PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Ways Studio

Aplicação web para a construção de conteúdos em áudio interativo

Frederico Lacis de Carvalho

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Ciência da Computação

Rio de Janeiro, novembro de 2021

Frederico Lacis de Carvalho

Ways Studio

Aplicação web para construção de conteúdos em áudio interativo

Relatório de Projeto Final, apresentado ao programa Ciência da Computação da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Ivan Mathias Filho

Departamento de Informática

Rio de Janeiro

Novembro de 2021

Agradecimentos

Aos meus pais, por não medirem esforços para me proporcionar uma formação de alta qualidade, além de estarem presentes nos altos e baixos desta jornada.

A minha namorada, Gabriela Costa, por dar apoio, incentivo e me entender como ninguém em cada momento bom ou ruim.

Ao amigo que fiz na graduação, David Jentjens, a quem pude recorrer a todo momento, incluindo este projeto.

Aos meus amigos e colegas de equipe, Pedro Miranda, Marcos Majeveski e Ana Ierusalimschy, por acreditarem no Ways e terem o transformado em realidade.

Ao meu orientador, Ivan, por ter dado toda a atenção e auxiliado em pontos cruciais do projeto.

Resumo

Lacis de Carvalho, Frederico; Mathias Filho, Ivan. **Ways Studio - Aplicação web para a construção de conteúdos em áudio interativo.** Rio de Janeiro, 2021. 49p. Projeto Final de Graduação - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Tendo em vista o crescente mercado de conteúdos de áudio como audiobooks e podcasts, surgiu o aplicativo Ways, que permite que o usuário faça escolhas nos conteúdos de áudio e assim mude o rumo da história. Para construir os conteúdos tocados no Ways, foi desenvolvido o Ways Studio, aplicação web para a construção de maneira visual dos diferentes caminhos que a história pode seguir.

Palavras-chave

áudio, escolhas, interatividade, web, servidor, api, react, node

Abstract

Lacis de Carvalho, Frederico; Mathias Filho, Ivan. **Ways Studio - Web application for building interactive audio content.** Rio de Janeiro, 2021. 49p. Projeto Final de Graduação - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

In view of the growing market for audio content such as audiobooks and podcasts, the Ways application was created, which allows the user to make choices in audio content and thus change the course of history. To build the contents played in Ways, Ways Studio was developed, a web application for the visual construction of the different paths that the story can follow.

Keywords

audio, choices, interactivity, web, server, api, react, node

Sumário

Tabela de Figuras

1. Introdução

Nos últimos anos, o mercado de Audiobooks e Podcasts cresceu consideravelmente, tendo como um grande destaque o Brasil, que segundo o último relatório da Voxnest[1], foi o país que teve maior índice de crescimento na criação de podcasts. Além disso, o mercado de audiobooks global foi avaliado 2.67 bilhões de dólares e com uma projeção de atingir o valor de 14.99 bilhões até 2027, segundo pesquisa realizada pela Grand View Research[2].

Em paralelo a isso, é nítida a força de conteúdos interativos presentes no mercado, que vão desde jogos (cujo mercado global foi avaliado em 162.32 bilhões de dólares em pesquisa divulgada pela Mordor Intelligence[3]) até livros-jogos, que permitem que o leitor faça escolhas dentro da narrativa. Como uma forma de inovação, a Netflix lançou filmes interativos em sua plataforma, o que aumentou o engajamento dos usuários.

Tendo esses dois movimentos de mercado em vista, surgiu o aplicativo iOS gratuito, Ways[4], que traz inovação aos tradicionais podcasts e audiobooks, adicionando interação a eles, permitindo que o ouvinte faça escolhas ao decorrer do conteúdo, podendo alterar completamente o rumo da história.

Porém, enquanto estão sendo investidos milhões na gravação de novos audiobooks e na criação de novos episódios de podcasts, ainda não é possível criar o conteúdo em áudio interativo que é necessário para que a plataforma Ways seja populada. Logo, o objetivo deste projeto será a criação de uma plataforma web que através de uma interface gráfica simples e intuitiva permitirá que criadores de conteúdo criem e publiquem conteúdos de áudio interativo.

O restante desta monografia está organizada da seguinte maneira: na seção 2 (Situação Atual), são dados mais detalhes sobre o aplicativo Ways que originou o projeto, além de fazer uma análise de softwares similares; na seção 3 (Proposta e Objetivos do trabalho) é exposto de forma mais detalhada o escopo do projeto e as tecnologias escolhidas; na seção 4 (Atividades Realizadas) são mencionados estudos necessários para a realização do projeto; na seção 5 (Projeto e especificação do sistema) são detalhados tópicos como levantamento de requisitos, modelagem de dados, prototipagem e arquitetura; a seção 6 (Implementação) apresenta detalhes sobre a implementação, com exemplos e trechos de código; a seção 7 (Avaliação do sistema), exibe como os softwares

desenvolvidos foram testados; e por fim, as conclusões, aprendizados e melhorias para o futuro são apresentadas na seção 8 (Considerações finais e trabalhos futuros).

2. Situação atual

O aplicativo Ways, exibido na figura 1 abaixo, foi desenvolvido por uma equipe multidisciplinar, da qual fiz parte, como um projeto de inovação na Apple Developer Academy PUC-Rio, e está disponível na App Store para iPhone e iPad. O aplicativo introduz no mercado um novo tipo de mídia que foi denominada áudio interativo, que consiste em um grafo onde cada nó possui um trecho de áudio e aponta para outros nós que são as opções do usuário. Este formato ainda não possui uma maneira simples de ser criado, logo, os conteúdos do aplicativo tiveram que ser criados manualmente no banco de dados para tornar o MVP[1] completo.

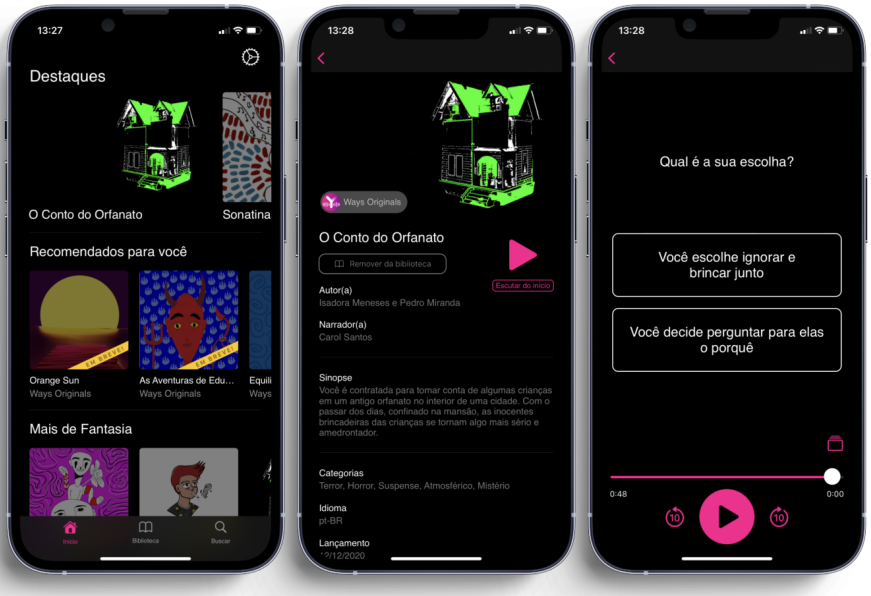

Figura 1: Aplicativo Ways

Este novo tipo de mídia chamou a atenção de novos escritores e narradores que gostariam de produzir conteúdo para o Ways, porém, a falta de

[1] Produto Mínimo Viável (MVP, de *Minimum Viable Product*) é a versão do produto contendo apenas as funcionalidades principais, que pode ser lançado com uma quantidade mínima de esforço e desenvolvimento.

uma ferramenta apropriada para isto tornou este processo inviável até o momento.

A aplicação desenvolvida neste projeto para solucionar este problema será chamada Ways Studio, e apesar de ser parte de uma solução inovadora no mercado de conteúdos de áudio, teve inspiração em algumas ferramentas já existentes:

- Voiceflow[5] - uma ferramenta online que permite que o usuário crie de maneira visual conversas interativas para diversos canais, como skills para Amazon Alexa, actions para Google Assistant, chatbots etc. A interface deste software consiste em uma área de trabalho onde o usuário pode criar caixas de ação e conectá-las, assim, gerando diferentes caminhos. Além disso, o usuário é capaz de organizar estas caixas de uma maneira que a visualização seja clara para o fluxo criado.

- Anchor[6] - aplicação gratuita desenvolvida pelo Spotify para criação, publicação e monetização de podcasts em diversas plataformas. Neste aplicativo, usuários sem experiência em criação de conteúdos em áudio são amparados por uma série de ferramentas em um único lugar. O Anchor possui funcionalidades que vão desde a criação do conteúdo em si, permitindo a gravação e a edição dos trechos de áudio, até sua publicação nos diversos agregadores de podcasts.

O Ways Studio busca unir a interface gráfica com funcionalidades de *drag and drop* e criação de diferentes fluxos oferecida pelo Voiceflow com a facilidade de gestão e publicação de conteúdos oferecida pelo Anchor. Assim tornando a criação de conteúdos em áudio interativo escalável para o aplicativo Ways.

3. Proposta e Objetivos do trabalho

O principal objetivo deste projeto é construir a plataforma de criação de áudios interativos, Ways Studio, para tornar o processo de construção e publicação de novos conteúdos no aplicativo Ways simples e assim torná-lo escalável.

Para atingir este objetivo, serão desenvolvidos dois artefatos de software: o back-end da aplicação, que será uma API RESTful construída utilizando NodeJS[7] e que rodará em um servidor; e o front-end que será construído utilizando ReactJS[8]. Ambos os artefatos foram implementados utilizando a linguagem TypeScript[9]. Este modelo foi escolhido para permitir que no futuro, o aplicativo Ways faça requisições a esta API e tenha acesso aos conteúdos criados no Ways Studio.

3.1. Estrutura do back-end

Por existirem diferentes caminhos, os conteúdos foram organizados segundo um grafo, onde cada nó possui um arquivo de áudio e aponta para as próximas escolhas possíveis. O back-end será capaz de criar e deletar nós, conectar nós diferentes, receber e armazenar os arquivos de áudio associados a um nó específico, alterar a descrição de um nó e das conexões que partem dele.

Todo conteúdo de áudio possui elementos de publicação como título, sinopse, nome do autor e narrador, uma referência ao usuário criador, idioma, data de lançamento e uma imagem de capa. O back-end será capaz de fazer a atualização de cada um destes campos individualmente.

Além de lidar com o grafo que constitui um conteúdo de áudio, a API deverá ser capaz de criar, deletar e atualizar usuários, além de fornecer um método de autenticação seguro.

3.2. Estrutura do front-end

A parte visual da aplicação será responsável por exibir todas as informações relevantes do back-end de maneira intuitiva para o usuário, de maneira que ele possa realizar todas as tarefas mencionadas na seção anterior.

O front-end possui duas páginas principais, a primeira é onde o usuário vê todas as suas histórias criadas e é capaz de criar novas histórias, a segunda, e mais importante, é onde o usuário faz a edição visual do grafo que constitui a história.

