Progressive Web Apps

Construa aplicações progressivas com React

Casa do Código

GUILHERME PONTES

ISBN

Impresso e PDF: 978-85-94188-54-0

EPUB: 978-85-94188-55-7

MOBI: 978-85-94188-56-4

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

A Deus, pela minha vida. À minha querida esposa Laura, que está presente e me apoia em todas as minhas batalhas – inclusive nesta. Aos meus pais, que sempre me incentivaram a fazer o dever de casa antes de jogar bola.

Como se trata de um livro de desenvolvimento, também devo agradecer a: Brendan Eich, por parir o JavaScript; Jordan Walke (ou aos desenvolvedores do Facebook), por ter criado o React; Andrew Tanenbaum, por ter codificado o MINIX em seu livro de Sistemas Operacionais, que posteriormente serviu de alicerce para Linus Torvalds criar o kernel do Linux; e ao próprio Linus Torvalds (+1), pela invenção do Git.

É justo também regraciar o trabalho de Vivian Matsui, cujas orientações contribuíram amplamente para o resultado final desta obra. Agradeço também a excelente revisão técnica realizada pelo Rodrigo Ferreira, e toda a equipe da Casa do Código, por terem me dado a oportunidade de realizar este sonho.

E é claro, agradeço a **você**, por me acompanhar nesta aventura.

SOBRE O AUTOR

O currículo de um profissional de TI é sempre um tédio. Literalmente destacamos algumas dezenas de letrinhas embaralhadas em uma linha do tempo, e o espectador pensa: *"ok, ele tem o certificado Y, conhece o framework X, trabalhou na empresa Z"*. Tentei fazer algo diferente aqui, colocando tudo em uma perspectiva de jornada.

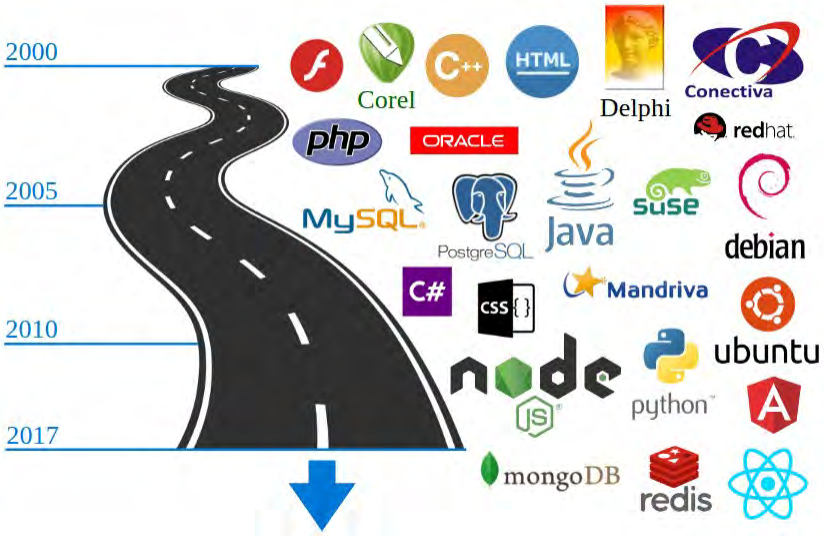

Figura 1: Jornada de aprendizado do autor

A disposição das imagens não está exatamente na ordem cronológica, mas representa bem minha linha de atuação. Apesar de ter experiência em infraestrutura, já faz alguns anos que trabalho com desenvolvimento de sistemas.

É claro, além disso, toco guitarra, já zerei quase todos os jogos

do Resident Evil, adoro livros do Bernard Cornwell e filmes do Star Wars. Trabalho no que gosto, amo minha esposa e tenho um cachorro chamado Luke.

PREFÁCIO

Neste momento, é muito importante alinharmos as expectativas: este livro trata da construção de uma aplicação progressiva com React, um framework JavaScript para front-end. A versão final da aplicação possui quatro telas. Como os conceitos e componentes do React são reaproveitáveis, não será necessário discursar minuciosamente sobre todas as telas. O objetivo é construir a primeira ressaltando todos os detalhes e conceitos necessários.

Entretanto, não vamos apenas construí-la. Vamos fundamentá-la e evoluí-la juntos, olhando de perto cada tecnologia adotada, destacando suas vantagens e desvantagens. No final, você terá um exemplo de *Progressive Web App* (ou simplesmente PWA) funcional e bem codificado.

Avançaremos gradativamente nos conceitos, cobrindo todo aparato primário ou acessório. Iniciaremos com uma introdução ao mundo das PWAs, a nova fronteira das aplicações web contemporâneas. Seguiremos com fundamentos dos frameworks **React**, **Pure.css** e do servidor **Nginx**. Por seguinte, faremos a configuração do ambiente e a codificação em si, destacando todos os passos da construção da aplicação.

De forma sucinta, veja na imagem a seguir os temas que serão abordados.

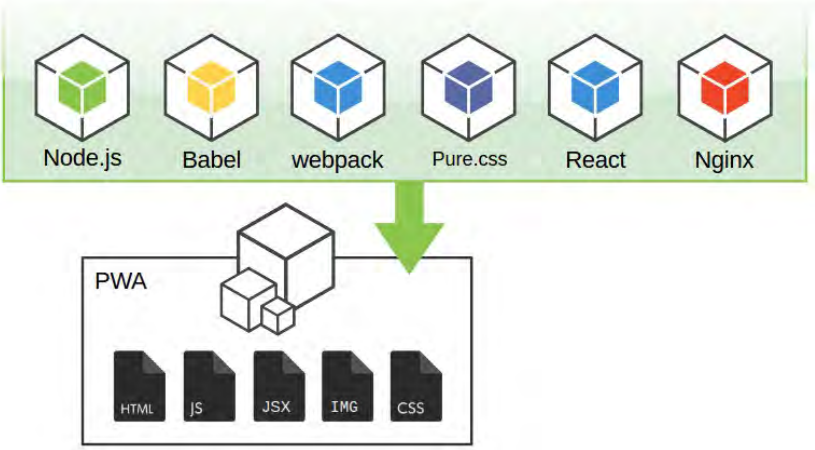

Figura 1: Temas tratados no livro

Sempre haverá um estudo minucioso dos frameworks. Abordaremos como implementá-los de forma coerente com as boas práticas de desenvolvimento de software. Na conclusão, estudaremos como publicar nossa aplicação em ambiente de produção e como validá-la, sempre segundo as boas práticas.

Caso esteja curioso com o resultado final, você tem duas alternativas:

1. Baixar o projeto completo no GitHub. O resultado final encontra-se no branch `master`, cujo endereço é: https://github.com/lgapontes/behappywith.me.
2. Acessar o endereço: https://behappywith.me.

A aplicação criada neste livro está publicada no endereço supracitado. Observação: a versão da URL encontra-se em inglês. O código em inglês é exatamente igual ao português e pode ser encontrado no branch `master-us`.

Você pode trocar ideias sobre este livro, compartilhar suas experiências e tirar dúvidas no nosso fórum: http://forum.casadocodigo.com.br/.

Caso queira me contatar pessoalmente, não hesite em acessar http://lgapontes.com/.

Público-alvo e pré-requisitos

Este livro é para desenvolvedores front-end com pouca experiência em HTML e JavaScript. Experiência em ECMAScript 6 (sexta edição do JavaScript) é desejável, mas eventualmente trataremos do assunto com mais profundidade quando a sintaxe exigir. Conhecimento prévio em CSS não será necessário, pois os tópicos de estilização abordados terão os devidos esclarecimentos.

As boas práticas de um projeto React implicam necessariamente no uso da sintaxe JSX, mas não é necessário nenhum conhecimento prévio sobre ela. Faz parte do escopo deste livro tratá-la de forma suficiente para que você possa exercitar os códigos apresentados.

Como haverá tópicos associados à programação orientada a objetos, é preciso algum conhecimento sobre esse paradigma, tais como classes, atributos, herança e objetos. Esse pré-requisito é perfeitamente sanado por conhecimentos básicos em linguagens orientadas a objetos, como Java ou C#.

O objetivo principal desta obra é construir uma aplicação progressiva. Nenhum conhecimento prévio sobre o que são ou como construí-las será necessário. Trataremos disso de forma preciosa, por assim dizer, atentando-nos a cada detalhe conceitual

ou técnico necessário.

Também faremos uso do Node.js, NPM, Webpack, Babel, Nginx, Pure.css e muitos outros frameworks e bibliotecas, acessórios à nossa aplicação. Se algum destes nomes lhe causar estranheza, não se preocupe! Falaremos sobre todos eles.

Sumário

1 Aplicações progressivas **1**

1.1 Requisitos de uma aplicação progressiva 2

1.2 Criando nosso backlog 19

1.3 Conclusão 31

2 Fundamentos sobre frameworks e ferramentas **33**

2.1 Fundamentos sobre o React 33

2.2 As principais engrenagens do React 45

2.3 IDE de desenvolvimento 61

2.4 Fundamentos sobre o Pure.css 62

2.5 Fundamentos sobre o Nginx 67

2.6 Conclusão 71

3 Configurando o ambiente de desenvolvimento **73**

3.1 React no Browser 74

3.2 Arquitetura React com Node.js 85

3.3 Node.js e NPM 87

3.4 Configuração manual do projeto 93

3.5 Criando o projeto com create-react-app 145

3.6 Conclusão 148

4 Primeiros passos do desenvolvimento 150

4.1 Definição dos componentes 151

4.2 Construindo o componente App 159

4.3 Criando o componente cabeçalho 167

4.4 Criando o componente do novo usuário 176

4.5 Criando o componente Label 185

4.6 Reajustando o componente NovoUsuario 195

4.7 Conclusão 196

5 Componentes com estado e fluxo de eventos 198

5.1 Estados dos componentes 199

5.2 Criando o componente Input 206

5.3 Criando o componente de seleção de gênero 222

5.4 Conclusão 257

6 Componentes complexos e domínio da aplicação 259

6.1 O domínio da aplicação 260

6.2 Criando o componente Button 264

6.3 Ajustando o componente NovoUsuario 276

6.4 Criando o componente ImageScroller 285

6.5 Ajustando os componentes App e NovoUsuario 331

6.6 Conclusão 341

7 Codificando os requisitos progressivos 343

7.1 Salvando os dados no navegador 344

7.2 Salvando arquivos no navegador 359

7.3 Requisitos com verificação manual 380

7.4 Requisitos com verificação automática 387
7.5 Conclusão 400

8 Publicando a aplicação em produção 402
8.1 Ambiente de produção 402
8.2 Relatório do Lighthouse 432
8.3 Conclusão 437

9 Referências bibliográficas 439

Capítulo 1

APLICAÇÕES PROGRESSIVAS

PWA é um acrônimo para *Progressive Web Apps*, ou Aplicações Web Progressivas. A palavra *progressiva* vem da ideia de *Progressive Enhancement*, ou melhoria progressiva. Neste contexto, a ideia por trás da palavra *progressiva* é condicionar uma aplicação a atender o maior número de pessoas possível, sob todos os aspectos.

Não é um estudo direcionado aos tipos de dispositivos, navegadores, versões das linguagens HTML, CSS ou JavaScript. Trata-se da criação de uma experiência cujos usuários terão acesso contínuo sem nenhum tipo de restrição tecnológica.

Podemos colocar da seguinte forma: desenvolver com melhoria progressiva é um paradigma em que a aplicação deverá estar disponível para todos, sejam usuários de microcomputadores ou smartphones, com browsers atualizados ou obsoletos, com conexão à internet ou não.

Sem internet? Isso mesmo. Uma aplicação progressiva deve estar disponível até mesmo quando o usuário estiver offline. Pode parecer um absurdo, mas a ideia de construir progressivamente

traz novos desafios ao desenvolvimento. Não podemos simplesmente ter um site *mobile-first* com design moderno. Ele precisa funcionar em qualquer dispositivo, sem conexão, para todo tipo de usuário, e continuar melhorando progressivamente até o infinito e além!

> **O QUE É MOBILE-FIRST?**
>
> O termo mobile-first é um conceito de desenvolvimento de software no qual o foco são os dispositivos mobile. Ou seja, ao se elaborar um site, cria-se toda a organização das páginas e a exibição dos dados com foco nos usuários de dispositivos mobile, mas sem se esquecer dos usuários de microcomputador. Como veremos em breve, atualmente existem muitos frameworks CSS para este tipo de desenvolvimento.

Tudo isso parece confuso no início, mas na verdade é bem simples. No próximo tópico, vamos levantar quais requisitos uma aplicação precisa respeitar para ser considerada uma verdadeira PWA.

1.1 REQUISITOS DE UMA APLICAÇÃO PROGRESSIVA

Faremos aqui um apanhado geral de todas as características que uma aplicação progressiva precisa atender. Existe uma ferramenta chamada Lighthouse que é capaz de gerar um relatório

completo das características de uma aplicação. Por enquanto, vamos entender cada item deste relatório com uma breve descrição de como atacá-lo. A programação envolvida na resolução dos itens será aprimorada nos próximos capítulos.

LIGHTHOUSE

Organizado e mantido pela equipe de desenvolvimento do Google, o Lighthouse é uma ferramenta de código aberto que permite qualificar aplicações web em relação aos requisitos de melhoria progressiva.

Figura 1.1: Lighthouse

Seu principal objetivo é gerar um relatório avaliando requisitos de performance, acessibilidade e o quão progressiva é uma aplicação. Mais adiante veremos como utilizá-la (extensão do Google Chrome) para medir a qualidade da aplicação construída neste livro.

Veremos agora, em tópicos, quais são os requisitos de uma aplicação progressiva.

Deve registrar um Service Worker

Service Worker é uma especificação W3C que permite executar um trecho de código JavaScript continuamente no nível do navegador. Por meio dele (não exclusivamente), é possível implementar recursos offline e organizar o cache dos arquivos estáticos da página web.

Um dos itens do relatório do Lighthouse é exatamente este: uso do Service Worker. Entretanto, podemos centrar-nos na ideia de que nossa aplicação deve funcionar offline, de preferência com os recursos oferecidos pelo Service Worker. No futuro, veremos as minúcias do uso e da implementação do Service Worker em detalhes.

Resposta com status 200 mesmo estando offline

Uma página web funciona sobre o protocolo HTTP, que, por sua vez, trabalha com vários status de retorno às solicitações dos browsers. O retorno de código 200 significa sucesso na solicitação. Uma aplicação progressiva deve simular tal status mesmo estando offline. Mas como isso é possível? Guardando os dados localmente.

Isso pode ser realizado via *Local Storage* (dados), *Application Cache* (arquivos), *IndexedDB* (dados), e o próprio Service Worker, que, além de nos ajudar com as respostas status 200, orquestra o cache dos arquivos. Mais uma vez, não se preocupe, veremos as vantagens e desvantagens de cada um em detalhes.

Deve exibir conteúdo quando o JavaScript estiver desabilitado

Permita-me contar uma rápida história: a muito tempo atrás (2008), trabalhei em um site de inscrições para um processo

seletivo. Nele, a principal informação do usuário era o CPF. Na página, havia um código JavaScript que acrescentava pontos no CPF.

Por uma falha no desenvolvimento, essa regra não estava replicada no back-end (PHP). Tudo corria bem até que um dos usuários, após ter se cadastrado com seu CPF devidamente pontuado, desabilitou o JavaScript e fez um segundo cadastro do mesmo CPF sem os pontos. Resultado: dentre todos os candidatos, esse camarada foi o único a ter o currículo avaliado duas vezes no processo. No final, descobrimos a farsa e o usuário foi desclassificado.

Qual o motivo dessa história? Uma PWA deve ter uma resposta programada para casos em que o JavaScript é desabilitado ou não suportado, mesmo que esta seja uma tela informando que não é possível usar a aplicação. Este requisito é bem simples de resolver, mas quase sempre é ignorado pelos desenvolvedores.

Usar HTTPS

Uma aplicação progressiva não pode ter seus dados transmitidos em claro. O protocolo HTTPS é obrigatório. No nosso caso, não haverá transmissão de dados entre o front-end e o back-end. A única responsabilidade do back-end será prover arquivos estáticos que serão armazenados localmente no navegador. Após isso, não haverá acesso remoto.

Isso nos permite descartar essa regra? Não! Acompanhe o raciocínio: não haverá tráfego de dados entre o navegador e o back-end **após** o primeiro carregamento da página pelo usuário. Como os arquivos estáticos ficarão armazenados localmente, a

partir do segundo acesso não haverá download de arquivos. Ok, mas e se no **primeiro** acesso houver um *man-in-the-middle* (literalmente, um homem no meio da comunicação) provendo arquivos falsos para, por exemplo, capturar seus dados e enviá-los para um servidor na nuvem?

