Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Ragic: Biblioteca de roteamento e navegação para desenvolvimento web em React

Lucas Vinicius Magalhães Pinheiro, Rafael Gonçalves de Paulo

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Genaina Nunes Rodrigues

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Ragic: Biblioteca de roteamento e navegação para desenvolvimento web em React

Lucas Vinicius Magalhães Pinheiro, Rafael Gonçalves de Paulo

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Genaina Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Dr. Daniel de Paula Porto
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 16 de dezembro de 2023

Dedicatória

Dedicamos esse trabalho às nossas mães, que sempre confiaram em nós e fizeram o possível e o impossível para que conseguíssemos chegar até aqui. Agradecemos também um ao outro, aos amigos que fizemos durante a graduação e antes dela, e à comunidade da CJR, empresa júnior de computação, onde aprendemos muito do que sabemos.

Agradecimentos

Primeiramente agradecemos às nossas famílias, pelo suporte que sempre nos deram e nos dão. Também a todos os amigos feitos ao longo da nossa jornada.

E agradecemos também à Prof.a Dr.a Genaina Nunes Rodrigues por ter confiado na nossa proposta, e nos acompanhado ao longo da elaboração deste trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

No contexto de desenvolvimento de aplicações web, ou *web apps*, uma importante funcionalidade é o roteamento e navegação. Para o roteamento, isso implica em definir uma árvore de rotas em modelo de diretório que multiplexem a *URL* do navegador para uma das páginas da aplicação, extraindo qualquer parâmetro, se houver. A navegação por outro lado envolve permitir que o usuário possa navegar internamente entre as páginas da aplicação, sem precisar manualmente editar a *URL*.

Esse trabalho propõe *Ragic*, uma biblioteca de roteamento e navegação para aplicações *web* em React.js implementada em *Typescript*. Além de conectar rotas com seus respectivos conteúdos, essa biblioteca almeja também promover uma experiência de programação menos suscetível a erros para desenvolvedores. Ela usa conceitos de tipagem disponíveis em Typescript para gerenciar de forma mais inteligente as rotas de um projeto, eliminando duplicação de informação e mantendo o princípio de *Don't Repeat Yourself (DRY)*, promovendo reuso e manutenibilidade.

Após o desenvolvimento de tal biblioteca, ela passou por uma série de testes e experimentos para evidenciar a sua funcionalidade básica, e seus diferenciais em relação às alternativas populares do ecossistema. Os testes consistiram em uma prova de conceito e um estudo de caso. Os testes foram bem sucedidos e apontam para uma utilidade real da ferramenta.

Palavras-chave: React, biblioteca, typescript, roteamento, web

Abstract

An important internal logic module in an web application (web app) is routing and navigation. Routing here means defining a tree if routes which can multiplex a browser URL to one of the application's pages, extracting the parameters, if any. As for navigation, it means allowing an user to navigate between an application pages without the need to manually change the URL.

This work proposes Ragic, a routing and navigation library for web apps in React.js, implemented in Typescript. This library has the goal of complementing other librarys currently on the market, with the expected behaviour of connecting routes with their respective contents. And in addition, it strives to promote a more comfortable developer experience. This library uses typing concepts in Typescript to create a smarter management of the routing process, removing code duplication and assuming a "Don't Repeat Yourself" approach, improving reuse and maintainability.

After the library's development, it underwent several tests and experiments to check if it can deliver its basic functionality, and also its add-ons compared to popular alternatives. The tests have been successful and point to its real-world usefulness.

Keywords: React, library, typescript, routing, web

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Ao se construir uma aplicação *web*, é necessário tratar do gerenciamento do conteúdo das páginas para direcionar qual conteúdo precisa ser renderizado em qual momento, e em qual página. Esse gerenciamento é feito pelo roteamento de páginas, que vincula uma *string* definida na *URL* (*Uniform Resource Locator*) da página (o nome da página a ser renderizada, como `"/menu"` ou `"/home"`) ao seu respectivo conteúdo.

Da forma como a maioria das opções atualmente disponíveis no ecossistema funcionam, o nome da rota exposto ao usuário é referenciado diretamente no código, como se fosse um *id* interno, de forma *hard-coded*. Ou seja, o nome é injetado diretamente no código, ao invés de ser gerado internamente por ele ou recebido de forma externa. Isso causa não só duplicação de código como também uma mistura entre lógica de implementação (referência para a página de uma rota) e detalhes de funcionalidade (nome ou apelido público da rota).

Em adição, como os nomes das rotas (que são tratados como `ids`) são definidos como `strings` , não é incomum um desenvolvedor introduzir um *bug* ao cometer um erro de digitação na lógica de navegação do *app*. Sendo `strings` , validação a nível sintático é impossível, e a única validação que os desenvolvedores podem usar são testes manuais e automáticos, e revisão manual.

Um exemplo desse tipo de *bug* introduzido via erro de digitação é uma situação hipotética como a seguinte:

Uma aplicação usa no seu *frontend* a *URI* (Uniform Resource Identifier) `/users/${id}` . No código, o programador tenta referenciar essa rota como `/user/${id}` , como na Figura 1.1. Esse é um erro humano comum, que quebra o contrato da lógica de roteamento da aplicação.

Tabela 1.1: Downloads de bibliotecas de roteamento (2023-12-05 até 2023-12-11).

Biblioteca	Downloads
RRD	9.827.680 [2]
Reach Router	856.219 [3]
Wouter	29.054 [4]
react navigation	1.231 [5]

```
<div>
  <span>{user_name}</span>
  <Link to={`/users/${id}`}>Acessar Perfil</Link>
</div>
```

Figura 1.1: Exemplificação de um possível erro de digitação.

1.1.1 Desalinhamento sintático e semântico do tipo de dados de rota

Com o contexto e a exemplificação apresentados para introduzir o tema, é possível perceber a causa da dificuldade, se analisado de um ponto de vista semântico. Tudo surgiu porque `/users/${id}` foi usado para a *prop* `"to"` quando, segundo a lógica de negócios, esse não era um valor válido para ela, uma vez que não mapeava para uma rota existente no *app*.

Como apresentado na Tabela 1.1, a biblioteca mais usada no mercado de desenvolvimento web em *React* atualmente é o *react-router-dom*, ou *RRD*. Segundo a definição do tipo de `Link` [6] nessa biblioteca, a propriedade `"to"` recebe um valor do tipo *string*. E como *string* se refere a qualquer sequência de caracteres, toda e qualquer *string* é válida nesse estágio estático da validação. Qualquer outra validação sobre o uso correto das rotas e seus nomes no *app* é feita por outras etapas (*linting*, testes unitários, *review* manual do código em *PR* (*pull request*), lógica de validação e potencialmente correção em *runtime*, dentre outras).

Cada uma dessas alternativas não só é menos comum do que o uso de tipagem estática, mas também tem menos chance de capturar os erros, ou requer consideravelmente maior trabalho do que se a definição do tipo do valor `"to"` no exemplo não tivesse essa dissonância com a lógica de negócios pretendida.

A intenção quando o componente `Link` é definido é que ele apenas receba valores que encaixem no padrão de rotas definidas do *app*. Se por um momento ignorarmos a interpolação do dado dinâmico `id` , que representa o identificador do usuário no banco de

dados, as rotas válidas são um conjunto finito e enumerável de valores. Esse tipo de valor que esperamos para a propriedade `"to"` é mais bem coberto por uma União de *string literals*, não como *string*.

1.1.2 *Union*

Union é uma ferramenta de *TypeScript* (ou *TS*) que permite atribuir diferentes tipos para um mesmo parâmetro ou variável. Como o nome sugere, é uma união de tipos, como na Figura 1.2.

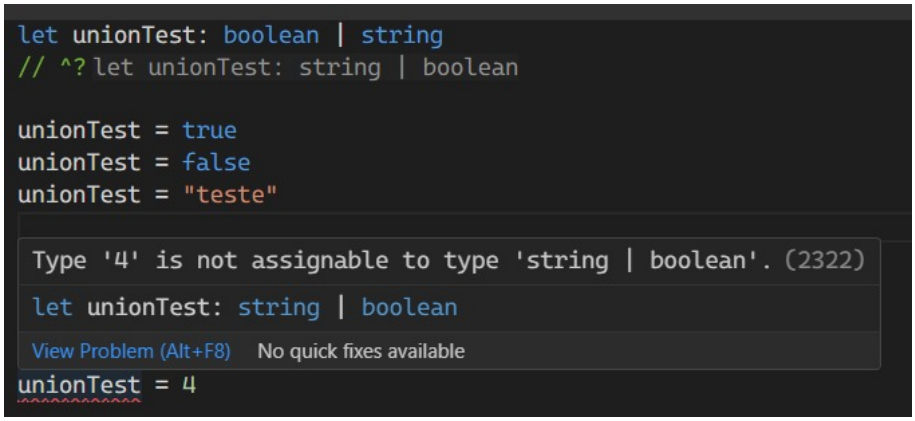

Figura 1.2: Exemplo simples de uma *Union* em *TS*.

No trecho de código 1.2, a variável `unionTest` irá aceitar tanto `boolean` quanto `string` como argumentos válidos. E como demonstrado abaixo da definição da variável, quando é atribuído para ela os valores `true` , `false` e `"teste"` , os valores foram aceitos. Mas quando se tenta atribuir o valor `4` , o código acusa um erro, já que a *union* da variável não aceita valores numéricos.

A usabilidade principal desse tipo de estrutura no desenvolvimento da biblioteca é o uso de uma estratégia chamada *narrowing* (estreitamento), que reduz as possibilidades que um tipo pode representar.

No caso de *ragic*, o estreitamento que queremos é que o valor `"to"` pare de aceitar toda e qualquer `string` , e passe a aceitar somente um conjunto reduzido de *strings*. *Strings* estas sendo as que existirem como uma rota previamente declarada.

Como demonstrado no fragmento de código da Figura 1.3, ao se declarar uma `union` com somente algumas strings seletas, a variável só aceitará estas tais `strings` em vez de aceitar qualquer `string` possível. Também é possível checar nesse fragmento o *autocomplete* do *TS* em ação, uma funcionalidade útil para o projeto e que é melhor detalhada na seção a seguir.

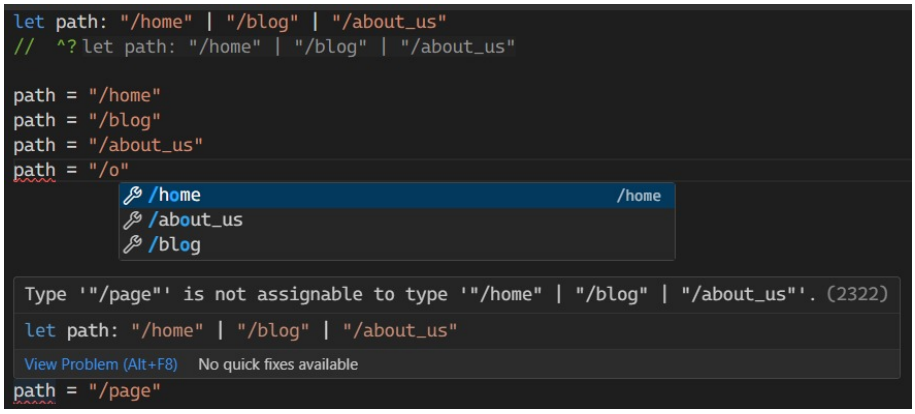

Figura 1.3: Exemplo do uso de *narrowing*.

Dessa forma, ao tentar passar para o `"to"` alguma *string* que não encaixe com nenhuma rota definida, haverá um erro de tipagem, permitindo ao desenvolvedor perceber mais rapidamente o problema.

1.1.3 Objetivos

De acordo com a situação anedótica apresentada na Seção 1.1, o principal objetivo do desenvolvimento da biblioteca *ragic* é criar uma ferramenta que, ao mesmo tempo que consegue disponibilizar todo o aparato já existente em outras ferramentas do mercado, traz junto novas camadas que reduzem os tipos de erros que podem aparecer enquanto o desenvolvedor coda especificamente a parte controladora de rotas de sua aplicação.

Dessa forma, propõe-se então que o objetivo primário da biblioteca é, acima de tudo, melhorar a *DX*, ou *Developer Experience*, a experiência do desenvolvedor. Oferecendo à pessoa que estiver programando, uma experiência mais cômoda e agradável.

E claro, como objetivos secundários, esse conforto não deve vir em troca de um custo ou penalidade a mais para o cliente desenvolvedor. Logo, a biblioteca não deve ser mais difícil de usar ou aprender, nem deve funcionar de maneira mais lenta comparado com as outras bibliotecas do ambiente. Principalmente comparando com o *RRD* que é, como mencionado anteriormente, a principal biblioteca para essa tarefa atualmente.

1.2 Estrutura do documento

O Capítulo 2 apresenta uma contextualização geral do ecossistema de desenvolvimento de aplicações *web* em *React*, explicando como a tarefa de roteamento é comumente feita atualmente. O Capítulo 3 apresenta a ferramenta proposta. O Capítulo 4 relata os diversos testes realizados pela equipe de desenvolvimento sobre a biblioteca e seus resultados.

O Capítulo 5 discorre brevemente sobre os planos futuros para a biblioteca, comentando sobre idealizações que não foram possíveis de serem concluídas no período desse trabalho.

Capítulo 2

Fundamentação teórica

Nesse capítulo serão explorados os conceitos e ferramentas centrais do domínio da ferramenta proposta, e *frameworks* usados no desenvolvimento da ferramenta.

2.1 *JavaScript* e *TypeScript*

Para este trabalho, foi feita a escolha de *TypeScript* como linguagem. Essa escolha se dá porque as possibilidades adicionais que o programador tem usando *TypeScript* são essenciais para o funcionamento da ferramenta proposta.

2.1.1 *JavaScript*

JavaScript [7] é uma linguagem de programação interpretada, com suporte para paradigmas de programação imperativo, orientada a objetos e funcional. A linguagem é dinâmicamente tipada [8], com variáveis tendo um tipo associado à elas em *runtime*.

