UNIVERSIDADE FEDERAL DO CEARÁ
INSTITUTO UFC VIRTUAL
CURSO DE SISTEMAS E MÍDIAS DIGITAIS

GUILHERME CÂNDIDO PRETTO DE OLIVEIRA

GENNAI API: DESENVOLVIMENTO DE UMA API COM A TEMÁTICA DE DIGIMON UTILIZANDO GRAPHQL

FORTALEZA
2022

GUILHERME CÂNDIDO PRETTO DE OLIVEIRA

GENNAI API: DESENVOLVIMENTO DE UMA API COM A TEMÁTICA DE DIGIMON UTILIZANDO GRAPHQL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto UFC Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Orientador: Prof. Dr. Gilvan Maia

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

O47g Oliveira, Guilherme Cândido Pretto de.
Gennai API: Desenvolvimento de uma API com a temática de Digimon utilizando GraphQL / Guilherme Cândido Pretto de Oliveira. – 2022.
66 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Instituto UFC Virtual, Curso de Sistemas e Mídias Digitais, Fortaleza, 2022.
Orientação: Prof. Dr. Gilvan Maia.

1. API. 2. Digimon. 3. GraphQL. I. Título.

CDD 302.23

GUILHERME CÂNDIDO PRETTO DE OLIVEIRA

GENNAI API: DESENVOLVIMENTO DE UMA API COM A TEMÁTICA DE DIGIMON UTILIZANDO GRAPHQL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto UFC Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Aprovada em: 17 de Fevereiro de 2022

BANCA EXAMINADORA

Prof. Dr. Gilvan Maia (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Me. Glaudiney Moreira Mendonça Junior
Universidade Federal do Ceará (UFC)

Me. Artur de Oliveira da Rocha Franco
Doutorando (Mestrado e Doutorado em Ciência da Computação - UFC)

RESUMO

O desenvolvimento de APIs é cada vez mais utilizado como solução sistemática para diversas empresas, sendo um conhecimento muito procurado no mercado de desenvolvimento para a web, assim como na criação de repositórios de dados para diversas mídias, como a PokéAPI, que provém dados relacionados à franquia Pokémon, ou The Rick and Morty API, que da mesma forma provém dados relacionados à séria animada Rick and Morty. Toda API segue um padrão de arquitetura, sendo o padrão REST o mais consolidado e utilizado atualmente no mercado. Entretanto, um novo padrão de arquitetura de API, chamado GraphQL, propõe uma abordagem diferente e mais prática, e está ganhando cada vez mais espaço no mercado, com expectativas de rivalizar e até superar o REST em alguns anos. Dessa forma, buscando desenvolver um projeto que possa ser utilizado como um repositório de dados e referencia para outros desenvolvedores, este trabalho propõe o desenvolvimento de uma API utilizando o padrão de arquitetura GraphQL, buscando prover dados relacionados à uma franquia midiática. A franquia escolhida foi Digimon, aproveitando-se do vasto universo midiático que o mesmo tem, como jogos, séries, TCG, filmes e outros, e da grande comunidade ativa.

Palavras-chave: API. Digimon. GraphQL.

ABSTRACT

The development of API's is increasingly used as a systematic solution for several companies, being a much sought after knowledge in the web development market, as well as in the creation of data repositories for various media, such as the PokéAPI, which provides data related to the Pokémon franchise, or The Rick and Morty API, which likewise provides data related to the Rick and Morty animated series. Every API follows an architectural pattern, the REST pattern being the most consolidated and used currently on the market. However, a new API architecture pattern, called GraphQL, proposes a different and more practical approach, and is gaining more and more space in the market, with expectations to rival and even surpass REST in a few years. Thus, seeking to develop a project that can be used as a data repository and reference for other developers, this work proposes the development of an API using the GraphQL architecture pattern, seeking to provide data related to a media franchise. The franchise chosen was Digimon, taking advantage of the vast media universe that it has, such as games, series, TCG, movies and more, and the large active community.

Keywords: API. Digimon. GraphQL.

LISTA DE FIGURAS

Figura 1 – PokéAPI Logo 19
Figura 2 – The Rick and Morty API Logo 19
Figura 3 – NASA APIs Logo 20
Figura 4 – Recebimento e envio de mensagens pelo padrão *Representational State Transfer* (REST) 21
Figura 5 – Recebimento e envio de mensagens pelo padrão *Graph Query Language* (GraphQL) 21
Figura 6 – Logo da Gennai API 27
Figura 7 – Estruturas de Dados dos tipos Digimon, Rank, Field, Attribute e Type 28
Figura 8 – Estruturas de Dados dos tipos *Universe*, *Movie*, *Series* e *Episode* 29
Figura 9 – Estruturas de Dados dos tipos Character, Digivice, Crest, Digimental e Spirit 30
Figura 10 – Enumerações dos tipos DigiviceType e SpiritElement 30
Figura 11 – Estruturas de Dados da Gennai API 32
Figura 12 – Interface do *Visual Studio Code* (VSCode) com a organização inicial de arquivos e pastas 36
Figura 13 – Interface do VSCode com a organização finalizada de arquivos e pastas . . . 37
Figura 14 – Tipo de dados Digimon representado pelo Framework Prisma (à esquerda) e pelo Framework Apollo (à direita) 38
Figura 15 – Tipo de dados Query com exemplos 38
Figura 16 – Tipo de dados Mutation com exemplos 39
Figura 17 – Resolvers para as propriedades do tipo Field 39
Figura 18 – Resolvers para as propriedades do tipo Query 39
Figura 19 – Resolvers para as propriedades do tipo Mutation 40
Figura 20 – Ordem de inserção de dados pelo tipo 41
Figura 21 – Documentação gerada à partir do Framework Prisma 42
Figura 22 – Estrutura de pastas e arquivos do Gennai Node 44
Figura 23 – Interface da página de documentação da PokéAPI 47
Figura 24 – Interface da página de documentação da *The Rick and Morty* API 48
Figura 25 – Interface do VSCode com a organização inicial de arquivos e pastas da página de documentação da Gennai API 49
Figura 26 – Página da documentação da Gennai API com cabeçalho e barra lateral . . . 50

Figura 27 – Página da documentação da Gennai API completa 51

LISTA DE TABELAS

Tabela 1 – Linguagens mais utilizadas em 2021 22
Tabela 2 – Linguagem primária mais utilizada em 2021 23

LISTA DE ABREVIATURAS E SIGLAS

REST	*Representational State Transfer*
GraphQL	*Graph Query Language*
VSCode	*Visual Studio Code*
API	*Application Programming Interface*
APIRD	*API de Repositório de Dados*
NPM	*Node Package Manager*
XML	*Extensible Markup Language*
JSON	*JavaScript Object Notation*
HTTP	*Hypertext Transfer Protocol*
SOAP	*Simple Object Access Protocol*
RPC	*Remote Procedure Call*
TCG	*Trading Card Game*
CRUD	*Create, Read, Update, Delete*
HTML	*HyperText Markup Language*
CSS	*Cascading Style Sheets*
SQL	*Structured Query Language*
ORM	*Object Relational Mapper*
CORS	*Cross-Origin Resource Sharing*

SUMÁRIO

1	**INTRODUÇÃO**	12
2	**REFENCIAL TEÓRICO**	16
2.1	**API**	16
2.2	**APIs de Repositório de Dados**	17
2.2.1	*PokéAPI*	18
2.2.2	*The Rick and Morty API*	19
2.2.3	*NASA Application Programming Interface (API)s*	19
2.2.4	*Marvel Comics API*	19
2.3	**GraphQL x REST**	20
2.4	**Linguagens de Programação**	22
3	**METODOLOGIA**	24
3.1	**Gennai API**	24
3.2	**Gennai Node**	24
3.3	**Página com Documentação**	24
4	**GENNAI API**	26
4.1	**Escolha do Nome**	26
4.2	**Análise de Dados**	27
4.3	**Escolha das Tecnologias**	31
4.3.1	*Linguagem de Programação*	31
4.3.2	*Frameworks*	33
4.3.3	*Plataforma de Desenvolvimento*	35
4.4	**Desenvolvimento da API**	35
4.4.1	*Criação da Estrutura Base*	35
4.4.2	*Aplicação das Estruturas de Dados da Gennai API*	37
4.4.3	*Processo de deploy da API*	39
4.5	**População dos Dados na API**	40
4.6	**Documentação da API**	41
5	**GENNAI NODE**	43
5.1	**Escolha da Linguagem de Programação**	43
5.2	**Desenvolvimento da Biblioteca**	43

5.3 Publicação da Biblioteca . . . 45
6 PÁGINA COM DOCUMENTAÇÃO . . . 46
6.1 Coleta de Referências . . . 46
6.2 Escolha das Tecnologias . . . 47
6.3 Desenvolvimento da Página . . . 48
6.4 Publicação da Página . . . 49
7 CONCLUSÃO E TRABALHOS FUTUROS . . . 52
REFERÊNCIAS . . . 53
GLOSSÁRIO . . . 54
APÊNDICES . . . 55
APÊNDICE A–ARQUIVO SERVER . . . 55
APÊNDICE B–ARQUIVO API.TS . . . 57
APÊNDICE C–ARQUIVO INDEX.TS DA PASTA ATTRIBUTE . . . 59
APÊNDICE D–ARQUIVO INTERFACES.TS DA PASTA ATTRIBUTE 62
APÊNDICE E–ARQUIVO INDEX.TS DA RAIZ DO PROJETO GENNAI NODE . . . 64

1 INTRODUÇÃO

O termo API, no Português, Interface de Programação de Aplicações, denota um conjunto de estruturas computacionais que permitem reusar chamadas para a construção de aplicações. Uma API tradicional pode ser bastante complexa, a exemplo dos motores de jogos (MAIA *et al.*, 2003) e interfaces proprietárias para a construção de *plugins*, i.e., extensões dinâmicas para softwares[1].

Uma interpretação mais moderna do conceito de API remete a uma forma com que duas aplicações de computadores "conversam" entre si através de protocolos de rede (e.g., HTTP, RPC, etc) usando uma linguagem que ambas entendam (JACOBSON *et al.*, 2012). Essa interpretação passou a ser mais difundida nos anos 2000, através da empresa Salesforce, embora o conceito da mesma já existia desde os primórdios da Ciência da Computação (PURKAYASTHA, 2020).

Em uma pesquisa realizada pela Apigee (2016), ficou evidenciado que o tráfego de APIs, apenas durante o período do janeiro de 2014 até dezembro de 2015, cresceu cerca de 2,8 vezes por ano. Em outra pesquisa realizada pela ProgrammableWeb (2019), percebe-se que de 2015 à data em que a pesquisa foi efetuada (2019), aproximadamente mais de 2.000 APIs foram desenvolvidas. Além disso, esta mesma pesquisa também informa que apenas nos seis primeiros meses do ano de 2019, houve um aumento de cerca de 30% no número de novas APIs publicadas, quando se compara o período aos 4 anos anteriores.

Essa grande procura por APIs se dá, dentre outros motivos, pela facilidade de desenvolvimento da conferida às aplicações. Isso é especialmente interessante quando a API é provida utilizando um padrão de arquitetura chamado de REST, que, resumidamente, contempla um conjunto de normas e regras a serem seguidas no momento da criação da aplicação, de forma a restringir o processo de desenvolvimento, tornando o trabalho do desenvolvedor mais organizado, padronizado e previsível. Em uma pesquisa realizada pela ProgrammableWeb (2017), as APIs com padrão de arquitetura REST compõem 81,53% do repositório de APIs estudada. Muito embora esse tipo de API seja majoritariamente predominante no mercado, outro padrão de arquitetura vem ganhando relevância pela sua abordagem bastante diferente da do REST: o padrão de arquitetura chamado de GraphQL.

Embora existam muitas diferenças entre as arquiteturas REST e GraphQL, a maior delas se verifica pela forma de construção e recebimento de mensagens na aplicação. Enquanto a

[1] http://docs.autodesk.com/MAYAUL/2014/ENU/Maya-API-Documentation/index.html

arquitetura REST procura pré-definir métodos de comunicação para com a API, a arquitetura GraphQL dá mais liberdade para o desenvolvedor, permitindo quem está acessando a API dizer o que e em qual momento deseja buscar. Além disso, a maior parte do padrão emergente GraphQL supera o padrão REST e, mesmo que GraphQL ainda não seja o tipo majoritário de APIs, GraphQL aponta de forma funcional para o fim da "era" REST (Nordic APIs, 2018), como descrito a seguir.

Criado pelo Facebook, o padrão GraphQL veio justamente para resolver problemas que o padrão REST não consegue resolver. Desde seu surgimento em 2012, o GraphQL teve um crescimento considerável e vem recebendo um grande suporte pela comunidade de desenvolvedores, de modo que diversos *frameworks* já foram criados para trabalhar com esse padrão específico.

Como será visto na Subseção 2.2, uma API pode ser classificada em diversos tipos por diversas categorias. Dentre estes tipos, existem APIs públicas e privadas que têm como função única prover dados sobre um tema específico. Pelo seu objetivo ser claro e direto, nomearemos esse tipo de API pela sua função como *API de Repositório de Dados* (APIRD).

As APIRDs costumam prover dados específicos sobre um assunto, sendo muito comum a escolha de obras midiáticas, como é o caso de Pokémon e Rick and Morty, além de muitas outras obras e franquias que têm suas APIRDs desenvolvidas e reconhecidas no momento da escrita deste relatório técnico (e.g., Lord of the Rings, Star Wars, etc). Apesar disso, muitas obras ainda não possuem APIRDs próprias ou contam com projetos normalmente criados por fãs, os quais são inacabados ou que contam com uma quantidade reduzida de dados. Um exemplo emblemático disto é a franquia Digimon, que tem características bem semelhantes às de Pokémon em diversos âmbitos, mas que não tem uma APIRD propriamente desenvolvida.

Uma pesquisa preliminar realizada no GitHub identificou a existência de cerca de 100 projetos envolvendo os termos "Digimon API". Contudo, numa busca mais detalhada, constatou-se que a maioria desses projetos se propunham a prover poucos tipos de dados da franquia, mais especificamente listavam apenas os Digimon (criaturas fantásticas da franquia homônima) e em muitos outros casos uma quantidade muito pequena dos mesmos. Em nenhum desses projetos foi detectado um nível maior de profundidade de detalhe e diversidade, além de que, em sua maioria, muitos tem suas últimas atualizações datam de 6 meses ou mais — uma evidência de que tais projetos foram abandonados.

A escolha da franquia Digimon como tema de APIRD, além dos problemas citados

anteriormente, se beneficia pelos seguintes fatores:

- Inexistência de APIs com profundidade de dados e que provenham dados detalhados e variados;
- Existência de páginas robustas de informações sobre o tema na web, tendo inclusive uma oficial, que contém dados diversos, facilitando o trabalho de coleta de dados; e
- Grande e ativa comunidade, podendo, a curto ou longo prazo, chamar a atenção de contribuidores, de forma a expandir e melhorar o projeto mais rapidamente. Note-se que tal produto guarda semelhanças com outras produções multimídia, de modo que APIRD se constitui produto diferenciado em um portfólio profissional ao mesmo passo que demonstra capacidade em se lidar com cenários similarmente complexos em outrod domínios.