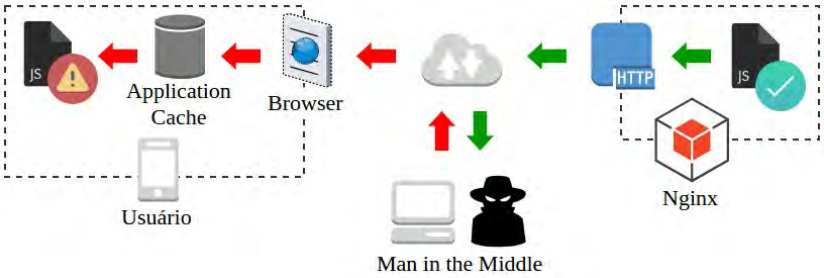

Figura 1.2: Man in the Middle

É nítido que estamos passíveis a tal problema. Não fomos os únicos a perceber isso. A própria especificação do Service Worker prega o uso obrigatório do HTTPS. No caso do Service Worker, é até mais grave, pois estaremos publicando um JavaScript de execução permanente no browser do usuário final.

Todo conteúdo HTTP deve ser redirecionado para HTTPS

Isso é muito simples. Por padrão, o navegador do usuário (mobile ou desktop) acessa o servidor web pelo protocolo HTTP, que trabalha sob a porta 80. Ou seja, todos os arquivos disponibilizados na porta 80 do servidor serão providos via HTTP. O acesso a um servidor sem criptografia é o caso mais comum, que ocorre quando um usuário apenas digita o endereço sem explicitar o protocolo.

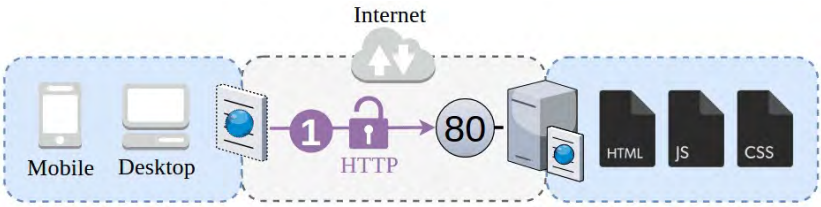

Figura 1.3: Acesso via HTTP

Outra possibilidade é acessar o servidor web pelo protocolo HTTPS, que nada mais é do que o próprio HTTP sobre o protocolo TLS (*Transport Layer Security*). Este é fornecido na porta 443 e basicamente estabelece um canal de comunicação criptografado entre o navegador e o servidor web. Por padrão, para acessar um servidor com esse recurso, o usuário precisará explicitar o uso do *https* antes do endereço do site (https://site-seguro.com).

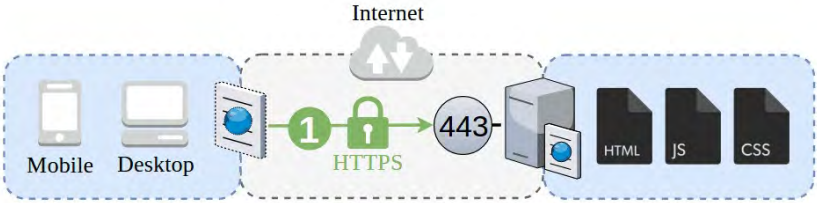

Figura 1.4: Acesso via HTTPS

Outra opção é configurar o servidor web para sempre redirecionar as solicitações recebidas na porta 80 para a porta 443, neste caso, obrigando o navegador a estabelecer um canal de comunicação seguro através do protocolo HTTPS. Se todas as mensagens em claro (sem segurança, via HTTP) forem redirecionadas para um canal criptografado (HTTPS), o usuário nunca poderá acessar a página de forma insegura. Vide uma

representação na figura adiante.

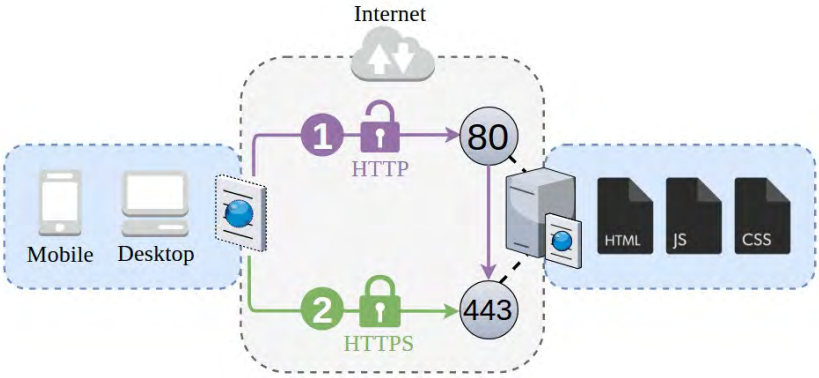

Figura 1.5: Conversão do HTTP para HTTPS

Vamos publicar nossa aplicação progressiva por meio de um servidor web chamado Nginx (cuja pronúncia é *enginex*). Ele possui vários recursos, inclusive o de redirecionar as mensagens HTTP da porta 80 para a 443, decorando-as com *Transport Layer Security* (TLS).

CONVERSANDO SOBRE SEGURANÇA (TLS)

Você já ouviu a expressão *"esta é uma terra de ninguém"*? Dizer que a internet encaixa-se nela não é uma blasfêmia! O tráfego de dados em claro (sem criptografia) nas redes contemporâneas é extremamente perigoso e deve ser desencorajado, principalmente quando se trata de dados confidenciais.

Foi pensando nisso que, no século passado, a equipe da Netscape inventou o protocolo *Secure Sockets Layer* (SSL),

cuja tradução pouco usual seria *Camada de Soquetes de Segurança*. O SSL é um protocolo de segurança que pode ser integrado a outros protocolos de comunicação, como se fosse uma camada extra aplicada para proteger um canal inseguro.

As primeiras versões do SSL já nasceram obsoletas, com vulnerabilidades de segurança que o impediram de ser utilizado. A partir de sua terceira versão, mundialmente conhecida como *SSL 3.0*, ele passou a ser usado como um recurso de segurança e foi padronizado pela entidade de padronização IETF através do documento RFC 6101.

Alguns anos depois, foi descoberta uma vulnerabilidade no SSL 3.0, na qual um *man-in-the-middle* poderia descobrir um byte criptografado monitorando algumas centenas de requisições. Isso tornou-o obsoleto, culminando em sua depreciação em 2015.

Em paralelo à descontinuidade do SSL 3.0, em 1999, a especificação RFC 2246 introduziu o algoritmo *Transport Layer Security* 1.0 (TLS), ou *Camada de Transporte Segura*, que se trata de uma evolução do próprio SSL. Atualmente, o padrão de mercado é o TLS 1.2, sendo que a versão 1.3 encontra-se em finalização.

Só como adendo, como o TLS foi baseado no SSL, é muito comum se referir à camada de segurança aplicada aos protocolos como *TLS/SSL* ou somente *SSL*, apesar de na prática o protocolo usado ser o TLS.

Com isso, resolveremos esta característica das PWAs. Veremos

com detalhes os benefícios e a configuração do Nginx no decorrer dos próximos capítulos.

Deve carregar rapidamente no 3G

Apesar do título estranho do Lighthouse – que estou transcrevendo *ipsis litteris* –, significa basicamente que a aplicação deve estar disponível para interação do usuário em um tempo inferior a 10 segundos. Seremos mais agressivos! Há estudos que indicam que uma aplicação que demora mais do que 3 segundos já é considerada lenta pelos usuários. Vamos estabelecer a seguinte meta: *three seconds and go* (três segundos e vai)!

O usuário deve ser questionado se deseja instalar a aplicação

As atuais especificações da W3C favorecem muito tal possibilidade, principalmente quando se trata de usuários mobile. Veremos que, para isso funcionar, além das configurações de instalação por dispositivo, teremos de organizar ícones em tamanhos específicos para cada caso. Quando estiver funcionando, haverá um ícone na área de trabalho para acessar a página, e o usuário poderá utilizá-la como se fosse uma app nativa.

Deve ser configurada com uma splash screen personalizada

O termo *splash screen* (tela de abertura) tem origem nos aplicativos desktop, em que uma pequena tela de inicialização com o logotipo é exibida enquanto o software é carregado. Veja adiante um exemplo da splash screen do GIMP, um software de edição de

imagens.

Figura 1.6: Software de manipulação de imagens

Isso também existe no mundo mobile. A ideia é simplesmente exibir uma página intermediária enquanto a aplicação é carregada.

Colorir a barra de endereço do navegador com as cores do site

Este é um item de menor relevância, mas estou destacando-o por ser uma das verificações realizadas pelo Lighthouse. Basicamente, deve-se colorir a barra de endereço com cor semelhante à do site. Resolveremos isso com uma estilização básica da página, no futuro.

Implementar a metatag viewport

A tag `<meta>` de nome `viewport` melhora a visualização da página nos dispositivos mobile. Viewport é a área visível do

usuário em página da web. Quanto menor a tela do dispositivo, menor será a viewport.

Obviamente ela será menor em um telefone celular do que no browser de um microcomputador. Veremos no decorrer do livro as melhores práticas para se construir telas responsivas e com os recursos adequados aos dispositivos mobile.

Para este item em especial, simplesmente devemos colocar o trecho de código a seguir no cabeçalho da página.

```
<meta name="viewport" content="width=device-width, initial-scale=
1">
```

Isso será feito na elaboração da tela do sistema, no *Capítulo 4*.

Redimensionar o conteúdo da página corretamente

Há duas formas de se construir uma aplicação compatível com diferentes tamanhos de viewport: layouts estáticos (organizados especificamente para os dispositivos), ou layouts responsivos.

A organização estática, no geral, oferece resultados de alta qualidade, mas possui baixa manutenibilidade. A ideia para se construir layouts estáticos passa primeiramente pela descoberta do dispositivo utilizado pelo usuário.

O protocolo HTTP possui um cabeçalho com várias informações. Uma delas é a `User-Agent`, que contém características do navegador e do sistema operacional do usuário, entre outras minúcias.

Por meio dessa propriedade, podemos descobrir se o usuário está utilizando um celular, um tablet ou um computador e, em

seguida, direcioná-lo para uma URL apropriada ao dispositivo. É muito comum o desenvolvimento de páginas próprias para smartphones em URLs com o prefixo *mobile* ou *m*.

Por exemplo, o site fictício http://sitequalquer.com pode ter uma versão mobile disponível em http://m.sitequalquer.com. Obviamente, isso significa que precisaríamos realizar manutenção em dois sites, fato que muitas vezes inviabiliza esse tipo de proposta.

Mas cuidado: no caso de criarmos um layout estático sem a lógica do redirecionamento (pelo `User-Agent`), o usuário enfrentará sérios problemas de usabilidade ao abrir uma página em um dispositivo inadequado. Não podemos nos esquecer que haverá microcomputadores bisbilhotando a web vez ou outra.

Então, vamos construir páginas por dispositivo e sofrer com a manutenção? Não. Por sorte, temos a outra alternativa: páginas com layout responsivo. E é justamente assim que vamos resolver este requisito.

As boas práticas apontam para a construção de páginas que se auto-organizam de acordo com o tamanho da viewport, escondendo elementos inapropriados, substituindo componentes, e ajustando o tamanho de imagens e fontes. Isso tudo é possível com a disposição correta das tags HTML, uma boa organização das classes CSS e do uso de *@media queries*.

Uma *@media queries* é um trecho do CSS que permite alterar o estilo de acordo com o tamanho da viewport. Por exemplo, um menu com várias opções no cabeçalho da página poderia ser substituído por um *hamburger* se visualizado em um smartphone.

Figura 1.7: Hamburger menu do Materialize.css

Veja anteriormente um suposto exemplo do uso da *@media queries*. Os componentes apresentados foram extraídos do Materialize.css, um framework CSS aderente ao *Material Design*, especificação de design criada pelo Google. Citei-o apenas para exemplificar um caso real de *@media queries*.

Nesta obra, vamos utilizar o Pure.css em vez do Materialize.css. Aliás, reforçando: Pure.css serve especificamente para auxiliar na estilização (CSS) da nossa aplicação progressiva com recursos de layout responsivo, obviamente.

Então, concluindo: layout responsivo na cabeça!

O site deve ser cross-browser

Para atender o maior número de usuários possíveis, devemos nos esforçar para tornar nosso código compatível com vários navegadores. Por sorte nossa, a maioria dos frameworks CSS atuais já resolvem esse problema. Entretanto, se utilizarmos um recurso restrito para os browsers mais modernos, podemos criar uma página do tipo "Desculpe-nos, mas seu browser não é compatível

com nosso site!". O Pure.css possui versões distintas para uso em browsers antigos.

Apesar de este requisito estar explícito no relatório do Lighthouse, não há um mecanismo automático de validação para ele. Para contornar esta situação, a documentação oficial de aplicações progressivas mantida pelo Google (https://developers.google.com/web/progressive-web-apps/checklist#site-works-cross-browser), que está intrinsecamente associada ao Lighthouse, mantém regras de avaliação manual para avaliarmos casos como este. No futuro, usaremos tais regras para validar esse requisito.

As transições entre páginas não devem ser sensíveis à velocidade de conexão

Esta é uma boa oportunidade para introduzirmos os conceitos associados à disposição das tags no navegador. Os elementos HTML são organizados através do DOM, cujo acrônimo original em inglês é *Document Object Model* (Modelo de Objeto de Documento). Basicamente, o DOM é uma estrutura de dados que organiza os elementos HTML em uma árvore. Cada nó desta árvore, além de manter informações sobre sua posição na estrutura, contém atributos e métodos para usufruto do navegador e de códigos JavaScript agregados à aplicação.

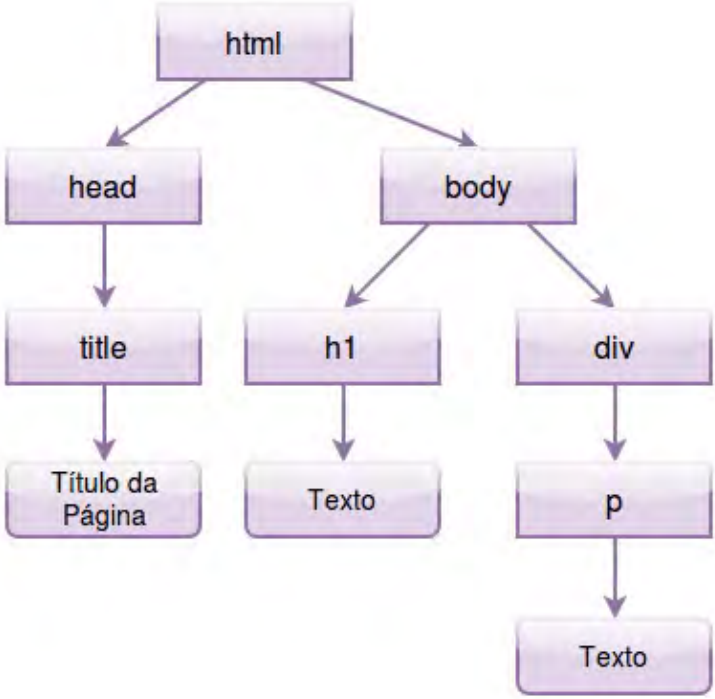

Figura 1.8: Representação do DOM

Mas nem só do DOM viverá o navegador. Para as coisas funcionarem, ele precisará baixar todos os arquivos associados ao HTML: arquivos de estilo (CSS), arquivos JavaScript, imagens etc. É sabido que, para fazer tudo isso, o navegador precisará de tempo, tanto no primeiro acesso quanto na transição entre as páginas. Este requisito das PWAs preocupa-se justamente em monitorar o período de transição.

Caso haja uma transição total, o que é desaconselhado, teremos de nos preocupar em torná-la rápida o suficiente para que o usuário não fique irritado. Neste contexto, *transição total* significa

que o navegador precisará baixar tudo (o HTML, o JavaScript, as imagens, o CSS) e ainda renderizar o DOM, **sempre** que o usuário acessar uma das páginas do sistema.

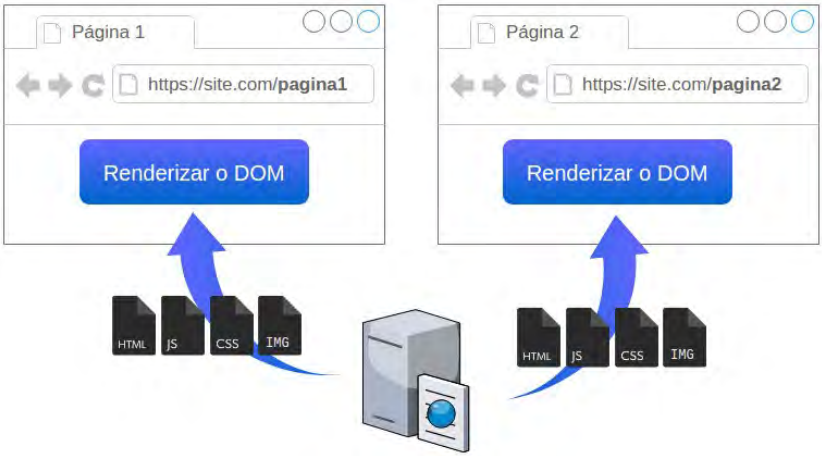

Figura 1.9: Transição total entre as páginas

Mesmo sem nos aprofundarmos, já é possível perceber o quanto essa abordagem é frágil. Veremos mais detalhes no próximo capítulo, mas só a renderização do DOM já é uma atividade que pode comprometer a performance no lado cliente (navegador). Além disso, se para cada página da PWA ainda precisássemos baixar todos os arquivos, certamente teríamos problemas de *lentidão*.