Graças à especificação padrão *ECMA-262* [9], *JavaScript* é usado por navegadores como a "linguagem da *web*". Dado que o objeto do trabalho é desenvolvimento de aplicações *web*, é necessário que o resultado final seja em *JavaScript*.

2.1.2 *TypeScript*

TypeScript [10] é uma linguagem de programação construída em cima de *JavaScript*. Sendo um *superset* de *JavaScript*, *TypeScript* busca melhorar a resiliência, velocidade de desenvolvimento e confiabilidade do código, adicionando a possibilidade de tipagem estática ao código *JavaScript*.

Aplicações *web* construídas em *TypeScript* tem em sua *toolchain* um compilador que transforma o código fonte *TypeScript* em *JavaScript*. Esse resultado é o código da apli-

cação final que é enviado para o computador do usuário final por *HTTP* e interpretado pela *engine* de *JavaScript* do seu navegador.

2.2 *JSX*

JSX [11], ou *JavaScript Syntax Extension*, foi criada pela *Facebook* especificamente para ser usada em *React*. *JSX* é uma extensão da sintaxe de *JavaScript* como definida em *ECMAScript*.

Seu objetivo é definir uma sintaxe concisa e familiar para definir estruturas de árvore com atributos. Seu uso mais comum, como exemplificado na sua documentação e no seu uso em *React*, é representar uma interface de usuário visual.

2.2.1 *TSX*

TSX [12] é a solução de *TypeScript* para inclusão de *JSX* na sua especificação. Por meio de uma mudança da extensão do arquivo (de `.ts` para `.tsx`) e da configuração do compilador, o programador *TypeScript* pode usar sintaxe *JSX* no seu código. Isso viabiliza o uso de bibliotecas como *React*, que utilizam a sintaxe *JSX*.

2.2.2 *React*

React [13] é uma biblioteca da linguagem *JavaScript* para construção de interfaces de usuário criada pela *Facebook*. *React* usa a sintaxe *JSX*, e pode ser usada para a construção de aplicações *web* com a biblioteca `react-dom` , aplicações *mobile* com `react-native` , e aplicações de linha de comando com `ink` [14], entre outras. *React* em si só define a lógica interna de composição, estado, construção e remoção dos componentes definidos, e o meio final usado para a aplicação é definido pela biblioteca de renderização usada, como as mencionadas anteriormente.

Componentes *React* podem ser definidos com o uso de classes ou funções. O uso de Componentes de Classe é defasado e não-recomendado pela comunidade, em favor de Componentes Funcionais e o estilo de programação declarativa que eles promovem.

2.2.2.1 *React Hooks*

Introduzidos na versão v16.8.0 de *React*, *React Hooks* [15] são uma ferramenta que permite a adição de estado e efeitos colaterais à Componentes Funcionais, originalmente conhecidos como Componentes *Stateless*. Por convenção, *Hooks* são funções com nome prefixado de `use` , que retornam os dados ou *callbacks* necessários para a lógica de gerência de estado ou de efeitos colaterais desejados.

2.3 Roteamento de páginas *web*

Tradicionalmente, um *website* é conceitualizado como um conjunto de páginas *HTML* relacionadas, acessíveis por *HTTP*. O servidor sabe qual página servir ao cliente a partir da rota (ou *path*) da requisição *HTTP* [16], e retorna o documento apropriado como resposta.

2.3.1 Roteamento *frontend* e *backend*

Frameworks de desenvolvimento *web fullstack*, como *Ruby on Rails*, tendem a usar o método tradicional de roteamento, nomeado também como roteamento por *backend*. Outra alternativa, usada por aplicações do estilo *SPA* [17], é roteamento *frontend*.

Aplicações que usam roteamento *frontend*, em vez de fazer uma requisição por um documento HTML inteiramente novo, possuem código *JavaScript* que dinamicamente altera o documento *HTML* atual para representar a nova página, requisitando dados necessários do servidor para populá-los no documento editado.

Existem prós e contras entre os dois paradigmas de roteamento, e também soluções híbridas como *server rendering*, mas por padrão aplicações *React* usam o paradigma de roteamento *frontend*. A biblioteca proposta, assim como as alternativas principais do ecossistema, é uma biblioteca de roteamento *frontend*.

2.4 *react-router-dom*

`react-router-dom` é parte do *toolset React Router* [2], que permite roteamento e navegação de aplicações *web*. `react-router-dom` é a ferramenta mais popular para roteamento no ecossistema, e o método mais comum de usá-la é com definição declarativa das rotas da aplicação no formato de uma árvore de componentes `Route` . A navegação em aplicações que usam `react-router-dom` usa o componente `Link` , que uma vez compilado e renderizado resulta em um elemento *HTML* âncora.

`react-router-dom` permite navegação *frontend* pelo seu uso da *JavaScript History API* [18] disponibilizada pelo navegador. Por meio do objeto global `history` , `react-router-dom` pode interceptar alterações da *URL* e do histórico, e lidar com essas mudanças localmente em vez de fazer uma requisição para o *backend* com a nova *URL*.

2.5 *LSP, Language Server Protocol*

Originalmente um protocolo fechado desenvolvido por *Microsoft* para *Visual Studio Code*, *LSP* [19] é um protocolo para comunicação com *IDEs* e editores de código sobre informações de tipo do código fonte sendo editado. Essa informação é usada por *IDEs* para *highlight* de sintaxe, validação de sintaxe, *auto-complete*, entre outros usos.

O uso de *LSP* permite uma maior portabilidade entre linguagens e editores de funcionalidades como as mencionadas anteriormente, simplificando e generalizando a integração entre compiladores e *IDEs*.

Capítulo 3

Proposta

Nesse capítulo é apresentada a proposta de solução para o problema introduzido no Capítulo 1, explorando decisões arquiteturais e de implementação.

3.1 Organização do repositório

Com base na idealização de como deveria ser feito a biblioteca e também no planejamento das fases de seu desenvolvimento, foi decidido que *ragic* seria construída em um *monorepo NX*, para que se pudesse fazer em um só repositório o desenvolvimento da biblioteca e os aplicativos de teste em *react-router-dom* e em *ragic*.

Um *monorepo* é um repositório *git* que comporta mais de um projeto dentro de si. Apesar de normalmente esses projetos serem interligados entre si, isso não é um requisito para a construção de um *monorepo*.

Já *NX* [20] é um sistema *open source* de construção de repositórios que tem como característica a facilidade de integrar diferentes projetos entre si. Pela necessidade de criar uma nova biblioteca e testar ela em um ambiente de desenvolvimento antes de publicá-la para o público, ter os dois aplicativos (a biblioteca e o *app* de teste) no mesmo repositório permitiu que a comunicação entre os dois fosse feita de forma mais eficiente.

Inicialmente, seguindo tutorial de construção de ambiente *NX* disponibilizado na documentação da ferramenta, tentou-se criar uma aplicação via comando `npx create-nx-workspace` e usando como *framework* para construção de aplicação o *Next.js*, um dos *frameworks* disponíveis no mercado. Por conta de conflitos de versões do *yarn* e *node*, e conflitos estruturais do repositório, essa primeira organização teve que ser repensada.

Num segundo momento, buscou-se outras alternativas para a construção. Foi decidido que em vez de se criar um *monorepo NX* logo de início, era uma melhor estratégia criar uma aplicação *React* e só em seguida migrá-la para um ambiente *NX*.

Em vez de usar novamente o *Next.js*, imaginando que poderiam ocorrer conflitos novamente, optou-se por usar outra forma de criação de aplicação, uma que permitisse fazer a configuração posteriormente, dando possibilidade de checar e acomodar possíveis conflitos que surgissem. Decidiu-se então o uso de *create react app* [21] via comando `npx create-react-app app-name` . E após isso foi feita a migração para *NX* via comando `npx nx@latest init` , segundo a documentação do *NX*.

Após a criação do repositório, de início, foram criadas algumas *issues* no repositório github do projeto:

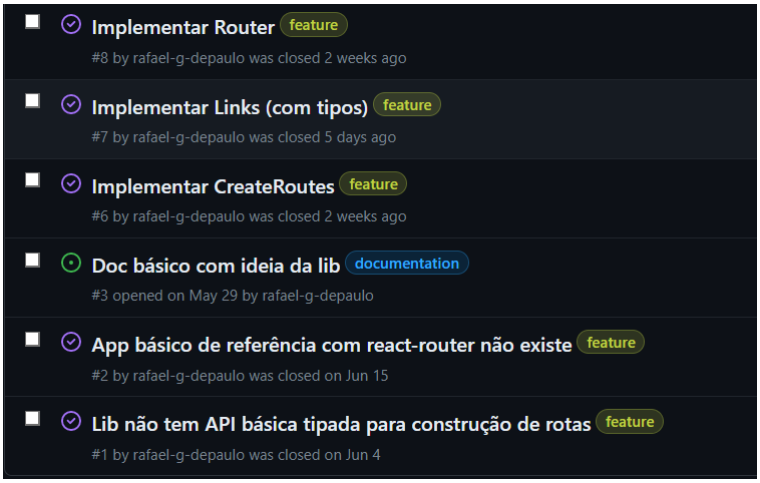

Figura 3.1: Primeiras *issues* criadas.

Os primeiros passos então podem ser resumidos em:

1. Construção de um *app* simples usando *RRD*: Como o objetivo é avaliar a facilidade em se construir um *app* com *ragic* em comparação com as bibliotecas atuais do ecossistema, é necessário construir um *app* básico no ecossistema atual para depois ser comparado com um *app* feito com *ragic*.

2. Implementar o componente de rotas: Para garantir a internalização da biblioteca, é necessário criar o componente que transforma um dicionário de rotas em um componente encapsulando as respectivas rotas.

3. Implementar *links*: Sendo o principal meio do usuário final navegar para outras rotas, a implementação de componentes de `Link` é um dos pilares necessários para o funcionamento da biblioteca.

4. Construção da documentação: Junto com o desenvolvimento do código de *ragic*, é necessário também manter uma documentação da biblioteca para que novos (futuros) usuários consigam a usar de forma apropriada.

5. Publicação da biblioteca em *NPM*: E para que futuros usuários possam utilizar a biblioteca, ela precisa estar disponível para download via *NPM*.

Durante o desenvolvimento e construção inicial da biblioteca, O seu conceito foi refinado e foi possível perceber quais eram realmente seus diferenciais entre as alternativas no ecossistema, que limitações essas características adicionavam na implementação, e como isso orientaria as decisões de alto nível de organização do projeto e *API* final.

3.2 Visão geral

O uso da biblioteca pelo usuário programador, e os módulos internos necessários para o funcionamento dela estão descritos nesta seção.

3.2.1 Roteamento

Para a definição da árvore de roteamento, é usado o *builder* `createRoutes` . `createRoutes` recebe por parâmetros informação de implementação e de tipagem de `Options` e `Pathname` , e de contexto da fábrica recebe a informação da `RouteTree` já construída. O *builder* retorna então um objeto do tipo `Routes` , construído dinamicamento pelo próprio *builder*. A figura 3.2 mostra o fluxo de execução e dependência do *builder*.

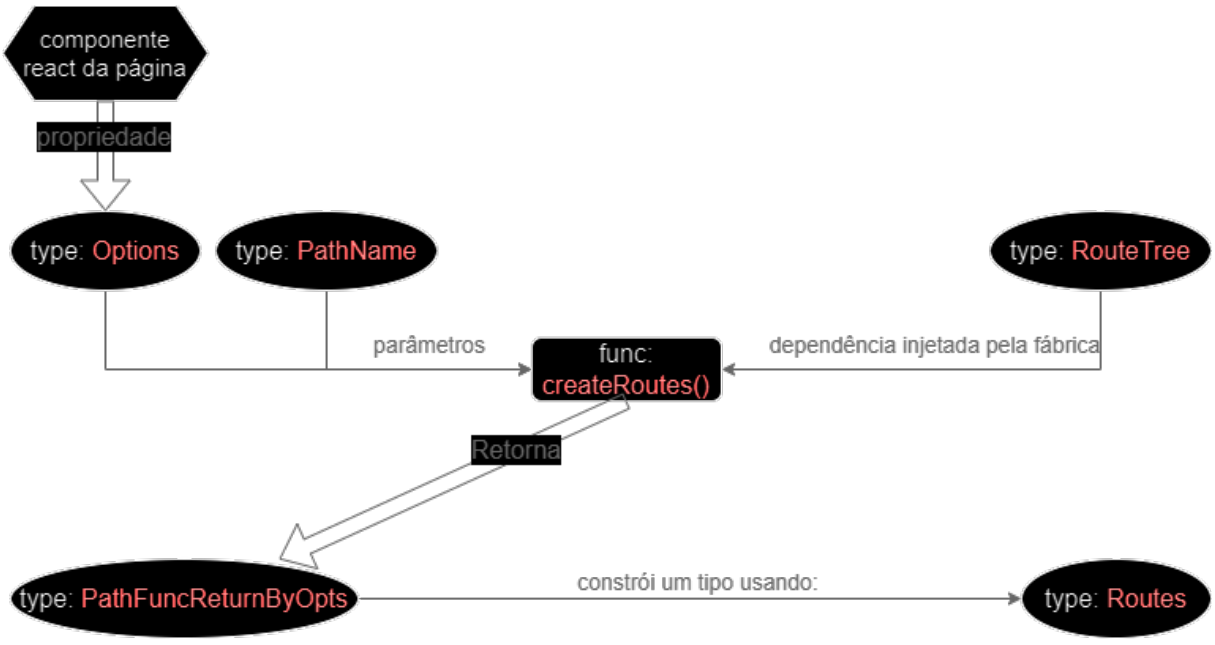

Figura 3.2: Proposta do fluxo para definição da árvore de roteamento.

Para uso dos parâmetros da rota atual dentro dos componentes da aplicação, é usado o *hook* `useRouteParams` . `useRouteParams` recebe da fábrica a dependência injetada `Routes` , que representa a árvore de roteamento válida da aplicação. Usando o parâmetro de tipo `IndexedRoute` , é calculado o tipo de retorno `RouteIndexParams` , que valida os índices existentes na rota. A figura 3.3 mostra o fluxo de execução e dependências do *hook useRouteParams*.