A página de edição da história é dividida em três componentes, uma barra no topo que contém ações de navegação e informações relevantes sobre a história, uma barra lateral onde, ao selecionar um nó, são exibidas informações relevantes sobre o nó selecionado, e na área central da página existe um canvas onde o usuário pode editar de maneira visual o grafo, criando novos nós, fazendo conexões com *drag and drop* etc. Além disso, é nesta área que o usuário faz o upload dos arquivos, edita os textos que são exibidos nos momentos de escolha etc.

Esta é uma aplicação comercial, logo, a experiência do usuário é uma prioridade. Para isso, foram seguidos os melhores princípios de design de interfaces aprendidos em disciplinas de interação humano-computador na graduação e em estudos externos. Além disso, a identidade visual do aplicativo Ways foi adaptada para o Ways Studio, por ter sido construída por designers competentes e para fazer com que a marca Ways seja reconhecida em ambas as aplicações.

4. Atividades Realizadas

4.1. Estudos preliminares

Primeiramente, foi preciso aprender a tecnologia escolhida para a implementação do *back-end*, o Node.js. Esta, foi aprendida através do curso Ignite[10] oferecido pela escola de desenvolvimento Rocketseat[11], que além da tecnologia em si, ensina suas boas práticas e a arquitetura que apliquei no projeto, a Clean Architecture[12].

Além disso, foi necessário aprender os conceitos de uma API RESTful e quais são os padrões utilizados no mercado para requisições e respostas HTTP. O curso mencionado ensina o básico sobre o assunto, porém, o aprofundamento foi feito em pesquisas à parte. Logo, foi aprendido a maneira correta de utilizar o *body* e *header* de uma requisição, quais códigos de *status* devem ser retornados de acordo com a situação, como retornar mensagens de erro e em quais situações devem ser usados os diferentes tipos de requisição (*get*, *post*, *patch*, *put* e *delete*).

As requisições mais comuns são as de tipo *get* e *post*. As de tipo *get* sempre são utilizadas quando se está buscando um recurso da API, já as de tipo *post* são utilizadas quando o objetivo é criar um novo recurso no servidor. As requisições do tipo *delete*, sempre são usadas para deletar algum recurso. Os tipos *patch* e *put* são utilizados para atualizar recursos, porém, são os que geram mais confusão, e existem diversas discussões sobre como deve ser feito seu uso. Neste projeto, as requisições *patch* foram utilizadas para atualizar campos de um recurso, enquanto as de tipo *put* foram utilizadas para sobrescrever um recurso por completo.

Em relação à tecnologia utilizada para o desenvolvimento do *front-end,* o ReactJS, havia um conhecimento básico prévio, logo, os estudos foram focados em aspectos arquiteturais, em como fazer requisições à API e receber respostas de maneira correta, em como armazenar dados na memória do *browser* e em como compartilhar dados entre as diferentes telas da aplicação.

4.2. Configuração do ambiente de desenvolvimento

Foi necessário aprender a configurar um ambiente de desenvolvimento que facilite o lançamento da aplicação para algum serviço de computação em nuvem posteriormente. A tecnologia escolhida para tal, foi o Docker[13], um serviço de virtualização que garante que o software será sempre executado no mesmo ambiente.

No Docker, foram criados dois contêineres (que funcionam como máquinas virtuais), um com a API desenvolvida, e um com o sistema gerenciador de banco de dados escolhido, o PostgreSQL. Cada um dos contêineres possuem seu próprio IP e porta de acesso, que tiveram que ser mapeadas para portas do meu IP local para possibilitar o acesso. Isto garante que o ambiente de desenvolvimento será replicável no momento do lançamento, sendo necessário apenas atualizar os ips e mapear as portas.

4.3. Acesso ao banco de dados com ORM

Para facilitar o acesso ao banco de dados, foi utilizada uma biblioteca de ORM (*Object Relational Mapping*), que faz o mapeamento dos dados do banco para objetos em código e também é capaz de fazer o inverso. O framework escolhido foi o TypeORM, por ser um dos mais utilizados na linguagem Typescript e consequentemente possuir uma vasta quantidade de material educativo online.

Ao utilizar esta biblioteca, também foi decidido utilizar a técnica de migrações para o versionamento de banco de dados. Esta técnica, permitiu que as tabelas e colunas fossem criadas incrementalmente durante o desenvolvimento sem a necessidade de fazer *queries* complexas em SQL, uma vez que o TypeORM oferece uma API para que as migrações sejam feitas inteiramente em Typescript. Além disso, a técnica permite que caso haja algum erro decorrente de uma alteração no banco de dados, seja feito um *rollback* de uma ou mais migrações, voltando o banco para um estado confiável.

5. Projeto e especificação do sistema

5.1. Prototipagem

Antes de iniciar o desenvolvimento, em conjunto com o levantamento de requisitos, foi criado um protótipo do fluxo principal da aplicação utilizando a ferramenta de prototipagem Figma[14] que pode ser visto na figura 2 a seguir. O protótipo auxilia no front-end pois serve como um guia para o desenvolvimento, tanto para a estilização dos componentes quanto para a disposição dos elementos em tela de maneira clara e intuitiva, permitindo que o desenvolvimento seja feito de maneira mais ágil.

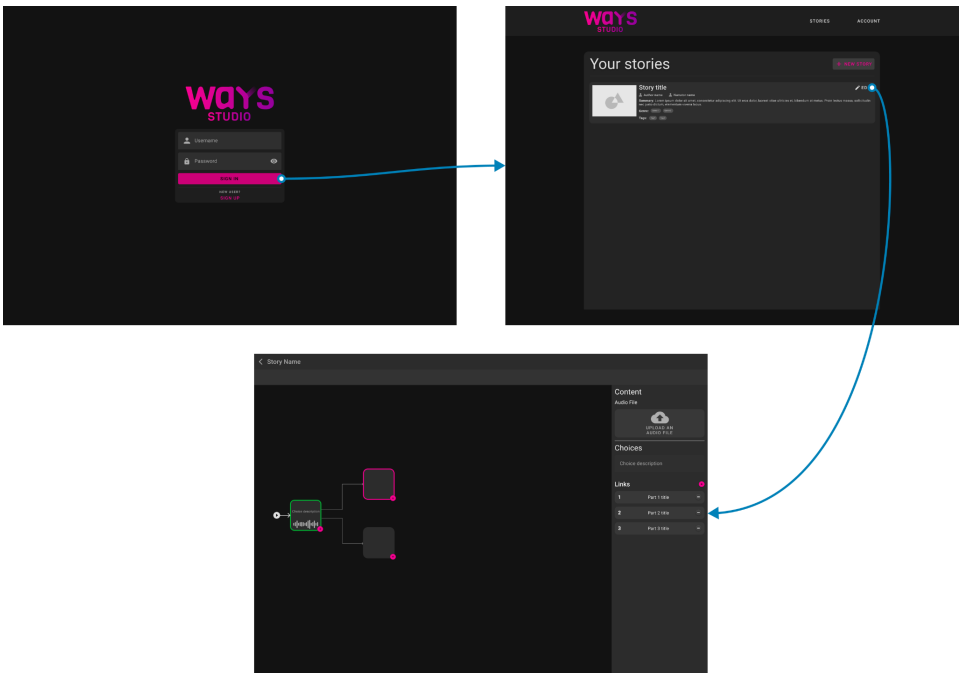

Figura 2: Protótipo visual

Além disso, o protótipo auxiliou no levantamento de requisitos para que nenhuma funcionalidade importante fosse esquecida. O protótipo consolida as ideias em algo palpável, o que auxilia na previsão de problemas e em uma definição mais específica do escopo. Dito isto, apesar de visual, o protótipo também auxilia no desenvolvimento do back-end por deixar mais claro o objetivo final.

Nesta etapa, foi decidido trabalhar com uma biblioteca de componentes de interface consolidada no mercado, para garantir que a aplicação fosse visualmente agradável, apresentasse boa usabilidade e também para acelerar o

desenvolvimento. A biblioteca escolhida foi o MaterialUI[15] que é amplamente utilizada no mercado, incluindo grandes empresas como: Netflix, Spotify, Amazon etc. Além de estar disponível para a tecnologia de front-end escolhida, o ReactJS, a biblioteca também oferece um pacote gratuito de elementos para prototipagem, logo, nesta etapa já ficou claro quais componentes seriam utilizados no momento do desenvolvimento.

Apesar do protótipo servir como um norte estético para o produto final, durante o desenvolvimento uma série de decisões de implementação foram tomadas que o tornaram levemente diferente.

5.2. Requisitos

O levantamento de requisitos é o ponto de partida para o desenvolvimento de qualquer software, e neste projeto foi feito em conjunto com a prototipagem, pois a visualização do projeto fez com que fosse possível levantar requisitos mais precisos.

Como será detalhado na seção 5.4, a arquitetura escolhida para o desenvolvimento do back-end é fortemente orientada a casos de uso, logo, a definição de requisitos também foi dividida desta forma. A figura 3 exibida a seguir mostra o exemplo de caso de uso de autenticação de usuário com seus requisitos funcionais e regras de negócio.

Módulo	Caso de Uso	Tipo	Descrição
Accounts	Autenticação de usuário	Funcional	Deve ser possível autenticar um usuário
		Regra de Negócio	Não deve ser possível autenticar um usuário sem 'username' ou 'password' preenchidos
		Regra de Negócio	Não deve ser possível autenticar um usuário cujo 'username' não está presente no banco de dados
		Regra de Negócio	Só deve ser possível autenticar um usuário se a senha cadastrada para o 'username' dado forem idênticas
		Funcional	Toda senha armazenada no banco de dados deve ser criptografada
		Funcional	Toda senha enviada via requisições HTTP deve ser criptografada

Figura 3: Requisitos do caso de uso de autenticação de usuário

Este modelo possibilitou entender ainda antes do desenvolvimento quais módulos seriam necessários, pois foi possível ver quais casos de uso se agrupavam semanticamente. A lista completa de requisitos do sistema se encontra no Apêndice A.

5.3. Modelagem de Dados

A modelagem de dados foi adaptada do aplicativo Ways, que originalmente utilizava um banco de dados orientado a documentos, e apenas consumia os dados, nesta aplicação, o Ways Studio, é utilizado um banco de dados relacional, e por isso, a modelagem de dados foi refeita levando em conta seus conceitos, que na figura 4 a seguir, são representados como tabelas relacionais.

