UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
DCET – DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS CURSO DE CIÊNCIA DA COMPUTAÇÃO

DIONLENO SILVA DE OLIVEIRA

DESENVOLVIMENTO DO JAPIS - UM GERADOR DE API RESTFUL COM BANCO DE DADOS NOSQL

VITÓRIA DA CONQUISTA - BA
MARÇO - 2021

DIONLENO SILVA DE OLIVEIRA

DESENVOLVIMENTO DO JAPIS UM GERADOR DE API RESTFUL COM BANCO DE DADOS NOSQL

Trabalho de conclusão de curso, para aprovação na disciplina Trabalho Supervisionado II e como requisito parcial para obtenção do título de Bacharel em Ciências da Computação, na Universidade Estadual do Sudoeste da Bahia - UESB.

Orientadora: Professora Dra. Maísa Soares dos Santos Lopes.

VITÓRIA DA CONQUISTA - BA
MARÇO - 2021

RESUMO

Este trabalho apresenta o projeto JAPIS, um gerador de API RESTful com banco de dados NoSQL, que tem como proposta unir a rapidez de desenvolvimento do BaaS e a flexibilidade ao produzir a própria API. Através da modelagem de uma API baseada em coleções com uma interface gráfica, o JAPIS gera o código com o CRUD completo das coleções e conta com uma rota específica para realizar buscas inteligentes. O código gerado segue os padrões de projeto MVC e DAO facilitando a implementação de novas funcionalidades no código gerado. Uma API de guia de restaurantes foi gerada e novas funcionalidades foram inseridas para a validação do sistema.

Palavras chaves: GERADOR DE CÓDIGO, API, REST, CRUD.

ABSTRACT

This work presents the JAPIS project, a RESTful API generator with NoSQL database, which proposes to combine the speed of BaaS development and the flexibility to produce the API itself. Through the modeling of a collection-based API with a graphical interface, JAPIS generates the code with the complete CRUD of the collections and has a specific route to perform intelligent searches. The generated code follows the MVC and DAO design standards, facilitating the implementation of new features in the generated code. A restaurant guide API was generated and new features were added for system validation.

Keywords: CODE GENERATOR, API, REST, CRUD.

Dedico este trabalho aos meus pais, meus avós, são eles as bases que me tornaram a pessoa que sou hoje, e a minha esposa por está sempre ao meu lado nessa jornada.

AGRADECIMENTOS

A minha família, que é minha base.

Aos meus professores que me ensinaram que a prática sem a teoria é falha.

Aos meus colegas de curso que fizeram com que a viagem fosse mais suportável.

A minha orientadora profa. Maísa, pela pessoa maravilhosa que é, e por ter aceitado esse desafio.

A Celina, pela pessoa maravilhosa, atenciosa e pelo carinho que sempre teve comigo.

“Se não der certo da primeira vez,
chame de versão 1.0.”

Sydney J. Harris

LISTA DE FIGURAS

Figura 01 - Introduction to REST API’s
Figura 02 - Data Modeling for API’s
Figura 03 - Custos de alterações como uma função do tempo em desenvolvimento
Figura 04 - Funcionamento do SCRUM
Figura 05 - Módulos do sistema
Figura 06 - Ferramenta Insomnia
Figura 07 - Ferramenta Mongo DB Compass
Figura 08 - Estrutura de dados - nome da coleção e descrição do atributos
Figura 09 - Estrutura de dados - configurações do projeto
Figura 10 - Estrutura de dados - regras de autenticação
Figura 11 - Comportamento do gerador
Figura 12 - Tela de modelagem das coleções
Figura 13 - Tela de configuração da API
Figura 14 - Tela de gerar a API
Figura 15 - API gerada descompactada
Figura 16 - Instalação das dependências
Figura 17 - API disponível
Figura 18 - Atribuindo função de admin para usuário
Figura 19 - Gerando token de admin
Figura 20 - Exemplo de estrutura de diretórios
Figura 21 - Modelagem de restaurante
Figura 22 - Modelagem de cardápio
Figura 23 - Configurações gerais
Figura 24 - Gerando código
Figura 25 - Verificando se API está funcionando
Figura 26 - Criando um usuário
Figura 27 - Gerando token de autenticação
Figura 28 - Renovando o token de autenticação
Figura 29 - Cadastro de restaurante
Figura 30 - Listagem de restaurantes

Figura 31 - Cadastro de cardápio
Figura 32 - Atualização de restaurante
Figura 33 - Remoção de restaurante
Figura 34 - Pesquisa por nome
Figura 35 - Pesquisa por nome retornando apenas um objeto
Figura 36 - Pesquisa por id com com filter de localização
Figura 37 - Pesquisa com populate de cardápio
Figura 38 - Pesquisa com populate, filter e findOne
Figura 39 - Instalação da biblioteca json2csv
Figura 40 - Alteração do arquivo restaurante-controller.js
Figura 41 - Alteração do arquivo route.js
Figura 42 - Exportando arquivo csv
Figura 43 - Alteração do arquivo restaurante-controller.js
Figura 44 - Alteração do arquivo restaurante-dao.js
Figura 45 - Alteração do arquivo restaurante.js
Figura 46 - Alteração do arquivo restaurante-route.js
Figura 47 - Listando restaurantes próximos

LISTA DE ABREVIATURAS E SIGLAS

API - Application Programming Interface
BaaS - Backend as a Service
CRUD - Create, Read, Update e Delete
GUI - Graphical User Interface
HTTP - HyperText Transfer Protocol
SOAP - Simple Object Access Protocol
JSON - JavaScript Object Notation
MD5 - Message-Digest algorithm 5
NoSQL - Não Somente SQL
REST - Representational State Transfer
TI - Tecnologia da informação
URI - Uniform Resource Identifier
URL - Uniform Resource Locator
XML - Extensible Markup Language

SUMÁRIO

INTRODUÇÃO **14**
JUSTIFICATIVA E MOTIVAÇÃO 15
OBJETIVOS 16
Objetivo geral 16
Objetivos específicos 16
METODOLOGIA 16
ORGANIZAÇÃO DO TRABALHO 17

REFERENCIAL TEÓRICO **18**
HTTP 18
Métodos de requisição 18
Cabeçalhos HTTP 19
Media Types 19
Códigos de status 20
WEB SERVICES 22
Tecnologias para a construção de Web Services 23
API RESTful 23
BAAS 24
BANCO DE DADOS NOSQL 25

DESENVOLVIMENTO DO SISTEMA **26**
SCRUM SOLO 26
REQUISITOS DO SISTEMA 28
Requisitos funcionais 28
Requisitos não-funcionais 30
FERRAMENTAS E LINGUAGENS UTILIZADAS 30
Insomnia 30
Mongo DB Compass 31
Node JS 32
API DO SISTEMA 33
FRONTEND DO SISTEMA 36
API GERADA 38
Primeiros passos 38
Estrutura de pastas e arquivos 41

GERANDO E UTILIZANDO UMA API DE GUIA DE RESTAURANTE **42**
ROTAS BASES E ROTAS DE USUÁRIOS 45
ROTAS DAS COLEÇÕES DE RESTAURANTES E CARDÁPIOS 46
ADICIONANDO AS FUNCIONALIDADES DE EXPORTAR CSV E LISTAR PRÓXIMOS NA COLEÇÃO DE RESTAURANTES 51
ANÁLISE DOS RESULTADOS 56

CONCLUSÃO E TRABALHOS FUTUROS **57**

REFERÊNCIAS **58**

1. INTRODUÇÃO

A necessidade de integrar sistemas criados em diferentes linguagens e com as mais variadas funcionalidades cada vez maior no setor de tecnologia da informação (TI), surgiu a necessidade de disponibilizar uma comunicação entre aplicações, seja ela, mobile, web ou desktop, com servidores web distintos.

A fim de sanar questões como estas, a tecnologia dos Web Services foi criada, permitindo assim, disponibilizar formas de integrar sistemas distintos, modularizar serviços e capacitar a integração e consumo de informações. Os Web Services tem se apresentado como uma nova e importante tecnologia de computação.

Dentre as arquiteturas de web services disponíveis, temos o Simple Object Access Protocol (SOAP), Representational State Transfer (REST) (Fielding, 2020) e o GraphQL.

SOAP é um protocolo para troca de informações estruturadas em uma plataforma descentralizada e distribuída, utiliza o protocolo HyperText Transfer Protocol (HTTP), limitando-se ao método POST para realizar múltiplas operações. Sua mensagem é baseada em Extensible Markup Language (XML) (POLO, 2018).

O modelo REST utiliza 100% de tudo que o protocolo HTTP oferece. O REST é um dos padrões mais utilizados atualmente, muito disso por oferecer uma interface uniforme e o princípio de endereçamento (OLIVEIRA, 2014).

O GraphQL, é definido como um modelo para consulta de Interface de Programação de Aplicativos (API), baseado em uma linguagem declarativa e um sistema de tipagem, que descreve as capacidades de requisitos dos modelos de dados, para operações entre cliente e servidor (VIEIRA, 2019).

Desenvolver aplicações web services não é trivial e exige profissionais qualificados, que não são fáceis de encontrar. Como alternativa, pode ser utilizado Backend as a Service (BaaS) ou Backend como Serviço. BaaS é um serviço de computação em nuvem que permite delegar as responsabilidades inerentes a manutenção e gerenciamento de servidores para terceiros e fornece ferramentas que possibilitam a criação de código de backend, o que acelerar o processo de

desenvolvimento e gera ganho de produtividade, diminuindo assim os custos do desenvolvimento do software (BATSCHINSKI, 2019).

Entretanto, os Baas tem desvantagens: Menos flexibilidade em comparação com a codificação customizada, nível mais baixo de personalização em comparação com um backend personalizado, aprisionamento tecnológico, também conhecido como Vendor Lock-In para plataformas de código fechado, desenvolvedor não tem domínio sobre o código e dependência da plataforma.

Com a necessidade ampla de integrar sistemas em um espaço de tempo cada vez mais curto, surge a necessidade de uma ferramenta que agilizasse esse processo e ao mesmo tempo seja eficiente e segura para o usuário. Para facilitar a geração de Web Services é necessário ter ferramentas que automatizam os processos de desenvolvimento, deixando os complicados detalhes de implementação escondidos do desenvolvedor (OLIVEIRA, 2019).

1.1. JUSTIFICATIVA E MOTIVAÇÃO

Para grande maioria das empresas que envolvem TI, disponibilizar uma API para integração seja ela, mobile, web ou desktop é fundamental atualmente. Cada vez mais se faz necessário o consumo de dados de um backend.

Com o valor de desenvolvimento de backend alto, uma solução é usar Baas, ganhando produtividade e diminuindo custos.

No entanto, os BaaS tem algumas desvantagens. Os usuários de BaaS ficam dependentes de uma plataforma, impedindo eles de migrar de provedor e estão sujeitos aos preços e planos condicionados pela plataforma. Por esse motivo, o presente trabalho sugere uma ferramenta que tenha uma modelagem simplificada com as das plataformas Baas, e que o programador/empresa seja o proprietário do seu código e o banco de dados Não Somente SQL (NoSQL) para hospedar e migrar seus projetos onde, como e quando quiser.

Os bancos de dados NoSQL tornaram-se os mais adequados para armazenamento de grandes volumes de dados não estruturados ou semiestruturados. Suas características principais são a escalabilidade horizontal, suporte nativo para replicação e API simples para acesso aos dados.

1.2. OBJETIVOS

1.2.1. Objetivo geral

Desenvolver uma ferramenta gráfica capaz de gerar código de API Restful com banco de dados NoSQL que auxilie no processo de desenvolvimento de software.

1.2.2. Objetivos específicos

- Analisar e modelar uma ferramenta gráfica de geração de código de API Restful.
- Permitir que a ferramenta suporte rotas, autenticação, autorização e banco de dados
- Estruturar a segurança da ferramenta baseando-se na implementação de meios de autenticação e autorização das Uniform Resource Identifier (URI) da API e também de algoritmos de criptografia para os dados críticos do banco de dados.
- Permitir que a API gerada tenha um código que facilite atualizações de funcionalidades.

1.3. METODOLOGIA

Este trabalho explora o desenvolvimento de sistemas de software utilizando tecnologia Web Service, mais especificamente a geração de API Restful.

Quanto a natureza da pesquisa, é uma pesquisa aplicada, pois tem como objetivo gerar conhecimentos para aplicação prática, dirigida à solução de problemas específicos (Kauark et al., 2010). Quanto a sua finalidade, é uma pesquisa exploratória, haja vista que a mesma tem por objetivo proporcionar uma maior familiaridade com o desenvolvimento de sistemas Web Service utilizando API Restful (GIL, 2002). Para o desenvolvimento do trabalho foram seguidas as seguintes etapas (Selltiz et ai., 1967, p. 63 apud Gil, 2002): (a) levantamento bibliográfico, foram utilizados documentos científicos e técnicos que ajudaram a delimitar melhor o trabalho e conhecer as contribuições já implementadas; (b) análise de ferramentas, foram analisadas serviços BaaS e ferramentas voltadas ao

desenvolvimento Web Services; (c) criação da ferramenta de geração de API Restful e; (d) teste da ferramenta desenvolvida.

A metodologia de desenvolvimento de software utilizada é a scrum solo, visto que este é um trabalho individual e que será implementado um conjunto de itens selecionados a partir de Product Backlog (PAGOTTO, 2016).

1.4. ORGANIZAÇÃO DO TRABALHO

O texto deste trabalho será organizado em cinco capítulos, além deste primeiro, dispostos a seguir:

- O segundo capítulo, apresenta o referencial teórico do trabalho.
- O terceiro capítulo, mostra a arquitetura da ferramenta e como ela foi desenvolvida.
- O quarto capítulo, mostra o teste do JAPIS, gerando uma API de guia de restaurantes e análise dos resultados.
- O quinto capítulo, conclusão e trabalhos futuros.

2. REFERENCIAL TEÓRICO

Neste presente capítulo são apresentados conceitos teóricos sobre o desenvolvimento de API's, com destaque para as API's RESTful, também é tratado sobre backend com serviço, o BAAS e sobre os bancos de dados NoSQL.

2.1. HTTP

O HTTP é um protocolo de comunicação para distribuição de objetos de hipermídia referenciados por uma *Uniform Resource Locator* (URL). Este protocolo de transferência é utilizado em toda a World Wide Web (TANENBAUM, 2003). A função do HTTP é bastante simples: realizar requisições e receber respostas entre um cliente e servidor (EMER, 2014).

A transmissão de informações entre um emissor e um receptor caracteriza uma comunicação. E da mesma forma que uma comunicação interpessoal, a cordialidade é essencial. O protocolo HTTP estabelece um cabeçalho para suas requisições e respostas. O cabeçalho de uma mensagem são informações complementares que são de uso do cliente e servidor. Através das informações passadas pelo cabeçalho, que são de uso exclusivo do servidor e cliente, é possível informar em uma requisição a preferência de idiomas, as codificações e os formatos de conteúdo para uma resposta. O cabeçalho da resposta, por sua vez, contém o código de status, codificação, formato e tempo de expiração do conteúdo (EMER, 2014).

2.1.1. Métodos de requisição

Durante a requisição é utilizado um Método de requisição (Ou Verbo). Existem 8 métodos: GET, POST, DELETE, HEAD, PUT, OPTIONS, TRACE e CONNECT (OLIVEIRA, 2015).

Os mais utilizados são:

- **GET** : utilizado para solicitar um recurso. Não utilizado para executar uma ação, somente para recuperar informações.
- **POST** : utilizado para executar uma ação ou criar um novo recurso.
- **PUT :** utilizado para atualizar um recurso.
- **DELETE :** utilizado para remover um recurso.

- **HEAD** : recupera informações sobre um recurso.

2.1.2. Cabeçalhos HTTP

Os cabeçalhos, em HTTP, são utilizados para trafegar todo o tipo de meta informação a respeito das requisições. Vários destes cabeçalhos são padronizados; no entanto, eles são facilmente extensíveis para comportar qualquer particularidade que uma aplicação possa requerer nesse sentido (SAUDATE, 2013).