Outro fator importante na criação de uma APIRD semelhante às citadas nas subseções da Seção 2.2 é o nome dado para a mesma. O nome é geralmente curto e simples, ou remeter a algo relevante e de fácil reconhecimento dentro do escopo do conteúdo. No caso da PokéAPI, percebe-se a união da palavra “Pokémon” com o sufixo “API”. Isso faz muito sentido levando em consideração o universo de Pokémon e como muitos termos dentro da franquia já se utilizam do prefixo “Poké” (e.g., Pokédex e Pokéball, dentre outros). Já no caso do *The Rick and Morty* API, o próprio nome da série animada foi escolhido, novamente procurando enfatizar capacidade de reconhecimento e memorização por parte do seu público de usuários.

Na mesma pesquisa citada anteriormente nesta seção, boa parte dos resultados apresentavam nomes de APIs com variantes da palavra “Digimon”. Dessa forma, pelo grande excesso da mesma, um novo termo se mostraria mais relevante e impactante, o termo escolhi foi Gennai. No universo de Digimon, Gennai é o nome dado para uma espécie de guru, um mentor, um ser digital na figura de um idoso, que tem como principal função ajudar e guiar os “digiescolhidos” (protagonistas da história) nas suas aventuras com diversas informações. O termo Gennai apareceu pela primeira vez na primeira e mais marcante série animada, *Digimon Adventure*. Dessa forma, podemos listar, de forma resumida, os 3 motivos para a escolha desse nome:

- A função que a API se propõe a realizar é prover dados de forma gratuita para quaisquer desenvolvedores e sistemas, assemelhando-se ao papel desempenhado pelo personagem Gennai na série animada;

- Excesso do termo Digimon e variantes em outros projetos de API; e
- Valor nostálgico, justificado pela sua importância na clássica série animada.

Partindo dos dados analisados previamente, chegamos à um questionamento: como se daria a criação de uma APIRD com tema Digimon e utilizando o padrão de arquitetura GraphQL?

Sendo assim, o objetivo deste trabalho é relatar o desenvolvimento de uma API, ou, mais especificamente, uma APIRD que se equipare à outras já existentes e conhecidas na comunidade de desenvolvedores GraphQL. Um objetivo técnico deste projeto é fazer com que a API utilize ferramentas tecnológicas modernas, como o padrão de arquitetura de protocolo GraphQL, utilizando-se dos dados disponíveis na internet de forma oficial ou feitas por fãs sobre a franquia Digimon. Além da API, serão relatados os processos de criação de 2 outras aplicações que complementam a ideia de uma APIRD. A primeira é uma biblioteca registrada no *Node Package Manager* (NPM), de forma a facilitar a utilização da API. E a segunda será uma página na web contendo a documentação da API, assim como tutoriais de sua utilização básica.

2 REFENCIAL TEÓRICO

Neste Capítulo será aprofundado o conceito de API e seus tipos. Também será detalhado o conceito de um tipo de API, exemplificando diferentes casos. Além disso, o padrão de arquitetura GraphQL será apresentado junto com suas diferenças em relação ao padrão de arquitetura REST. Por fim, serão apresentados dados relacionados às linguagens de programação utilizadas no mercado.

2.1 API

Como dito anteriormente, o conceito de API é um artefato computacional (uma biblioteca ou mesmo uma aplicação) que funciona como interface para outras aplicações se comunicarem. No contexto da Web, essa comunicação é geralmente intermediada por dois formatos de mensagem mais difundidos, ambos para intercâmbio de dados, sendo *Extensible Markup Language* (XML) o primeiro historicamente utilizado e mais tarde o *JavaScript Object Notation* (JSON). Esse termo teve seu nascimento durante o evento IDG Demo 2000, lançado pela primeira vez pela empresa Salesforce em 7 de Fevereiro de 2000, utilizando XML como formato de mensagens (Mundo API, 2016). Uma API pode ser utilizada para diversos áreas da programação, embora seja, atualmente, muito conhecida pela sua grande presença e fácil desenvolvimento no mercado de programação para a web, principalmente através do protocolo *Hypertext Transfer Protocol* (HTTP). Desde a sua primeira implementação, muitas empresas adotaram esse modelo de aplicação como solução tecnológica, como o portal eBay também no ano 2000 seguindo a Salesforce, até o ano 2010 outras empresas muito conhecidas mundialmente, como Amazon, Facebook, Instagram e Twitter (Mundo API, 2016).

Existem algumas formas de categorizar APIs no mercado, seriam elas pela disponibilidade, pelo serviço e pelo protocolo. A categoria de disponibilidade diz respeito à exclusividade de acesso à mesma, sendo dividida em 4 tipos (Link API, 2021):

- Públicas (ou abertas): são APIs com acesso gratuito para qualquer tipo de usuário e em qualquer sistema;
- Privadas (ou internas): são APIs criadas para utilização interna em outros sistema de uma mesma empresa;
- De parceiros: são APIs que não estão disponíveis publicamente e para seu uso é necessário adquirir direitos para o mesmo, geralmente elas resolvem problemas

já conhecidos no mercado e se propõem a resolvê-los como um serviço pago; e

- Compostas: são APIs que combinam vários dados ou APIs diferentes.

Já a categoria de serviço, diz respeito à função na qual a API se propõem a realizar, sendo também dividida em 4 tipos (Link API, 2021):

- Dados: são APIs que tem como função prover dados específicos para outros sistemas;
- Serviços Internos: são APIs que disponibilizam serviços internos para uma empresa;
- Serviços Externos: são APIs criadas por terceiros que disponibilizam serviços que podem ser agregados por outras empresas; e
- Experiência do Usuário: são APIs que buscam ajudar o desenvolvedor a fornecer a experiência ideal para um dispositivo.

Por fim, a categoria de protocolo diz respeito à forma de transmissão de mensagens entre a API e outros sistemas. Existem diversos protocolos, cada um com características bem definidas, sendo estes os mais famosos e utilizados no mercado:

- REST: de forma sucinta, é um padrão que espera requisições HTTP do tipo GET, POST, DELETE e UPDATE, sendo cada uma dessas representada por um comportamento em relação à uma entidade;
- *Simple Object Access Protocol* (SOAP): é um padrão que também se utiliza do protocolo HTTP e consiste na utilização de XML para a transferência de mensagens (IBM, 2020);
- *Remote Procedure Call* (RPC): é o tipo mais antigo e simples de API, e embora seja bem semelhante ao REST, todo o código é muito acoplado, o que impacta fortemente em manutenções futuras, sendo assim pouco utilizado atualmente; e
- GraphQL: é uma abordagem diferente da do REST, ela trabalha apenas com as requisições HTTP dos tipos GET e POST, e dá a liberdade para escolher que informações serão enviadas pela API, de forma a economizar mais banda e não trazer dados desnecessários (EZ Devs, 2019).

2.2 APIs de Repositório de Dados

Utilizando as 2 primeiras categorias de classificação de API citadas na seção 2.1, é possível desenvolver diferentes tipos de aplicações para diferentes propósitos. Dentre essas

combinações, ao escolher a disponibilidade como pública e a função como sendo de prover dados, temos como resultado uma API que proverá dados de forma gratuita para quaisquer outros sistemas e desenvolvedores. Neste trabalho, tal combinação de tipos de API adotará a nomenclatura de APIRD.

Como dito anteriormente no Capítulo 1, as APIRDs têm como função única prover dados, geralmente sobre um conteúdo específico e com acesso gratuito. Esse tipo de API ainda está longe ser o mais desenvolvido no mercado, principalmente pelo seu teor Open Source, o que, em sua grande maioria, não lhe confere, necessariamente, um retorno financeiro. Sendo assim, muitas dessas aplicações originam-se do interesse pessoal de um ou mais desenvolvedores acerca de um assunto.

O conteúdo de uma APIRD pode variar bastante, desde um produto midiático a dados técnicos. Há diversas aplicações usadas para uso pessoal e profissional (e.g., derivação de produtos, introdução de referências, criação de histórias, etc). Algumas são criadas por fãs, pelas próprias empresas detentoras dos direitos do conteúdo ou por empresas estatais.

2.2.1 PokéAPI

Feita sem fins lucrativos por fãs, essa API provém diversos tipos de dados sobre a franquia de Pokémon, contemplando desde itens de um jogo específico aos personagens de uma série animada. No momento da escrita deste documento, o projeto conta com cerca de 64 contribuidores na sua página do GitHub. A franquia costuma ser muito comparada com Digimon devido à sua similaridade em muitos aspectos:

- Em Pokémon, existem criaturas chamadas de Pokémon (Pocket Monsters), enquanto em Digimon existem criaturas chamadas de Digimon (Digital Monsters). Ambas as criaturas compartilham características bem semelhantes, desde aspectos evolutivos à habilidades;
- Ambos surgiram em anos bem próximos, atraindo a atenção dos mesmos fãs, criando-se uma certa rivalidade por parte de muitos; e
- Ambos têm seu universo expandido com jogos, séries animadas, filmes e *Trading Card Game* (TCG).

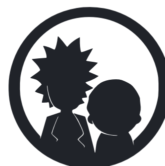
Figura 1 – PokéAPI Logo

Fonte: PokéAPI Website.

2.2.2 *The Rick and Morty API*

Uma API baseada na série animada “Rick and Morty”, desenvolvida sem fins lucrativos e também mantida por fãs, ela atualmente provém dados de centenas de personagens, imagens, localizações e episódios. Embora a franquia não guarde tantas semelhanças com o universo de Digimon, há algumas analogias no aspecto tecnológico devido ao uso de tecnologias como o padrão de arquitetura GraphQL e pelo fato de ser feita usando as mesmas linguagens de programação que foram utilizadas na criação da Gennai API. Além disto, como dito anteriormente, essa API também provém dados de episódios e personagens, como a Gennai API.

Figura 2 – The Rick and Morty API Logo



Fonte: The Rick and Morty API Website.

2.2.3 *NASA APIs*

Um conjunto de APIs oficiais que servem diversos dados astrológicos disponíveis gratuitamente. Atualmente engloba 17 APIRDs diferentes, provendo dados de diversos planetas, sóis, luas e outros. Este é exemplo de API desenvolvida por uma empresa estatal e que dispõe de diversas interfaces aderentes ao padrão REST, mas que seguem o conceito de APIRD.

2.2.4 *Marvel Comics API*

Totalmente gratuita, trata-se de uma API oficial baseada no universo de quadrinhos da Marvel. Provém dados de personagens, quadrinhos, histórias e muito mais. Um exemplo

Figura 3 – NASA APIs Logo

Fonte: NASA APIs Website.

de APIRD desenvolvida oficialmente pela Marvel, muito reconhecida pela sua estruturação dos dados, que é um exemplo em termos de boas práticas.

2.3 GraphQL x REST

O padrão GraphQL, utilizado no desenvolvimento da Gennai API, permite a livre inclusão e exclusão do conteúdo que será requisitado. Isso difere do que ocorre no padrão REST, que trabalha com rotas específicas[1] para diferentes requisições. Assim, o GraphQL recebe requisições apenas de uma rota, sendo que é no corpo (ou parâmetros) da mensagem enviada nessa requisição que o servidor vai identificar que dados enviar como resposta.

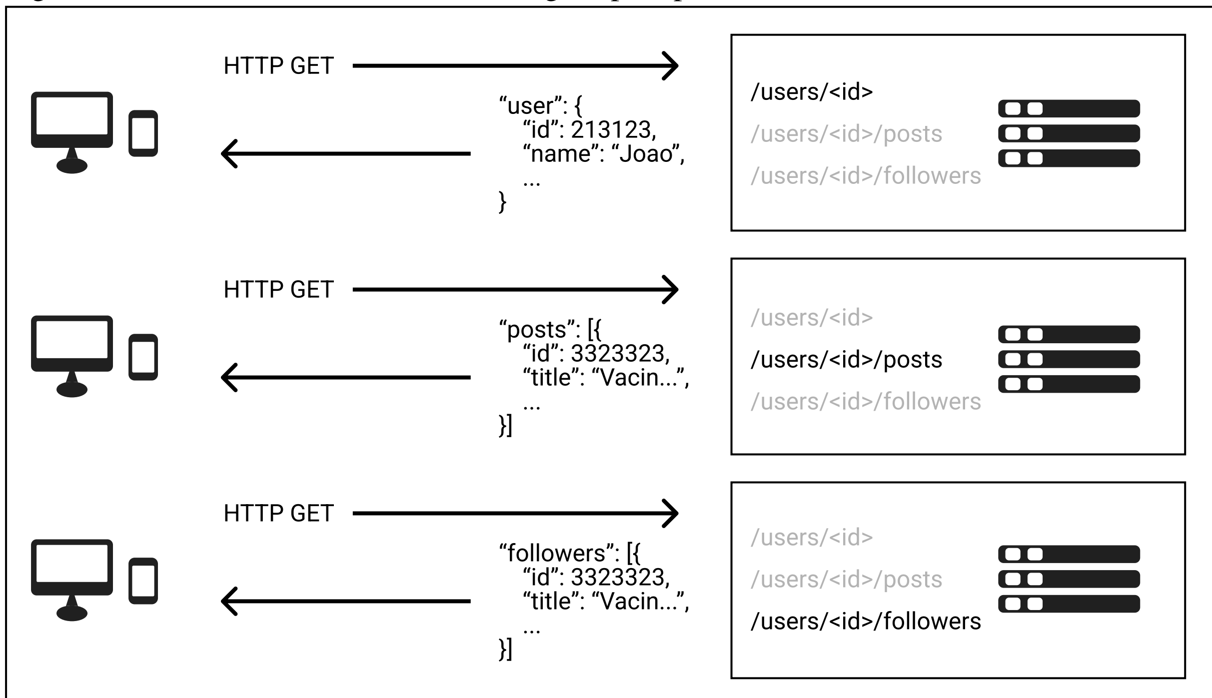
Essa diferença fica mais visível ao comparar a Figura 4 com a Figura 5. Nelas, podemos ver como funcionam os recebimentos e envios de requisições.

Na Figura 4 podemos constatar que existem 3 diferentes *endpoints* que podem ser usados na requisição REST, cada um com a responsabilidade de trazer um dado específico. Dessa forma, é notável que esse processo se repetirá muitas vezes para sistemas de maior escala. Consequentemente, há maior custo de desenvolvimento, além de não haver a liberdade de escolha de quais dados seriam retornados da requisição: nesse caso, são recebidos todos os dados das entidades “user”, “post” ou “follower” mesmo que tais dados não venham a ser utilizados na aplicação e apesar de seu impacto ser leve, em grande escala esse tipo de problema pode implicar na performance da aplicação.

Já na Figura 5, podemos ver que existe apenas um *endpoint* responsável por qualquer tipo de requisição GraphQL. Assim, o que difere o que a aplicação retornará de dados é o que o corpo da própria requisição pedir, no exemplo apresentado pedimos os dados da entidade “User” especificando o seu id e logo após definimos que dados queremos receber dessa requisição. Sendo assim, é possível concluir que o padrão de arquitetura GraphQL é bem mais flexível na forma de receber e enviar mensagens se comparado ao padrão de arquitetura REST.

