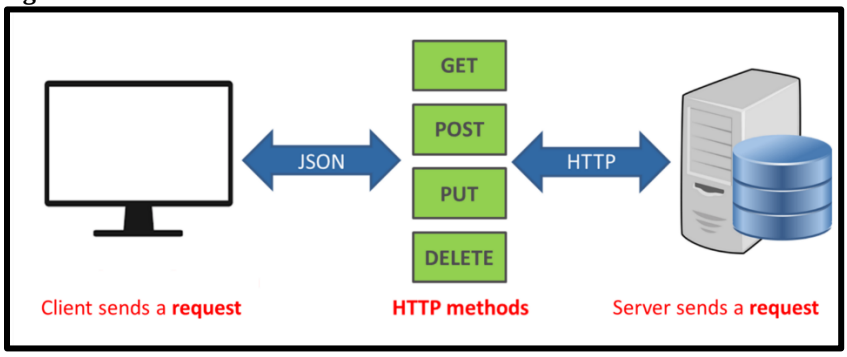
2.7 REST

Saudate (2013, p.29) diz que "REST é baseado nos conceitos do protocolo HTTP. Além disso, este protocolo é a base para a web como a conhecemos, sendo que a própria navegação nesta pode ser encarada como um uso de REST". O REST baseia-se no seguinte:

> Todo serviço REST é baseado nos chamados recursos, que são entidades bem definidas em sistemas, que possuem identificadores e endereços (URL's) próprios. No caso da aplicação que estamos desenvolvendo, brejaonline.com.br, podemos assumir que uma cerveja é um recurso. Assim, as regras de REST dizem que as cervejas devem ter uma URL própria e que esta URL deve ser significativa. Desta forma, uma boa URL para cervejas pode ser /cervejas. (SAUDATE, 2013, p.30)

A Figura 4 mostra a arquitetura do REST.

Figura 4 - REST



Fonte: https://medium.com/totvsdevelopers/rest-no-protheus-b%C3%A1sico-do-b%C3%A1sico-a4b4431a4e3e

Saudete (2013, p.57) diz “JSON é uma sigla para JavaScript Object Notation. É uma linguagem de marcação criada por Douglas Crockford e descrito na RFC 4627, e serve como uma contrapartida a XML.”

Costumeiramente o REST usa JSON, pois segundo Saudete (2013, p.57), “Tem por principal motivação o tamanho reduzido em relação a XML, e acaba tendo uso mais propício em cenários onde largura de banda (ou seja, quantidade de dados que pode ser transmitida em um determinado intervalo de tempo) é um recurso crítico.”. A Figura 5 mostra um código JSON.

Figura 5 - Código JSON

Fonte: https://www.alura.com.br/artigos/o-que-e-json

2.8 REACT

Segundo Stefanov (2019, p.15), “React é uma biblioteca para construção de UIs – ela ajuda você a definir a UI de uma vez por todas. Então, quando o estado da aplicação mudar, a UI será reconstruída de modo a reagir à alteração, e você não precisará fazer nada adicional.”

Segundo Pontes (2018, p.37), “renderização dos componentes React é realizada por meio de um recurso conhecido como Virtual DOM, de performance muito superior à manipulação do DOM convencional dos navegadores.”

> Mas por que exatamente é ruim usar o DOM nativo? Porque manipular o DOM não é performático. Tratando-se de uma PWA cuja execução é totalmente influenciada pela velocidade com que o framework é executado no navegador, considerar o tempo de renderização é muito importante. (PONTES, 2018, p.38)

Segundo Pontes (2018, p.42), “O React renderiza seus componentes através do método render() , que fica dentro da classe do componente.”. Como disse Pontes (2018, p.16), “Um componente possui vários métodos para controle do ciclo de vida. O único de uso obrigatório é o render() . Toda vez que render() for executado, ele desencadeará uma atualização do DOM. Isso acontece quando o componente é criado, ou quando seu estado (this.state) é atualizado. Por estado, entenda um conjunto de dados inerentes ao funcionamento do componente.”

Ainda segundo Pontes (2018, p.46), “O fluxo de dados (propriedades e estados) do React é unidirecional. Um componente pai passa dados para seus componentes filhos, que, por sua vez, passa dados aos componentes netos, e assim por diante.”

2.9 ORM

Segundo FONSECA (2020, n.p), “Object-Relational Mapping (ORM), em português, mapeamento objeto-relacional, é uma técnica para aproximar o paradigma de desenvolvimento de aplicações orientadas a objetos ao paradigma do banco de dados relacional.”

> As bibliotecas ou frameworks ORM definem o modo como os dados serão mapeados entre os ambientes, como serão acessados e gravados. Isso diminui o tempo de desenvolvimento, uma vez que não é necessário desenvolver toda essa parte. Outra vantagem está na adaptação de novos membros na equipe, como muitos projetos comerciais utilizam a mesma ferramenta, é possível encontrar membros que já estão acostumados com o padrão de trabalho. (Fonseca, 2020, n.p)

Segundo Fonseca (2020, n.p), “Independente da linguagem de programação que o ORM é implementado, geralmente ele segue um padrão bem definido.”

2.10 PRISMA

Segundo Buzzi (2022, n.p), “´[...] Prisma nasceu no ecossistema JavaScript com a promessa de ser uma ferramenta facilitadora e produtiva para devs que trabalham diretamente com databases.”

> Na descrição oficial da ferramenta, Prisma é descrito como um novo tipo de ORM, fundamentalmente diferente dos modelos tradicionais que eram aplicados anteriormente. Uma alternativa para outras ORMs, como TypeORM e Sequelize. Todo o conceito do Prisma está no schema proporcionado por eles no desenvolvimento nestas três camadas citadas anteriormente. Segundo eles, o schema

é descrito como o principal arquivo de configuração para o Prisma. (BUZZI, 2022, n.p)

Segue um exemplo do Schema do Prisma na Figura 6.

Figura 6 - Schema do Prisma

```
datasource db {
  url      = env("DATABASE_URL")
  provider = "postgresql"
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  email     String   @unique
  name      String?
  role      Role     @default(USER)
  posts     Post[]
}

model Post {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  published Boolean  @default(false)
  title     String   @db.VarChar(255)
  author    User?    @relation(fields: [authorId], references: [id])
  authorId  Int?
}

enum Role {
  USER
  ADMIN
}
```

Fonte: https://blog.rocketseat.com.br/prisma-uma-das-melhores-coisa-que-ja-aconteceu-no-ecossistema/

3. METODOLOGIA DA PESQUISA

O presente trabalho é uma pesquisa descritiva e aplicada com o intuito de desenvolver uma ferramenta de auxílio de trabalho.

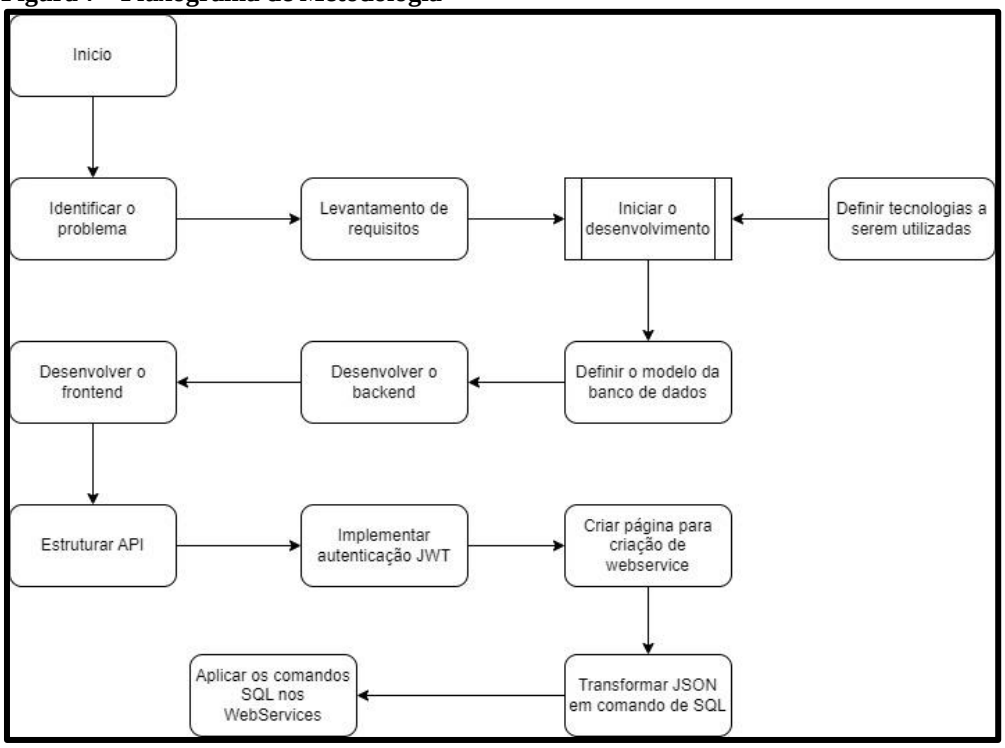
A primeira etapa consiste na fundamentação teórica, fazendo assim um levantamento básico dos conceitos utilizados para o desenvolvimento do trabalho.

A segunda etapa é finalizar o protótipo com funcionalidades básicas, para definir testes e aplicações praticar para verificar se o produto é realmente eficaz.

A última etapa trata do levantamento dos dados, a fim de analisar o impacto da feramente no dia a dia de um desenvolvedor backend. Gerando assim estatísticas para visualizar como seria o impacto.

Os passos do desenvolvimento podem ser vistos na Figura 7:

Figura 7 - Fluxograma de Metodologia



Fonte: acervo do autor (2022)

Por fim, iremos averiguar o trabalho que um desenvolvedor leva para criar um webservice REST, e como uma feramente low-code pode auxiliá-lo no processo de criação e disponibilização do endpoint do serviço.

O método de avaliação do resultado dessa pesquisa será uma demonstração de que o protótipo esteja funcionando da maneira espera. Assim garantindo que a pretensão inicial de criar um webservice de consulta funciona da maneira desejada.

3.1 ESTADO DA ARTE

Esta seção apresentará duas ferramentas disponíveis no mercado, que tem as suas funcionalidades semelhantes com o propósito geral da ferramenta a ser desenvolvida.