Alguns dos cabeçalhos mais utilizados são:

- **Host:** mostra qual foi o DNS utilizado para chegar a este servidor.
- **User-Agent:** fornece informações sobre o meio utilizado para acessar este endereço.
- **Accept:** realiza negociação com o servidor a respeito do conteúdo aceito.
- **Accept-Language:** negocia com o servidor qual o idioma a ser utilizado na resposta.
- **Accept-Encoding:** negocia com o servidor qual a codificação a ser utilizada na resposta.
- **Connection:** ajusta o tipo de conexão com o servidor (persistente ou não).
- **Content-Type:** indica o tipo e o subtipo do conteúdo da mensagem.

2.1.3. Media Types

Os Media Types são formas padronizadas de descrever uma informação. São divididos em tipo e subtipos.

Os tipos mais comuns são:

- **application**
- **audio**
- **image**
- **text**
- **video**
- **vnd**

Cada um desses tipos é utilizado com diferentes propósitos. Application é utilizado para tráfego de dados específicos de certas aplicações. Audio é utilizado

para formatos de áudio. Image é utilizado para formatos de imagens. Text é utilizado para formatos de texto padronizados ou facilmente inteligíveis por humanos. Video é utilizado para formatos de vídeo. Vnd é para tráfego de informações de softwares específicos (por exemplo, o Microsoft Office).

Em serviços REST, vários tipos diferentes de Media Types são utilizados. As maneiras mais comuns de representar dados estruturados, em serviços REST, são via Extensible Markup Language (XML) e JavaScript Object Notation (JSON), que são representados pelos Media Types application/xml e application/json , respectivamente (SAUDATE, 2013).

2.1.4. Códigos de status

Saudate (2013, p. 23) diz que 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:

- **1xx** - Informacionais
- **2xx** - Códigos de sucesso
- **3xx** - Códigos de redirecionamento
- **4xx** - Erros causados pelo cliente
- **5xx** - Erros originados no servidor

Os códigos mais importantes e utilizados são:

2xx

- **200 - OK -** Indica que a operação indicada teve sucesso.
- **201 - Created -** Indica que o recurso desejado foi criado com sucesso. Deve retornar um cabeçalho Location , que deve conter a URL onde o recurso recém-criado está disponível.
- **202 - Accepted -** 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. Por esse motivo, pode retornar um

cabeçalho Location , que trará uma URL onde o cliente pode consultar se o recurso já está disponível ou não.

- **204 - No Content -** Usualmente enviado em resposta a uma requisição PUT , POST ou DELETE

onde o servidor pode recusar-se a enviar conteúdo.

- **206 - Partial Content -** Utilizado em requisições GET parciais, ou seja, que demandam apenas parte do conteúdo armazenado no servidor (caso muito utilizado em servidores de download).

3xx

- **301 - Moved Permanently -** Significa que o recurso solicitado foi realocado permanentemente. Uma resposta com o código 301 deve conter um cabeçalho Location com a URL completa (ou seja, com descrição de protocolo e servidor) de onde o recurso está atualmente.
- **303 - See Other -** É utilizado quando a requisição foi processada, mas o servidor não deseja enviar o resultado do processamento. Ao invés disso, o servidor envia a resposta com este código de status e o cabeçalho Location , informando onde a resposta do processamento está.
- **304 - Not Modified -** É utilizado, principalmente, em requisições GET condicionais - quando o cliente deseja ver a resposta apenas se ela tiver sido alterada em relação a uma requisição anterior.
- **307 - Temporary Redirect -** Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.

4xx

- **400 - Bad Request -** É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.
- **401 - Unauthorized -** Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação (ou a autenticação fornecida for inválida).
- **403 - Forbidden -** Utilizado quando o cliente está tentando realizar uma operação sem ter a devida autorização.
- **404 - Not Found -** Utilizado quando o recurso solicitado não existe.

- **405 - Method Not Allowed -** Utilizado quando o método HTTP utilizado não é suportado pela URL. Deve incluir um cabeçalho Allow na resposta, contendo a listagem dos métodos suportados (separados por “,”).
- **409 - Conflict -** Utilizado quando há conflitos entre dois recursos. Comumente utilizado em resposta a criação de conteúdos que tenham restrições de dados únicos - por exemplo, criação de um usuário no sistema utilizando um login já existente. Se for causado pela existência de outro recurso (como no caso citado), a resposta deve conter um cabeçalho Location , explicitando a localização do recurso que é a fonte do conflito.
- **410 - Gone -** Semelhante ao 404, mas indica que um recurso já existiu neste local.
- **412 - Precondition failed -** Comumente utilizado em resposta a requisições GET condicionais.
- **415 - Unsupported Media Type -** Utilizado em resposta a clientes que solicitam um tipo de dados que não é suportado - por exemplo, solicitar JSON quando o único formato de dados suportado é XML.

5xx

- **500 - Internal Server Error -** É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica.
- **503 - Service Unavailable -** Indica que o servidor está atendendo requisições, mas o serviço em questão não está funcionando corretamente.

2.2. WEB SERVICES

Um serviço da Web (Web Services) é um termo genérico para uma função de software interoperável máquina a máquina hospedada em um local endereçável na rede.

Um Web Services possui uma interface, que oculta os detalhes da implementação, para que possa ser usado independentemente da plataforma de hardware ou software em que é implementado e independentemente da linguagem de programação na qual está escrito. Essa independência incentiva os aplicativos baseados em Web Services a serem implementações de tecnologias cruzadas de baixo acoplamento, orientadas a componentes. Os Web Services podem ser usados

sozinhos ou com outros serviços da Web para realizar uma agregação complexa ou uma transação comercial (IBM, 2019).

2.2.1. Tecnologias para a construção de Web Services

- SOAP - No SOAP as mensagens trocadas entre serviço e cliente consumidor devem ser armazenadas em envelopes SOAP. Este protocolo de comunicação dita um formato de envio de mensagens entre aplicações, o qual é descentralizado e distribuído, ou seja, qualquer plataforma de comunicação pode ser utilizada, seja ela proprietária ou não [W3C 2000].

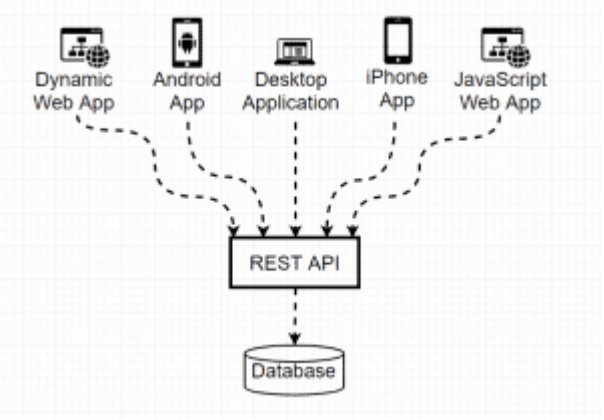
- REST - O Representational State Transfer (REST) é um estilo de arquitetura para sistemas de hipermídia distribuídos. Esse termo foi utilizado pela primeira vez por Roy Fielding (um dos criadores do protocolo HTTP), em sua tese de doutorado publicada no ano 2000. Assim, é perceptível que o protocolo REST é guiado pelo que seriam as boas práticas de uso de HTTP (SAUDATE, 2013, p. 26).

- GRAPHQL - O GraphQL é uma linguagem de consulta para API's que realiza as consultas em tempo de execução com os dados existentes. O GraphQL fornece uma descrição completa e compreensível dos dados em sua API, oferece aos clientes o poder de solicitar exatamente o que eles precisam e nada mais, facilita a evolução das API's ao longo do tempo e permite poderosas ferramentas de desenvolvedor (GRAPHQL, 2019) .

2.3. API RESTful

API RESTful é uma interface de programação que segue o modelo arquitetural REST. Uma Application Programming Interface (API) é um conjunto de definições e protocolos para criar e integrar softwares de aplicações(FIELDING, 2000). Diversos tipos de sistemas podem consumir uma API, seja ele mobile, desktop ou web, como mostra a Figura 01.

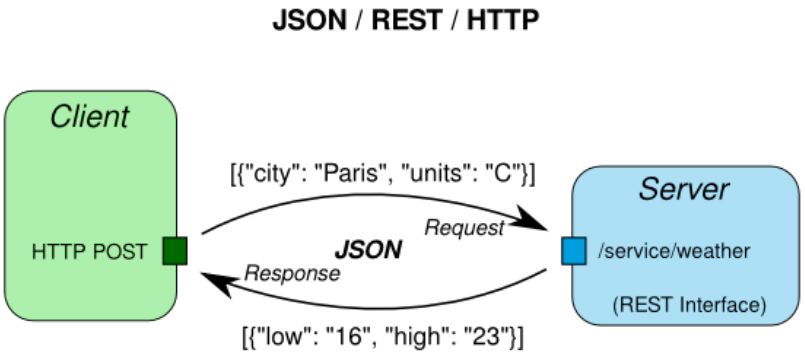
Figura 01: Introduction to REST API's



Fonte: Parallels, 2019

Quando uma solicitação é feita por meio de uma API RESTful, essa API transfere uma representação do estado do recurso ao solicitante. Essa informação, ou representação, é fornecida utilizando um dos vários formatos possíveis via HTTP: Javascript Object Notation (JSON), HTML, XLT ou texto simples. O formato JSON é o mais usado porque, apesar de seu nome, é independente de qualquer linguagem e pode ser lido por máquinas e humanos (REDHAT, 2019). Na Figura 02 é mostrado um exemplo de uma interação com uma API usando JSON na comunicação.

Figura 02: Data Modeling for API's



Fonte: Linked, 2014

2.4. BAAS

BaaS ou mBaaS ou Backend como Serviço é um plataforma que automatiza o desenvolvimento de backend e cuida da infraestrutura de nuvem. Utilizando um

BaaS, você terceiriza as responsabilidades inerentes a manutenção e gerenciamento de servidores para um terceiro e para focar no desenvolvimento do frontend. Além disso, um BaaS fornecerá um conjunto de ferramentas para ajudá-lo a criar um código de backend e acelerar o processo de desenvolvimento. Ele está pronto para usar recursos como gerenciamento de dados, APIs, integrações de mídia social, armazenamento de arquivos e notificações push.

Vantagens de um backend como serviço

- Velocidade de desenvolvimento
- Preço de desenvolvimento
- É serverless e você não precisa gerenciar a infraestrutura

Desvantagens de um backend como serviço

- Menos flexibilidade em comparação com a codificação customizada
- Um nível mais baixo de personalização em comparação com um backend personalizado
- Aprisionamento tecnológico, Vendor Lock-In para plataformas de código fechado

2.5. BANCO DE DADOS NOSQL

Sabe-se que existem vários modelos de Banco de Dados (BD), porém com o crescimento acelerado de informações geradas diariamente, percebeu-se a necessidade de desenvolver uma alternativa para resolver problemas vinculados à estruturação do Banco de Dados, permitindo uma maior flexibilidade dos dados armazenados. Segundo Soares (2012, p. 17) o BD NoSQL é uma alternativa para lidar com esses casos, por possuir uma arquitetura diferenciada, de forma a facilitar o tratamento com alta escalabilidade de dados.

Os bancos de dados NoSQL apresentam algumas características fundamentais que os diferenciam dos tradicionais sistemas de bancos de dados relacionais, tornando-os adequados para armazenamento de grandes volumes de dados não estruturados ou semiestruturados. A seguir, descrevemos algumas destas características (LÓSCIO; OLIVEIRA; PONTES, 2011).

- **Escalabilidade horizontal** - à medida que o volume de dados cresce, aumenta a necessidade de escalabilidade e melhoria de desempenho.
- **Ausência de esquema ou esquema flexível** - uma característica evidente dos bancos de dados NoSQL é a ausência completa ou quase total do esquema que define a estrutura dos dados modelados.
- **Suporte nativo a replicação** - outra forma de prover escalabilidade é através da replicação. Permitir a replicação de forma nativa, diminui o tempo gasto para recuperar informações.
- **API simples para acesso aos dados** - o objetivo da solução NoSQL é prover uma forma eficiente de acesso aos dados, oferecendo alta disponibilidade e escalabilidade, ou seja, o foco não está em como os dados são armazenados e sim como poderemos recuperá-los de forma eficiente.

3. DESENVOLVIMENTO DO SISTEMA

Este capítulo trata do processo de desenvolvimento escolhido para a criação da ferramenta. Posteriormente são apresentados os requisitos do sistema, as ferramentas utilizadas, a API e frontend da aplicação e por fim, uma API gerada como exemplo.

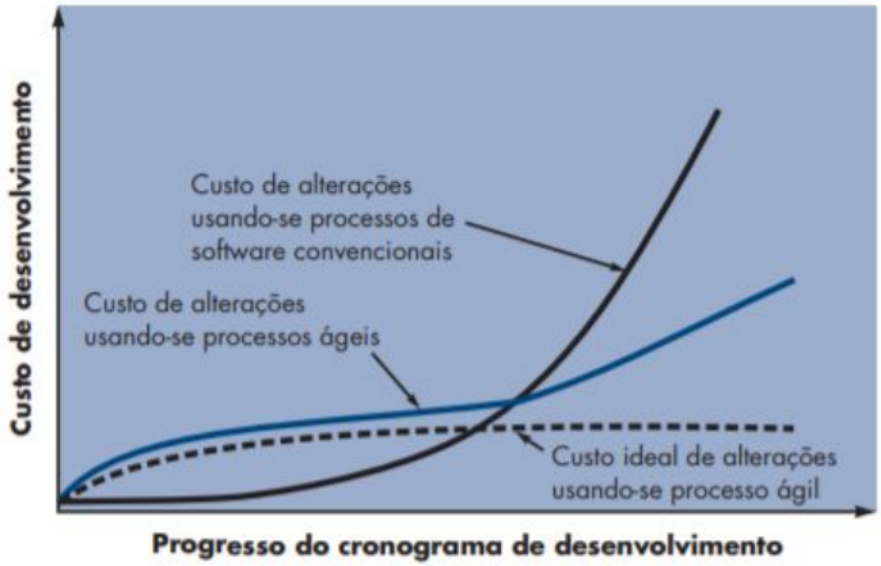
3.1. SCRUM SOLO

Durante as décadas de 1970 e 1980, diversos modelos de desenvolvimento surgiram e foram utilizados e melhorados, como o modelo em cascata, o modelo de desenvolvimento incremental e o modelo de métodos formais. Esses modelos de desenvolvimento foram muito utilizados até o início dos anos 2000, quando começaram a ser gradativamente sendo substituídos por métodos ágeis, que condizem melhor com o cenário atual de desenvolvimento, onde as equipes contam com poucos membros, os prazos de entrega são reduzidos e as especificações de desenvolvimento mudam constantemente.

> Nos dias de hoje, as empresas operam em um ambiente global, com mudanças rápidas. Assim, precisam responder a novas oportunidades e novos mercados, a mudanças nas condições econômicas e ao surgimento de produtos e serviços concorrentes. [...] muitas empresas estão dispostas a trocar a qualidade e o compromisso com requisitos do software por uma implantação mais rápida do software de que necessitam. (SOMMERVILLE, 2011, p.38).

Como observado na Figura 03, o uso de métodos ágeis torna as alterações realizadas em etapas finais menos custosas.