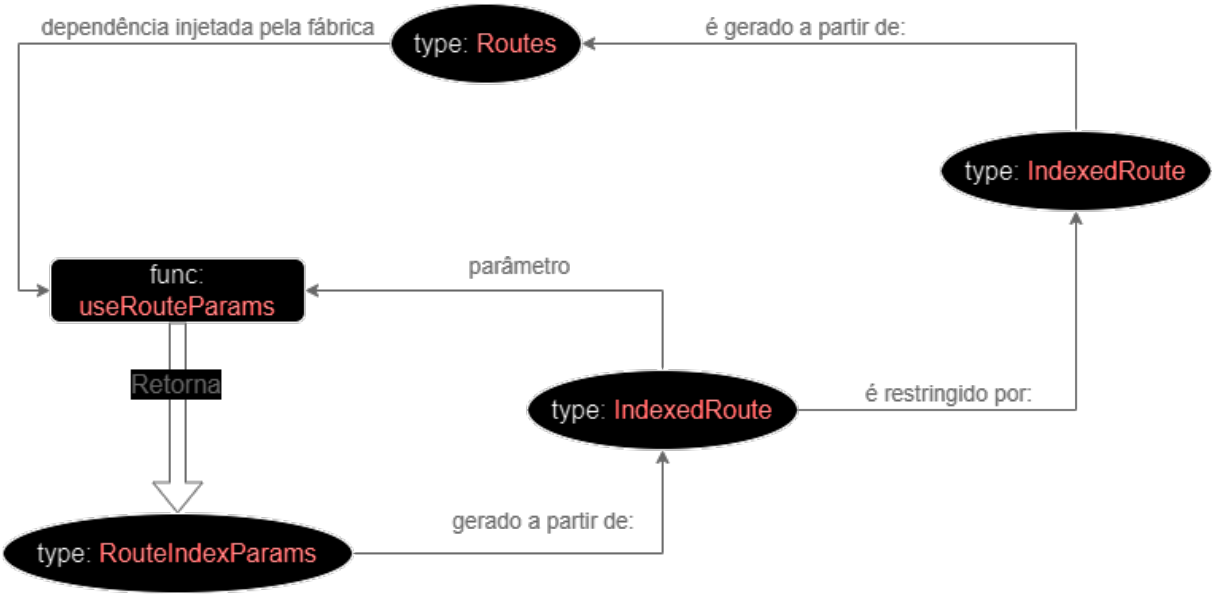

Figura 3.3: Proposta do fluxo para extração da informação dos índices da rota atual.

3.2.2 Navegação

Para navegação na aplicação é usado o componente *React* `Link` . `Link` recebe como dependência injetada pela fábrica o tipo `Routes` . Como parâmetros, `Link` recebe a propriedade *"to"*, que é de um tipo *string literal* restringido por `ConcretePaths<Routes>` , e a propriedade *"params"*, gerado por `RouteIndexParams` a partir do tipo da propriedade *"to"*. A figura 3.4 mostra o fluxo de execução e dependências do componente *Link*.

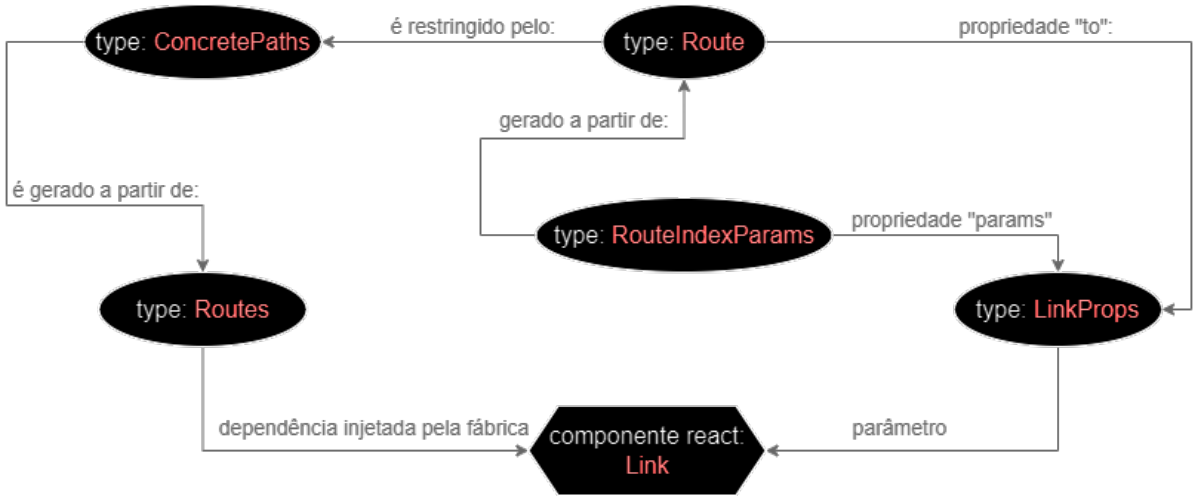

Figura 3.4: Proposta do fluxo para navegação na aplicação.

3.3 Desenvolvimento

Nessa seção são explorados conceitos internos da biblioteca, detalhes de implementação seu conceito central e quais ferramentas de engenharia de software foram necessárias para sua construção.

3.3.1 Tipos de segmentos rotas

Dentro de *ragic*, segmentos de rotas pertencem a uma de 4 categorias. Essas categorias são usadas internamente para definir como a *API* das funções e componentes da biblioteca devem interagir com rotas específicas, a partir das categorias dos seus segmentos.

3.3.1.1 Concreto simples

O tipo mais comum de rota, é uma rota que está ligada a um componente qualquer. Essa rota é invariável e sempre que ela for chamada, ela mostrará a mesma informação. Alguns exemplos de rotas desse tipo podem ser `/home` , `/contato` , `/sobre` .

3.3.1.2 Vazio

Esse tipo de rota não se conecta com nenhum componente. Dessa forma, é uma rota sem conteúdo com o único intuito de habilitar outras rotas subsequentes de existirem. Como por exemplo, em uma rota `/pagina/3` . Se alguém tentar acessar somente a rota `/pagina` , ocorrerá um erro pois essa rota não contém nenhum conteúdo. E para definir o tipo

de rota em que o conteúdo é variável dependendo do valor passado na URL, nós temos a rota índice.

3.3.1.3 Índice

As rotas índice tem a característica de ter o seu conteúdo disponibilizado ser dependente de um valor passado na *URL*. No caso mais simples, como no exemplo acima de `pagina/03` , o componente ligado à esse tipo de rota pode capturar a informação necessária (a *string* `"03"`) e utilizar ela para determinar dentro do componente qual informação deve ser apresentada.

Existem rotas concretas que também alteram seu conteúdo com base em outras informações, como por exemplo uma rota `"/editar-perfil"` para um usuário alterar alguma informação do seu perfil pessoal em um site qualquer. Dependendo de qual usuário está usando o site, o conteúdo da página será diferente.

A diferença desse tipo de rota concreta para uma rota índice é que o conteúdo dessa rota concreta é alterado com base em alguma informação interna do código, enquanto uma rota índice tem seu conteúdo alterado por alguma informação recebida do navegador, contida na *URL* da página visitada.

3.3.1.4 Apelido

Parte da ideia inicial de*ragic* era a possibilidade de, ao criar uma rota, poder atribuir para ela além de um componente, alguns "apelidos". Como exemplo, uma rota principal `"/home"` seja ligada ao componente *React* `HomePage` , e as rotas `"/"` e `"/main"` também sejam ligadas ao mesmo componente.

Isso é uma funcionalidade adicional, e não necessariamente necessária. Apesar de que é possível só criar uma nova rota e encaminhar ela para uma rota pré existente, a intenção era de que na hora de declarar a primeira rota fosse também possível definir dentro do contexto dela os respectivos nomes-apelido.

3.3.2 Tipo *TS* correspondente de uma árvore de roteamento

Para fazer a validação do uso das rotas, a ferramenta proposta precisa de uma definição interna da estrutura da árvore de roteamento. O objetivo é com essa inferência de tipos conseguir extrair a informação necessária sobre a estrutura da árvore de roteamento e o nome dos segmentos de rota, em algo similar com o exemplo do Bloco de Código 1.

```
type Routes = [
  "/",
  [
    ["/home", []],
    ["/blog", [
      ["/:blog_id", []],
    ]],
    ["/about", [
      ["/us", []]
    ]],
  ]
]
```

Bloco de Código 1: Ideação inicial de uma forma de codificar as informações de uma árvore de roteamento em um tipo *TypeScript*.

Assim é definida uma árvore de roteamento como um nó de rota, e um nó de rota é uma tupla com dois elementos: o nome do segmento de rota como uma string literal e uma Tupla contendo todos os nós filhos do nó atual. O exemplo acima, então, descreve as seguintes rotas:

- `"/"`
- `"/home"`
- `"/blog"`
- `"/blog/:blog_id"`
- `"/about"`
- `"/about/us"`

Com essa ideia básica, um exemplo simples como o da Figura 3.5 pode ser construído.

Figura 3.5: Teste para extração e compilação da união desejada de *string literals* a partir de um tipo *Routes*.

Aplicando esse conceito básico e incluindo a informação sobre a categoria dos segmentos de rota, a tipagem de uma árvore de roteamento usa os tipos utilitários `Routes` e `Segment` , resultando em um tipo como o do Bloco de Código 2. O Bloco de Código 2 representa a tipagem equivalente de uma aplicação com as rotas concretas `"/"` , `"/home"` , `"/blog"` , `"/blog/:blog_id"` e `"/about/us"` .

```
1  Routes<[
2    Segment<"/", "concrete", never>,
3    Segment<"/blog", "concrete", [
4        Segment<"/:blog_id", "concrete", never>,
5    ]>,
6    Segment<"/news", "link", never>,
7    Segment<"/about", "concrete", never>,
8    Segment<"/about", "empty", [
9        Segment<"/us", "concrete", never>
10   ]>
11 ]>
```

Bloco de Código 2: Tipagem de uma árvore de roteamento exemplo.

3.3.3 Definição da árvore de roteamento

Para definição da árvore de roteamento, é usada a função `createRoutes` . `createRoutes` é uma função que usa o *design pattern Builder*[22].

O tipo resultante da árvore é inferido a partir da inferência de tipos para funções parametrizadas em *TypeScript*, construíndo a árvore modularmente. Para o funcionamento desse *Builder*, a implementação dele foi feita com uma fábrica. Pela fábrica, a função recebe como dependência injetada um "contexto", que é a árvore pai da sub-árvore sendo construída. Análise do funcionamento da fábrica se encontra na Seção 3.3.6.

Foram definidos os seguintes objetivos para a tipagem inicial do *builder*:

- Uma chamada para uma função `createRoutes` criará o objeto nulo inicial do *builder*.
- Método `path` , que recebe o nome de um segmento de rota, e um objeto de opções.
- O nome do segmento de rota recebido por `path` pode ser um segmento simples ou de índice.
- O objeto de opções tem o componente *React* da página a ser associada com aquela rota, se for concreta.
- Se o segmento tiver nós filhos na árvore de roteamento, o objeto de opções terá os filhos representados por uma chamada do próprio *builder*, associados com uma propriedade `children` .
- Para facilidade de escrita e leitura, existirá uma função apelido `path` , que quando chamada será equivalente a chamar `createRoutes().path` .

O intuito final desses objetivos é chegar numa tipagem final que permita um uso como o da Figura 3.6. O fluxo da implementação final é o representado pela Figura 3.9.

```
export const routes = createRoutes()
    .path("/", { component: () => <PageHome /> })
    .path("/blog", {
        component: () => <PageBlogList />,
        children: path("/:blog_id", { component: () => <PageBlogPost /> }),
    })
    .path("/news", { link_to: "/blog" })
    .path("/about", {
        children: path("/us", { component: () => <PageAboutUs /> })
    })
```

Figura 3.6: Idealização do uso da *API* de construção da árvore de roteamento.

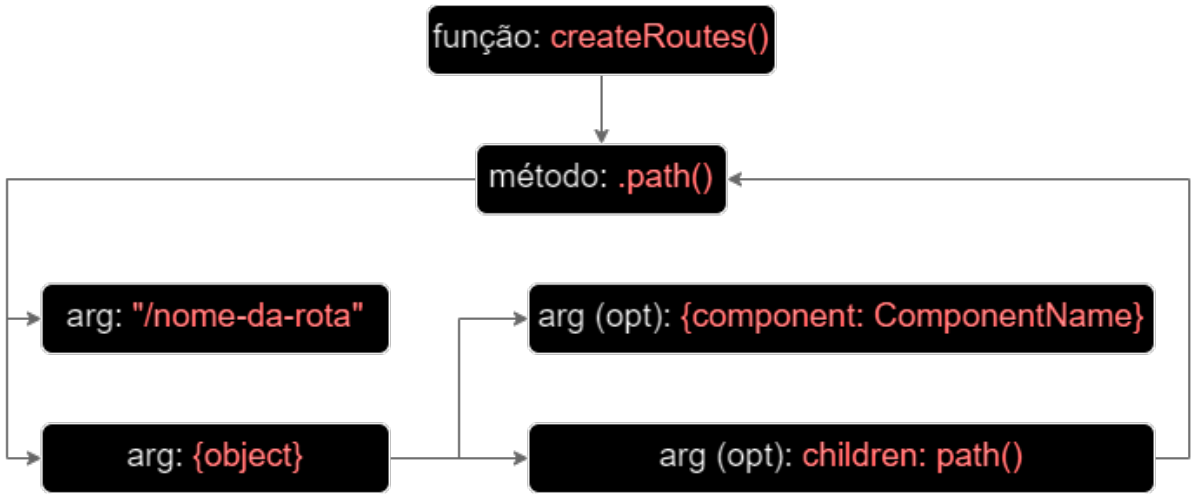

Figura 3.7: Diagrama representando a lógica de uso do `createRoutes` .

