Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Spotunes

Uma aplicação web para organização de bibliotecas musicais digitais

Édio Cerati Neto

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Antonio Deusany de Carvalho Junior

São Paulo
2021

“Enquanto tem bambu, tem flecha!” - Everaldo Marques

Resumo

Édio Cerati Neto. **Spotunes: *Uma aplicação web para organização de bibliotecas musicais digitais***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

O projeto consistiu no desenvolvimento de um sistema web chamado *Spotunes*, que permite ao usuário uma organização completa dos metadados de sua biblioteca de músicas digitais, com dados e reprodução baseados nos serviços do *Spotify*. A motivação se baseou em dar poder total ao usuário de organizar sua biblioteca como melhor preferir, podendo editar os metadados das músicas de acordo com sua preferência, bem como criar playlists da maneira que achar conveniente. Ele também pode exportar sua biblioteca em *JSON* e importá-la, tendo sempre fácil acesso à informação criada. Foram usadas tecnologias web, pois, além de aplicações web serem de fácil usabilidade (requerem apenas um navegador convencional), também são facilmente compatíveis entre plataformas e ambientes diferentes. Houve também um cuidado para deixar a aplicação minimamente dependente de um *back-end* (utilizado apenas para buscas na *API* do Spotify). O resultado final pode ser conferido em https://spotunes.netlify.app.

Palavras-chave: Spotify. iTunes. Gerenciador de biblioteca musical.

Abstract

Édio Cerati Neto. **Spotunes:** ***A web application to organize digital music libraries***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

The project consisted in the development of a web platform called *Spotunes*, which allows the user a complete organization of the metadata of their digital music library, with data and reproduction based on *Spotify* services. The motivation was to give full power to the user to organize their library as they prefer, being able to edit the metadata of the songs according to their preferences, as well as creating playlists in the way they see fit. It can also export the library in *JSON* and import it, always having easy access to the created information. Web technologies were used because, in addition to web applications being easy to use (they only require a conventional browser), they are also highly compatible across different platforms and environments. There was also a goal of leaving the application minimally dependent on a *back-end* (used only for searches in Spotify's *API*). The final result can be found at https://spotunes.netlify.app.

Keywords: Spotify. iTunes. Music library manager.

Lista de Figuras

Sumário

Capítulo 1

Introdução

No final da década de 90 e começo dos anos 2000, surgiram diversos dispositivos portáteis, capazes de reproduzir arquivos de música digital. Junto com a popularização de formatos de compressão de áudio, como o MP3, usuários podiam levar milhares de músicas em seus bolsos, em um dispositivo menor que uma fita cassete ou CD, sendo que estes últimos armazenavam, em média, de 10 a 15 músicas. Portanto, foi inevitável que tocadores de MP3 ganhassem a preferências dos usuários para o consumo de músicas.

Já no início da década de 2010, com o surgimento de *smartphones* e acesso mais comum à Internet móvel, o *streaming* ganha força pela praticidade. Como agora os arquivos de música estão centralizados em um servidor que provê tal serviço, o usuário passa a ter acesso a milhões de músicas *online*, bastando selecioná-las. Diante da quantidade colossal de escolhas, surge o desafio que este trabalho procura solucionar: **como organizar e personalizar o acesso a uma coleção de milhões de arquivos de música?**

A alternativa proposta para solucionar tal desafio foi o desenvolvimento de uma plataforma web chamada *Spotunes*, onde o usuário poderá adicionar músicas provenientes de serviços de *streaming* em sua biblioteca virtual, alterando seus metadados como preferir. Através dela, o usuário tem acesso a dados sobre a reprodução das músicas, assim como a capacidade de criar *playlists* manuais ou auto-atualizáveis com base em critérios pré-determinados.

1.1 Bibliotecas de Música Digital

Programas como o *iTunes* e *Windows Media Player* criam uma biblioteca virtual a partir dos arquivos de música do usuário. Eles são capazes de organizar as músicas digitais a partir de seus metadados, permitindo buscar uma música específica por título, álbum, artista, gênero, dentre várias outras informações. Para auxiliar na organização, o utilizador também pode editar esses metadados como bem entender.

Neste trabalho, focou-se nas características do *iTunes* (Apple, 2021a), pois ele oferece uma vasta gama de opções ao usuário, incluindo o armazenamento de informações de reprodução de cada faixa musical. Alguns exemplos das informações armazenadas:

- Quantidade de reproduções;
- Data da última reprodução;
- Data de adição na biblioteca;
- Classificação de uma a cinco estrela.

1.2 *Playlists*

No desenvolvimento do *Spotunes*, há dois tipos de playlists:

- *Playlists* criadas manualmente;
- *Playlists* Inteligentes aos moldes do *iTunes* (Apple, 2021b);

Playlists Manuais

Playlists manuais são listas de músicas em que o usuário manualmente define quais músicas devem participar e em qual ordem. Geralmente estão ligadas a questões subjetivas do usuário. Um exemplo seria: “Minhas 10 músicas favoritas", na qual o usuário pode definir a *playlist* na ordem adequada para tais 10 músicas.

Playlists Inteligentes

Playlists Inteligentes são um recurso implementado pelo *iTunes* desde suas primeiras versões. O usuário define critérios que os metadados das músicas devem cumprir para que elas sejam adicionadas à *playlist*. Alterações na biblioteca atualizam essas playlists inteligentes automaticamente. Exemplos de *Playlists* Inteligentes:

- Músicas cuja classificação seja 5 estrelas;
- Músicas de determinado artista, lançadas antes de 1990, que o usuário não tenha ouvido no último ano;
- Músicas ainda sem classificação (Zero estrelas).

1.3 *APIs* Públicas

Grande parte dos serviços de streaming fornece uma *API* publicamente, para que desenvolvedores possam construir novas ferramentas com base no que já existe na aplicação oficial. Isto é a força motriz por trás do desenvolvimento do *Spotunes*, onde as *APIs* fornecem os dados sobre as músicas, assim como a reprodução delas. *APIs* dos principais serviços:

- *Spotify Web API* [Spotify, 2021b];
- *Youtube Data API* [Google, 2021b];
- *Soundcloud API* [SoundCloud, 2021c];

Parte do trabalho consistiu em testar essas APIs de forma a checar suas funcionalidades e requisitos de uso. Ao final, decidiu-se o enfoque nas funcionalidades apenas da API do Spotify, por ser de fácil utilização, contendo uma vasta gama de dados disponíveis, sem limites de uso e fornecendo reprodução de músicas por meio de widgets para o streaming.

1.4 Usuário

O usuário terá total controle sobre o conteúdo de sua biblioteca, podendo fazer edições de metadados como melhor preferir, junto com capacidades de exportação e importação de *JSON*, tornando possível até que conteúdo seja gerado fora do Spotunes através de scripts e outras ferramentas, facilitando a construção e manutenção de uma biblioteca de usuário. Somado a isso, os repositórios do Spotunes possuem explicação claras e sucintas de como o usuário pode realizar sua própria instalação do Spotunes.

Por ser *open source*, a plataforma continuará sendo desenvolvida para além do escopo deste Trabalho de Conclusão de Curso.

1.5 Objetivos

Resumindo os objetivos do trabalho, apresentados e contextualizados na seção anterior:

- Modelar a biblioteca do usuário, armazenando os metadados das músicas incluídas (tanto da faixa musical em si quanto dados de reprodução) em um banco de dados;
- Dar suporte a *Playlists* Inteligentes e Manuais;
- Escolher uma API dentre serviços mais consagrados: *Spotify*, *Youtube* e *SoundCloud*;
- Permitir o maior controle possível ao usuário de como sua biblioteca é gerenciada, bem como a capacidade de exportar e importar suas informações facilmente;
- Tornar a aplicação minimamente dependente de um *back-end* de forma que a maior parte do uso seja gerenciada pelo próprio cliente (navegador do usuário).

Capítulo 2

Referencial Teórico

Neste Trabalho foi realizado um estudo sobre diversas ferramentas e tecnologias que envolvem sistemas de bibliotecas musicais digitais. Estes temas e outros serão melhor detalhados a seguir.

2.1 Sistemas Gerenciadores de Bibliotecas Musicais Digitais

Sistemas Gerenciadores de Bibliotecas Musicais Digitais surgiram ao final da década de 90 e início dos anos 2000. Estes sistemas ganharam atenção por serem úteis como forma de organizar coleções de arquivos de músicas digitais, popularizados em formatos como o *mp3*.

2.1.1 *Winamp*

O *Winamp* é um *player* de áudio e vídeo, lançado em 1997. Suas funcionalidades são bastante similares a outros programas, como o *iTunes*, descrito a seguir na Seção 2.1.3, contendo *features* como edição de metadados, criação de *playlists*, classificação de músicas e histórico de reprodução. Porém, o *Winamp* mais novo, apresentado na Figura 2.1, possui uma interface com muitos recursos e que pode ser considerada complexa por alguns usuários, por apresentar muitas informações simultaneamente na mesma tela. Além disso, não permite *playlists* inteligentes (auto-atualizáveis com base em critérios) e os metadados e estatísticas sobre reprodução são mais limitados.

Alguns detalhes de sua interface são interessantes e trazem bons *insights* de usabilidade: a barra lateral esquerda permite navegação rápida entre componentes da biblioteca, enquanto que a barra lateral direita mostra informações adicionais do item selecionado atualmente (álbum, *playlist*, música, etc).

Figura 2.1: *Tela principal do* Winamp. *[Fonte: Radionomy, 2021]*

2.1.2 *Musicmatch Jukebox*

Musicmatch Jukebox foi um *player* e sistema gerenciador de biblioteca muito popular durante o começo dos anos 2000. A tentativa de estudar seu design e funcionalidade não rendeu muitos frutos pois, por ter sido descontinuado em 2008, com última versão de download constando 2005, não há suporte oficial para os sistemas operacionais mais recentes. No entanto, foi possível testar sua usabilidade por meio de uma máquina virtual do Windows 7.

Um quesito interessante de sua interface é que o design de seu *player* foi inspirado nos aparelhos de som *automotivos* populares no começo dos anos 2000, como pode ser visto na Figura 2.2. A ideia de ter uma interface semelhante a algo já comum aos usuários tende a fazer o usuário já saber onde encontrar seus botões e funcionalidades caso já tenha feito uso de aparelhos semelhantes no seu cotidiano.

2.1.3 iTunes

O *iTunes* é um *player* de áudio e vídeo desenvolvido pela *Apple*, lançado em 2001. Originalmente foi vendido como um organizador de músicas digitais para facilitar a sincronização com um dispositivo *iPod*, porém ganhou popularidade até mesmo para usuários de outras plataformas por permitir controle completo sobre seus arquivos de música digital.

A interface do *iTunes* tem sua usabilidade baseada na navegação da barra lateral, onde pode-se aplicar a filtragem do que se deseja que a aplicação mostre em sua área principal, no meio. Uma visualização dos álbuns disponíveis em uma biblioteca é apresentada na

Figura 2.2: *Tela principal do* Musichmath Jukebox. *[Fonte: YAHOO, 2021]*

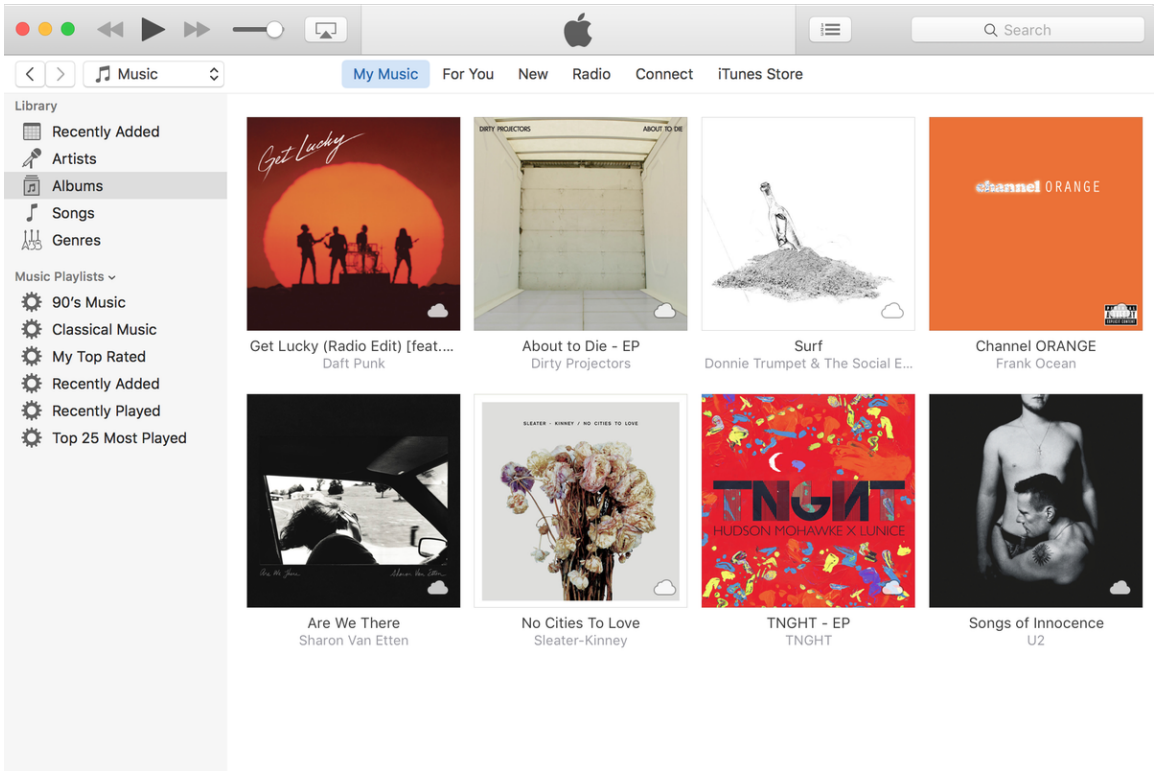

Figura 2.3: *Tela principal do* iTunes. *[Fonte: APPLE, 2021c]*

Figura 2.3.