Para resolver, podemos simplesmente evitar transições completas por meio da construção de uma *Single-Page Application* (SPA), ou aplicação de única página. Nas SPAs, as transações entre as páginas são fluentes, sem a necessidade de refazer o download dos arquivos e renderizar totalmente do DOM.

Mas afinal, o que é uma SPA? Uma aplicação de única página não significa literalmente que ela **deve** ter uma única página. Ela pode ter quantas páginas precisarmos. A diferença está no que ocorre na transição: as páginas não precisam ser recarregadas. Veja o fluxo a seguir.

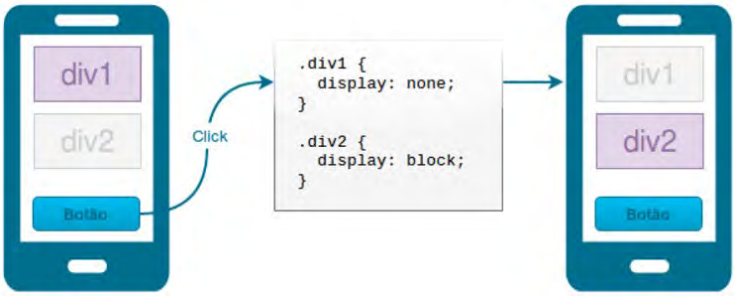

Figura 1.10: Single Page Application

Na representação do dispositivo à esquerda, note que temos duas tags `<div>` : a `<div>` 1 sendo exibida e a `<div>` 2, escondida. Suponha que cada `<div>` seja uma página completa, com vários componentes. O elemento *botão* seria o gatilho de transição entre elas. Ao ser clicado, além de obter os dados do servidor (ou do cache local), ele vai esconder a `<div>` 1 e apresentar a `<div>` 2.

Este foi um exemplo muito simplório do comportamento padrão de uma SPA, contudo, servirá como base para entendermos a ideia principal: a transição entre as páginas deve ser transparente aos usuários, sem o `reload` ou `redirect` completo da página. Como isso também implica em **não** baixar todos os arquivos novamente, teremos maior perspectiva de performance em nossa aplicação.

O Lighthouse não possui mecanismo automático para avaliar este requisito. Sua validação também se dará a partir de verificações manuais que faremos no *Capítulo 7*.

Cada página deve ter uma URL

O termo técnico para essa característica é *bookmarkable* (sem tradução literal). Na prática, uma página bookmarkable é aquela que pode ser acessada por uma URL única - ou seja, pode ser adicionada aos favoritos do navegador. Além de ter uma URL específica, é importante defini-la com um nome amigável ao usuário final.

Por exemplo, um blog sobre 10 curiosidades de cães africanos poderia ter a seguinte URL: https://africageographic.com/blog/10-interesting-facts-african-wild-dogs/

Em aplicações não-SPA, cujas transições de páginas são feitas inteiramente, esse fator vem praticamente de graça – é necessário apenas definir boas URLs. Em uma SPA, todavia, como o usuário fará a transição entre as views (páginas) sem recarregar, há maior complexidade em resolver este requisito.

Veremos algumas práticas para tratá-lo com o *React Router*, um framework complementar ao React para organizar as rotas da aplicação. Este requisito também precisará de uma validação manual – que será realizada mais adiante.

1.2 CRIANDO NOSSO BACKLOG

A seção anterior apresentou os principais itens de avaliação fornecidos pelo Lighthouse. Estes não são os únicos. Há também

outras características de performance, boas práticas, acessibilidade e usabilidade. Não é escopo deste livro conceituar todas. Daremos foco nos que são associados às PWAs.

Todavia, há uma série de outros requisitos inerentes à página da aplicação que vamos construir. Vamos entendê-los e agrupá-los por afinidade a seguir. Para materializar uma ideia mais próxima do resultado final deste livro, vamos também organizar tudo em épicos, acompanhados de um protótipo de baixa fidelidade quando necessário.

O QUE SÃO ÉPICOS E HISTÓRIAS?

Essa terminologia (épicos, histórias etc.) advém das metodologias ágeis e é fortemente adotada pelo Scrum. Uma história pode ser considerada como um requisito e um épico como um grupo de requisitos.

Scrum é um framework de gerenciamento de projetos organizados em períodos fixos (Sprints) com um objetivo claro para toda equipe participante. Ele foca em reuniões de planejamento, nas quais itens do *Backlog* (lista de requisitos) são priorizados pelo dono do projeto e incluídos na Sprint. O Scrum não formaliza uma notação específica, porém, em grande parte dos projetos, é comum o uso de *Histórias* e *Épicos*.

Apesar de a grande vantagem do Scrum estar associada a projetos que beiram o limite do caos (requisitos e complexidades incertos), podemos usá-lo plenamente em projetos com escopo conhecido. Neste livro, organizaremos as entregas em Épicos e Histórias, que serão resolvidos pontualmente nos capítulos relacionados.

Quer conhecer mais sobre o assunto? Leia o livro *Scrum: Gestão ágil para projetos de sucesso*, de Rafael Sabbagh (https://www.casadocodigo.com.br/products/livro-scrum).

Histórias da PWA

Começaremos pela PWA. Como são requisitos técnicos

associados ao sistema como um todo, não haverá protótipo.

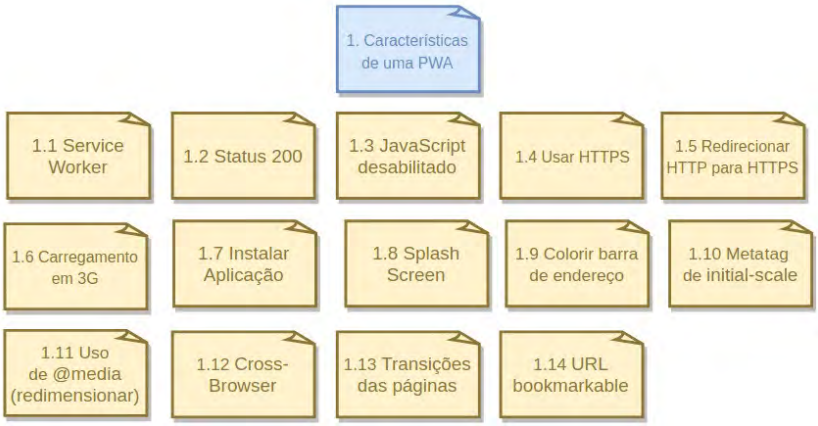

Figura 1.11: Requisitos de uma PWA

Apenas transcrevemos o estudo da seção anterior em um quadro de histórias. Isso nos dá uma ideia da dimensão que precisaremos atacar nos próximos capítulos.

Visão geral da aplicação

Nossa aplicação será chamada de **Be happy with me** (https://behappywith.me) e terá como principal objetivo organizar e agendar *gentilezas* do usuário.

Figura 1.12: Logotipo

Vamos entendê-la: um usuário faz uma gentileza com uma outra pessoa. Que tipo de gentileza? Um abraço, um aperto de

mãos, um bom-dia, enfim, qualquer tipo de gentileza. Em seguida, com poucos cliques em seu smartphone, ele poderá registrar a gentileza na aplicação. Ao registrá-la, ele automaticamente ganhará uma quantidade específica de experiência (XP) de acordo com o tipo da gentileza.

A aplicação deve eventualmente gerar uma gentileza aleatória. Essa gentileza ficará como *pendente* até o usuário realizá-la ou descartá-la. A ideia por trás dessa regra é fomentar novas gentilezas para que o usuário torne-se uma pessoa mais gentil. Todas as gentilezas, realizadas ou agendadas, serão exibidas na página principal da aplicação.

Essa foi uma descrição bem abrangente. Não precisamos (e nem queremos) perder muito tempo com isso. Veja graficamente um resumo de todas as funcionalidades disponíveis na aplicação.

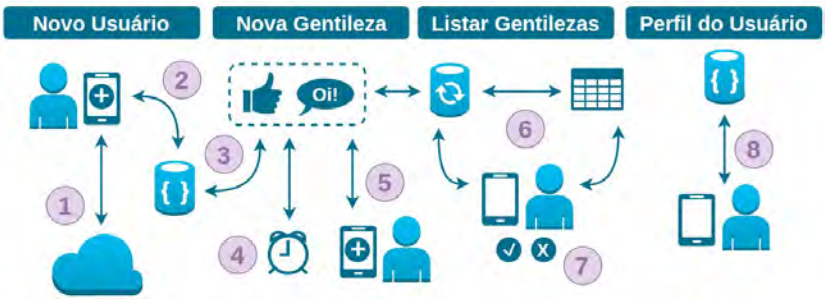

Figura 1.13: Ações da aplicação

Como você verá no decorrer dos próximos capítulos, os componentes criados pelo React são altamente reaproveitáveis. Tudo o que será criado nos requisitos associados à tela de *Novo Usuário* (primeira seção da imagem anterior) será reutilizado, o que torna o trabalho de desenvolvimento das telas *Nova Gentileza*,

Listar Gentilezas e *Perfil do Usuário* demasiadamente repetitivo.

O objetivo aqui é fazê-lo compreender toda a tecnologia necessária para construir aplicações progressivas com auxílio do framework React. Com o desenvolvimento de *Novo Usuário* concluído, além de estar apto à construção das demais as telas, você poderá construir sua própria aplicação progressiva.

Dito isso, fica claro que só será necessário descrever em detalhes o funcionamento da tela *Novo Usuário*. As demais serão apenas citadas brevemente para que você entenda a aplicação como um todo, mas sem ênfase nos detalhes técnicos necessários para o desenvolvimento.

Vejamos agora o protótipo de *Novo Usuário*. Fique atento aos números apresentados na imagem anterior; eles serão referenciados adiante.

Novo usuário

Na primeira vez que um usuário acessar a aplicação, todos os arquivos estáticos serão baixados do servidor HTTPS (Nginx) para o dispositivo. Isso está representado no número 1 da figura a seguir.

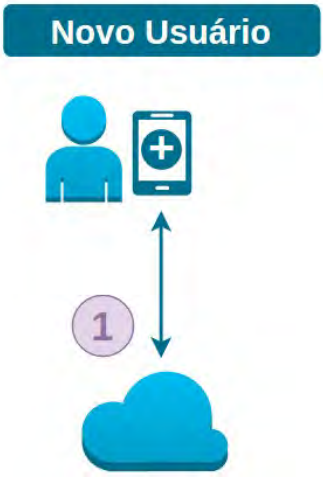

Figura 1.14: Carregar página

Todos os dados do usuário e das gentilezas serão salvos localmente, no cache do navegador. Veremos a parte técnica envolvida em detalhes no futuro. Por enquanto, só precisamos fixar a lógica: se a aplicação **não** encontrar os dados do usuário, a tela de cadastro será apresentada. Ao terminar o cadastro, a aplicação salvará os dados no cache do navegador, fato que permitirá o uso das demais telas.

O item 2 da imagem a seguir representa ambos os passos: salvar e obter os dados do usuário do cache do navegador.

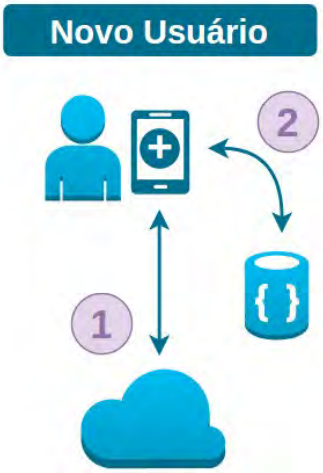

Figura 1.15: Salvar e consultar dados no cache local

Veja na figura a seguir o protótipo de baixa fidelidade da tela de novo usuário.

Figura 1.16: Protótipo da tela de novo usuário

Os componentes presentes nesta tela formarão nossa *suíte* de componentes para toda a aplicação. Nos próximos capítulos, veremos em detalhes a construção de cada um deles, obedecendo as melhoras práticas de aplicações progressivas e do framework React. Tudo isso será realizado a partir do protótipo.

Essa tela será considerada um épico. Não vamos desdobrá-lo em históricas agora, pois isso será feito nos capítulos associados à sua construção.

Demais funcionalidades

Apenas reforçando, as demais funcionalidades reutilizarão os componentes criados na tela de *Novo Usuário*, não sendo

necessário para nós, sob o ponto de vista de aprendizado, reproduzir uma codificação que não acrescentará novos conceitos. Sob o ponto de vista do entendimento do projeto **Be happy with me** como um todo, entretanto, é interessante que você compreenda seu funcionamento. Só como adendo, o código completo do projeto está disponível no repositório Git, cujo link é https://github.com/lgapontes/behappywith.me. Opcionalmente, você poderá estudá-las por lá, se quiser.

Veja na imagem a seguir os requisitos da tela de *Nova Gentileza*.

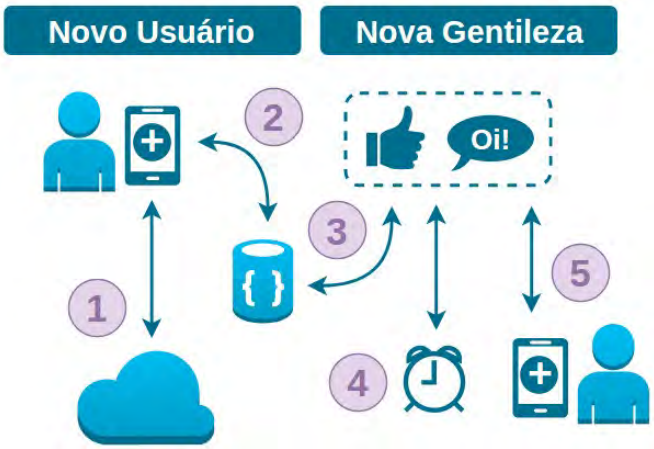

Figura 1.17: Nova Gentileza

Após salvar os dados do usuário, a aplicação criará automaticamente a primeira gentileza *agendada*. Isso está representado no número 3. Chamamos de *agendada* pois o usuário terá 24 horas para realizá-la ou descartá-la.

Diariamente, a aplicação criará gentilezas agendadas (número

4). Por exemplo, em determinado dia pode surgir uma gentileza **Dar um doce a um amigo**. Neste exemplo, *amigo* é o **destinatário** da gentileza. Toda gentileza terá um **destinatário**, que nada mais é do que o seu receptor. Teremos seis tipos de destinatários.

Figura 1.18: Destinatários

Mas afinal, o que seria uma gentileza? Trata-se de um registro no tempo de uma ação realizada com alguém. Na terminologia da aplicação, a ação será conhecida como **tipo de gentileza**. O sistema terá sete tipos de gentileza, e haverá uma imagem representativa para cada uma delas, conforme a figura a seguir.

Figura 1.19: Tipos de gentileza

O último requisito relacionado à criação das gentilezas é o fluxo manual, indicado pelo número 5. Por ele, o usuário poderá registrar uma gentileza *realizada*. Este tipo já nasce resolvida.

Dando sequência, veremos agora os passos 6 e 7, relacionados à tela *Listar Gentilezas*.

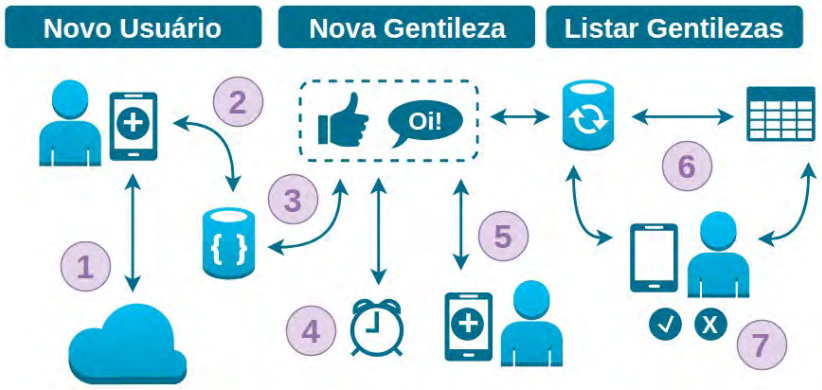

Figura 1.20: Listar gentilezas

A lista de gentilezas exibirá as gentilezas, agendadas ou realizadas, ordenadas da mais recente para a mais antiga. Isso está representado no número 6. Gentilezas criadas pelo usuário já estarão automaticamente resolvidas. As demais vão oferecer botões de *confirmação* ou *descarte* na lista de gentilezas, ambos representados pelo número 7.

A última tela trata da exibição consolidada do perfil do usuário da aplicação, representada pelo número 8 da imagem a seguir.