Figura 03: Custos de alterações como uma função do tempo em desenvolvimento



Fonte: PRESSMAN, 2013, p.83

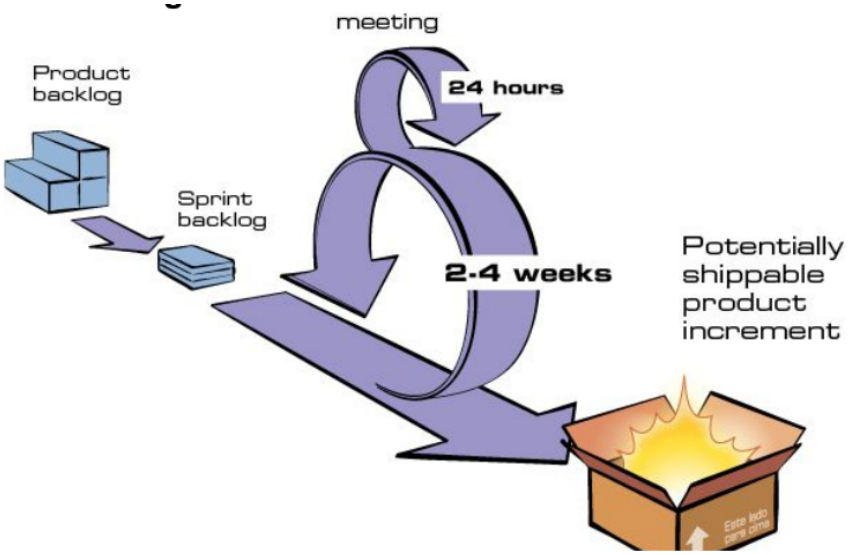
As metodologias de desenvolvimento ágil permitem uma maior fluidez no processo de desenvolvimento, e esse fator foi determinante na escolha de uma metodologia ágil para o projeto, mais especificamente o Personal Scrum , também chamado de Scrum Solo ou Scrum for One uma metodologia ágil baseada no Scrum, mas com modificações para o desenvolvimento com apenas um programador na equipe.

O Scrum é uma metodologia de desenvolvimento ágil desenvolvida nos anos 1990 por Jeff Sutterland, e que segue os princípios definidos pelo manifesto ágil (SOMMERVILLE, 2011). Existem duas etapas comuns ao Scrum padrão e o Scrum Solo que são fundamentais no processo de desenvolvimento deste método, os backlogs e os sprints.

Os backlogs constituem a fila de trabalhos a serem realizados, os itens dessa fila possuem graus de prioridade diferentes e ela pode ser atualizada a qualquer momento, tanto no intuito de adicionar novos trabalhos, quanto para alterar a prioridade deles. Já os sprints são as unidades de trabalho que serão gastas para realizar um backlog, no Scrum essas unidades são de curta duração (um dia, por exemplo), e diariamente são definidas quais atividades serão executadas no sprint.

Como observado na Figura 04, cada entrega de backlog pode ser um incremento ao produto já em funcionamento, permitindo que o sistema seja desenvolvido e entregue aos poucos, não necessitando que todos os módulos sejam finalizados para então ocorrer a entrega do produto.

Figura 04 Funcionamento do SCRUM



Fonte: Pagina Desenvolvimento Ágil 6

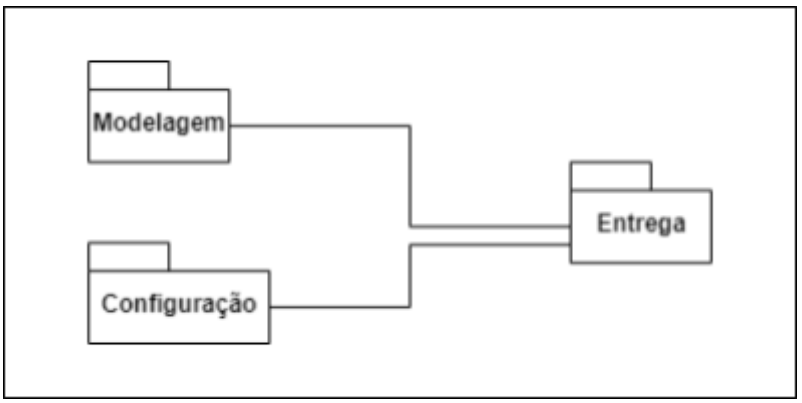
3.2. REQUISITOS DO SISTEMA

O sistema desenvolvido no presente trabalho tem como foco auxiliar programadores de API REST, criando um CRUD (acrônimo do inglês Create, Read, Update and Delete) completo, e com acesso ao código de fácil entendimento seguindo o padrão de projeto MVC(Model, View, Controller).

3.2.1. Requisitos funcionais

O sistema é dividido em três módulos principais: modelagem da API, configurações da API e gerar API. Como mostra o diagrama da Figura 05:

Figura 05: Módulos do sistema



Fonte: Elaborado pelo autor

- **Modelagem da API a ser gerada** - Neste módulo, é definido a estrutura base do sistema, que é formada por coleções, que por sua vez podem conter outras coleções ou atributos. É possível também definir o tipo dos atributos e controle de acesso de cada coleções. Este módulo possui as seguintes funcionalidades:
  - RF001 - Cadastrar coleção recursivamente
  - RF002 - Cadastrar regras de autenticação e autorização da coleção
  - RF003 - Editar coleção recursivamente
  - RF004 - Listar coleção recursivamente
  - RF005 - Cadastrar atributo recursivamente
  - RF006 - Editar atributo recursivamente
  - RF007 - Listar atributo recursivamente
- **Configuração da API a ser gerada -** Neste módulo é definido as configurações mais gerais da aplicação. Dentre elas, nome, porta e URL do banco de dados. Neste módulo é possível:
  - RF008 - Cadastrar nome da aplicação
  - RF009 - Cadastrar versão da aplicação
  - RF010 - Cadastrar salt key de criptografia da aplicação
  - RF011 - Cadastrar porta da aplicação
  - RF012 - Cadastrar URL do banco de dados da aplicação
  - RF013 - Cadastrar configuração de autenticação
  - RF014 - Cadastrar configuração de listagem de dados

- **Entrega da API gerada -** Neste módulo, o usuário irá enviar a estrutura que ele modelou para o servidor e terá como resposta um arquivo compactado (.zip) da API.
    - RF015 - Gerar código da API REST

3.2.2. Requisitos não-funcionais

Além dos requisitos funcionais, algumas características importante do sistema são:

- **Usabilidade**
    - NF001 - A interface deverá seguir padrões de design já estabelecidos no mercado com Bootstrap e Material-UI
- **Confiabilidade**
    - NF002 - O sistema deverá gerar uma API REST que funcione sem erros de execução
- **Funcionalidade**
    - NF003 - Eficiente, o sistema gerado deverá ser capaz de fazer CRUD de coleções

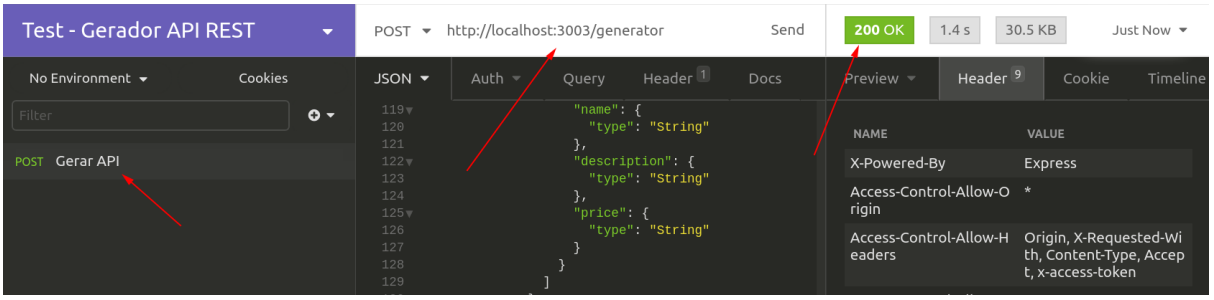
3.3. FERRAMENTAS E LINGUAGENS UTILIZADAS

Nesta seção é apresentada uma breve descrição das ferramentas e linguagens utilizadas no desenvolvimento deste trabalho.

3.3.1. Insomnia

O Insomnia é um aplicativo gratuito de plataforma cruzada para desktop que elimina a complexidade de interagir com API's baseadas em HTTP. O Insomnia combina uma interface fácil de usar com funcionalidades avançadas, como auxiliares de autenticação , geração de código e variáveis de ambiente . Também existe a opção de assinar um plano pago para obter acesso à sincronização de dados criptografados e colaboração em equipe (INSOMNIA, 2021). O insomnia foi utilizado para realizar os testes da API do gerador, como mostra a Figura 06.

Figura 06: Ferramenta Insomnia

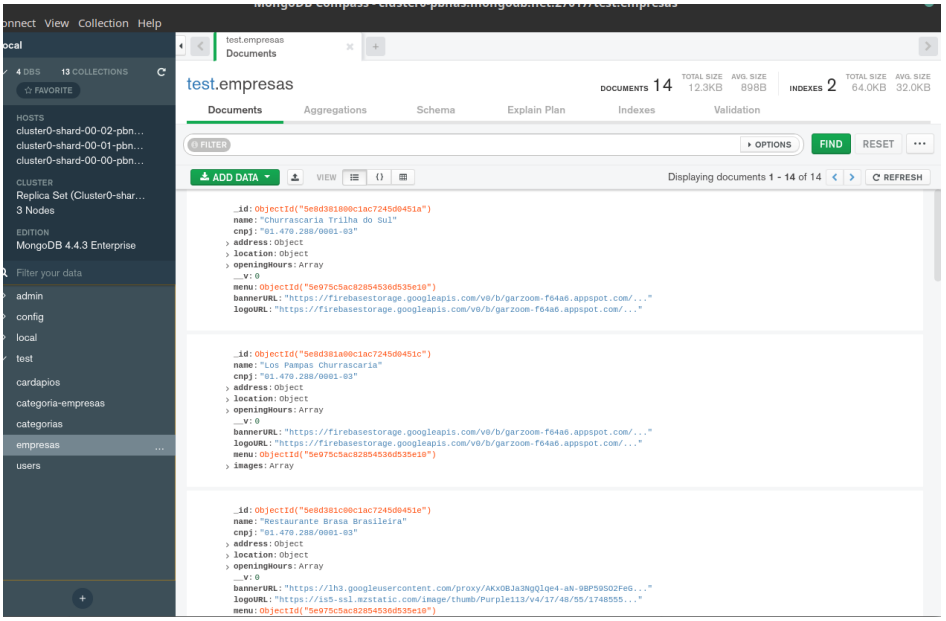


Fonte: Elaborado pelo autor

3.3.2. Mongo DB Compass

MongoDB Compass é a Graphical User Interface (GUI) do MongoDB. O Compass permite analisar e entender o conteúdo dos dados sem conhecimento formal da sintaxe de consulta do MongoDB. Além de explorar os dados em um ambiente visual, é possível usar o Compass para otimizar o desempenho da consulta, gerenciar índices e implementar a validação de documentos (MONGODB, 2021). O Mongo DB Compass foi utilizado para fazer os testes dos bancos de dados das APIs geradas, como mostra o exemplo da Figura 07.

Figura 07: Ferramenta Mongo DB Compass



Fonte: Elaborado pelo autor

3.3.3. Node JS

O Node.js se caracteriza como um ambiente de execução JavaScript. Com ele, o usuário pode criar aplicações sem depender do browser para isso. Com alta capacidade de escalabilidade, boa flexibilidade, arquitetura e baixo custo, torna-se uma ótima opção para programação (TOTVS, 2020). O Node.js foi utilizado no desenvolvimento do gerador de API e também a API que o sistema gerar tem todo o seu código em Node.js.

- **npm** - É o gerenciador de pacotes para Node.js . Ele foi criado em 2009 como um projeto de código aberto para ajudar os desenvolvedores de JavaScript a compartilhar facilmente módulos de código empacotados (NPM, 2021).
- **express** - É uma estrutura de aplicativo da web Node.js mínima e flexível que fornece um conjunto robusto de recursos para aplicativos da web e móveis. Com uma grande quantidade de métodos de utilitário HTTP e middleware à sua disposição, criar uma API robusta é rápido e fácil (EXPRESS, 2021).
- **mongoose** - Fornece uma solução direta e baseada em esquema para modelar os dados do seu aplicativo. Inclui conversão de tipo embutida, validação, construção de consulta, ganchos de lógica de negócios e muito mais, prontos para uso (MONGOOSE, 2021).
- **JWT** - JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma maneira compacta e independente para transmitir informações com segurança entre as partes como um objeto JSON. Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente (JWT, 2021).

Assim como o Node.js, npm e o express foram utilizados no desenvolvimento do gerador de API e também está inserido no código gerado pelo sistema.

3.4. API DO SISTEMA

O sistema disponibiliza uma API REST para gerar código. Essa API é utilizada pelo frontend do sistema, no entanto, pode ser usada separadamente sem a necessidade do frontend, basta apenas fazer uma chamada POST com a estrutura de dados que ela aceita. O objeto JSON enviado para o servidor é composto por outros três objetos, essa estrutura é mostrada abaixo:

- **Collections -** Neste objeto mostrado na Figura 08, é definido o nome da coleção e seus atributos. É recomendado colocar o nome da coleção no plural, porque se refere a uma coleção de objetos. No entanto, as classes de modelo dentro da API são no singular, então para aquelas coleções que o seu nome no singular é diferente do nome no plural, menos o "s" do final da palavra, é possível colocar um nome personalizado através do atributo "nameSingular".

Figura 08: Estrutura de dados - nome da coleção e descrição do atributos

```
{
    "collections":[
        {
            "name":"clientes",
            "nameSingular":"cliente",
            "attributes":{
                "nome":{
                    "type":"String",
                    "required":true
                },
                "cpf":{
                    "type":"String"
                },
                "telefone":{
                    "type":"String",
                    "required":true
                },
                "dataNascimento":{
                    "type":"Date"
                }
            }
        }
    ],
```

Fonte: Elaborado pelo autor

- **ConfigProject -** Neste objeto mostrado na Figura 09, são definidas as configurações da API. Com esse tipo de arquivo na API gerada concentramos as configurações em um só lugar, facilitando até mesmo a reutilização do código. Esses atributos são listados abaixo:
    - **nameProject** - nome do projeto
    - **versionApplication** - versão da API
    - **port** - porta que a API vai rodar no servidor
    - **saltKey** - string concatenada com o hash Message-Digest algorithm 5 (MD5) dos dados que precisam serem guardados em segurança
    - **tokenExpiresIn** - tempo de expiração de um token JWT
    - **expirationAux** - caso a expiração seja um valor fora do padrão JWT, usamos esse campo para definir um tempo em segundos
    - **headersNameToken** - nome do token JWT que são adicionados nas requisições
    - **urlDb** - URL de conexão com o banco de dados
    - **limit** - limite máximo de itens por paginação em listagem de dados

Figura 09: Estrutura de dados - configurações do projeto

```
23      ],
24      "configProject":{
25         "nameProject":"api_clientes",
26         "versionApplication":"1.0.0",
27         "port":"3000",
28         "saltKey":"mysaltkey",
29         "tokenExpiresIn":"1d",
30         "expirationAux":"1d",
31         "headersNameToken":"auth",
32         "urlDb":"localhost:27000",
33         "limit":"50"
34      },
```

Fonte: Elaborado pelo autor

- **Rules -** Neste objeto mostrado na Figura 10, é definido as configurações de autenticação da API. Com essas regras é possível definir o nível de acesso de uma determinada rota, podendo ter três comportamentos:
    - livre para qualquer usuário
    - somente usuários comuns e admin autenticados
    - somente admin