A exibição da biblioteca de usuário no iTunes se baseia na tabela apresentada ao centro da aplicação. O usuário pode escolher quais dados serão exibidos sobre cada música, bem como a ordenação deles. Os metadados dos arquivos de música digital podem ser editados de acordo com a preferência do usuário e, junto com os dados de reprodução e da biblioteca,

podem ser usados para compor playlists que o usuário julgue conveniente.

2.2 Documentação de APIs

API significa *Application Programming Interface*, ou seja, é um software que funciona como intermediário permitindo que duas aplicações diferentes se comuniquem e interajam. Como a ideia por trás do desenvolvimento do *Spotunes* envolve receber informações de músicas (bem como capacidade de reprodução) através de serviços de *streaming* como *Spotify*, *YouTube* e *SoundCloud*, foram estudados o funcionamento de suas APIs.

O teste de cada *API* consistiu em ler a documentação oficial e, com base nela, ser capaz de buscar músicas e reproduzir uma escolhida da seleção. Alguns serviços fornecem *widgets* para reprodução de músicas através de *iframes*, de forma que o mecanismo de reprodução é todo gerenciado pelo serviço original. Outra opção fornecida pelo *Spotify*, descrito na Seção 2.2.1, é o *Web Playback SDK*, onde a aplicação utilizadora fica encarregada de gerenciar a reprodução por contra própria (necessitando construir um *player* em *HTML* e lidar com as ações de reprodução).

2.2.1 *Spotify*

O *Spotify* é um dos maiores serviços de *streaming* de músicas da atualidade, contando com um plano grátis (baseado em anúncios) e também um plano *premium* através de assinatura paga mensalmente. Dada sua popularidade, o *Spotify* é capaz de manter um dos maiores acervos dentre os serviços de *streaming*, fornecendo músicas desde os mais famosos artistas até pequenos compositores independentes.

A *API* do *Spotify* retorna os dados sobre as músicas do catálogo do serviço diretamente em *JSON*. Esta API segue princípios *REST* RestfulAPI, 2021, então a nomenclatura é intuitiva para utilização. Por exemplo, receber informações sobre as características de áudio de uma música de id específico é tão simples quanto:

```
GET https://api.spotify.com/v1/audio-features/{id}
```

O *endpoint* acima fornece características extraídas dos áudios das músicas já pré-computadas, tornando-se uma poderosa abstração de técnicas mais complexas de computação musical. Isso torna construir um sistema de recomendação baseado no *Spotify* em uma tarefa muito mais simples. Instruções sobre como usar a *API* para obter informações específicas de músicas podem ser encontradas em Spotify, 2021b.

Quanto à reprodução de músicas por streaming, ela pode ser feita através de *widgets* fornecidos pelo *Spotify* por meio de elementos *HTML* do tipo *iframe* ou por meio do *SDK*. O uso dos *iframes* está especificado em Spotify, 2021c, bastando fornecer o link para a música desejada no *iframe*, que se encarrega de carregar um *player* com álbum e nome da música automaticamente. Os *widgets* do *Spotify* funcionam para contas que não sejam *Premium*.

Sobre o *SDK*, ele requer um *token* de usuário para se conectar. De acordo com Spotify, 2021d, autenticação por *OAuth* é recomendada, especialmente para uso em produção. Isto torna a integração com uma plataforma de terceiros, como o *Spotunes*, um pouco mais

complexa. Efetuando a conexão, a aba do navegador atual surge como um dispositivo atrelado à conta do *Spotify*, portanto é possível requisitar a reprodução nesta aba conectada a partir do aplicativo do *Spotify* em outro dispositivo.

Porém, para ser capaz de executar a função *play* do *SDK*, é necessário que o usuário autenticado pague o serviço *Premium* do *Spotify*, do contrário um erro HTTP 403 (Proibido) é gerado. Documentação do *SDK* em Spotify, 2021a.

2.2.2 *YouTube*

O *YouTube* é um dos mais populares serviços de *streaming* atualmente, tendo em seu início focado em vídeos, porém com o crescimento da plataforma, hoje conta com recursos de *streaming* de músicas, em uma subseção como o *YouTube Music*, além de possuir em sua extensa biblioteca vídeos musicais.

A *API* do *YouTube* requer que uma chave de *API* seja gerada, de acordo com [Google, 2021b]. Por padrão, o *YouTube* disponibiliza um limite diário de 10 mil cotas de utilização por chave de API, sendo que uma busca gasta 100 unidades. É possível requisitar ao *Google* uma expansão dessa cota, porém esta precisa ser devidamente justificada, com uso bem documentado do que será feito em sua aplicação.

No geral, a *API* do *YouTube* parece mais simples e mais robusta que a do *Spotify*, porém tem como contra a cota de uso diário mencionada. Todavia, seria possível adquirir informações do *YouTube* sem depender diretamente da *API*, pois basta requisitar o *HTML* do próprio site do *YouTube* (passando o parâmetro de busca desejado) e *parsear* as informações contidas nele.

Uma forma possível de reprodução é através da utilização de *iframes* também, que recebe a URL do vídeo a ser reproduzido. O *iframe* pode ser configurado tanto através da *API* quanto sem depender dela, apenas inserindo o elemento HTML com os atributos adequados. Uma limitação do uso dos *iframes* do *YouTube* é que por padrão não permitem o uso em tela cheia na aba atual do navegador.

2.2.3 *SoundCloud*

O SounCloud é um serviço de *streaming* com uma proposta diferente de outros serviços mencionados anteriomente, como *Spotify* e *YouTube*: o seu acervo é composto por arquivos de áudio digital enviados por qualquer usuário cadastrado. Sendo assim, tornou-se uma ferramenta importante para a divulgação de artistas independentes, bem como programas de *podcasts*.

A *API* do *SoundCloud* também possui um *SDK* [SoundCloud, 2021b], que esbarra nas mesmas questões de praticidade mencionadas anteriormente na seção da *API* do *Spotify*. Além disso, o uso da *API* do *SDK* possui termos de uso bem restritivos [SoundCloud, 2021d], exigindo que a aplicação desenvolvida não possua referências à marca ou logo do *SoundCloud*. Devido a essa proteção de sua propriedade intelectual e de seus serviços, aliada à alta demanda por chaves de sua *API*, o *SoundCloud* parou de liberá-las temporariamente.

Uma solução possível é a mesma descrita na seção anterior (sobre a *API* do *YouTube*): *parsear* o *HTML* da página web do *SoundCloud* para obter informações e passá-las ao *widget* (também um *iframe*) oferecido pelo serviço.

Há também a *API* de busca, que requer uma chave do cliente para a utilização. Sua requisição também foi desabilitada temporariamente pelo *SoundCloud* assim como a *API* do *SDK*. Funcionamento da *API* de busca pode ser encontrada em [SoundCloud, 2021a].

2.3 Recomendação musical

Um dos intuitos dos principais sistemas de reprodução de mídias digitais é disponibilizar recomendação de novas mídias. A seguir são apresentados três trabalhos recentes que tratam do tema de maneira diversa e objetiva.

No artigo Jannach *et al.*, 2018, os autores discorrem sobre técnicas utilizadas em algoritmos de recomendação musical, incluindo os desafios de descoberta de novas músicas e recomendação da próxima faixa musical, focando na experiência prática do *Spotify*. Há 4 tipos de recomendações:

- Não personalizadas: músicas que estão em alta dentre os usuários da plataforma no momento atual
- Contextualizadas porém não personalizadas: playlists geradas a partir de um artista ou música
- Personalizadas porém não contextualizadas: playlists recomendadas a partir do comportamento do usuário na plataforma
- Personalizadas e contextualizadas: recomendação de faixas enquanto o usuário constrói uma *playlist* nova, sendo tais faixas recomendadas baseadas no gosto do usuário e também no contexto atual da *playlist* sendo construída

Já o trabalho de Hu e Ogihara, 2011 tenta resolver o problema de recomendar novas músicas para um usuário, levando em conta seu gosto pessoal, ser uma música inédita e se encaixar no padrão do histórico de músicas ouvidas pelo usuário. O *feedback* do usuário durante a música recomendada ajuda a compor a próxima recomendação, visando se aproximar mais de seu gosto musical.

Uma forma de ter *feedback* sobre uma música recomendada para o usuário é analisar se ela foi reproduzida até o fim (indicando que foi uma boa recomendação) ou se foi pulada em seu início (forte indício que o usuário não gostou da recomendação).

O método utilizado determina a próxima recomendação com base em 5 pilares: gênero, ano, favorecimento ao gosto do usuário, quão nova a música é para o usuário e rotina de seu uso. Os 5 pilares são quantificados para gerar uma pontuação para cada música da coleção, onde a música de pontuação mais alta é a próxima a ser recomendada para o usuário.

Por sua vez, a pesquisa desenvolvida em Kohli, 2019 busca lidar com três problemas ao mesmo tempo: elaborar um método para prever se uma música será popular, mesmo antes de seu lançamento, encontrar fatores geográficos que possam afetar a percepção

quanto a sentimento de uma música e, também, criar um modelo para recomendações de músicas baseado no histórico do usuário. Foram utilizadas técnicas de Análise Exploratório de Dados e modelos de *Machine Learning*, resultando em uma precisão de 70%.

2.4 Tecnologias

Outra parte importante do trabalho preliminar de estudo envolve analisar tecnologias para o desenvolvimento do *front-end* e do *back-end*, para que seja possível fazer uma escolha adequada às necessidades do projeto.

2.4.1 *React*

React é uma biblioteca *JavaScript* para facilitar a construção de interfaces de usuário. Ele abstrai a construção de *HTML*, *CSS* e *JavaScript* de forma a integrar estrutura (*HTML*), estilização (*CSS*) e lógica (*JS*) em componentes, visando a reutilização e melhor organização e estruturação de código. É comumente usado para *SPAs* (*Single-Page Applications*), aplicações em que o conteúdo da página é re-renderizado de acordo com a navegação do usuário, ao invés de carregar novas páginas inteiras vindas do servidor. Isso faz com que a experiência fique mais fluida e similar ao de aplicações nativas.

A documentação do *React* (Facebook, 2021) é bastante completa e possui diversos exemplos para compreender as especificidades de sua utilização, que inclui: dominar a renderização dos elemento; *componentização* dos elementos da interface, bem como seus estados e propriedades; entender o ciclo de vida dos componentes *React* e como a informação flui dos elementos pais para os filhos. Por fim, os *Hooks* do *React* são funções que podem ser importadas ao seu código de forma a facilitar a manipulação do estado e do ciclo de vida dos componentes.

2.4.2 *Express.js*

Express.js é um *framework* para *back-end* de aplicações *web* para o *Node.js*. O *Node.js*, por sua vez, é um ambiente de *runtime* capaz de executar *JavaScript* fora de um *browser*, sendo assim é adequado para construção de *back-end* (servidores). O *Express* facilita a construção de um servidor, abstraindo funcionalidades do *Node.js* e simplificando o desenvolvimento *back-end* em aplicações web.

Dentre as abstrações de maior nível que o *Express* oferece algumas valem ser citadas: roteamento (permitindo definir quais métodos *HTTP* e *URIs* devem responder às chamadas do usuário da aplicação), criação de *middlewares* (funções que podem ser encadeadas dentro do ciclo de requisição e resposta do servidor, agindo de acordo com a necessidade da aplicação) e facilitar o *gerencimento* de erro e *debugging* do *back-end* construído. Mais detalhes podem ser encontrados na documentação oficial (OpenJS, 2021).

2.4.3 *Angular*

Angular, assim como *React*, é utilizado para construção de interfaces de usuário em aplicações web, porém possui diferenças significativas na sua filosofia de uso. Angular é um

framework TypeScript (superset de JavaScript), enquanto *React* é uma biblioteca *JavaScript*. *Frameworks* tendem a exigir do desenvolvedor uma estrutura de código mais específica, de forma que o *framework* se encarregue de usar seu código para gerar a aplicação final. A vantagem de tal abstração é que o código torna-se mais estruturado e organizado, de acordo com o padrão especificado. Para aplicações grandes isto favorece a diminuição de *bugs* de cunho arquitetural e facilita a padronização do desenvolvimento entre uma equipe de desenvolvedores. Por outro lado, a desvantagem é que agora há a necessidade de se dominar mais o *framework* para ser capaz de desenvolver, afastando-se da simplicidade inerente de se usar *HTML*, *CSS* e *JavaScript* de forma pura.

A utilização de *TypeScript* obrigatoriamente no Angular está alinhada à filosofia de priorizar organização e estruturação. *TypeScript* será explorado mais a fundo na próxima seção.