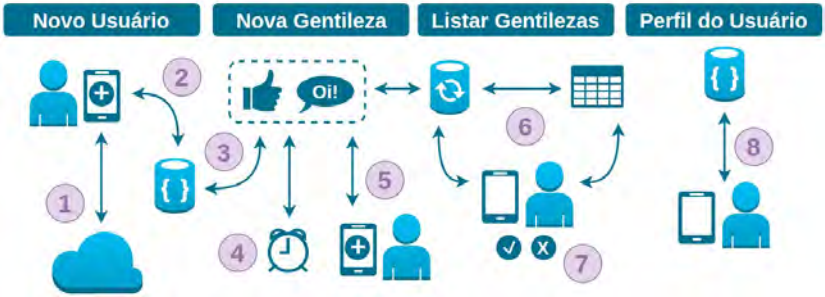

Figura 1.21: Perfil do usuário

Esta tela exibe o nome, o avatar e a experiência do usuário. Como resultado consolidado, são exibidos o total de gentilezas, o total de atrasos, o total de gentilezas descartadas, e a média de gentilezas por dia.

1.3 CONCLUSÃO

Neste capítulo, introduzimos os fundamentos das aplicações progressivas. Alinhamos também a expectativa sobre quais requisitos técnicos uma PWA deve atender, e o porquê de implementá-los. Ratificamos a proposta do livro, que é construir

parte de uma aplicação progressiva usando como arcabouço o framework de front-end React. Organizamos os requisitos da nossa aplicação em épicos e, no caso especial das características de uma PWA, em histórias.

Nos próximos capítulos, avançaremos passo a passo na resolução do backlog, sempre oferecendo uma fundamentação necessária para o uso e a escolha das ferramentas e frameworks. É sabido também que, eventualmente, algumas tecnologias (como o ECMAScript 6, por exemplo) terão de ser introduzidas como alicerce. Isso será feito gradativamente, sempre que necessário.

Capítulo 2

FUNDAMENTOS SOBRE FRAMEWORKS E FERRAMENTAS

Prepare-se: este é um capítulo teórico em que fundamentamos a escolha do React, suas principais características, vantagens e desvantagens. Destacamos também a escolha da IDE, do framework de estilização Pure.css e do servidor web Nginx.

Isso pode parecer desnecessário, mas a verdade é que o ecossistema de ferramentas para suportar aplicações progressivas é imenso. É muitíssimo importante conhecermos as ferramentas envolvidas em nosso projeto para criarmos uma base de conhecimento sólida para os próximos capítulos.

2.1 FUNDAMENTOS SOBRE O REACT

Você já sabe que a escolha de nosso framework da camada cliente é o React. Vamos entender a partir de agora o motivo desta decisão. Não se engane, escolher um framework front-end é uma tarefa árdua! Existem ótimas opções no mercado e todas merecem respeito, cada uma com seu *approach* específico e suas vantagens bem ressaltadas. No final, tudo depende do tipo de projeto.

Ok, se depende do projeto, então vamos destacar alguns pontos importantes sobre o nosso escopo para embasar a decisão:

1. **Trata-se de uma aplicação web progressiva:** fato bastante discutido no *Capítulo 1*.
2. **Trata-se de uma Single Page Application (SPA):** a usabilidade e a performance esperadas de uma PWA praticamente exigem SPAs. Digo *usabilidade* porque o uso fica mais agradável, e *performance* porque o tráfego do HTML, imagens, CSS, JavaScript e a montagem da estrutura da página são substituídos pelo envio e recebimento assíncrono dos JSONs.
3. **O design deve ser mobile-first, mas com compatibilidade no mínimo aceitável em computadores:** aplicações progressivas devem atender a todos os usuários, mesmo que a essência gire em torno do público móvel.
4. **Deve ter um ciclo de vida pequeno mas com boa manutenibilidade:** o ciclo de construção de software geralmente é longo, caro e demasiadamente desgastante à equipe e ao *owner* (dono) do projeto. Estabelecer objetivos claros e metas intermediárias nos permite trabalhar com ciclos menores e entrega de valor a curto prazo. Considero isso como um mantra pessoal, mas caso queira se aprofundar no assunto, dê uma olhada na obra *Guia da Startup: Como startups e empresas estabelecidas podem criar produtos de software rentáveis*, de Joaquim Torres, publicado na Casa do Código (https://www.casadocodigo.com.br/pages/sumario-startup-guia).
5. **Forks serão bem-vindos no GitHub:** projetos como esse podem se tornar fabulosos com auxílio da comunidade. Os colaboradores, porém, precisam estar confortáveis com a

tecnologia utilizada. Apontar para a direção contrária do que se usa no mercado certamente será prejudicial. Aliás, esta é uma realidade em qualquer tipo de projeto.

Você pode achar que um ou outro item transcende nosso escopo, mas na verdade os três primeiros são necessários às aplicações progressivas, enquanto os dois últimos são boas práticas para todo projeto e podem gerar continuidade a um trabalho de caráter *open-source*.

Vamos agora analisar as características do framework, destacando os pontos positivos e negativos sempre que necessário.

Estrelas no GitHub

Obviamente, no momento em que você estiver lendo este livro, os números serão outros, mas já podemos notar uma tendência.

Figura 2.1: Estrelas no GitHub

Escolher um framework *da moda* tem vantagens e desvantagens. A falsa maturidade que um framework pode apresentar às vezes é ofuscada pelo modismo. Por outro lado, em projetos de ciclo de vida pequeno, o fato de mais pessoas curtirem a tecnologia geralmente representa maior número de desenvolvedores, mais fóruns, mais bugs corrigidos, mais componentes prontos etc. E como uma suposta continuidade com ajuda da comunidade é sempre bem-vinda, não nos resta dúvidas aqui.

Tamanho dos arquivos

Pode parecer ridículo preocupar-nos com meros kbytes, mas a verdade é que isso faz toda a diferença no carregamento das páginas. Imagens relativamente grandes (por volta dos 300kb) são aceitáveis, mas arquivos JavaScript de 900kb são imperdoáveis.

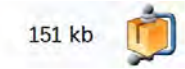

Figura 2.2: Tamanho dos arquivos

Adianto que, por estarmos criando uma PWA, o JavaScript baixado ficará no *Application Cache* do navegador. Isso significa que apenas o primeiro acesso será influenciado pelo tamanho. Ok, mas você se lembra da meta *three seconds and go*? Desejamos que a página esteja disponível em, no máximo, 3 segundos, e isso inclui o primeiro acesso.

Só uma observação: há meios de reduzir drasticamente o tamanho dos arquivos React. Em outras palavras, se fizermos um *tuning* (ajuste fino), os arquivos do React serão menores.

Linguagem

Por *linguagem*, entenda como a principal sintaxe utilizada para criar os componentes.

Figura 2.3: Linguagem JSX

Como veremos ao longo do livro, o React fornece boas APIs para quaisquer versões do JavaScript. Versões do JavaScript? Sim,

existem algumas, e acho que é um bom momento para alinharmos o assunto.

O JavaScript surgiu nos primórdios da internet, a partir de uma linguagem chamada LiveScript criada pela Netscape (empresa). Ela começou a ganhar força após sua submissão à *Ecma International*, uma organização de padrões para sistemas de comunicação e informação, que passou a chamá-la de ECMAScript.

O avanço entre as versões ECMAScript 5, que atualmente é compatível com a grande maioria dos navegadores, e 6 (ou ECMAScript 2015) é o foco da nossa atenção aqui. Quando digo que o React fornece APIs para todas as versões, na verdade, refiro-me às versões ECMAScript 5 e 6. Além do ECMAScript 6, o React também oferece a poderosa sintaxe JSX.

Segundo a definição oficial, JSX é uma extensão do XML para o ECMAScript, cujo acrônimo significa *JavaScript XML*. Por enquanto, não há intenção de ele se tornar compatível com os navegadores ou ser incorporado à especificação ECMAScript. Seu uso fica condicionado a compiladores que transformam o código JSX em ECMAScript compatível com os browsers. Um pouco adiante, ainda neste capítulo, faremos uma introdução à sintaxe.

Manipulação do DOM

No capítulo anterior, fizemos uma introdução sobre o que é o DOM. A renderização dos componentes React é realizada por meio de um recurso conhecido como Virtual DOM, de performance muito superior à manipulação do DOM convencional dos navegadores. Há um estudo profundo sobre as vantagens do Virtual DOM ainda neste capítulo, em uma seção

mais à frente.

Mas por que exatamente é ruim usar o DOM nativo? Porque manipular o DOM não é performático. Tratando-se de uma PWA cuja execução é totalmente influenciada pela velocidade com que o framework é executado no navegador, considerar o tempo de renderização é muito importante.

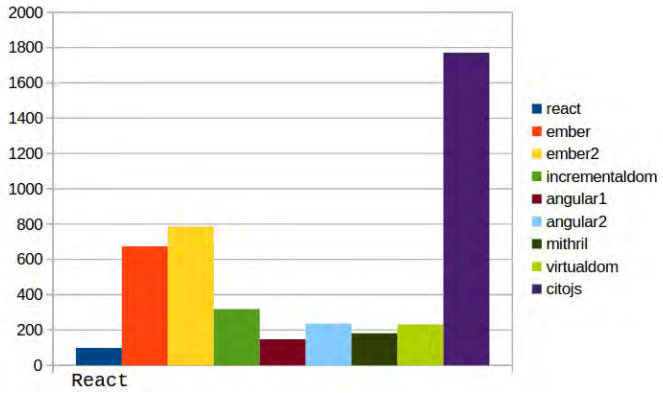

Figura 2.4: Tempo de renderização dos frameworks

A imagem anterior apresenta uma comparação entre os principais frameworks JavaScript do mercado. O gráfico de barras representa quantos milissegundos são necessários para o browser redesenhar as páginas. O React se destaca nitidamente.

Essa pesquisa foi realizada pelo site *auth0.com* (PEYROTT, 2016). Optei em divulgar apenas o tempo de renderização, mas o site faz várias comparações de performance entre os frameworks, caso queira mais detalhes.

Tempo de aprendizado

Dizer que uma tecnologia é mais fácil de aprender que a outra não é tão simples. Faça o seguinte: veja no YouTube qualquer curso para iniciantes React. Você encontrará vídeos de 1 a 3 horas, em média. Realmente acho pouco tempo para iniciar com o desenvolvimento de um framework. Certamente, cursos mais avançados demandarão muitas horas, o que é comum em qualquer framework de mercado.

Transição dos dados entre o DOM e os componentes

O termo *two-way binding* (ligação bidirecional) muito difundido no framework Angular significa que há uma ligação de duas vias entre o DOM e os objetos no JavaScript. O React não oferece esse recurso. Ele até possui formas para contornar a situação, mas na prática é *one-way binding*.

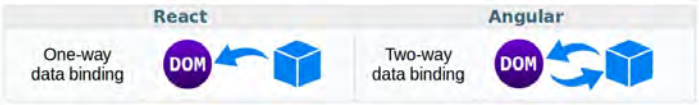

Figura 2.5: Transição dos dados

Mas o que realmente representa o termo *binding*? Existem dois lados da força: um deles é o DOM (tags HTML da página); o outro é composto pelos objetos JavaScript. Quando a ligação acontece em ambos, sempre que um dos lados for alterado, o outro receberá instantaneamente a atualização.

Ou seja, se o usuário alterar um elemento `<input>` na página, o objeto JavaScript vinculado será atualizado. De forma semelhante, se o objeto for atualizado, o valor do `<input>` também será.

O *two-way binding*, apesar de ser muito prático, possui duas desvantagens:

1. Performance;
2. Sintaxe HTML poluída com os mecanismos de binding.

O React só oferece uma direção do binding. O estado e as propriedades do componente (objeto JavaScript) são copiados para o DOM pelo método `render()`. O retorno do dado, entretanto, é realizado por meio da captura de eventos (como o `onChange`, por exemplo). Com base nisso, é correto dizer que o retorno dos dados do DOM para o JavaScript é indireto.

Conforme veremos no decorrer dos próximos capítulos, isso não é exatamente uma desvantagem. Muito pelo contrário: torna o desenvolvimento muito mais simples e baseado fortemente em funções. Só como um adendo, a comunidade entende que o binding de duas vias é mais sofisticado. Espero ajudá-lo a formar uma opinião adequada sobre o assunto até o final desta obra.

Performance

Já conversamos um pouco sobre o assunto na seção de manipulação do DOM. Grande parte da superioridade de execução do React se dá pelo Virtual DOM. Além disso, a suposta desvantagem do *one-way binding* traz um benefício de performance que nos faz refletir melhor sobre qual proposta de fluxo é superior.

Em outras palavras, a performance obtida pelo fluxo de uma via é incontestável (lembre-se do gráfico apresentado na seção *Manipulação do DOM*). Vamos agora construir um raciocínio

simples para tentar explicar o quão eficiente é o binding de uma via.

Pense comigo: em frameworks que trabalham com o *two-way binding*, como o Angular, os atributos dos objetos JavaScript que estiverem ligados ao DOM vão refletir suas alterações instantaneamente. Isso é ótimo! Há poucos anos, desenvolvedores do jQuery precisavam atualizar os valores do DOM manualmente. Por outro lado, quando um desses elementos do DOM (devidamente configurado para trabalhar com o *two-way binding*) sofrer uma alteração, o atributo do JavaScript também será atualizado.

Minha pergunta é: isso é realmente necessário? Se você trabalha com Angular, deve estar injuriado pensando *"Mas nem sempre é necessário! Há alternativas, como event binding"*. Para quem não conhece, o *event binding* (ligação por evento) do Angular é uma das formas de binding que permite passar dados dos templates (DOM) para o JavaScript.

É verdade, o Angular oferece binding mais comportados. Porém, na prática, quando construímos os componentes, são raros os momentos em que exploramos apenas um dos quatro tipos de binding. No geral, usamos todos misturados – e se você desenvolve com Angular, sabe que estou falando a verdade.

Vou aprimorar a pergunta: será que, em um formulário HTML, é realmente necessário atualizar os atributos no JavaScript para todas as tags alteradas? Não seria melhor atualizá-lo **somente** quando o usuário clicasse no `<submit>` ? Do ponto de vista de performance, sim. O melhor é atualizar quando **for necessário** atualizar.

O React ganha nesse sentido porque ele enfatiza fortemente a atualização do estado dos objetos via eventos (algo como o *event binding* do Angular). Ele condena o acesso aos componentes explicitamente, e isso torna-o simples, rápido e de fácil entendimento.

Não se preocupe com os detalhes técnicos agora. Vamos evoluí-los massivamente neste e nos próximos capítulos.

Componentes

O React renderiza seus componentes através do método `render()`, que fica dentro da classe do componente. Basicamente, isso significa que as classes ficam com muitas responsabilidades – que é uma má prática, apesar de facilitar muito a manutenibilidade.

Só vou deixar uma pergunta no ar: será que a proposta do React, na qual as tags HTML ficam dentro do código, é realmente muito pior do que as condicionais e os laços iterativos que outros frameworks JavaScript colocam dentro do HTML (templates)?

Arquitetura MVC

MVC é um padrão arquitetural de projetos que segrega as responsabilidades principais de uma aplicação em três camadas, por assim dizer. O *Model* (referente ao *M* do acrónimo) é responsável por manter os dados e a lógica de negócios da aplicação. A *View* (*V*) é a camada de apresentação, e a *Controller*, como o próprio nome já diz, é responsável por controlar o fluxo das ações realizadas pelo usuário na *View*.

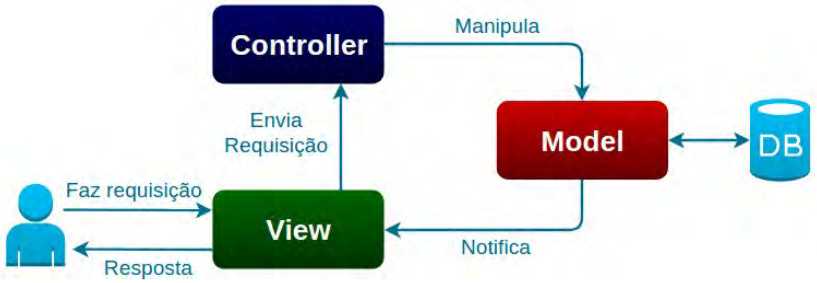

Figura 2.6: Arquitetura MVC

Há representações mais conservadoras que indicam a comunicação do usuário direto com a *Controller*. Na prática, o usuário é uma entidade externa que realiza sua requisição a partir de um evento disponibilizado na *View*, que, por sua vez, repassa a requisição para a *Controller*.

Toda lógica e organização das ações são tratadas pela *Controller*. Pode-se entender que a conclusão do trabalho desta camada geralmente resulta em uma espécie de manutenção dos dados de negócio da aplicação, refletidos diretamente no modelo da aplicação (ou *Model*).