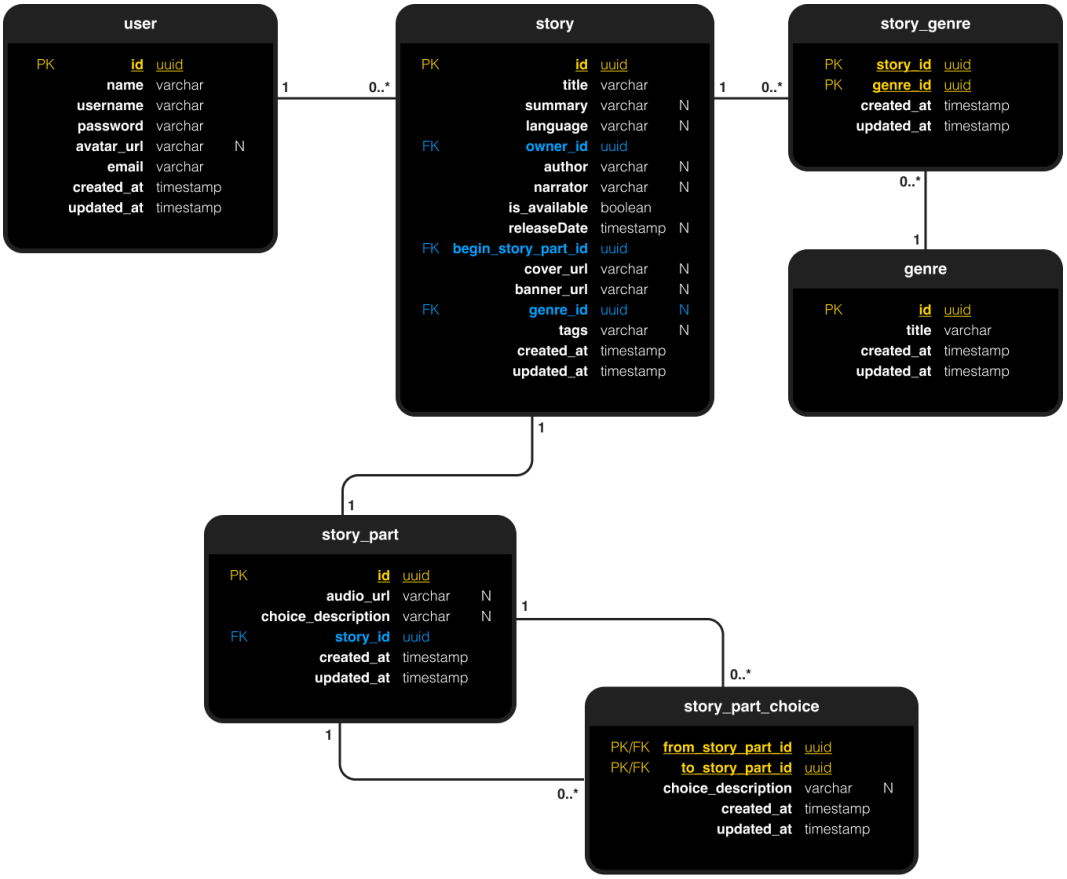

Figura 4: Modelo Relacional

A modelagem conta com uma tabela de usuário, que armazena dados básicos como nome, nome de usuário, senha etc. Além disso, na parte central, existe uma tabela que armazena dados gerais sobre a história, como título, sumário, nome do autor, etc. A história pode possuir diversos gêneros predefinidos, o que gera um relacionamento muitos para muitos, e para isso foi feita uma tabela de relacionamento.

A parte principal da modelagem está na parte da história (story_part) e na escolha (story_part_choice), que é um auto-relacionamento de muitos para

muitos onde a tabela de relacionamento armazena a descrição da escolha que aponta para a próxima parte. Este modelo gera um grafo direcionado onde as escolhas (story_part_choice) são as arestas e as partes (story_part) são os nós.

5.4. Arquitetura

Como mencionado anteriormente, o projeto possui dois módulos independentes: o back-end que é uma API RESTful desenvolvida com NodeJS e o front-end que é uma interface web desenvolvida em ReactJS. O desacoplamento completo entre a API e a parte visual é ideal para o projeto, pois permite que o back-end seja consumido por outro aplicativo no futuro sem a necessidade de mudanças. A comunicação entre a API e o front-end é feita via requisições HTTP, como visto na figura 5 abaixo, retornando em sua maioria respostas no formato JSON.

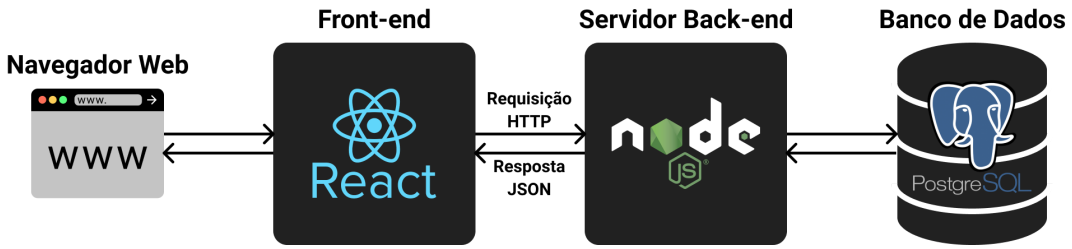

Figura 5: Arquitetura geral do sistema

Tanto no back-end quanto no front-end, foram seguidos os princípios de desenvolvimento SOLID[16], que consiste em cinco princípios na programação orientada a objetos que facilitam o desenvolvimento de software, tornando-os mais testáveis, melhorando sua manutenibilidade e facilitando sua extensão.

5.4.1. Back-End

O back-end foi desenvolvido baseado nos conceitos da *Clean Architecture*[17], onde um dos principais objetivos é criar camadas desacopladas e substituíveis para que o desenvolvimento não se torne dependente dos frameworks que estão sendo utilizados. Por exemplo, o acesso ao banco de dados é feito utilizando o framework TypeORM, porém, toda essa lógica fica armazenada na camada de repositórios e pode ser facilmente substituído por um repositório que utilize outro framework desde que que siga a mesma interface.

Isto caracteriza o *Open-Closed Principle* do SOLID, que sugere que todo comportamento extensível deve ser colocado atrás de uma interface.

Outro forte princípio da arquitetura escolhida é o DDD[18] (Domain-driven Design), que se baseia na divisão do código em domínios semânticos, denominados na arquitetura por módulos. Por exemplo, a aplicação possui dois módulos principais: um para lidar com tudo relacionado às histórias e outro para lidar com tudo relacionado a usuários.

O diagrama exibido na figura 6 abaixo mostra como os módulos são divididos e o que eles encapsulam. Todo módulo possui um roteador local, e um conjunto de casos de uso e repositórios, que serão detalhados a seguir.

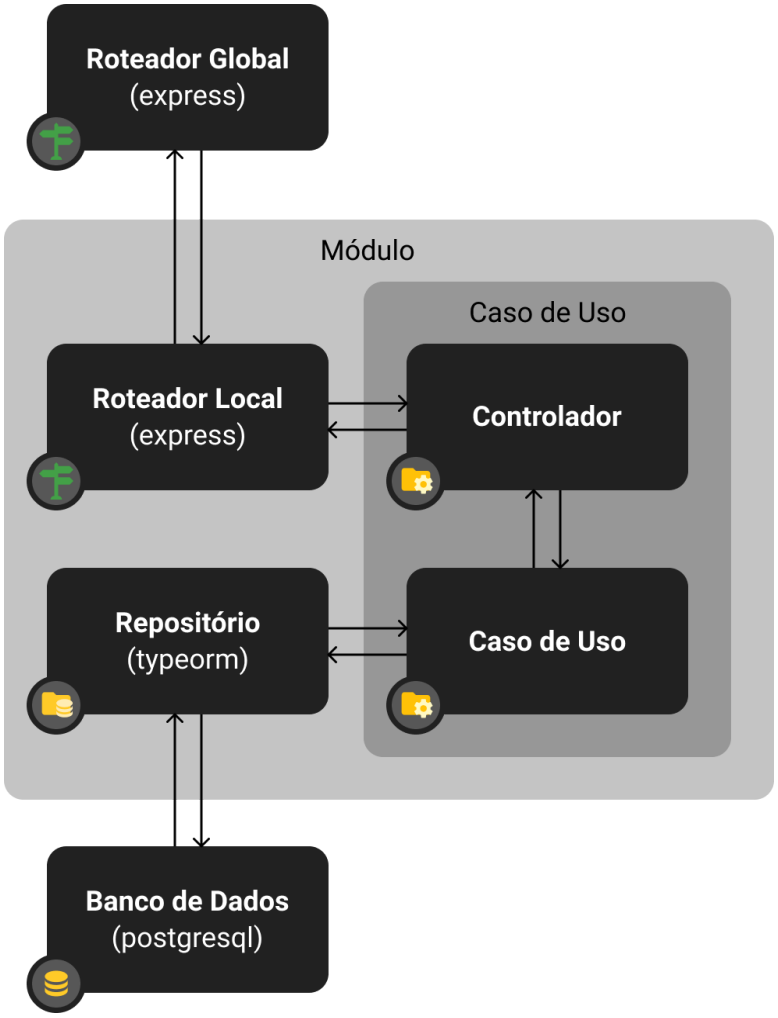

Figura 6: Arquitetura do back-end

O roteador global é o ponto de entrada das requisições HTTP e tem a responsabilidade de enviá-las para os roteadores locais de cada módulo, as

requisições então, passam por *middlewares* antes de serem enviadas para o controlador e então para o caso de uso.

Middlewares são processos intermediários para garantir o tratamento dos dados. Por exemplo, a aplicação possui um *middleware* amplamente utilizado para garantir que a requisição feita possui um token de autenticação válido que aponta para um usuário existente. Apesar dos controladores possuírem uma responsabilidade parecida no tratamento dos dados, os middlewares abstraem funções que seriam repetidas pelo código.

Dentro de um caso de uso, possuímos seu controlador e o caso de uso propriamente dito. O controlador é responsável por extrair os parâmetros da rota, que podem estar no cabeçalho, no corpo ou na URL da requisição feita. Além disso, faz o tratamento dos dados extraídos e verifica se cumprem os requisitos do caso de uso, caso os dados sejam aceitos, o controlador instância e executa o caso de uso com os dados e captura seu retorno para encaminhá-lo novamente ao roteador que enviará esta resposta a quem fez a requisição.

No caso de uso, é onde fica toda a regra de negócio da aplicação. Existe um caso de uso para cada funcionalidade, por exemplo: criar usuário, deletar usuário, conectar duas partes da história, atualizar a descrição de uma escolha etc. Estes, são responsáveis por chamar funções nos repositórios, que acessam o banco de dados, para então aplicar a lógica nos dados retornados.

Os repositórios são conjuntos de abstrações de ações comuns de acesso aos dados, por exemplo, um repositório de usuários já possui funções capazes de buscar um usuário pelo nome, buscar um usuário por e-mail ou até criar um usuário, cada uma destas funções encapsula toda a lógica de conexão e acesso aos dados, tornando os casos de uso muito mais simples.

5.4.2. Front-End

O front-end foi desenvolvido seguindo os princípios da arquitetura MVC, porém, com uma quantidade mínima de processamento sendo feita no browser, uma vez que as rotas desenvolvidas no back-end já suprem quase todas as suas necessidades.

O diagrama na figura 7 a seguir mostra como as camadas do front-end são divididas. Olhando para o padrão MVC (*Model View Controller*), podemos

entender as páginas como as *Views*, que são responsáveis por receber as ações dos usuários e realizar ações a partir disso. API Services e Providers são os *Controllers* da aplicação, pois são responsáveis pela lógica que é aplicada aos dados e pelas requisições que são feitas ao servidor. Os dados retornados das chamadas à API respeitam formatos pré-definidos em interfaces, estas, que representam o *Model* da arquitetura.