3.1.1 Strapi

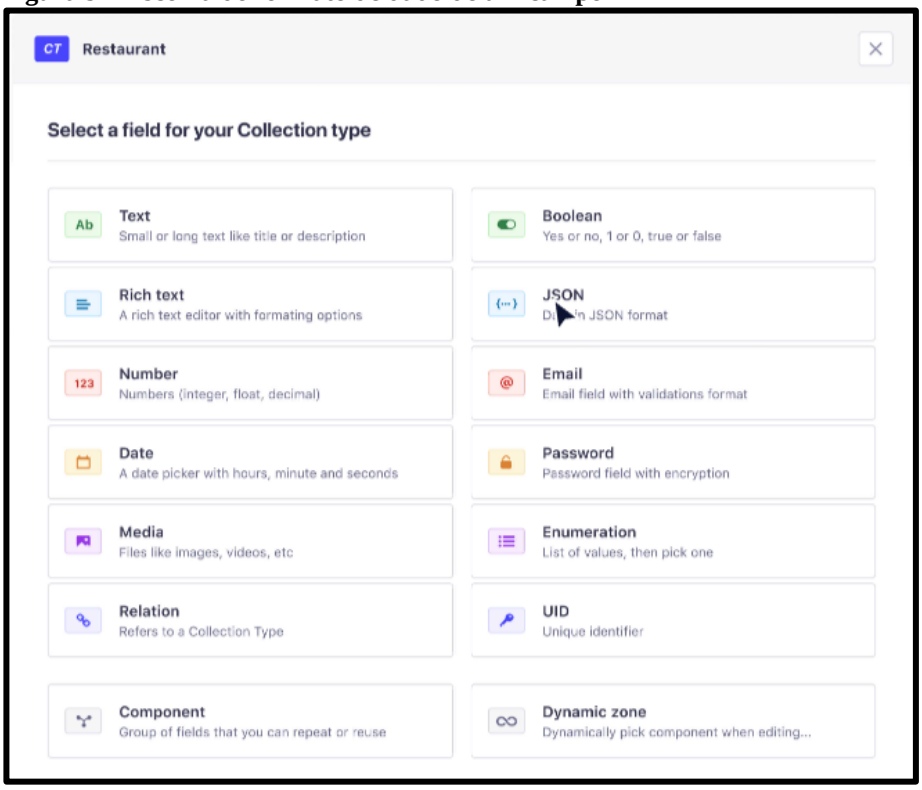
Strapi é um serviço online de **CMS** que permite a criação rápida de API's em node de alta velocidade para cadastros e consultas com REST. A Figura 8 mostra a tela de seleção de formato de dados do Strapi.

Figura 8 – Exemplo de formato de dado com Strapi

Fonte: Strapi (2022)

A aplicação fornece a possibilidade de informar o formato de um dado específico, e já mostra como esse dado seria retornado em um reponse de uma API REST disponibilizada pelo serviço. A Figura 9 mostra e escolha do formato de dado de um campo específico.

Figura 9 – Escolha do formato de dado de um campo



Fonte: https://strapi.io/

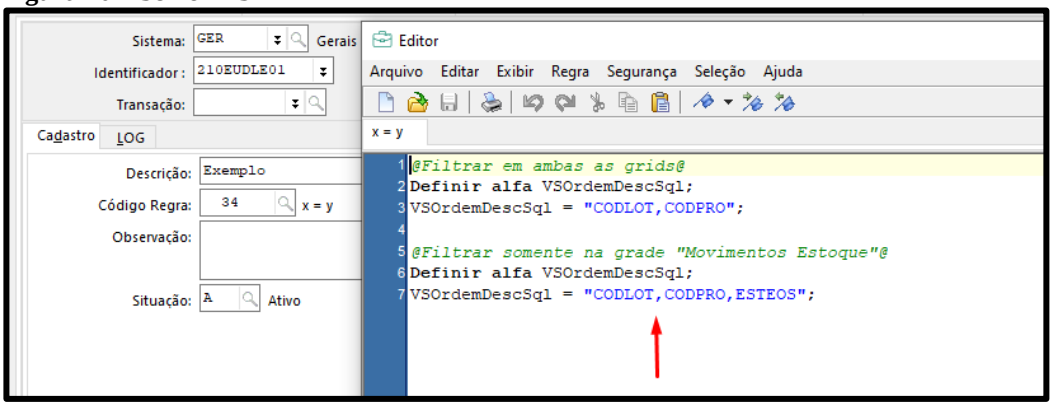
Para usar o Strapi, basta instalar o NodeJS na sua máquina e instalar a **CLI** do Strpi via **npm** para começar a utilizar o serviço na sua versão gratuita.

3.1.2 Senior LSP

A Senior é uma das maiores empresas de ERP do Brasil, situada em Blumenal, oferece soluções de software de gestão empresarial.

O software da Senior possui um editor de WebServices que é programado uma linguagem própria chamada LSP, que é uma linguem própria semelhante ao Prolog, em que é possível criar regras lógicas para o comportamento dos webservices do sistema, a interface é mais complexa conforme visto da Figura 10.

Figura 10 – Senior LSP



Fonte: https://suporte.senior.com.br/hc/pt-br/articles/4408642170772-ERP-Distribui%C3%A7%C3%A3o-de-Lotes-Como-ordenar-a-distribui%C3%A7%C3%A3o-de-lotes-de-forma-personalizada

No entanto, essa linguagem tem uma curva de aprendizado maior, já que existem várias regras e padronizações que tem que ser obedecidas, desde declaração de variáveis até a forma como se escreve a regra SQL do cursor.

3.1.3 Protótipo versus estado da arte

O proposito do protótipo é ser uma plataforma mais fácil de ser utilizada, no Quadro 7, pode-se ver algumas comparações entre o projeto e o estado da arte.

Quadro 7 – Comparação entre protótipo e estado da arte

Protótipo	**Strapi**	**Senior LSP**
Possível escolher os campos com um clique.	Possível escolher os campos com um clique.	É necessário escrever a regra manualmente.
Curva de aprendizado menor.	Curva de aprendizado menor.	Curva de aprendizado maior.
Foca na facilidade de realizar consultas ao banco de dados	Criar toda a estrutura de requisição de um site	Têm estruturas fixas e é mais difícil de personalizar.

Fonte: Acervo do autor

A partir dessa comparação, é possível averiguar que mesmo que o protótipo desenvolvido nesse trabalho não tenha os mesmos recursos oferecidos pelos concorrentes, ele tem o diferencial de desde o início ser focado apenas na facilidade de operação e funcionamento, bastando poucos passos para colocá-lo em produção.

4 PROTÓTIPO DE GERADOR DE WEBSERVICES POR ITERFACE GRÁFICA

O protótipo foi desenvolvido com a intenção de facilitar o trabalho de desenvolvimento de API's RESTFULL com JSON para consultas de dados de uma base para consulta, para possibilitar a criação desses webservices com poucos cliques, poupando assim tempo de desenvolvimento que poderá ser destinado ao outras atividades mais focadas no negócio.

4.1 ANÁLISE DO PROJETO

Essa seção é realizada um levantamento a respeito das funcionalidades básicas que o protótipo final devera suprir, levando em consideração as demandas a serem requisitadas.

4.2 VISÃO GERAL

Esse protótipo de software tem a pretensão de facilitar o dia a dia de um desenvolvedor de software, com a vantagem de diminuir os passos necessários que o usuário/desenvolvedor levaria para criar um webservice de consulta em REST.

O objetivo geral desse protótipo é diminuir o tempo em que um desenvolvedor gasta no seu expediente ou em projetos pessoais para criar webservices para consultar dados de uma base qualquer.

E a principal vantagem desse sistema é que com o tempo salvo que o desenvolvedor iria gastar desnecessariamente com o desenvolvimento de forma individual de cada consulta do seu projeto, ele possa basicamente escolher uma tabela e as suas respectivas colunas para uma consulta simples.

4.3 LEVANTAMENTO DE REQUISITOS

Nessa seção serão levantados os requisitos essenciais para o funcionamento básico do sistema, as regras de negócio e os casos de uso da aplicação que serão usadas pelo usuário final.

4.3.1 Requisitos Funcionais

No decorrer do projeto foram levantados os requisitos funcionais para o funcionamento do sistema. Os requisitos levantados são essenciais para o funcionamento adequado do sistema. O Quadro 8, representa os requisitos funcionais do sistema.

Quadro 8 – Requisitos Funcionais

Número	Nome	Descrição
RF-01	Login de Sistema	O sistema deverá permitir o login do usuário para o acesso, será necessário usuário e senha.
RF-02	Montagem de dicionário de dados	O sistema deverá realizar a montagem do dicionário de dados.
RF-03	CURD de menus	O sistema deverá ter as operações de CRUD para a tabela de menu.
RF-04	CRUD de status	O sistema deverá ter as operações de CRUD para a tabela de status.
RF-06	CRUD de webservices	O sistema deverá ter as operações de CRUD para a tabela de webservices.
RF-07	CRUD de webservicesobj	O sistema deverá ter as operações de CRUD para a tabela de webservicesobj.
RF-08	CRUD de webservicesroutes	O sistema deverá ter as operações de CRUD para a tabela de webservicesroutes.
RF-09	Consultas automáticas	O sistema deverá montar as consultas automaticamente com base em um array de campos
RF-10	Formulários automáticos	O sistema deverá montar os formulários automaticamente com base em um array de campos

Fonte: Acervo do autor (2022)

4.3.2 Requisitos Não-Funcionais

No decorrer do projeto foram levantados os requisitos não-funcionais para o funcionamento do sistema. Os requisitos levantados são essenciais para o funcionamento adequado do sistema. O Quadro 9, apresenta os requisitos não-funcionais do sistema.

Quadro 9 – Requisitos não-funcionais

Número	Descrição	Classificação	Fonte
RFN-01	Interface deve ser intuitiva e prática.	Usabilidade	Usuário
RFN-02	O sistema de estar otimizado.	Performance	Sistema
RFN-03	O sistema deve exigir autenticação para acesso dos dados.	Segurança	Sistema

Fonte: Acervo do autor (2022)

4.3.3 Regras de Negócio

No decorrer do projeto foram levantadas as regras de negócio para o funcionamento do sistema. Os requisitos levantados são essenciais para o funcionamento adequado do sistema. O Quadro 10, apresenta as regras de negócio do sistema.