Para a construção tanto de `path` quanto `createRoutes().path` , foi criada a fábrica `makePathFunc` , visível na Figura 3.8.

```ts
const makePathFunc =
  <RouteTree extends unknown[] = []>(context: RouteTree) =>
    <PathName extends EnsureLiteral<PathName>, Options extends SegmentKindOpts>(
      path: PathName,
      opts: Options = {} as Options
    ): PathFuncReturnByOpts<Options, RouteTree, PathName> => {
      const concreteChildRoutes =
        'children' in opts
          ? Object.fromEntries(
            Object
              // get all concrete routes from children
              .entries<Component>(opts.children as Record<string, Component>)
              // only routes, not methods and stuff
              .filter(([key]) => key[0] === '/')
              // append the current path as it's parent
              .map(([key, value]) => [`${path}${key}`, value])
          )
          : {};

      // if current segment is a concrete route, add it
      const concreteCurrentRoute =
        'component' in opts ? { [path]: opts.component } : {};

      // compile new concrete routes based on context, children and current
      const concreteRoutes = {
        ...context,
        ...concreteCurrentRoute,
        ...concreteChildRoutes,
      } as Record<
        ConcretePaths<NewRouteTree<RouteTree, PathName, Options>>,
        Component
      >;

      const pathFunc = makePathFunc(concreteRoutes as RouteTree);

      const newRoutes = {
        [type_brand_key]: {} as NewRouteTree<RouteTree, PathName, Options>,
        path: pathFunc,
        ...concreteRoutes,
        // explicit conversion below needed to ensure proper typing
      } as unknown as PathFuncReturnByOpts<Options, RouteTree, PathName>;

      return newRoutes;
    };
```

Figura 3.8: Implementação da função fábrica usada para definir `createRoutes` .

Internamente, `makePathFunc` recebe as sub-árvores das rotas filhas do segmento de rota atual como `opts.children` , e as árvores dos seus segmentos irmãos como contexto. Ela então constóri um um dicionário com essas informações relacionando as rotas completas com seus respectivos componentes, como explorado em mais detalhes na Seção 3.3.6. O fluxo de `makePathFunc` é representado pela Figura 3.9.

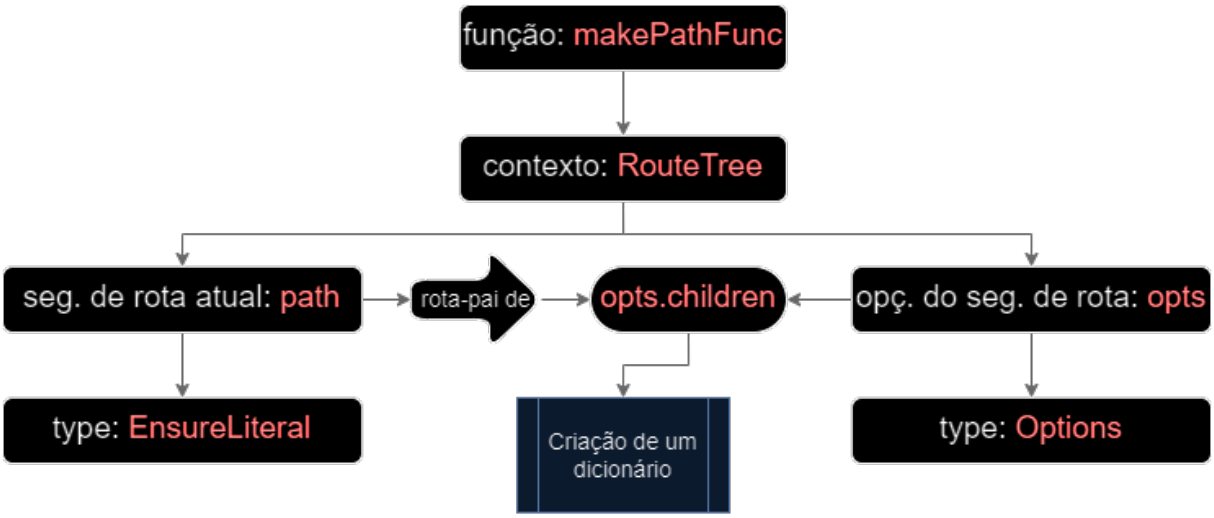

Figura 3.9: Diagrama representando como o *makePathFunc* cria um dicionário de rotas.

3.3.4 Uso da árvore de roteamento

Uma vez com a lógica de roteamento definida, é necessário integrar essa lógica nas funcionalidades de roteamento e navegação da aplicação.

Para isso são necessárias novas utilidades, cada uma com sua tipagem derivada do tipo `Routes` da árvore de rotas específica da aplicação. Essas utilidades foram centralizadas numa função `UseRoutes` , que recebe as rotas construídas pelo *builder* e o tipo respectivo como parâmetros, passa eles para as fábricas das utilidades necessárias e retorna um objeto com elas.

Uma fábrica, ou *Factory*, é um *design pattern* [22] em que é construída uma interface que consegue criar objetos de uma determinada superclasse sem uma classe definida. E a decisão de qual classe será atribuída à nova instância criada é delegada para dentro dessa interface.

Esse método ajuda a reduzir a quantidade de duplicação de código, uma vez que não é necessário fazer uma interface e eventuais decisões subsequentes para cada gerador de instância, já que é possível centralizar as decisões em uma só interface.

```ts
export const UseRoutes = <UserRoutes>(routes: UserRoutes) => ({
  Link: makeLink<UserRoutes>(routes),
  Router: makeRouter<UserRoutes>(routes),
  useRouteParams: makeUseRouteParams<UserRoutes>(routes),
})
```

Bloco de Código 3: Definição de `UseRoutes` , usando as fábricas `makeLink` , `makeRouter` e `makeUseRouteParams` .

3.3.5 Camada de integração com *react-router-dom*

Como exemplo, na Figura 3.10, a implementação do roteador em si consiste em somente um uso direto dos componentes `BrowserRouter` , `Routes` e `Route` de *react-router-dom*, junto com o dicionário de rotas gerado do construtor da árvore de roteamento.

```ts
import { BrowserRouter, Route, Routes } from "react-router-dom"
import { Component } from "./types/globals"

// função que filtra o objeto de rotas para extrair somente as rotas, sem métodos internos
const getConcreteRoutes = <UseRoutes,>(routes: UseRoutes) =>
  Object.fromEntries(
    Object.entries(routes as object).filter(([key]) => key[0] === "/")
  ) as { [k: string]: Component }

export const makeRouter =
  <UserRoutes,>(routes: UserRoutes) =>
    () => {
      return (
        <BrowserRouter>
          <Routes>
            {" "}
            {Object.entries(getConcreteRoutes(routes)).map(
              ([path, Component]) => (
                <Route path={path} Component={Component as any} />
              )
            )}
          </Routes>
        </BrowserRouter>
      )
    }
```

Figura 3.10: Construção do roteador da biblioteca, usando internamente *react-router-dom*.

3.3.6 Definições utilizadas

Nesta seção, é apresentado brevemente as estruturas que foram criadas para comporem a biblioteca *ragic*, e o seu respectivo funcionamento.

- `EnsureLiteral` : Garante que o compilador *TS* só aceite tipos *string literal*, em vez do tipo *string*. Isso é feito usando um tipo parametrizado condicional, e o tipo base `never` .

- `makePathFunc` : Constrói um dicionário de rotas, usando conceitos de contexto, informação atual e informação de filhos. A informação atual é o segmento de rota atual `path` e a informação dos filhos vem de `opts` , com outras opções do segmento de rota.
- `PathFuncReturnByOpts` : Funções de utilidade para construção do tipo da árvore de roteamento. `PathFuncReturnByOpts` checa se o tipo entrando `UserOpts` é válido. Caso passe nessa checagem, os tipos são recebidos por `PathFuncReturn` .
- `PathFuncReturn` : Identifica qual o tipo de segmento sendo construído (vazio, concreto ou apelido), e constrói uma nova árvore de roteamento a partir dos parâmetros e da rota anterior usando os tipos de utilidade `Routes` e `Segment` .
- `Segment` : Tipo feito para estruturar a informação sobre um segmento da árvore de rotas, na forma de tupla de segmentos de rota. Onde os filhos de um nó são representados como tupla no parâmetro `ChildRouteTree` .
- `Routes` : Dicionário contendo todas as rotas concretas na árvore, o método `path` , o `builder` e um *type brand*.
- `ConcretePaths` : Tipo de utilidade que a partir do tipo `RouteTree` , recursivamente extrai e compila o caminho com todas as rotas concretas da árvore. Usa recursão de cauda e tipos condicionais para construir o tipo final.
- `CompilePaths` : Compila uma tupla de *string literals* a partir de um parâmetro `Routes` . Usa recursão de cauda e tipos condicionais para construir o tipo final.
- `JoinPaths` : Concatena as *string literals*, por meio de recursão de cauda.
- `Router` : Componente *React* representando a lógica de roteamento em si, multiplexando entre as páginas conforme a *URL* do navegador.
- `Link` : Componente *React* usado para realizar a navegação entre rotas diferentes. Para validação usa os tipos `LinkProps` , `RouteIndexParams` e `ConcreteRoutes` .
- `useRouteParams` : *Hook* usado para buscar os parâmetros da rota atual. Para validação usa os tipos `RouteIndexParams` e `IndexedPaths`

3.4 Documentação

Uma parte importante da etapa de publicação, é o confecção de um guia tanto de download da biblioteca quando para seu uso, uma documentação. O desafio nessa etapa então foi

refazer os passos feitos no planejamento da biblioteca, reimaginar seus cenários de uso e explorar possíveis dificuldades e gargalos da experiência de desenvolvedor que poderiam ser ignorados ou esquecidos.

A principal preocupação é a de passar de forma clara e concisa para o usuário as informações que ele precisa saber para usar o básico da biblioteca, e deixá-lo preparado para experimentar e explorar usos mais diversificados da biblioteca por si.

Para tanto, foi construído um projeto simplificado com o objetivo de utilizar *ragic* do ponto de vista de um usuário não acostumado com a biblioteca, e anotar durante o desenvolvimento as etapas que se sabia serem necessárias para o uso da biblioteca, de forma que tais anotações sirvam de guia para os novos usuários. Além das etapas básicas e necessárias, propositalmente procurar casos de borda onde a biblioteca poderia não funcionar de forma devida ou mesmo apresentar erros. E usando essas descobertas como motivação, foi possível melhorar *ragic* várias vezes ao longo dessa etapa.

A página da biblioteca no NPMJS, bem como sua documentação, se encontram no link a seguir: (`https://www.npmjs.com/package/ragic`).

Capítulo 4

Validação da ferramenta

Após a conclusão da etapa de desenvolvimento e de publicação da biblioteca no *NPMJS*, foi preciso então testar seu uso.

Pensando nisso, foi feita uma série de testes variados para evidenciar a eficiência da biblioteca do ponto de vista de vários parâmetros, dando foco à usabilidade da biblioteca e o que ela agrega para a *DX*, uma vez que esse é o principal objetivo da ferramenta.

4.1 Prova de conceito: *ragic-pokedex*

O primeiro e mais simples teste foi um teste de conceito. Foi criado do zero uma aplicação mínima para confirmar e exemplificar as funcionalidades centrais da biblioteca. Dessa motivação, foi criada a *ragic-pokedex*.

Com inspiração pela tecnologia de mesmo nome da famosa série multimídia *Pokèmon*, foi criada uma aplicação para visualizar de forma geral e específica as informações sobre as criaturas titulares da série. Além de familiarização com o conteúdo original, a maior motivação para o conceito foi a *PokeAPI*, uma *API* pública e *open-source* disponível no link https://pokeapi.co, que possibilita simples e fácil acesso às informações necessárias para a construção da aplicação. Como uma lógica de *backend* básica era necessária para o funcionamento da aplicação mas a implementação desse *backend* não auxilia no objetivo de teste da biblioteca, a escolha por uma *API* pública similar à essa foi simples.

4.1.1 Definição da Aplicação

Para ter a aplicação mais simples possível que envolvesse todas as funcionalidades básicas da biblioteca, primeiro foi imaginada uma página inicial simples, *Home*. O intuito dessa página, visível na Figura 4.1, e rota é de verificar o roteamento padrão da rota, e funcionar como menu principal para acesso às outras rotas.

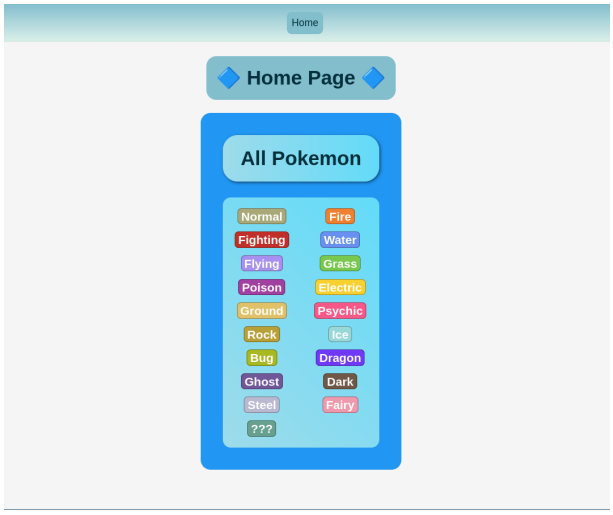

Figura 4.1: Página *Home* da aplicação teste.

É necessário também ter uma página simples para listar os *Pokemons*, com *links* para as páginas de cada *Pokemon*. Essa página é mostrada na Figura 4.2

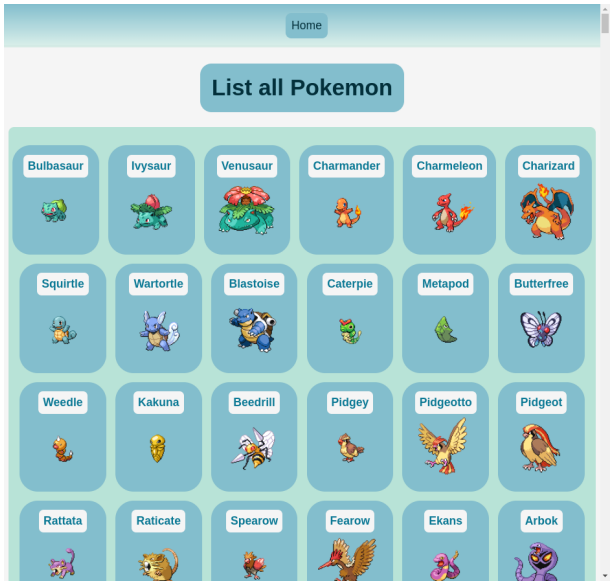

Figura 4.2: Página de listagem de todos os *Pokemons*.