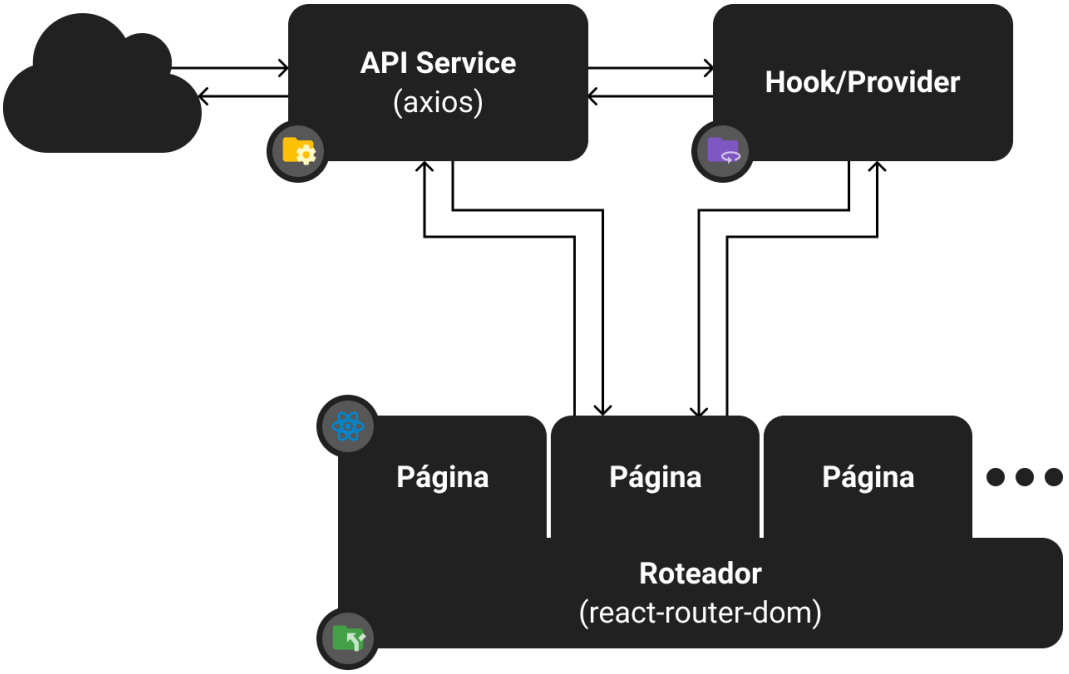

Figura 7: Arquitetura do front-end

No ReactJS, todos os elementos que aparecem em tela são componentes, por exemplo: uma página completa é um componente, um elemento de texto é um componente, um menu é um componente, etc. Desta forma, foram aplicados os princípios SOLID também na camada visual do sistema, de forma que fossem criados componentes com responsabilidades únicas, o que facilita e acelera o desenvolvimento, pois podemos reutilizar estes componentes em mais telas.

O API Service é a classe responsável por fazer as requisições ao back-end. Este, possui funções para os diferentes tipos de requisições, podendo inserir parâmetros no cabeçalho, corpo ou na URL destas. Além disso, a classe é capaz de receber um tipo genérico e então converter a resposta da API ao modelo escolhido se possível.

Os Providers, também chamados de Hooks, são classes que armazenam dados e funções que precisam estar disponíveis em diversos componentes do sistema. Diferente do API Service, os Providers aplicam lógica aos dados antes de os enviarem ao componente. Além disso, são capazes de gerenciar estados da aplicação, logo, se um componente fizer alterações aos dados de um Provider, os outros componentes que o utilizam serão notificados e consequentemente atualizados.

5.4.3. Estruturação das rotas da API

Como mencionado na seção 4.1, foram estudados os conceitos e as boas práticas para a construção de APIs RESTful. Parte deste estudo foi focado em como criar as URLs seguindo os padrões do mercado. Com isso, as URLs da API possuem sempre um significado semântico e funcionam de forma hierárquica.

Podemos entender o padrão escolhido para as URLs como dividido por coleções. Por exemplo, na URL "url_base/stories" o primeiro item é a URL base da API, que durante o desenvolvimento foi utilizada "http://localhost:4444", e o segundo item(stories) identifica a coleção que está sendo solicitada. Para o exemplo dado, quando chamado como uma requisição do tipo *get*, será retornada a lista completa de histórias do usuário que fez a requisição, caso ele esteja autenticado.

Como dito anteriormente, as URLs seguem um modelo hierárquico de coleções, então podemos tornar as requisições ainda mais específicas adicionando mais itens à URL. Por exemplo, adicionando um identificador de uma história válida na URL apresentada no parágrafo anterior, teremos "url_base/stories/906c44b2-bb8d-442d-9856-40bbf23076ee", que agora, ao ser chamada em uma requisição do tipo get, retornará a história com o identificador passado.

Seguindo a hierarquia das histórias, dentro de uma história específica podemos acessar suas partes. Por exemplo, a URL "url_base/stories/906c44b2-bb8d-442d-9856-40bbf23076ee/part/b13baed2-961c-4825-9398-ffe8ef750580" chamada em uma requisição do tipo *put* com o parâmetro "choice_description" em seu corpo, atualizará a descrição da parte com o identificador passado.

Este modelo funciona bem para acessar itens não muito profundos na hierarquia de coleções, porém, as URLs poderiam crescer muito ao buscar por itens mais específicos. Por isso, algumas requisições fogem do padrão adicionando parâmetros no corpo da requisição. Como é o caso da requisição que cria uma nova escolha entre duas partes da história, que funciona da seguinte forma: a requisição feita é do tipo *post* e possui URL "url_base/stories/906c44b2-bb8d-442d-9856-40bbf23076ee/choice", assim, estamos acessando a coleção de *choices* dentro de uma história específica e os identificadores das partes que serão conectadas por esta escolha são passados no corpo da requisição, seguindo o modelo exibido na figura 8 a seguir.

```
{
  "from_part_id": "b13baed2-961c-4825-9398-ffe8ef750580",
  "to_part_id": "9a38a556-146f-488f-bd29-6de6570248a1"
}
```

Figura 8: Corpo da requisição para criação de escolhas

6. Implementação

A implementação foi dividida em três etapas: na primeira, foram implementados os casos de uso principais no back-end para que existisse uma base de requisições prontas; na segunda, foram desenvolvidas as telas, componentes e providers do front-end; e por fim, na terceira, o desenvolvimento da parte visual e do servidor foram feitos em paralelo.

6.1. Autenticação

O primeiro módulo desenvolvido no back-end foi o módulo de usuários, chamado de "accounts", este, foi o primeiro por influenciar diretamente em todo o resto do desenvolvimento. O módulo possui quatro casos de uso sendo o principal o que cria uma sessão de *login* para o usuário.

A autenticação foi implementada utilizando JWT[19] (*JSON Web Token*), que é um método padrão da indústria para realizar autenticação entre duas partes através de um token assinado que autentica a requisição feita. A assinatura mencionada consiste em uma string única conhecida apenas pelo servidor.

O JWT é dividido em três partes, como pode ser visto na figura 9: a primeira é o Header, que guarda qual é o algoritmo de criptografia utilizado e qual o tipo do token; a segunda é o Payload e é onde são armazenadas as informações referentes à própria autenticação, que neste caso são a data de criação do token, sua data de validade e um *subject,* que foi de o identificador do usuário autenticado; a terceira e última parte é a assinatura, que é construída a partir do conteúdo das duas partes anteriores com a chave secreta definida pela aplicação.

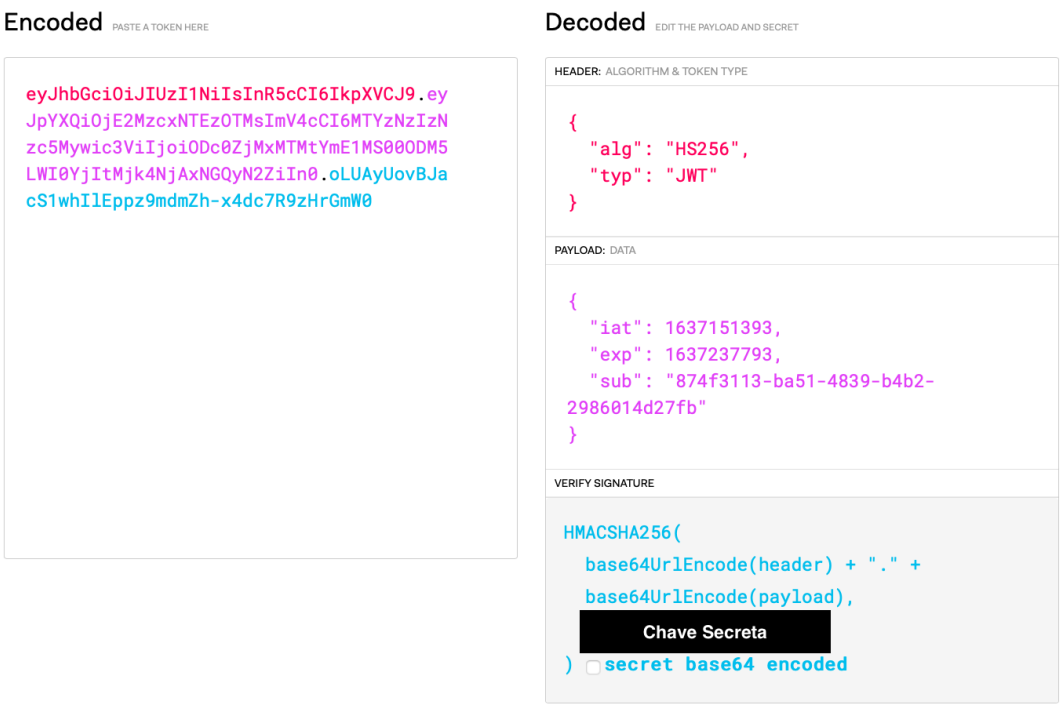

Figura 9: JWT decodificado

Na prática, é feita uma requisição cujo corpo contém o e-mail e senha do usuário e caso as informações sejam corretas, é retornada uma resposta JSON com o objeto completo do usuário e o token gerado, como pode ser visto na figura 10 abaixo.