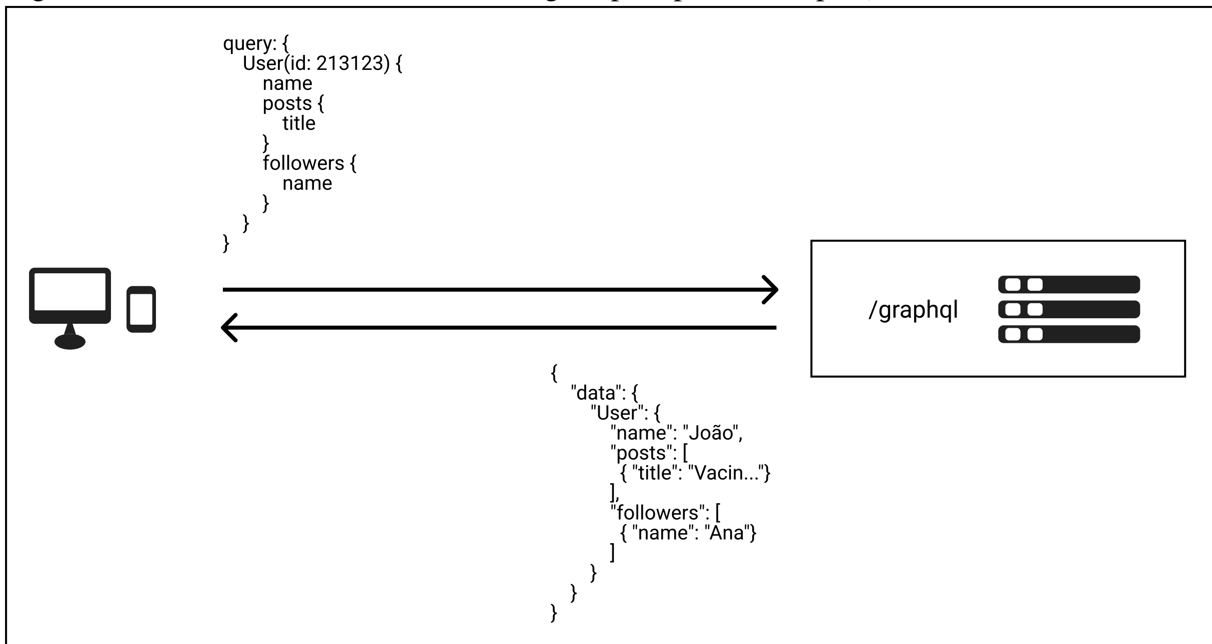
[1] Um endereço associado a uma funcionalidade da API, o qual produz um resultado dinâmico em função dos valores dos parâmetros oferecidos durante a invocação àquela função via requisição.

Figura 4 – Recebimento e envio de mensagens pelo padrão REST



Fonte: O Autor.

Figura 5 – Recebimento e envio de mensagens pelo padrão GraphQL



Fonte: O Autor.

Além disso, no padrão de arquitetura GraphQL, as requisições só podem ser feitas através dos métodos GET e POST, embora ainda funcionem como um *Create, Read, Update, Delete* (CRUD), além de dividir internamente as requisições em 2 tipos:

1. *Query*: um tipo de estrutura de dados já existente que funciona apenas para a leitura de dados. Na semântica do CRUD, utilizado no padrão REST, esse tipo

é responsável apenas pela leitura (*Read*) de dados, não permitindo alterá-los. O GraphQL permite a utilização de ambos os métodos GET e POST em uma requisição do tipo Query; e

2. *Mutation*: um tipo de estrutura de dados já existente que funciona para qualquer modificação dos dados remotamente. Na semântica do CRUD, utilizado no padrão REST, esse tipo é responsável pela atualização (*Update*), inserção (*Create*) e remoção (*Delete*) de dados. O GraphQL permite a utilização apenas do método POST em uma requisição do tipo *Mutation*.

2.4 Linguagens de Programação

Considerando o desenvolvimento prático de virtualmente qualquer tipo de software, uma das etapas mais importantes é a escolha das linguagens de programação que serão utilizadas no processo. Essa escolha deve levar em conta diversos fatores, tais como:

- Plataforma em que o software será utilizado;
- Funcionalidades que o software deverá ser capaz de realizar;
- Manutenções posteriores à finalização e implantação do software; e
- Escalabilidade do projeto.

Seguindo essa lógica, uma API pode ser desenvolvida em diversas linguagens de programação, desde que elas tenham suporte às características e objetivos que reflitam as necessidades da aplicação que encapsula a API.

Para a escolha da linguagem utilizada no desenvolvimento da Gennai API, foi utilizado um levantamento feito pela JetBrains (2021), em que foi apresentada uma lista das linguagens de programação mais utilizadas no ano de 2021. Removendo as linguagens que não são capazes de serem usadas para desenvolver uma API e limitando para apenas as 4 com maior adesão de uso, temos o resultado apresentado na Tabela 1.

Tabela 1 – Linguagens mais utilizadas em 2021

Linguagem de programação	Porcentagem de uso
JavaScript	69
Python	52
Java	49
PHP	32

Fonte: JetBrains (2021)

Ainda nesta mesma pesquisa, foi apresentada a linguagem de programação primária

mais utilizada pelos desenvolvedores. Também removendo as linguagens que não servem para desenvolver APIs e limitando às 4 maiores porcentagens, temos o resultado apresentado na tabela 2.

Tabela 2 – Linguagem primária mais utilizada em 2021

Linguagem de programação	Porcentagem de desenvolvedores
JavaScript	39
Java	32
Python	29
PHP	22

Fonte: JetBrains (2021)

Analisando os dados pontuados da pesquisa, é possível determinar que JavaScript é a linguagem de programação que tanto teve mais utilizações pelos desenvolvedores, quanto também é a mais usada como linguagem primária pelos desenvolvedores. Note-se que Python e Java vêm logo em seguida nesse *ranking*.

3 METODOLOGIA

Este capítulo irá pontuar, de forma breve, as etapas do processo de criação dos softwares descritos como objetivo geral no Capítulo 1. Primeiro será abordada a Gennai API, seguida pela a biblioteca que facilitará o acesso à API, cujo nome é *Gennai Node*. Por fim, é descrita uma página na web que servirá de documentação para Gennai API e *Gennai Node*.

3.1 Gennai API

O processo de desenvolvimento da Gennai API se deu em 5 etapas:

1. Escolha do Nome: escolha do nome utilizado para representar a API;
2. Análise de Dados: atribuição dos recursos e dados providos pela API;
3. Escolha das Tecnologias: escolha das ferramentas tecnológicas que foram usada para o desenvolvimento da API;
4. Desenvolvimento da API: processo detalhado de desenvolvimento da API;
5. População dos Dados na API: inserção de dados na API; e
6. Documentação da API: documentação dos métodos e recursos que a API dispõe.

3.2 Gennai Node

O processo de desenvolvimento do *Gennai Node* se deu em 3 etapas:

1. Escolha da Linguagem de Programação: escolha da linguagem que receberia uma Lib que auxiliasse no acesso à API, i.e., encapsulando chamadas e oferecendo estruturas comuns, de alto nível, simplificando o trabalho do desenvolvedor no lado da aplicação;
2. Desenvolvimento da Biblioteca: trata-se do próprio processo de construção do código-fonte que implementa a biblioteca; e
3. Publicação da Biblioteca: processo de disponibilização da biblioteca por meio do NPM.

3.3 Página com Documentação

O processo de desenvolvimento da página web da Gennai API se deu em 3 etapas:

1. Coleta de Referências: análise de outras páginas de APIs conhecidas;

2. Escolha das Tecnologias: escolha das ferramentas tecnológicas que foram usada para o desenvolvimento a página web; e
3. Desenvolvimento da Página: processo de desenvolvimento da página web; e
4. Publicação da Página: processo de publicação da página na internet.

4 GENNAI API

Este capítulo irá descrever, de forma minuciosa, as etapas do processo de criação da Gennai API. Como dito anteriormente na Seção 3.1, o processo de desenvolvimento se deu em seis etapas:

1. Escolha do Nome;
2. Análise de Dados;
3. Escolha das Tecnologias;
4. Desenvolvimento da API;
5. População dos Dados na API; e
6. Documentação da API.

4.1 Escolha do Nome

Durante o Brainstorming realizado previamente ao início do desenvolvimento da API, foi decidido o termo Gennai como parte do nome da API. Essa escolha se deu por dois motivos:

1. Uma pesquisa realizada no GitHub, discutida no capítulo 1; e
2. Valor simbólico representado pelo termo Gennai no universo das séries animadas de Digimon.

No caso da pesquisa, ficou evidenciado que boa parte dos resultados (que são projetos de API de Digimon) apresentavam nomes de APIs com variantes da palavra "Digimon", como foi o caso também da PokéAPI (Pokémon + API). Dessa forma, pelo grande excesso da mesma, um novo termo se mostraria mais relevante e impactante.

Já pelo valor simbólico, se analisado o termo e sua origem na série animada, fica mais clara a escolha do mesmo e como ele se adéqua perfeitamente à ideia de uma API. No universo de Digimon, Gennai é o nome dado para uma espécie de guru, um ser digital, na figura de um idoso, que tem como principal função ajudar e guiar os "digiescolhidos", protagonistas da história, nas suas aventuras com diversas informações. Ele apareceu pela primeira vez na primeira e mais marcante série animada, intitulada *Digimon Adventure*. Como a função que a API se propõe a realizar é prover dados de forma gratuita para quaisquer desenvolvedores e sistemas, assemelha-se ao papel desempenhado pelo personagem Gennai na série animada.

Por fim, utilizando-se da fonte característica do título da primeira série animada, a

logo foi construída, como pode ser visto na Figura 6.

Figura 6 – Logo da Gennai API

Fonte: O Autor.

4.2 Análise de Dados

A etapa de análise de dados teve como objetivo entender que dados seriam utilizados como recurso da API e como ficariam suas estruturas, além das motivações por trás da escolha do nome Gennai API. Como a franquia de Digimon é muito vasta, foi necessário limitar o escopo temporariamente e realizar uma seleção dos recursos principais, de forma a trabalhar inicialmente neles. Partindo disto, o escopo foi resumido às 5 primeiras séries animadas (e seus respectivos filmes):

- Digimon Adventure;
- Digimon Adventure 02;
- Digimon Tamers;
- Digimon Frontier; e
- Digimon Savers;

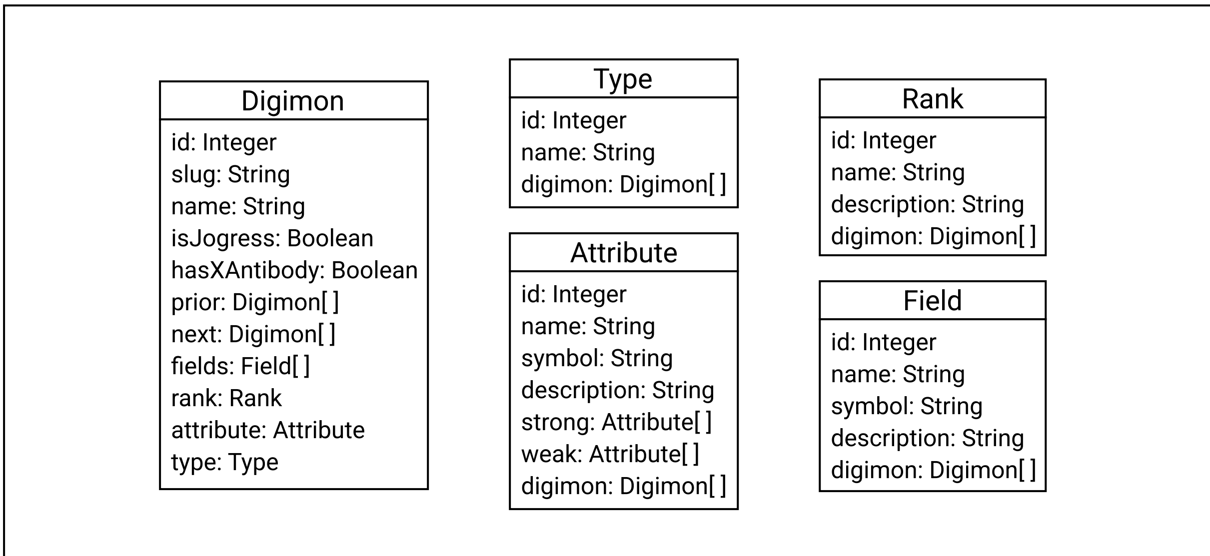
Logicamente, o recurso principal dessa franquia é a própria criatura, o *digimon*. Dessa forma, o recursos principais selecionados para trabalhar inicialmente foram os seguintes:

- Digimon;
- *Rank*;
- *Field*;
- *Attribute*; e
- *Type*.

Como a franquia gira em torno dos próprios digimon, era importante que o tipo Digimon fosse muito bem estruturado, assim como os outros tipos que o complementam. Sendo assim, as estruturas de dados completas dos tipos já citados podem ser visualizadas na Figura 7. Vale ressaltar que apenas com esses tipos de dados já seria o suficiente para se equiparar com a

maioria dos projetos existentes de APIRD sobre Digimon.

Figura 7 – Estruturas de Dados dos tipos Digimon, Rank, Field, Attribute e Type



Fonte: O Autor.

Após a estrutura do tipo Digimon estar bem consolidada, foi o momento de expandir os dados que seriam providos pela API. Mas antes de partir para séries, personagens, filmes ou outros, viu-se necessária a criação de uma divisão entre os dados como um próprio tipo de dados, assim foi criado o tipo de dados *Universe*. O tipo *Universe* tem como única função servir para diferenciar e agrupar informações de diferentes universos dentro da franquia Digimon. Dessa forma, a maioria dos dados podem ser incluídos dentro de um universo específico. Um exemplo claro disto seriam as séries animadas, no caso da primeira e segunda série que se passam na mesma linha temporal, ambas fazem parte do mesmo tipo *Universe*, já a terceira série animada se passa em outra linha temporal/universo, de forma que faria parte de um tipo *Universe* diferente da anterior.