Como mencionado antes, deve existir uma rota de índice que captura da *URL* o identificador do *Pokemon* em questão e busca os dados necessários do *backend* e mostra eles. Essa página é visível na Figura **??**

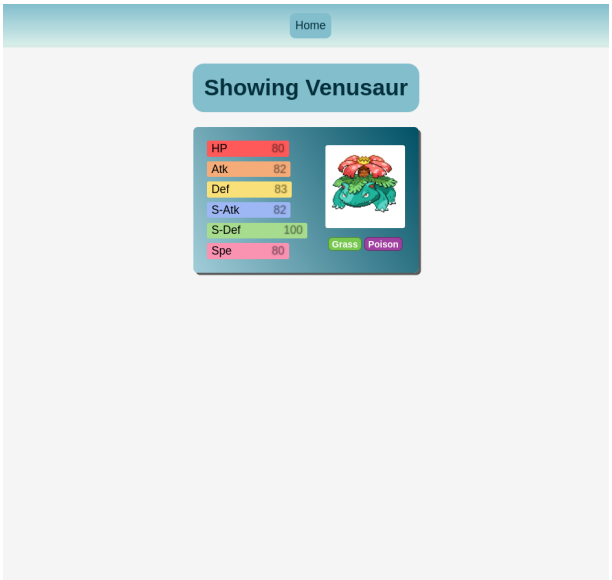

Figura 4.3: Página para visualização dos detalhes de um único *Pokemon*.

Com o intuito de testar rotas do tipo vazio, foi pensada uma rota de índice que filtra os *Pokemons* por tipo. Ao manter essa página na rota `"/type/:type_id"` , mas não relacionar nenhuma página com a rota `"/type"` , a rota `"/type"` é tida como um segmento vazio que pode ser verificado. Um exeplo dessa página é visível na Figura 4.4.

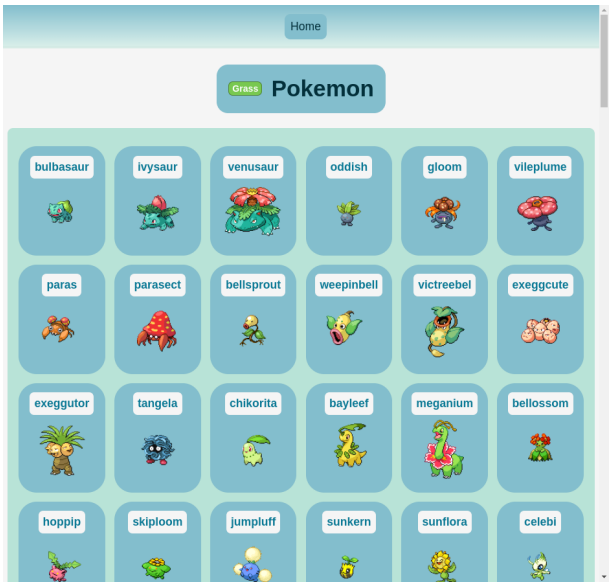

Figura 4.4: Página para listagem de todos os *Pokemons* de um tipo específico.

4.1.2 Implementação

A aplicação de teste foi construída como planejado. Usando uma infraestrutura de *monorepo*, o código fonte dela está disponível no mesmo repositório onde está o código fonte da implementação de *ragic*. Foi feita uma implementação usando *react-router-dom*, disponível na pasta `app-react-router` , e uma usando *ragic* em `app-ragic` .

4.1.3 Definição da árvore de roteamento

Em *react-router-dom*, a definição deve ser feita utilizando os componentes `Routes` e `Route` como no Bloco de Código 4

```
import { Route, Routes } from 'react-router-dom';

import { HomePage } from './pages/HomePage';
import { ListAllPokemonPage } from './pages/ListAll';
import { ShowPokemonPage } from './pages/ShowPokemon';
import { ListPokemonByTypePage } from './pages/ListByType';

export const Router = () => (
  <Routes>
    <Route index element={<HomePage />} />
    <Route path="pokemon">
      <Route index element={<ListAllPokemonPage />} />
      <Route path=":pokemon_id" element={<ShowPokemonPage />} />
    </Route>
    <Route path="type">
      <Route path=":type_id" element={<ListPokemonByTypePage />} />
    </Route>
  </Routes>
);
```

Bloco de Código 4: Definição da árvore de roteamento em *react-router-dom*.

Na implementação em *ragic*, é usado o *builder* como no Bloco de Código 5

```
import { UseRoutes, createRoutes, path } from "@ragic/ragiclib"
import { HomePage } from "./pages/HomePage"
import { ListAllPokemonPage } from "./pages/ListAll"
import { ListPokemonByTypePage } from "./pages/ListByType"
import { ShowPokemonPage } from "./pages/ShowPokemon"

const routes = createRoutes()
  .path("/", { component: HomePage })
  .path("/type", {
    children: path("/:type_id", { component: ListPokemonByTypePage }),
  })
  .path("/pokemon", {
    component: ListAllPokemonPage,
    children: path("/:pokemon_id", { component: ShowPokemonPage }),
  })

export const { Link, Router, useRouteParams } = UseRoutes(routes)
```

Bloco de Código 5: Definição da arvore de roteamento usando `createRoutes` .

Na implementação em *ragic*, no arquivo `src/components/Navbar.tsx` foi usado pela primeira vez o componente `Link` gerado por `UseRoutes` para fazer o *link* de volta para a página inicial.

```
import styled, { createGlobalStyle } from "styled-components"
import { Link } from "../routes"

/* ... */

export const Navbar = () => {
  return (
    <Header>
        /* ... */
        <Link to="/">Home</Link>
      </NavList>
    </Header>
  )
}
```

Bloco de Código 6: Uso de `Link` para criação de *link* para a página *Home*.

Em *react-router-dom*, apesar da implementação final ser similar, como a tipagem da propriedade `"to"` de `Link` espera uma `string` , a validação da rota não é feita. Em contrapartida, quando na implementação em *ragic*, se espera receber o tipo `"/" || "/pokemon" || "/pokemon/:pokemon_id" || "/type/:type_id"` , o que retorna um erro de compilação com um caminho inválido.

Na Figura 4.5 pode-se ver um exemplo de uso válido do componente `Link` em qualquer uma das duas bibliotecas. Na Figura 4.6, apesar de o `Link` de *react-router-dom* receber uma rota inválida, nenhum erro é apontado. A mesma situação usando *ragic*, como na Figura 4.7, permite um *feedback* imediato do erro para o programador.

```
export const Navbar = () => {
  return (
    <Header>
      <NavbarParentStyle />
      <NavList>
        {/* válido */}
        <Link to="/"></Link>
      </NavList>
    </Header>
  )
```

Figura 4.5: Uso válido de `Link` em *react-router-dom* ou *ragic*.

```
export const Navbar = () => {
  return (
    <Header>
      <NavbarParentStyle />
      <NavList>
        {/* inválido, mas sem erro */}
        <Link to="/home">Home</Link>
      </NavList>
    </Header>
  );
};
```

Figura 4.6: Uso de um caminho inválido em um `Link` em *react-router-dom*.

```
export const Navbar = () => {
  return (
    <Header>
      <NavbarParentStyle />
      <NavList>
        {/* inválido */}
        <Link to="/home"></Link>    Type '"/home"' is not assignable to type '"/" | "/
      </NavList>
    </Header>
  )
}
```

Figura 4.7: Uso de um caminho inválido em um `Link` em *ragic*.

Em `app-ragic/src/pages/HomePage.tsx` são usados um *link* normal para a página de listagem de todos os *pokemons* e um de índice para as páginas de listagem filtrada por

tipo. No *link* das páginas de tipo temos o primeiro uso de *links* com parâmetros, onde é garantido pela tipagem estática que o componente de `Link` receberá como propriedade `"params"` um objeto com as informações necessárias para todos os índices existentes da rota designada pela propriedade `"to"` , e somente os existentes. Um *snippet* desse código é visível na Figura 4.8

```
export const HomePage = () => {
  return (
    <>
      <Navbar />
      <PageContent>
        <PageTitle>🔷 Home Page 🔷</PageTitle>

        <NavigationPannel>
          <Link to="/pokemon">All Pokemon</Link>
          <ul>
            {PokemonTypes.map(({ name, id }) => (
              <li key={name}>
                <Link to="/type/:type_id" params={{ type_id: `${id}` }}>
                  <TypeBadge name={name}>{name}</TypeBadge>
                </Link>
              </li>
            ))}
          </ul>
        </NavigationPannel>
      </PageContent>
    </>
  )
}
```

Figura 4.8: Uso de `Link` na página inicial.

De forma similar com os *links* para as páginas de filtro por tipo, nas páginas que contém listas de pokemons usam *links* do tipo índice para redirecionar o usuário corretamente, como no Bloco de Código 7.

```
export const ListAllPokemonPage = () => {
    const data = useAllPokemon()
    return (
        <>
            /* ... */
            <PokelistContainer>
                /* ... */
                data.map((pokemon) => (
                    <Link to="/pokemon/:pokemon_id" params={{ pokemon_id: pokemon.id
                    ↪ }}
                        <PokeCard pokemon={pokemon} key={pokemon.name} />
                    </Link>
                ))
            </PokelistContainer>
            /* ... */
        </>
    )
}
```

Bloco de Código 7: Uso de *links* de índice para redirecionar o usuário para a página de detalhes de um *Pokemon*.

Tanto para a página de detalhes de um único *Pokemon* quanto para a página de listagem de *Pokemon* por tipo é necessário extrair informação da *URL* antes de requisitar do *backend* os dados necessários. Para isso foi feito uso do *hook* `useRouteParams` , como exemplificado no Bloco de Código

```
import { useRouteParams } from "../routes"
import { usePokemon } from "../api/fetchPokemonInfo"

export const ShowPokemonPage: FC = () => {
  const { pokemon_id } = useRouteParams("/pokemon/:pokemon_id")
  const pokemonData = usePokemon(pokemon_id)

  return (
    <>
        /* uso das informações sobre o pokemon inclusas no objeto pokemonData */
    </>
  )
}
```

Bloco de Código 8: Uso de `useRouteParams` para extrair informação da *URL*.

4.1.4 Testes *end-to-end* com *Cypress*

Para validar que a implementação usando *ragic* contempla o contrato da ideia da aplicação de exemplo e evidencia a validade da biblioteca, foram construídos testes *end-to-end* usando a ferramenta *Cypress*. As definições dos testes estão disponíveis na pasta *app-ragic/e2e* no repositório *git* da biblioteca. O restultado dos testes é visível na Figura 4.9

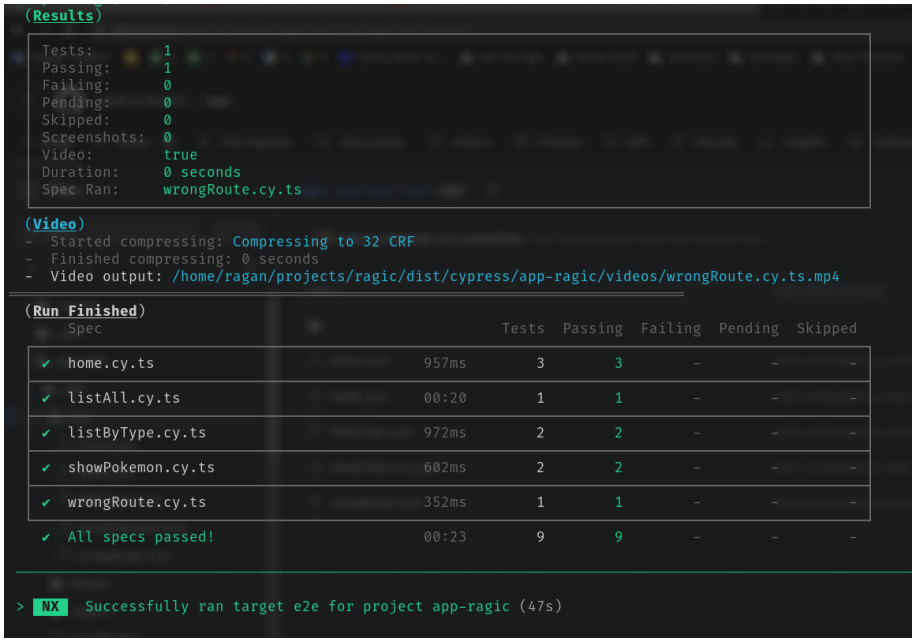

Figura 4.9: Resultado após rodar os testes *e2e* da implementação da aplicação teste em *ragic*.

4.2 Estudo de caso: *CGR*

CGR é um projeto interno da empresa júnior de ciência da computação da UnB, *CJR*. O intuito do *CGR* é ser um repositório para o banco de horas da contribuição semanal dos membros, junto com uma loja de gameficação com recompensas simples para gastar as moedas recebidas por contribuições para a empresa e uma base de conhecimento geral da empresa, desde tecnologias de desenvolvimento de sites e aplicações até estratégias de venda e de publicidade.

Com permissão da *CJR*, o código fonte do *frontend* do *CGR* foi cedido, e o projeto reimplementado em *ragic*.

A intenção com isso é não só evidenciar a funcionalidade básica da biblioteca, como no teste anterior, mas também fazer uma análise que indique alterações na performance final das duas aplicações.

Como efeito secundário, a reimplementação de uma aplicação já existente em vez de uma idealizada pela equipe de desenvolvimento, é removida uma das possíveis fontes de viés nesse teste.

Com o código fonte recebido da *CJR* foi criado um repositório *git* público, disponível em https://github.com/rafael-g-depaulo/ragic-cgr. A implementação original, usando *react-router-dom*, está no *commit* rotulado pela tag *react-router-dom*, e similarmente para a implementação usando *ragic*.