Quadro 10 – Regras de Negócio

Número	Descrição
RN-01	O usuário poderá realizar os cadastros de status extras
RN-02	O usuário poderá realizar o cadastro de webservices
RN-03	O usuário poderá cadastrar quantos objetos de webservice por webservice
RN-04	O usuário poderá selecionar quais tabelas e colunas um objeto de webservice irá consultar.
RN-05	O usuário poderá cadastrar as um código para a execução do webservice.

Fonte: Acervo do autor (2022)

4.4 TELAS DE CONSULTA E FORMULÁRIO DO PROTÓTIPO

Como foi utilizado o framework React para o desenvolvimento das telas do protótipo, e como ele permite a reutilização de componentes de funcionalidades e interface, todas as telas da consulta da aplicação foram padronizadas.

Isso torna todas as consultas com a mesma aparência, sem nenhum diferencial com a exceção de telas que tenham funcionalidades mais especificas. Assim foi mantida a identidade visual em todas as páginas do sistema.

Além das telas de consulta padronizadas, os formulários também são montados automaticamente de acordo com um array de inpus, assim poupando o tempo que levaria para montar uma tela de formulário caso fosse adicionado os inputs individualmente.

4.4.1 Tela de Login de Usuário

Assim que o usuário entra no sistema, a primeira tela é a de login para conseguir se autenticar no sistema. A Figura 11 exibe o formato da tela de login.

Figura 11 – Tela de Login

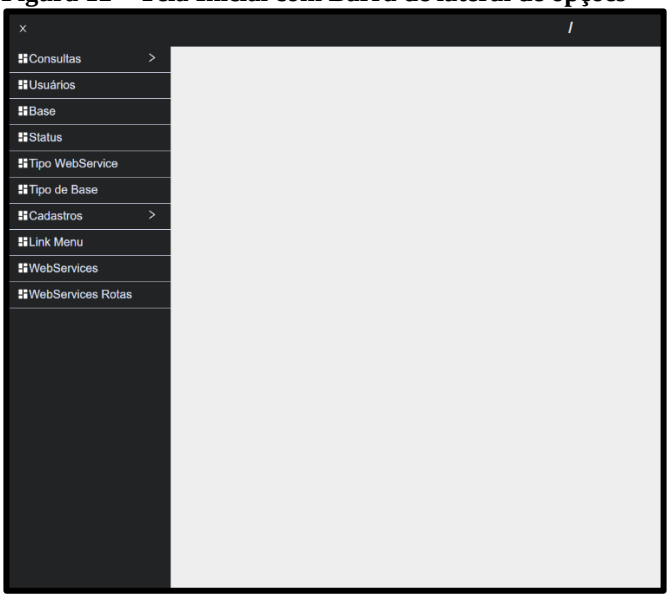
Fonte: Acervo do autor (2022)

Após o login de usuário ser efetuado com sucesso, o sistema redireciona para a página de seleção de menus e submenus.

4.4.2 Tela Inicial

Durante o desenvolvimento do projeto, foi optado pela tela inicial mostrar por meio de uma barra lateral os menus padrões da aplicação, separados entre menus da consulta e de cadastro. A Figura 12 demonstra a tela inicial do protótipo com a barra lateral de opções.

Figura 12 – Tela Inicial com Barra de lateral de opções



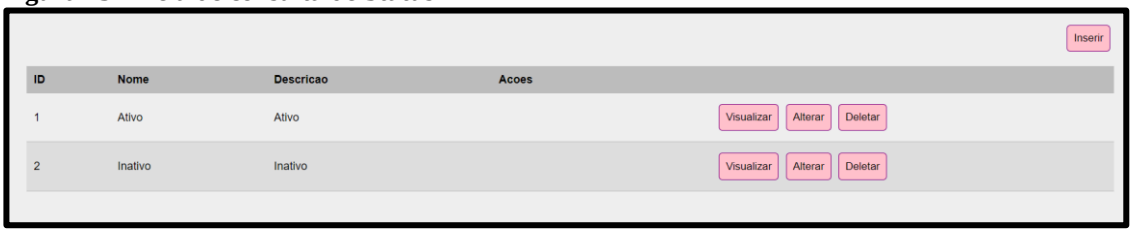
Fonte: Acervo do autor (2022)

É possível notar que as opções na barra lateral é recolhivel, elas foram feitas de forma recursiva, portanto podem ter quantos submenus necessários, mas para esse protótipo, dois níveis são o suficiente.

4.4.3 Tela de Status

A seguir, a tela para consulta de status da aplicação. Como exemplificado na Figura 13 de consulta de Status.

Figura 13 – Tela de consulta de Status



Fonte: Acervo do autor (2022)

Nesta tela é apresentado os status padrões para outros registros da aplicação. É possível adicionar status personalizados para os registros do protótipo.

Além da consulta, ao clicar no botão inserir, a página é redirecionada para o formulário automático de status conforme mostrado na Figura 14.

Figura 14 – Formulário de Status

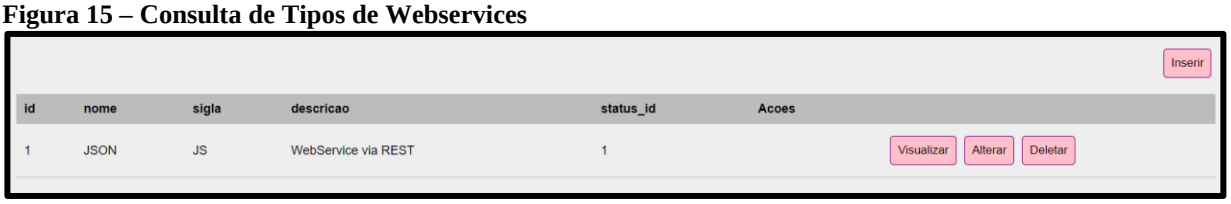
Fonte: Acervo do autor (2022)

O formulário de criação de status conte os campos de id que é gerado automaticamente no backend, nome e descrição.

4.4.4 Tela de Tipo de WebService

Essa tela apresenta o tipo de WebService que o protótipo é atualmente capaz de criar. Podendo futuramente adicionar novas formas. Conforme visto na Figura 15 de consulta de tipo de webservices.

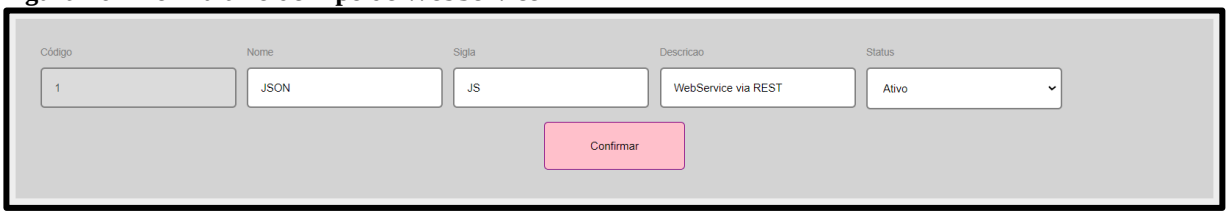
Figura 15 – Consulta de Tipos de Webservices



Fonte: Acervo do autor (2022)

Esta tela permite uma consulta rápida dos tipos de webservices que o sistema pode prover. Ainda temos a tela do formulário do tipo de webservice demonstrado na Figura 16.

Figura 16 – Formulário de Tipo de WebService



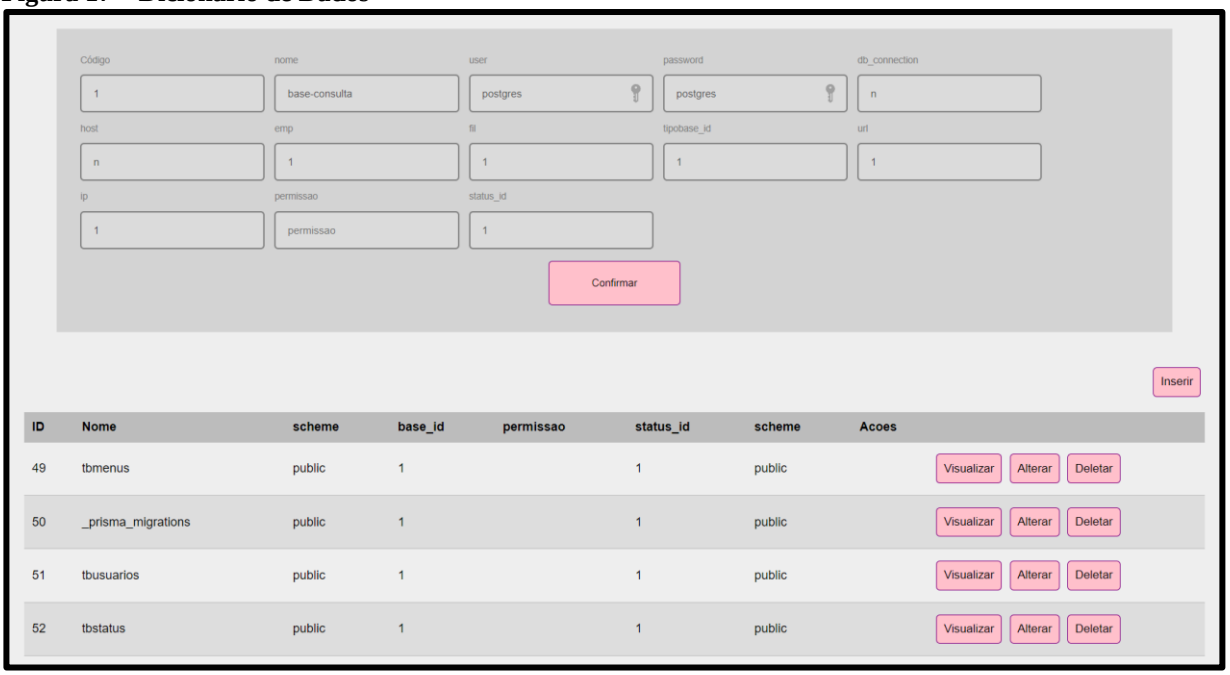
Fonte: Acervo do autor (2022)

O formulário de tipo de webservice conta com os campos de id, nome, sigla, descrição e status.

4.4.5 Dicionário de Dados

A tela do dicionário de dados é uma das mais importantes da aplicação, pois é com ela que é possível identificar todas as entidades da aplicação. Como demonstrado na Figura 17.