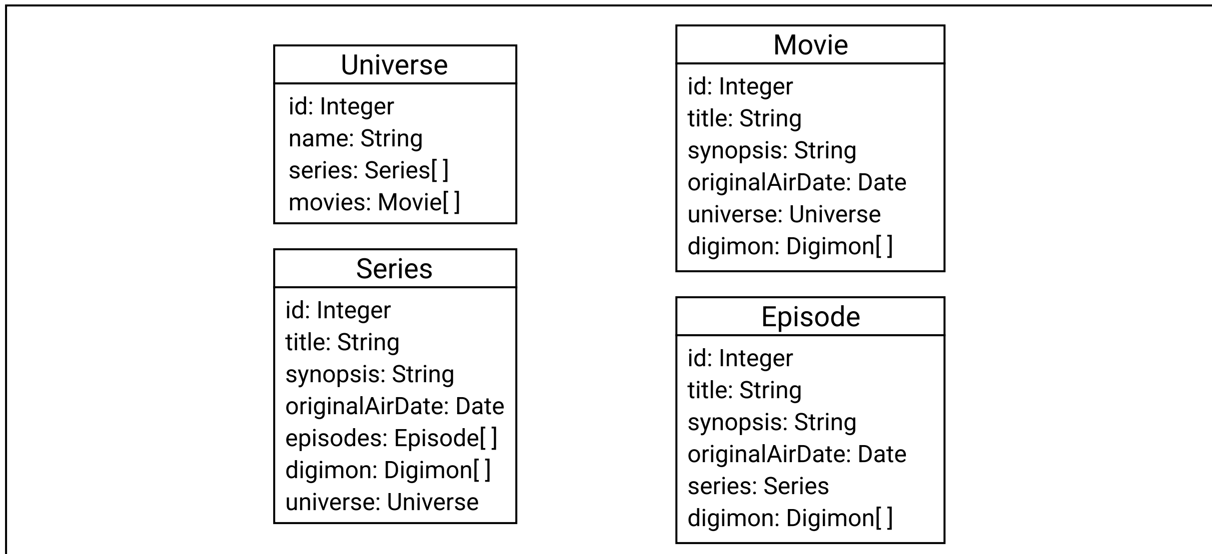
Após a criação do tipo de dados *Universe*, foi a vez de introduzir as estruturas de dados de séries, filmes e episódios, utilizando a *The Rick and Morty* API como referência. Sendo assim, os novos tipos de dados foram:

- *Universe*;
- *Movie*;
- *Series*; e
- *Episode*.

As novas estruturas de dados podem ser vistas mais detalhadas na Figura 8. A adição dessas novas estruturas de dados acarretaram na adição de novas propriedades na estrutura de dados do

tipo Digimon, como visto na Figura 11.

Figura 8 – Estruturas de Dados dos tipos *Universe*, *Movie*, *Series* e *Episode*



Fonte: O Autor.

O próximo passo foi a criação das estruturas de dados dos personagens, e seus complementos. Analisando principalmente as séries animadas da franquia, viu-se a necessidade de adição dos seguintes tipos de dados:

- *Character*;
- *Digivice*;
- *Crest*;
- *Digimental*; e
- *Spirit*.

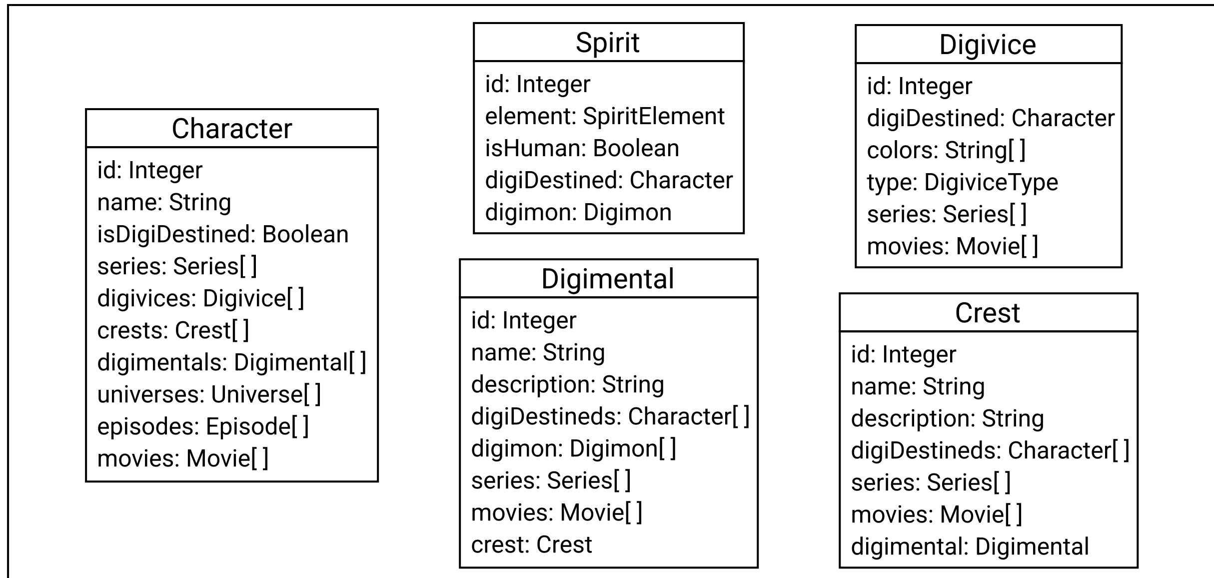
O tipo de dado *Character* tem como função guardar dados de quaisquer personagens que tiveram algum aparecimento na franquia Digimon, dessa forma, como personagens em diferentes universos têm especifidades diferentes, foram adicionados tipos para cada especificidade que fizesse sentido. As recém adicionadas estruturas de dados podem ser vistas em detalhes na Figura 9. De forma mais detalhada, os motivos que levaram à criação dos outros tipos foram as seguintes:

- *Digivice*: foi incluído pela grande quantidade de características e tipos de *digivices* que são encontras na franquia ao longo de todas as obras;
- *Crest*: foi adicionado por ser um tipo de item muito importante para o enredo da 1ª série animada, *Digimon Adventure*;
- *Digimental*: assim como o tipo *Crest*, este tipo tem uma relevância muito grande

no enredo da 2ª série animada, *Digimon Adventure 02*; e

- *Spirit*: itens principais utilizados no enredo na 4ª série animada, *Digimon Frontier*.

Figura 9 – Estruturas de Dados dos tipos Character, Digivice, Crest, Digimental e Spirit



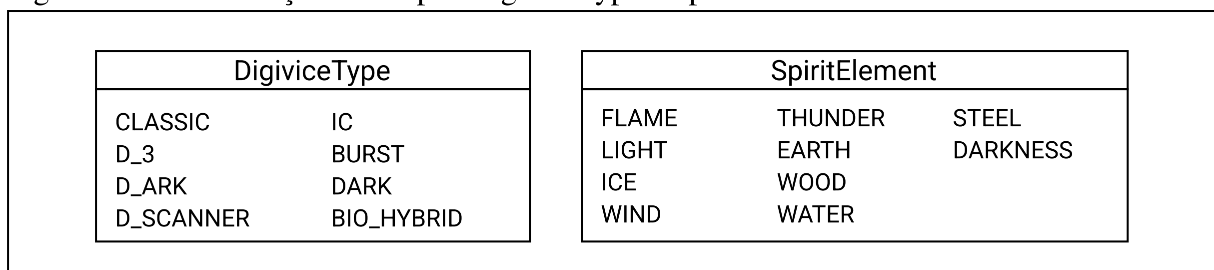
Fonte: O Autor.

Juntamente dos recém criados tipos, foram criadas também duas enumerações:

- *DigiviceType*: contém os tipos fixos de *digivices*, que é um atributo do tipo *Digivice*; e
- *SpiritElement*: contém os tipos fixos de elementos que podem ser utilizados como um atributo no tipo *Spirit*.

Essas enumerações e seus valores podem ser vistos na Figura 10.

Figura 10 – Enumerações dos tipos DigiviceType e SpiritElement



Fonte: O Autor.

A adição desses novos tipos de estrutura de dados também vieram a modificar as estruturas de dados já existentes, sendo elas Digimon, *Movie*, *Series* e *Episode*, como é visto na Figura 11.

Para finalizar, foram feitas pequenas adições de dados nos tipos Digimon, *Character*, resultando na criação de outros 3 tipos de estruturas de dados:

- *DigimonName*: complemento do tipo Digimon, responsável por armazenar os diferentes nomes atribuídos à um digimon em outras línguas;
- *DigimonGroup*: tipo de dados responsável por representar um grupo de digimons. Existem diversos na franquia, como “Royal Knights” e “Olympus XII”; e
- *CharacterName*: complemento do tipo *Character*, responsável por armazenar os diferentes nomes atribuídos à um personagem em outras línguas.

Por fim, após todas as modificações e adições, as estruturas finalizados podem ser visualizadas na Figura 11.

4.3 Escolha das Tecnologias

Com as estruturas dos dados que serão utilizados na Gennai API finalizadas, o desenvolvimento seguiu para a etapa de escolha de tecnologias. Essa etapa é responsável pela escolhas das melhores e mais adequadas tecnologias para o desenvolvimento da API. De forma direta, é onde ocorreram os questionamentos sobre quais linguagens de programação iriam ser utilizadas, assim como quais Frameworks seriam mais adequados, em qual plataforma a API será desenvolvida e que Libs ajudariam no processo.

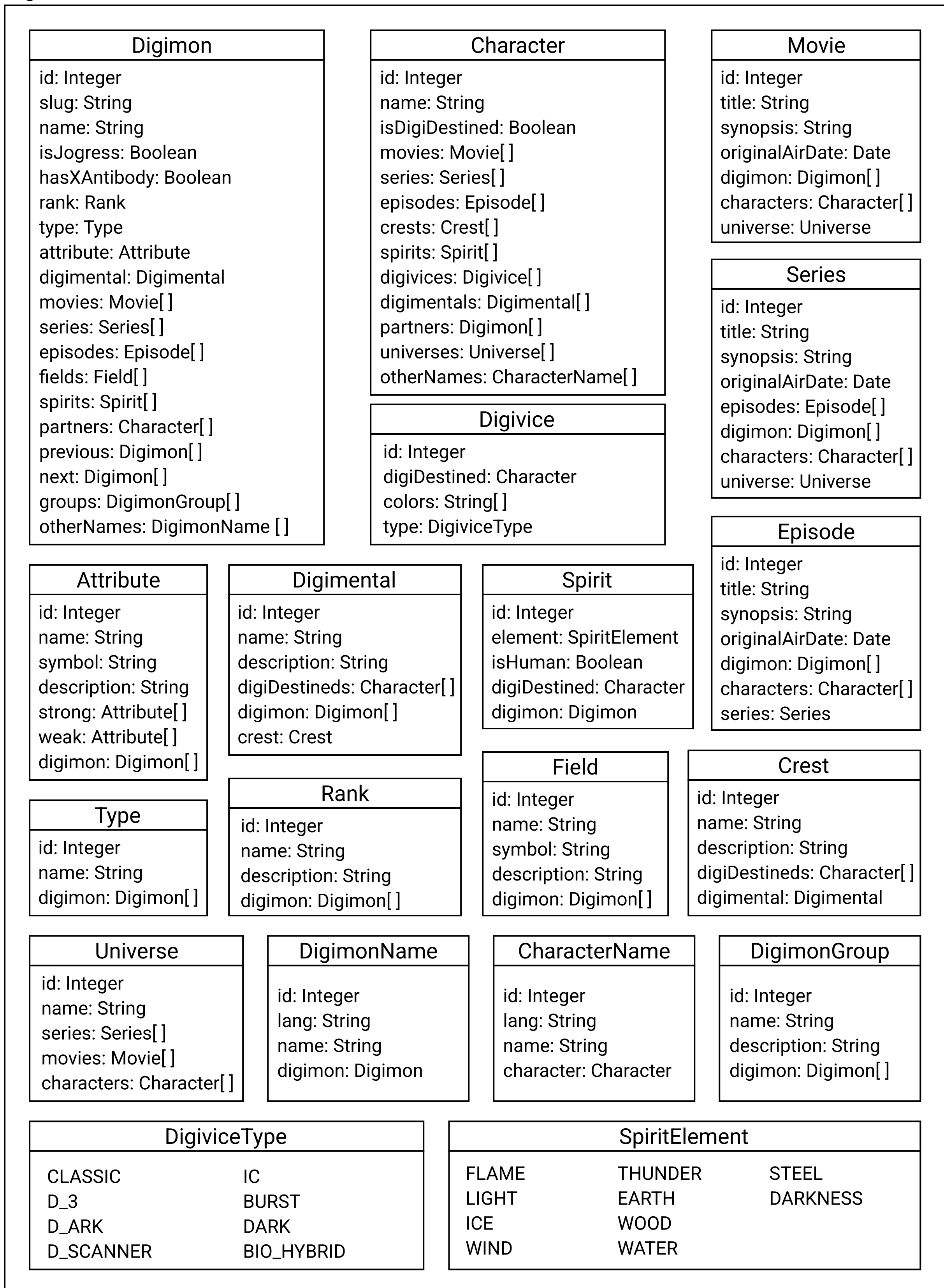
4.3.1 Linguagem de Programação

Através da análise realizada e dos dados discutidos na subseção 2.4, foi escolhida a linguagem de programação JavaScript para o desenvolvimento da Gennai API. Essa escolha se dá por 3 motivos:

1. Utilizando-se dos dados adquiridos da análise citada anteriormente, esta linguagem encontra-se sendo primariamente utilizada pelos desenvolvedores;
2. É uma linguagem muito conhecida pelos Frameworks e recursos disponíveis no desenvolvimento de APIs, e se destaca ainda mais no padrão de arquitetura GraphQL; e
3. Conhecimento pessoal da linguagem, o que facilitaria o desenvolvimento além de resultar em um projeto mais organizado e elaborado.

JavaScript, assim como Python e Java, é uma linguagem de programação muito

Figura 11 – Estruturas de Dados da Gennai API



Fonte: O Autor.

utilizada para desenvolver diversos tipos de aplicações. Desde jogos à aplicativos *mobile*, sua grande diferença é a fácil utilização e interpretação de código. Diferente de outras linguagens,

JavaScript é uma linguagem não-tipada, que significa a não necessidade de declarar os tipos de variáveis no código-fonte, diferentemente de Java que é totalmente tipada. Também é classificada como fraca, por ser uma linguagem que consegue realizar operações entre tipos diferente de variáveis sem mostrar erros, diferente de Python, que é forte. Pela sua grande utilização no *front-end* de uma aplicação, muitas vezes juntamente com *HyperText Markup Language* (HTML) e *Cascading Style Sheets* (CSS), JavaScript ganhou muito público e investimento, sendo hoje capaz de construir APIs e sistemas robustos sem depender de linguagens terceiras.

Além da linguagem utilizada para o desenvolvimento da API, foi necessário decidir neste momento o tipo de banco de dados utilizado para armazenar os dados. Também aproveitando-se da pesquisa citada anteriormente na subseção 2.4, é mostrado que *Structured Query Language* (SQL) é a linguagem mais utilizada e presente no cotidiano dos desenvolvedores. Dessa forma, entre as opções de bancos de dados disponíveis no mercado e que utilizam SQL, temos como as mais famosas as seguintes:

- MySQL;
- PostgreSQL; e
- SQLite.

Dentre os citados, todos serviriam ao propósito da API, além de serem bem parecidos entre si. Apesar disso, o PostgreSQL foi escolhido. Essa escolha se deu principalmente pela experiência pessoal com a mesma, além de ter uma documentação muito boa sobre a mesma.