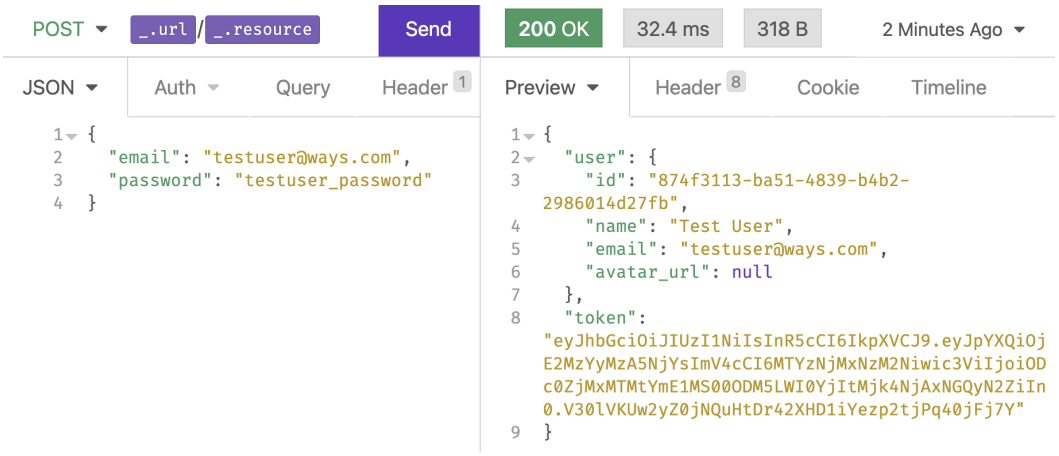

Figura 10: Requisição de autenticação

O token retornado deve ser armazenado pela aplicação cliente para ser incluído em requisições futuras que dependam de autorização. O token nas novas requisições deve ser enviado como Bearer Token[20] no campo "Authorization" no cabeçalho da requisição.

Vale ressaltar que tanto o JWT quanto o Bearer Token são tecnologias amplamente utilizadas e validadas pelo mercado e que foram padronizadas pelo IETF[21] (Internet Engineering Task Force). Sendo o JWT documentado no código RFC-7519[22] e o Bearer Token no código RFC-6750[23].

6.2. Middlewares

Os middlewares são partes essenciais do fluxo de dados da aplicação, pois, como dito anteriormente, são processos intermediários que aplicam lógica aos dados recebidos na requisição antes do seu envio para os casos de uso, como pode ser visto na figura 11 a seguir.

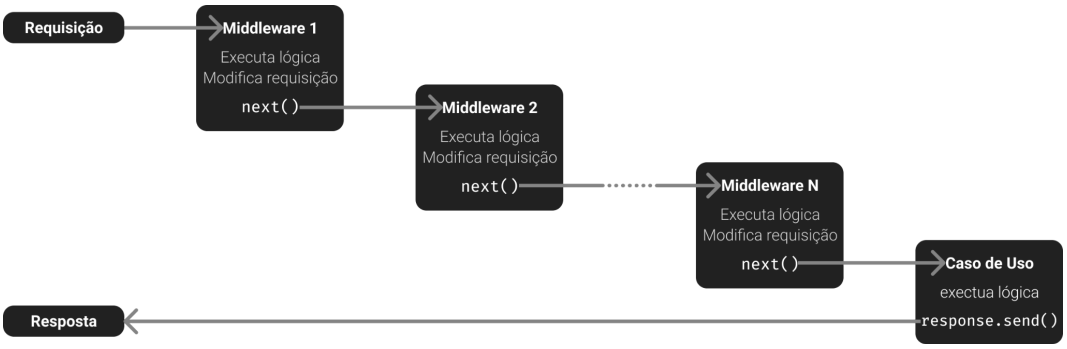

Figura 11: Fluxo de requisições com middlewares

No NodeJS em conjunto com o Express (biblioteca utilizada para controlar requisições HTTP), existe um fluxo de dados predefinido que passa inicialmente pelo roteador, então passa por uma sequência de middlewares opcionais, e então é enviado para o caso de uso.

Na aplicação desenvolvida, os middlewares foram utilizados para abstrair validações que seriam muito repetitivas. Um exemplo é o middleware "ensureAuthenticated", que automatiza o processo de verificação do token. Este, pode ser visto na figura 12 a seguir e primeiramente verifica se a requisição possui um token. Caso possua, o extrai e verifica sua validade, se o token for válido, o middleware é responsável por buscar o usuário no repositório e inseri-lo no corpo da requisição, assim, os próximos middlewares e o caso de uso poderão acessar o usuário através da própria requisição.

```
export async function ensureAuthenticated(
  request: Request,
  response: Response,
  next: NextFunction
): Promise<void> {
  const authHeader = request.headers.authorization;

  if (!authHeader) {
    throw new AppError("Token missing", 401);
  }

  // Formato do token → "Bearer <token>"
  const [, token] = authHeader.split(" ");

  try {
    const { sub: user_id } = verify(
      token,
      Chave secreta
    ) as IPayload;

    const usersRepository = new UserRepository();
    const user = usersRepository.findById(user_id);

    if (!user) {
      throw new AppError("User does not exists!", 401);
    }

    request.user = {
      id: user_id,
    };

    next();
  } catch {
    throw new AppError("Invalid token", 401);
  }
}
```

Figura 12: Exemplo de implementação de middleware

6.3. Dados

Referente ao armazenamento de dados, existem alguns tópicos importantes na implementação que devem ser abordados: o primeiro é a criação das tabelas, suas colunas e relações; o segundo é como cada tabela é mapeada

para uma entidade que pode ser utilizada em código; e por fim, como o acesso ao banco de dados é feito.

Como dito na seção 4.3, a criação das tabelas do banco de dados foi realizada utilizando a técnica de migrações, por meio da biblioteca TypeORM, que permite que as migrações sejam construídas diretamente em Typescript, sem a necessidade de escrever códigos SQL. Toda migração é composta necessariamente por duas partes: a de criação, chamada de *up*; e a de destruição, chamada de *down*. Isto é necessário pois as migrações são reversíveis.

Migrações não funcionam apenas para a criação de tabelas, mas também para alterações de maneira geral. Um exemplo de migração feita, foi a migração que adiciona os campos de coordenadas do canvas às partes da história, e pode ser vista na figura 13 a seguir.

```
export class AddCoordinatesToStoryPart1632927398114
  implements MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.addColumns("story_part", [
      new TableColumn({
        name: "canvas_x",
        type: "numeric",
        default: "0.0",
      }),
      new TableColumn({
        name: "canvas_y",
        type: "numeric",
        default: "0.0",
      }),
    ]);
  }

  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.dropColumn("story_part", "canvas_x");
    await queryRunner.dropColumn("story_part", "canvas_y");
  }
}
```

Figura 13: Exemplo de migração

Apesar de fornecer a funcionalidade de migrações, a biblioteca tem seu foco no mapeamento das tuplas do banco de dados para objetos Typescript, e isto é feito por meio da definição de entidades. Uma entidade do TypeORM é uma classe que possui propriedades especiais que definem como o mapeamento deve ser feito.

Toda entidade deve possuir ao menos uma chave primária, como é a regra dos banco de dados relacionais, e a propriedade que a representa é indicada pelo marcador "@PrimaryColumn()" no código. As propriedades da classe referentes a colunas comuns da tabela são indicadas pelo marcador "@Column()". Uma característica importante deste processo é que caso a propriedade possua o mesmo nome da coluna no banco de dados, apenas o marcador é o suficiente, caso contrário, o marcador precisaria receber um parâmetro indicando o nome correto.

A parte que facilita muito desenvolvimento, é a definição das relações dentro das entidades, que faz com que os objetos já sejam mapeados incluindo as colunas para as quais suas chaves estrangeiras apontam. O TypeORM já fornece marcadores para os diferentes tipos de relações possíveis, como "@OneToOne()", "@ManyToMany()", "@ManyToOne()" etc. Um exemplo da aplicação destes conceitos pode ser observado na figura 14 abaixo, onde foi definida a entidade de uma escolha da história, esta é uma tabela relacionamento que conecta as partes para a geração do grafo.

```typescript
@Entity("story_part_choice")
class StoryPartChoice {
  @PrimaryColumn()
  from_story_part_id: string;

  @PrimaryColumn()
  to_story_part_id: string;

  @Column()
  choice_description: string;

  @OneToOne(() => StoryPart)
  @JoinColumn({ name: "from_story_part_id" })
  from_story_part: StoryPart;

  @OneToOne(() => StoryPart)
  @JoinColumn({ name: "to_story_part_id" })
  to_story_part: StoryPart;

  @CreateDateColumn()
  created_at: Date;

  @UpdateDateColumn()
  updated_at: Date;
}
```

Figura 14: Exemplo de entidade do TypeORM

A biblioteca também oferece diversas maneiras de acesso ao banco de dados. A mais simples seria utilizar um "QueryBuilder", que auxilia na construção de uma query SQL e já retorna o resultado mapeado para as entidades definidas. Porém, utilizar isto diretamente nos casos de uso iria quebrar os padrões SOLID que estão sendo seguidos no projeto.

Logo, foi utilizado o padrão de repositório, que foi introduzido como parte do Domain-driven Design e funciona como uma abstração para o acesso aos dados. Este, é implementado inicialmente como uma interface genérica com os métodos necessários, como no exemplo da figura 15 abaixo, e esta interface possui duas implementações: uma com TypeORM e outra em memória que é utilizada para testes. O padrão permite então, que os dados sejam acessados via métodos simples e diretos e que caso seja necessário alterar a biblioteca de

acesso aos dados, o resto do código não seja dependente disso, bastando apenas criar uma nova implementação da interface.

```
interface IStoryRepository {
  create(data: ICreateStoryDTO): Promise<Story>;
  findById(id: string): Promise<Story>;
  findByUser(userId: string): Promise<Story[]>;
  resolve(story_id: string): Promise<Story>;
}
```

Figura 15: Exemplo de interface de repositório

6.4. Módulos e Casos de uso

Voltando ao principal conceito do *Domain-driven Design*, a implementação do back-end foi dividida em dois módulos: Accounts e Stories. Onde em cada módulo existem a seguinte estrutura de pastas: dtos, sigla para *Data Transfer Objects*, onde ficam armazenadas interfaces auxiliares para o módulo; infra, onde ficam armazenadas todas as implementações dependentes de algum framework; repositories, onde ficam todas as interfaces dos repositórios que cabem ao domínio do módulo em questão; e finalmente, useCases, onde ficam armazenados todos os casos de uso dos módulos. A figura 16 a seguir exibe a estrutura do módulo Stories.