4.2.1 Performance de aplicações *web* e *lighthouse*

Aplicações *web* sofrem algumas restrições particulares por conta da sua integração com os protocolos nativos da internet, o modelo de cliente-servidor em si e a própria infraestrutura da *web* da qual a aplicação depende para ser entregue no computador do usuário final e finalmente poder ser executada por um navegador.

Quando se trata de performance *web* e como essa performance afeta a *UX* final e a usabilidade da aplicação, vários fatores e características da aplicação e da infraestrutura integrada da solução *web* devem ser consideradas, como *caches*, diferentes formas de organização e entrega dos arquivos e dados da aplicação, entre outros.

Algumas métricas surgem desses fatores potencialmente negativos para a performance da aplicação, e ferramentas são construídas para analisar, compilar e relatar essas métricas. Uma das mais populares e confiadas no ecossistema é *Google Lighthouse*.

4.2.2 *First Contentful Paint*

Uma das métricas mais importantes para uma aplicação *web* é *First Contentful Paint* (*FCP*[23]). Essa métrica mede o tempo em segundos desde a primeira requisição para o servidor até o recebimento da página, carregamento dela e o primeiro instante onde conteúdo significativo é disponibilizado para o usuário final. Um exemplo de FCP pode ser visto na Figura 4.10.

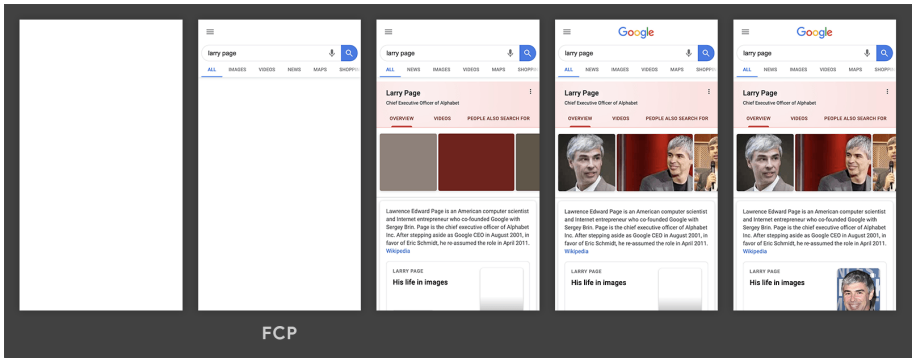

Figura 4.10: Exemplo de *FCP* provida como exemplo na documentação oficial de *lighthouse* [1].

Apesar de ser uma ótima métrica para análise de performance de uma aplicação real em ambiente de produção, *FCP* por si só não é ideal para uma análise puramente do código final de uma aplicação em vez de uma infraestrutura completa de uma solução *web*. *FCP* acaba incorporando *delays* e outros ruídos por seu escopo holístico de medição. De forma similar como um teste *end-to-end* ou de integração pode não ser o ideal para testar uma parte pequena de um projeto de software, existe uma outra métrica que influencia diretamente o *FCP* que pode ser usada para o teste.

4.2.3 *Application Bundle* e *Bundle size*

Uma prática comum para distribuições de aplicações *web*, como notado por [24], é compilar e minificar os arquivos *HTML, JS, CSS* e outros *assets* que compõem a aplicação, e servir um subconjunto desses arquivos para o cliente quando ele requisitar a aplicação.

Minification, ou "minificação", é o nome dado à pratica de se simplificar códigos em *JavaScript* no contexto de aplicações *web*, com o intuito de fazer com que a transmissão do código fonte pela rede seja feita de forma mais eficiente.

Essa simplificação envolve na redução de nomes de variável e a remoção de espaços em branco, comentários, caracteres de nova linha, entre outros que são adicionados para aumentar a legibilidade do código. Mas obviamente, essa simplificação deve ocorrer sem prejuízo à função do código. Ou seja, devem ser remoções e reduções que mantenham a funcionalidade do código.

Minificação é uma prática comum no contexto de desenvolvimento *web*, dado que a redução do conteúdo do código acarrete em melhora na velocidade e na acessibilidade do site, impactando diretamente e positivamente a experiência de usuário de alguém que usa dado site. Essa prática é relevante para o teste de *bundle size*, descrito na Seção 4.2.3.

Essa prática reduz o tamanho total da aplicação a ser transferido pela rede. Como o tempo de transferência desse *bundle* é o maior gargalo para a performance da aplicação sob um ponto de vista de experiência do usuário final, foi utilizado o tamanho total dos arquivos constituintes desse *bundle* como métrica para o teste entre as duas implementações do *CGR*.

A métrica escolhida para a análise final entre as duas implementações da aplicação é então o *bundle size*, uma medição simples to tamanho total dos arquivos constituintes do *bundle* da aplicação compilada.

O fator que mais contribui para *FCP* é o tempo até a página inicial, ou seja o *bundle*, ser entregue após a requisição inicial. Esse tempo é definido pelo tempo de *round-trip* entre o computador do usuário e servidor, e pelo tempo de transmissão do *bundle*. Esse tempo de transmissão é diretamente afetado pelo *bundle size*, e por isso foi considerado uma métrica aceitável para o teste.

Como efeito secundário, essa análise é interessante também porque *ragic* é intencionalmente uma biblioteca que não oferece muitas possibilidades novas para o programador da aplicação, e só tem foco na melhora da confiabilidade final do *software* e na velocidade de desenvolvimento. Dado isso, um dos maiores possíveis *drawbacks* da biblioteca deve ser o possível impacto negativo na performance da aplicação, e por meio disso na experiência do usuário final.

Isso é uma possibilidade particularmente importante de ser investigada, dado que a implementação inicial da biblioteca é puramente uma camada de abstração em cima de *react-router-dom*. Ou seja, *ragic* não dá por si só nenhuma melhoria direta para o usuário final, mas é provável que afete negativamente o *bundle size*.

4.2.4 Análise de *bundle size* gerado

O código fonte da implementação original do *CGR*, uma vez compilada, gera um *bundle* inicial de 113,492KB distribuído entre 5 arquivos, como mostra a Figura 4.11.

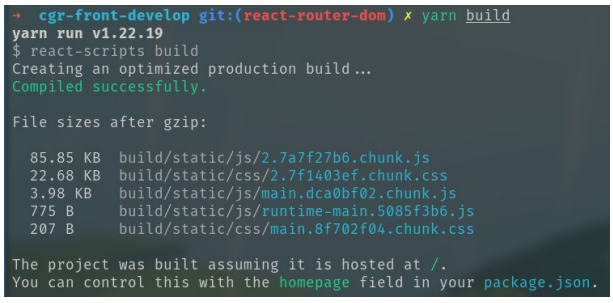

Figura 4.11: *Bundle* resultante da compilação da implementação da aplicação teste em *react-router-dom*.

A implementação em *ragic* gerou um *bundle* inicial de 142,662KB, como mostra a Figura 4.12.

```
➜  cgr-front-develop git:(ragic) yarn build
yarn run v1.22.19
$ react-scripts build
Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  114.78 KB  build/static/js/2.4f90842b.chunk.js
  22.68 KB   build/static/css/2.7f1403ef.chunk.css
  4.22 KB    build/static/js/main.d3d3c422.chunk.js
  775 B      build/static/js/runtime-main.5085f3b6.js
  207 B      build/static/css/main.8f702f04.chunk.css

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.
```

Figura 4.12: *Bundle* resultante da compilação da implementação da aplicação teste em *ragic*.

Para completar o *bundle* final da aplicação, são adicionados como objetos os arquivos públicos da aplicação que não precisam ser compilados. Esses arquivos são idênticos nas duas implementações, então não afetam a diferença absoluta entre os *bundle sizes*. Como mostra a Figura 4.13, os arquivos públicos do *CGR* contribuem para 2,27KB do *bundle* final.

```
➜  public git:(ragic) ls -l -h
total 12K
-rw-rw-r-- 1 ragan ragan 1,9K set 15  2021 index.html
-rw-rw-r-- 1 ragan ragan  306 set 15  2021 manifest.json
-rw-rw-r-- 1 ragan ragan   67 set 15  2021 robots.txt
➜  public git:(ragic) git switch --detach react-router-dom
Previous HEAD position was 5d56d95 Add Ragic
HEAD is now at 6716573 Add resolution for acorn
➜  public git:(react-router-dom) ls -l -h
total 12K
-rw-rw-r-- 1 ragan ragan 1,9K set 15  2021 index.html
-rw-rw-r-- 1 ragan ragan  306 set 15  2021 manifest.json
-rw-rw-r-- 1 ragan ragan   67 set 15  2021 robots.txt
```

Figura 4.13: Tamanho dos arquivos públicos usados nas duas implementações da aplicação.

Ao todo, o *bundle* final da implementação com *react-router-dom* ficou em 115,76KB, e o em *ragic* ficou em 144,93KB. Uma simples comparação entre os dois é mostrada na Figura 4.14.

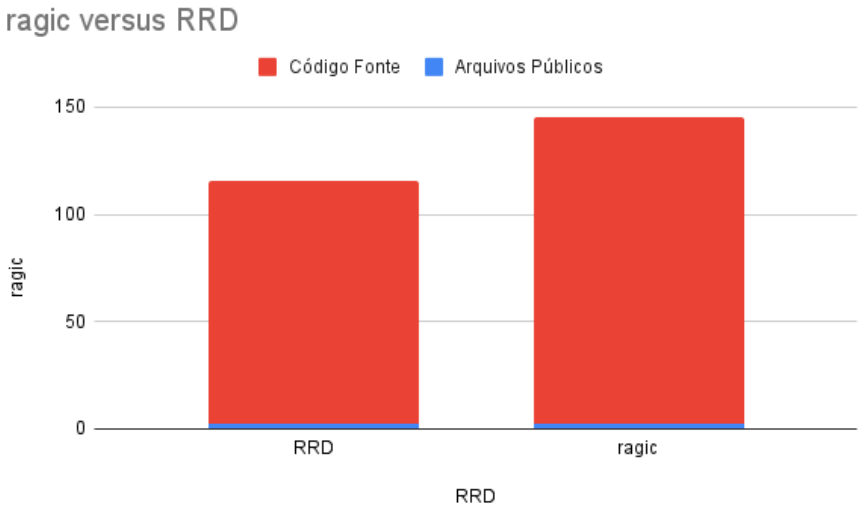

Figura 4.14: Gráfico comparando o tamanho final do *bundle* das duas implementações.

Com uma diferença total de 29,17KB de diferença entre os dois *bundles*, o impacto de usar *ragic* na implementação do *CGR* no tamanho do *bundle* final é de aproximadamente 25,1%.

Em valores relativos, um aumento de 25,1% é considerável. Em valores absolutos, entretanto, 29,17KB está dentro de uma margem aceitável para o atual escopo da biblioteca e o atual estado do seu desenvolvimento. Considerando também que a seção de lógica de roteamento, quando existente, é geralmente uma parcela pequena de uma aplicação, então esse resultado negativo é considerado como um indicativo de melhorias importantes, e não como problema de uma escala que invalide o uso da biblioteca numa parcela grande dos casos de uso.

A diferença percentual de *bundle size* entre as duas implementações de *CGR* é grande, e isso se dá parcialmente pelo escopo pequeno da aplicação. Isso é relevante ao se analizar que esse aumento do *bundle size* da implementação em *ragic* não é um aumento percentual em cima da implementação em *RRD*, mas sim um aumento constante, dado que a diferença entre *ragic* e *RRD* é uma quantia fixa de código a mais.

Considerando o contexto apresentado, em aplicações mais complexas com *bundle sizes* medianos, uma diferença de 30KB é razoável. Mesmo assim, existem claramente situações onde uma diferença de 30KB é significante, e o impacto no *bundle size* é um dos maiores pontos negativos para adoção de *ragic*. Particularmente em aplicações de escopo pequeno com foco em acessibilidade que possuam um público alvo cujo acesso à internet seja muito limitado.

Quando um usuário depende de usar seus limitados dados *3G* para acessar uma aplicação, cada *byte* é valioso, e da mesma forma eles impactam dispositivos com pouca memória *RAM* disponível. Dado isso e o impacto de *bundle size* em *FCP* mencionado na Seção 4.2.2, a redução do impacto de *ragic* no *bundle size* é um ponto de grande foco para o futuro da biblioteca.

Como conclusão final observa-se que a implementação em *ragic* é funcional, e apesar do *bundle size* ser maior, tendo em mente o contexto de que *ragic* usa *RRD* como dependência, esse aumento era esperado. O planejamento é então de, no futuro próximo após a remoção da dependência de *RRD*, realizar novamente esse teste e reavaliar as diferenças entre as implementações com *RRD* e com *ragic*.

4.3 Experimento qualitativo preliminar

Pensando em averiguar se o projeto está no caminho certo de promover uma melhora da *DX*, foi realizada uma pesquisa com alguns desenvolvedores *React* com a intenção não só de checar uma opinião inicial sobre a biblioteca, mas também de coletar *feedbacks* sobre quais áreas da *ragic* precisam de mais foco e atenção no seu desenvolvimento contínuo, para que ela se adéque ao ambiente de desenvolvimento e ás aspirações iniciais que motivaram seu desenvolvimento.

4.3.1 Perfil dos entrevistados

Como a intenção é avaliar *ragic* em comparação com outras bibliotecas de roteamento do ambiente, o público alvo dessa pesquisa são desenvolvedores *React* com alguma experiência prévia.

Pelo contexto acadêmico do trabalho, deu-se uma preferência ao convite de alunos e ex-alunos de computação da UnB, mas o requisito de se ter experiência prévia em *React* limitou essa busca, sendo necessário então encontrar alguma fonte com abundância de desenvolvedores que se encaixem nesse perfil.

Um grupo encontrado com essas características é a CJR, empresa júnior de ciências de computações da UnB. Por ter em sua *stack* o *framework react*, vários membros atuais e ex-membros da empresa tem o conhecimento necessário. Primeiramente foi feito um convite aberto e amplo em canais de comunicação internos da empresa com a ajuda de alguns membros, e em seguida foram feitos convites individuais para pessoas que demonstraram interesse em participar da pesquisa.