Angular possui também a Angular CLI, uma ferramenta de linha de comando que ajuda a criar automaticamente a estrutura de uma aplicação Angular, bem como a facilitar a expansão, com a criação de novos componentes. Mais detalhes sobre Angular e Angular CLI podem ser encontrados na documentação oficial (Google, 2021a).

2.4.4 *TypeScript*

TypeScript é um *superset* de *JavaScript*, ou seja, mantém todas as funcionalidades de *JavaScript* e adiciona mais elementos em cima, como tipagem estática e mais elementos de Programação Orientada a Objetos (Interfaces, por exemplo), trazendo mais robustez à linguagem para aplicações de larga escala. Pode ser tanto usada na porção que executa do lado do cliente (*front-end*), em conjunto com *frameworks* como *Angular*, quanto no lado do servidor (*back-end*), em conjunto com *Node.js*, por exemplo. Porém, TypeScript não é executado diretamente, mas sim *transpilado* para *JavaScript* automaticamente, permitindo sua execução em navegadores ou servidores Node. Mais detalhes na documentação oficial (Microsoft, 2021).

Capítulo 3

Metodologia

3.1 Planejamento do Desenvolvimento

Tendo uma motivação bem definida, conforme visto no *Capítulo 1*, a etapa de planejamento no desenvolvimento é tão essencial quanto a implementação em si. Decisões ruins no projeto tendem a se acumular e tornam-se difíceis de serem solucionadas no futuro.

Com isto em mente, também é necessário ressaltar que as necessidades de um projeto de software também vão ficando mais claras aos envolvidos no projeto, conforme ele evoluiu. Sendo assim, beber da fonte das **Metodologias Ágeis**, com a definição de *sprints* quinzenais, auxiliou a nortear o rumo do projeto. Reuniões quinzenais foram preferidas em detrimento de semanais pois permitem mais tempo para trabalhar em novas *features* dentro da mesma *sprint*, sem que o desenvolvimento se prenda de mais em um único aspecto da aplicação.

Sendo assim, as etapas do desenvolvimento podem ser resumidas com:

- Definição da *stack* tecnológica e estudos preparatórios (justificativa da *stack* e da arquitetura podem ser encontrados neste *Capítulo 3*. Estudo preparatórios descritos no *Capítulo 2*).
- Testes de APIs públicas de serviços de *streaming* (Descritos no *Capítulo 2*)
- Desenvolvimento da aplicação (*front-end* e *back-end*) (Justificativas das decisões de desenvolvimento neste *Capítulo 3* e apresentação dos resultados no próximo - *Capítulo 4*)

3.2 Decisões da arquitetura da aplicação

O *Spotunes* foi inicialmente planejado como uma plataforma centralizada, onde haveria uma autenticação de usuário para acessar sua biblioteca através de um login e senha. Porém, com base na ideia original de ser uma aplicação de organização de biblioteca musical que colocasse mais controle ao seu usuário, decidiu-se executar a aplicação o máximo possível

dentro do próprio *browser* do usuário, armazenando os dados da biblioteca no *LocalStorage* do navegador (permitindo exportar e importar arquivos *JSON* para backup da biblioteca), utilizando um *back-end* minimalista através de um servidor no *Heroku*, que funciona na prática como uma *API* simples que abstrai as funcionalidades necessárias da *API* do *Spotify*, delegando o processo de autorização do uso da *API* do *Spotify* (que requer uma chave de cliente e outra chave secreta da aplicação).

Dessa forma, a aplicação se torna mais ágil e responsiva, pois depende apenas do que está sendo executado localmente no *browser*, inspirando-se no que seria uma aplicação nativa para o sistema do usuário. A vantagem de ser uma aplicação web que roda em um *browser* é a alta compatibilidade, facilitando que usuários tenham acesso à mesma aplicação independente da plataforma em que se encontrem, o que simplifica o desenvolvimento da aplicação.

3.3 *Wireframe*

Um *wireframe* é um esboço simples da interface de usuário de uma aplicação web. Trata-se de uma boa prática fazer tal prototipagem pois ajuda a definir um design inicial aproximado para a aplicação que será desenvolvida. O *wireframe* feito para o *Spotunes* está apresentado na Figura 3.1[1].

Figura 3.1: Wireframe *para o* Spotunes

[1] Este wireframe pode ser conferido integralmente em https://www.figma.com/file/v0aTcp8ozSLX53JhDiIo6Z/Spotunes-Wireframe

3.4 Escolha de *React* para o *Front-End*

A escolha de *React* para desenvolver a interface de usuário (*front-end*) se deu por conta da relevância e flexibilidade dessa tecnologia. *React* é o *framework* web mais popular utilizado atualmente como pode ser visto na Figura 3.2, baseada no 2021 *Stack Overflow Survey* (StackOverflow, 2021).

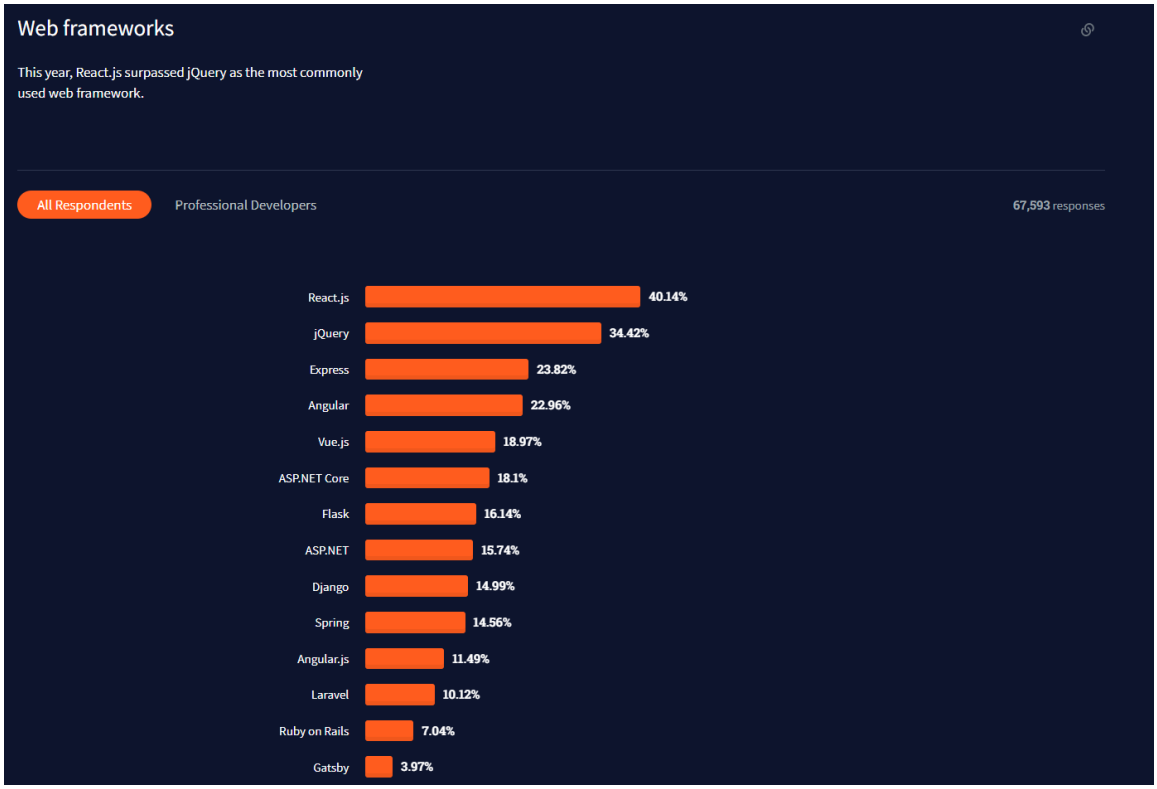

Figura 3.2: *Porcentagem de* web developers *que usam* React *em 2021*

A popularidade crescente de *React* reflete em maior quantidade de oportunidades no mercado de trabalho, bem como uma vasta gama de bibliotecas de terceiros e documentações, tutoriais e fóruns disponíveis para dúvidas de desenvolvimento. Quanto à flexibilidade, *React* permite estruturar com maior liberdade a aplicação, facilitando o desenvolvimento. Isto pode vir a custo de um pouco de organização e estruturação do código, mas é mitigado pelo fato de ser um desenvolvimento onde apenas um desenvolvedor está envolvido.

3.5 Decisões quanto ao desenvolvimento

3.5.1 Funcionalidades de busca e adição na biblioteca

Por conta da necessidade de efetuar buscas na *API* do *Spotify*, não foi possível manter o *Spotunes* livre de um *back-end* separadamente, pois a *API* do *Spotify* requer um *token* de autorização. A biblioteca do *npm* (gerenciador de pacotes do *Node.js*) *node-spotify-api* ajuda a abstrair vários processos do fluxo de autenticação, mas ainda é necessário um servidor

para gerenciar os *tokens* (e, preferencialmente, não deixá-los expostos ao *front-end*). Há tutoriais de uso da *API* do *Spotify* disponíveis em diversos sites e em Batistoti, 2020.

Quanto à parte visual: ao clicar na caixa de pesquisa, o usuário vê um modal sendo aberto com duas músicas por linha, que utilizam um *card* do *Bootstrap* para mostrar a imagem da arte do álbum e informações como o nome do artista e álbum, junto de um botão de adicionar na biblioteca. Ao clicar neste botão, a música é adicionada na biblioteca salva no *LocalStorage* do navegador do usuário.

3.5.2 Integração do *front-end* com o *back-end* em *production*

O servidor simplificado que gerencia a utilização da *API* do *Spotify* teve seu *deploy* para produção no *Heroku* (em https://spotunes-server.herokuapp.com/, enquanto que o *front-end* teve *deploy* no *Netlify* (https://spotunes.netlify.app/).

Essa dissociação em dois serviços diferentes de hospedagem foi feita pois o pacote básico do *Heroku* acaba hibernando o servidor ao detectar que está ocioso, enquanto que o *Netlify*, por servir páginas estáticas, ou seja, necessitando muito menos de poder de processamento dos servidores do serviço de hospedagem, está sempre disponível. Isso permite que um usuário tenha acesso fácil e constante ao *Spotunes*, enquanto os dados ficam salvos no seu próprio navegador e podem ser facilmente exportados para ter um backup. A primeira busca no *Spotunes* depois de um longo período sem uso de usuário algum acaba por demorar um pouco de tempo a mais, pois o *Heroku* irá acordar o processo que estava hibernando.

Como estamos consumindo dados de uma *API* externa (no *Heroku*) ao *front* (localizado no *Netlify*), é natural que os mecanismos de segurança do navegador gerem um erro de *CORS*, portanto os *headers* das requisições e resposta do *back-end* precisam de um *allow* (permissão) para a URL adequada do *front-end* no *Netlify*.

3.5.3 Variáveis de ambiente

O *back-end* do *Spotunes* requer que sejam configuradas dois *tokens* como variáveis de ambiente: o *SPOTIFY_CLIENT_ID* e o *SPOTIFY_CLIENT_SECRET*. Ambos os *tokens* podem ser obtidos do portal do desenvolvedor da *API* do *Spotify*.

3.5.4 Decisões da interface de usuário

- Cursor se altera para *"pointer"* ao fazer *hover* sobre a tabela de músicas para indicar que ela é clicável
- A tabela de música deve ser um componente *React*
- O *iframe* para reprodução de músicas do *Spotify* deve ser um componente *React*
  - Com nenhuma música ainda selecionada, o *iframe* é substituído por uma imagem *placeholder*
  - Tema escuro no *iframe* deixou a identidade visual mais coesa com o resto da interface de usuário

- *Favicon* da página foi pego do próprio site do *Spotify*
- Título da página definido como *"Spotunes"* apenas
- Paleta de cores usando a variante *"dark"* do *Bootstrap* sempre que possível (botões, *hover*, etc)
- Componentizar *ListGroups* das *Sidebars*
  - *Padding* nos títulos
  - Componente recebe como *props* uma *string* que representa o título da seção e um *array* com os itens da lista
- Componentização dos botões do *Player* (pular música e *stop*)
- Componentização do *forms* de busca junto com sua lógica para buscar, invocando o *back-end*
- Componentização do modal de busca junto com a lógica da adição na biblioteca
- Separar cada seção da interface com bordas (*navbar*, *sidebars*, *main content* com a tabela de músicas e o *footer*). Facilita a visualização de cada componente de maneira separada.

3.5.5 Persistência dos dados na biblioteca

Para manter o *Spotunes* mais ágil e com maior controle do usuário sobre a aplicação, tentou-se minimizar a necessidade de um *back-end*, portanto uma das decisões cruciais para isso foi armazenar os dados localmente no *LocalStorage*.

Ao abrir a aplicação pela primeira vez, a biblioteca está vazia e mostra-se um *Alert Warning* do *Bootstrap* para instruir o usuário a adicionar novas músicas. No *footer* é mostrado a quantidade de itens atual na biblioteca, bem como sua duração total.

As músicas na biblioteca são salvas através de um estado do *React* (*songData*) com o uso do *hook useState*. Através do *hook useEffect*, pode-se definir um comportamento toda vez que o estado *songData* for alterado. No caso, deseja-se que *songData* seja salvo no *LocalStorage* toda vez que for alterado.