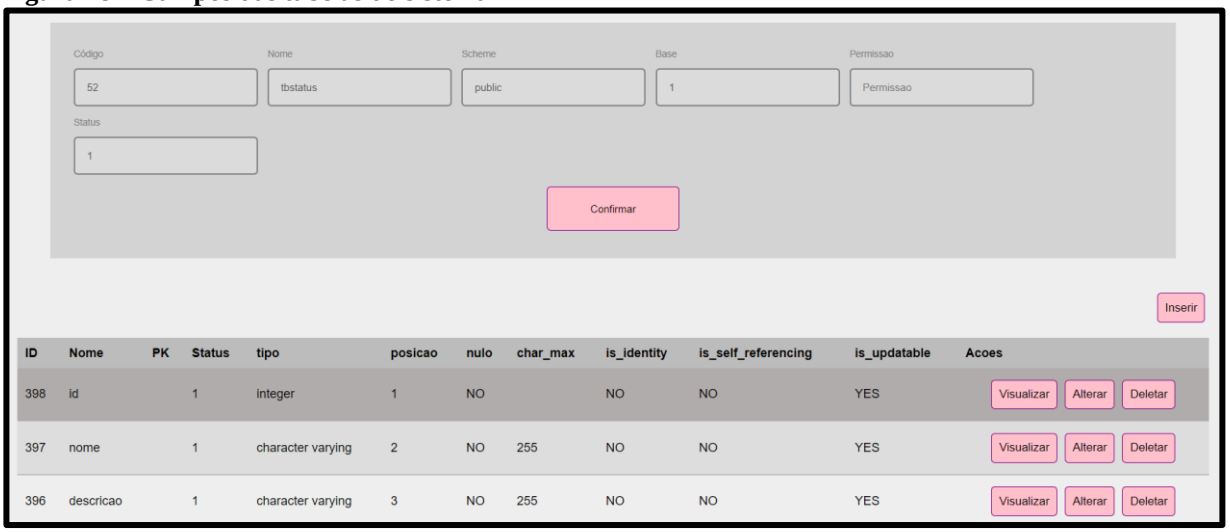
Figura 17 – Dicionário de Dados



Fonte: Acervo do autor (2022)

Essa tela possui as informações da listagem das entidades do banco do sistema com consultas individuais para cada entidade para poder visualizar quais campos e seus formatos de dados cada entidade possui, como pode ser visto na Figura 18.

Figura 18 – Campos das tabelas do sistema



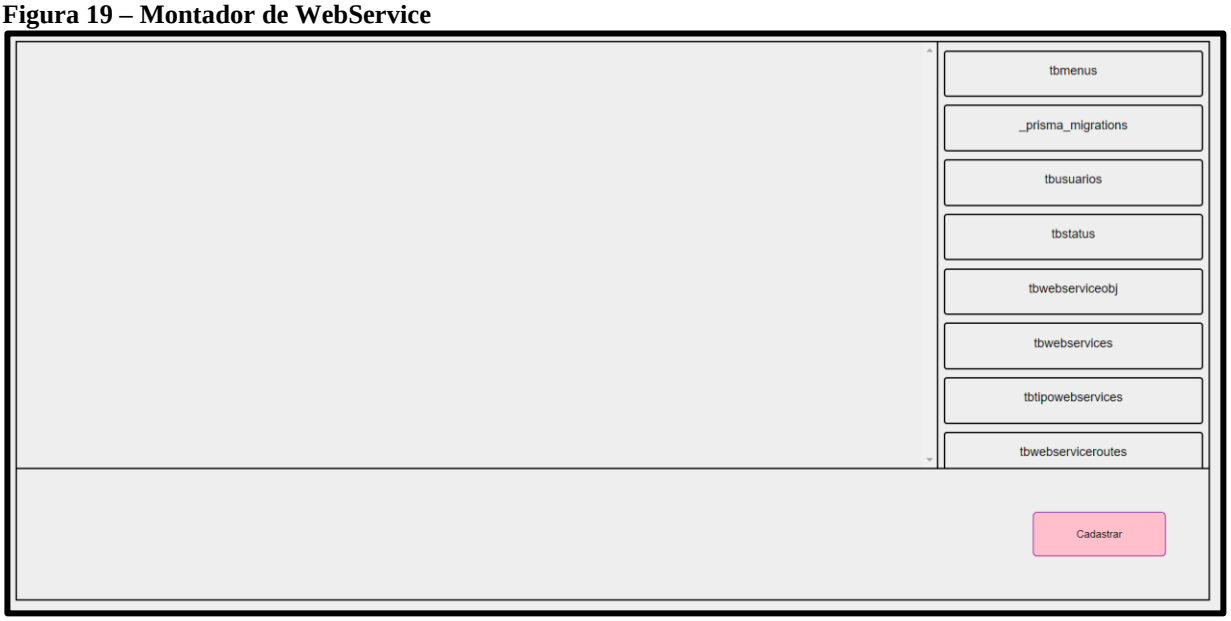
Fonte: Acervo do autor (2022)

Assim os todos os campos das entidades podem ser visualizados para serem utilizados posteriormente no montador de webservices.

4.4.6 Montador de WebServices

A tela de montagem de webservices é o ponto central do protótipo, pois nesse ambiente é possível escolher exatamente quais campos de uma base de dados podem ser escolhidos para usar em um webservice REST. A Figura 19 mostra o funcionamento da tela.

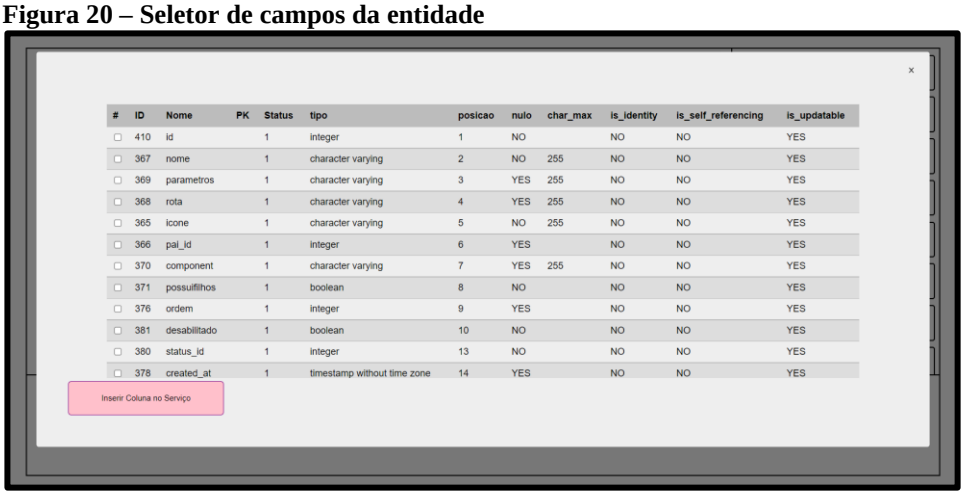
Figura 19 – Montador de WebService



Fonte: Acervo do autor (2022)

Essa tela permite ao usuário escolher qual entidade do banco de dados será usado no webservice criado. Para escolher quais campos serão desejados no serviço, deve-se clicar em qualquer uma das entidades que irá abrir um model com os campos específicos da entidade selecionada. Como visto na Figura 20.

Figura 20 – Seletor de campos da entidade



Fonte: Acervo do autor (2022)

Esse modal exibe uma tabela com caixas selecionáveis para quais campos serão mandados para o webservice principal. Figura 21, mostra como será tratado os campos selecionados.

Figura 21 – Montador com campos selecionados

Fonte: Acervo do autor (2022)

Nessa etapa do processo, o componente trava a seleção de outras entidades, a menos que se exclua as que já foram selecionadas, para aí sim selecionar outra entidade. A Figura 22 mostra o resultado desta operação.

Figura 22 – Objetos do Webservice

Fonte: Acervo do auto (2022)

Os objetos do webservice, são cada possibilidade de consulta que foi criada na tela de montagem.

4.5 DESENVOLVIMENTO

O desenvolvimento do protótipo foi feito com React no frontend, que auxilia a criar componentes reutilizáveis e funcionalidades extras na interface, facilitando o trabalho que teria com HTML, Javascript e CSS caso não utilizasse nenhuma biblioteca.

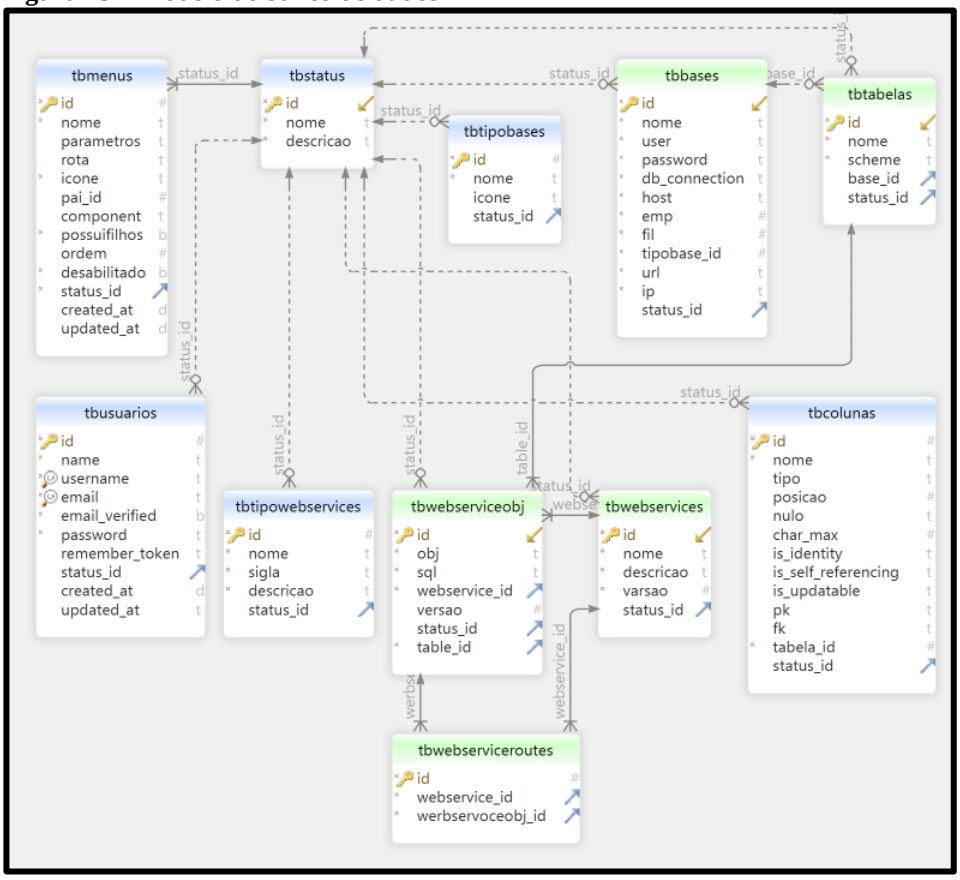
O desenvolvimento do backend foi feito com NodeJs, que permite a utilização de Javascript como linguagem principal, além disso, foi utilizado o Prisma ORM para centralizar e gerenciar o acesso ao banco de dados, cujo foi utilizado o PostgreSQL.