O convite em aberto teve alcance de 150 pessoas, mas o que se mostrou mais efetivo foram os convites individuais em privado, que foram feitos para 20 avaliadores em poten-

cial. Dos 20 *devs* convidados individualmente, 9 voluntários conseguiram dispor de tempo e esforço para a realização do teste e participação da pesquisa.

4.3.2 Realização da pesquisa

Após o contato inicial com *devs* desses grupos informando o motivo do pesquisa e pedindo pela participação, foi disponibilizado um repositório contendo já pronto um *web app* com integração com *API*, componentes visuais e grande parte da lógica de construção de páginas já prontos, faltando somente a criação de um sistema de roteamento entre as páginas, que deveria ser feito utilizando *ragic*.

Esse repositório é o mesmo apresentado na Seção 4.1, o *ragic-pokedex*. Apesar da *branch* principal conter a aplicação já funcional, há uma *branch* em que a definição das rotas não foi feita, e nem *ragic* foi instalada. A ideia é que os participantes clonem o repositório e usem esta *branch* para testarem a biblioteca.

o *README* desse repositório contém informações sobre de que forma esse roteamento deve ser feito, como se a equipe de desenvolvimento fosse o *Project Manager* dos desenvolvedores apresentando para eles os requisitos de um projeto. Adicionalmente, foi disponibilizado também a página da biblioteca no *NPMJS* (`https://www.npmjs.com/package/ragic`) que contém as instruções de instalação e a documentação da biblioteca.

Manteve-se um contato constante com os voluntários para acompanhar o progresso e poder descobrir o mais cedo possível as dificuldades e problemas enfrentados no caminho por eles, e checar se esses problemas são devido à logica interna da biblioteca ou à falta de clareza na documentação. Ambas situações pedem da equipe de desenvolvimento a interferência e atualização da biblioteca.

Ao final do desenvolvimento do *app web*, foi passado para os participantes um formulário onde eles poderiam avaliar diversos aspectos da experiência de uso da biblioteca, como usabilidade, facilidade do aprendizado, simplicidade, conforto de uso e clareza da documentação. Após a coleta desses dados, a intenção é de verificar em qual(is) área(s) *ragic* tem uma performance melhor que a biblioteca que um determinado participante relatou ser sua biblioteca de uso padrão. E para as áreas que *ragic* não performou tão bem quanto, verificar o motivo e se é possível fazer algo que melhore essa avaliação em uma possível bateria de testes futura. O formulário está disponível no seguinte link: `https://docs.google.com/forms/d/e/1FAIpQLSfSE-8KWzb_SdBir87QAVSmnTtyoT0ehUr4gpKP-4bLkCViOQ/viewform?usp=sharing`.

Nesse formulário de avaliação, a primeira coisa que é perguntada para os desenvolvedores é qual a biblioteca de roteamento eles usam para desenvolvimento *React*. A intenção dessa pergunta é avaliar se a amostra de desenvolvedores corresponde com a observação feita pela Tabela 1.1, que relata um domínio de *RRD* no mercado atual. E de forma

secundária, buscamos coletar outras bibliotecas para avaliar quais as outras competidoras que *ragic* tem atualmente.

Em seguida, pede-se para que eles pontuem de 1 a 5 alguns parâmetros relacionados à experiencia de uso da biblioteca, onde "1"quer dizer "preferência à biblioteca que já uso", 2 quer dizer "inclinação para a biblioteca que já uso", 3 quer dizer "não tenho preferências", 4 quer dizer "inclinação para *ragic*"e 5 quer dizer "preferência a *ragic*".

A decisão de se perguntar comparando *ragic* com "a biblioteca que já uso"em vez de direta e explicitamente com*RRD* se deve à expectativa de que alguns dos desenvolvedores não tenham experiência com *RRD*, ou tenham mais prática ou preferência por alguma outra biblioteca.

Os critérios de avaliação foram apresentados na primeira pessoa para reforçar que a resposta deve representar a opinião dos entrevistados.

Dessa forma, as perguntas desse questionário são como a seguir:

1. Eu achei mais simples de usar:
2. Eu achei mais confortável:
3. Eu achei mais intuitivo:
4. Eu usaria em projetos *React* simples, com poucas rotas/páginas:
5. Eu usaria em projetos *React* complexos, com várias rotas/páginas:

A escolha dessas perguntas é baseada no princípio que norteou o desenvolvimento da biblioteca, que é o de oferecer uma experiência mais confortável para o usuário (desenvolvedor). Logo, faz mais sentido checar se a biblioteca cumpre sua atividade básica de fazer o gerenciamento das rotas de uma aplicação, e ao mesmo tempo oferecendo uma *developer experience* melhor do que as outras opções.

Já as duas últimas perguntas se baseiam no *feedback* recebido por um dos desenvolvedores, que relatou sentir um conforto em usar *ragic* no experimento dado que o experimento continha poucas rotas, mas que em projetos maiores e mais complexos, ele talvez se sentisse desestimulado a usar *ragic* devido á natureza aninhada das declarações de rotas e rotas filhas.

Em seguida, são feitas perguntas direcionadas a *ragic* isoladamente, sem a necessidade de uma comparação direta com quaisquer outra ferramenta. Essas perguntas servem basicamente para avaliar o quão bem a biblioteca é apresentada. Dessa vez é pedido novamente um voto de 1 a 5, mas nessa seção o voto 1 quer dizer "muito ruim", "muito pouco", e o voto 5 quer dizer "muito bom", "bastante".

Estas perguntas são:

1. Quanto à clareza e facilidade de entendimento da documentação de *ragic*,
2. Quais as chances de você recomendar *ragic* para alguém?

A primeira pergunta não tem a finalidade de avaliar a usabilidade da biblioteca como as da sessão anterior, mas sim checar quão bem a documentação explica e apresenta a biblioteca, se isso é feito de forma sucinta e completa, relatando todas as informações necessárias para que a biblioteca possa ser utilizada.

Já a segunda pergunta é de certa forma, uma avaliação geral da biblioteca, como se compilasse todas as opiniões sobre diferentes aspectos dela em um só: Se o voluntário acha que valeria a pena alguém que não conhece a biblioteca, conhecer no futuro.

E por último, há um espaço para comentários gerais, *feedbacks* sobre a biblioteca como um todo, e sugestões para a sua forma atual e até possíveis implementações futuras. Apesar de ser um campo opcional, vários entrevistados apresentaram suas opiniões nesse campo.

4.3.3 Respostas

Após o fim do experimento qualitativo, e após os desenvolvedores responderem as perguntas do formulário de avaliação, obteve-se dados relevantes pertinentes ao uso da biblioteca e ao seu conforto em geral.

As figuras a seguir contém gráficos com as respostas dos desenvolvedores.

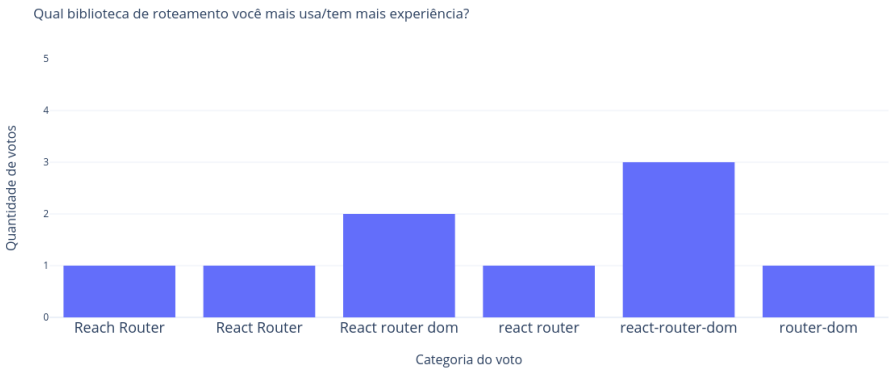

Figura 4.15: Bibliotecas usadas pelos testadores.

Por conta do campo de resposta da pergunta na Figura 4.15 ser um campo de escrita livre, facilmente houveram diferenças no formato de escrita entre os desenvolvedores. Mesmo que a maioria deles tenha respondido *RRD*, houveram varias diferenças na forma com que eles escreveram o nome da biblioteca. A exceção foi um dos voluntários, que

disse usar *Reach Router*. Isso corrobora a observação inicial de que *RRD* é a ferramenta mais utilizada no mercado.

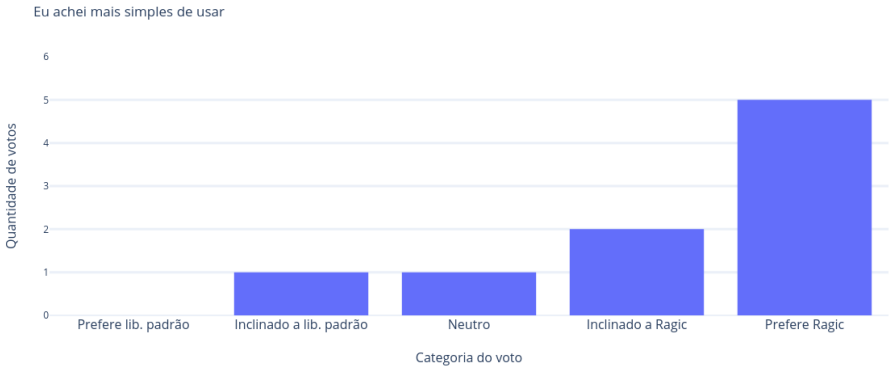

Figura 4.16: Opinião sobre simplicidade de *ragic*.

A Figura 4.16 relata a segunda pergunta, e dá a confirmação de que o uso de *ragic* é mais simples e mais fácil de se entender como funciona, o que contribui de forma direta para uma experiência de desenvolvimento melhor.

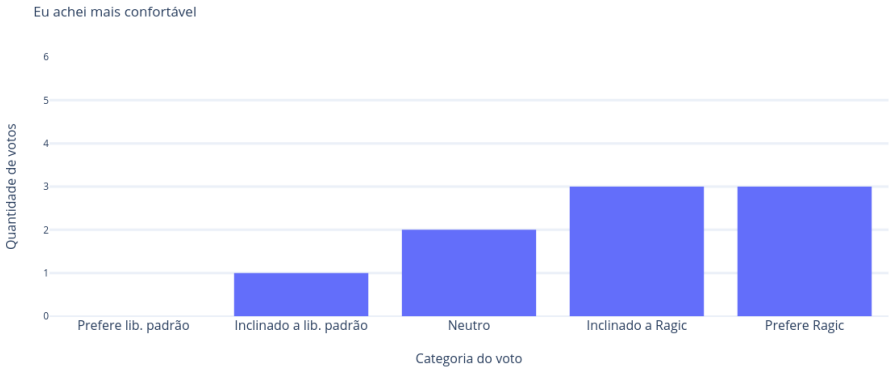

Figura 4.17: Opinião sobre conforto de *ragic*.

Já na Figura 4.17, observa-se que para a maioria dos desenvolvedores, *ragic* se mostrou de fato uma opção mais confortável comparada com a biblioteca de mais experiência deles, cumprindo o papel proposto pela biblioteca de oferecer mais conforto ao programador.

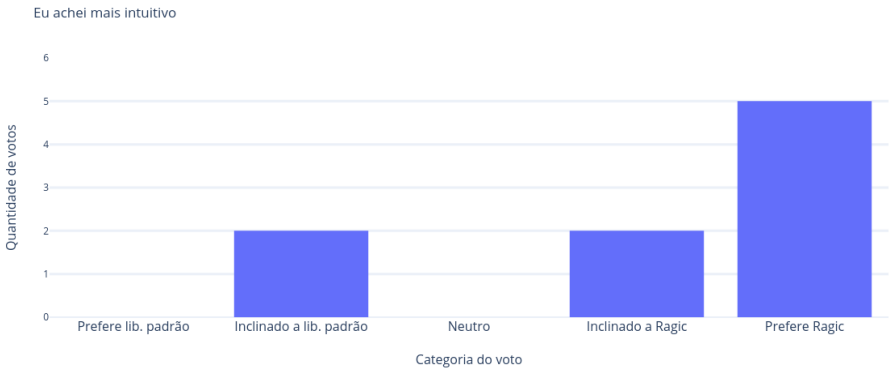

Figura 4.18: Opinião sobre quão intuitiva *ragic* é.

A Figura 4.18 comunica com os esforços da equipe de desenvolvimento de fazer uma ferramenta com boa apresentação e fácil aprendizado. O que, de novo, impacta diretamente na experiência do desenvolvedor.

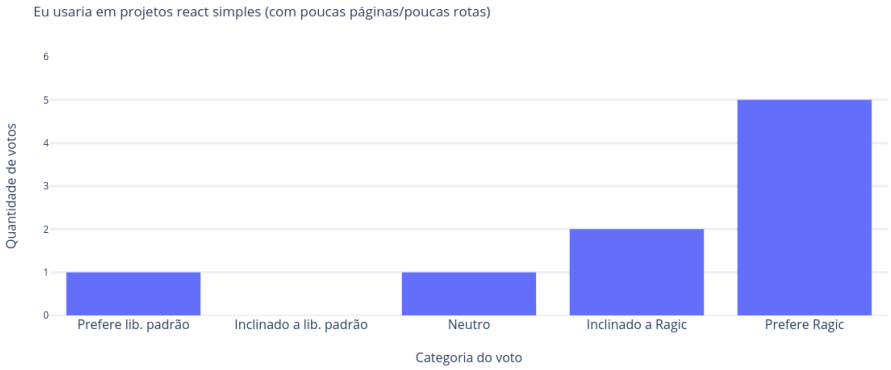

Figura 4.19: Opinião sobre *ragic* em projetos pequenos.

Figura 4.20: Opinião sobre *ragic* em projetos grandes.

O motivo das duas perguntas anteriores já foi explicado na apresentação do formulário da pesquisa na Seção 4.3.2, e a Figura 4.19 demonstra que para projetos menores, a maioria dos testadores considera que *ragic* seria sim uma opção mais agradável de uso do que as outras bibliotecas que eles tem costume de usar.