Para carregar os dados do *LocalStorage* ao abrir a aplicação foi criada uma função *loadSongData* que busca a informação armazenada em *string* no *LocalStorage* e *parseia* para *JSON*, atribuindo ao estado *songData*. Para conseguir efetuar essa execução apenas quando a aplicação carregar, pode-se utilizar o *hook useEffect* também, porém marcando uma variável vazia, o que significa que é executado apenas na primeira renderização do *React*.

3.5.6 Contagem de reproduções de músicas

Por questões de segurança, *iframes* não permitem o uso de *listeners*, dificultando o monitoramento de um clique do mouse (que seria o indicador de que a música foi reproduzida

pelo *iframe* do *Spotify*). Porém, há uma maneira de solucionar isso. [StackOverflow, 2010]

Basicamente, clica se um *interval* do *JavaScript* que executa uma função a cada 100 milissegundos. Esta função irá verificar se há um elemento focado atualmente na página e caso ele seja do tipo *iframe* executa a função *built-in* do *JavaScript blur*, que remove o foco do elemento focado atualmente. Dessa forma, se o *iframe* foi clicado, o foco estará nele e com isso podemos assumir que isso se comporta de maneira aproximada a um *listener* de clique. A limitação óbvia seria que um clique em menos de 100 milissegundos de outro não seria registrado, mas pelo fato de que queremos registrar apenas uma reprodução de faixa musical para cada carregamento de música no *iframe*, este comportamento pode ser ignorado sem prejudicar as regras de negócio da aplicação.

3.5.7 Adicionar à *playlist*

Por padrão, o *Spotunes* carrega 5 playlists inteligentes para servirem de exemplo ao usuário (Mais tocada, recentemente adicionadas, recentemente modificadas, nunca tocadas e as melhores classificadas).

Para adicionar uma música a uma *playlist*, o usuário pode clicar em uma música da tabela de músicas e isso fará com que os detalhes da música sejam abertos na *sidebar* à direita. Um dos botões na *sidebar* será o *"Add to playlist"*. Ao clicar neste botão, um modal com um formulário contendo *checkboxes* é aberto. O usuário deve clicar em todas playlists em que desejar adicionar tal música.

O estado de cada *checkbox* é gerenciado por um *array* de estados do *React*, dessa forma pode-se facilmente determinar quais playlists foram marcadas para adição. As playlists são salvas em um array de objetos, onde cada objeto representa uma *playlist*, contendo: *uuid*, título da *playlist*, *booleana* indicando se é *playlist* inteligente ou não e um *array* de *IDs* de músicas. O id que identifica cada música na biblioteca é o próprio ID utilizado pelo *Spotify* em sua *API*.

3.5.8 Rotas para cada *playlist*

Ao clicarmos em uma *playlist* na *sidebar* à esquerda, queremos filtrar o conteúdo da *SongTable* com apenas as músicas da *playlist* atual. Para isso podemos navegar para uma rota na forma */playlist/:id*, onde *:id* representa o id da *playlist* clicada. Ao navegar para uma nova rota, um estado chamado *filteredSongData* armazena as músicas de *songData* que cumpram os requisitos de filtragem da rota atual. As informações do *footer* (quantidade de músicas e duração total) também deve ser alteradas para as informações do conteúdo filtrado.

A biblioteca usada para o roteamento foi *react-router-dom*. Para identificar a rota atual em que a aplicação se encontra, pode-se utilizar *useLocation().pathname* e assim pode-se marcar o elemento ativo correspondente na *sidebar* à esquerda.

Como o *Spotunes* foi desenvolvido como um SPA (*Single-Page Application*), clicar em um link dentro dele não deve recarregar a página, mas sim usar o roteamento fornecido pela biblioteca *react-router-dom*. Ao clicar em um elemento de navegação *ListGroupItem*

na *sidebar* esquerda, a página estava sendo recarregada. A solução para isso foi remover os *href* dos *ListGroupItem*, colocando os *ListGroupItem* ao redor de uma *tag LinkContainer* da biblioteca *react-router-bootstrap*.

A *tag LinkContainer* do *react-router-bootstrap* foi necessário pois a *tag* convencional Link do *react-router-bootstrap* não é compatível com o uso de *bootstrap* ao mesmo tempo, causando problemas e gerando *reload* na página, ferindo o princípio de um SPA. Porém, usando *LinkContainer*, o elemento ativo do *ListGroupSection* se torna problemático, não funcionando para marcar a rota atual. A solução foi checar a rota atual com *useLocation().pathname* e alterar o CSS do *ListGroupSection* e do *ListGroupItem* manualmente de acordo com a lógica adequada.

Também convém deixar a interface intuitiva. Um exemplo seria ao fazer *hover* em um *ListGroupItem*, a cor de fundo desse item deve ser alterado para sugestionar ao usuário que ele pode clicar para ir para outra rota. *React* não possui um *listener* de *onHover*, mas pode facilmente ser substituído por um comportamento de *onMouseEnter* e análogo para *onMouseLeave*.

3.5.9 Nodemon

O *nodemon* é uma dependência de desenvolvimento bastante útil para o *back-end*, pois permite que o servidor seja reiniciado automaticamente com as alterações feitas toda vez que o arquivo é salvo no editor de texto.

Como ele já foi configurado como dependência de desenvolvimento no *package.json* do repositório do *back-end*, será instalado automaticamente ao executar *npm install* para instalar todas as dependências.

3.5.10 Rota de Álbuns

Ao clicar em *Albums*, o usuário é direcionado à rota */albums* e tem listado todos os álbuns em sua coleção, com imagem da capa, nome do artista, nome do álbum e um botão *Go To Album*, para filtrar apenas as músicas pertencentes a esse álbum.

O botão *Go To Album* precisa ser envolto em um *LinkContainer* para evitar o *refresh* do SPA. Ao ser clicado, a aplicação é direcionado para a rota *albums/:id* e a *SongTable* é carregada apenas com as músicas filtradas do álbum atual, ordenadas em ordem alfabética. Na rota de álbuns, o *footer* mostra a quantidade de álbuns encontrados na biblioteca.

3.5.11 Rota de Artistas

Ao clicar em *Artists*, o usuário é direcionado à rota */artists* e tem listado todos os artistas de sua biblioteca podendo clicar em um artista para ser redirecionado a */artists/:id*, de maneira análoga à rota de álbuns e à rota de um álbum específico. Na rota de artistas, o *footer* deve mostrar a quantidade de álbuns encontrados na biblioteca.

3.5.12 Rota de Gêneros

Ao clicar em *Genres*, o usuário é direcionado à rota */genres* e tem listado todos os gêneros musicais de sua biblioteca podendo clicar em um gênero para ser redirecionado a */genres/:genre-name*, de maneira análoga à rota de álbuns e à rota de um álbum específico. Uma limitação de *API* do *Spotify* é que os gêneros musicais não são ligados a uma música específica, mas sim a um artista. Uma solução para isso foi estabelecer que ao adicionar uma música, todos os gêneros do artista são atribuídos a ela.

Para salvar os dados do artista (URL da imagem do artista e seus gêneros), é feita uma requisição separada à *API* do *Spotify* após o usuário clicar para adicionar uma música à biblioteca. Como os gêneros são únicos, não é necessário um ID para identificá-los, bastando usar *:genre-name* na rota. Na rota de gêneros, o *footer* deve mostrar a quantidade de álbuns encontrados na biblioteca.

3.5.13 Playlist Inteligente - Músicas Recentemente Adicionadas

Uma música é considerada como "recentemente adicionada"se esta adição ocorreu nas últimas 24 horas. Para ser capaz de checar isso, foi adicionado um campo *createdAt* ao adicionar as músicas na biblioteca. Um problema enfrentado foi que ao usar o *new Date()* do *JavaScript* para criar a data e ao salvar no *LocalStorage*, o objeto é convertido para *string*, sendo assim, traz uma complexidade de lógica adicional para *parsear* essa *string* e converter para *number* novamente, de forma que seja possível efetuar cálculos em cima. A solução ideal para isso foi converter a data por meio da função *valueOf()*, que retorna o tempo passado da data em questão desde 01/01/1970, em milissegundos. Assim, é uma tarefa simples calcular se a música foi adicionada nas últimas 24 horas, pois *JSON* aceita que sejam salvas propriedades com o tipo *number*.

3.5.14 Playlist Inteligente - Mais Tocadas

Esta *playlist* filtra as 25 músicas mais tocadas da biblioteca (desde que possuam ao menos uma reprodução) e as lista em ordem decrescente (da mais reproduzida para a menos reproduzida). Para músicas com a mesma quantidade de reproduções, estas são ordenadas em ordem alfabética. A *SongTable* foi alterada para mostrar também a estatística de quantidade de reproduções em cada música.

3.5.15 Playlist Inteligente - Músicas Recentemente Modificadas

Playlist desenvolvida de maneira análoga à *playlist* inteligente "Músicas Recentemente Adicionadas", bastando adicionar um campo *modifiedAt* no objeto que representa cada música, tanto no momento da adição da música na biblioteca, quanto em qualquer eventual edição de seus atributos.

3.5.16 Playlist Inteligente - Nunca Tocadas

Playlists que filtram todas as músicas da biblioteca com quantidade de reproduções igual a zero. Como o propósito dessa *playlist* é convidar o usuário a de fato ouvir tais músicas, não há um limite fixo como na *playlist* "Mais tocadas". De maneira análoga, as músicas são ordenadas por ordem alfabética.

3.5.17 Playlist Inteligente - Músicas com Melhor Classificação

Cada música na biblioteca possui um campo *rating* que é um *number* de 0 a 5 (representam de nenhuma a 5 estrelas). Músicas com 0 estrelas são consideradas não classificadas ainda. Para o componente de estrelas clicáveis pelo usuário no campo determinado da *SongTable* foi utilizada a biblioteca do *npm*, *react-rating-stars-component*, que já oferece um *listener* de *onChange* que é ativado quando o usuário clica em uma da quantidade de estrelas possível.

Para ser possível utilizar o *listener onChange* da biblioteca adequadamente, foi preciso modularizar melhor o código, quebrando o componente *SongTable* em subcomponentes, considerando cada linha da tabela de músicas um *SongTableRow*, responsável por lidar com as informações de apenas uma música, recebendo o *spotifyID* como *prop*. Tendo o id, tornou-se possível atualizar o *rating* no *LocalStorage* através do *onChange*.

Quanto à implementação da *playlist* inteligente em si, a filtragem foi trivial por ser análoga aos outros exemplos. Primeiramente, bastou checar qual era o *rating* máximo presente na biblioteca (geralmente 5 na maior parte dos casos). E filtrar todas as músicas que possuam *rating* igual a esse maior *rating* encontrado.

3.5.18 Feature de backup da biblioteca

Os botões para exportar e importar a biblioteca foram colocados na *sidebar* esquerda, acima das opções de *filtagrem* da biblioteca. Para tornar suas funcionalidades visualmente intuitivas, foram colocados ícones equivalentes a download e *upload*, provenientes do *Bootstrap Icons*. Há várias formas de incluí-los no código, preferiu-se copiar todo o *SVG* que gera os ícones para tentar manter a aplicação menos dependente de um ícone que é carregado de uma fonte externa.

Ao clicar em exportar, o usuário recebe um *prompt* de seu sistema operacional perguntando onde deseja salvar o arquivo *library.json* gerado a partir do *songData*, que representa os dados de sua biblioteca. A implementação da função de exportar é bem simples utilizando recursos nativos do próprio *JavaScript*: converte-se *songData* para *string* para representar o arquivo *JSON*, cria-se um *Blob* de texto a partir deste, *linkando* este *blob* ao resultado de clicar em um botão de download genérico e invisível criado apenas no DOM e não associado à página em si.

A função de importar é similar do ponto de vista de usabilidade, mas por trás dos panos é bem mais complexa. Quando o usuário clica no botão de importar, também recebe um *prompt* para selecionar o arquivo *library.json* exportado em outrora. A implementação se dá criando um elemento *HTML* de *input* com tipo *file*, de forma que o *display* seja *none* (pois queremos visualizar apenas o botão de *import*). Utiliza-se o objeto *FileReader*, nativo

do *JavaScript*, que ao ler o arquivo presente no *input* invisível, irá parseá-lo como *JSON* e após isso atribuir o estado de *songData* com o objeto resultante do *parseamento*.

3.5.19 Criação de novas playlists

O usuário pode criar novas playlists clicando no botão *New Playlist* na *sidebar* esquerda. Ao clicar, um modal é carregado, contendo um *form* com campo de texto para o título da *playlist* a ser criada.

Como o usuário pode criar quantas playlists quiser, foi adicionado um comportamento de *scroll* quando ocorrer *overflow* na *sidebar*. Além disso, cada nova *playlist* criada é adicionada ao topo da *ListGroupSection* de playlists (logo abaixo do botão *New Playlist*, de forma que seja de fácil o usuário ter um *feedback* visual que sua nova *playlist* foi criada como ele pretendia.

3.5.20 Botões de edição e remoção de *playlist*

Os botões de edição e remoção de *playlist* foram adicionados logo acima da *SongTable* nas rotas que forem referentes a playlists em si.

O botão de remoção pede uma confirmação com um *popup* do próprio navegador. Após a confirmação do usuário, filtra as playlists, restando apenas aquelas que não são a atual, finalmente atualizando o estado das playlists e redirecionando para a página inicial do *Spotunes*.