4.5.1 Banco de Dados

Para o banco de dados, foi utilizado o PostgreSQL, que é um dos bancos mais utilizados no mundo. A estrutura da base de dados foi desenvolvida pensando em como as principais funcionalidades de webservices iriam se comportar no protótipo.

O modelo inicial da estrutura da base de dados tinha 21 tabelas, esse número foi expressivamente reduzido para 12 tabelas, tornando-a mais enxuta e simples. Conforma a Figura 23.

Figura 23 – Modelo de banco de dados

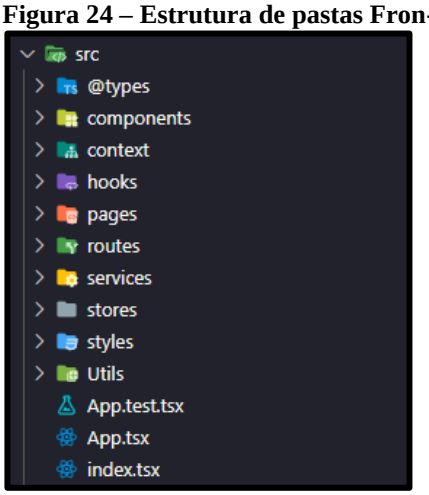


Fonte: Acervo do auto (2022)

4.6 DESENVOLVIMENTO FRONT-END

Para o desenvolvimento do front-end, foi adotado uma estrutura de pastas para melhor organizar o código do software de acordo com o seu contexto. A Figura 24 mostra a estrutura de pastas do front.

Figura 24 – Estrutura de pastas Fron-End



Fonte: Acervo do autor (2022)

O significado de capas pasta nessa estrutura está descrito no Quadro 11.

Quadro 11 – Estrutura de pastas Front-End

Pasta	Objetivo
@types	Contém arquivos que refletem o formato de dados das tabelas do sistema.
componentes	Contém componentes visuais utilizados em toda parte visual.
context	Contém arquivos de contexto do React, que servem para o compartilhamento de informações entre componentes.
hooks	Contém hooks, que no React são funções com retornos que podem ser valores, objetos, arrays ou funções para serem utilizados nos componentes.
pages	Contém os arquivos das páginas do sistema.
routes	Contém os arquivos de rotas do sistema.
services	Contém os arquivos para realizar consultas ao back-end.
stores	Contém arquivos que utilizam o pacote do Zustand para não precisar usar as contexs em certas ocasiões.
styles	Contém arquivos de estilização global.
utils	Contém vários arquivos de interface para organizar que tipo da dados alguns componentes devem receber.

Fonte: Acervo do autor (2022)

Com essa estrutura de pastas, fica fácil organizar o código do software, e não se perder no momento do desenvolvimento.

4.6.1 UseFetch

Durante o desenvolvimento, notou-se uma necessidade de padronizar as chamadas de API do back-end, pois havia muito código repetido com o mesmo propósito, então com a utilização da biblioteca Axios para chamadas de API REST, foi desenvolvido um componente único para esse propósito, como pode ser visto na Figura 25.

Figura 25 - UseFetch

```
5   export function useFetch<T = unknown>(url: string, options?: AxiosRequestConfig) {
6       const [ data, setData ] = useState<T | null>();
7       const [ statusCode, setStatus ] = useState<number | null>(null);
8       const [ error, setError ] = useState<Error | null>(null);
9       const [ loadding, setLoadding ] = useState(false);
10
11      const fetch = useCallback( async () => {
12          await setError(null);
13          await setLoadding(true);
14          await api(url, options)
15          .then( response => {
16              const { data, status } = response;
17              setData(data);
18              setStatus(status)
19          } )
20          .catch( erro => {
21              setError(erro);
22          } )
23          .finally( () => {
24              setLoadding(false);
25          })
26      }, [url] );
27
28      useEffect(() => {
29          fetch();
30        }, [fetch]);
31
32      return {
33          data,
34          statusCode,
35          error,
36          loadding,
37      };
38  };
```

Fonte: Acervo do autor (2022)

Esse Hook pode ser chamado em qualquer parte da aplicação para realizar consultas de dados, ele retorna de forma automática os dados, o código http da resposta do servidor, se está consultando e se houve algum erro.

4.6.2 Autenticação

O sistema requer autenticação par ao acesso as rotas restritas, todas as funções de autenticação ficam centralizadas em um contexto que compartilha as informações do usuário para todos os componentes. Além disso, pelo fato que as requisições HTTP serem statless, é salvo no localstorage no cliente-side informações do usuário e um token de acesso. A Figura 26 mostra como foi feito o contexto de autenticação.

Figura 26 - AuthContext

```
export const AuthProvider: React.FC<AuthProviderProps> = ({children}) => {
    const [ user, setUser ] = useState<object | null>(null);
    const [ loginLoaging, setLoginLoaging ] = useState<boolean>(false);

    useEffect( () => {
        const storageUser = localStorage.getItem("USER");
        const storageToken = localStorage.getItem("TOKEN");
        if (storageUser && storageToken) {
            api.defaults.headers.common['Authorization'] = `Bearer ${String(storageToken)}`;;
            setUser(JSON.parse(storageUser));
            setLoginLoaging(false);
        } else if (!storageUser && !storageToken) {
            redirect("/")
        }
    }, [] );

    async function signIn(username: string, password: string) {
        const AuthController = await new Auth();
        const response = await AuthController.signIn(username, password);
        await console.log("response", response);
        if ((response.USER != undefined) && (response.TOKEN != undefined && typeof(response.TOKEN) == "string")) {
            api.defaults.headers.common['Authorization'] = `Bearer ${String(response.TOKEN)}`;;
            await localStorage.setItem("USER" , JSON.stringify(response.USER));
            await localStorage.setItem("TOKEN" , String(response.TOKEN));
            return await window.location.replace("/");
        };
        return await window.location.replace("/");
    };

    async function signOut() {
        await localStorage.removeItem("USER");
        await localStorage.removeItem("TOKEN");
        await setUser(null);
        return redirect("/");
    };
```

Fonte: Acervo do usuário (2022)

Esse contexto de autenticação realiza três funções básicas descritas no Quadro 12.

Quadro 12 – Funções de Login

Função	Resultado
useEffect para o local storage	Verifica se o usuario já está logado, e utiliza como padrão o TOKEN de autorização nas requisições.
signIn	Manda as informações de username e password para realizar a autenticação no back-end.
signOut	Desloga o usuário da aplicação.

Fonte: acervo do autor (2022)

A importância desse contexto é que ele consegue lidar com os aspectos de autenticação do usuário de forma localizada e centralizada, assim, em qualquer lugar do sistema é possível aproveitar dessas funções.

Outro exemplo seria se futuramente fosse implantado um sistema de permissões de usuário, usando essa mesma estratégia, em qualquer lugar da aplicação seria possível carregar essas permissões.

4.6.3 Rotas

Os arquivos de rotas do protótipo definem quais componentes serão renderizados a depender da URL da página. A Figura 27 mostra como estão organizadas essas rotas.

Figura 27 – Rotas Front-End

```
const AuthRoutes = () => {      You, 3 weeks ago • sistema de login …
    return (
        <BrowserRouter>
            <MenuProvider children={<>
                    <SideBar />
                    <FormProvider children={<>
                            <WebServiceProvider children={
                                <>
                                    <Routes>
                                        <Route path="/" element={<Dashboard />} />
                                        <Route path="usuario-consulta" element={<UserQuery />} />
                                        <Route path="usuario-manutencao/:op/:id" element={<UsersMaintenance />} />
                                        <Route path="menu-consulta" element={<MenuQuery />} />
                                        <Route path="menu-manutencao/:op/:id" element={<MenuMaintenance />} />
                                        <Route path="menu-manutencao/:op" element={<MenuMaintenance />} />
                                        <Route path="tabelas-consulta" element={<TabelasQuery />} />
                                        <Route path="tabelas-manutencao/:op/:id" element={<TabelasMaintenance />} />
                                        <Route path="status-contulta" element={<StatusQuery />} />
                                        <Route path="status-manutencao/:op/:id" element={<StatusMaintenance />} />
                                        <Route path="status-manutencao/:op" element={<StatusMaintenance />} />
                                        <Route path="tipowebservice-contulta" element={<TipoWebServiceQuery />} />
                                        <Route path="tipowebservice-manutencao/:op/:id" element={<TipoWebServiceMaintenance />} />
                                        <Route path="colunas-manutencao/:op/:id" element={<ColunasMaintenance />} />
                                        <Route path="bases-consulta" element={<BasesQuery />} />
                                        <Route path="bases-manutencao/:op/:id" element={<BasesMaintenance />} />
                                        <Route path="tipo-base-consulta" element={<TipoBaseQuery />} />
                                        <Route path="tipo-base-manutencao/:op/:id" element={<TipoBaseMaintenance />} />
                                        <Route path="tipo-base-manutencao/:op" element={<TipoBaseMaintenance />} />
                                        <Route path="webservices-consulta" element={<WebServicesQuery />} />
                                        <Route path="webservices-manutencao/:op/:id" element={<WebServicesMaintenance />} />
                                        <Route path="webservices-manutencao/:op" element={<WebServicesMaintenance />} />
                                        <Route path="webservicesobj-consulta" element={<WebServicesObjQuery />} />
                                        <Route path="webservicesobj-manutencao/:op/:id" element={<WebServicesObjMaintanance />} />
                                        <Route path="webservicesobj-manutencao/:op" element={<WebServicesObjMaintanance />} />
                                        <Route path="webserviceroutes-consulta" element={<WebServiceRoutesQuery />} />
                                        <Route path="webserviceroutes-manutencao/:op/:id" element={<WebServiceRoutesMaintenance />} />
                                        <Route path="webserviceroutes-manutencao/:op" element={<WebServiceRoutesMaintenance />} />
                                        <Route path={"*"} element={<PageNotFound />}/>
                                    </Routes>
                                </>
                            }/>
                        </>
                    }/>
                </>
            } />
        </BrowserRouter>
    );
```

Fonte: Acervo do autor (2022)

A estratégia de organização das rotas é feita de três formas descritas do Quadro 13.