Já a Figura 4.20 mostrou uma preferência reduzida para *ragic* em projetos mais complexos, e de acordo com os relatos de alguns desenvolvedores isso se dá por conta de como a declaração de rotas é tratada em *ragic*.

Isso não foi uma possibilidade levada em conta durante o planejamento do design da biblioteca, então esse resultado foi um tanto inesperado. Ainda assim, o fato de ainda haver algumas respostas favoráveis para *ragic* indica que não é um consenso geral entre todos que participaram da pesquisa. O que sugere que há espaço para possíveis modificações no design no futuro para que essa parte em específico da biblioteca fique mais confortável e simplificada no futuro.

E mesmo na situação atual, as respostas apontam que para projetos mais simples, com exceção de um dos participantes, todos os outros julgaram *ragic* como uma melhor escolha ou no pior dos casos, uma escolha em pé de igualdade com outras opções. Projetos desse tipo são geralmente projetos de portfólio e sites institucionais por exemplo, o que incentiva um desenvolvedor em aprendizado a utilizar *ragic*.

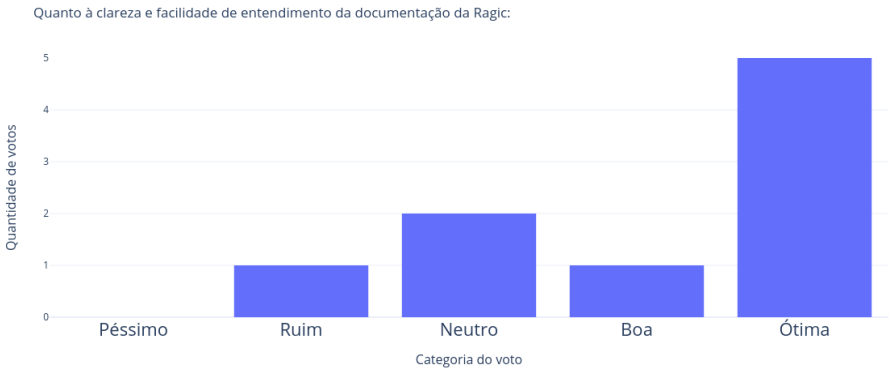

Figura 4.21: Opinião sobre a documentação de *ragic*.

Com uma resposta majoritariamente positiva, a Figura 4.21 indica que, apesar de algumas falhas, a documentação conseguiu apresentar a biblioteca de forma efetiva para novos usuários. Como mencionado antes, essa pergunta avalia mais a competência de transferência de informação da documentação do que o desempenho geral da biblioteca em si.

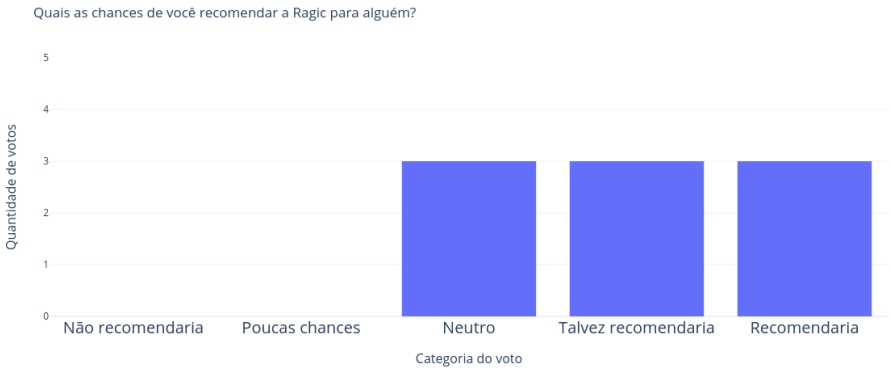

Figura 4.22: Opinião sobre a possibilidade de recomendação da *ragic*.

E por fim, a última pergunta objetiva, na Figura 4.22 condensa de forma geral a experiência da utilização da biblioteca como um todo. As respostas são em sua maioria positivas, indicando que para a maioria dos voluntários nessa pesquisa *ragic* conseguiu cumprir com sua obrigação enquanto ferramenta de roteamento. Enquanto que as respostas anteriores indicam um sucesso em sua proposta dentro da amostra de voluntários.

4.3.4 Conclusão

Com as respostas adquiridas nesse experimento observamos que, para a maioria dos participantes desta pesquisa, *ragic* promoveu sim uma experiência mais confortável; apesar da ocorrência de algumas situações inesperadas como a falta de interesse no uso de *ragic* para projetos grandes, com várias rotas.

Mesmo com essa resistência, por conta da aceitação em geral da biblioteca e do voto majoritariamente positivo para a simplicidade e intuitividade da biblioteca, tem-se que ela é uma boa opção também para desenvolvedores com menos experiência.

A pior métrica da biblioteca é realmente sobre seu uso em grandes projetos. Essa pergunta se deu por conta de *feedback* de um dos voluntários então tem-se uma ideia do porquê essa resistência existir. Esse dado é útil para desenvolvimentos futuros da biblioteca, pois ter essa informação em mente facilita o trabalho contínuo com a intenção de tentar melhorar esse aspecto.

Capítulo 5

Trabalhos futuros

Com a finalização desse projeto para fins de conclusão de curso e os *feedbacks* recebidos pela pesquisa realizada, restaram ainda algumas funcionalidades planejadas mas que, por conta de problemas inesperados e limitações de tempo, não conseguiram ser implementadas para a versão atual da biblioteca. Algumas dessas funcionalidades são:

5.1 Remoção da dependência de *RRD*

Como mencionado na Seção 3.3.5, no planejamento inicial da biblioteca o *RRD* seria usado como base para o desenvolvimento, e as implementações idealizadas em cima da lógica dele. O raciocínio por trás dessa decisão era de que, em *ragic*, a lógica de comunicação com o navegador para detectar qual a rota atual e com isso decidir qual o componente a ser usado, seria a mesma do *RRD*. As adições que foram idealizadas para a lógica de tipos funcionariam partindo desse ponto inicial.

Logo, faria sentido o uso de *RRD* como base em um momento inicial para trabalhar no ponto principal da biblioteca, e em um momento onde ela estivesse estável e funcional, seria possível remover o *RRD* como dependência e refazer a lógica de comunicação com o navegador.

Infelizmente por conta de algumas dificuldades durante o desenvolvimento, mudanças de lógica e descoberta de limitações não previstas, a finalização da biblioteca se deu em um momento mais tardio que o planejado, de forma que não seria possível remover essa dependência sem comprometer outras etapas do projeto, como a realização de testes e a escrita do relatório. Dessa forma, em comunicação com os orientadores, foi decidido apresentar o trabalho com essa dependência, e a remoção desta ficou como objetivo futuro após a finalização do trabalho de conclusão de curso, como projeto pessoal para a equipe de desenvolvedores e como passo necessário para lançar a biblioteca para o público geral.

5.2 Implementação de rotas apelido

Como mencionado na Seção 3.3.1.4, uma das ideias planejadas inicialmente para *ragic* é a possibilidade de, ao declarar uma rota, poder entregar à ela outros nomes-apelido os quais redirecionariam para o mesmo componente. Devido à mudanças na estrutura da lógica da biblioteca, essa ideia acabou sendo momentaneamente colocada de lado em favor das implementações mais essenciais. Por conta dos problemas mencionados anteriormente, no momento em que já se tinha uma versão funcional da biblioteca não era mais possível fazer adições e adaptações maiores sem comprometer o tempo separado para testes e escrita do relatório.

A ideia era de que fosse possível declarar uma rota da seguinte maneira:

```
1  import { UseRoutes, createRoutes, path } from "ragic";
2
3  const routes = createRoutes()
4    .path("/", { component: HomePage })
5    .path("/home", { link_to: "/" })
6    .path("/blog", { component: BlogPage })
7    .path("/news", { link_to: "/blog" })
8
```

Bloco de Código 9: Proposta de criação de rotas apelido.

Internamente, esse código é funcional. Ou seja, se um usuário tentar acessar a página com a rota `"/"` , ele ira visualizar o mesmo conteúdo que a rota `"/home"` , o componente `HomePage` .

O problema que impede o uso desse tipo de rota é que, em conflito com a proposta de oferecer uma *DX* melhor para o usuário, as rotas apelido não são reconhecidas pelo *autocomplete*, o que afeta a ideia da biblioteca de fazer uso inteligente de tipos para facilitar o processo de desenvolvimento.

Apesar de ser funcional, é um uso que não faz vantagem das principais implementações da bibliotecas, e é portanto um esforço maior com pouco retorno aproveitável.

Então é desejo da equipe de desenvolvimento de conseguir implementar essa funcionalidade na biblioteca em um futuro próximo.

5.3 Simplificar *API*

Apesar da *API* atual da biblioteca ser aceitável, acredita-se que existe ainda um nível de refinamento que pode ser feito para melhorar a *DX*.

Por exemplo, no comum caso de uma rota concreta simples sem filhos (ou seja, uma rota com um objeto de opções com somente uma entrada para `component`) exemplificado pelo Bloco de Código 10, seria interessante permitir que o objeto de opções seja omitido, inserido no lugar dele o componente da rota. O resultado dessa mudança na *API* permitiria algo como o Bloco de Código 11.

```
1 const routes = createRoutes()
2   .path("/home", { component: HomePage })
```

Bloco de Código 10: Exemplo da *API* atual para construção de uma rota concreta simples.

```
1 const routes = createRoutes()
2   .path("/home", HomePage)
```

Bloco de Código 11: Exemplo de código possível com alteração na *API*.

5.4 Adicionar documentação em inglês

Para o estágio de desenvolvimento inicial escolhemos documentar a biblioteca em português, por conta da natureza do público alvo dos usuários beta. Para o crescimento da biblioteca dentro do ecossistema, porém, é melhor usar inglês como língua primária para comunicação, organização e documentação. Mas é desejável manter disponível a documentação em português, pensando principalmente na acessibilidade da ferramenta.

Referências

[1] *Pontuação de desempenho do Lighthouse*. https://developer.chrome.com/docs/lighthouse/performance/performance-scoring?hl=pt-br, acesso em 2023-11-26. ix, 35

[2] *react-router-dom*, dezembro 2023. https://www.npmjs.com/package/react-router-dom, acesso em 2023-12-12. 2, 8

[3] *@reach/router*, junho 2020. https://www.npmjs.com/package/@reach/router, acesso em 2023-12-12. 2

[4] *wouter*, outubro 2023. https://www.npmjs.com/package/wouter, acesso em 2023-12-12. 2

[5] *react-navigation-helpers*, julho 2023. https://www.npmjs.com/package/react-navigation-helpers, acesso em 2023-12-12. 2

[6] *Link v6.20.1*. https://reactrouter.com/en/main/components/link, acesso em 2023-06-07. 2

[7] *JavaScript | MDN*, setembro 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript, acesso em 2023-12-15. 6

[8] *Dynamic typing - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*, junho 2023. https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing, acesso em 2023-12-15. 6

[9] Ecma, ECMA: *262: Ecmascript language specification*. ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,, 1999. 6

[10] *The starting point for learning TypeScript*. https://www.typescriptlang.org/docs/, acesso em 2023-05-10. 6

[11] *JSX*. https://facebook.github.io/jsx/, acesso em 2023-12-15. 7

[12] *tsx*, dezembro 2023. https://www.npmjs.com/package/tsx, acesso em 2023-12-16. 7

[13] *React Reference Overview – React*. https://react.dev/reference/react, acesso em 2023-12-15. 7

[14] Demedes, Vadim: *vadimdemedes/ink*, dezembro 2023. https://github.com/vadimdemedes/ink, acesso em 2023-12-15, original-date: 2017-06-12T06:12:28Z. 7

[15] *Built-in React Hooks – React*. https://react.dev/reference/react/hooks, acesso em 2023-12-15. 7

[16] Nielsen, Henrik, Roy T. Fielding e Tim Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.0*. Request for Comments RFC 1945, Internet Engineering Task Force, maio 1996. https://datatracker.ietf.org/doc/rfc1945, acesso em 2023-12-15, Num Pages: 60. 8

[17] *SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*, agosto 2023. https://developer.mozilla.org/en-US/docs/Glossary/SPA, acesso em 2023-12-15. 8

[18] *History API - Web APIs | MDN*, julho 2023. https://developer.mozilla.org/en-US/docs/Web/API/History_API, acesso em 2023-12-15. 8

[19] *Official page for Language Server Protocol*. https://microsoft.github.io/language-server-protocol/, acesso em 2023-12-14. 9

[20] *Intro to Nx*. https://nx.dev/getting-started/intro, acesso em 2023-04-08. 10

[21] *Create React App*. https://create-react-app.dev/, acesso em 2023-04-15. 11

[22] Gamma, Erich, Richard Helm, John Vlissides e Ralph Johnson: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. 17, 21

[23] *First Contentful Paint (FCP) | Articles*. https://web.dev/articles/fcp?hl=pt-br, acesso em 2023-11-29. 34

[24] Wills, Craig E., Gregory Trott e Mikhail Mikhailov: *Using bundles for Web content delivery*. Computer Networks, 42(6):797–817, agosto 2003, ISSN 13891286. https://linkinghub.elsevier.com/retrieve/pii/S1389128603002214, acesso em 2023-11-30. 35

[25] *IntelliSense in Visual Studio Code*. https://code.visualstudio.com/docs/editor/intellisense, acesso em 2023-04-15.

[26] *Langserver.org*. https://langserver.org/, acesso em 2023-11-15.

[27] Preston-Werner, Tom: *Semantic Versioning 2.0.0*. https://semver.org/, acesso em 2023-09-29.

[28] *HTML: HyperText Markup Language | MDN*, julho 2023. https://developer.mozilla.org/en-US/docs/Web/HTML, acesso em 2023-05-21.

[29] *State of JavaScript 2022: Front-end Frameworks*. https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/, acesso em 2023-12-15.

[30] Fox, Armando e David A. Patterson: *Engineering software as a service: an Agile approach using cloud computing*. Strawberry Canyon LLC, San Francisco, Calif, first edition, 1.2.2 edição, 2013, ISBN 978-0-9848812-4-6 978-0-9848812-3-9.