Por fim, de forma sucinta, as linguagens escolhidas para desenvolver a Gennai API foram JavaScript, para a programação do sistema, SQL para realizar as conexões com o banco de dados e PostgreSQL como tipo de banco de dados.

4.3.2 Frameworks

O JavaScript sozinho não consegue ser executado ou utilizado como um sistema, é necessário que algo compile o código e o executa. Nos navegadores isso é feito internamente pelos mesmos, já quando estamos desenvolvendo uma aplicação fora do navegador, utilizamos o Node.js, que basicamente existe como um ambiente de execução para códigos JavaScript.

Aproveitando-se disso, um dos grande benefícios de utilizar a linguagem de programação JavaScript juntamente com o *Node.js*, é o imenso repositório de Libs que são disponibilizados para se trabalhar. Dentre essas Libs, existem aquelas que se propõem a facilitar o desenvolvimento orientando o desenvolvedor a trabalhar de uma forma específica, seguindo uma

lógica provida por eles, esse tipo de Lib é chamado de Framework.

Visando o melhor e mais organizado desenvolvimento da API, as opções para a escolha de Frameworks se resumiram às seguintes:

- Node.js: é possível utilizar apenas as ferramentas providas pelo *Node.js* para a criação de uma API, inclusive com suporte em GraphQL. Contudo, por ser bem genérico, o *Node.js* acaba não sendo a melhor opção para desenvolvimento;
- Express.js: um dos Frameworks mais utilizados no desenvolvimento de sistemas Back-End usando JavaScript. Ele procura facilitar o roteamento de requisições, além de entregar uma fácil configuração para a produção de APIs. Ele também tem suporte para diversos padrões de arquitetura de API, como o REST e o GraphQL; e
- Apollo: um Framework visado para a construção de APIs e pensado para o desenvolvimento utilizando o padrão de arquitetura GraphQL. Embora tenha suas restrições, ele consegue cumprir muito bem o que se propõe. Por ser direcionado para um tipo específico de padrão de arquitetura, ele consegue entregar o máximo de facilidade e eficiência durante o desenvolvimento com o mesmo.

Como a Gennai API é uma API feita exclusivamente em GraphQL, decidiu-se que o Framework Apollo era mais adequado, principalmente pelos recursos específicos disponíveis deste padrão de arquitetura. Sendo mais específico, este Framework é dividido em outros dois, o Apollo Server e o Apollo Client. O Apollo Server é o Framework principal, utilizado para o desenvolvimento de APIs. O Apollo Client é uma Lib que facilita o acesso de dados à APIs feitas em cima do Apollo Server. Para este momento apenas o Apollo Server importava, já que o objetivo era o desenvolvimento primariamente da API em si.

Além do Apollo, outro Framework foi escolhido para ajudar no desenvolvimento da API, chamado Prisma. Esse Framework funciona como um *Object Relational Mapper* (ORM), que tem como objetivo mapear entidades de tabelas de bancos de dados, através dele é possível a criação das estruturas de dados de cada tabela, definindo suas propriedades e seus tipos, no próprio JavaScript. Além disso, ele é capaz de gerar essas estruturas de forma automática no servidor do banco de dados, sem a necessidade de fazer isso manualmente. Sendo assim, o Framework Prisma irá prover uma representação visual dentro do código enquanto ainda funciona como uma ponta entre a API e o banco onde os dados estão armazenados.

Por fim, outro Framework já citado anteriormente, Express.js, foi utilizado apenas

como Middleware para o Apollo, de forma a unicamente facilitar as configurações de *Cross-Origin Resource Sharing* (CORS) da API.

4.3.3 Plataforma de Desenvolvimento

Após as definições dos tipos de linguagem que seriam utilizados e dos Frameworks que seriam incluídos, foi o momento de decidir em que plataforma o projeto seria desenvolvido. Com plataforma me refiro ao software utilizado para realizar a programação. Apesar de terem muitas opções utilizadas no mercado, de forma geral o VSCode é um software que, por experiência própria, tem todas as funcionalidades necessárias para facilitar o desenvolvimento da Gennai API. Além de ser gratuito, sua navegação é muito simples e o software é bastante leve. Dessa forma, o software foi escolhido como ferramenta de desenvolvimento.

Outra ponto importante decidido neste momento foi a plataforma de hospedagem da aplicação. Pelos fins acadêmicos iniciais, foi escolhida a plataforma Heroku para realizar o Deploy, hospedando a API. Além disso, vale ressaltar que o projeto foi todo desenvolvimento utilizando a linguagem de versionamento Git, através da plataforma online GitHub.

4.4 Desenvolvimento da API

Com as estruturas de dados e tecnologias definidas, foi o momento de iniciar o desenvolvimento. Essa etapa pode ser dividida em 3 fases:

1. Criação da Estrutura Base;
2. Aplicação das Estruturas de Dados da Gennai API;
3. Processo de Deploy da API;

4.4.1 Criação da Estrutura Base

O objetivo dessa fase foi desenvolver a base da API, ainda sem a introdução das estruturas de dados criadas na subseção 4.2. Apenas conectando os Frameworks de forma a fazê-los funcionar conjuntamente. Sendo assim, o primeiro passo foi a criação de um projeto inicial em *Node.js*, e posteriormente a organização de pastas e arquivos seguindo a lógica do Framework Apollo, como visto na Figura 12.

No Framework Apollo, existem 2 conceitos principais:

- Schema: utilizado para descrever o formato dos dados disponíveis na API; e

Figura 12 – Interface do VSCode com a organização inicial de arquivos e pastas

Fonte: O Autor.

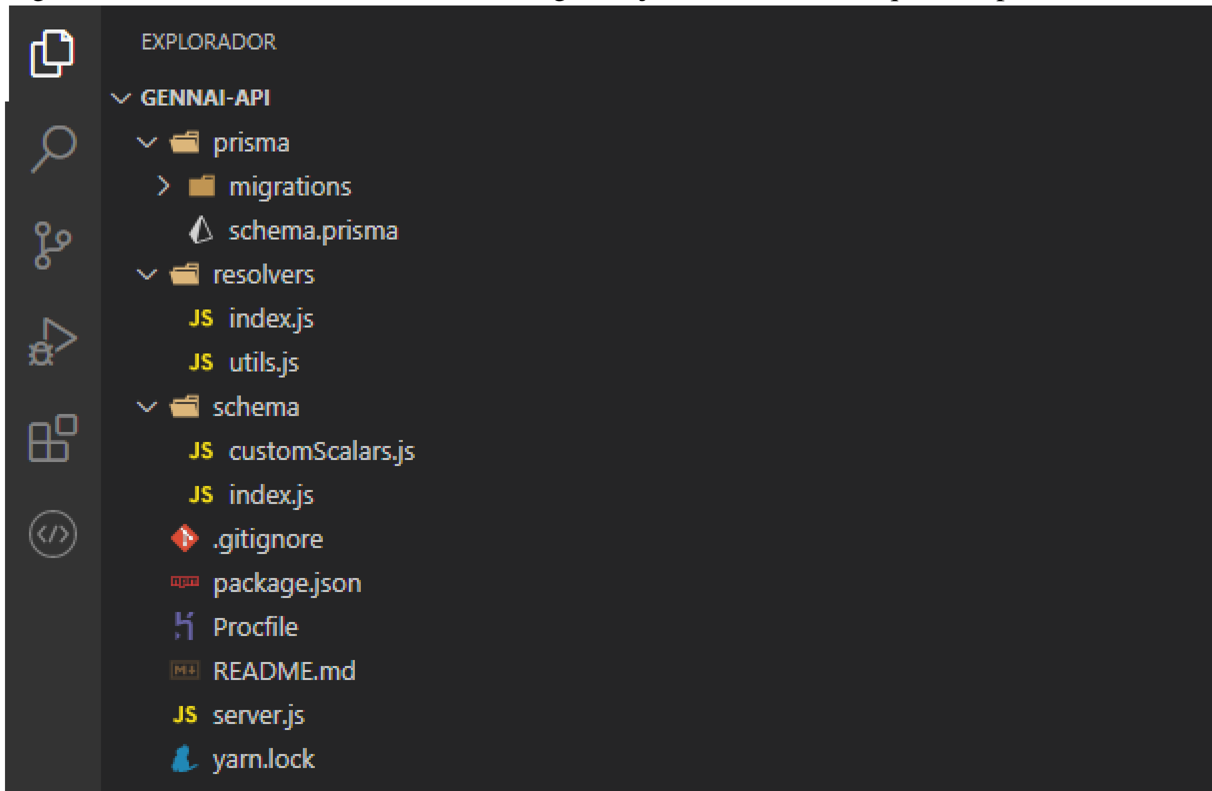
- Resolvers: são funções responsáveis por popular os dados para uma campo específico dentro do schema.

De forma mais simplificada, o *schema* são conjuntos de estruturas feitas em JavaScript, cada uma delas representando uma entidade de dados, como as citadas previamente na Figura 11. Já os resolvers tem como função definir como as diferentes propriedades das estruturas de dados definidas no schema serão adquiridas, é aqui que diremos de onde e como os dados irão vir, um exemplo seria uma query que busca os digimon pela API, nesse caso teríamos um resolver específico para cada campo da estrutura do tipo Digimon, além de também ser o lugar aonde seria feita a busca dos dados em um banco dados.

Dessa forma, é possível ver na Figura 12 que o arquivo "index.js"dentro da pasta "schema"é responsável por conter o schema da API, assim como o arquivo "index.js"dentro da pasta "resolvers"é responsável por conter os resolvers da API. Além disso, o arquivo "server.js"é responsável por conter todos os dados e lógicas envolvendo a criação do servidor que funcionará como uma API, além de realizar a conexão entre os arquivos de schema e de resolvers, finalizando, assim, a lógica de funcionamento do Apollo. Ainda nessa fase, foi feita a adição do Framework Express.js como um Middleware, possibilitando a utilização do formato de configuração de CORS provida pelo mesmo, facilitando e simplicando futuras modificações. O código-fonte do arquivo "server.js"está visível no apêndice A.

Finalizando essa fase, teve-se como objetivo a integração do projeto com o Framework Prisma. Para este Framework, foi apenas necessário mapear as tabelas referentes ao banco de dados, sendo assim apenas um arquivo chamado "schema.prisma"foi adicionado dentro de uma pasta chamada "prisma"na raiz do projeto. Esse arquivo assemelhasse bastante com o

Figura 13 – Interface do VSCode com a organização finalizada de arquivos e pastas



Fonte: O Autor.

arquivo do schema citado anteriormente. O resultado final da organização de pastas e arquivos pode ser visto na Figura 13.

4.4.2 Aplicação das Estruturas de Dados da Gennai API

Após a construção da estrutura do projeto, a próxima etapa é inserir todas as estruturas de dados definidas na seção 4.2. Como foram citados na subseção 4.4.1, os arquivos "index.js"da pasta "schema"e "schema.prisma"da pasta "prisma"foram atualizados com as mesmas estruturas apresentadas na Figura 11, salvo diferenças de sintaxe, representadas no exemplo da Figura 14.

Embora as duas estruturas mostradas na Figura 14 representem o mesmo tipo de dados, fica visível que a utilizada no Framework Prisma requer mais propriedades e campos, já que a mesma é a responsável pelo mapeamento direto dos tipos com o banco de dados. Além dos tipos de dados, é também no arquivo referente ao mapeamento da Apollo que são mapeadas as *Queries* e *Mutations*, citadas na Seção 2.3 e representadas, respectivamente, nas Figuras 15 e 16.

Para finalizar a aplicação dos tipos de dados da Gennai API, o próximo arquivo a ser atualizado foi o referente aos resolvers, citado anteriormente. É a partir dos métodos desenvolvidos no mesmo que a conexão entre as estruturas de dados do Apollo e do Prisma

Figura 14 – Tipo de dados Digimon representado pelo Framework Prisma (à esquerda) e pelo Framework Apollo (à direita)

```
model Digimon {
  id            Int
  slug          String
  name          String
  isJogress     Boolean
  hasXAntibody  Boolean
  rank          Rank
  rankId        Int
  attribute     Attribute
  attributeId   Int
  type          Type?
  typeId        Int?
  digimental    Digimental?
  digimentalId  Int?
  movies        Movie[]
  series        Series[]
  episodes      Episode[]
  fields        Field[]
  spirits       Spirit[]
  partners      Character[]
  previous      Digimon[]
  next          Digimon[]
  groups        DigimonGroup[]
  otherNames    DigimonName[]
}
```

```
type Digimon {
  id: ID!
  slug: String
  name: String
  isJogress: Boolean
  hasXAntibody: Boolean
  rank: Rank
  attribute: Attribute
  type: Type
  digimental: Digimental
  movies: [Movie]
  series: [Series]
  episodes: [Episode]
  fields: [Field]
  spirits: [Spirit]
  partners: [Character]
  previous: [Digimon]
  next: [Digimon]
  groups: [DigimonGroup]
  otherNames: [DigimonName]
}
```

Fonte: O Autor.

Figura 15 – Tipo de dados Query com exemplos

```
type Query {
  # Field
  getFields(options: OptionsInput): [Field]
  getFieldById(id: Int!, options: OptionsInput): Field
  getFieldByName(name: String!, options: OptionsInput): Field
                          .
                          .
                          .
  # Spirit
  getSpirits(options: OptionsInput): [Spirit]
  getSpiritById(id: Int!, options: OptionsInput): Spirit
  getSpiritsByElement(element: SpiritElement!, options: OptionsInput): [Spirit]
  getSpiritElements(options: OptionsInput): [SpiritElement]
}
```

Fonte: O Autor.

se realizam. Assim como no arquivo de estruturas do Apollo, os resolvers existem para cada tipo de dados próprio da Gennai API, assim como para os tipos especiais Query e Mutation, representados, respectivamente, pelas Figuras 17, 18 e 19.

Figura 16 – Tipo de dados Mutation com exemplos

```
type Mutation {
  # Field
  createField(data: FieldInput!): Field!
  updateField(data: FieldInput!): Field!
  deleteField(id: Int!): Field!
                    .
                    .
                    .
  # Spirit
  createSpirit(data: SpiritInput!): Spirit!
  updateSpirit(data: SpiritInput!): Spirit!
  deleteSpirit(id: Int!): Spirit!
}
```

Fonte: O Autor.

Figura 17 – Resolvers para as propriedades do tipo Field

```
Field: {
  digimon: (parent, args, ctx, info) ⇒ prisma.field.findUnique({
    where: {
      id: parent.id
    }
  }).digimon(),
}
```

Fonte: O Autor.

Figura 18 – Resolvers para as propriedades do tipo Query

```
Query: {
  // Field
  getFields: (prt, args, ctx, info) ⇒ prisma.field.findMany({
    ... getOptions(args?.options)
  }),
  getFieldById: (prt, args, ctx, info) ⇒ prisma.field.findUnique({
    where: {
      id: args.id
    }
  }),
  getFieldByName: (prt, args, ctx, info) ⇒ prisma.field.findUnique({
    where: {
      name: args.name
    }
  }),
  // Rank
```

Fonte: O Autor.