O botão de editar *playlist* abre um modal com o *forms* de edição da *playlist*. Foi criado um componente similar ao modal de criação de *playlist*, porém especializado na edição (que consiste em já carregar o valor atual do título da *playlist* no *input*, gerenciado através do *useState* do *React*. Ao digitar ou apagar no *input*, um campo *onChange* estabelece o novo valor do estado *React*, que serve como valor para o campo).

O componente *SongTable* gerencia os botões de edição e remoção bem com suas respectivas funções a serem executadas ao serem clicados. Para obter o id da *playlist*, presente na rota, foi utilizado o *hook useParams*(). Já o *hook useHistory* é usado para redirecionar da rota atual da *playlist* deletada para a rota inicial da aplicação.

3.5.21 Botões do *player* de música

Por limitações de reprodução impostas pelo *iframe* e pelo *browser*, funcionalidades como *Pause* e *Play* não foram colocados no *Player* na *Navbar* (pois as mesmas podem ser efetuadas pelo próprio *iframe* do *Spotify*) e nem um *slider* para ajustar o volume, pois nem o *iframe* e nem o *browser* permitem interagir com o áudio sendo reproduzido.

Os botões de fato utilizados foram os de pular música (para frente e para trás, navegando ciclicamente entre a seleção atual de músicas na *SongTable*) e um botão de *stop*, que remove o *iframe* do *Spotify* ativo atualmente, voltando com a imagem de *placeholder* no lugar. Ao passar com o mouse em cima de um dos botões, o cursor do usuário se torna um *pointer* e o fundo dos botões se destaca, para sugerir ao usuário que são componentes utilizáveis.

Para a funcionalidade de stop, o componente *MusicPlayerControls* precisa receber o estado *playerSongID*, que guarda a ID da música do *Spotify* para o elemento atual a ser mostrado no *iframe*. De maneira análoga a outros ícones utilizados na aplicação, os botões de *skip* e de *stop* são ícone do *Bootstrap Icons*, tendo o *svg* copiado para não depender de carregá-los de uma fonte externa.

Botões de *skip*

Os botões de pular para música seguinte ou anterior se baseiam no estado *currentSongIndex*, que é inicializado com -1 e tem seu valor somado ou subtraído, de acordo com o botão clicado. Resulta em um fluxo cíclico entre a atual seleção de músicas mostradas (*filteredSongData*)

A música selecionada é mostrada no *iframe* e tem uma *badge* "*Active*"marcada na *SongTable*. A *Badge* não foi utilizado como componente do *react-bootstrap* pois há algum *bug* na versão atual da biblioteca, que não está compilando o arquivo *Badge.js*. Utilizar a classe *badge* do *Bootstrap* convencional resolveu esse problema e o funcionamento é idêntico ao esperado. O *bug* não foi resolvido nem mesmo atualizando da versão *2.0.0* para *2.0.2* do *react-bootstrap*.

Para tornar consistente, foi padronizado usar a *filteredSongData* para todas músicas listadas na *SongTable*, mesmo quando o filtro é o "/*all*". Assim o ciclo de pular música mantém-se funcional e consistente. Ao clicar múltiplas vezes rapidamente no botão de pular música, o *iframe* acaba ficando "selecionado"em azul pelo *browser*, esse comportamento pode ser resolvido com a propriedade *user-select* no CSS, com valor *none*.

3.5.22 *Badge* para reproduzir música

Cada *SongTableRow* tem uma *badge* de *play* que age como um botão para configurar o *iframe* do *Spotify* com a música clicada. Ao ser clicado a *badge* se transforma em uma *Active* para o item atual.

3.5.23 Informações de músicas na *sidebar*

A *sidebar* à direita no *Spotunes* é responsável por mostrar informações mais detalhadas de uma música escolhida na *SongTable*. Ao iniciar a aplicação, a *sidebar* mostra um *warning*, explicando que deve-se clicar na *badge* "*Show*"em alguma música da biblioteca para mostrar as informações na própria *sidebar*.

Ao clicar em "*Show*"em alguma música, a *sidebar* passa a mostrar as ações disponíveis para a música atual (Adicionar a uma *playlist*, editar informações da música e deletar da biblioteca), junto com um *card* com as informações relevantes ao usuário (capa do álbum, nome da música, artista, nome do álbum, duração, data de criação e modificação, quantidade de reproduções e classificação).

Clicar no "*Show*"de uma música faz com que a aplicação busque o objeto em *songData* que contém o mesmo id do elemento clicado, adicionando o objeto encontrado ao estado do *React* que guarda a música selecionada.

Um problema resolvido foi que a data armazenada no *songData* tanto para criação quanto modificação está em milissegundos desde 01/01/1970, por isso para mostrar um valor mais expressivo ao usuário (no formato "ano/mês/data", junto com o horário no formato "hora:minuto") foi criada a função *formatDateString*() para auxiliar na conversão.

Além disso, foi criado o componente *SongInfoSidebar* para extrair a *sidebar* responsável pela informação das músicas, agregando os estados correspondentes e as funções auxiliares para melhorar a organização do código.

3.5.24 Comportamento de *scroll* durante *overflow*

As *sidebars* e a *SongTable* estavam com a propriedade de *overflow* atribuída como *scroll*, para criar barras de rolagem quando seu conteúdo não coubesse no espaço que lhes foi reservado, porém a propriedade *overflow* atribui um scroll horizontal, que ficava inutilizado e tomando espaço na tela. Isso é facilmente resolvido trocando por *overflow*-y, que cria apenas a barra de rolagem vertical, aproveitando melhor o espaço da tela para a aplicação.

Capítulo 4

Resultados

Ao longo deste capítulo serão apresentadas as telas e resultados finais alcançados com o desenvolvimento do *Spotunes*, bem como breves explicações sobre cada componente e usabilidade do seu *front-end* e também do *back-end*.

O código resultante do desenvolvimento encontra-se hospedado no *Github*, bem como suas instruções de uso nos próprios repositórios. Seguem os links para o *front-end* e *back-end*, respectivamente:

- https://github.com/darthHunterous/spotunes
- https://github.com/darthHunterous/spotunes-server

4.1 Resultados do *Front-End*

4.1.1 Tela Principal

A tela mostrando como é o uso geral do *Spotunes*, rodando no navegador e exibindo a rota raiz de uma biblioteca de usuário, pode ser conferida na Figura 4.1.

4.1.2 *Navbar*

A *Navbar* (mostrada na Figura 4.2) é composta por 3 componentes *React*: *MusicPlayerControls*, *SpotifyiFrame* e *SearchForm*.

MusicPlayerControls

O componente *MusicPlayerControls* (mostrado na Figura 4.3) é composto por um botão de voltar para a música anterior, outro de *stop* e outro de pular para a música seguinte, respectivamente. Ao fazer *hover* sobre um dos botões o cursor se torna um *pointer* e o fundo do botão se torna mais escuro para sugerir ao usuário que pode ser clicado.

Botão de voltar para música anterior Ao ser clicado determina a música imediatamente anterior na *SongTable* como sendo a música ativa. A *SongTable* é cíclica, caso este

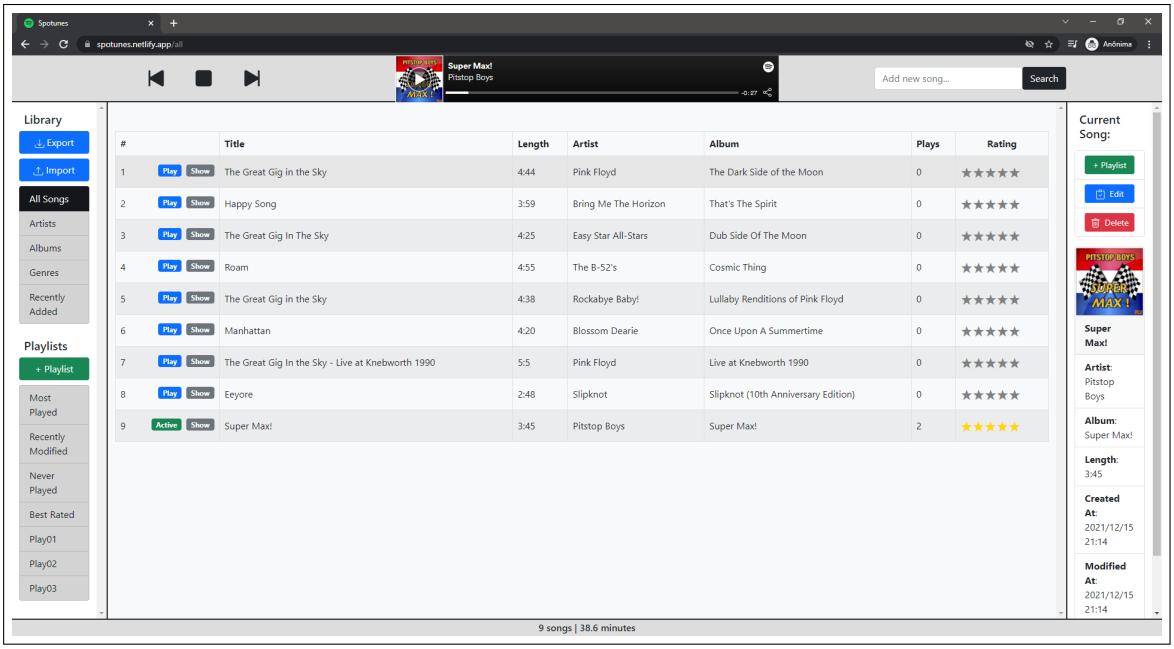

Figura 4.1: *Tela principal do* Spotunes

Figura 4.2: Navbar *do* Spotunes

Figura 4.3: Hover *do usuário sobre um botão do* MusicPlayerControls

botão seja clicado quando a primeira música está ativa, a última música tornar-se-á ativa. Este comportamento está exemplificado nas Figuras 4.4 e 4.5.

Figura 4.4: SongTable *antes do botão de voltar ser clicado*

Figura 4.5: SongTable *depois do botão de voltar ser clicado*

Botão de *stop* Ao ser clicado remove o *iframe* sendo reproduzido atualmente, colocando o *placeholder* no lugar. Este comportamento está exemplificado nas Figuras 4.6 e 4.7.

Figura 4.6: Iframe *reproduzindo música, antes do clique no* stop

Figura 4.7: Placeholder *do* iframe*, depois do clique no* stop

Botão de pular para música seguinte Comportamento análogo ao botão de voltar para música anterior.

SpotifyiFrame

O componente *SpotifyiFrame* renderiza o *iframe* fornecido pelo *Spotify* para reprodução de uma música, se há uma música selecionada. Caso contrário, renderiza uma imagem de *placeholder*. Este comportamento está exemplificado nas Figuras 4.8 e 4.9.

Figura 4.8: *Exemplo de* iframe *fornecido pelo* Spotify

Figura 4.9: Placeholder *do* iframe*, quando não há música selecionada*

SearchForm

O componente *SearchForm* (mostrado na Figura 4.10) contém uma caixa de busca para o preenchimento do título da música que deve ser buscada na *API* do *Spotify* através do *back-end* do *Spotunes*. Ao clicar em buscar, o *front-end* faz a requisição para o *back-end*, que por sua vez encaminha a requisição para a API do Spotify. O *back-end* retorna ao *front* os 10 resultados mais relevantes por padrão. O *front* por sua vez abre o componente *SearchResultModal* (mostrado na Figura 4.11) para exibir os resultados ao usuário, que pode adicionar novas músicas clicando no botão *"Add To Library"* da música desejada.

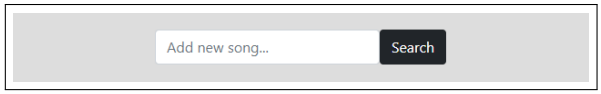

Figura 4.10: *Componente* SearchForm*: campo de busca de novas músicas*

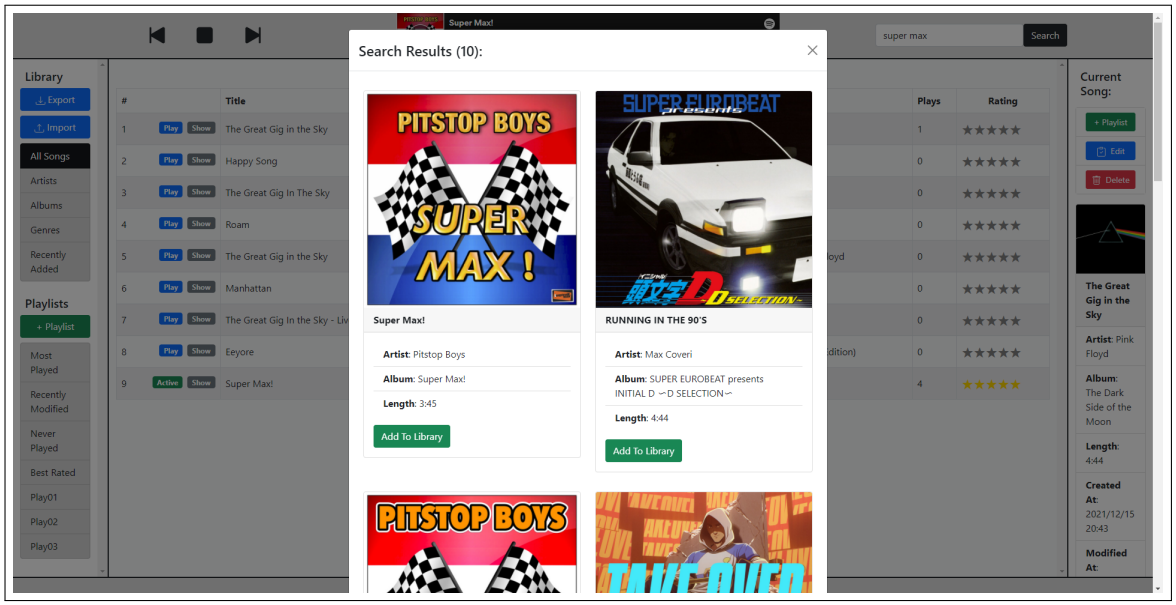

Figura 4.11: *Modal mostrando os resultados da busca retornados pelo* back-end

4.1.3 *Main content*

O conteúdo principal da aplicação fica na sua maior porção, centralizada. É ocupado por um de quatro componentes possíveis, dependendo da rota em que a aplicação se encontra atualmente. Para a rota */albums*, é renderizado o componente *AlbumsList*. Para a rota */artists*, o componente *ArtistsList*. Para a rota */genres*, o componente *GenresList* e em qualquer outra rota, o componente *SongTable*.