O modelo da aplicação nada mais é do que um conjunto de entidades de domínio que representam os dados do sistema em uma estrutura de classes e atributos. A palavra *domínio* denota o *escopo* que a aplicação pretende resolver. Ou seja, em um site de *e-commerce* (vendas), a classe `Produto` seria uma entidade do domínio. Como os dados são preservados nessa camada, é muito comum estreitá-la com os objetos de um banco de dados, persistindo entidades do modelo em tabelas ou documentos (JSON).

Após a manipulação do *Model*, a *View* é notificada e automaticamente exibe as alterações para o usuário. É muito comum que esta *notificação* seja realizada por um padrão de projeto chamado *Observer*, que, em poucas palavras, permite que um objeto (da *View*) perceba a alteração no estado de outro (do *Model*) sem criar um acoplamento forte entre eles. Por fim, depois de o usuário perceber os ajustes na *View*, ele geralmente faz outra requisição, e o fluxo se repete.

Há uma infinidade literária sobre o assunto. Caso queira se aprofundar no MVC, leia o *post* de Pablo Pastor (2010) chamado *MVC for Noobs*. Caso queira se aprofundar no padrão de projeto Observer, leia o post *Observer Design Pattern* (em inglês) do site BlackWasp (CARR, 2009).

Por que toda essa história? Muito se fala que o React não se adequa bem ao padrão MVC, fato em parte verdadeiro. Só preciso ressaltar que na verdade o React **não** aspira se adequar ao MVC. A equipe mantenedora do React tem um vídeo (https://facebook.github.io/flux/docs/videos.html) explicando as desvantagens do MVC.

Há muita discussão, e não pretendo me prolongar preterindo um ou outro padrão arquitetural. O que é importante você perceber nesta seção é que o React não provê uma estrutura natural para se encaixar ao MVC. Se você quiser trabalhar com esse padrão, terá de criar seus próprios mecanismos para isso.

Neste livro, vamos adotar uma camada de modelo (também chamada de *domínio*), que será bastante discutida no *Capítulo 6*. Lá você verá que a classe JavaScript será criada sem utilizar qualquer recurso do React. Será um simples código ECMAScript 6,

construído para ser consumido pelos componentes React e organizar os dados de negócio da nossa aplicação.

2.2 AS PRINCIPAIS ENGRENAGENS DO REACT

React é um framework de camada de apresentação, fundamentado em componentes. Certo, já sabemos disso! Vamos desconstruir essa assertiva e focar no *componente*, o alicerce das aplicações React. Um componente é um objeto JavaScript que contém propriedades, estado e um ciclo de vida gerenciado pelo framework para possibilitar a atualização do DOM de forma fácil e organizada.

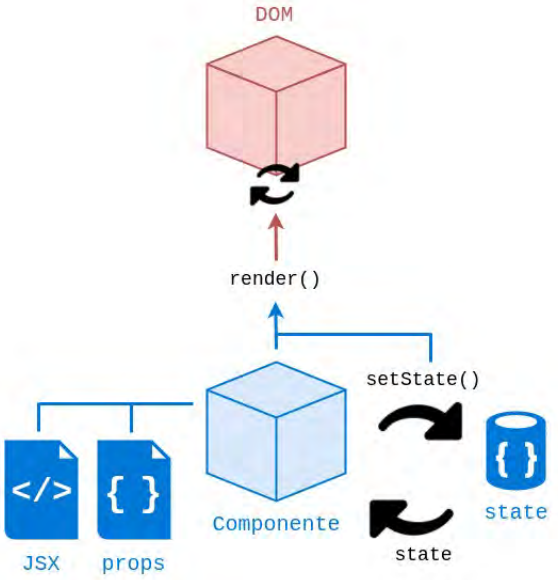

Figura 2.7: Componente do React

A figura anterior mostra superficialmente como os componentes se encaixam na arquitetura front-end de uma aplicação web. Um componente pode receber parâmetros para customizar sua renderização no DOM, que, no mundo do React, são chamados de `props` (propriedades).

A forma como o componente será renderizado é organizada pela sintaxe JSX. Ela estrutura tags semelhantes ao HTML e facilita muito a compreensão das coisas. Veja no código a seguir um pequeno trecho destacado no `return` do método `render()`. Isso é HTML? Não. Isso é um código JSX.

```
render() {
    return (
        <h1>Eu sou um código JSX!</h1>
    );
}
```

Um componente possui vários métodos para controle do ciclo de vida. O único de uso obrigatório é o `render()`. Toda vez que `render()` for executado, ele desencadeará uma atualização do DOM. Isso acontece quando o componente é criado, ou quando seu estado (`this.state`) é atualizado. Por estado, entenda um conjunto de dados inerentes ao funcionamento do componente.

No geral, a leitura dos dados do estado é feita pela sintaxe `this.state`, enquanto a escrita será realizada pelo método `setState()`. Há exceções que veremos em detalhes durante o livro.

O fluxo de dados (propriedades e estados) do React é unidirecional. Um componente pai passa dados para seus componentes filhos, que, por sua vez, passa dados aos componentes netos, e assim por diante.

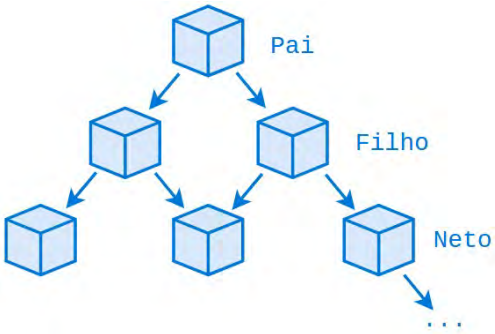

Figura 2.8: Direção do fluxo de dados do React

Já vimos alguns conceitos, mas sinto falta de alguma coisa...

Cadê o código?

Se você for um desenvolvedor raiz, já deve estar pensando: *"Pare com essas figurinhas bonitinhas e mostre logo o código"*. Certo, vamos lá.

Nosso primeiro contato será conduzido com uma conversão das figuras expostas há pouco em código real. O exemplo tratado aqui contém código de complexidade média, fato este que nos levará a segregar seu total entendimento pelos próximos capítulos. Ou seja, agora vamos mostrar a sintaxe superficialmente. Nas seções seguintes deste e dos próximos capítulos, vamos esclarecer em detalhes todos os recursos até culminarmos em sua reprodução completa no *Capítulo 5*.

Vamos lá! Por questões didáticas, por enquanto vamos omitir o código HTML e as importações JavaScript necessárias para que o exemplo funcione. Caso você queira vê-lo na íntegra, o código está disponível no repositório GitHub do projeto

(https://github.com/lgapontes/behappywith.me). O link para acesso direito ao arquivo é: https://raw.githubusercontent.com/lgapontes/behappywith.me/master/docs/cap2/exemplo.html.

Uma das formas de se criar componentes é através da sintaxe `class`, do ECMAScript 6. No mundo da programação orientada a objetos (POO), uma classe é uma espécie de *template* a partir do qual criamos novos objetos.

Basicamente, uma classe é composta por características e comportamentos. Isso significa que todos os objetos criados a partir dela serão formados por esses elementos.

Como trata-se de um código de exemplo, nada mais justo do que nomear a classe como `Exemplo`.

```
class Exemplo extends React.Component {
}
```

Se você está habituado à sintaxe Java, já sabe que `extends` é utilizado para indicar que a classe `Exemplo` herda os atributos e métodos da classe `React.Component`. Sob o prisma da Orientação a Objetos, é muito comum reaproveitar características e comportamentos dos objetos por meio de um recurso conhecido como *herança* (ou `extends`). Com ela, podemos especificar uma classe *filha* (ou subclasse) a partir de uma classe *mãe* (ou superclasse).

No nosso caso, tudo que existe na superclasse `React.Component` será repassado para a subclasse `Exemplo`. Se quisermos definir um estado inicial para os componentes, podemos defini-lo dentro do `constructor` da classe, conforme

adiante.

```
class Exemplo extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            cor: 'red',
            contador: 0
        };
    }
}
```

Além de outras minúcias que serão discutidas no futuro, estamos definindo um estado inicial com uma `cor` igual a vermelho e um `contador` igual a zero. Isso é realizado pelo atributo `this.state`. Note também que o `constructor` (um método que sempre é chamado durante a criação do objeto) recebe as propriedades através do parâmetro `props` e, em seguida, repassa-o para a superclasse `React.Component` pela sintaxe `super()`. Esta é bastante difundida em linguagens orientadas a objetos para representar a convocação do `constructor` da superclasse.

Você lembra de que conversamos brevemente sobre a necessidade de um método para renderizar os componentes no DOM? Mais uma vez, este método é o `render()`. Neste exemplo, `render()` conterá uma pequena estilização para a tag `<h1>` e um retorno de alguns elementos JSX. Desta vez, o código do `constructor` foi omitido.

```
class Exemplo extends React.Component {
    // constructor
    render() {
        const estilo = {
            padding: 10,
            background: this.state.cor
        }
```

```
        return (
            <h1
                style={estilo}
                onClick={this.trocarCor.bind(this)}
            >
                {this.props.texto}
            </h1>
        )
    }
}
```

Aproveito para introduzir o comando `bind()`, uma sintaxe bastante difundida no JavaScript. O termo oficial para ele é *bound function*, ou função vinculada. Este é um recurso implementado no ECMAScript 5 para permitir repassar o objeto corrente (`this`) para a função chamada (`trocarCor()`, no nosso caso).

Em outras palavras, o uso do `this` dentro do método `trocarCor()` só será possível porque utilizamos a sintaxe `bind()` para repassá-lo implicitamente para a função. Ratifico que a passagem é *implícita*! Ou seja, não precisamos declarar um parâmetro chamado `this` na assinatura da função. O `bind()` já gerencia isso para nós.

Além do `bind()`, há muito a ser estudado sobre essa sintaxe. Abordaremos tudo com muita calma ao longo do livro. Por enquanto, reforço apenas que essas tags não são HTML, e sim JSX. Note também que estamos invocando um método chamado `trocaCor()` no evento `onClick` da tag `<h1>`.

Mas onde está o código de `trocaCor()`? Vamos implementá-lo agora. Neste método, obtém-se o valor de `count` do estado do componente para realizar uma condição ternária e definir as cores vermelha ou azul.

Uma condição ternária é aquela cujo `if` é omitido em prol de uma sintaxe simplificada, tal como `condicao ? casoVerdadeiro : casoFalso`. As cores vão variar de acordo com o `mod` (resto da divisão) do incremento de `count`. O mais importante aqui é o uso do método `setState()`, responsável por atualizar o estado do componente.

```
class Exemplo extends React.Component {
    // constructor

    trocarCor(e) {
        let contador = this.state.contador;
        let cor = ( ++contador % 2 ) == 0 ? 'red' : 'blue';
        this.setState({
            cor: cor,
            contador: contador
        });
    }

    // render()
}
```

Este exemplo nos mostrou um típico componente React *stateful* (com estado). Para renderizá-lo no DOM, precisamos utilizar uma função `ReactDOM.render()` interna do React. Nela, basicamente informamos o componente que será renderizado e o elemento HTML no qual o código será injetado.

```
ReactDOM.render(
    <Exemplo texto="Olá Mundo!" />,
    document.querySelector("#exemplo")
)
```

Não deixe de notar que estamos passando um valor `"Olá Mundo!"` como parâmetro na chamada do componente. Este é um exemplo do envio de propriedades.

Nosso exemplo está concluído. Se executado, ele produzirá um

simples `<h1>` que altera seu `background` entre vermelho e azul a cada clique recebido (sobre ele). Veja a renderização na figura adiante.

Figura 2.9: Execução do componente Exemplo

Não se preocupe se você achou tudo muito estranho (ou até mesmo bizarro). Esse é um código completo no qual você estará apto a construir em breve. Vamos entender cada sintaxe apresentada aos poucos, começando a partir da próxima seção.

Desmistificando o JSX

O objetivo do JSX é facilitar a implementação dos componentes. É importante ratificar, entretanto, que você não é obrigado a utilizá-lo. O React oferece APIs nativas do ECMAScript 5, fato que torna tanto o JSX quanto o ECMAScript 6 não obrigatórios.

Então, por que usá-lo? Porque ele realmente torna a confecção do código mais elegante e manutenível. Veja adiante uma comparação entre a sintaxe nativa do React e do JSX.

```
// Codigo nativo do React
```

```
var ExemploNativo = React.createClass({
    render: function() {
        return React.DOM.div(null,
            React.DOM.h3(null, 'Ex h3'),
            React.DOM.span(null, 'Ex span')
        );
    }
});

// Componente com JSX
class ExemploJSX extends React.Component {
    render() {
        return (
            <div>
                <h3>Ex h3</h3>
                <span>Ex span</span>
            </div>
        )
    }
}
```

A notação antiga é notoriamente antiquada e deve ser desencorajada. Veja que a sintaxe JSX de `ExemploJSX`, que reproduz os mesmos elementos do `ExemploNativo`, continua com uma notável semelhança ao HTML. Veremos suas diferenças ao longo do livro.

À primeira vista, há certo preconceito em ter um suposto código HTML dentro do JavaScript. É o que todos dizem, inclusive eu. Para enfatizar que isso não é tão ruim assim, vou refazer a pergunta que deixei em aberto no tópico *Componentes* da primeira seção deste capítulo: será que ter código HTML (na verdade, JSX) dentro do JavaScript é realmente muito pior do que ter lógicas condicionais dentro do HTML? Vejamos algo concreto para melhorar nossa análise.

No React, como as tags são expostas dentro do JavaScript, é muito comum criar funções que retornem um trecho a partir de

uma condição específica. Imagine, por exemplo, um atributo booleano chamado `this.valor` que, ao assumir o valor `true`, provocará a exibição de um parágrafo `<p>` com o texto *"Condição verdadeira"*. Por outro lado, se ele for `false`, a tela deve exibir o texto *"Condição falsa"*.

Essa lógica pode ser implementada com uma simples função que condiciona os dois possíveis retornos através de um `if` associado ao atributo `this.valor`. Veja o código adiante.

```
// Exemplo React
umaCondicao() {
    if (this.valor) {
       return <p>Condição verdadeira</p>;
    } else {
       return <p>Condição falsa</p>;
    }
}

render() {
    return umaCondicao();
}
```

Para ajudar em nossa análise, vou citar brevemente uma sintaxe do framework Angular (da versão 2, adiante). No Angular, como o código HTML (template) fica exposto separadamente da classe de gerenciamento da lógica da tela, em vez da sintaxe JavaScript nativa, podemos criar regras condicionais a partir de uma tag própria chamada `ngIf`.

Vamos reproduzir a mesma lógica dos parágrafos `<p>` com texto *"Condição verdadeira"* e *"Condição falsa"* condicionado pelo valor do atributo `this.valor`. Só como observação, o uso de lógica no HTML é muito comum em outras linguagens Web (antigas), tais como PHP, JSF ou ASP.

```
<!-- Exemplo Angular -->
<p *ngIf="valor">
    Condição verdadeira
</p>
<p *ngIf="!valor">
    Condição falsa
</p>
```

O código Angular mostra uma das formas de se codificar condições nos templates. Também há outros formatos. Será que usar diretivas `*ngIf` no meio de código HTML é realmente uma boa prática? Só estou colocando essa comparação para você meditar sobre o assunto, e não absorver todas as polêmicas em volta do JSX que pipocam na internet.

Neste livro, sempre adotaremos o ECMAScript 6 revestido das tags JSX, só utilizando ECMAScript 5 ou a API nativa do React quando for estritamente necessário.

Virtual DOM

É sabido que o Virtual DOM é vencedor no que diz respeito à performance de renderização. A partir de agora, vamos analisar de perto como o React realiza essa tarefa.

Muito se fala que manipular elementos do DOM não é performático. O **porquê** de não ser performático é que muitas vezes não fica tão claro aos desenvolvedores. Na verdade, há uma certa confusão nessa definição de *lento* ou *pouco performático*. Vamos esclarecer exatamente onde o calo aperta.

Remover ou adicionar objetos no DOM não é uma operação lenta. Lentas são as operações *síncronas* que o browser precisa fazer para ajustar o layout e o CSS toda vez que o DOM muda. E como

ele muda sempre, essa operação tida como lenta precisa ser realizada várias vezes durante o uso da página.

Reforcei a palavra *síncrona* propositalmente porque, diferente do comportamento assíncrono e de I/O não bloqueante do JavaScript, o browser fica *travado*, por assim dizer, ao realizar a atualização do DOM.