4.4.3 Processo de deploy da API

Como citado anteriormente na subseção 4.3.3, a plataforma Heroku foi escolhida para hospedar a API e aproveitando-se da utilização da plataforma de repositórios GitHub, que

Figura 19 – Resolvers para as propriedades do tipo Mutation

```
Mutation: {
  // Field
  createField: (parent, args, ctx, info) => prisma.field.create({
    data: {
      ...args.data,
      digimon: {
        connect: args?.data?.digimon?.map(d => {
          return {
            id: parseInt(d.id)
          }
        })
      }
    }
  }),
```

Fonte: O Autor.

já estava sendo utilizada para versionar o projeto, foi criado um repositório no Heroku de nome "gennai"e conectado ao projeto no GitHub, de forma a facilitar o processo de Deploy ao ponto de necessitar de apenas um clique de um botão.

Sendo assim, com a finalização da estrutura da API, com todas as estruturas de dados implementadas na mesma e com a API funcionando ativamente, a etapa de desenvolvimento foi finalizada.

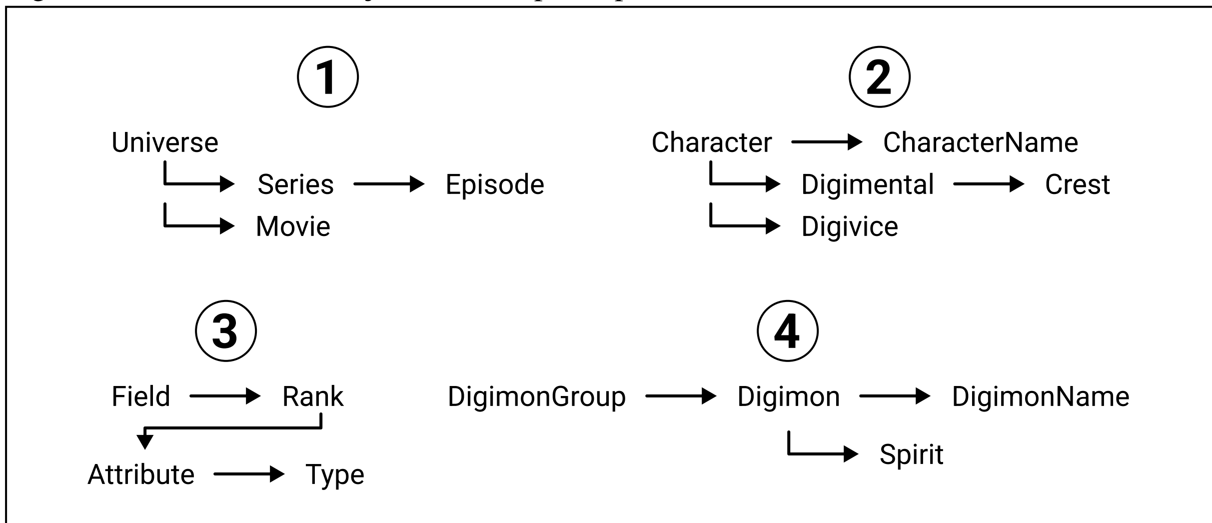
4.5 População dos Dados na API

Apesar da API estar funcionando ativamente, nenhum dado havia sido introduzido até então. Sendo assim, essa etapa foi responsável pela inserção de todos os dados relacionados às estruturas de dados definidas na seção 4.2.

Antes de inserir os dados, fez-se necessário procurar fontes que contribuíssem para a completude da API. Foram escolhidas 2 principais fontes de dados relacionados à franquia Digimon:

- Wikimon: a maior enciclopédia existente de Digimon, é um site desenvolvida por fãs de todo o mundo e ativa desde dezembro de 2005. Atualmente contém mais de 10.000 artigos diversos; e
- Digimon.net: a fonte de dados oficial de Digimon. Embora sua criação seja recente e não tenha muita variedade de tipos de dados, ela provém informações de diversos digimon além de seus complementes como atributos, famílias etc.

Figura 20 – Ordem de inserção de dados pelo tipo



Fonte: O Autor.

Sendo assim, os dados dos tipos Digimon, Rank e Attribute foram adquiridos através dos dados fornecidos pela página de enciclopédia do portal Digimon.net, assim como os dados restantes foram adquiridos através da enciclopédia digital Wikimon.

O processo de inserção de dados na API seguiu uma ordem de quais tipos de dados seriam introduzidos inicialmente baseando-se na lógica envolvendo os mesmos. Essa lógica é apresentada na Figura 20.

Todos os dados foram inseridos usando os métodos previamente criados do tipo *Mutation*, como exemplificado na Figura 19. Por fim, após todos os dados terem sido inseridos no banco de dados, a Gennai API estava oficialmente populada e ativa para ser utilizada.

4.6 Documentação da API

Nesta última etapa do desenvolvimento, iniciou-se o processo de documentação da Gennai API. Aqui foram detalhados todos as estruturas de dados, descrevendo o significado de cada propriedade delas. Para isto, utilizou-se uma Lib chamada *Prisma Documentation Generator* que se utiliza do próprio mapeamento de estruturas de dados feito na parte do Prisma e previamente estabelecido na Subseção 4.4.2. Dessa forma, foi gerada uma página web que mostra todas as entidades com seus respectivos atributos, de forma a deixar clara a estrutura utilizada nas tabelas do banco de dados. Essa primeira documentação é bem detalhada e técnica, não sendo muito adequada para apenas quem vai consumir a API. Uma parte dela pode ser vista na Figura 21. Apesar disso, outra documentação, mais direta e simples, foi desenvolvida tendo seu processo descrito anteriormente.

Figura 21 – Documentação gerada à partir do Framework Prisma

Digimon

Description: Creatures made of data

Fields

Name	Type	Attributes	Required	Comment
id	Int	**@id @default(autoincrement())**	**Yes**	Id of the digimon
slug	String	**@unique**	**Yes**	Slug of the digimon
name	String	**@unique**	**Yes**	Name of the digimon
isJogress	Boolean	**@default(false)**	**Yes**	True if the digimon is a Jogress, false if not
hasXAntibody	Boolean	**@default(false)**	**Yes**	True if the digimon has a X Antibody, false if not
rank	Rank	-	**Yes**	Rank of the digimon
rankId	Int	-	**Yes**	Id of the rank of the digimon
attribute	Attribute	-	**Yes**	Attribute of the digimon
attributeId	Int	-	**Yes**	Id of the attribute of the digimon
type	Type?	-	No	Type of the digimon
typeId	Int?	-	No	Id of the type of the digimon
digimental	Digimental?	-	No	Digimental of the digimon
digimentalId	Int?	-	No	Id of the digimental of the digimon
movies	Movie[]	-	**Yes**	List of movies that the digimon appears
series	Series[]	-	**Yes**	List of series that the digimon appears
episodes	Episode[]	-	**Yes**	List of episodes that the digimon appears
fields	Field[]	-	**Yes**	List of fields that the digimon belongs
spirits	Spirit[]	-	**Yes**	List of spirits that the digimon contains
partners	Character[]	-	**Yes**	List of partners that the digimon already had

Fonte: O Autor.

5 GENNAI NODE

Este capítulo irá descrever, de forma minuciosa, as etapas do processo de criação do Gennai Node. Como dito anteriormente na seção 3.2, o processo de desenvolvimento se deu em 3 etapas:

1. Escolha da Linguagem de Programação
2. Desenvolvimento do Biblioteca
3. Publicação da Biblioteca

5.1 Escolha da Linguagem de Programação

Para encapsular o acesso à Gennai API, decidiu-se pela criação de uma Lib que tivesse como único propósito facilitar a busca de dados da mesma. Sendo assim, também foi necessária a escolha de apenas uma linguagem de programação para servir de objetivo dessa Lib. Aproveitando-se de todo o escopo do projeto da API ter sido feito pela linguagem de programação JavaScript, escolheu-se a linguagem de programação TypeScript para a criação desta Lib.

TypeScript e JavaScript provêm essencialmente as mesmas funcionalidades, com a diferença do TypeScript permitir uma maior organização e estruturação do código através de diversas funcionalidades. A principal característica do TypeScript é a chamada de interface, um tipo de estrutura onde são definidos as propriedades, seus tipos e o que se espera delas. Essa escolha se deu também pelo teor público e de grande acesso que a Lib se propõe a ter, fatores que têm muito a melhorar caso possuam os recursos providos pelo TypeScript. Também foi utilizado o Framework Axios, que tem como objetivo facilitar a criação e configuração de requisições.

Além disso, o nome dado para essa Lib foi "Gennai Node", "Gennai" pois remete à Gennai API e "Node" que remete ao *Node.js*, Framework utilizado para o desenvolvimento da mesma. Também foi neste momento que foi decidido a utilização do NPM para hospedar a Lib e suas versões.

5.2 Desenvolvimento da Biblioteca

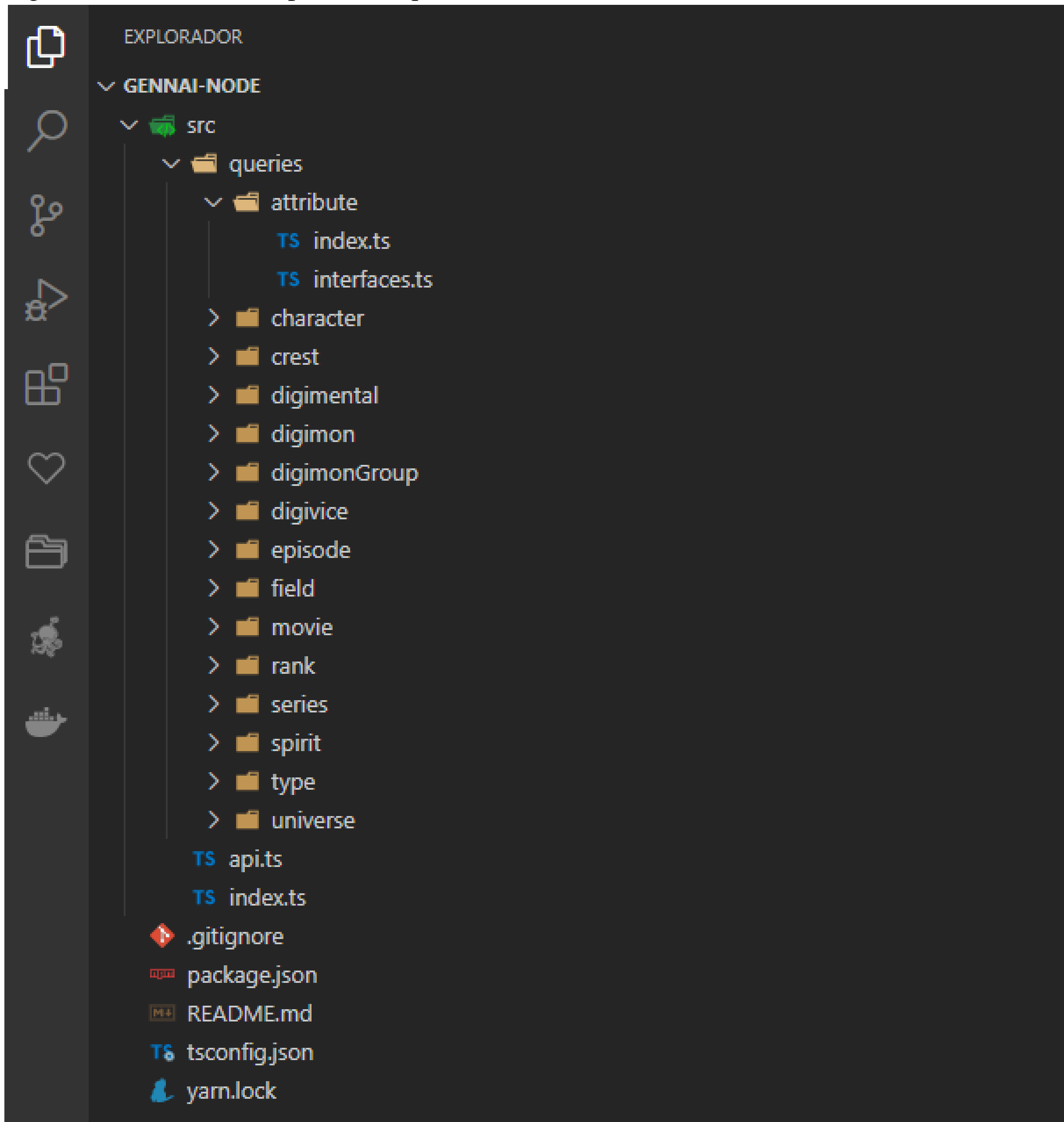
Havia dois objetivos em mente antes do processo de desenvolvimento começar:

1. A Lib tinha que ser leve, ou seja, pesar pouco na memória; e
2. Ser bem descritiva, utilizando-se dos melhores recursos que o TypeScript poderia

prover.

Sendo assim, o primeiro passo foi a estruturação do projeto. Nesse momento ocorreu a criação das pastas e arquivos de forma generalizada, como visto na imagem 22.

Figura 22 – Estrutura de pastas e arquivos do Gennai Node



Fonte: O Autor.

O arquivo "api.ts" tem como função conter todos os dados relacionados à conexão com a Gennai API, sendo assim o usuário só iria se preocupar com que informações ele deseja obter. O código-fonte deste arquivo encontra-se no Apêndice B.

A pasta "queries" contém outras pastas, cada uma representando uma estrutura de

dados existente na Gennai API, com cada uma delas contendo um arquivo “index.ts” e um arquivo “interfaces.ts”. O primeiro tem como função conter todas as *queries* já desenvolvidas na Subseção 4.4.2 referentes ao tipo que a pasta representa, já o segundo é responsável por conter todas as informações referentes às estruturas do tipo da pasta, informações essas que são usadas no arquivo “index.ts” da mesma pasta.

No Apêndice C, é possível inspecionar o código-fonte do arquivo “index.ts” da pasta “attribute”, já no Apêndice D é possível ver o código-fonte do arquivo “interfaces.ts” da pasta “attribute”. A lógica de código é a mesma para todos os outros tipos de dados.

Por fim, o arquivo “index.ts” que se encontra na raiz do projeto é responsável por listar todos os métodos previamente estabelecidos nas pastas de mesmos nomes. Vale pontuar que essa Lib é modular, ou seja, ela não requer nenhuma instanciação de classe, é possível importar apenas o que é necessário pelo desenvolvedor. O código-fonte deste arquivo encontra-se no Apêndice E.

5.3 Publicação da Biblioteca

Após o fim do desenvolvimento do *Gennai Node*, a próxima e última etapa foi a de publicação da Lib. Apesar de ter sido uma etapa bem simples, foi necessária a adição de palavras-chave, informações sobre o repositório do GitHub e escolha do nome que seria registrado na Lib. Por fim, foi feita a publicação da Lib no NPM com o nome de “gennai-node”e associada às seguintes palavras-chave:

- digimon;
- typescript;
- package;
- npm-package;
- graphql;
- api; e
- gennai.