AlbumsList

Este componente mostra um *Alert* (exemplificado na Figura 4.12) informando que não há álbuns ainda na biblioteca do usuário, requisitando a adição de novas músicas (se a biblioteca está vazia) ou mostra a lista dos álbuns presentes na biblioteca (comportamento exemplificado na Figura 4.13), cada um sendo um *card* com imagem, nome do álbum e nome do artista, contendo também um link para a rota que filtra apenas as músicas do álbum selecionado.

ArtistsList

Este componente mostra um *Alert* (exemplificado na Figura 4.14) informando que não há artistas ainda na biblioteca do usuário, requisitando a adição de novas músicas (se a biblioteca está vazia) ou mostra a lista dos artistas presentes na biblioteca (comportamento exemplificado na Figura 4.15), cada um sendo um *card* com imagem e nome do artista, contendo também um link para a rota que filtra apenas as músicas do artista

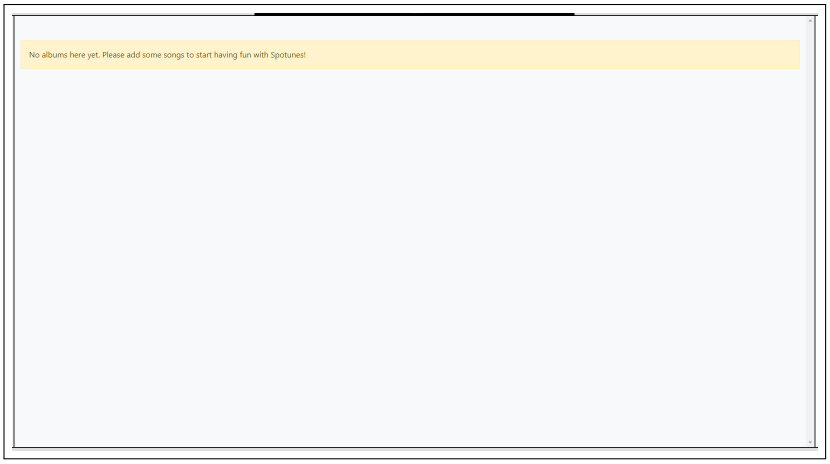

Figura 4.12: *Componente* AlbumsList *quando a biblioteca está vazia*

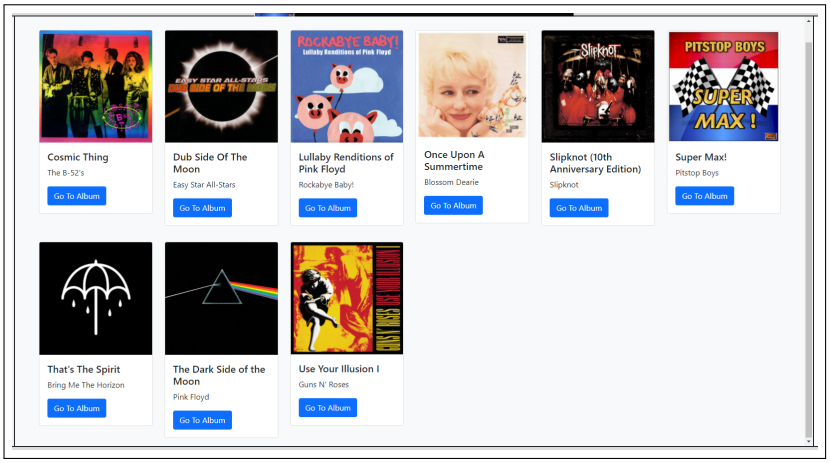

Figura 4.13: *Componente* AlbumsList *quando há conteúdo na biblioteca para ser mostrado*

selecionado.

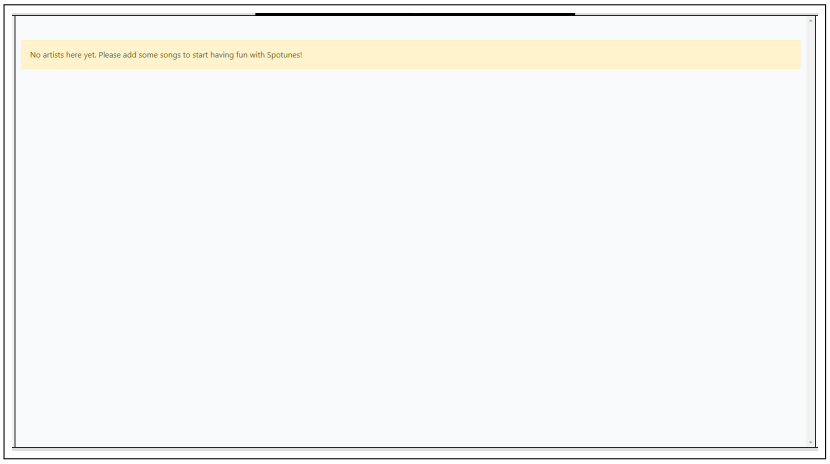

Figura 4.14: *Componente* ArtistsList *quando a biblioteca está vazia*

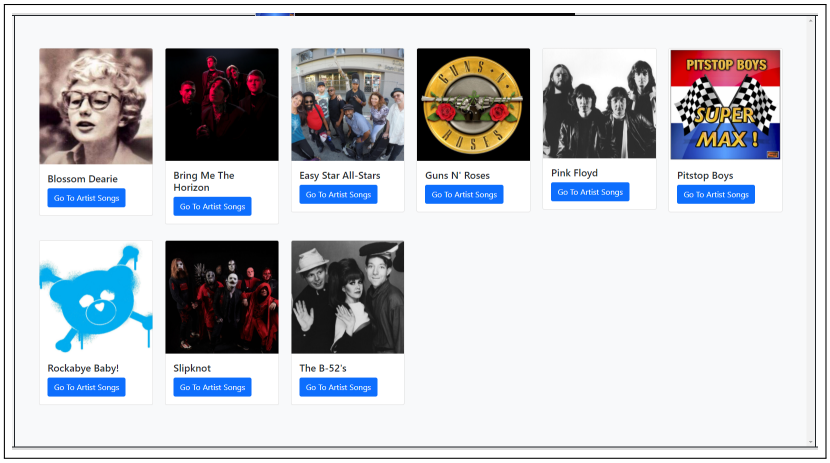

Figura 4.15: *Componente* ArtistsList *quando há conteúdo na biblioteca para ser mostrado*

GenresList

Este componente mostra um *Alert* (exemplificado na Figura 4.16) informando que não há gêneros musicais listados ainda na biblioteca do usuário, requisitando a adição de novas músicas (se a biblioteca está vazia) ou mostra a lista dos gêneros presentes na biblioteca (comportamento exemplificado na Figura 4.17), cada um sendo um *card* com o nome do gênero, contendo também um link para a rota que filtra apenas as músicas do gênero selecionado.

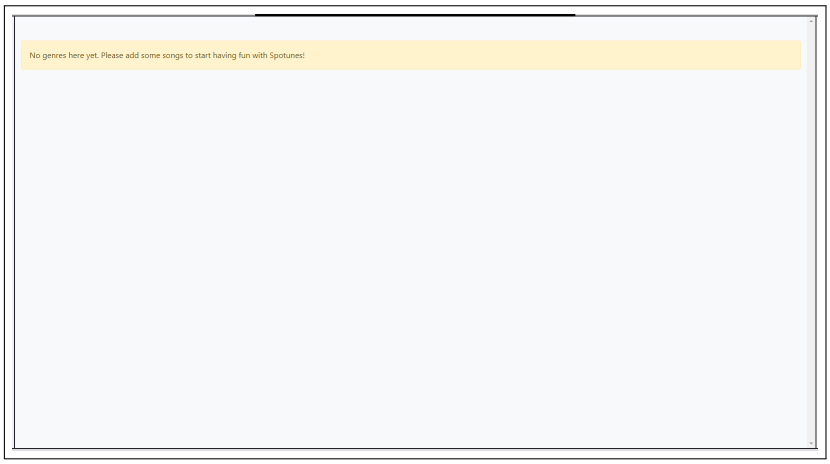

Figura 4.16: *Componente* GenresList *quando a biblioteca está vazia*

SongTable

O componente *SongTable* renderiza um *header* personalizado (mostrado na Figura 4.18) se estamos na rota de uma *playlist*, com botões para editar e apagar a *playlist* atual. Ao clicar no botão *Edit Playlist*, é aberto o modal *PlaylistEditFormModal* (mostrado na figura Figura 4.19) , contendo um *form* para edição do título da *playlist* em questão. Já ao clicar no botão *Delete Playlist*, é aberto um *prompt* de confirmação do próprio navegador quanto à deleção (mostrado na Figura 4.20).

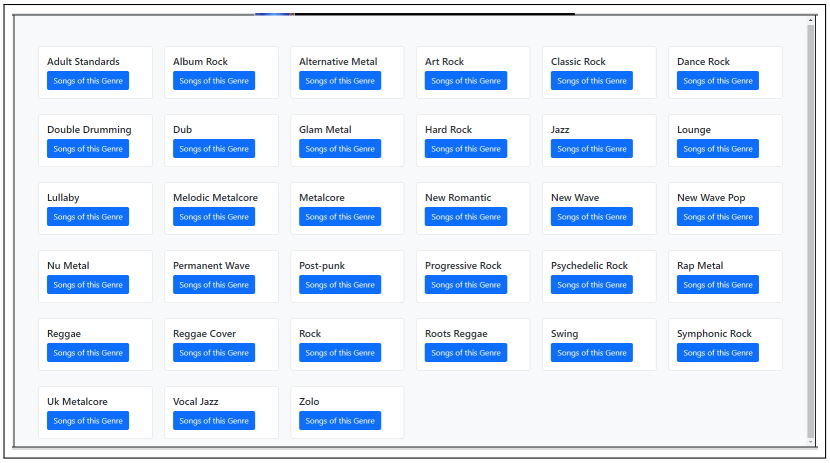

Figura 4.17: *Componente* GenresList *quando há conteúdo na biblioteca para ser mostrado*

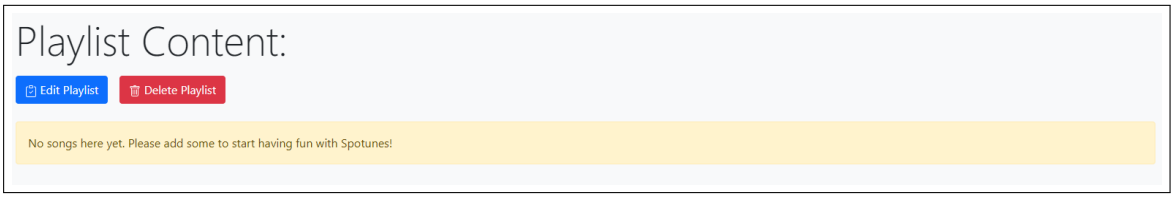

Figura 4.18: Header *da* SongTable *em uma rota de* playlist, *com botões de edição e deleção*

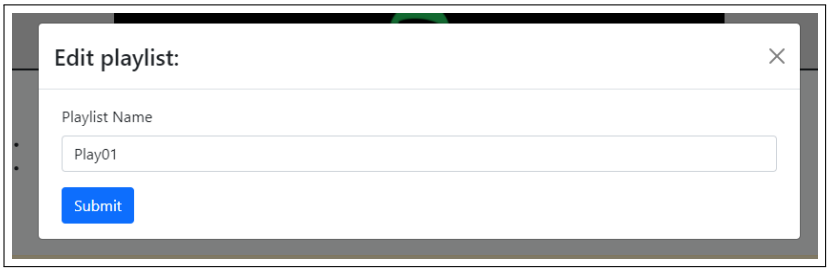

Figura 4.19: *Modal com o* form *de edição. Mostrado ao clicar no botão* "Edit Playlist"

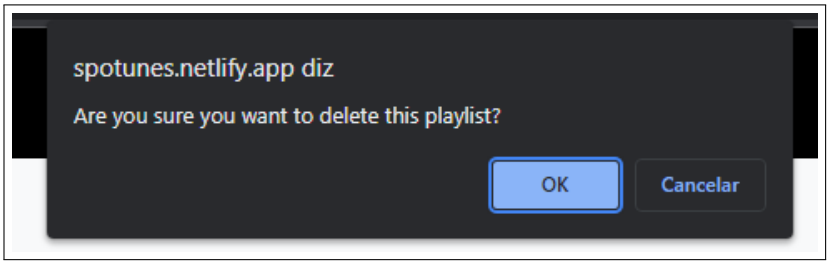

Figura 4.20: *Confirmação da deleção da* playlist *atual, ao clicar no botão* "Delete Playlist"

Se não há músicas na biblioteca, é mostrado um *Alert* (exemplificado na Figura 4.21) informando que não há música ainda na biblioteca do usuário, requisitando a adição de novas músicas. Caso haja músicas, é renderizado um componente *SongTableRow* (mostrado na Figura 4.23) para cada música da filtragem atual da biblioteca, que pode ser visualizada na Figura 4.22, contendo as seguinte colunas: número da música na seleção atual (junto com *badge buttons* de *Play* e *Show*, ou *Active* (este último exemplificado na Figura 4.24),

caso seja a música selecionada para reprodução), título da música, duração, nome do artista, nome do álbum, quantidade de reproduções e classificação em estrelas.