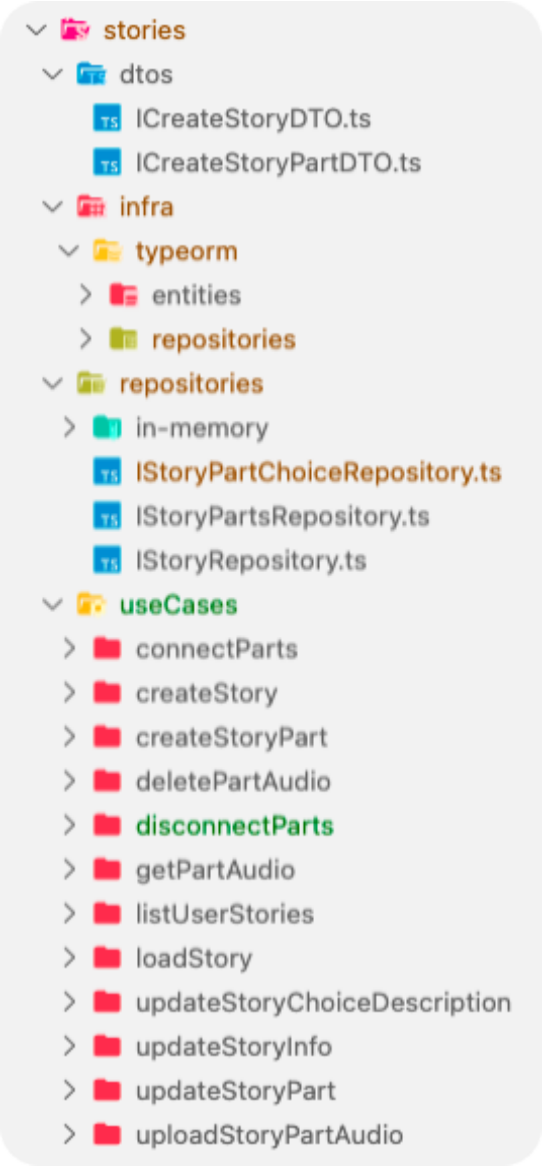

Figura 16: Estrutura de pastas do módulo stories

Os DTOs são subconjuntos de dados de classes do projeto e são utilizados a fim de simplificar a comunicação entre diferentes camadas da aplicação. Por exemplo, na figura 15 exibida anteriormente, a função de criar uma história recebe como parâmetro um objeto do tipo "ICreateStoryDTO", que pode ser visto na figura 17 abaixo, e armazena todas as informações relevantes para a criação de uma história.

```
interface ICreateStoryDTO {
  id?: string;
  begin_part_id?: string;
  owner_id?: string;
  title?: string;
  summary?: string;
  language?: string;
  author?: string;
  narrator?: string;
  tags?: string;
}
```

Figura 17: Exemplo de DTO

A pasta infra contém as implementações dependentes de frameworks, e no contexto desta aplicação são as entidades e as implementações dos repositórios, ambas dependentes do framework TypeORM. Já a pasta *repositories* armazena tanto as interfaces genéricas dos repositórios, quanto suas implementações em memória utilizadas para testes.

Como explicado na seção 5.4.1, um módulo possui diversos casos de uso, estes ficam armazenados na pasta useCases, onde existe uma subpasta para cada um deles que contém um arquivo para seu controlador, um para a implementação da sua regra de negócio e um para seus casos de teste, como é exibido na figura 18 abaixo.

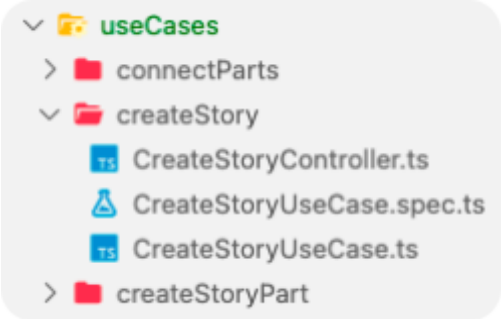

Figura 18: Arquivos de um caso de uso

A implementação dos controladores consiste em quatro etapas principais: a extração dos dados da requisição, seja do corpo, header ou parâmetros, a validação dos dados recebidos; a instanciação do caso de uso através do *container* de injeção de dependência; a execução do caso de uso passando os dados extraídos da requisição; e por fim, a captura do retorno do caso de uso e

seu envio para o roteador. Na figura 19 abaixo pode ser observada a implementação do controlador do caso de uso da criação de uma história.

```
class CreateStoryController {
  async handle(request: Request, response: Response): Promise<Response> {
    const { title } = request.body;
    const { user } = request;
    const createStoryUseCase = container.resolve(CreateStoryUseCase);
    const story = await createStoryUseCase.execute({
      title,
      owner_id: user.id,
    });
    return response.json(story);
  }
}
```

Figura 19: Exemplo de controlador de caso de uso

As implementações das regras de negócios ficam nos arquivos de cada caso de uso, estes arquivos possuem seu conteúdo muito variável, porém, seguem um formato padrão onde no construtor são recebidos os repositórios que serão usados na execução, e possuem uma única função pública chamada *execute*. Na figura 20 a seguir é exibida a implementação do caso de uso da criação de uma história.

```typescript
@injectable()
class CreateStoryUseCase {
  constructor(
    @inject("StoryRepository")
    private storyRepository: IStoryRepository,
    @inject("StoryPartsRepository")
    private storyPartRepository: IStoryPartsRepository
  ) {}

  async execute({ title, owner_id }: IRequest): Promise<Story> {
    // Creates the start point of the story
    const storyBeginPart = await this.storyPartRepository.create({});

    // Creates the story with a reference to its first part
    const story = await this.storyRepository.create({
      title,
      begin_part_id: storyBeginPart.id,
      owner_id,
    });

    // Updates the storyBeginPart to contain a reference to the story
    await this.storyPartRepository.create({
      id: storyBeginPart.id,
      story_id: story.id,
    });

    return story;
  }
}
```

Figura 20: Exemplo de implementação de caso de uso

6.5. Componentes e estilização

Como exposto na seção 5.4.2, no ReactJS, todos os elementos exibidos em telas são componentes, e na sua implementação seguindo os padrões SOLID, todos têm uma responsabilidade única e são separados de maneira que cada subcomponente que pode ser reutilizado, se torne um componente único.

Os componentes são implementados utilizando uma extensão da linguagem Typescript chamada TSX (Typescript XML), que foi criada pela equipe de desenvolvedores do ReactJS. Estes arquivos introduzem na linguagem a possibilidade de escrever tags de marcação presentes na linguagem HTML e tags personalizadas dentro do código Typescript, tornando o desenvolvedor capaz de inserir lógica na linguagem de marcação. Os arquivos TSX são descritos pela extensão ".tsx" enquanto arquivos Typescript são

descritos pela extensão ".ts". Esta diferença é necessária pois os arquivos TSX possuem diferenças na maneira como eles são interpretados.

Na estrutura de pastas do front-end, todo componente é representado por uma pasta com dois arquivos: "index.tsx" e "styles.ts". No arquivo "index" é definido o componente, com seu nome, sua funcionalidade e sua estrutura de subcomponentes. No arquivo "styles" é onde definimos componentes personalizados baseados nas tags padrões HTML, mas já inserindo estilização CSS nelas, este processo é feito utilizando o framework Styled Components[24].

A utilização desta biblioteca para estilização também permite que lógica seja adicionada dentro do CSS. Na figura 21 a seguir, podemos observar a declaração de um componente que é uma abstração de uma tag "div" do HTML padrão. Este componente recebe as propriedades descritas na interface "ContainerProps" e as utiliza para aplicar estilos diferentes dependendo dos valores dos booleanos.

```
interface ContainerProps {
  isBegin: boolean;
  isSelected: boolean;
}

export const Container = styled.div<ContainerProps>`
  background-color: #2c2c2c;
  width: 180px;
  height: 180px;
  border-radius: 15px;
  padding: ${(props) => props.theme.spacing(1)}px;
  border: ${({ isBegin, isSelected, theme }) => {
    if (isSelected) {
      return `1.5px solid ${theme.palette.primary.main}`;
    }
    if (isBegin) {
      return "1.5px solid green";
    }
    return "none";
  }};
`;
```

Figura 21: Exemplo de definição de componente estilizado

A principal página da aplicação é a de edição de uma história, onde é exibido o grafo de escolhas e suas opções de edição. Na figura 22 a seguir, podemos ver como esta página está estruturada e como é a sua divisão em subcomponentes, destacando os principais deles.

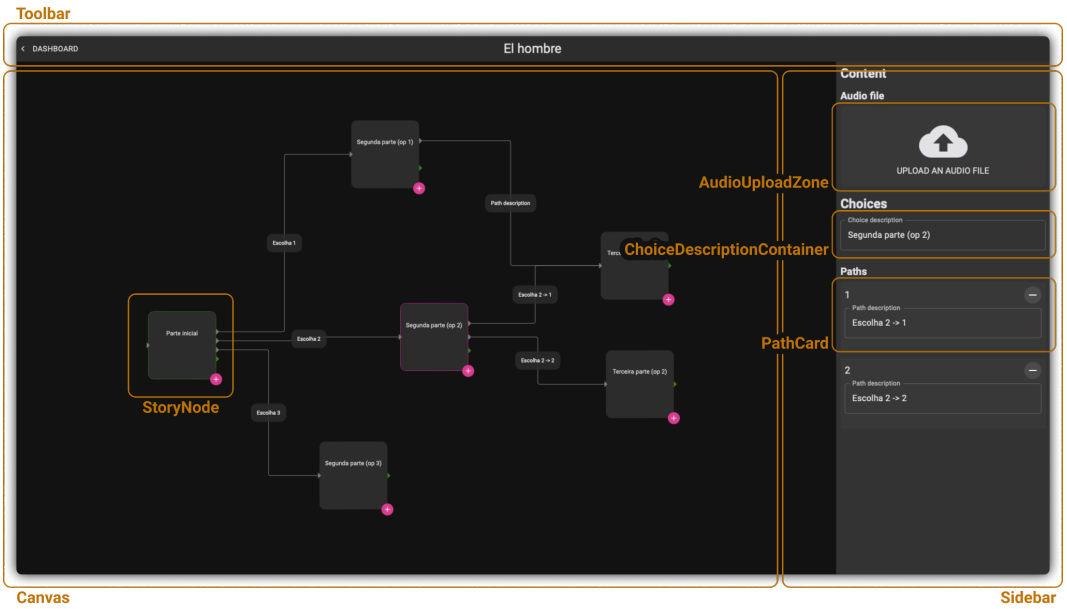

Figura 22: Subcomponentes da página de edição de histórias

6.6. Providers

Em aplicações React, os dados são passados de cima para baixo, via *props* (parâmetros passados na criação de subcomponentes), porém, estes possuem uma limitação de não serem mutáveis. Logo, o React possui uma API chamada *Context*[25] que foi criada para o compartilhamento de dados entre diversos componentes. Para compartilhar os dados entre componentes, é necessário encapsula-los em um Provider, que é oferecido pela API Context, este processo pode ser observado na figura 23 abaixo.

```
return (
  <StoryContext.Provider
    value={{
      story: data.story,
      selectedPart: data.selectedPart,
      storyReactFlowElements,
      setStoryId,
      selectPart,
    }}
  >
    {children}
  </StoryContext.Provider>
);
```

Figura 23: Exemplo de Provider a partir de Context

Associado a isto, foi utilizado um design pattern específico do React chamado *Hooks*. Este, visa tornar simples o compartilhamento de estados entre vários componentes além de permitir que a lógica que lida com estes estados seja armazenada e compartilhada com outros estados.

No total, foram desenvolvidos três *Hooks* com seus respectivos *Contexts*: o primeiro é o "*useAuth*", que dá acesso aos componentes ao usuário que está logado, e as funções de "*signIn*", "*signOut*" e "*updateUser*"; o segundo é o "*useToast*", que permite que os componentes exibam notificações ao usuário, como mensagens de erro por exemplo; e finalmente, o "*useStory*", que dá acesso a todos os estados relevantes a uma história, como a história atual, a parte selecionada, os componentes que constituem o grafo, além de algumas funções para a mudanças destes estados.

Um grande desafio do projeto foi a exibição do grafo que constitui a história, e para isso, foi utilizado o framework React Flow[26], que facilita o processo. Um dos desafios, foi transformar o retorno da requisição que carrega uma história completa em dados no formato que o framework aceita. Para isso, no *Hook* "*useStory*" foi implementada uma função que percorre todo o grafo e gera separadamente componentes para cada nó e para cada aresta, criando as conexões através dos identificadores retornados do servidor.