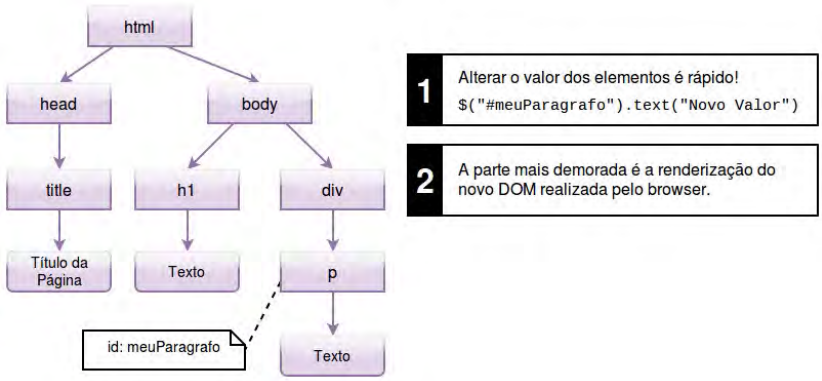

Figura 2.10: Atualização do DOM

Dito isso, adivinhe: é neste ponto que o Virtual DOM atua! O Virtual DOM, assim como o DOM real, é uma árvore que organiza o conteúdo e os atributos dos elementos em forma de objetos. Cada vez que os dados dos componentes são alterados, o React cria uma nova árvore com seus componentes. Isso é realizado com o método `render()`.

A partir dessa árvore, o React renderiza uma nova representação da interface baseada no Virtual DOM. Digo *nova* porque ele não descarta a antiga representação; ela ainda será útil em breve. Note que até o momento, o DOM real não foi afetado. O React apenas criou uma renderização para o DOM virtual.

De posse da antiga e da nova representação, o React calcula a diferença entre elas. A partir daí, ele descobre tudo o que de fato foi alterado. Por fim, ele somente aplica ao DOM do browser as alterações contidas na diferença entre as representações.

Achou confuso? Os passos são:

1. No primeiro acesso à página, o React cria os componentes;
2. Os componentes são renderizados no Virtual DOM (antigo);
3. Como se trata da primeira renderização, o DOM real é atualizado na íntegra;
4. Os dados (`this.state`) dos componentes são alterados;
5. Os componentes são renderizados no Virtual DOM (novo);
6. O React calcula a diferença entre o Virtual DOM antigo e o novo;
7. Por fim, o React atualiza o DOM real apenas com as diferenças.

Veja uma representação adiante:

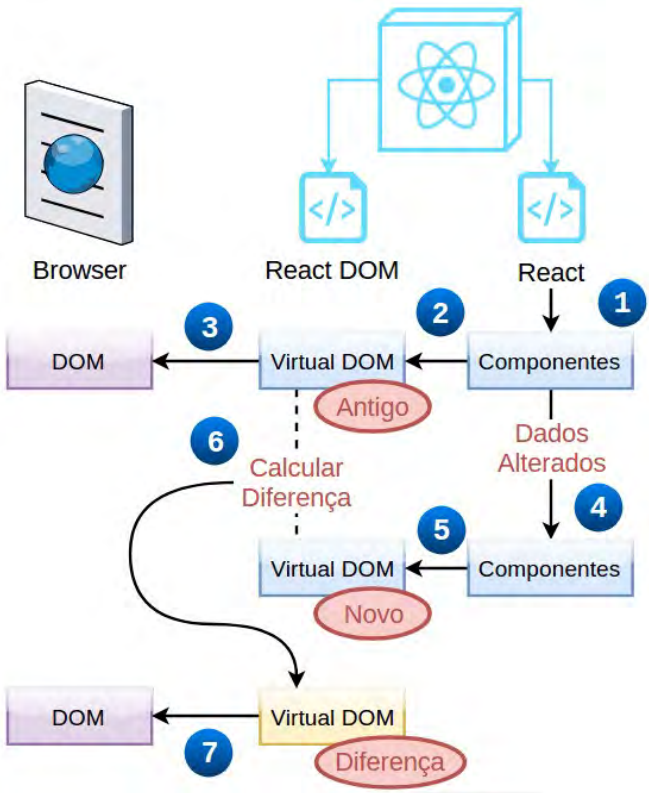

Figura 2.11: Atualização do Virtual DOM

Esse algoritmo é muito próximo à técnica de *patch* usada nos gerenciadores de versão, como o Git, por exemplo.

Algoritmo patch

Suponha um repositório X. Um usuário qualquer cria um *fork* (uma bifurcação de um projeto já existente) de X e nomeia-o como Y. O repositório Y evolui com várias alterações. Depois de algum tempo, o usuário deseja saber quais alterações foram realizadas em Y em relação ao repositório original X. Essa coleção de alterações realizadas é chamada de *patch*.

Fecharemos este estudo com um contra-argumento importante: apesar da eficácia do Virtual DOM, sua performance fica aquém da manipulação do DOM via JavaScript puro. Um estudo foi realizado pelo site Object Partners (https://objectpartners.com/), chamado *Comparing React.js performance vs. native DOM*, que, entre outras conclusões, apresentou a seguinte comparação (OBJECT PARTNERS, 2015):

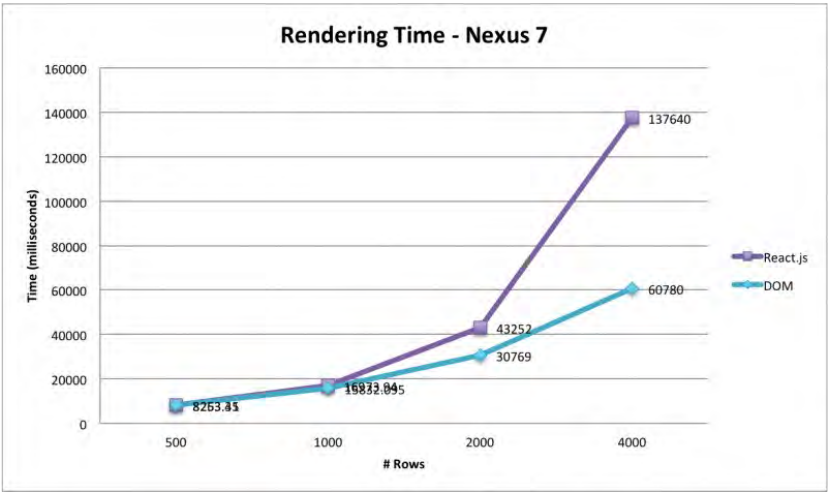

Figura 2.12: Renderização React vs. DOM

Este gráfico mostra o tempo de renderização do React e do DOM nativo, sendo ambos executados no Chrome do tablet Nexus 7. No eixo das abscissas, note que o desempenho do React vai piorando com o aumento do número de *rows* (a simulação insere linhas em uma tabela dinamicamente).

Apesar desta desvantagem, o autor do artigo também reforça que o desenvolvimento através de um framework como o React amplifica questões como sustentabilidade e manutenibilidade do código. Em outras palavras, você perde alguns pontinhos na performance, mas compensa no andamento do projeto.

Fechamos aqui o ciclo introdutório sobre os conceitos envolvidos com a codificação React. Logo no início do *Capítulo 3*, construiremos outro exemplo. Nesta ocasião, contudo, vamos destrinchar todos os recursos agregados de forma completa.

As próximas seções deste capítulo vão apresentar os fundamentos associados à IDE, ao framework Pure.css, e ao servidor web Nginx.

2.3 IDE DE DESENVOLVIMENTO

IDE é um acrônimo para *Integrated Development Environment*, ou Ambiente de Desenvolvimento Integrado. Você pode usar a IDE que lhe for mais confortável. Caso não tenha nenhuma preferência, permita-me sugerir uma das três possibilidades que mais gosto.

Sublime 3

Um excelente editor de texto. Ele pode ser baixado gratuitamente, porém possui uma licença para uso contínuo. Caso você não a coloque, ele continua funcionando sem problemas. Ele é o mais *leve* entre as três opções apresentadas. O link para download é https://www.sublimetext.com/.

Atom

Outro excelente editor. Este é open-source, funciona em todas plataformas e é mantido por uma comunidade ativa no GitHub. Entre no site https://atom.io/ para mais informações. Possui um visual mais *cool* se comparado ao Sublime.

Visual Studio Code

Não se assuste! O VSCode nada tem a ver com o Visual Studio convencional, gigante e pesado. Este é gratuito, leve, e tão bom

quanto seus concorrentes. Reza a lenda que ele é um *fork* do Atom, mas vamos deixar esse assunto para outra hora. Você pode baixá-lo em https://code.visualstudio.com/.

Outros editores

Particularmente, estou usando o VSCode, mas fique à vontade para escolher sua IDE. Além das apresentadas, há também o *TextMate*, o *Coda*, o *Notepad++* e o *Vim*. Escolha uma e instale-a no sistema operacional de sua preferência.

2.4 FUNDAMENTOS SOBRE O PURE.CSS

Existem dezenas de frameworks CSS. Qual deles escolher? Pode parecer uma tarefa simples, mas é muito importante considerar as características do projeto para não arriscar em uma escolha frustrada. Sabemos que nossa aplicação é progressiva, deve ser uma Single-Page Application, modelada primeiramente para ser compatível com dispositivos mobile, ter um curto ciclo de vida e utilizar frameworks aceitos pela comunidade.

Pelo que já vimos neste capítulo, podemos acrescentar uma nova característica que pode nos ajudar: o framework front-end será o React. Mas isso faz alguma diferença? Sim! E para que você entenda o motivo, precisaremos nos aprofundar um pouco em como o React injeta estilo nos componentes.

Estilizando os componentes

Como vimos, o React organiza a aplicação em componentes, que são peças independentes mantenedoras de propriedades, de

um estado que pode ser manipulado ao longo do tempo, métodos de ciclo de vida e código JSX. Além disso, podemos também considerar que um componente é responsável por seu próprio CSS.

Figura 2.13: Componente do React

Por baixo dos panos, um componente é semelhante a uma função JavaScript. Ele recebe propriedades (`props`) como parâmetro e possui mecanismos para renderizar elementos no DOM do navegador. Tais propriedades podem ser valores (como um texto para ser exibido) ou uma função (para viabilizar a captação de eventos oriundos da interface).

Dito isso, é importante ratificar que o CSS fica **dentro** do componente. O React aconselha-nos a usar a estilização inline *por componente*. Isso significa que cada componente deve ter os elementos CSS necessários para sua exibição correta na página.

Isso pode parecer estranho aos olhos dos designs (e até dos programadores), mas há uma justificativa razoável: quando desenvolvemos projetos com base em componentes, há expectativa de que cada componente seja um artefato reusável dentro ou fora do projeto. Se um componente tiver seu layout quebrado quando for utilizado em outro trecho do sistema, esse propósito se

perderia. Manter as classes CSS inline no componente ajuda a prevenir esse problema.

Esse entendimento é importante para que você compreenda que, além dos frameworks *queridinhos* (como Bootstrap, Materialize e Pure.css), há também aqueles construídos especificamente para atender a estilização inline do React. Isso nos leva a crer que os escolher seria o caso mais óbvio. Todavia, como você verá a seguir, existem vários problemas de compatibilidade que influenciariam negativamente no projeto.

Por que não utilizar os frameworks CSS inline?

Escolher um framework criado para o React seria a opção mais lógica, se olharmos o contexto rapidamente. Uma análise mais de perto, todavia, nos mostrará várias desvantagens. O objetivo desta seção é fazê-lo perceber que nem sempre o óbvio é a melhor opção.

Existem vários frameworks *React-like*. Aqui, porém, vamos conversar sobre *React Bootstrap*, *React Pure*, o *Elemental-UI* e *React Toolbox*.

Para quem não conhece, Bootstrap (http://getbootstrap.com/) é o mais famoso framework CSS do mercado. Há um projeto não oficial (https://react-bootstrap.github.io/) que encapsulou as classes do Bootstrap em componentes React. É muito bem aceito pela comunidade, mas não há garantia de que haverá compatibilidade com a próxima versão (de número 4) do Bootstrap, o que pode ser um problema de continuidade em nosso projeto (e qualquer outro).

Por que não devemos utilizá-lo? Os principais motivos são que

ele apresenta problemas de compatibilidade com as versões mais novas do Webpack (uma biblioteca necessária para projetos React, que será tratada no *Capítulo 3*) e, apesar de ser muito aceito pela comunidade, não possui uma definição clara de quando sofrerá manutenção para acompanhar a evolução do Bootstrap oficial.

Assim como o Bootstrap, o Pure.css também contém uma versão específica para o React, chamada de *React Pure*. Trata-se de um projeto não oficial (https://github.com/hailocab/react-pure) que encapsulou a especificação do Pure.css em componentes React. Durante a escrita deste livro, ela possuía ínfimas 95 estrelas no GitHub, o que pode resultar em uma descontinuidade. Isso já é suficiente para descartá-lo.

Falta-nos discorrer sobre o *Elemental-UI* e *React Toolbox*. Ambos foram criados para suportar aplicativos React. Em outras palavras, eles não existem para aplicativos que não o utilizam.

No caso do *Elemental-UI* (http://elemental-ui.com/), apesar de ele ter componentes muito bonitos, com visual agradável e com um ótimo template para botões, ele possui problemas de compatibilidade com versões antigas do Firefox, fato que a documentação não trata com a devida atenção, infelizmente. Escolhê-lo nos levaria a um cenário nebuloso do tipo: *"Quais usuários do Firefox podem usar nossa aplicação?"*.

O *React Toolbox* (http://react-toolbox.com/) é mais sofisticado, e suas 6k estrelas representam bem este fato. Podemos entender que ele seria a melhor opção entre os frameworks *React-like* apresentados, exceto por dois graves problemas de compatibilidade:

1. Ele só funciona com Internet Explorer 11+. Parece não ser importante, mas as estatísticas (http://caniuse.com/usage-table) apontam cerca de 0,86% usuários com Internet Explorer anteriores à versão 11. Isso significa que quase 1% dos usuários não poderá usar nossa aplicação progressiva.
2. Ele também apresenta incompatibilidade com o *Webpack 3*, pacote crucial na orquestração da arquitetura de nossa PWA (como veremos no *Capítulo 3*).

Não fosse pelos problemas de incompatibilidade ressaltados, poderíamos escolhê-lo. Mas acreditem: problemas de configuração ou compatibilidade provocados pelos frameworks CSS são uma das piores coisas que podemos vivenciar em projetos de software.

Pure.css

Acredite se quiser, mas os arquivos deste framework possuem só 17kb. Isso mesmo, são apenas 17 kbytes de arquivos! É uma excelente opção para nós, que precisamos otimizar o primeiro carregamento da página.

Ele possui uma boa compatibilidade e pode ser executado sem problemas até com o IE7+. O único ponto de desvantagem é que sua documentação é muito vaga quando cita a menor versão dos browsers Chrome, Safari e Firefox.

Só como observação, do ponto de vista de aceitação, ele possui 16 mil curtidas no GitHub. Um valor bem razoável, que abafa qualquer risco de descontinuidade. Opcionalmente, se quiser mais detalhes, acesse https://purecss.io/.

Aos leitores amantes dos frameworks Bootstrap e Materialize,

não fiquem chateados! Nesta obra, optei em não os escolher porque ambos pecam no tamanho, que chega a ser quase dez vezes maior do que o Pure.css.

Lembre-se: a palavra *progressiva* é o nosso direcionador para as melhores escolhas. E o tamanho dos arquivos é um fato importante.

2.5 FUNDAMENTOS SOBRE O NGINX

Nesta seção, vamos conversar um pouco sobre o Nginx (de pronúncia *enginex*), um dos servidores web mais utilizados na internet segundo as estatísticas do site https://w3techs.com. Tudo bem que esses dados mudam constantemente, e é bem provável que, quando você estiver lendo este parágrafo, o panorama esteja diferente. De qualquer forma, seguem os top 5.

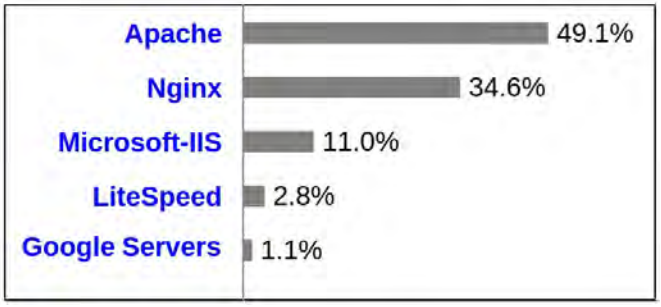

Figura 2.14: Uso de servidores web na internet

Nossa aplicação web não terá back-end dinâmico. Ela será construída apenas com código React, que, no fim das contas, será convertido no mais puro ECMAScript 5, otimizado e minificado. Isso significa que qualquer servidor web HTTP que forneça

arquivos estáticos poderia entrar na briga.

Além de servir arquivos, existem outras particularidades que precisamos atender. Suponho que você se lembre dos requisitos das aplicações progressivas estudados no *Capítulo 1*. Alguns deles serão exclusivamente tratados e/ou auxiliados pelo servidor web.

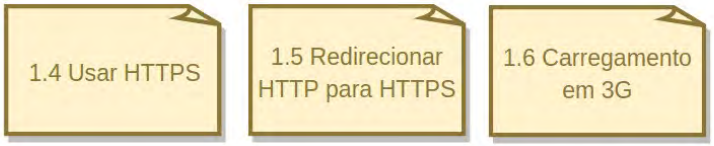

Figura 2.15: Histórias associadas ao servidor web

A história 1.6 está relacionada à velocidade de carregamento da página. A velocidade que o servidor fornece aos arquivos estáticos é muito importante para resolvermos com maestria esse requisito. Prover arquivos, qualquer servidor web faz. Prover arquivos com performance excelente, são poucos.