Figura 4.21: Alert *avisando ao usuário que a biblioteca está vazia e que deve adicionar novas músicas.*

Figura 4.22: *Componente* SongTable *mostrando todas as músicas da biblioteca, na filtragem da rota* /all.

Figura 4.23: *Exemplo de componente* SongTableRow *listando as informações de uma música.*

Figura 4.24: *Componente* SongTableRow *com* badge *de* Active *para a música sendo reproduzida no* iframe.

4.1.4 *Footer*

O *footer* é composto pelo componente *FooterStats*. Ele é usado para mostrar estatísticas da seleção atual do conteúdo no *main content*. Para a rota */artists*, o *footer* mostra a quantidade total de artistas presentes na biblioteca, o que pode ser visto na Figura 4.25. Para a rota */albums*, o *footer* mostra a quantidade total de álbuns presentes na biblioteca, exemplificado na Figura 4.26. Para a rota */genres*, o *footer* mostra a quantidade total de gêneros musicais presentes na biblioteca, visualizado na Figura 4.27. Para qualquer outra rota que não seja uma das três citadas, mostra a quantidade de músicas filtradas atualmente e a duração total delas em minutos, o que pode ser conferido na Figura 4.28.

4.1.5 *Sidebar* à esquerda

A *sidebar* da esquerda da aplicação é composto pelo componente *LeftSidebar*. Este, por sua vez, é composto por dois *components ListGroupSection*: o primeiro, mostrado na

Figura 4.25: *Quantidade total de artistas sendo mostrada no* footer*, na rota /*artists*.*

Figura 4.26: *Quantidade total de álbuns sendo mostrada no* footer*, na rota /*albums*.*

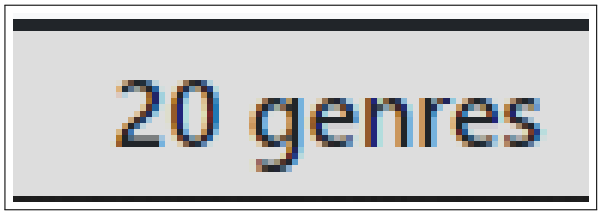

Figura 4.27: *Quantidade total de gêneros musicais sendo mostrada no* footer*, na rota /*genres*.*

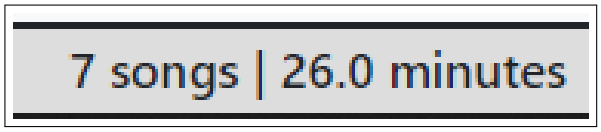

Figura 4.28: *Quantidade total de músicas da filtragem atual sendo mostrada no* footer*.*

Figura 4.29, lida com as opções da biblioteca (botões de backup e opções de filtragem: todas músicas; filtragem por artista, álbum ou gênero musical e filtragem das músicas recentemente adicionadas) e o segundo, exemplificado na Figura 4.30, lida com as playlists (botão de criar nova *playlist* e opções de filtragem para cada *playlist* presente na biblioteca). A rota atual é demarcada com a cor de fundo mais escura, dentre todos os botões de filtragem entre os dois *ListGroupSection*.

No segundo *ListGroupSection* (referente às ações relacionadas a playlists), há o botão para criação de novas playlists. Ao ser clicado, abre o componente *PlaylistCreationForm-Modal* (mostrado na Figura 4.31), que contém um *form* que recebe o título da *playlist* como *input*.

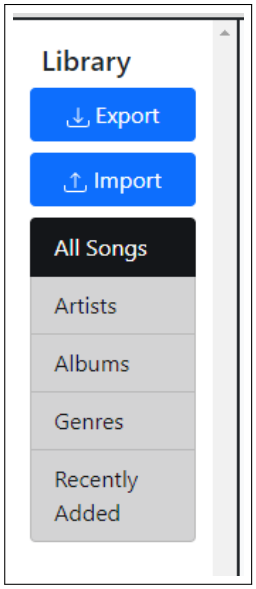

Figura 4.29: ListGroupSection *referente à biblioteca na* LeftSidebar.

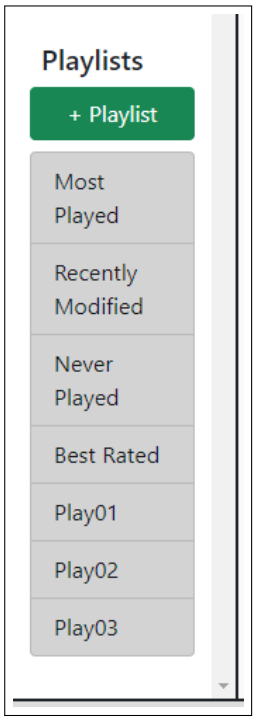

Figura 4.30: ListGroupSection *referente às playlists na* LeftSidebar.

4.1.6 *Sidebar* à direita

A *sidebar* da direita é composta pelo componente *SongInfoSidebar*, sendo responsável por fornecer maiores informações e mais ações sobre uma música específica, quando o usuário clicar no *badge button Show* da respectiva música no componente *SongTableRow*. Ao iniciar a aplicação, não há uma música selecionada ainda e, portanto, a *SongInfoSidebar* exibe um *Alert* (mostrado na Figura 4.32), informando ao usuário para clicar no *Show* de alguma música.

Figura 4.31: *Modal com o* form *de criação de nova* playlist.

Com uma música selecionada, o componente exibe dois *Cards*: o primeiro, exibido na Figura 4.33, lida com as ações possíveis para a música selecionada (adicionar a uma *playlist*, editar informações e deletar a música atual) e o segundo, mostrado na Figura 4.34, exibe as informações sobre a música (capa do álbum, título da música, nome do artista, duração, data de criação, data de modificação, reproduções e classificação).

Ao clicar no botão *+ Playlist* para adicionar a música selecionada a uma ou mais playlists, o componente *AddToPlaylistModal*, exibido na Figura 4.35, é aberto, composto por um modal contendo um *Card* com as informações da música e um Form composto de *checkboxes* (um para cada *playlist* atualmente na biblioteca).

Clicar no botão *Edit* abre o componente *SongInfoEditFormModal*, mostrado na Figura 4.36, que consiste em um modal com um *Form* para edição de informações da música atualmente selecionada, como: título da música, nome do artista e título do álbum. As informações já vêm pré-carregadas nos campos de *input* com as informações atuais da biblioteca.

Já clicar no botão *Delete* da música selecionada atualmente abre uma confirmação nativa do próprio *browser*, para evitar deleções acidentais, comportamento mostrado na Figura 4.37

4.2 Resultados do *Back-End*

A usabilidade do *Back-end* do *Spotunes* é bem mais simples que a do *Front-end*. O usuário final não precisa se preocupar em acessar o *Back-end*, pois isto é automatizado pelo *Front-End*. Porém, é possível fazer requisições à rota */api/spotify/search*, através do protocolo *GET*.

A rota em questão requer dois parâmetros: *name* (Nome do artista ou música a ser buscada na *API* do *Spotify*) e o parâmetro *type* (que pode assumir dois valores: *artist* ou *track*). O *back-end* retorna os dez resultados mais relevantes para a busca efetuada na *API* do *Spotify*.

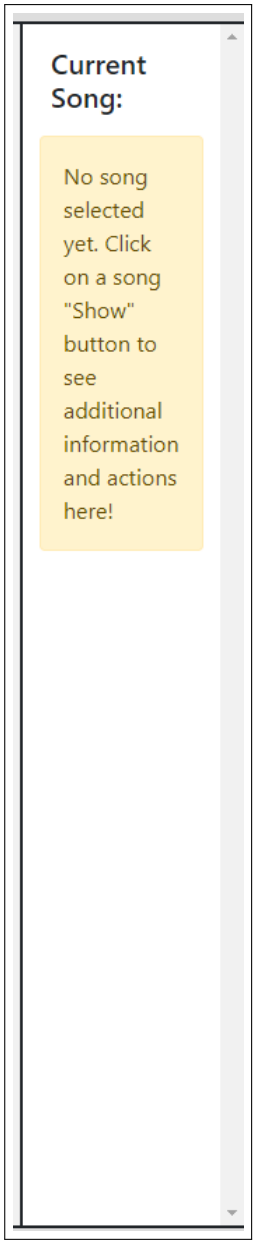

Figura 4.32: *Componente* SongInfoSidebar *quando não há uma música selecionada.*

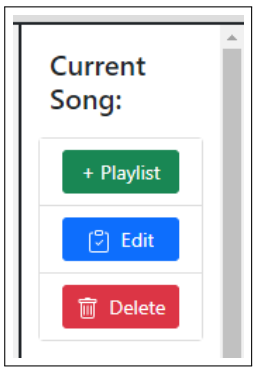

Figura 4.33: *Primeiro* card *do componente* SongInfoSidebar*, quando há uma música selecionada.*

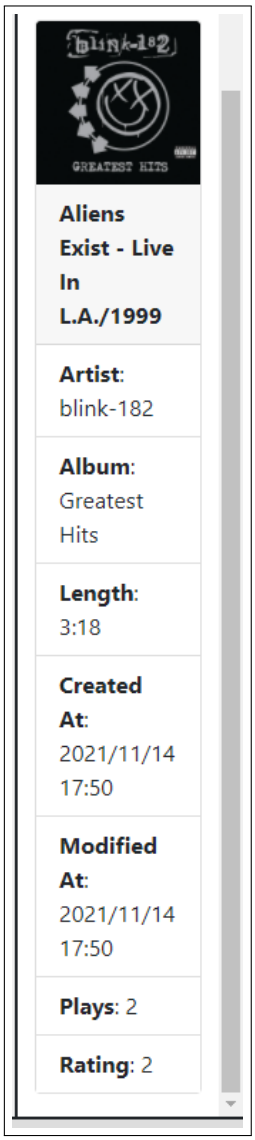

Figura 4.34: *Segundo* card *do componente* SongInfoSidebar*, quando há uma música selecionada*

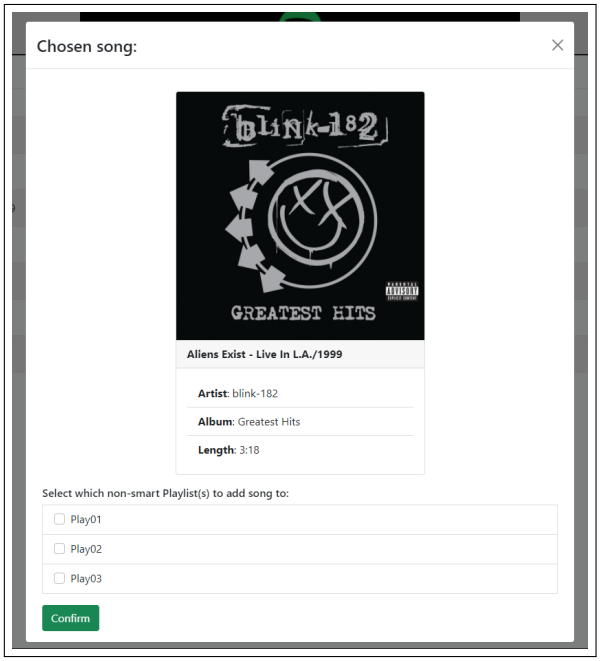

Figura 4.35: *Componente* AddToPlaylistModal*, exibido ao clicar no botão* + Playlist *do* SongInfoSidebar

Figura 4.36: *Componente* SongInfoEditFormModal*, exibido ao clicar no botão* Edit *do* SongInfoSidebar