Quadro 13 – Estratégia de rotas no front.

Forma	Descrição
Rota de consulta	Renderiza uma tabela com todos os dados
Rota de manutenção individual	Renderiza uma página com os dados de um único registro.
Rota de criação	Renderiza uma página com um formulário dinâmico para a criação de novos registros.

Fonte: acervo do autor (2022)

4.6.4 Componente DataGrid

O componente de DataGrid é especialmente importante pois ele é quem realiza todas as consultas para a montagem das tabelas com os dados do back-end. O código desse componente está na Figura 28.

Figura 28 - DataGrid

```
export const DataGrid: React.FC<GridProps> = ({ columns, data, loading, pathMantencao, ...props }) => {
    const teste: any[] = data;
    const navigate = useNavigate();
    const addButtonAction = useButtonStore(state => state.addButton);
    const cleanButtonAction = useButtonStore(state => state.cleanButton);

    const buildMaintenanceURL = useCallback( (btn: string, op: number, id?: any) => {
        let url = btn + "-manutencao" + `/${op}` + `/${id}`;
        return navigate("/"+url);
    }, [navigate]);

    const buttonInserFormValues = useCallback( (op: number) => {
        let url = pathMantencao + "-manutencao" + `/${op}`;
        return navigate("/"+url);
    }, [navigate] );
        You, last month • mudança de botões …
    const handdleSelectButtonActions = useCallback( async (btn: string, operation: number, id: number) => {
        switch(operation) {
            case Operation.ALTER:
                await cleanButtonAction();
                await buildMaintenanceURL(btn, operation, id);
            break;
            case Operation.DELETE:
                await cleanButtonAction();
                await buildMaintenanceURL(btn, operation, id);
            break;
            case Operation.VIEW:
                await cleanButtonAction();
                await buildMaintenanceURL(btn, operation, id);
            break;
            default:
                await addButtonAction({
                    button: btn,
                    action: operation,
                    id: id,
                    title: "",
                });
            break;
        }
    }, [addButtonAction, buildMaintenanceURL]);

```

Fonte: Acervo do autor (2022)

O componente recebe um array de campos e outro array com os dados de consulta, e em seguida monta uma tabela usando comparação por chaves para identificar a posição de cada valor na tabela, além manter uma coluna reservada para botões com funções padrões ou funções que podem ser adicionadas via call-back.

4.6.5 Componente de FormBuilder

O componente FormBuilder foi desenvolvido para automatizar a tarefa de criar formulário automaticamente, passando apenas um array de inputs com os seus nomes e tipos, como number, string ou caixa de seleção.

É o componente mais extenso do protótipo pois ele foi feito de tal forma a centralizar nele, além da construção dos formulários, mas também as chamadas dos dados para o servidor, cobrindo todas as operações de CRUD para as telas do sistema. A Figura 29 mostra o início do componente.

Figura 29 - FormBuilder

```
export const FormBuilder: React.FC<FormProps> = ({  op, data, campos, callBack, urlBakc, ...props }) => {
    const [ formValues, setFormValues ] = useState({});
    const [ backResponse, setBackResponse ] = useState<string>();
    const [ showSnackBar, setShowSnackBar ] = useState<boolean>(false);
    const navigate = useNavigate();
    const { signIn } = useAuth();

    const { setChangeState } = useForm();

    useEffect( () => {
        switch(op) {
            case Operation.INSERT:
                return setFormValues({});
            case Operation.ALTER:
                if (data?.[0]) {
                    return setFormValues({
                        ...data?.[0]
                    });
                };
            break;
            case Operation.DELETE:
                if (data?.[0]) {
                    return setFormValues({
                        ...data?.[0]
                    });
                };
            break;
            case Operation.VIEW:
                if (data?.[0]) {
                    return setFormValues({
                        ...data?.[0]
                    });
                };
            break;
            default:
                return setFormValues({});
            break;
        };
    }, [data, op] );
```

Fonte: Acervo do autor (2022)

O componente guarda os dados do formulário em um único objeto, e quando não houver retorno dos dados do servidor, ele apenas montar os inputs do formulário com valores nulos. A Figura 30 mostra como está organizado o código do submit.

Figura 30 – FormBuilder Submit

```
const submitFormToBakc = async () => {
    switch (op) {
        case Operation.INSERT:
            try {
                await api.post(urlBakc, {
                    data: JSON.stringify(formValues)
                }).then(response => {
                    const { status } = response;
                    navigate(-1)
                }).catch(async error => {
                    await setBackResponse(error.response.data.message.code);
                    await setShowSnackBar(true);
                }).finally(
                );
            } catch (error) {
                console.log(error);
            };
        break;
        case Operation.ALTER:
            try {
                await api.put(urlBakc, {
                    data: JSON.stringify(formValues)
                }).then(response => {
                    const { status } = response;
                    navigate(-1)
                }).catch(async error => {
                    await setBackResponse(error.response.data.message.code);
                    await setShowSnackBar(true);
                }).finally(
                );;
            } catch (error) {
                console.log(error);
            };
        break;
        case Operation.DELETE:
            try {
                await api.delete(urlBakc, {
                        params: {
                            id: findValueById(formValues, "id")
                        }
                    }
                ).then(response => {
                    const { status } = response;
                    navigate(-1);
                }).catch(async error => {
                    await setBackResponse(error.response.data.message.code);
                    await setShowSnackBar(true);
                }).finally(
```

Fonte: Acervo do autor (2022)

Esta parte do componente serializa todos os dados dos inputs que estão em um único objeto, para uma string JSON para mandá-la para o servidor, e lá ser tratado adequadamente.

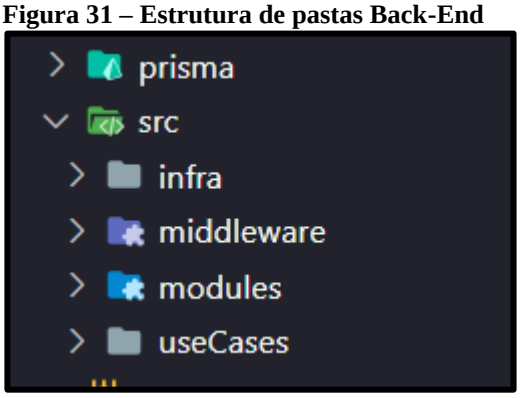
4.7 DESENVOLVIMENTO BACK-END

O desenvolvimento do back-end do protótipo foi totalmente feito em NodeJs por oferecer recursos que facilitem a implementação, tais como pacotes de terceiros que auxiliam para que não seja necessário reinventar a roda.

4.7.1 Estrutura de Pastas

Semelhante ao desenvolvimento do fron-end, o back-end também usa uma estrutura de pastas para a melhor organização dos arquivos do sistema, como pode ser visto na Figura 31.

Figura 31 – Estrutura de pastas Back-End



Fonte: Acervo do autor (2022)

Segue no Quadro 14 os objetivos de cada pasta dessa estrutura utilizada.

Quadro 14 – Estrutura de pastas Back-End

Pasta	Objetivo
prisma	Contém arquivos de migrations para abstração da camada de persistência.
src	Nomenclatura padrão para a pasta ontem contém todos os outros arquivos do sistema.
infra	Contém arquivos para a criação do servidor em Express bem como as suas rotas.
middleware	Contém funções que são executadas a cada requisição para coisas com autenticação.
modules	Contém arquivos para funcionalidades especificas do protótipo.
useCases	Contém arquivos para os CRUDS da aplicação.

Fonte: Acervo do autor (2022)

Essa estrutura permitiu que o código ficasse muito bem estrutura, e as suas intenções claras, o que facilita a sua manutenibilidade conforme o passar do tempo, além de facilitar a compreensão de outros desenvolvedores que podem usar o mesmo código.

4.7.2 Rotas

Para a criação das rotas do protótipo, foi utilizado a biblioteca Express, que permite uma criação de um servidor em NodeJS com menos trabalho e um resultado mais rápido. Podemos ver isso na Figura 32.

Figura 32 – Rotas Back-End

```
import ListWebServiceRoutesController from "../../../useCases/webServiceRo
import CreateWebServiceRoutesController from "../../../useCases/webService
import UpdateWebServiceRoutesController from "../../../useCases/webService
import DeleteWebServiceRoutesController from "../../../useCases/webService

const Routes = Router();

Routes.post("/users", CreateUserController.handle);
Routes.post("/login", AuthencitateUserController.handle);
Routes.get("/usuario", tokenAuthenticate, ListUserController.handle);

Routes.get("/table", tokenAuthenticate, ListTableController.handle);
Routes.post("/table", tokenAuthenticate, CreateTableController.handle);
Routes.put("/table", tokenAuthenticate, UpdateTableController.handle);
Routes.delete("/table", tokenAuthenticate, DeleteTableController.handle);

Routes.get("/menu", tokenAuthenticate, ListMenuController.handle);
Routes.post("/menu", tokenAuthenticate, CreateMenuController.handle);
Routes.put("/menu", tokenAuthenticate, UpdateMenuController.handle);
Routes.delete("/menu", tokenAuthenticate, DeleteMenuController.handle);
```

Fonte: Acervo do autor (2022)

Podemos averiguar que para a criação das rotas de get, post. put e delete, é só necessário a importação dos controller e a utilização deles como parâmetro de cada requisição.

4.7.3 Middleware de Autenticação

Para o desenvolvimento da autenticação da aplicação para manter uma segurança de acesso aos recursos, foi feito um middleware que a cada requisição feito no front-end, é feito uma verificação da validade do TOKEN de acesso, como visto da Figura 33.

Figura 33 – Middleware de autenticação

```
require('dotenv').config();
import { NextFunction, Request, Response } from "express";
import { verify } from "jsonwebtoken";

function tokenAuthenticate(
    request: Request,
    response: Response,
    next: NextFunction
) {
    const authToken = request?.headers?.authorization;
    if (!authToken) {
        return response.status(401).json({
            message: "Invalid Authotization",
            }
        );
    };
    const [, token] = authToken.split(" ");
    try {
        verify(token, `${process.env.JWT_SECRET}`);
        return next();
    } catch(err) {
        return response.status(401).json({
            message: "Invalid Token",
            }
        );
    };
};

export default tokenAuthenticate;       You, 3 months ago • re
```

Fonte: Acervo do autor (2022)

O middleware de autenticação verifica a validade do TOKEN de acesso JWT utilizando uma função de verificação, que dependendo do resultado, se for válido ou não, ele retorna sucesso ou bloqueia o acesso e retorna um código de erro.