Figura 10: Estrutura de dados - regras de autenticação

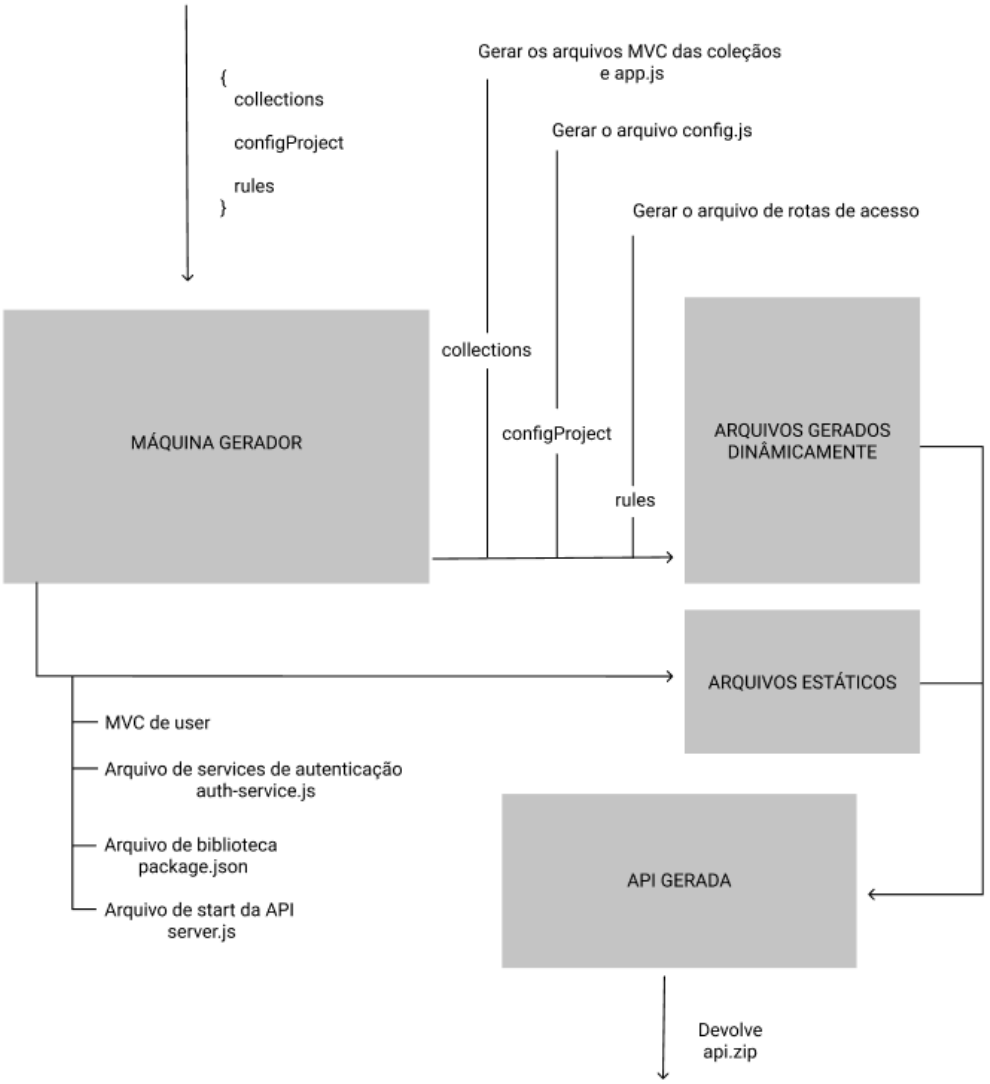
```
"rules":{
   "collections":[
      {
         "name":"clientes",
         "routes":[
            {
               "name":"get",
               "auth":{
                  "authorize":true,
                  "isAdmin":false
               }
            },
            {
               "name":"post",
               "auth":{
                  "authorize":true,
                  "isAdmin":false
               }
            },
            {
               "name":"put",
               "auth":{
                  "authorize":true,
                  "isAdmin":false
               }
            },
            {
               "name":"delete",
               "auth":{
                  "authorize":false,
                  "isAdmin":true
               }
            }
         ]
      }
```

Fonte: Elaborado pelo autor

Quando a API do gerador(máquina gerador) recebe uma requisição do tipo POST com o JSON , ela faz um processamento desses metadados e gera a API REST. Como acontece esse processamento é mostrado na Figura 11.

A estrutura da API gerada é formada pela composição de arquivos estáticos e arquivos dinâmicos. Os arquivos dinâmicos são gerados a partir dos metadados recebidos, já os arquivos estáticos são os mesmo para todas as API's.

Figura 11: Comportamento do gerador

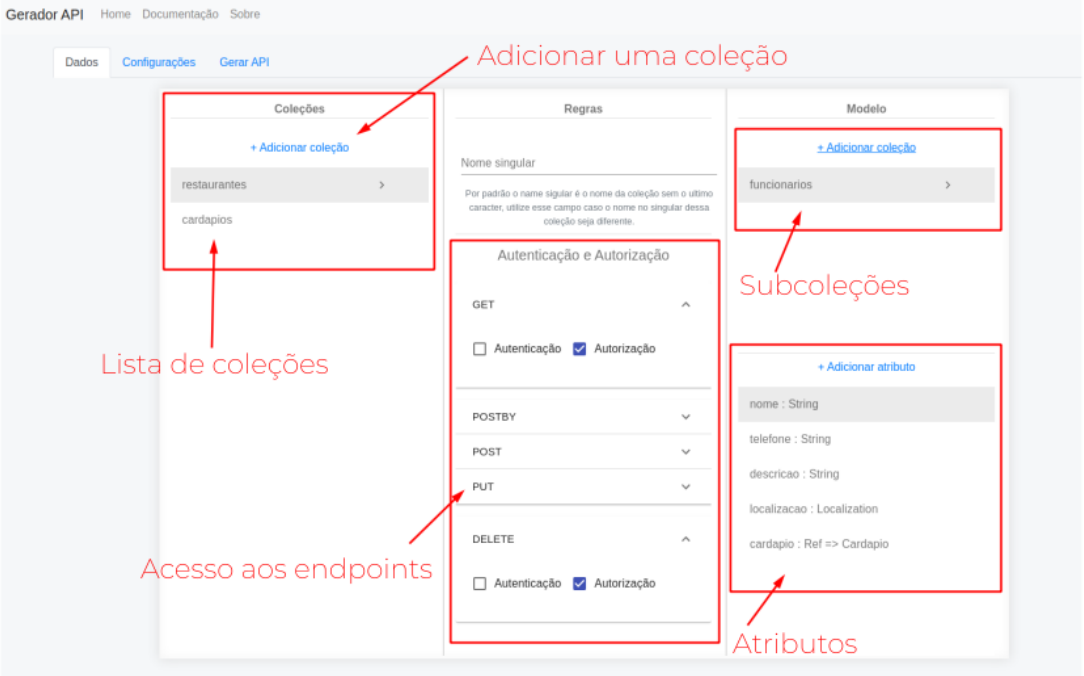


Fonte: Elaborado pelo autor

3.5. FRONTEND DO SISTEMA

Na primeira tela do sistema o usuário realiza a modelagem da API a ser gerada. Ele pode cadastrar as coleções, as subcoleções e os atributos de uma coleção. Nessa tela também é definido a questão de acessos aos endpoints da coleção. A tela inicial é demonstrada na Figura 12.

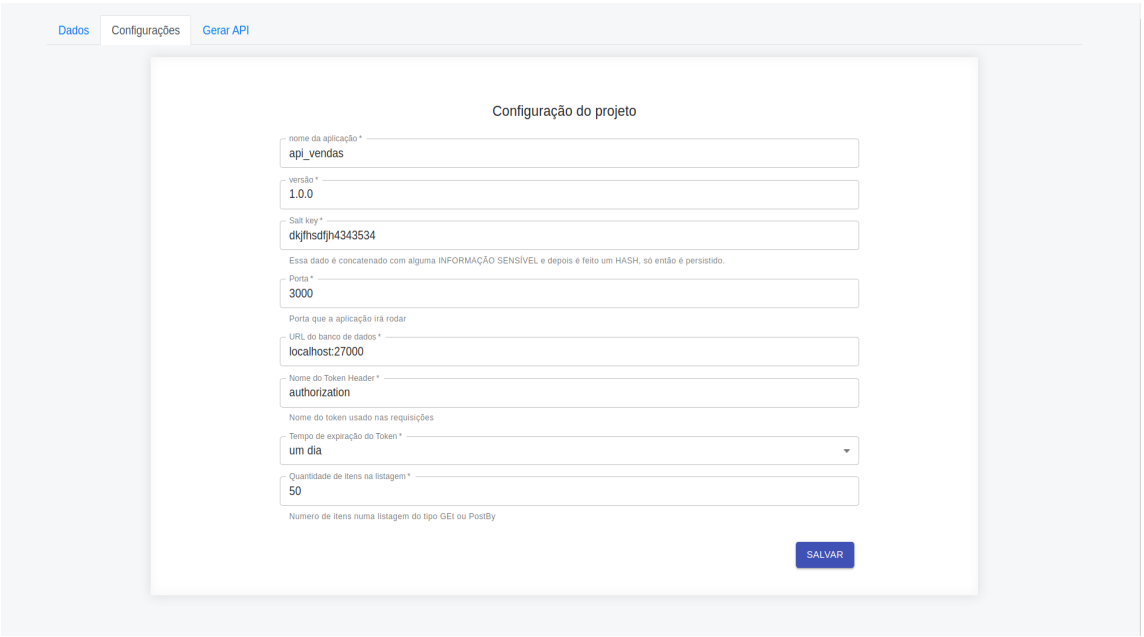
Figura 12: Tela de modelagem das coleções



Fonte: Elaborado pelo autor

Na segunda tela do sistema, o usuário realiza o cadastro das informações de configuração da API. Essas configurações são informações gerais que são utilizadas internamente na API gerada. A tela de configuração é demonstrada na Figura 13.

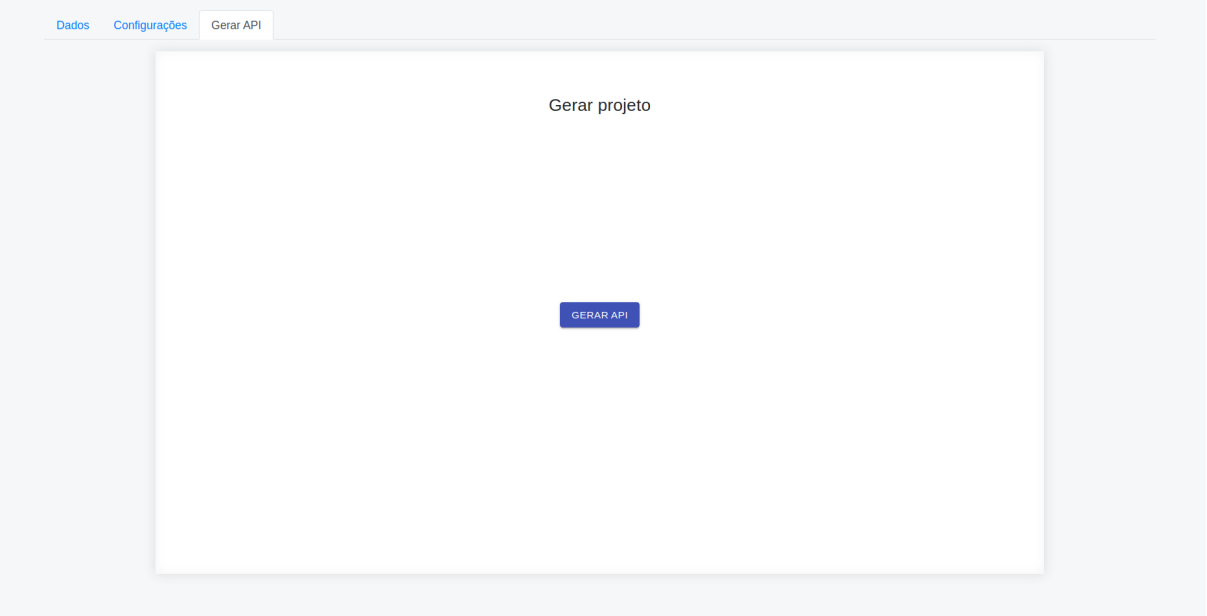
Figura 13: Tela de configuração da API



Fonte: Elaborado pelo autor

Na terceira e última tela do gerador, o usuário tem acesso a um botão para gerar a API. Após clicar no botão, os metadados por ele modelados na primeira e segunda tela são enviados ao servidor, que envia como resposta um arquivo .zip com a API gerada e o download é iniciado imediatamente pelo navegador. A tela de download da API é demonstrada na Figura 14.

Figura 14: Tela de gerar a API



Fonte: Elaborado pelo autor

3.6. API GERADA

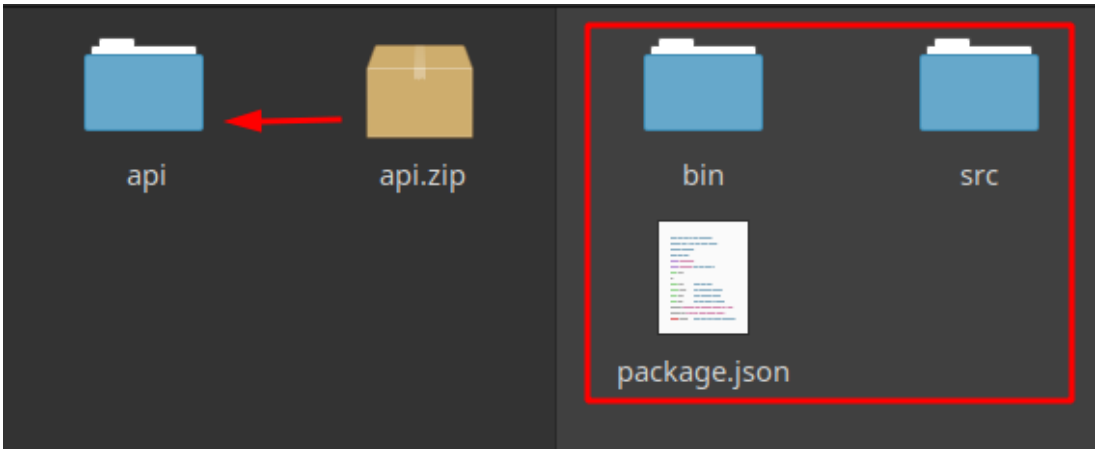
Nesta seção é mostrado os primeiros passos de configuração da API, e também é mostrado a sua estrutura de pasta e arquivos, isso é fundamental para futuras alterações na API.

3.6.1. Primeiros passos

Após gerar a API, é necessário executar alguns passos simples para colocá-la em produção. Como pré-requisito é obrigatório que o computador tenha instalado o nodeJS com a versão 8.0 ou superior e o npm com a versão 6.13.4 ou maior. O mongoDB só é necessário caso o banco de dados seja local.

O primeiro passo é descompactar os arquivos. A Figura 15 demonstra os arquivos descompactados.

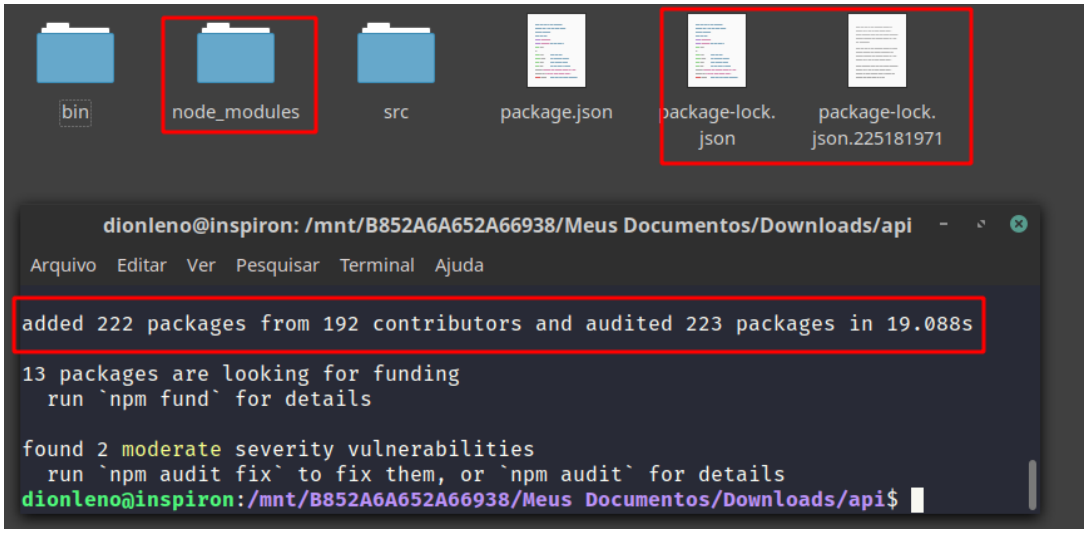
Figura 15: API gerada descompactada



Fonte: Elaborado pelo autor

A API usa bibliotecas de software livre externas ao node, como o express e o mongoose. Para baixar essas dependências é necessário executar o comando npm i, após a execução as bibliotecas são baixadas, como demonstra a Figura 16.