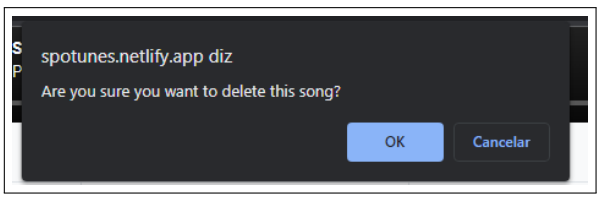

Figura 4.37: *Confirmação de deleção de música, exibido ao clicar no botão* Delete *do* SongInfoSidebar

https://spotunes-server.herokuap

spotunes-server.herokuapp.com/api/spotify/search?name=super+max&type=track


```
// 20211218155034
// https://spotunes-server.herokuapp.com/api/spotify/search?name=super+max&type=track

[
  {
    "spotifyID": "4gqOzJPGNCvEnS1u5YHOkp",
    "title": "Super Max!",
    "length_string": "3:45",
    "length_ms": 225360,
    "artist": "Pitstop Boys",
    "album": "Super Max!",
    "albumCover": "https://i.scdn.co/image/ab67616d0000b273563a30f88023b3cb5518756e",
    "albumID": "5giBzXuOs4UpWuKEV7bHXX",
    "artistID": "24UwzhsMtaOjBOL6483Jr6"
  },
  {
    "spotifyID": "6DlPa2rrVK3BygXJ48WYo3",
    "title": "RUNNING IN THE 90'S",
    "length_string": "4:44",
    "length_ms": 283833,
    "artist": "Max Coveri",
    "album": "SUPER EUROBEAT presents INITIAL D ∽D SELECTION∽",
    "albumCover": "https://i.scdn.co/image/ab67616d0000b273c27e997313531dbc7b01b247",
    "albumID": "1BbK1fKw0xg1HK3G0zVNTW",
    "artistID": "4aFf48VQYRT631OJGgVwUv"
  },
  {↔},
  {↔},
  {↔},
  {↔},
  {↔},
  {↔},
  {↔},
  {↔}
]
```


Figura 4.38: *Exemplo de busca de uma música* (track) *na* API *do* Spotify *por meio do* Spotunes Server

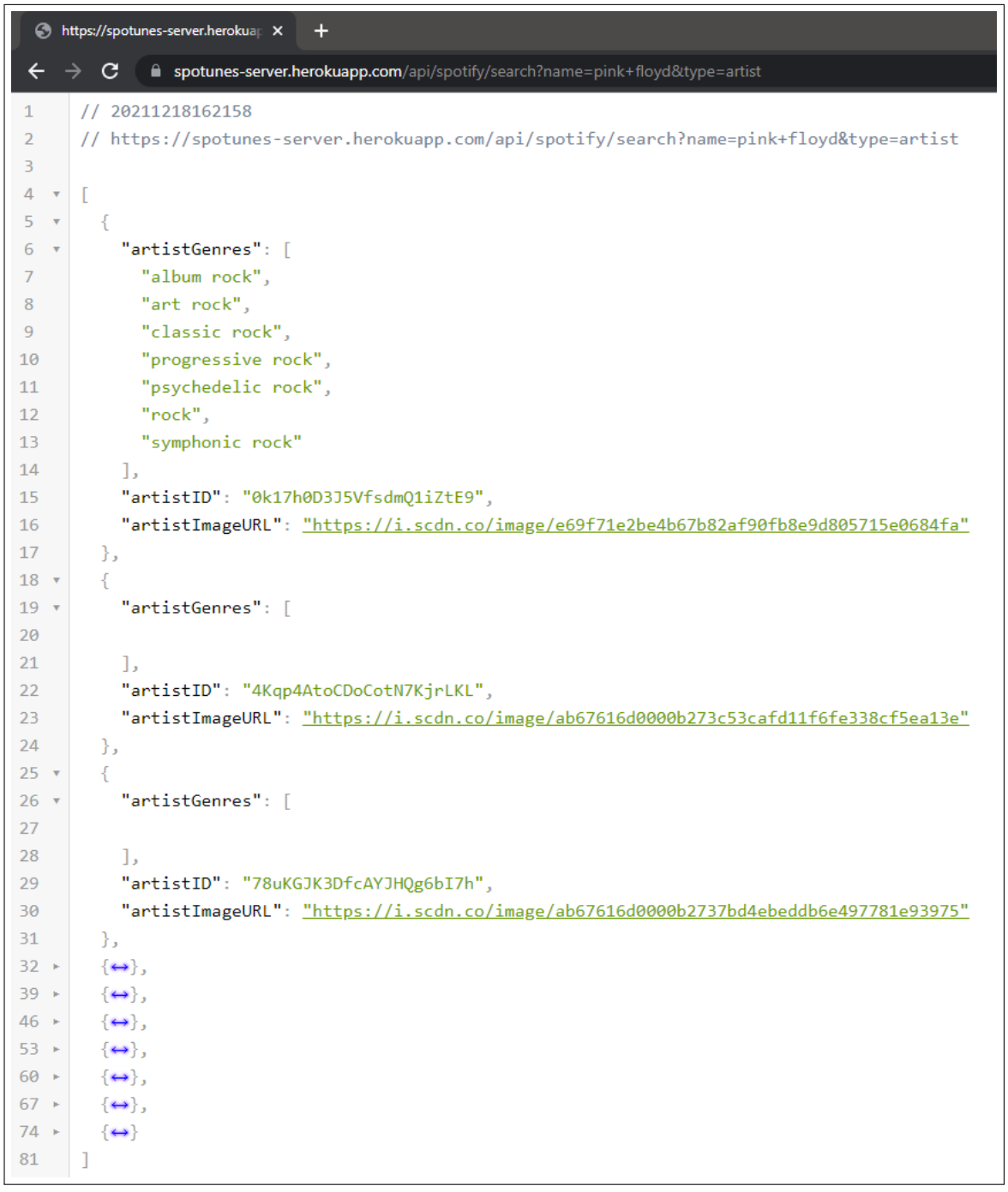

Figura 4.39: *Exemplo de busca de um artista* (artist) *na* API *do* Spotify *por meio do* Spotunes Server

Capítulo 5

Conclusão

O desenvolvimento do *Spotunes* foi uma longa jornada com muito aprendizado sobre organizar e estruturar uma aplicação mais robusta, cujo desenvolvimento levou vários meses. Foi interessante ver como os requisitos mudaram ao longo do desenvolvimento: algumas pré-concepções de ideias antes do desenvolvimento são alteradas conforme notamos melhorias a serem feitas por conta dos próprios teste de usabilidade. Junto com isso, houve várias mudanças de escopo sobre quais funcionalidades a aplicação abrangeria ou não.

Um exemplo mais claro disso é a ideia original de que o *Spotunes* seria uma plataforma centralizada, onde usuários acessariam suas respectivas bibliotecas por meio de um sistema de autenticação. A alteração para uma aplicação descentralizada, dependendo muito pouco de um *back-end* permitiu cumprir melhor a filosofia original da proposta: dar mais poder ao usuário sobre sua biblioteca digital.

Quanto a decisões técnicas que poderiam ter sido feitas de uma maneira diferente, vale ressaltar que, após a conclusão da etapa de desenvolvimento do *Spotunes* para este trabalho, ficou evidente que a utilização do combo *TypeScript + Angular* seria benéfico, mesmo o *Spotunes* não sendo uma aplicação de larga escala. Com a utilização destas tecnologias em vez de *JavaScript + React*, o código naturalmente teria ficado melhor organizado e estruturado, facilitando o desenvolvimento e certas decisões arquiteturais, pois no caso do *Spotunes* em especial, os dados salvos na biblioteca tem uma estrutura bem definida e imutável.

Segundo a máxima de que um software jamais fica pronto de fato, pois seu processo de desenvolvimento é um ciclo contínuo de melhorias, o estado atual do *Spotunes* é uma mera formalidade de um *subset* de requisitos e funcionalidades que possam ser considerados "completos"para o uso. Há margem para continuar o desenvolvimento como um projeto pessoal de longo prazo: procurando meios de integrar mais *APIs*, como *Youtube* e *Soundcloud*, aprimorar a interface visual e sistema de playlists inteligentes, oferecer recomendações de música do *Spotify* com base na biblioteca atual do usuário e, também, integrar mais tipos de mídias na biblioteca do usuário, como vídeos e *podcasts*, por exemplo.

Referências

[Apple 2021a] Apple. *Gerenciador de biblioteca virtual iTunes*. Acessado em 22/05/2021. 2021. URL: https://www.apple.com/br/itunes/ (citado na pg. 1).

[Apple 2021b] Apple. *Informação de uso de playlists inteligentes no iTunes*. Acessado em 22/05/2021. 2021. URL: https://support.apple.com/en-gb/guide/itunes/itns3001/windows (citado na pg. 2).

[Apple 2021c] Apple. *Tela principal do iTunes*. Acessado em 10/11/2021. 2021. URL: https://cdn.vox-cdn.com/thumbor/dFF2qTvqH_bLbOwiQ-Vwye5siiQ=/0x0:2036x1357/1200x800/filters:focal(0x0:2036x1357)/cdn.vox-cdn.com/uploads/chorus_image/image/49604071/itunes124.0.0.png (citado na pg. 7).

[Batistoti 2020] Vitória Batistoti. *Consumindo a API do Spotify: um breve passo a passo*. Acessado em 05/05/2021. 2020. URL: https://medium.com/reprogramabr/consumindo-a-api-do-spotify-um-breve-passo-a-passo-fd210312fdd (citado na pg. 16).

[Facebook 2021] Facebook. *Documentação do React*. Acessado em 15/07/2021. 2021. URL: https://pt-br.reactjs.org/docs/getting-started.html (citado na pg. 11).

[Google 2021a] Google. *Documentação do Angular*. Acessado em 15/07/2021. 2021. URL: https://angular.io/docs (citado na pg. 12).

[Google 2021b] Google. *Youtube Data API*. Acessado em 22/05/2021. 2021. URL: https://developers.google.com/youtube/v3/getting-started (citado nas pgs. 2, 9).

[Hu e Ogihara 2011] Yajie Hu e Mitsunori Ogihara. "Nextone player: a music recommendation system based on user behavior." Em: jan. de 2011, pgs. 103–108 (citado na pg. 10).

[Jannach *et al.* 2018] Dietmar Jannach, Iman Kamehkhosh e Geoffray Bonnin. "Music recommendations: algorithms, practical challenges and applications". Em: nov. de 2018, pgs. 481–518. ISBN: 978-981-327-534-8. DOI: 10.1142/9789813275355_0015 (citado na pg. 10).

[Kohli 2019] Kashish Kohli. *M.A.R.S. - Music Analysis and Recommendation System*. Acessado em 22/05/2021. 2019. URL: https://medium.com/sfu-cspmp/m-a-r-s-music-analysis-recommendation-system-f30424c2c362 (citado na pg. 10).

[Microsoft 2021] Microsoft. *Documentação do TypeScript*. Acessado em 15/07/2021. 2021. URL: https://www.typescriptlang.org/docs/ (citado na pg. 12).

[OpenJS 2021] OpenJS. *Documentação do Express*. Acessado em 15/07/2021. 2021. URL: http://expressjs.com/en/guide/error-handling.html (citado na pg. 11).

[Radionomy 2021] Radionomy. *Tela principal do Winamp*. Acessado em 10/11/2021. 2021. URL: https://i0.wp.com/www.extremetech.com/wp-content/uploads/2021/11/winamp-screen-640x412.jpg (citado na pg. 6).

[RestfulAPI 2021] RestfulAPI. *REST Architectural Constraints*. Acessado em 24/12/2021. 2021. URL: https://restfulapi.net/rest-architectural-constraints/ (citado na pg. 8).

[SoundCloud 2021a] SoundCloud. *Documentação API de busca do SoundCloud*. Acessado em 22/05/2021. 2021. URL: https://developers.soundcloud.com/docs/api/guide#search (citado na pg. 10).

[SoundCloud 2021b] SoundCloud. *Documentação do SoundCloud SDK*. Acessado em 22/05/2021. 2021. URL: https://developers.soundcloud.com/docs/api/sdks (citado na pg. 9).

[SoundCloud 2021c] SoundCloud. *SoundCloud for Developers*. Acessado em 22/05/2021. 2021. URL: https://developers.soundcloud.com/ (citado na pg. 2).

[SoundCloud 2021d] SoundCloud. *Termos de uso da API do SoundCloud*. Acessado em 22/05/2021. 2021. URL: https://developers.soundcloud.com/docs/api/terms-of-use#commercial (citado na pg. 9).

[Spotify 2021a] Spotify. *Documentação do Spotify SDK*. Acessado em 22/05/2021. 2021. URL: https://developer.spotify.com/documentation/web-playback-sdk (citado na pg. 9).

[Spotify 2021b] Spotify. *Documentação do Spotify Web API*. Acessado em 22/05/2021. 2021. URL: https://developer.spotify.com/documentation/web-api/ (citado nas pgs. 2, 8).

[Spotify 2021c] Spotify. *Documentação do Spotify Widget*. Acessado em 22/05/2021. 2021. URL: https://developer.spotify.com/documentation/widgets/generate/embed/ (citado na pg. 8).

[Spotify 2021d] Spotify. *Guia de Autorização do Spotify*. Acessado em 22/05/2021. 2021. URL: https://developer.spotify.com/documentation/general/guides/authorization-guide (citado na pg. 8).

[StackOverflow 2010] StackOverflow. *Detect Click into Iframe using JavaScript*. Acessado em 10/09/2021. 2010. URL: https://stackoverflow.com/questions/2381336/detect-click-into-iframe-using-javascript (citado na pg. 18).

[StackOverflow 2021] StackOverflow. *Stack Oveflow Survey 2021*. Acessado em 15/11/2021. 2021. URL: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe (citado na pg. 15).

[Yahoo 2021] Yahoo. *Tela principal do Musicmatch Jukebox*. Acessado em 10/11/2021. 2021. URL: https://img-21.ccm2.net/NQs5DRPC06r7YfVivWAxQuZpPx0=/500x/8e9c581258c5483c8e6fef16b638f0f3/ccm-download/17032-wmR34ZyRiAQo0JSH-s-.png (citado na pg. 7).