6 PÁGINA COM DOCUMENTAÇÃO

Este capítulo irá descrever, de forma minuciosa, as etapas do processo de criação da página web que serve como documentação para a Gennai API. Como dito anteriormente na seção 3.3, o processo de desenvolvimento se deu em 4 etapas:

1. Coleta de Referências
2. Escolha das Tecnologias
3. Desenvolvimento da Página
4. Publicação da Página

6.1 Coleta de Referências

Essa etapa descreve o processo de coleta de referências utilizadas para a criação da página de documentação da Gennai API.

Antes da começar o desenvolvimento da página web, foi feita uma pesquisa em outras páginas de documentação de APIRDs. As escolhidas para a análise foram a PokéAPI e a *The Rick and Morty* API, já que ambas exercem a mesma funcionalidade que a Gennai API.

No caso da PokéAPI, por ser uma API que aceita tanto REST quanto GraphQL, é possível ver na figura 23 que as estruturas de dados são bem detalhadas com descrições e tipos, incluindo até *endpoints* específicos para cada uma. Mostra também exemplos de respostas de requisições.

Já no caso da The Rick and Morty API, que também é uma API que aceita tanto REST quanto GraphQL, é possível ver na figura 23 que a barra lateral é bem mais simples e objetiva, além das informações estarem mais claras em relação às estruturas de dados, podemos ver até mesmo as respostar das requisições mais bem visíveis.

Essa diferença entre as duas APIs se dá, provavelmente, pela diferença na quantidade de tipos de dados. Enquanto na PokéAPI tem dezenas, a *The Rick and Morty* API tem apenas 3. Partindo desse princípio, e aproveitando o fato de que a API está apenas com suas estruturas iniciais, ou seja, pouca quantidade de tipos de dados, fez muito sentido procurar uma abordagem mais próxima da de *The Rick and Morty* API, e para o futuro, quando houvessem mais tipos de dados, seria necessário apenas incluí-las na documentação.

Figura 23 – Interface da página de documentação da PokéAPI

Contents

Information
Fair Use Policy
Slack
Wrapper Libraries

Resource Lists/Pagination

Berries
Contests
Encounters
Evolution
Games
Items
Locations
Machines
Moves
Pokémon
Abilities
Characteristics
Egg Groups
Genders
Growth Rates
Natures
Pokeathlon Stats
Pokemon
Pokemon Location Areas
Pokemon Colors
Pokemon Forms
Pokemon Habitats
Pokemon Shapes
Pokemon Species
Stats
Types

Utility

slot	Pokémon have 3 ability 'slots' which hold references to possible abilities they could have. This is the slot of this ability for the referenced pokemon.	*integer*
pokemon	The Pokémon this ability could belong to.	*NamedAPIResource* (*Pokemon*)

Characteristics (endpoint)

Characteristics indicate which stat contains a Pokémon's highest IV. A Pokémon's Characteristic is determined by the remainder of its highest IV divided by 5 (gene_modulo). Check out Bulbapedia for greater detail.

GET https://pokeapi.co/api/v2/characteristic/{id}/

id: 1
gene_modulo: 0
▸ possible_values: [] 7 items
▾ highest_stat: {} 2 keys
name: "hp"
url: "https://pokeapi.co/api/v2/stat/1/"
▾ descriptions: [] 1 item
▾ 0: {} 2 keys
description: "Loves to eat"
▾ language: {} 2 keys
name: "en"
url: "https://pokeapi.co/api/v2/language/9/"

☐ View raw JSON (0.383 kB, 26 lines)

Characteristic (type)

Name	Description	Type
id	The identifier for this resource.	*integer*
gene_modulo	The remainder of the highest stat/IV divided by 5.	*integer*
possible_values	The possible values of the highest stat that would result in a Pokémon recieving this characteristic when divided by 5.	list *integer*

Egg Groups (endpoint)

Fonte: PokéAPI Website.

6.2 Escolha das Tecnologias

Nesta etapa foram definidas as linguagens de programação e *frameworks* utilizados para a criação da página web. Assim como na Gennai API e no Gennai Node, o mesmo critério sobre a linguagem de programação vai ser utilizado, sendo TypeScript a linguagem escolhida. Outro fator válido para esta escolha é a grande experiência pessoal nesta linguagem, de forma a acelerar o processo e deixá-lo mais organizado.

Com relação aos frameworks utilizados, foram escolhidos 2 que são bem famosos para o desenvolvimento web:

1. *React.js*: framework mais famoso da linguagem JavaScript/TypeScript, muito utilizado no mercado e de fácil implementação; e
2. *Next.js*: framework desenvolvido em cima do *React.js*. Originalmente era um um framework que tinha como objetivo facilitar o roteamento entre páginas de uma

Figura 24 – Interface da página de documentação da *The Rick and Morty* API

Fonte: *The Rick and Morty API Website*.

aplicação em *React.js*, contudo, hoje em dia é o framework com mais ascensão no mercado.

Além das linguagens de programação, outro ponto pensado foi a plataforma de hospedagem utilizado para fazer *deploy* da página. Utilizou-se serviço da Vercel, a empresa dona também do *Next.js*, o que deixa o processo de deploy extremamente rápido e prático.

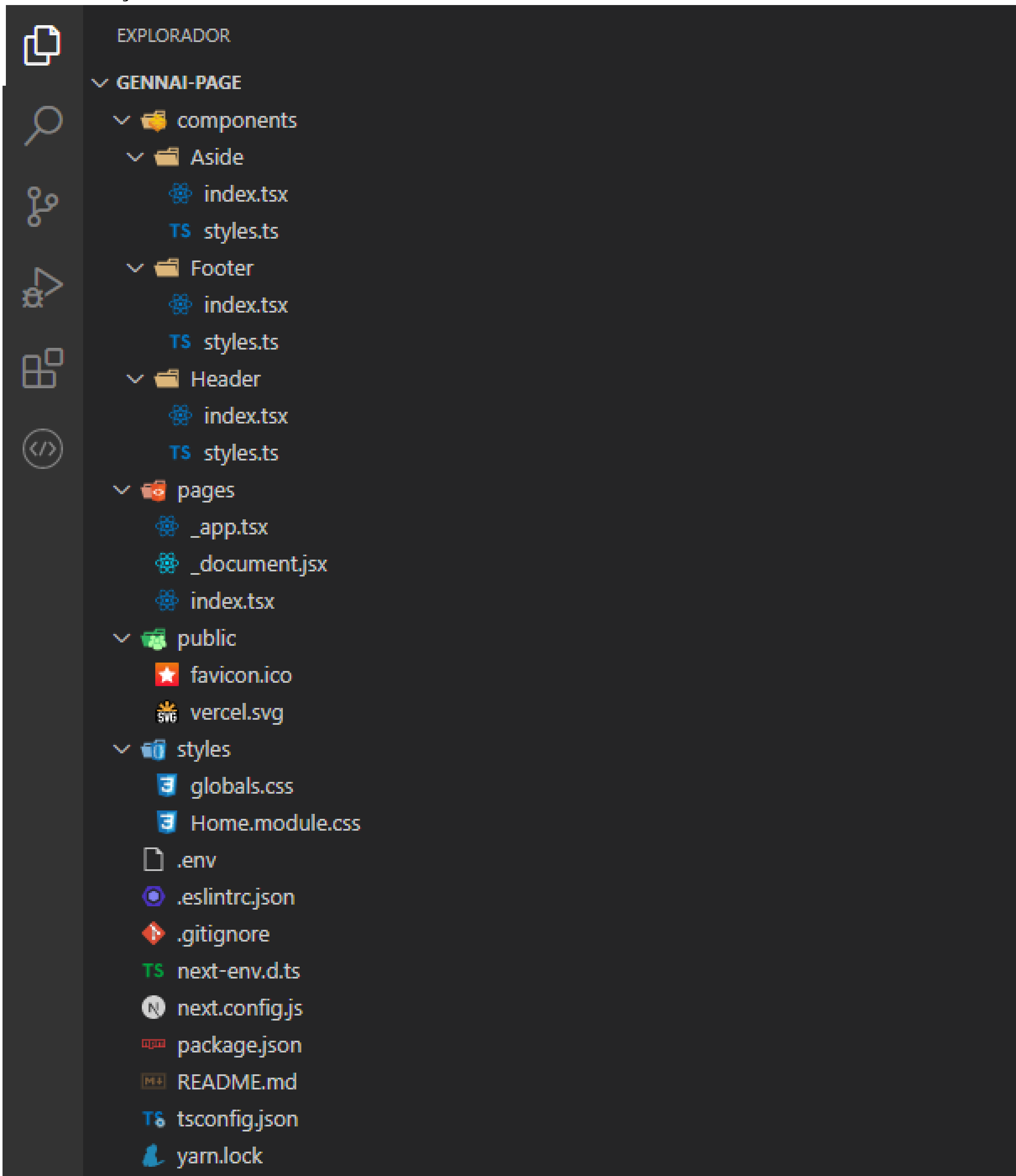
6.3 Desenvolvimento da Página

Nesta etapa será descrito o processo de desenvolvimento da página de documentação da Gennai API. O primeiro passo foi a estruturação do projeto. O próprio *Next.js* já provem uma estrutura inicial para diversos *templates*. Escolhendo o *template* com suporte para TypeScript, uma estrutura básica é gerada, como mostra a figura 25.

Partindo destra estrutura, foi desenvolvida a interface do cabeçalho e barra lateral da página, como mostra a Figura 26.

Por fim, foi desenvolvida a estrutura do conteúdo principal para cada tópico da barra lateral, como mostra a Figura 27.

Figura 25 – Interface do VSCode com a organização inicial de arquivos e pastas da página de documentação da Gennai API

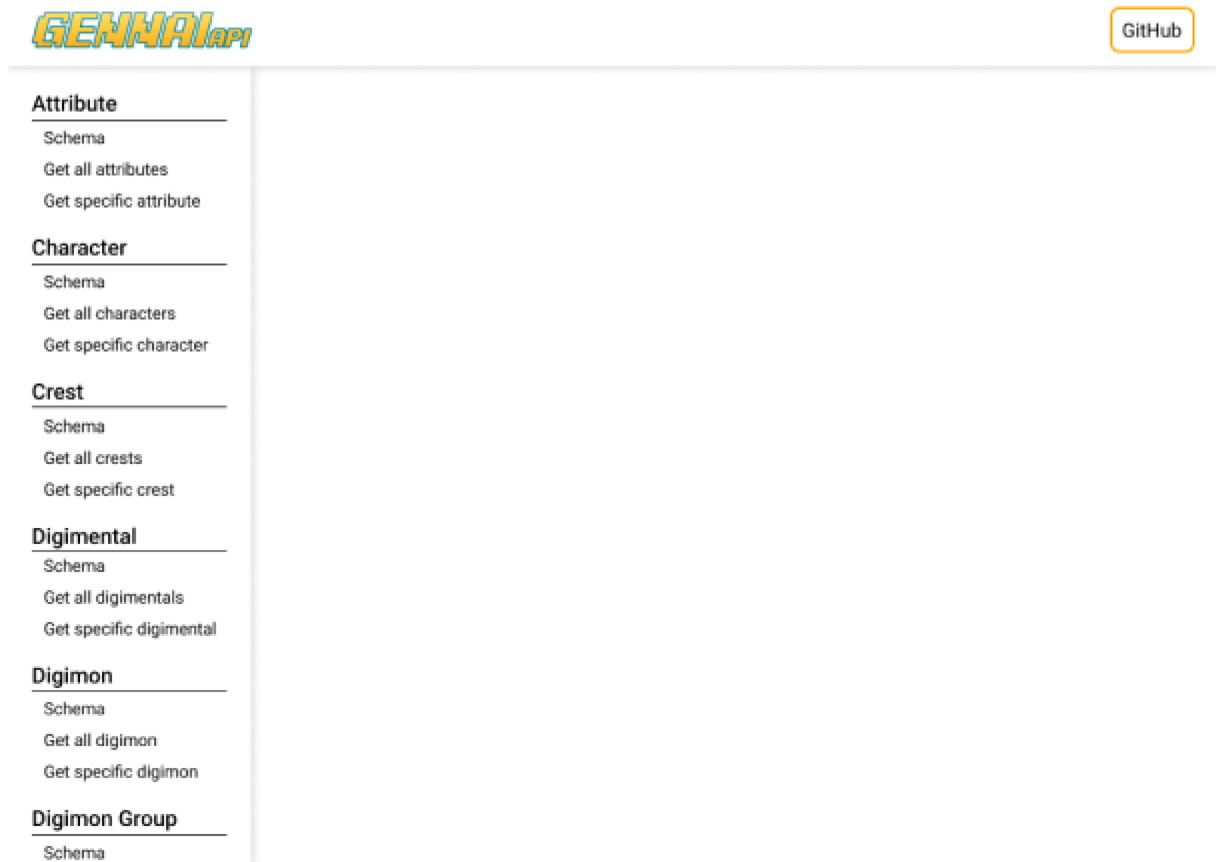


Fonte: O Autor.

6.4 Publicação da Página

Esta última etapa detalha o processo de publicação da página de documentação da Gennai API. Para isto, como dito anteriormente na seção 6.2, foi escolhida a plataforma Vercel para realizar o *deploy*. Para isto foi necessário conectar o repositório no GitHub com o repositório criada na Vercel. Após essa conexão, a própria plataforma já realizou o *deploy* e

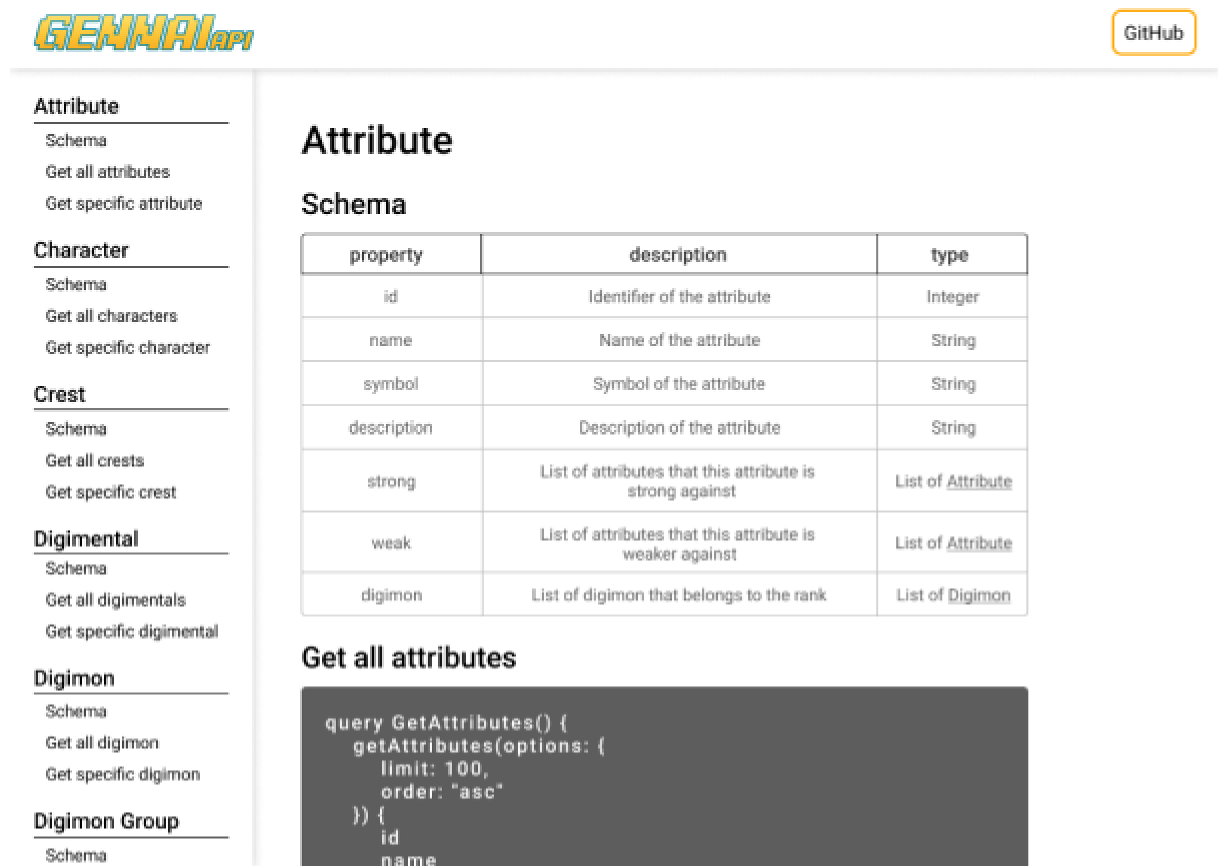
Figura 26 – Página da documentação da Gennai API com cabeçalho e barra lateral



Fonte: O Autor.

deixou a página disponível para acesso.