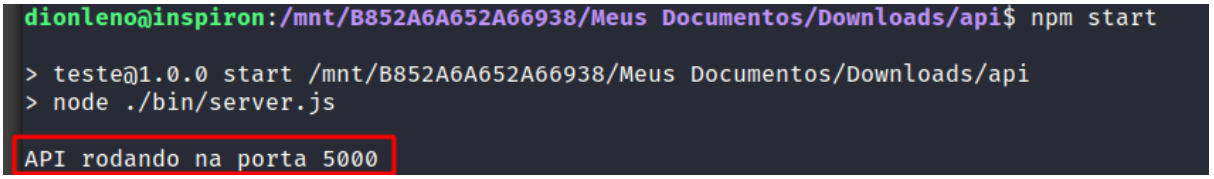
Figura 16: Instalação das dependências



Fonte: Elaborado pelo autor

Com as dependências instaladas, usamos o comando npm start para colocar a API disponível (Figura 17).

Figura 17: API disponível



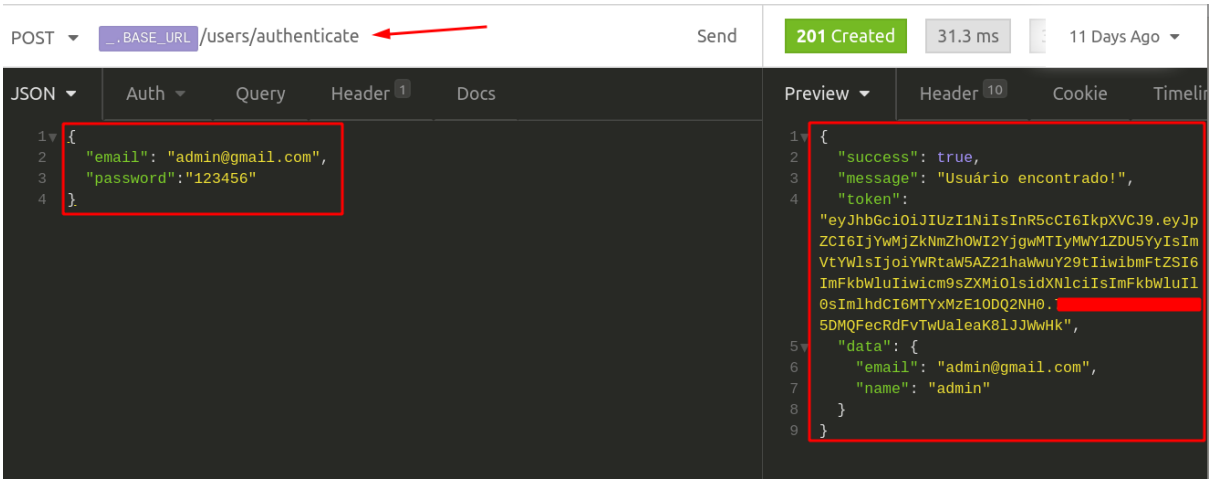
Fonte: Elaborado pelo autor

Caso tenha definido na modelagem rotas de acesso restritas a administradores, é preciso configurar no banco de dados esses usuários. O processo de configuração é simples, é necessário adicionar um elemento no array de rules chamado “admin” e gerar um novo token através da rota de autenticação como mostram as Figuras 18 e 19.

Figura 18: Atribuindo função de admin para usuário

Fonte: Elaborado pelo autor

Figura 19: Gerando token de admin



Fonte: Elaborado pelo autor

3.6.2. Estrutura de pastas e arquivos

Como a API gerada é pensada em suportar alterações e atualizações, a estrutura de diretórios é organizada para facilitar este trabalho. A Figura 20 mostra a estrutura de uma API de exemplo de CRUD de estabelecimentos.

Na pasta do projeto existem dois diretórios, bin e src. No bin fica apenas o arquivo inicial da API. No src os outros 5 diretórios:

1. **controllers** - arquivos de controller das coleções
2. **dao** - arquivos de comunicação com banco de dados
3. **models** - arquivos de modelos das coleções
4. **routes** - arquivos de rotas de cada coleção
5. **services** - arquivos de serviços

Figura 20: Exemplo de estrutura de diretórios

Fonte: Elaborado pelo autor

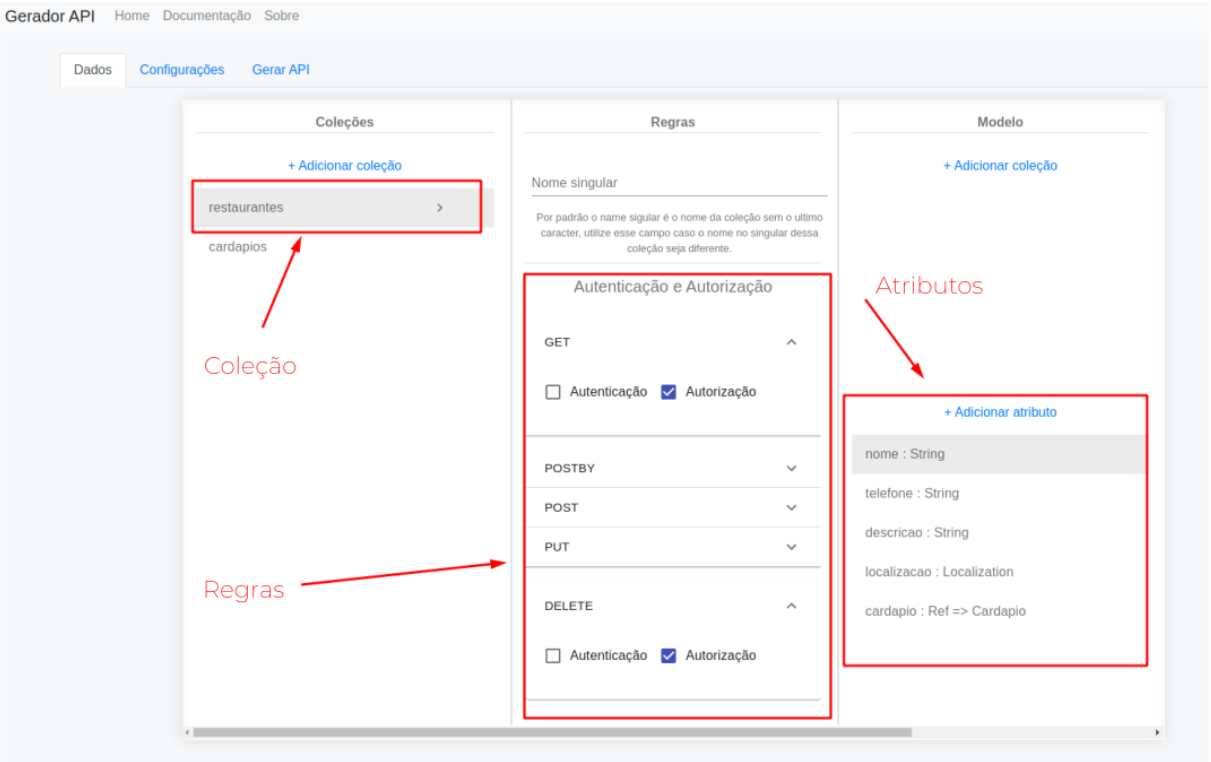
4. GERANDO E UTILIZANDO UMA API DE GUIA DE RESTAURANTE

Para validar o gerador JAPIS, foi utilizado um sistema de guia de restaurante que necessita de um CRUD de restaurantes e cardápios, também de duas funcionalidade específicas, são elas: gerar um arquivo .csv das coleções e um endpoint para listar os restaurantes ordenadamente por proximidade a partir de uma localização. Esse sistema foi escolhido devido a necessidade do CRUD das duas coleções, mas também pela necessidade das funcionalidades extras, que demonstra um dos objetivos que a facilidade de atualizações e incrementações na API gerada.

Para gerar uma API, é necessário acessar o frontend do gerador e em seguida começar a modelagem das coleções.

Na Figura 21 é demonstrada a modelagem da coleção de restaurantes, que têm suas rotas acessadas apenas administradores e tem como atributos nome, telefone, descrição, localização e um cardápio que é uma referência para um documento da coleção de cardápios.

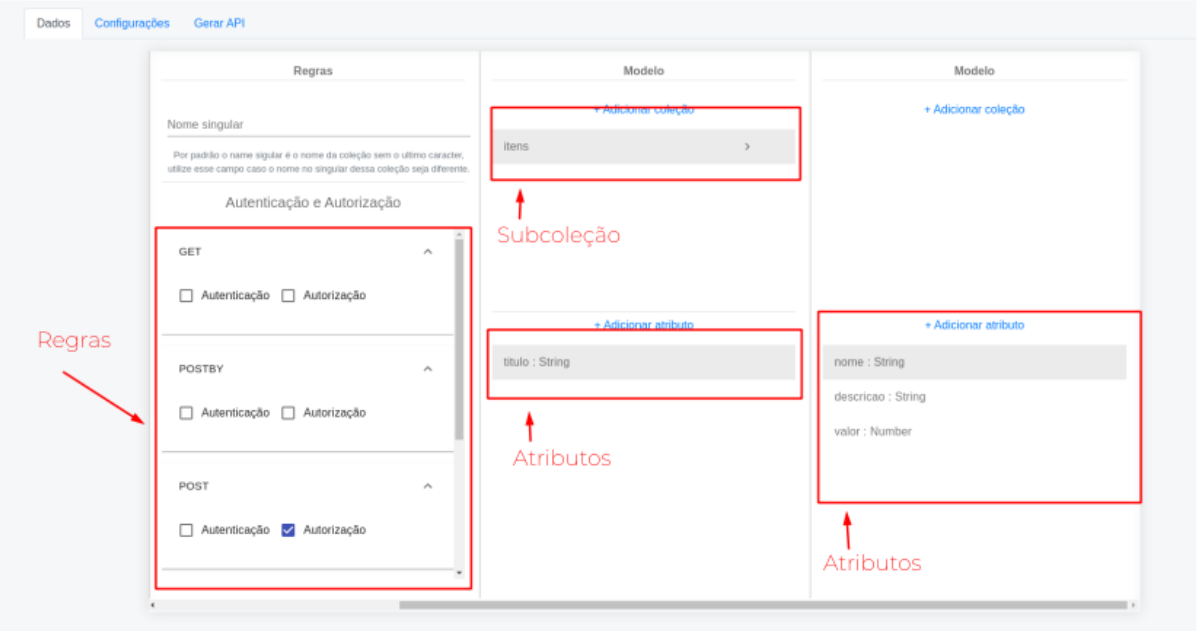
Figura 21: Modelagem de restaurante



Fonte: Elaborado pelo autor

Na Figura 22 é demonstrada a modelagem da coleção de cardápios:, que têm suas rotas de busca GET e POSTBY livre para acesso e as de manipulação dos dados POST, PUT e DELETE apenas para administradores. Tem como atributo título e também uma subcoleção de itens que por sua vez tem como atributos nome, descrição e valor.

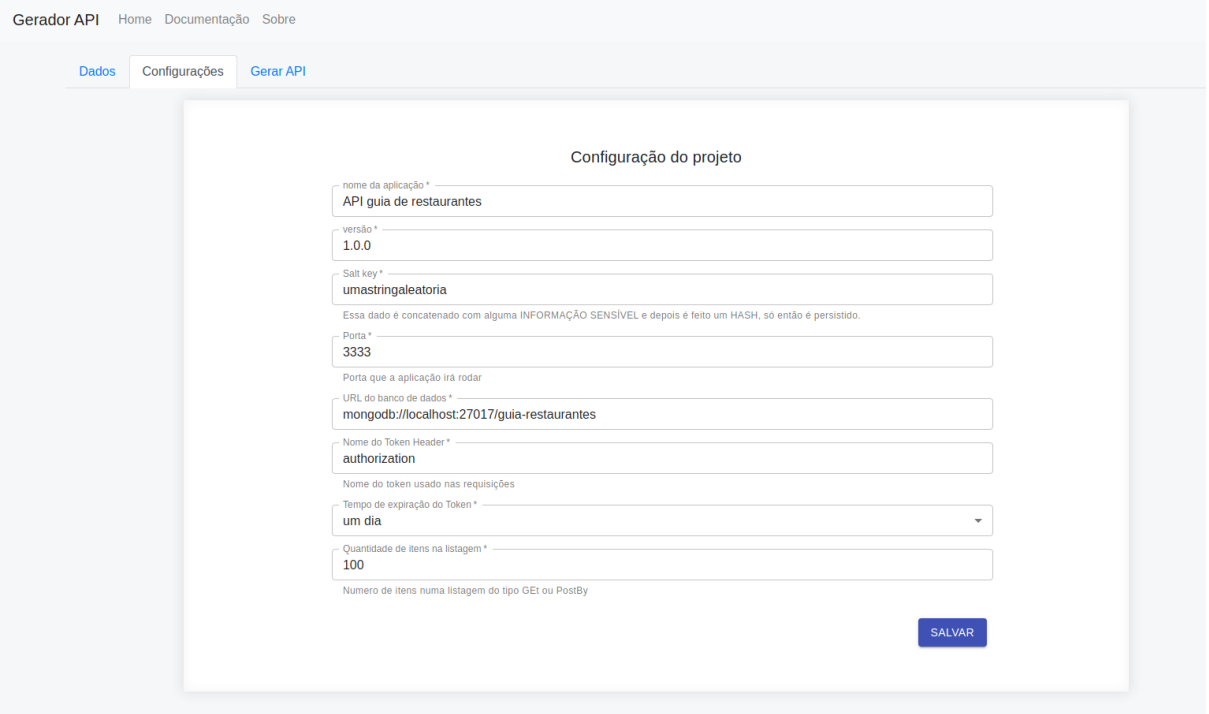
Figura 22: Modelagem de cardápio



Fonte: Elaborado pelo autor

Na Figura 23 é demonstrada as configurações gerais da aplicação, são elas: nome da aplicação, versão da API, salt key para o hash das informações sensíveis, porta em que o servidor irá rodar, URL do banco de dados, nome do token de autenticação e autorização, tempo de expiração do token e a quantidade de itens nas listagem.

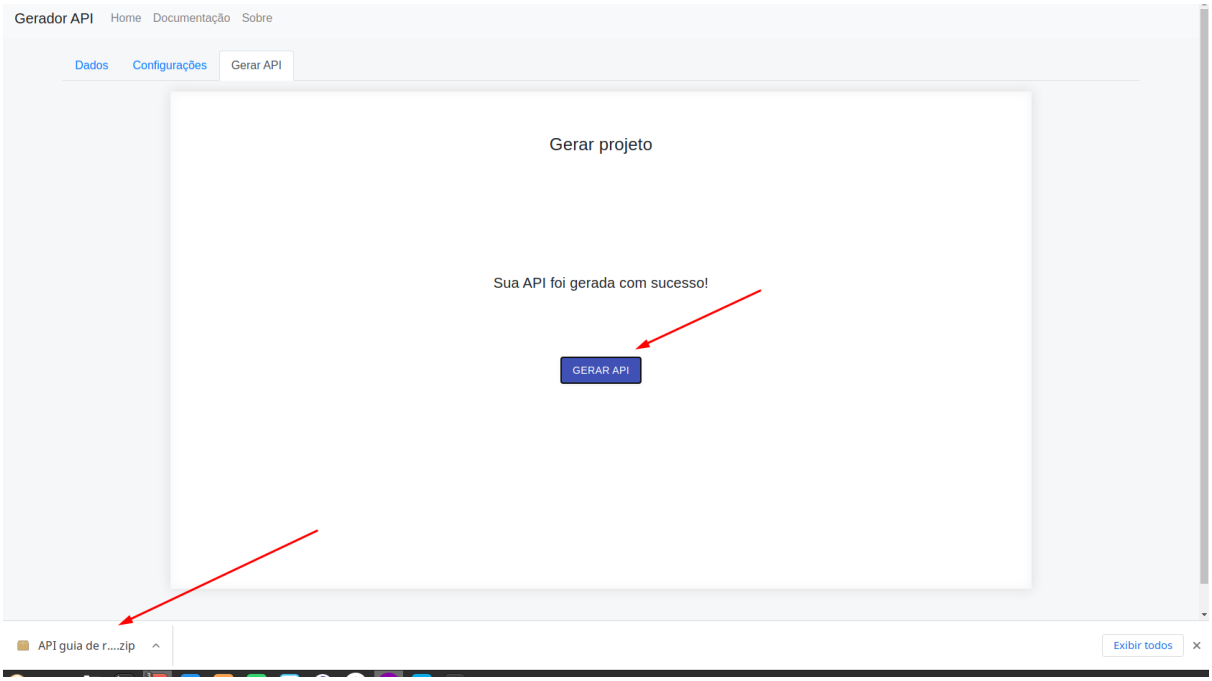
Figura 23: Configurações gerais



Fonte: Elaborado pelo autor

Depois de definida a modelagem das coleções e as configurações gerais, foi feita a chamada para gerar a API clicando no botão “GERAR API” e teve como resultado a API em formato zip, como mostra a Figura 24.