4.7.4 Migrations

Migration é uma forma de abstrair o formato das tabelas de um banco de dados, na forma de modelo em um arquivo de Schema, o qual será utilizado para gerar as entidades na base conforme essa descrição, como pode ser visto na Figura 34.

Figura 34 – Schema para migrations

```
model Users {
  id             Int       @id @default(autoincrement())
  name           String    @db.VarChar(255)
  username       String    @unique @db.VarChar(255)
  email          String    @unique @db.VarChar(255)
  email_verified Boolean   @default(false)
  password       String    @db.VarChar(255)
  remember_token String?   @db.VarChar(255)
  status_id      Int?
  status         Status?   @relation(fields: [status_id], references: [id])
  created_at     DateTime? @default(now())
  updated_at     String?   @db.VarChar(255)

  @@map("tbusuarios")
}

model Menus {
  id           Int       @id @default(autoincrement())
  nome         String    @db.VarChar(255)
  parametros   String?   @db.VarChar(255)
  rota         String?   @db.VarChar(255)
  icone        String    @db.VarChar(255)
  pai_id       Int?
  component    String?   @db.VarChar(255)
  possuifilhos Boolean   @default(false)
  ordem        Int?
  desabilitado Boolean   @default(false)
  status_id    Int
  status       Status?   @relation(fields: [status_id], references: [id])
  created_at   DateTime? @default(now())
  updated_at   DateTime?

  @@map("tbmenus")
}
```

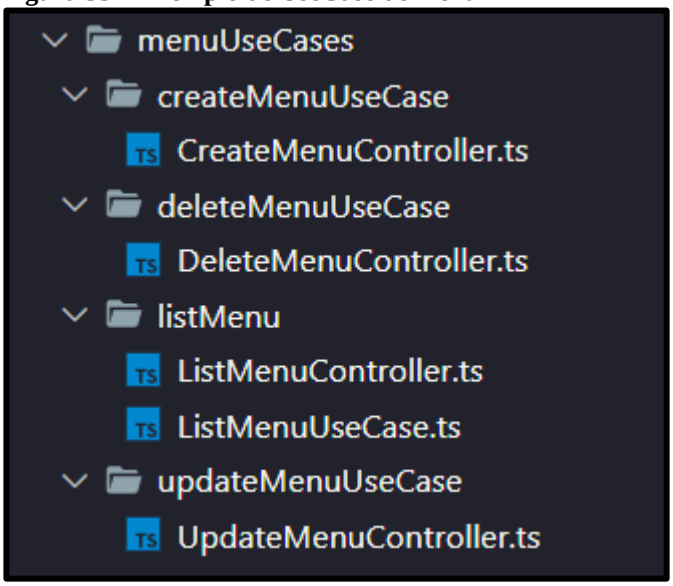
Fonte: Acervo do autor (2022)

A utilização dessa forma de interação com o banco de dados, permite com que sejam geradas diversão migrations para cada alteração nas tabelas do sistema, e assim podendo criar versão da base de dados, possibilitando o retorno para versões anteriores caso seja reportado algum bug.

4.7.5 UseCases

A de useCases serve para a separação de cada operação do CRUD em arquivos separados, assim em vez de centralizar as reponsabilidades dessas operações em um único controller, elas são separadas em 4 arquivos que facilitam a sua manutenção. Conforme visto na Figura 35.

Figura 35 – Exemplo de UseCase de menu



Fonte: Acervo do autor (2022)

No decorrer do desenvolvimento do projeto, essa estrutura permitiu uma experiencia de desenvolvimento mais satisfatória e produtiva, tendo em vista que quando fosse trabalhar em uma funcionalidade, ela estava em um arquivo dedicado somente a ela.

4.7.6 Modules

Os módulos são funcionalidade essenciais para os objetivos principais da aplicação. Existem dois módulos principais, sendo eles ExecuteWebServiceController e ObjToSqlController.

O modulo de ObjToSqlController tratar de transformar os arquivos JSON gerados no front para comandos SQL que serão executados nos seus webservices dedicados, conforme visto na Figura 36.

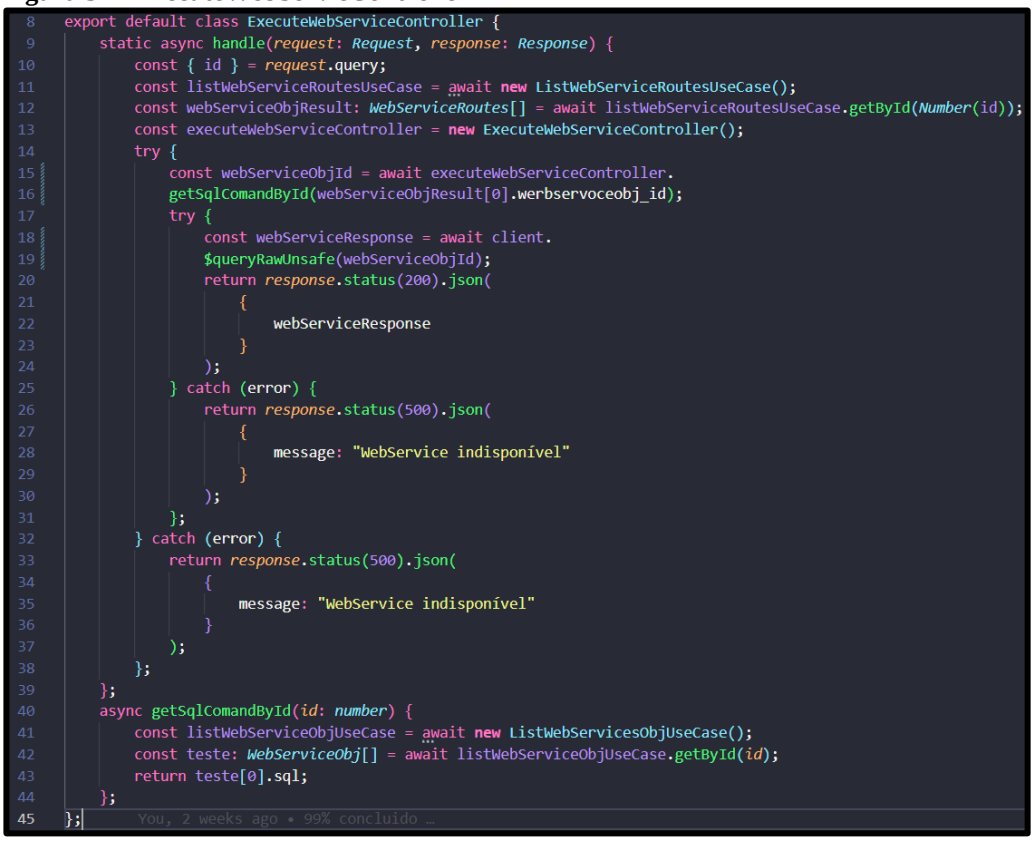
Figura 36 - ObjToSqlController

```
23      async ObjToSql(obj: any, id: number) {
24          const tableId = obj.tableId;
25          const objJson: any[] = obj.data;
26          const listTableUseCase = new ListTableUseCase();
27          let select = "SELECT ";
28          let columns = "";
29          let from = "FROM ";
30          let tb = await listTableUseCase.getById(tableId);
31          from = from + tb[0].nome;
32          await objJson.forEach(el => {
33              columns = columns + el.nome + " ,";
34          })
35          columns = columns.substring(0, columns.length - 1);
36          select = select + columns + from;
37          try {
38              const updateWebServiceObj = await client.webServiceObj.update({
39                  where: {
40                      id: Number(id)
41                  },
42                  data: {
43                      sql: select,
44                  }
45              });
46              if (updateWebServiceObj) {
47                  return true;
48              };
49          } catch (error) {
50              return false;
51          };
52
53      };
```

Fonte: Acervo do autor (2022)

Esse modulo criar variáveis do tipo string, e concatena com os campos JSON dos webservices presentes no banco da dados. Após gerar o comando, o registro do webservice no banco é atualizado com o novo comando para ser executado posteriormente no modulo de execução, como pode ser visto na Figura 37.