Figura 27 – Página da documentação da Gennai API completa



Fonte: O Autor.

7 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho iniciou-se como um projeto pessoal, que depois de cerca de 2 anos foi finalmente construído. No início, quando ainda era apenas uma ideia, muitas das tecnologias utilizadas neste trabalho eram desafiadoras e ainda incompreensíveis para mim. Contudo, ao adquirir experiências diversas na área programação durante esses anos, a ideia se transformou em um objetivo.

Além de ter sido ainda muito desafiador, o desenvolvimento da Gennai API permitiu a exploração de um padrão de arquitetura altamente moderno juntamente com as tendências tecnológicas do mercado. Acredito que este trabalho aperfeiçoou e acrescentou em minhas habilidades de programação, permitindo absorver conceitos e padrões que antes não tinha conhecimento. Por fim, a iniciativa resultou em um projeto muito bem estruturado, escalável e reproduzível.

Além da própria API, dois outros produtos foram desenvolvidos: uma *lib* responsável por facilitar o acesso aos dados em um ambiente de desenvolvimento que utilize Node.js; e uma página web que contém a documentação detalhada de cada estrutura de dados da Gennai API. Muito embora houvesse bastante experiência no desenvolvimento para a web, foi minha primeira experiência desenvolvendo uma *lib* a ser publicada no NPM.

Mesmo que ainda seja muito nova, a Gennai API cumpre o que se propôs a fornecer: provê dados estruturas de diversos elementos da franquia Digimon. É um projeto aberto para todos colaborarem ou utilizarem para quaisquer fins, servindo inclusive como referência para outras APIs, como a PokéAPI e o *The Rick and Morty* API foram para este.

Embora existam muitos tipos de dados diferentes providos pela Gennai API, ainda existem muitos mais para serem analisados, catalogados e inseridos na mesma. Sendo assim um projeto que apenas recebeu vida e ainda tem um longo caminho a percorrer em termos de trabalhos futuros, possivelmente com a revisão de conteúdo, inclusão de interfaces para perguntas e respostas (*question answering*), chatbots e acessibilidade para o público em geral.

REFERÊNCIAS

Apigee. **The State of APIs: 2016 Report on Impact of APIs on Digital Business**. 2016. Disponível em: https://pages.apigee.com/rs/351-WXY-166/images/apigee-state-of-APIs-report-2016-03.pdf.

EZ Devs. **GraphQL x REST: Qual utilizar?** 2019. Disponível em: https://ezdevs.com.br/graphql-x-rest-qual-utilizar/s.

IBM. **O que É SOAP?** 2020. Disponível em: https://www.ibm.com/docs/pt-br/integration-bus/10.0?topic=ssmkhh-10-0-0-com-ibm-etools-mft-doc-ac55770--htm.

JACOBSON, D.; BRAIL, G.; WOODS, D. **APIs: A strategy guide**. [*S. l.*]: "O'Reilly Media, Inc.", 2012.

JetBrains. **The State of Developer Ecosystem 2021**. 2021. Disponível em: https://www.jetbrains.com/lp/devecosystem-2021/.

Link API. **Quais são os tipos de APIs?** 2021. Disponível em: https://www.linkapi.solutions/blog/quais-sao-os-tipos-de-apis.

MAIA, J. G. R.; CAVALCANTE-NETO, J. B.; VIDAL, C. A. Crabge: Um motor gráfico customizável, expansível e portável para aplicações de realidade virtual. In: **Proceedings of the VI Symposium on Virtual Reality, SVR2003**. [*S. l.*: *s. n.*], 2003. v. 1, p. 3–14.

Mundo API. **Uma Breve História das APIs com a Pluga**. 2016. Disponível em: https://mundoapi.com.br/materias/uma-breve-historia-das-apis-com-a-pluga/.

Nordic APIs. **GraphQL or Bust**. 2018. Disponível em: https://19yw4b240vb03ws8qm25h366-wpengine.netdna-ssl.com/wp-content/uploads/GraphQL-or-Bust-2018.pdf.

ProgrammableWeb. **Which API Types and Architectural Styles are Most Used?** 2017. Disponível em: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26.

ProgrammableWeb. **APIs show Faster Growth Rate in 2019 than Previous Years**. 2019. Disponível em: https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17.

PURKAYASTHA, S. **The Evolution of APIs: Past, Present and the Future**. 2020. Disponível em: https://blog.api.rakuten.net/evolution-of-apis/.

GLOSSÁRIO

Framework: termo em inglês que significa estrutura, na programação, um framework é um conjunto de códigos genéricos capaz de unir trechos de um projeto de desenvolvimento.

Open Source: termo em inglês que significa código fonte de software aberto.

Lib: abreviação do termo em inglês "library" que significa pacote ou biblioteca.

Brainstorming: termo em inglês que significa uma atividade desenvolvida para explorar a potencialidade criativa de um indivíduo ou de um grupo.

Back-End: termo em inglês que serve para referenciar servidores, bancos de dados, segurança, estrutura, gerenciamento de conteúdo e atualizações, ou seja, sistemas que não são diretamente visíveis para o usuário.

Middleware: termo em inglês que refere-se ao software de computador que fornece serviços para softwares aplicativos além daqueles disponíveis pelo sistema operacional.

Deploy: termo em inglês que o processo de implantação de um software em alguma plataforma.

APÊNDICE A – ARQUIVO SERVER

```
1  import { ApolloServerPluginDrainHttpServer, AuthenticationError } from
   ↪  'apollo-server-core';
2
3  import { ApolloServer } from 'apollo-server-express';
4  import cors from 'cors';
5  import express from 'express';
6  import http from 'http';
7  import resolvers from './resolvers/index.js';
8  import typeDefs from './schema/index.js';
9
10 const startServer = async () => {
11   const app = express();
12
13   app.use('/playground', express.static('playground.html'))
14
15   app.use(cors({
16     preflightContinue: true,
17     credentials: true,
18     methods: "GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS"
19   }))
20
21   const httpServer = http.createServer(app);
22
23   const server = new ApolloServer({
24     typeDefs,
25     resolvers,
26     plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],
27     introspection: true,
28     context: ({ req }) => {
29       if (req.method === 'POST' && req.headers.authorization !==
         ↪  process.env.AUTH) {
```

```
30          throw new AuthenticationError("You do not have permission to use
            ↪  POST method.")
31        }
32      },
33      formatError: (err) => {
34        return {
35          message: err.message,
36          code: err.extensions.code
37        }
38      }
39    });
40
41    await server.start();
42
43    server.applyMiddleware({
44      app,
45      path: '/graphql'
46    });
47
48    await new Promise(resolve => httpServer.listen({ port: process.env.PORT
      ↪  || 8080 }, resolve));
49    console.log(`Server is ready!`);
50  }
51
52  startServer()
```

APÊNDICE B – ARQUIVO API.TS

```
1  import axios, { AxiosError } from "axios";
2
3  interface ApiProps {
4    operationName: string;
5    query: string;
6    variables: {
7      [key: string]: any;
8    };
9  }
10
11 const lowercaseFirstLetter = (word: string) => {
12   return word.charAt(0).toLowerCase() + word.slice(1);
13 };
14
15 const axiosClient = axios.create({
16   baseURL: "http://gennai.herokuapp.com/graphql",
17   headers: {
18     "Content-Type": "application/json;charset=UTF-8",
19     Accept: "*/*",
20   },
21 });
22
23 const api = async (data: ApiProps) => {
24   try {
25     const res = await axiosClient.get("", {
26       params: data,
27     });
28     if (res.status === 200) {
29       return res.data.data[lowercaseFirstLetter(data.operationName)];
30     } else {
31       throw new Error(`Ocorreu um erro com este código: ${res.status}`);
```

```
32      }
33    } catch (_error: any) {
34      let error: AxiosError = _error;
35      return error?.response?.data?.errors;
36    }
37  };
38
39  export default api;
```

APÊNDICE C – ARQUIVO INDEX.TS DA PASTA ATTRIBUTE

```
1  import {
2    AttributeBasicProps,
3    AttributeFullProps,
4    OptionsProps,
5    attributeSchema,
6  } from "./interfaces";
7
8  import api from "../../api";
9  import { digimonSchema } from "../digimon/interfaces";
10
11 export const getAttributes = (
12   options?: OptionsProps
13 ): Promise<AttributeBasicProps[]> =>
14   api({
15     operationName: "GetAttributes",
16     query: `query GetAttributes($options: OptionsInput) {
17       getAttributes(options: $options) {
18         ${attributeSchema}
19       }
20     }`,
21     variables: {
22       options: options,
23     },
24   });
25
26 export const getAttributeById = (id: number): Promise<AttributeFullProps>
   ↪  =>
27   api({
28     operationName: "GetAttributeById",
29     query: `query GetAttributeById($id: Int!) {
30       getAttributeById(id: $id) {
```

```
31          ${attributeSchema}
32          strong {
33            ${attributeSchema}
34          }
35          weak {
36            ${attributeSchema}
37          }
38          digimons {
39            ${digimonSchema}
40          }
41        }
42      }`,
43      variables: {
44        id: id,
45      },
46    });
47
48  export const getAttributeByName = (name: string):
    ↪  Promise<AttributeFullProps> =>
49    api({
50      operationName: "GetAttributeByName",
51      query: `query GetAttributeByName($name: String!) {
52        getAttributeByName(name: $name) {
53          ${attributeSchema}
54          strong {
55            ${attributeSchema}
56          }
57          weak {
58            ${attributeSchema}
59          }
60          digimons {
61            ${digimonSchema}
```

```
62         }
63       }
64     }`,
65     variables: {
66       name: name,
67     },
68   });
```

APÊNDICE D – ARQUIVO INTERFACES.TS DA PASTA ATTRIBUTE

```
import { DigimonBasicProps } from "../digimon/interfaces";

export const attributeSchema = `
  id
  name
  symbol
  description
`;

export interface AttributeBasicProps {
  id: number;
  name: string;
  symbol: string;
  description: string;
}

export interface AttributeFullProps {
  id: number;
  name: string;
  symbol: string;
  description: string;
  strong: AttributeBasicProps;
  weak: AttributeBasicProps;
  digimons: DigimonBasicProps[];
}

export interface OptionsProps {
  offset?: number;
  limit?: number;
  order?: string;
  orderBy?: "id" | "name" | "symbol" | "description";
```

```
32 }
```

APÊNDICE E – ARQUIVO INDEX.TS DA RAIZ DO PROJETO GENNAI NODE

```
1  import {
2    getAttributeById,
3    getAttributeByName,
4    getAttributes,
5  } from "./queries/attribute";
6  import {
7    getCharacterById,
8    getCharacterByName,
9    getCharacters,
10 } from "./queries/character";
11 import { getCrestById, getCrestByName, getCrests } from
   ↪  "./queries/crest";
12 import {
13   getDigimentalById,
14   getDigimentalByName,
15   getDigimentals,
16 } from "./queries/digimental";
17 import {
18   getDigimonById,
19   getDigimonByName,
20   getDigimons,
21 } from "./queries/digimon";
22 import {
23   getDigimonGroupById,
24   getDigimonGroupByName,
25   getDigimonGroups,
26 } from "./queries/digimonGroup";
27 import {
28   getDigiviceById,
29   getDigiviceTypes,
30   getDigivices,
```

```
31    getDigivicesByType,
32  } from "./queries/digivice";
33  import {
34    getEpisodeById,
35    getEpisodeByTitle,
36    getEpisodes,
37  } from "./queries/episode";
38  import { getFieldById, getFieldByName, getFields } from
    ↪  "./queries/field";
39  import { getMovieById, getMovieByTitle, getMovies } from
    ↪  "./queries/movie";
40  import { getRankById, getRankByName, getRanks } from "./queries/rank";
41  import { getSeries, getSeriesById, getSeriesByTitle } from
    ↪  "./queries/series";
42  import {
43    getSpiritById,
44    getSpiritElements,
45    getSpirits,
46    getSpiritsByElement,
47  } from "./queries/spirit";
48  import { getTypeById, getTypeByName, getTypes } from "./queries/type";
49  import {
50    getUniverseById,
51    getUniverseByName,
52    getUniverses,
53  } from "./queries/universe";
54
55  export {
56    getSeriesById,
57    getAttributeById,
58    getAttributeByName,
59    getAttributes,
```

```
60  getSeriesByTitle,
61  getSeries,
62  getCharacterById,
63  getCharacterByName,
64  getCharacters,
65  getDigimentalById,
66  getDigimentalByName,
67  getDigimentals,
68  getCrestById,
69  getCrestByName,
70  getCrests,
71  getDigimonById,
72  getDigimonByName,
73  getDigimons,
74  getDigimonGroupById,
75  getDigimonGroupByName,
76  getDigimonGroups,
77  getDigiviceById,
78  getDigiviceTypes,
79  getDigivices,
80  getDigivicesByType,
81  getTypeById,
82  getTypeByName,
83  getTypes,
84  getEpisodeById,
85  getEpisodeByTitle,
86  getEpisodes,
87  getFields,
88  getFieldById,
89  getFieldByName,
90  getMovieById,
91  getMovieByTitle,
```

```
92    getMovies,
93    getRankById,
94    getRankByName,
95    getRanks,
96    getSpiritById,
97    getSpiritElements,
98    getSpirits,
99    getSpiritsByElement,
100   getUniverseById,
101   getUniverseByName,
102   getUniverses,
103 };
```