Figura 24: Gerando código



Fonte: Elaborado pelo autor

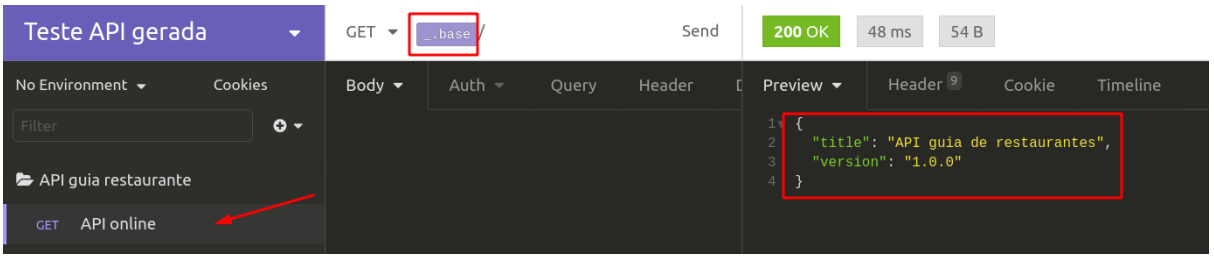
O código gerado está disponível no repositório publico do github e pode ser acessado pelo link: https://github.com/DionlenoPiata/api-guia-restaurantes.

4.1. ROTAS BASES E ROTAS DE USUÁRIOS

Com a ferramenta insomnia podemos usar uma rota base, que é a URL do domínio da API em produção ou ip e porta quando o servidor está funcionando localmente. Por padrão, a API gerada conta com uma rota base para verificar se a API está em funcionamento e as rotas de usuários para fazer cadastro e autenticação. Para esse exemplo é http://localhost:3333. Essas rotas são mostradas abaixo.

A primeira rota serve para verificar se a API está disponível, qual o título e a sua versão atual (Figura 25).

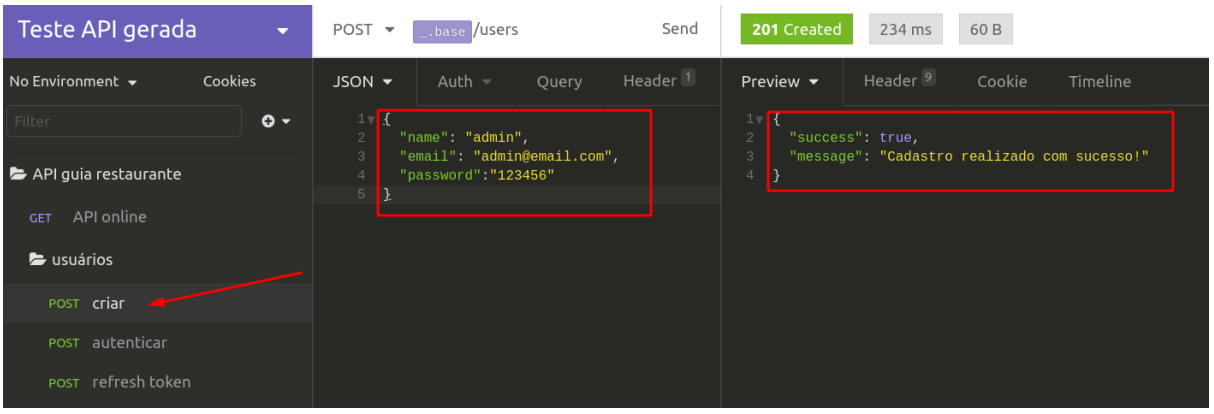
Figura 25: Verificando se API está funcionando



Fonte: Elaborado pelo autor

A segunda rota está relacionada com o usuário, e nela que os usuários da API são criados. O usuário básico da API necessita de um nome, email e uma senha. Caso haja a necessidade de mais atributos pode ser adicionado no código gerado no arquivo de modelo de usuários (Figura 26).

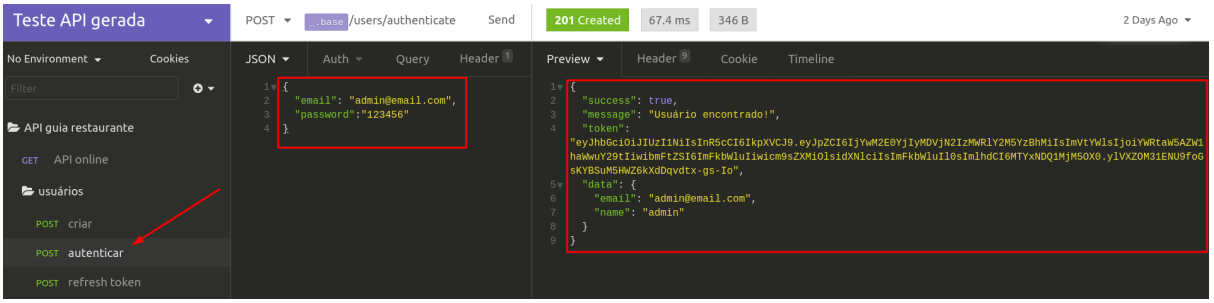
Figura 26: Criando um usuário



Fonte: Elaborado pelo autor

Logo após o cadastro é possível gerar um token JWT para acesso às rotas provadas da API gerada. Para gerar é necessaŕio informar o email e senha criados no cadastro (Figura 27).

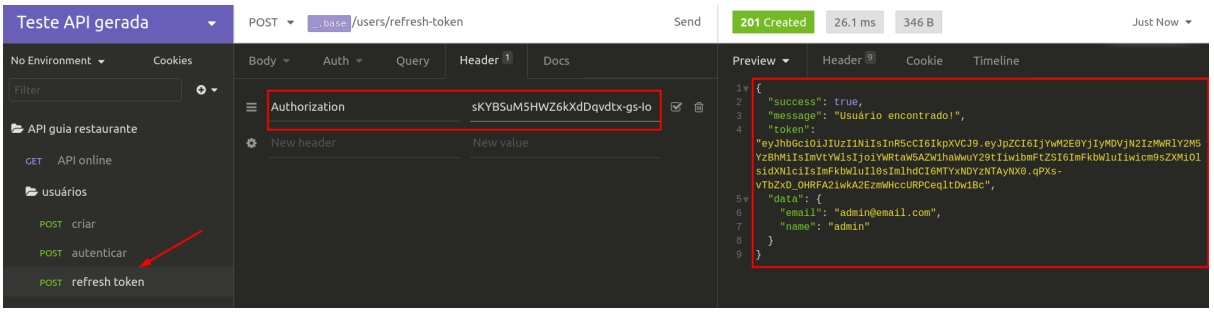
Figura 27: Gerando token de autenticação



Fonte: Elaborado pelo autor

É possível também renovar um token com um outro token válido, em caso que o tempo de expiração do token é pequeno essa pode ser uma boa opção para renovar a sessão (Figura 28).

Figura 28: Renovando o token de autenticação



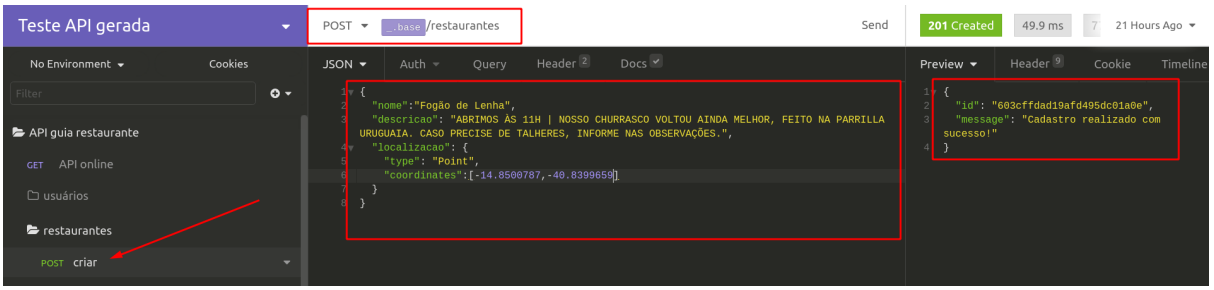
Fonte: Elaborado pelo autor

4.2. ROTAS DAS COLEÇÕES DE RESTAURANTES E CARDÁPIOS

A API gerada conta com os métodos de CRUD do padrão REST: GET, POST, PUT, DELETE e mais um outro método POST para uma pesquisa personalizada através da requisição POSTBY.

Para consumir a API do guia de restaurantes, inicialmente foram feitas requisições POST, para alimentar a base de dados. Foi feito três cadastros de restaurantes da cidade de Vitória da Conquista-BA (Figura 29).

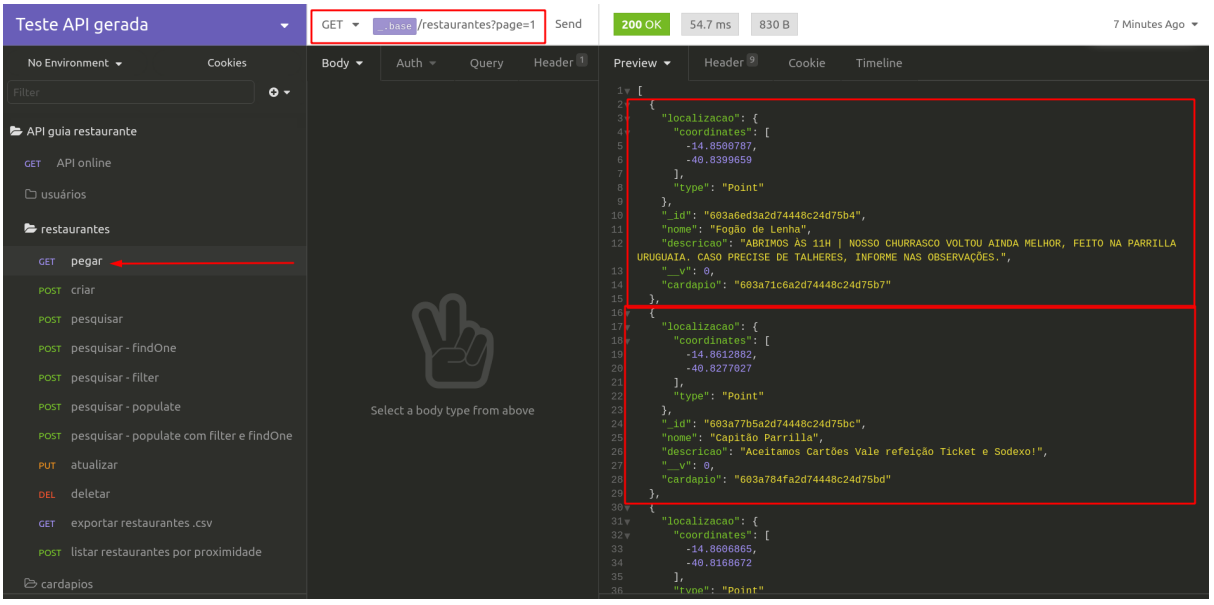
Figura 29: Cadastro de restaurante



Fonte: Elaborado pelo autor

Em seguida foi feita uma requisição do tipo GET, para pegar os restaurantes cadastrados. Neste tipo de requisição se não fizermos a implementação de uma paginação na API, ela irá retornar todos os dados de uma coleção, mas a API gerada conta com essa tratativa, quando geramos configuramos quantos itens ela deve retornar por página. Informamos na requisição qual página queremos buscar, se não for informada, por padrão é usado a página 1. Essa requisição é mostrada na Figura 30.

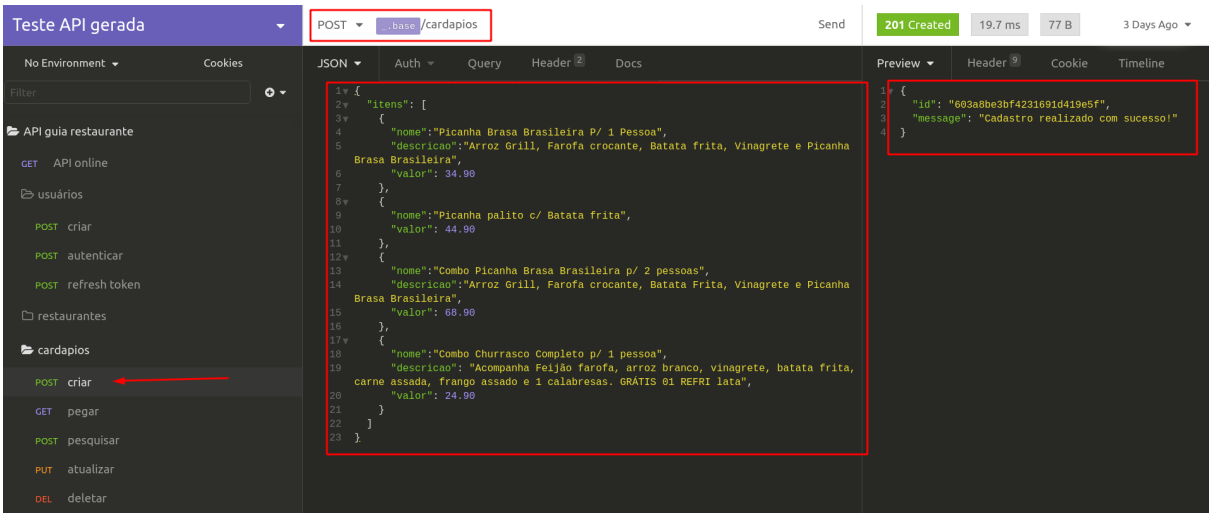
Figura 30: Listagem de restaurantes



Fonte: Elaborado pelo autor

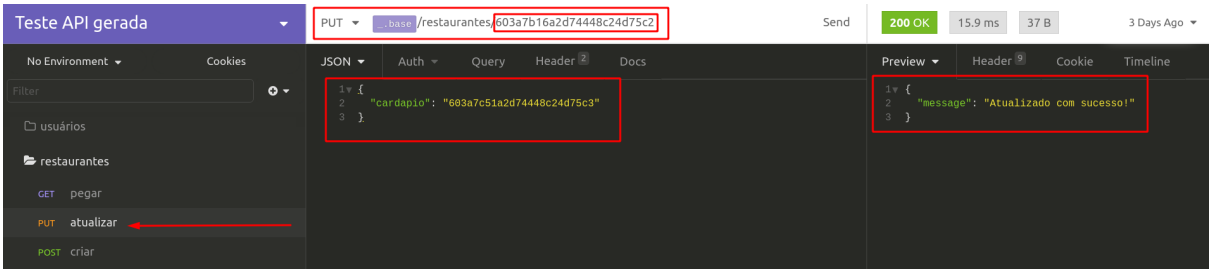
Para demonstrar o exemplo do PUT na coleção de restaurantes, foi feito um cadastro de cardápio e em seguida a atualização de um restaurante inserindo a referência para esse cardápio como mostram as Figuras 31 e 32.