7. Avaliação do sistema

Durante a implementação foram realizados alguns testes no sistema para garantir seu funcionamento. As técnicas de teste variaram entre testes exploratórios, testes unitários e testes com usuários, como será detalhado nas seções 7.1 e 7.2 a seguir.

Além dos testes feitos, tanto no back-end quanto no front-end foi utilizado um analisador estático para garantir que o código esteja livre de maus cheiros e siga determinados padrões de estilo. O analisador utilizado foi o ESLint[27], amplamente utilizado no mercado por desenvolvedores Javascript e Typescript. Quanto ao estilo de código que é reforçado no projeto, foi escolhido seguir o guia de estilos da empresa Airbnb[28], que se provou aderente às necessidades do mercado ao ser utilizado por diversas grandes empresas.

7.1. Testes do back-end

No servidor, durante todo o desenvolvimento foram realizados testes exploratórios utilizando a ferramenta Insomnia[29], um cliente REST que permite que sejam criadas requisições teste para qualquer rota da API e permite que analisemos de maneira fácil a resposta e outros dados, como tempo de resposta, status, tamanho etc.

A ferramenta permite também que sejam criados diversos ambientes e diferentes variáveis para que seja fácil, por exemplo, alternar entre ambientes de desenvolvimento e produção. Na figura 24 abaixo, pode-se observar a lista de requisições testadas organizadas em diferentes pastas e algumas variáveis de ambiente, como "_.url", "_.resource”, "_.token".

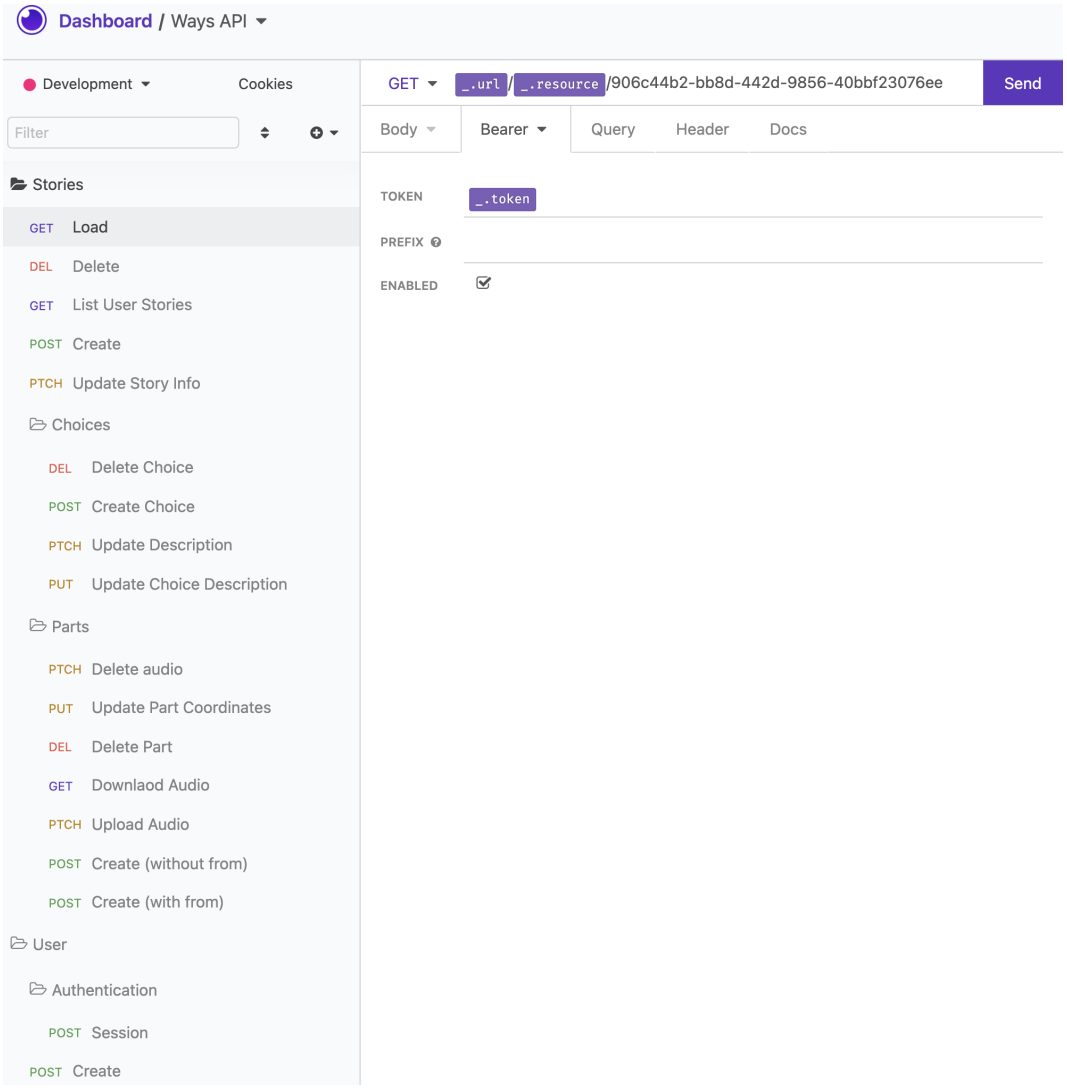

Figura 24: Estrutura de testes exploratórios no Insomnia

Para os testes unitários, foi utilizado o Jest[30], um framework voltado para testes em projetos Javascript e Typescript. A arquitetura escolhida facilitou nesse aspecto, pois a divisão em casos de uso permitiu que fosse fácil organizar tanto as estruturas de pastas quanto as regras de negócio que seriam testadas.

Além disso, como mencionado na seção 6.3, foram implementados repositórios em memória específicos para testes. Desta forma, os testes focam em validar as regras de negócios e não precisam acessar o banco de dados. Na figura 25 a seguir, podemos ver como são descritos os casos de teste de um caso de uso, neste exemplo os testes feitos são para a autenticação de usuários.

```
describe("Authenticate User", () => {
  beforeEach(() => {
    userRepositoryInMemory = new UserRepositoryInMemory();
    authenticateUserUseCase = new AuthenticateUserUseCase(
      userRepositoryInMemory
    );
    createUserUseCase = new CreateUserUseCase(userRepositoryInMemory);
  });

  it("Should be able to authenticate an user with correct credentials", async () => {
    const user: ICreateUserDTO = {
      username: "the_username",
      email: "user@teste.com",
      password: "1234",
      name: "User Test",
    };

    await createUserUseCase.execute(user);

    const result = await authenticateUserUseCase.execute({
      email: user.email,
      password: user.password,
    });

    expect(result).toHaveProperty("token");
  });

  it("Should not be able to authenticate a non existent user", async () => {-
  });

  it("Should not be able to authenticate an user with incorrect password", async () => {-
  });
});
```

Figura 25: Exemplos de testes para caso de uso

7.2. Testes de interface

Como dito ao longo do texto, uma grande preocupação do projeto é em relação a usabilidade do sistema. Logo, foram feitos testes de usuário para entender os pontos fracos da interface. Nestes testes, diversos problemas foram encontrados, porém eles foram a etapa final do desenvolvimento deste trabalho, logo, suas correções serão feitas no futuro e fora do escopo do projeto final.

Nos testes, foi pedido aos usuários que executassem uma série de ações e tentassem descobrir como fazê-las sozinhos. Enquanto as tarefas foram executadas, foi observado o comportamento dos testadores para identificar em quais momentos sentiram dificuldades. Além disso, foi pedido que falassem o que estavam pensando enquanto faziam as atividades.

A seguir, são detalhados alguns dos principais pontos que precisam ser melhorados na usabilidade do sistema:

- Ao selecionar uma parte da história, todas as suas informações e a possibilidade de edição aparece na barra lateral, incluindo o botão para deletar a parte selecionada. Uma unanimidade entre os usuários foi tentar utilizar o teclado para realizar algumas tarefas como: tecla DEL para deletar uma parte; tecla ENTER para salvar a edição feita em um dos campos; setas do teclado para fazer um ajuste fino na posição dos nós do grafo. Logo, serão adicionados controles via teclado.

- Algo comum entre os testadores foi não entender rapidamente que o canvas onde o grafo se encontra é algo móvel, que pode ser arrastado para visualizar outras áreas e pode ter o zoom alterado para visualizar mais itens. Então, serão adicionados controles na lateral do canvas para ações de zoom in e out, além de um mini mapa para indicar qual parte do canvas está sendo visualizada. Desta forma ficará mais claro a mobilidade do canvas.

- Uma queixa dos usuários foi a inabilidade de desfazer alguma ação errada. Logo, será adicionada a função tanto no formato de um botão, quanto no atalho CTRL+Z do teclado.

- De maneira geral, os testadores inicialmente tentaram editar os textos referentes aos nós e as arestas do grafo diretamente no canvas, em vez de selecioná-los e fazer sua edição na barra lateral. Então, será adicionada a opção de editar os parâmetros de uma parte e suas escolhas diretamente no canvas, ao dar um duplo clique sobre o texto.

- Na tela de dashboard, onde são exibidas as histórias de um usuário, ao clicar no botão "New Story", um alerta é aberto e solicita o nome da história para sua criação, e após isto, a nova história aparece na lista. Porém, ao criar a história, os usuários esperavam já ser direcionados para a tela de edição. Logo, este comportamento será alterado.

Na maior parte dos casos, houve uma curva de aprendizado rápida em relação ao uso da plataforma. Mesmo nos casos que os usuários tentaram fazer algo de maneira equivocada, logo descobriram a maneira correta de realizar a ação.

8. Considerações finais e trabalhos futuros

A execução e a conclusão deste projeto só foi possível graças a diversas disciplinas da grade curricular do curso de Ciência da Computação, como: Programação Orientada a Objetos, que apresentou este paradigma e introduziu a importância dos design patterns; Interação Humano-Computador, que demonstrou o valor da experiência de usuário; Bancos de Dados, que mostrou como utilizar os bancos de dados relacionais da melhor maneira; Testes e Medidas de Software, que apresentou diferentes métodos de testes e sua relevância; Estruturas de Dados Avançadas, que me familiarizou com as estruturas de dados utilizadas; além de muitas outras que em conjunto formaram a bagagem necessária para o projeto.

Além disso, os estágios feitos em laboratórios da PUC-Rio foram primordiais para a execução do projeto, em especial o Apple Developer Academy PUC-Rio que mostra como há espaço para inovação nos mais diferentes meios e foi onde se deu o início deste projeto.

O projeto foi repleto de desafios técnicos que me trouxeram uma maior capacidade de trabalhar com back-end, que vão desde a configuração do ambiente que rodará no servidor, até o acesso ao banco de dados. Além disso, foi possível sair da minha zona de conforto relacionada ao front-end, pois foi necessário ir a fundo no funcionamento do ReactJS e de algumas bibliotecas, como o React Flow, que foi customizado para se adequar ao projeto. Além de ter sido uma excelente experiência relacionada ao desenvolvimento de interfaces e de experiência do usuário.