Figura 37 - ExecuteWebServieController

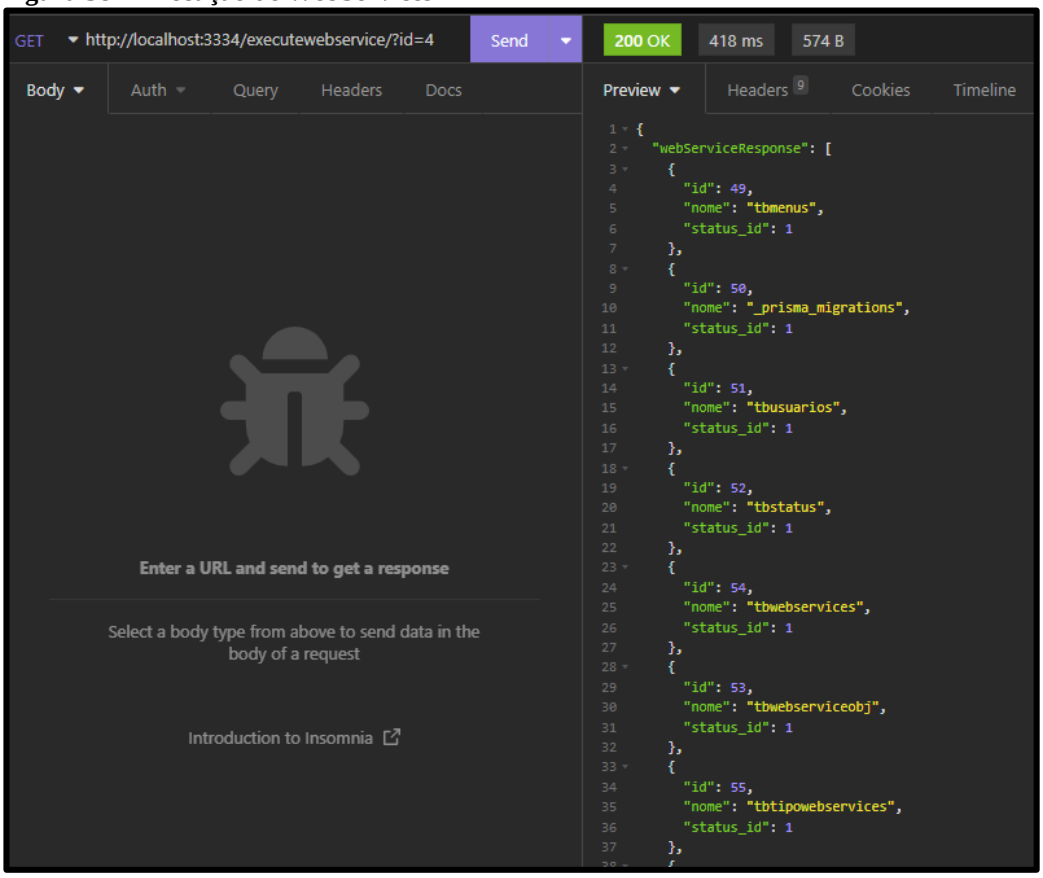


```
export default class ExecuteWebServiceController {
    static async handle(request: Request, response: Response) {
        const { id } = request.query;
        const listWebServiceRoutesUseCase = await new ListWebServiceRoutesUseCase();
        const webServiceObjResult: WebServiceRoutes[] = await listWebServiceRoutesUseCase.getById(Number(id));
        const executeWebServiceController = new ExecuteWebServiceController();
        try {
            const webServiceObjId = await executeWebServiceController.
            getSqlComandById(webServiceObjResult[0].werbservoceobj_id);
            try {
                const webServiceResponse = await client.
                $queryRawUnsafe(webServiceObjId);
                return response.status(200).json(
                    {
                        webServiceResponse
                    }
                );
            } catch (error) {
                return response.status(500).json(
                    {
                        message: "WebService indisponível"
                    }
                );
            };
        } catch (error) {
            return response.status(500).json(
                {
                    message: "WebService indisponível"
                }
            );
        };
    };
    async getSqlComandById(id: number) {
        const listWebServiceObjUseCase = await new ListWebServicesObjUseCase();
        const teste: WebServiceObj[] = await listWebServiceObjUseCase.getById(id);
        return teste[0].sql;
    };
};      You, 2 weeks ago • 99% concluido …
```

Fonte: Acervo do autor (2022)

Esse modulo por sua vez, é executado por uma URL padrão do sistema, que recebe como parâmetro o id do objeto específico do webservice criado, esse webservice é executado e devolve os resultados da consulta na resposta da requisição. Como visto na Figura 38.

Figura 38 – Execução de WebServices



Fonte: Acervo do autor (2022)

Conforme pode ser visto na Figura 39 anteriormente, a execução dos webservices do protótipo retorna uma response com os dados dos campos solicitados por cada webservice cadastrado no protótipo, assim sem a necessidade de cria-los individualmente.

4.7.7 Dicionário de dados

O dicionário de dados é uma forma de armazenar metadados da base de dados da aplicação em tabelas que armazenam os nomes da base, as suas tabelas e seus campos, com as características de cada um, tais como nome do campo, formato de dado e tamanho máximo. Conforme visto na Figura 39.

Figura 39 – Create tables

```
export class CreateTableUseCase {
    async creatTables(data?: any) {
        const tables = client.tabelas.create({
            data: {
                nome: data.table_name,
                scheme: data.table_schema,
                base_id: data.base_id,
                status_id: 1,
            }
        });
        return tables;
    };

    async generateTableDictionari() {
        let tables: any[] = [];
        const listtableUseCase = new ListTableUseCase();
        tables.push(await listtableUseCase.getTable());
        tables = tables[0];

        tables.forEach(async (table) => {
            await this.creatTables(table);
        });
    };
};
```

Fonte: Acervo do autor (2022)

Essa classe é responsável por coletar os nomes das tabelas do sistema e armazenadas em uma tabela especifica para ser utilizada posteriormente na catalogação das colunas como poder ser visto na Figura 40.

Figura 40 – Catalogação de colunas por tabela

```
export class CreateColunaUseCase {
    async createColumn(table_id: number, columns: any[]) {
        columns.forEach(async (column) => {
            await client.colunas.create({
                data: {
                    nome: column.column_name,
                    tipo: column.data_type,
                    posicao: column.ordinal_position,
                    nulo: column.is_nullable,
                    char_max: column.character_maximum_length,
                    is_identity: column.is_identity,
                    is_self_referencing: column.is_self_referencing,
                    is_updatable: column.is_updatable,
                    tabela_id: table_id,
                    status_id: 1,
                }
            });
        });
    };

    async generateColumnDic() {
        let tables: any[] = [];
        let colunas: any[] = [];
        const listTableUseCase = new ListTableUseCase();
        const listColunaUseCase = new ListColunaUseCase();
        tables.push(await listTableUseCase.getAll());
        tables = tables[0];
        for (let i = 0; i < tables.length; i++) {
            colunas.push(await listColunaUseCase.getColunaByTableName(tables[i].nome));
        }
        colunas.forEach(async (column) => {
            await this.createColumn(column.table_id, column.columns);
        });
    };
};
```

Fonte: Acervo do autor (2022)

O resultado das operações feitas nessa classe é uma tabela na base da dados com as informações das colunas de cada tabela publica do sistema que são consideradas essenciais para o bom funcionamento e organização dos metadados para a utilização futura na criação dos webservices.

5 CONCLUSÃO

Esse trabalho de conclusão de curso teve como principal objetivo a prototipação de um protótipo automatizado de gerador de WebService REST para ser utilizado por outros desenvolvedores.

O intuito do protótipo também é facilitar a vida dos desenvolvedores em seus trabalhos profissionais ou pessoais. E com a componentização do React e o Back-End feito com NodeJS, esse trabalho foi possível de ser desenvolvido com um código bastante reutilizável, sendo possível utilizar para outros propósitos.

Com relação ao objetivo geral, ele foi concluído com sucesso, visto que a plataforma desenvolvida consegui os resultados desejados como uma ferramenta funcional para criação de webservices REST para consulta de dados de uma base.

Por sua vez os objetivos específicos também foram todos concluídos, no início do projeto, alguns desses objetivos pareciam mais complexos, porém no decorrer do desenvolvimento do trabalho, foi possível desenvolvê-los todos de forma prática, por consequência, formando o que se esperava no início do planejamento.

O servidor da aplicação também consegui cumprir com os três principais objetivos, a criação do dicionário de dados que permite que a aplicação tenha um conhecimento próprio das suas tabelas, que podem ser utilizadas nos webservices. O objetivo de transformar JSON em SQL também foi cumprido, sendo esse o módulo que permite a execução dos serviços, e o módulo de centralização dos webservices, que torna possível executá-los de forma facilitada.

Os resultados gerais da aplicação foram bem-sucedidos, tendo em vista que os funcionamentos batem com as especificações e o resultado descrito na metodologia foi concluído.

5.1. TRABALHOS FUTUROS

O protótipo inicial está concluído, e com as funcionalidades primárias concluídas, o que abre espaço para novas abordagens que podem abranger ainda mais os serviços que poderiam ser oferecidos.

Algumas novas funcionalidades podem ser implementadas nesse projeto como; transferir o acesso ao banco de dados em um microsserviço, fazendo com que o acesso não esteja sendo feito de forma direta, o que ajuda na segurança; além disso pode-se também

implementar uma função para inserir, alterar e deletar os dados de uma base e não se limitar a consulta.

Para tornar o sistema mais intuitivo, é preciso criar uma área especifica no front-end somente para testar os webservices criados, levando o usuário final a ter uma maior assertividade e certeza de que está tudo funcionando perfeitamente.

Outro ponto a se pensar é na possibilidade de importar webservices de terceiros, e gerar uma integração automática entre eles, de forma que seja possível direcionar de onde esses dados veem e para onde mandá-los.

REFERÊNCIAS

AQUILES, Alexandre; FERREIRA, Rodrigo. **Controlando versões com Git e GitHub.** São Paulo: Casa do Código, 2014. 220 p.

BUZZI, Felipe. **Prisma:** uma das melhores coisas que já aconteceu no ecossistema?. Uma das melhores coisas que já aconteceu no ecossistema?. 2022. Disponível em: https://blog.rocketseat.com.br/prisma-uma-das-melhores-coisa-que-ja-aconteceu-no-ecossistema/. Acesso em: 23 jun. 2022.

CARVALHO, Vinícius. **PostgreSQL:** banco de dados para aplicações web modernas. São Paulo: Casa do Código, 2017. 220 p.

DUCKETT, Jon. **HTML and CSS:** projete e construa websites. Rio de Janeiro: Alta Books, 2016. 512 p.

ELMASRI, Ramez; NAVATHE, Shamkant B.. **Sistemas de Banco de Dados.** 4. ed. São Paulo: Pearson, 2005. 744 p.

FONSECA, Elton. **O que é ORM?** 2020. Disponível em: https://www.treinaweb.com.br/blog/o-que-e-orm. Acesso em: 23 jun. 2022.

GOMES, Rafael Corrêa. **Comandos Git**: aprenda git do básico ao avançado. Aprenda Git do básico ao avançado. 2016. Disponível em: https://comandosgit.github.io/. Acesso em: 09 jun. 2022.

GONÇALVES, Ariane. **O que é CSS?** Guia Básico para Iniciantes. Disponível em: https://www.hostinger.com.br/tutoriais/o-que-e-css-guia-basico-de-css#:~:text=CSS%20%C3%A9%20a%20sigla%20para,de%20marca%C3%A7%C3%A3o%20HTML%20ou%20XHTML.. Acesso em: 09 jun. 2022.

PEREIRA, Caio Ribeiro. **Aplicações web real-time com Node.js.** São Paulo: Casa do Código, 2013. 186 p.

PONTES, Guilherme. **Progressive Web Apps:** construa aplicações progressivas com react. São Paulo: Casa do Código, 2018. 443 p.

SAUDATE, Alexandro. **REST: construa API's inteligentes de maneira simples.** 1. ed. São Paulo: 2013.

STEFANOV, Stoyan. **Primeiros passos com React:** construindo aplicações web. São Paulo: Novatec, 2019. 279 p.

ZEMEL, Tárcio. **CSS Eficiente:** técnicas e ferramentas que fazem a diferença nos seus estilos. São Paulo: 1, 2015. 144 p.