Figura 31: Cadastro de cardápio



Fonte: Elaborado pelo autor

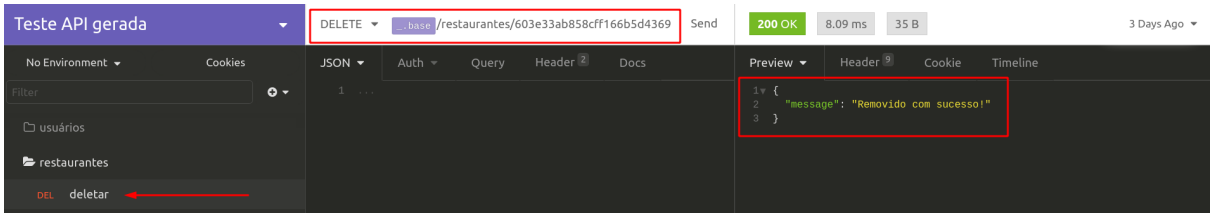
Figura 32: Atualização de restaurante



Fonte: Elaborado pelo autor

Em seguida foi feita a exclusão de um restaurante, demonstrada na Figura 33. Nesta requisição passamos o ID do objeto que é retornado no cadastro ou pode ser pego nas requisições GET.

Figura 33: Remoção de restaurante

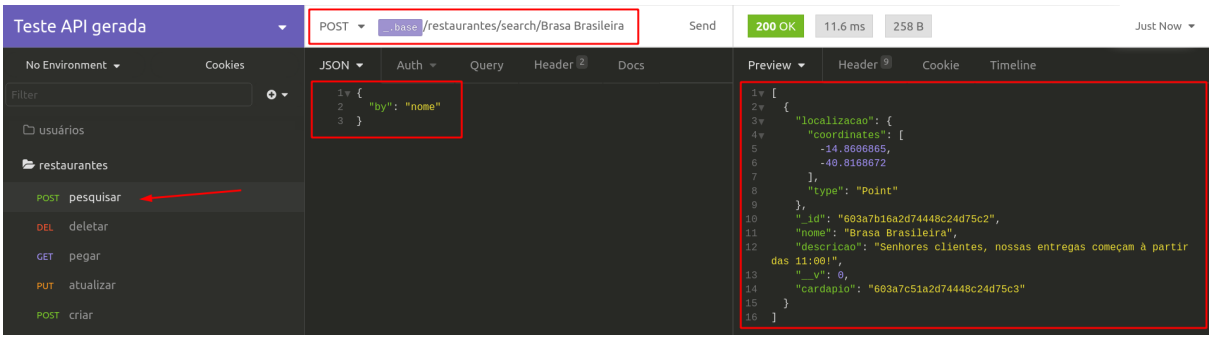


Fonte: Elaborado pelo autor

Para realizar buscas na nossa API, ela conta com uma rota “search”, com essa rota é possível buscar objetos por atributos, fazer filtros de retorno de objeto e também popular o retorno com objetos referenciados na coleção em que estamos buscando, essas requisições são mostradas e detalhadas abaixo.

Inicialmente foi feito uma pesquisa por nome de objeto, buscamos se existe algum restaurante com o nome “Brasa Brasileira”, e API retorna um array um elemento (Figura 34).

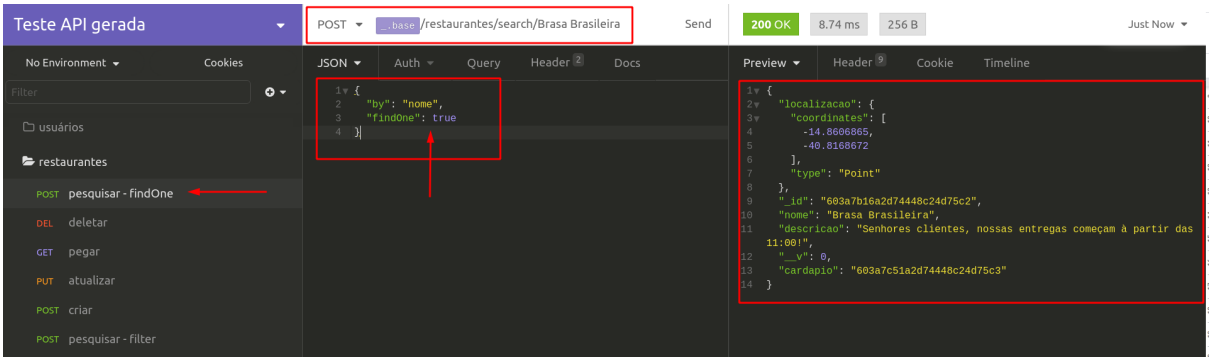
Figura 34: Pesquisa por nome



Fonte: Elaborado pelo autor

Em situações que temos a certeza que só existe um objeto com na busca por id, ou quando queremos pegar apenas o primeiro resultado podemos adicionar na requisição o atributo findOne, como mostra a Figura 35.

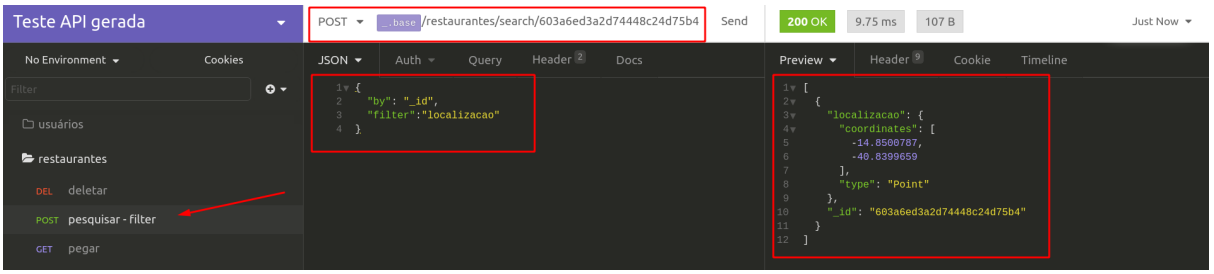
Figura 35: Pesquisa por nome retornando apenas um objeto



Fonte: Elaborado pelo autor

Em situações que não queremos o retorno de todo o objeto, podemos fazer um filtro pelos atributos. No exemplo da Figura 36 pegamos apenas a localização de objeto de determinado ID.

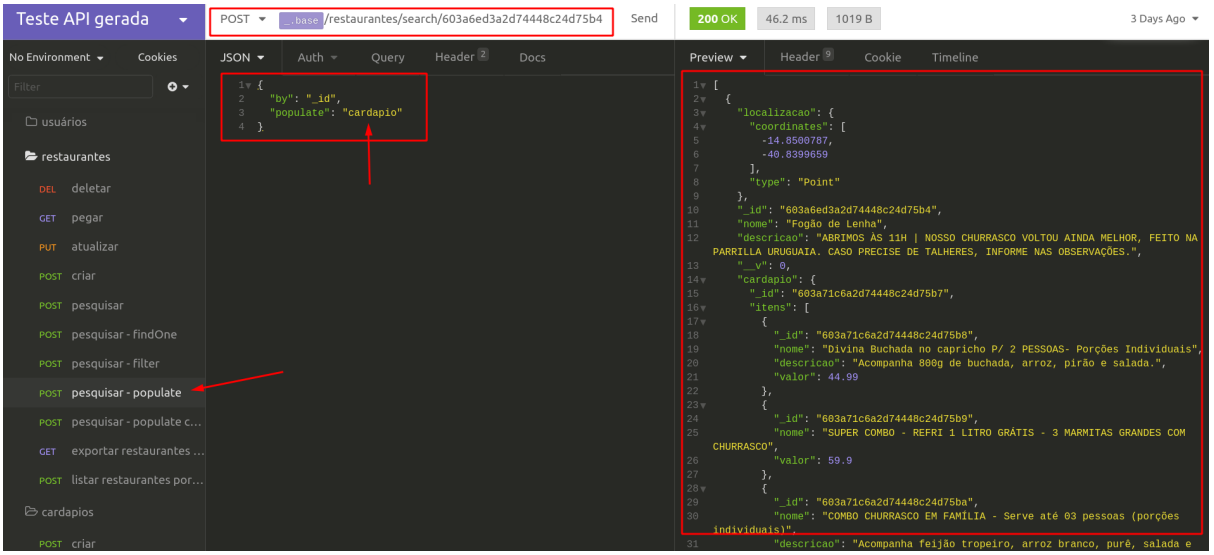
Figura 36: Pesquisa por id com com filter de localização



Fonte: Elaborado pelo autor

Em situações que temos no objeto o ID de outra coleção e queremos o objeto dessa outra coleção, como é o caso de cardápio em restaurante, não é necessário fazer duas requisições, uma para pegar o id do cardápio e outra para pegar o cardápio completo. Neste caso, passamos um atributo no corpo da requisição chamado populate e dizemos qual atributo ele irá popular. A Figura 37 mostra um exemplo.

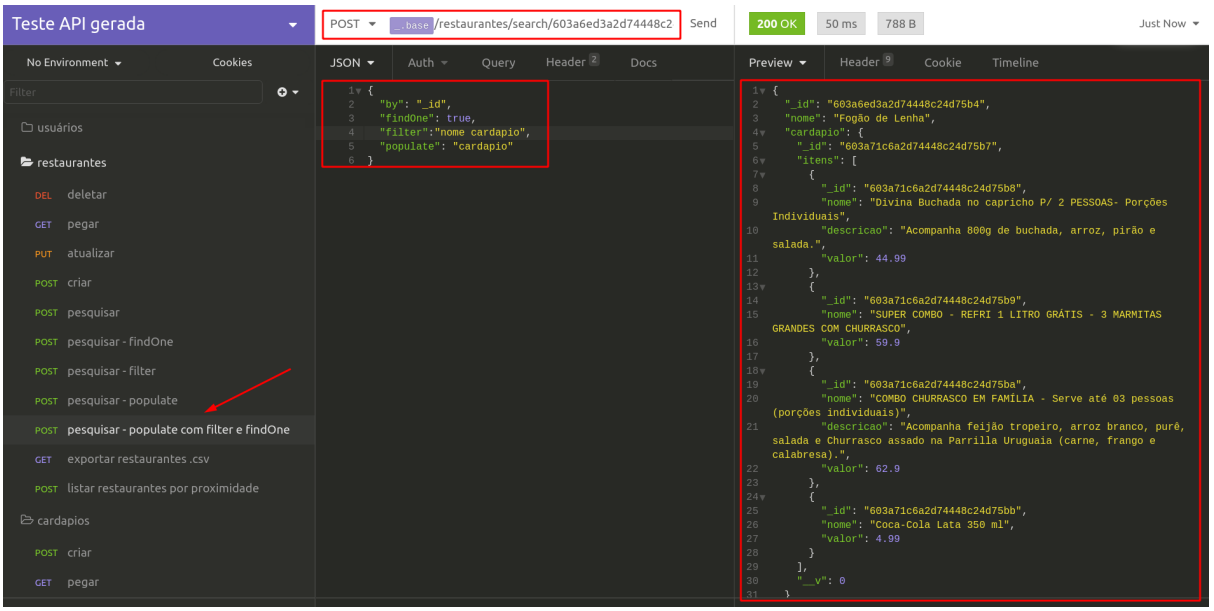
Figura 37: Pesquisa com populate de cardápio



Fonte: Elaborado pelo autor

O by, filter, findOne e o populate podem ser usados em conjunto, fazendo assim uma busca mais eficiente e limpa. A Figura 38 mostra um exemplo com todos eles.

Figura 38: Pesquisa com populate, filter e findOne



Fonte: Elaborado pelo autor

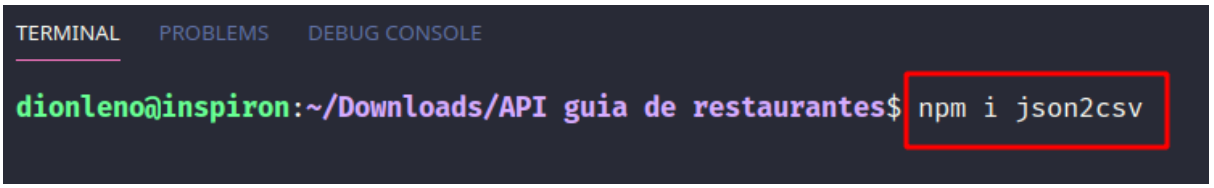
4.3. ADICIONANDO AS FUNCIONALIDADES DE EXPORTAR CSV E LISTAR PRÓXIMOS NA COLEÇÃO DE RESTAURANTES

A API gerada é bem estruturada e segue o padrão de projeto MVC, com isso ela facilita a evolução da API, podendo ser implementada novas funcionalidades. Nesse exemplo a API de guia de restaurantes tem como requisito a implementação de duas funcionalidade extras a API gerada, são elas: gerar um arquivo de exportação .csv das coleções de restaurantes e um endpoint que lista os restaurantes próximos de uma localização.

Inicialmente foi implementada a funcionalidade de exportar os arquivos .csv. As alterações dos arquivos são mostradas abaixo e podem ser encontradas neste commit do github https://github.com/DionlenoPiata/api-guia-restaurantes/commit/be917c14b53d2d2c52d028aac356c5c54e02d9a2.

O primeiro passo foi instalar a biblioteca json2csv para converter um objeto JSON em um arquivo CSV (Figura 39).

Figura 39: Instalação da biblioteca json2csv



Fonte: Elaborado pelo autor

O segundo passo foi alterar o arquivo src/controllers/restaurante-controller.js, para adicionar a funcionalidade. Foi importado o Parser da biblioteca json2csv e criando a função que faz e devolve o arquivo com csv (Figura 40).

Figura 40: Alteração do arquivo restaurante-controller.js

20 src/controllers/restaurante-controller.js

```
@@ -1,5 +1,6 @@
  "use strict";

+ const { Parser } = require("json2csv");
  const dao = require("../dao/restaurante-dao");

  exports.get = async (req, res, next) => {
@@ -80,3 +81,22 @@ exports.delete = async (req, res, next) => {
      });
    }
  };
+ exports.getCsv = async (req, res, next) => {
+   try {
+     let restaurantes = await dao.get();
+
+     const fields = ["_id", "nome", "telefone", "descricao", "localizacao"];
+
+     const json2csv = new Parser({ fields });
+     const csv = json2csv.parse(restaurantes);
+     res.header("Content-Type", "text/csv");
+     res.attachment("restaurantes.csv");
+     return res.status(200).send(csv);
+   } catch (e) {
+     console.log("error:", e);
+     res.status(500).send({
+       message: "Falha ao processar a requisição!",
+       error: e.message,
+     });
+   }
+ };
```

Fonte: Elaborado pelo autor

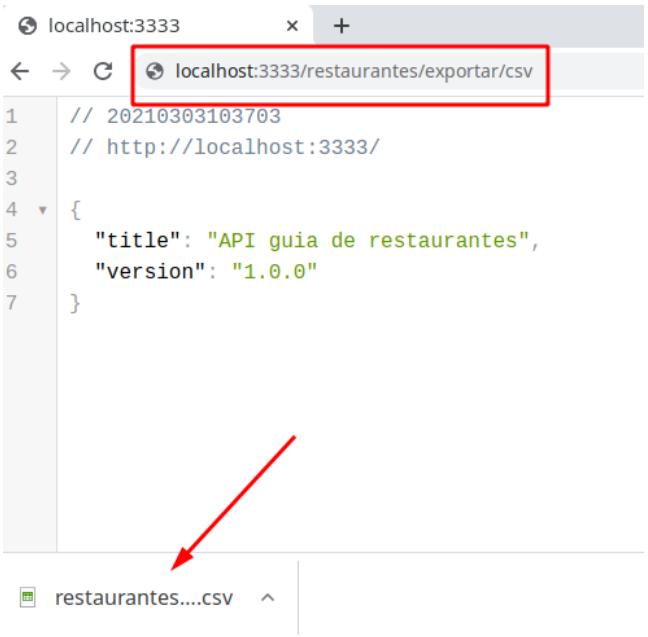
Por fim, foi alterado o arquivo src/routes/restaurante-route.js, e adicionamos um rota para exportar o arquivo (Figura 41).

Figura 41: Alteração do arquivo route.js

Fonte: Elaborado pelo autor

Com a funcionalidade implementada, foi feita uma chamada no navegador e o arquivo csv foi retornado (Figura 42).