Algo que poderia ter sido feito de maneira diferente no projeto foi a ordem de execução das tarefas. O back-end foi construído quase completamente antes do front-end, porém, o trabalho fluiu muito mais quando as duas camadas foram desenvolvidas em paralelo, pois desta forma as rotas da API puderam ser testadas e seus problemas identificados mais facilmente. Além disso, com o

front-end desenvolvido mais cedo, seria possível realizar testes da interface mais cedo e assim evitado alguns problemas.

Para o lançamento da aplicação serão necessárias as aquisições de um domínio, servidor para a hospedagem do back-end e servidor para a hospedagem do front-end, o que acarreta em diferentes custos e por isso não pôde ser incluído no escopo do projeto. Porém, está nos planos para o futuro próximo, lançar uma versão de testes do sistema para captar os primeiros usuários e executar mais rodadas de testes.

Por fim, para tornar todo o ecossistema completo, o aplicativo Ways terá sua camada de acesso aos dados reformulada para consumir os dados da API. Assim, os usuários do Ways Studio serão capazes de publicar suas histórias enquanto os usuários do Ways serão capazes de as consumir.

Referências bibliográficas

[1] THE State of The Podcast Universe 2020 Mid-Year Preview, Voxnest, 30 de jun. 2020. Disponível em: <https://blog.voxnest.com/2020-mid-year-podcast-industry-report>. Acesso em: 8 de mar. de 2021.

[2] AUDIOBOOKS Market Size, Share & Trends Analysis Report By Genre, By Preferred Device (Smartphones, Laptops & Tablets, Personal Digital Assistants), By Distribution Channel, By Target Audience, By Region, And Segment Forecasts, 2020 - 2027, Grand View Research, jun. de 2020. Disponível em: <https://www.grandviewresearch.com/industry-analysis/audiobooks-market>. Acesso em: 8 de mar. de 2021.

[3] GAMING Market - Growth, Trends, Covid-19 Impact, and Forecasts (2021 - 2026), Mordor Intelligence, 2020. Disponível em: <https://www.mordorintelligence.com/industry-reports/global-games-market>. Acesso em: 8 de mar. de 2021.

[4] APP Store, Ways Áudios Interativos. Disponível em: <https://apps.apple.com/br/app/ways/id1537176656>. Acesso em: 8 de mar. de 2021.

[5] VOICEFLOW. Design, prototype and launch voice apps. Disponível em: <https://www.voiceflow.com>. Acesso em: 8 de mar. de 2021.

[6] ANCHOR, The easiest way to make a podcast. Disponível em: <https://anchor.fm>. Acesso em: 8 de mar. de 2021.

[7] NODE.JS. Disponível em: <https://nodejs.org/en/>. Acesso em: 8 de mar. de 2021.

[8] REACT, A JavaScript library for building user interfaces. Disponível em: <https://reactjs.org>. Acesso em: 8 de mar. de 2021.

[9] TYPESCRIPT, Typed JavaScript at Any Scale. Disponível em: <https://www.typescriptlang.org>. Acesso em: 15 de jun. de 2021.

[10] IGNITE, Rocketseat. Disponível em: <https://rocketseat.com.br/ignite>. Acesso em: 15 de jun. de 2021.

[11] ROCKETSEAT, Evolua rápido com a tecnologia. Disponível em: <https://rocketseat.com.br/sobre>. Acesso em: 15 de jun. de 2021.

[12] CLEAN, Coder Blog. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 15 de jun. de 2021.

[13] DOCKER, Empowering App Development for Developers. Disponível em: <https://www.docker.com>. Acesso em: 15 de jun. de 2021.

[14] FIGMA, The collaborative interface design tool. Disponível em: <https://www.figma.com>. Acesso em: 16 de nov. de 2021.

[15] MUI, The React component library you always wanted. Disponível em: <https://mui.com>. Acesso em: 16 de nov. de 2021.

[16] DIGITAL OCEAN, SOLID: The First 5 Principles of Object Oriented Design. Disponível em: <https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>. Acesso em: 16 de nov. de 2021.

[17] CLEAN CODER BLOG, The Clean Architecture. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html >. Acesso em: 16 de nov. de 2021.

[18] FULLCYCLE, O que é DDD - Domain Driven Design. Disponível em: <https://fullcycle.com.br/domain-driven-design/>. Acesso em: 16 de nov. de 2021.

[19] JWT.IO, JSON Web Token Introduction. Disponível em: <https://jwt.io/introduction>. Acesso em: 16 de nov. de 2021.

[20] SWAGGER, Bearer Authentication. Disponível em: <https://swagger.io/docs/specification/authentication/bearer-authentication/>. Acesso em: 16 de nov. de 2021.

[21] IETF, Internet Engineering Task Force. Disponível em: <https://www.ietf.org >. Acesso em: 16 de nov. de 2021.

[22] RFC7519, JSON Web Token (JWT). Disponível em: <https://datatracker.ietf.org/doc/html/rfc7519>. Acesso em: 16 de nov. de 2021.

[23] RFC6750, The OAuth 2.0 Authorization Framework: Bearer Token Usage. Disponível em: <https://datatracker.ietf.org/doc/html/rfc6750>. Acesso em: 16 de nov. de 2021.

[24] STYLED-COMPONENTS. Disponível em: <https://styled-components.com>. Acesso em: 16 de nov. de 2021.

[25] REACT, Context. Disponível em: <https://pt-br.reactjs.org/docs/context.html>. Acesso em: 16 de nov. de 2021.

[26] REACT FLOW, An open source library for building node based applications. Disponível em: <https://reactflow.dev>. Acesso em: 16 de nov. de 2021.

[27] ESLINT, Pluggable JavaScript linter. Disponível em: <https://eslint.org>. Acesso em: 16 de nov. de 2021.

[28] AIRBNB, Engineering & Data Science. Disponível em: <https://airbnb.io>. Acesso em: 16 de nov. de 2021.

[29] INSOMNIA, The API Design Platform and API Client. Disponível em: <https://insomnia.rest>. Acesso em: 16 de nov. de 2021.

[30] JEST, Delightful JavaScript Testing. Disponível em: <https://jestjs.io>. Acesso em: 16 de nov. de 2021.

Apêndice A - Requisitos do sistema

Módulo	Caso de Uso	Tipo	Descrição
Accounts	Cadastro de usuário	Funcional	Deve ser possível cadastrar um novo usuário
		Regra de Negócio	Não deve ser possível cadastrar dois usuários com o mesmo `username`
		Regra de Negócio	Não deve ser possível cadastrar dois usuários com o mesmo `email`
Accounts	Autenticação de usuário	Funcional	Deve ser possível autenticar um usuário
		Regra de Negócio	Não deve ser possível autenticar um usuário sem 'username' ou 'password' preenchidos
		Regra de Negócio	Não deve ser possível autenticar um usuário cujo 'username' não está presente no banco de dados
		Regra de Negócio	Só deve ser possível autenticar um usuário se a senha cadastrada para o 'username' dado forem idênticas
		Funcional	Toda senha armazenada no banco de dados deve ser criptografada
		Funcional	Toda senha enviada via requisições HTTP deve ser criptografada
Accounts	Exclusão de usuário	Funcional	Um usuário deve ser capaz de excluir seu cadastro
		Regra de Negócio	Uma mensagem de confirmação deve ser exibida antes da exclusão do cadastro
Stories	Listagem de histórias	Funcional	Deve ser possível listar todas as histórias pertencentes a um usuário
		Regra de Negócio	Caso o usuário liste suas próprias histórias, devem estar inclusas as histórias publicadas e não publicadas
		Regra de Negócio	Caso o usuário liste as histórias de outro usuário, devem estar inclusas apenas as histórias publicadas

Stories	Criação de histórias	Funcional	Um usuário deve ser capaz de criar uma história
		Regra de Negócio	Toda história deve ser criada como indisponível
		Regra de Negócio	Deve ser possível criar uma história apenas se o com o campo nome for preenchido
		Regra de Negócio	Toda nova história deve ser inicializada a com uma parte inicial em branco
		Regra de Negócio	Uma história pode ser criada com apenas seu título
		Regra de Negócio	Uma história só pode ser criada por um usuário autenticado
Stories	Criação de partes	Funcional	Deve ser possível criar novas partes dentro de uma história
		Funcional	Deve ser possível criar uma parte a partir de outra parte, com uma conexão criada automaticamente
		Regra de Negócio	Toda nova parte deve ser exibida à direita da parte que a originou
		Regra de Negócio	Apenas o usuário dono da história pode criar novas partes
Stories	Conexão de partes	Funcional	Deve ser possível criar uma conexão/escolha entre duas partes diferentes
		Regra de Negócio	Só deve ser possível existir uma única conexão entre duas partes
Stories	Exclusão de partes	Funcional	Deve ser possível excluir uma parte de uma história
		Regra de Negócio	Apenas o usuário dono da história pode excluir partes dela
		Regra de Negócio	Toda história deve possuir ao menos uma parte, logo, a última parte não pode ser excluída
		Funcional	Ao excluir uma parte, todas as conexões/escolhas ligadas a ela também devem ser excluídas

		Funcional	Ao excluir uma parte, caso existe um arquivo de áudio associada a ela, o arquivo deve ser excluído
Stories	Upload de áudio	Funcional	Deve ser possível fazer o upload de um arquivo de áudio associado a uma parte da história
		Regra de Negócio	Apenas o usuário dono da história deve ser capaz de fazer o upload de um arquivo de áudio para uma parte dela
		Regra de Negócio	Deve existir no máximo um arquivo de áudio associado a cada parte da história
		Regra de Negócio	Os arquivos de áudio devem ser dos formatos: MP3, FLAC, AAC, MP4, M4A, WAV ou WMA
Stories	Exclusão de um áudio	Funcional	Deve ser possível excluir um arquivo de áudio associado a uma parte
		Regra de Negócio	Antes da exclusão, deve ser exibida uma mensagem de confirmação ao usuário
Stories	Carregamento de uma história	Funcional	Deve ser possível carregar uma história completa, com todos os seus caminhos
		Regra de Negócio	Usuários não autenticados podem carregar uma história completa
Stories	Desconexão de partes	Funcional	Deve ser possível desfazer uma conexão entre partes
		Regra de Negócio	Apenas o usuário dono da história em questão pode desconectar partes dela
Stories	Atualização da descrição de uma escolha	Funcional	Deve ser possível atualizar a descrição de uma escolha
		Regra de Negócio	Apenas o usuário dono da história em questão pode atualizar descrições de escolhas dela
Stories	Atualização dos dados de uma história	Funcional	Deve ser possível atualizar os campos title, summary, language, author, narrator e tags de uma história
		Regra de Negócio	Apenas o usuário dono da história pode atualizar seus dados

Stories	Atualização da descrição de uma parte	Funcional	Deve ser possível atualizar a descrição de uma parte
		Regra de Negócio	Apenas o usuário dono da história em questão pode atualizar descrições de partes dela
Stories	Upload de capa da história	Funcional	Deve ser possível fazer o upload de uma imagem para a capa de uma história
		Regra de Negócio	A imagem deve ser do formato JPG ou PNG
		Regra de Negócio	A imagem deve possuir largura e altura iguais, ou seja, ser quadrada