E há também um poderosíssimo recurso de compactar os arquivos antes de enviá-los para os usuários. Nosso servidor (alerta de spoiler) será capaz de fazer isso!

As histórias 1.4 e 1.5 estão respectivamente relacionadas à obrigatoriedade do tráfego seguro (HTTPS) e ao redirecionamento do HTTP para o HTTPS. Conforme vimos, o HTTP com criptografia será usado para proteger o acesso aos arquivos estáticos do projeto (React e afins), e o redirecionamento é necessário para evitar acessos explicitamente pelo HTTP em claro.

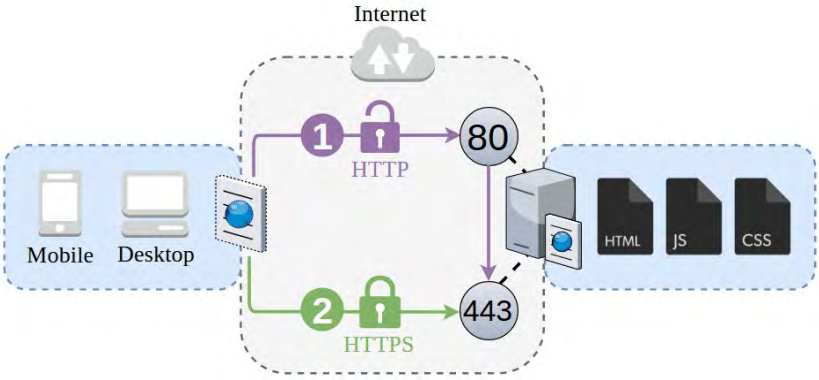

Figura 2.16: Redirecionamento HTTP para HTTPS

Em outras palavras, toda a conversa entre nosso navegador e o servidor web deve ocorrer de forma protegida.

Olhando o Nginx mais de perto

Se comparado com os grandes (leia-se Apache), o Nginx é um servidor web relativamente recente, de 2004. Ele foi desenvolvido para viabilizar o provimento de recursos quando o número de usuários concorrentes é extremo.

Isso é possível pelo fato de ele trabalhar com uma arquitetura assíncrona (não bloqueante) baseada em eventos. O Nginx provê conteúdo estático com maestria, realizando suas ações com base em URIs e traduzindo para a localização física no disco quando for necessário.

Ele não é capaz de realizar processamento dinâmico, mas oferece um proxy reverso (um tipo de servidor que recebe solicitações dos clientes e repassa a outros servidores) para conduzir tais solicitações. No nosso caso, isso não será necessário.

Se olharmos pelo prisma da escalabilidade, o Nginx nasceu com foco na resolução de problemas onde os acessos crescem para a casa das dezenas de milhares. A literatura denomina esse tipo de situação com o acrônimo **C10K**, cujo significado original é *concurrently handling ten thousand connections* (*manipulação simultânea de dez mil conexões*).

Por conta disso, o Nginx é tido como um dos servidores mais performáticos disponíveis. Tudo fica mais fácil de observar com os números. Veja na figura adiante um comparativo publicado no site DreamHost (https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison).

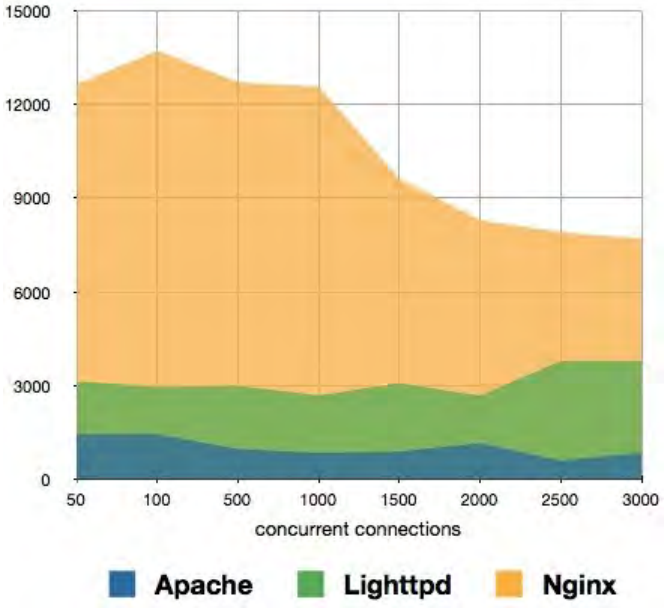

Figura 2.17: Apache vs. Nginx

O *Lighttpd* é outro servidor web um pouco menos difundido

mas bem aceito na comunidade Linux. Não vamos discursar sobre ele. Caso queira conhecê-lo, acesse https://www.lighttpd.net/.

Caro leitor, os números não mentem! De acordo com o comparativo da imagem anterior, o Nginx é capaz de suportar algo em torno de quatro vezes o número de requisições concorrentes em relação aos seus companheiros. Então, podemos encerrar por aqui.

Vamos baixá-lo (https://nginx.org/en/) e instalá-lo no futuro, com passos bem detalhados.

2.6 CONCLUSÃO

Peço desculpas por tê-lo afogado em um conteúdo efetivamente conceitual. É sabido que um capítulo notoriamente teórico como este, que você bravamente concluiu, é enfadonho. Tudo bem, eu reconheço. Mas eu não poderia deixá-lo à mercê das *modinhas* da internet sem abastecê-lo dos prós e contras das tecnologias utilizadas em nossa PWA.

Acredite se quiser, mas escolher um *bom* framework para uma determinada aplicação é uma tarefa árdua! Aconselho fortemente praticar uma conceituação como esta em seus projetos. Sai muito mais em conta do que substituir uma roda com o carro em movimento.

Se você terminou este capítulo confiante (ou pelo menos parcialmente convencido) de que estaremos empregando tudo o que há de melhor em nossa aplicação progressiva, nosso objetivo foi concluído. Lembre-se de que a palavra *progressiva* é justamente uma condição de esforço continuamente empregado ao produto

entregue aos usuários. Se tivéssemos escolhido as ferramentas citadas sem este preciosismo, estaríamos indo de encontro a esse preceito.

Próxima parada: um pouco mais de código React e configuração do ambiente de desenvolvimento.

Capítulo 3

CONFIGURANDO O AMBIENTE DE DESENVOLVIMENTO

Vencemos a barreira teórica! Agora vamos enfrentar a barreira técnica. No final deste capítulo, você estará com o ambiente de desenvolvimento React devidamente configurado e com uma base sobre sua arquitetura. Conversaremos sobre React, ECMAScript 6, Babel, Node.js, Webpack e outras ferramentas.

Há duas maneiras de trabalhar com React:

1. Uma para estudos, direto no browser;
2. E outra pelo Node.js, mais recomendada para ambientes de desenvolvimento real.

Vamos estudar ambas, porém a primeira será apresentada brevemente apenas para lhe dar a possibilidade de exercitar alguns códigos mais simples. Caso você já esteja intencionado a reproduzir a página apresentada neste livro, recomendo fortemente que opte pela opção do Node.js, cuja explicação será logo após a opção do browser. Chega de papo, vamos rodar nossa primeira aplicação React!

3.1 REACT NO BROWSER

A partir de agora, vamos estudar um componente React construído diretamente no browser. Apesar de servir apenas para estudo e ser impraticável no projeto real, vale a pena se debruçar sobre isto neste momento, pelo menos rapidamente, pois é uma boa maneira de ter um primeiro contato, em termos didáticos.

Vamos criar juntos o memorável projeto *Hello World*, com uma sutil adaptação do texto exibido para *Olá React!!!*. Para isto, a primeira coisa que precisamos fazer é criar um diretório para guardarmos nosso código. Vamos chamá-lo de `exemplo`. Crie-o em qualquer lugar do seu sistema operacional.

Convencionalmente, o principal arquivo de uma aplicação, gigante ou minúscula, é chamado de `index.html`. Em nosso exemplo, não faremos diferente. Crie um arquivo `index.html` (vazio) dentro do diretório `exemplo`. Ele precisar estar em branco para iniciarmos do zero.

Normalmente arquivos com essa extensão automaticamente já são *direcionados* aos navegadores. Um simples duplo clique sobre ele seria suficiente para abrir seu browser padrão. Agora, no entanto, digitaremos o código-fonte acessando-o via nossa IDE. Abra-a, procure pelo arquivo `exemplo/index.html` e vamos começar.

Com o arquivo aberto, estamos prontos para digitar as próximas linhas integralmente. Sugiro fortemente que você, sempre que possível, entre pessoalmente com os códigos apresentados no decorrer desta obra. Isso vai ajudá-lo a memorizar a sintaxe, e logo tudo ficará leve como uma pena. A explicação será

feita adiante, logo após sua execução. Vamos ao código.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Exemplo no Browser</title>
        <script src="https://unpkg.com/react@15.6.1/dist/react.js
></script>
        <script src="https://unpkg.com/react-dom@15.6.1/dist/reac
t-dom.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/babel.m
in.js"></script>
    </head>
    <body>
        <div id="exemplo"></div>
        <script type="text/babel">
            class OlaReact extends React.Component {
                render() {
                    return <h1>Olá React!!!</h1>
                }
            }

            ReactDOM.render(
                <OlaReact />,
                document.querySelector("#exemplo")
            );
        </script>
    </body>
</html>
```

A critério de validação, você poderá comparar o código digitado com uma versão publicada no repositório do livro no GitHub, cujo link é https://raw.githubusercontent.com/lgapontes/behappywith.me/master/docs/cap3/browser/index.html.

Abra o arquivo `index.html` em um navegador de sua preferência. Você verá algo similar à imagem a seguir.

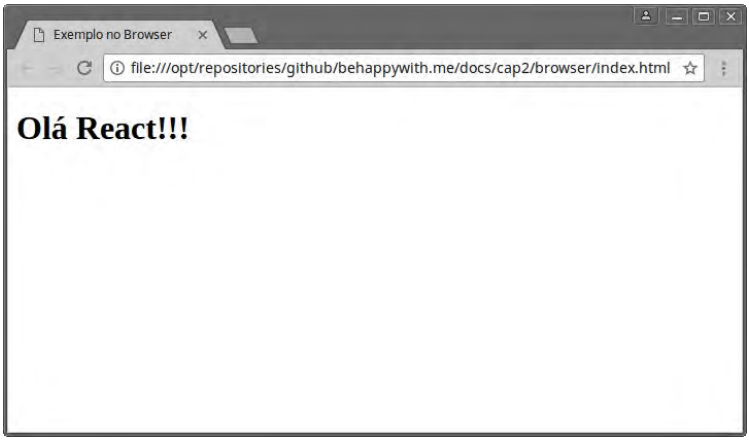

Figura 3.1: Exemplo no browser

Pronto! Nossa primeira aplicação com o React! Antes de encerrarmos o exemplo, abra a ferramenta de apoio ao desenvolvimento do seu navegador (geralmente com o botão `F12`), clique na aba *console* e repare na exibição de um *warning* (aviso).

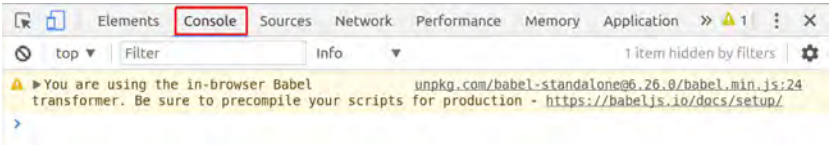

Figura 3.2: Console do exemplo no browser

Não se preocupe, não há falha em seu código. Esta mensagem está lhe avisando que o Babel está compilando (o termo correto é *transpiler*) o código JavaScript em tempo de execução e que, para um ambiente de produção, é necessário realizar uma configuração mais adequada.

Compilando? Babel? Na próxima seção, vamos esmiuçar cada

detalhe.

Os mistérios da torre de Babel

Neste primeiro exemplo, vamos passar a sintaxe por completo para entender com detalhes o que ocorreu no código anterior, concentrando-se principalmente nas tags e componentes associados ao React. Comecemos pelo HTML.

Se tirarmos os trechos de JavaScript, você verá que ele tem um esqueleto simples, como qualquer outro:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Exemplo no Browser</title>
        <!-- Algumas importações quaisquer ... -->
    </head>
    <body>
        <div id="exemplo"></div>
        <script type="text/babel">
            // Código JavaScript ...
        </script>
    </body>
</html>
```

É importante destacarmos as três referências JavaScript importadas no `<header>` da página. A primeira delas, como pode ser visto, diz respeito ao core do React. Por meio dessa biblioteca, podemos criar nossos componentes, gerenciar seus ciclos de vida e utilizar os recursos da sintaxe JSX.

```
<script src="https://unpkg.com/react@15.6.1/dist/react.js"></scri
pt>
```

A segunda importação obtém o gerenciamento do DOM. Essa biblioteca deve ser utilizada em conjunto com a `react.js`. Ela

disponibiliza o principal método de renderização no DOM da página: `ReactDOM.render()`.

```
<script src="https://unpkg.com/react-dom@15.6.1/dist/react-dom.js
></script>
```

Você deve estar se perguntando por que a equipe do React separou a manipulação do DOM, que de fato faz as coisas acontecerem, do código principal. Na verdade, nas primeiras versões, ambas eram providas em uma única publicação. *A posteriori*, o time de desenvolvimento achou bom separá-las com a expectativa de evoluir outras vertentes de desenvolvimento além do DOM – como o *React Native*, para aplicações mobile nativas.

Atualmente, o React Native é distribuído com pacotes totalmente separados do React e ReactDOM, o que nos leva a crer que este propósito inicial se perdeu. Você verá durante o desenvolvimento que, às vezes, dependendo do caso, importaremos uma ou ambas bibliotecas.

Continuando a análise, há mais uma importação necessária: o *transpiler* do Babel.

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"><
script>
```

Mas afinal, o que faz esse tal de Babel? No *Capítulo 2*, conversamos um pouco sobre a evolução do JavaScript ao longo do tempo, muito especialmente sobre as versões 5 e 6 do ECMAScript. Neste ponto, é importante reforçar que, apesar de o ECMAScript 6 ser uma excelente opção de codificação dos aplicativos React, atualmente ela não é inteiramente suportada pelos navegadores. E é aí que entra o Babel!

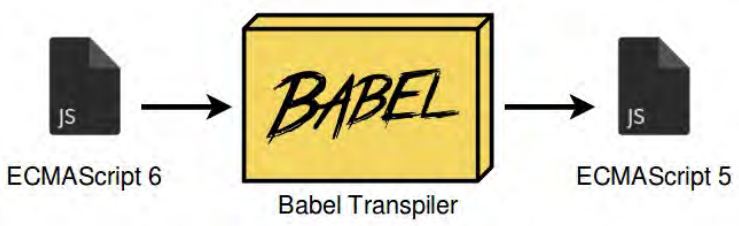

Figura 3.3: Babel Transpiler

O Babel é um transpiler que converte (ou compila) a versão mais recente do JavaScript para uma versão compatível com os browsers.

> **Compiler ou Transpiler?**
>
> A ciência dos compiladores é extensa e nem me atrevo a adentrá-la. Mas, grosseiramente, um *compiler* transforma o código-fonte escrito em uma linguagem de programação em outra linguagem binária conhecida carinhosamente como código-objeto. Um *transpiler*, ou *transcompiler*, é um tipo de compilador que transforma código-fonte em código-fonte.

Resumindo, o Babel converte o ECMAScript 6 para ECMAScript 5. Ele consegue realizar essa proeza porque o código do React está encapsulado dentro da tag script do tipo `text/babel` (em vez do tradicional `text/javascript`).

```
<script type="text/babel"> ... </script>
```

Se você for aficionado com detalhes de performance, como eu, deve estar chateado com essa compilação em tempo de execução.

Não somos os únicos! O próprio Babel também se preocupa. No entanto, ele nos mostrou aquela mensagem de *warning* no console das ferramentas de desenvolvimento do navegador. Vou replicá-la a seguir, no original, em inglês.

```
You are using the in-browser Babel transformer. Be
sure to precompile your scripts for production
```

O Babel está alertando que estamos usando o transformador *in-browser*, e que devemos compilar nossos scripts para uso em ambiente de produção. Em outras palavras, *isso não está cheirando muito bem*. Mas não se preocupe, essa questão será resolvida na próxima seção, quando configurarmos o ambiente de desenvolvimento via Node.js.

Prosseguindo, note que há uma tag `<div>` cujo `id` é `exemplo`, que propositalmente foi deixada vazia.