Figura 42: Exportando arquivo csv



Fonte: Elaborado pelo autor

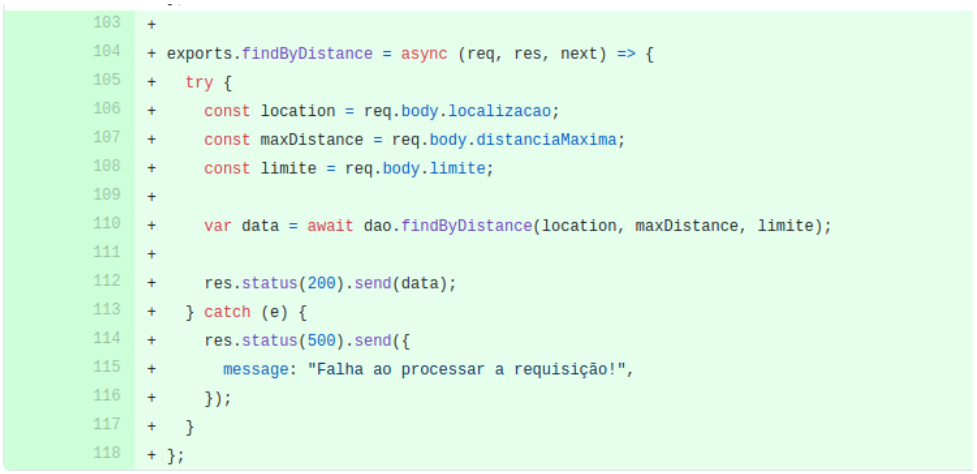
A segunda funcionalidade, listar os restaurantes próximos são mostradas abaixo e podem ser encontradas neste commit do github https://github.com/DionlenoPiata/api-guia-restaurantes/commit/6b5eddd05d17fbf023be46ebeca10789208e2219.

O primeiro passo foi adicionar a função findByDistance no arquivo src/controllers/restaurante-controller.js (Figura 43).

Essa função espera três variáveis:

1. Location - localização que vai buscar os próximos
2. MaxDistance - raio de abrangência em metros
3. Limite - limite de itens retornado

Figura 43: Alteração do arquivo restaurante-controller.js



```
+
+ exports.findByDistance = async (req, res, next) => {
+   try {
+     const location = req.body.localizacao;
+     const maxDistance = req.body.distanciaMaxima;
+     const limite = req.body.limite;
+
+     var data = await dao.findByDistance(location, maxDistance, limite);
+
+     res.status(200).send(data);
+   } catch (e) {
+     res.status(500).send({
+       message: "Falha ao processar a requisição!",
+     });
+   }
+ };
```

Fonte: Elaborado pelo autor

O segundo passo foi alterar o arquivo src/dao/restaurante-dao.js para realizar os cálculos com o banco de dados mongo (Figura 44).

Figura 44: Alteração do arquivo restaurante-dao.js

```
+
+ exports.findByDistance = async (location, maxDistance, limite) => {
+   const res = await Document.aggregate([
+     {
+       $geoNear: {
+         near: {
+           type: "Point",
+           coordinates: [location.latitude, location.longitude],
+         },
+         spherical: true,
+         key: "localizacao",
+         distanceField: "distancia",
+         maxDistance: maxDistance,
+       },
+     },
+     { $limit: limite },
+   ]);
+
+   return res;
+ };
```

Fonte: Elaborado pelo autor

O terceiro passo foi alterar o arquivo src/models/restaurante.js para que o modelo da coleção de restaurantes suporte o 2dsphere do mongo (Figura 45).

Figura 45: Alteração do arquivo restaurante.js

```
28  28
    29  + schema.index({ localizacao: "2dsphere" });
    30  +
```

Fonte: Elaborado pelo autor

Por fim foi alterado o arquivo src/routes/restaurante-route.js, e adicionamos um rota para exportar o arquivo (Figura 46).

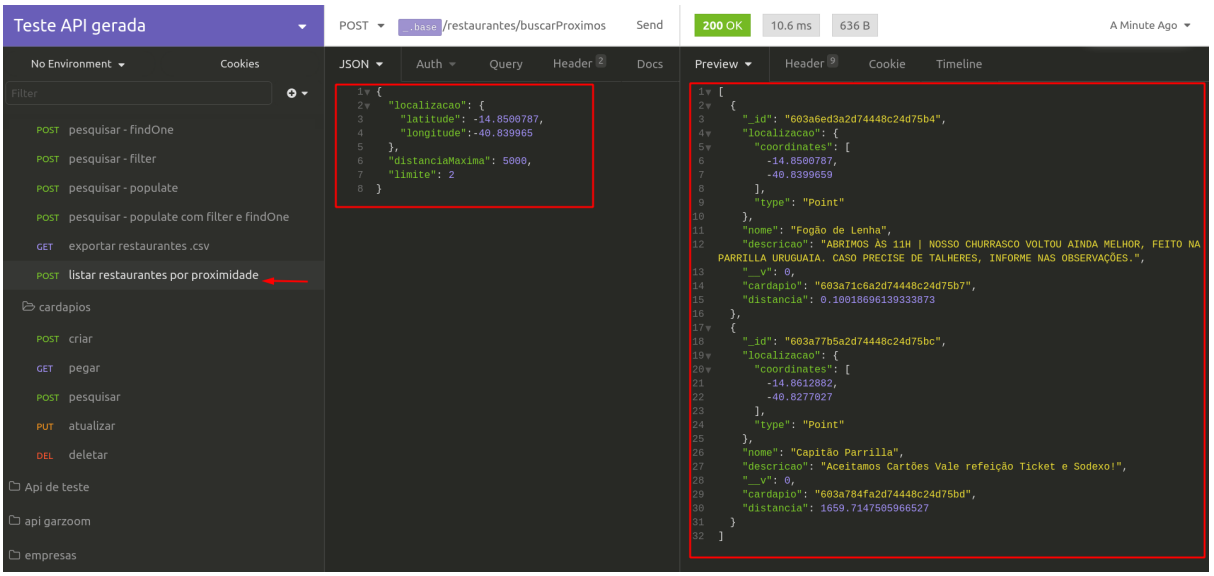
Figura 46: Alteração do arquivo restaurante-route.js

```
@@ -11,6 +11,11 @@ router.get("/exportar/csv", authService.isAdmin, controller.getCsv);
11  11  router.post("/search/", authService.authorize, controller.get);
12  12  router.post("/search/:by", authService.authorize, controller.postBy);
13  13  router.post("/", authService.isAdmin, controller.post);
    14  + router.post(
    15  +   "/buscarProximos",
    16  +   authService.authorize,
    17  +   controller.findByDistance
    18  + );
14  19  router.put("/:id", authService.isAdmin, controller.put);
15  20  router.delete("/:id", authService.isAdmin, controller.delete);
16  21
```

Fonte: Elaborado pelo autor

Com a funcionalidade implementada, foi feita uma requisição com uma localização e com um raio de 5km no insomnia para a listagem. Como é mostrado na Figura 47.

Figura 47: Listando restaurantes próximos



Fonte: Elaborado pelo autor

4.4. ANÁLISE DOS RESULTADOS

Disponibilizar uma API para integração, seja ela web, mobile ou desktop é indispensável na grande maioria das empresas que envolve TI atualmente, no entanto, profissionais de backend são caros, por isso muitas empresas usam os BaaS, suas facilidades de modelagem, ganho tempo e custo são atrativos interessantes, mas isso cria uma dependência do usuário a plataforma. O JAPIS une os dois mundos, pois disponibiliza uma interface para a modelagem como os Baas, e gera um código legível, seguindo padrões de projetos como os programadores backend, que é disponibilizado ao usuário.

Durante o desenvolvimento do sistemas, algumas dificuldades foram encontradas, a principal delas foi a parte de mapeamento dos documentos para gerá-los dinamicamente através dos metadados.

O JAPIS se mostrou eficiente na geração de código. Foi gerado uma API de Guia de restaurantes que tinha como requisitos CRUD de restaurantes e cardápios, mas precisava de inserir as funcionalidades de gerar um arquivos .csv e poder listar restaurantes próximos, devido a estrutura da API gerada, foi simples a adição das novas funcionalidades.

5. CONCLUSÃO E TRABALHOS FUTUROS

Os cientistas estão sempre em busca de padrão. Sempre que um padrão é encontrado um conhecimento pode ser gerado e assim evoluirmos em algum sentido. Através de uma análise do padrão MVC e DAO usado na construção de API's com o node JS, foi possível encontrar um padrão, com isso criar o JAPIS, um gerador de código de API Restful que permite gerar automaticamente CRUD para aplicações web service REST.

O JAPIS possibilita que a geração de API's seja realizada em um período de tempo mais curto, com baixo custo para ser implementado, facilidade na manutenção e segurança das informações trafegadas.

Para a validação do sistema foi gerado uma API de guia de restaurantes, foi feita a instalação das dependências e a API foi colocada disponível sem erros, foi feito também o consumo da API, inserindo e fazendo consultas dos dados.

A construção deste trabalho perpassou pelo conhecimento adquirido em uma série de disciplinas da grade curricular do curso, desde Algoritmos e programação I e II abordada no primeiro e segundo semestre, até Desenvolvimento de Software no oitavo semestre, além de outras que são cerne do trabalho, como Engenharia de Software, Desenvolvimento de Sistemas Web, Compiladores e Teste de Software. Sem os conhecimentos teóricos adquiridos durante o curso, assim como outros práticos desenvolvidos em projetos acadêmicos e profissionais não teria sido possível construir este trabalho.

Durante o desenvolvimento as maiores dificuldades encontradas foram encontrar um padrão de estrutura de projeto que fosse dinâmico e que facilitasse gerar os arquivos de forma organizada, mapear os documentos e fazer a formação dos documentos gerados.

Apesar das dificuldades encontradas, os objetivos gerais e específicos definidos no capítulo 1 foram atingidos, o gerador de API foi desenvolvido. Foram realizados testes no gerador e uma API de exemplo foi gerada e consumida.

Como sugestão de trabalhos futuros, é recomendado a realização de testes com os usuários finais, hospedar o sistema em um provedor da internet. O gerador pode ter melhorias nas API geradas também como suportar banco de dados relacional, armazenar arquivos como um file storage, variáveis de ambiente e gerar documentação automaticamente.

6. REFERÊNCIAS

BATSCHINSKI, George. **Backend as a Service – O que é?**. [*S. l.*], 25 jul. 2019. Disponível em: https://blog.back4app.com/2019/07/25/backend-as-a-service/. Acesso em: 19 out. 2019.

EXPRESS. **Fast, unopinionated, minimalist web framework for Node.js**. [*S. l.*], 2021. Disponível em: http://expressjs.com/. Acesso em: 10 fev. 2021.

GIL, Antonio Carlos. **Como Elaborar Projetos de Pesquisa**. 4. ed. [*S. l.*: *s. n.*], 2002.

GRAPHQL. **Uma linguagem de consulta para sua API.** [*S. l.*], 2019. Disponível em: https://graphql.org/. Acesso em: 3 dez. 2019.

IANNI, V. **"Introdução aos bancos de dados NoSQL"**. Disponível em: http://www.devmedia.com.br/introducao-aos-bancos-de-dados-nosql/26044. Acesso em 15/10/2019.

INSOMNIA. **Getting Started with Insomnia**. [*S. l.*], 7 jan. 2021. Disponível em: https://support.insomnia.rest/article/11-getting-started. Acesso em: 7 fev. 2021.

JWT. **Introduction to JSON Web Tokens**. [*S. l.*], 2021. Disponível em: https://jwt.io/introduction. Acesso em: 10 fev. 2021.

KAUARK, F. et al. **Metodologia de pesquisa: guia prático.** Itabuna: Via Litterarum, 2010.

LÓSCIO, Bernadette Farias; OLIVEIRA, Hélio Rodrigues; PONTES, Jonas César. **NoSQL no desenvolvimento de aplicações Web colaborativas. Researchgate**, [*s. l.*], 2011. Disponível em: https://www.researchgate.net/profile/Bernadette_Loscio/publication/268201466_NoSQL_no_desenvolvimento_de_aplicacoes_Web_colaborativas/links/576aa72008aef2a864d1ef8c/NoSQL-no-desenvolvimento-de-aplicacoes-Web-colaborativas.pdf. Acesso em: 18 fev. 2021.

MONGODB Compass: **O que é MongoDB Compass ?**. [*S. l.*], 2021. Disponível em:

https://docs.mongodb.com/compass/current/. Acesso em: 23 fev. 2021.

MONGOOSE. **Elegant mongodb object modeling for node.js**. [*S. l.*], 2021. Disponível em: https://mongoosejs.com/. Acesso em: 10 fev. 2021.

MORO, Tharcis D.; DORNELES, C.; REBONATTO, M. T. **Web services WS-* versus Web Services REST.** Instituto de Ciências Exatas e Geociências, Universidade de Passo Fundo (UPF), 2011.

NPM. **About npm**. [*S. l.*], 2021. Disponível em: https://www.npmjs.com/about. Acesso em: 10 fev. 2021.

OLIVEIRA, Paulo Henrique Cardoso. **Desenvolvimento de um gerador de API REST seguindo os principais padrões da arquitetura.** [S. l.: s. n.], 2014. Disponível em: https://aberto.univem.edu.br/handle/11077/1008. Acesso em: 24 ago. 2019.

OLIVEIRA, William. **Protocolo HTTP**. [*S. l.*], 2015. Disponível em: https://woliveiras.com.br/posts/protocolo-http/. Acesso em: 3 dez. 2019.

PAGOTTO, Tiago. Scrum Solo: **Processo de software para desenvolvimento individual.** *In*: MAGNO, Alexandre. **Scrum Solo**. [*S. l.*], 2016. Disponível em: https://engenhariasoftware.files.wordpress.com/2016/04/scrum-solo.pdf. Acesso em: 31 out. 2019.

PEREIRA, Felipe S. **Utilização de Banco de Dados NoSql em Ambientes Corporativos. Databases**, [*s. l.*], 2013. Disponível em: https://studylibpt.com/doc/5855636/utiliza%C3%A7%C3%A3o-de-banco-de-dados-nosql-em-ambientes-corporativos. Acesso em: 15 out. 2019.

POLO, Gabriel. **Protocolo SOAP**. [*S. l.*], 8 mar. 2018. Disponível em: https://medium.com/@gabrielpolo/protocolo-soap-e21566caf06f. Acesso em: 8 fev. 2021.

PRESSMAN, Roger S. **Engenharia de Software - Uma Abordagem Profissional.** Porto Alegre: AMGH, 2011.

REDHAT. **O que é uma API?. *In*: O que é uma API?**. [*S. l.*], 2019. Disponível em: https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces. Acesso em: 19 set. 2019.

FIELDING, R. T., "**Architectural styles and the design of network-based software architectures**," Ph.D. dissertation, Inf. Comput. Sci., Univ.California, Irvine, CA, USA, 2000.

SELLTIZ, Claire; [et al.]. **Métodos de pesquisa nas relações sociais.** São Paulo: Ed. Herder, 1967.

SOARES, B. E.. **Uma avaliação experimental de desempenho entre Sistemas Gerenciadores de Banco de Dados Colunares e Relacionais.** Cascavel, 2012. 102p. Trabalho de conclusão de curso (Graduação em Bacharelado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná – UNOPA, Cascavel, PA. Disponível em: < 60http://www.inf.unioeste.br/~tcc/2012/TCC_Bruno.pdf >. Acesso em: 06 de Janeiro de 2014, às 12h.

SOMMERVILLE, Ian. **Engenharia de software.** 9 ed. São Paulo: Pearson, 2011.

TANENBAUM, A. S. – **Redes de Computadores – 4ª Ed**., Editora Campus (Elsevier), 2003.

W3C. **Simple Object Access Protocol (SOAP)** 1.1: W3C Note 08 May 2000. Disponível em: < http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>. Acesso em: 1 nov. 2019.

TOTVS. **Node.js: O que é, quais as características e vantagens?**. [*S. l.*], 27 mar. 2020. Disponível em: https://www.totvs.com/blog/developers/node-js/. Acesso em: 10 fev. 2021.