```
<div id="exemplo"></div>
```

Ela serve para receber o código renderizado do React. Vamos avançar agora para a criação do componente `OlaReact`. O React permite-nos organizar nosso código em componentes, que na prática são objetos autocontidos que mantêm um estado, e possuem propriedades e métodos para gerenciar seu ciclo de vida.

Bom, isso você já sabe! Vamos avançar pelo desconhecido: esses *componentes* devem ficar na tag `<script>` justamente para permitir que o Babel faça seu trabalho.

E como o componente imprime as tags na tela do navegador? Por meio do método `render()`. Aliás, o único método que obrigatoriamente deve ser sobrescrito é o `render()`. Como ele é

o responsável por retornar a sintaxe JSX que será renderizada, não tê-lo codificado tornaria o componente inútil.

```
class OlaReact extends React.Component {
   render() {
       return <h1>Olá React!!!</h1>
   }
}
```

Observação: se `render()` não retornar JSX, ele precisará retornar `null` ou `false`.

Caso não tenha reparado no código anterior, observe que não há aspas duplas ou simples envolvendo o código JSX. Então, a pergunta é: como o navegador entende esse código JavaScript?

Na verdade, não entende. O *transpiler* entra em ação mais uma vez. O Babel trabalha com *presets* que viabilizam um tipo de compilação mais aperfeiçoado, que, além de transformar o ECMAScript 6, também transforma o JSX, tudo em um código compatível com os navegadores. Mais especificamente, o *React Preset* transforma o trecho JSX em chamadas de funções do próprio React.

Veja adiante uma comparação entre a sintaxe JSX e uma chamada da função `createElement()`, sintaxe nativa do React, compatível com ECMAScript 5. Ambas produzem o mesmo resultado.

```
// Classe de componente com JSX
class OlaReact extends React.Component {
    render() {
        return <h1>Olá React!!!</h1>
    }
}
ReactDOM.render(
    <OlaReact />,
```

```
    document.querySelector("#exemplo")
);

// API nativa do React
var OlaReact = React.createElement(
    'h1',
    null,
    'Olá React!!!'
);
ReactDOM.render(
    OlaReact,
    document.querySelector("#exemplo")
);
```

Note que `createElement()` recebe uma `<div>`, um valor nulo (que poderia ser um objeto representando o estilo do componente) e seu conteúdo. Esses parâmetros são usados para criar o componente que foi renderizado na função `render()`.

O método `createElement()` faz parte de uma API de mais baixo nível, oferecida pelo React para suportar projetos sem o uso do *ECMAScript 6*. Não vamos nos aprofundar nessas funções nativas. Eventualmente, todavia, faremos uma abordagem rápida para compreender melhor como as coisas funcionam por baixo dos panos.

Resumindo, o que o Babel faz através do *preset* citado é transformar o código HTML-like do JSX em uma declaração semelhante à API nativa. Achou pouco produtiva a sintaxe da função `createElement()`? Eu também. Só a destaquei aqui para você entender um pouco mais sobre a arquitetura do framework.

Herança no JavaScript

E a sintaxe `class` e `extends`? São comandos do React? Não. Apresento-lhe um excelente recurso do ECMAScript 6. Esse

código, muito próximo ao da linguagem Java, é o formato pelo qual a versão ECMAScript 6 viabiliza a herança entre as classes no mundo JavaScript.

Isso mesmo, não precisaremos mais criar nossos objetos por meio da declaração de funções e herdá-los pelo *prototype* explícito. Digo explícito porque, por baixo dos panos, a Orientação a Objetos do JavaScript continua sendo baseada em protótipos (ou clones).

Muita informação? Vamos aos poucos.

No JavaScript, a herança entre os objetos se dá pela clonagem de todas as propriedades do objeto pai para o objeto filho e, na maioria das vezes, acréscimo de outras propriedades. O conceito de classe introduzido no ECMAScript 6 não se trata de um novo *elemento* presente no JavaScript. Elas são apenas uma sintaxe mais elegante para esconder a herança por *prototype*, que eventualmente não é tão evidente aos desenvolvedores. Outro detalhe: as ditas *propriedades* presentes nos objetos são atributos e métodos.

Resumindo, o ECMAScript 6 escondeu a definição baseada em funções pela notação mais agradável de declaração de classes, atributos, métodos e herança pelo comando `extends`. Mas atenção: reforço que não há burocratização dos objetos em esqueletos (classes) prefixados como nas linguagens Java ou C#. A sintaxe `class` é apenas uma abstração à criação dos objetos JavaScript que estamos habituados.

Para mais detalhes, esta e outras novidades do ECMAScript 6 estão resumidas em http://es6-features.org/. Nela, você encontrará comparações entre todas as novidades do ES6, e como tais recursos eram implementados no ES5. É uma boa referência rápida.

Todavia, caso você queira se aprofundar no assunto, recomendo fortemente a agradável leitura do livro *ECMAScript 6: Entre de cabeça no futuro do JavaScript*, de Diego Martins de Pinho. O livro é uma publicação da Casa do Código (https://www.casadocodigo.com.br/products/livro-ecmascript6).

Renderizando o componente no DOM

Avançando mais um pouco em nosso estudo sobre o código de `OlaReact`, chegamos ao trecho de renderização através da biblioteca `ReactDOM`.

```
ReactDOM.render(
   <OlaReact />,
   document.querySelector("#exemplo")
);
```

Este método `render()` recebe dois argumentos:

1. O trecho JSX que será renderizado, que neste caso é uma tag chamada `OlaReact`. Certamente você já descobriu que este nome é exatamente o mesmo do componente criado. E é assim mesmo que funciona, cada componente criado no React estará disponível como uma tag.
2. O elemento do DOM em que o JSX será colocado. Neste caso, estamos apontando para a `<div>` cujo `id` é `exemplo`.

Mas esse método `querySelector()` é tão *JavaScript Old-School*? Como vimos no *Capítulo 2*, o React trabalha com o DOM Virtual. A ideia desse parâmetro é informar onde o componente será injetado. Então, não se preocupe, este será o único brotossauro que veremos caminhando em nosso código.

Você deve estar achando tudo meio bizarro: código HTML no meio de JavaScript, com chamadas a APIs nativas, criação de classes, compilar código-fonte para código-fonte. Por enquanto, peço o seguinte: dê um voto de confiança ao framework. Você verá na prática (com a codificação) que a componentização e a praticidade oferecidas pelo React nos dá produtividade e manutenibilidade incríveis.

3.2 ARQUITETURA REACT COM NODE.JS

Fato: não é factível construir um projeto React cuja compilação será realizada em tempo de interpretação no próprio browser do usuário. Mas então, como faremos? Uma imagem vale mais do que mil palavras.

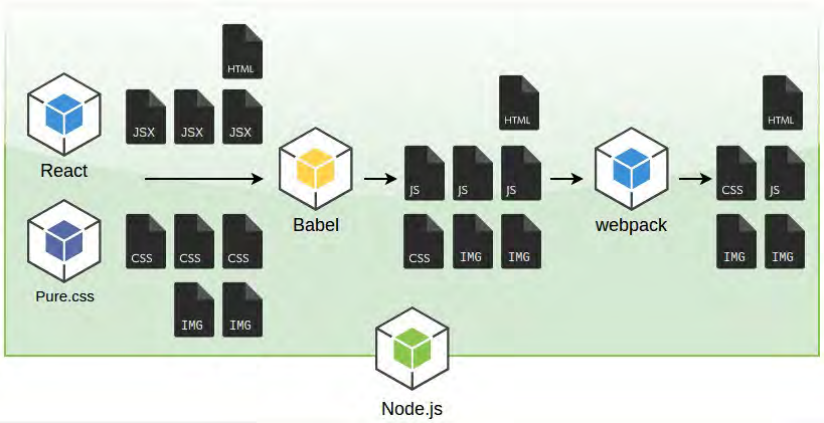

Figura 3.4: Arquitetura React com Node.js

Vamos analisá-la passo a passo:

1. O React trabalha com arquivos JSX para criar seus

componentes e injetá-los em um arquivo HTML. Lembre-se de que as sintaxes JSX e ECMAScript 6 são inúteis para os navegadores.

2. Pure.css é o framework CSS integrado ao React para ajudar-nos a compor os componentes. Teremos CSS para os componentes e para a página HTML. Eventualmente, teremos algumas imagens.
3. Como vimos, o Babel é capaz de converter JSX e ECMAScript 6 em ECMAScript 5. Na arquitetura do Node.js, ele também terá esse papel.
4. Observe que os arquivos CSS transformaram-se em um. Isso ocorre porque, no final, a dupla Webpack/Babel colocará tudo em um único arquivo. O ECMAScript 5 continuará utilizando as bibliotecas do React, neste caso, chamando a função `createElement()` com o trecho CSS como parâmetro.
5. Depois do papel de *transpiler* do Babel, teremos nossos arquivos prontos para rodar. Entretanto, o Webpack dará mais um passo: com ele, vamos juntar todos os JavaScripts em um único arquivo final.
6. Além disso, ele vai minificar CSS e JavaScript e injetá-los no HTML, respectivamente pelas tags `<link>` e `<script>`. Estes arquivos serão a distribuição de nossa PWA.
7. E o Node.js? É ele quem orquestra todos esses passos, organiza as dependências (via NPM) e viabiliza os comandos necessários para cada etapa do processo (pelo Webpack).

Incrível, não? Já conversamos sobre o React, Pure.css e Babel. Falta-nos discorrer um pouco sobre o Node.js, o NPM e o Webpack. Vamos usá-los com detalhes, e é importante que você

entenda exatamente a proposta de cada um. Além do breve conteúdo teórico, veremos também os passos de instalação e alguns conceitos básicos de uso do Node.js e do NPM.

3.3 NODE.JS E NPM

Por volta de 2012, em uma breve conversa com um amigo, descobri que uns magos do JavaScript haviam criado (em 2009) um artefato *back-end* com o V8 do Google Chrome. Pense comigo: colocar uma linguagem de I/O não bloqueante dentro de uma máquina poderosíssima (V8) para receber solicitações no *back-end*. Não tem como dar errado!

Aliás, na época, era uma ideia fantástica. O mercado estava abarrotado de frameworks pesados, que exigiam camadas e camadas de servidores, com projetos intermináveis. Uma nova *frente* de desenvolvimento *rápido e direto* surgia: frameworks VC (View e Controller) embarcados no navegador e APIs RESTful construídas em JavaScript no *back-end*.

De lá para cá, muita coisa mudou. Grandes empresas agora o veem como *"a"* ferramenta. O próprio JavaScript virou *"a"* linguagem. Evoluiu de um recurso acessório a núcleo de aplicações inteiras. Até o JSON venceu a inquebrável barreira dos bancos relacionais e entrou na briga. Tudo isso gerou tanto ânimo que o termo MEAN stack virou o novo padrão de mercado: MongoDB + Express + Angular + Node.

Após essa breve percepção pessoal sobre o panorama do Node.js, vou reproduzir aqui a definição (traduzida) do site oficial: *"Node.js é um executor JavaScript construído com a engine V8 do*

Google Chrome. Node.js usa um modelo de I/O não bloqueante baseado em eventos que o faz leve e eficiente. O gerenciador de pacotes do Node.js, chamado de NPM, é o maior ecossistema de bibliotecas open-source do mundo."

Apesar do tom de propaganda, isso é a mais pura verdade. Desejam saber mais? Leiam as excelentes obras de Caio Ribeiro Pereira, intituladas de *Construindo APIs REST com Node.js* (https://www.casadocodigo.com.br/products/livro-apis-nodejs) e *Aplicações web real-time com Node.js* (https://www.casadocodigo.com.br/products/livro-nodejs), ambas publicadas pela Casa do Código.

E o NPM? O NPM está para o Node.js assim como o NuGet está para o C#, ou o Maven está para o Java. A diferença é que o NPM (*Node Package Manager*) é o mais poderoso gerenciador de pacotes do mundo. A adaptação da palavra *famoso* da documentação oficial para *poderoso* é por minha conta.

Os números envolvidos no NPM são um verdadeiro significado de poder. Semanalmente, são realizados mais de 2 bilhões de downloads dos seus quase meio milhão de pacotes. Se você criar uma boa biblioteca JavaScript e publicar no NPM, no dia seguinte haverá pelo menos uns 100 downloads. Se você fizer uma ótima biblioteca, passará dos 1.000 downloads fácil. Há bibliotecas que recebem mais de um milhão de downloads por dia.

Ao longo deste livro, utilizaremos o NPM para criar o projeto, instalar suas dependências e executar scripts internos. Os comandos são muito intuitivos. Não vamos introduzi-los agora, mas de forma semelhante a como faremos com o React, sempre que um novo comando do NPM for apresentado, vamos discutir

sobre sua sintaxe e finalidade.

Antigamente, os desenvolvedores precisavam instalar o NPM manualmente. Desde a versão 0.6.3 do Node.js, o NPM foi incorporado aos releases oficiais. Isso aconteceu em 25 de novembro de 2011. Nessa época, eu sequer conhecia-os, mas penso que ambos erão inseparáveis.

Agora vamos à instalação. Não vamos estressar todas as plataformas. Mostraremos apenas duas opções: Linux Ubuntu 16.04.2 LTS e Windows 10. Caso você precise instalar em outros ambientes, acesse a página https://nodejs.org/en/download/package-manager/.

Instalando Node.js no Windows

Primeiramente, acesse o site oficial do Node.js, em https://nodejs.org/.

Recomendo a instalação da versão LTS (Long-term support), que é a última versão estável disponibilizada. Durante a escrita deste livro, estamos na 6.11.0, mas não se omita em instalar uma versão mais recente, se houver. Baixe o pacote `node-v6.11.0-x64.msi` e execute-o como administrador do sistema.

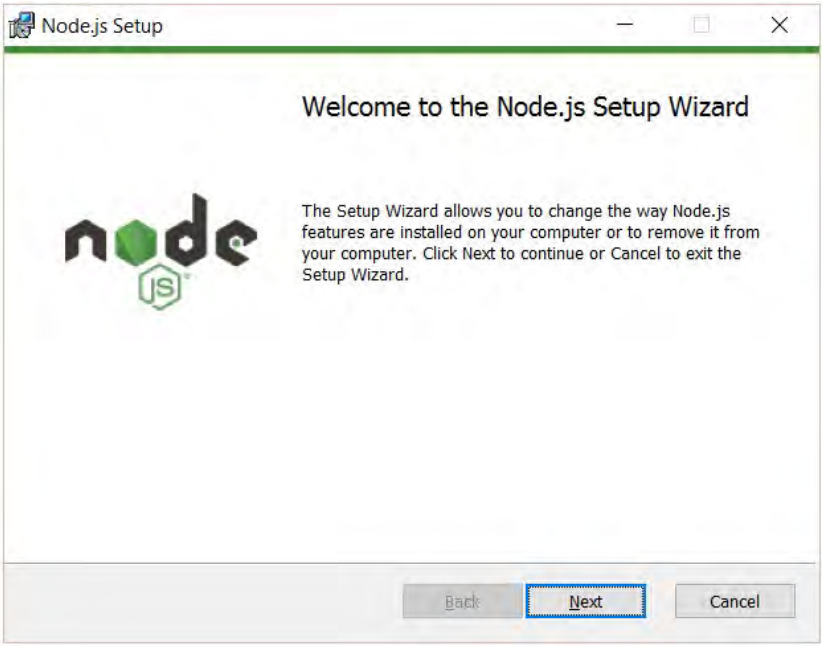

Figura 3.5: Wizard do Node.js

Esta é uma típica instalação *"Next, Next, Finish"*, da qual não será necessário expor aqui todos os passos. A única observação interessante é que ela já adicionará o executável ao PATH do Windows, deixando-o habilitado para ser acessado via prompt de comando.

Após a finalização do Wizard, faremos mais alguns passos para termos certeza de que tudo ocorreu com sucesso. Abra o Prompt de comando (ou o PowerShell) e execute os comandos `node -v` e `npm -v` para, respectivamente, exibir a versão do Node.js e do NPM.

```
node -v
npm -v
```

Você verá algo semelhante à figura a seguir.

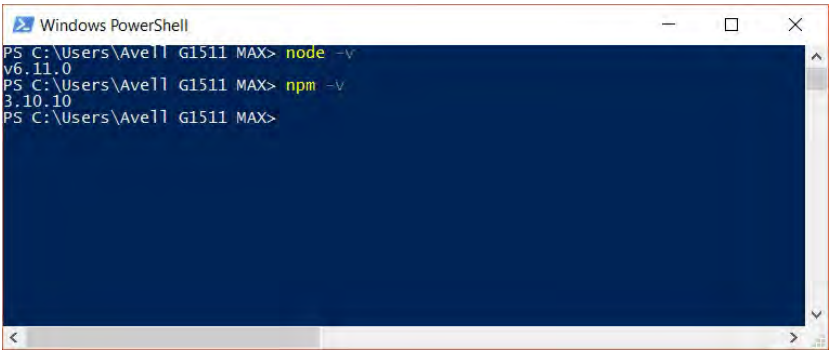

Figura 3.6: PowerShell do Windows

Se você não tiver o perfil de administrador da máquina, poderá baixar o executável do Node.js e configurar o PATH do Windows manualmente. Não mostrarei este processo aqui, mas, caso precise, sugiro a leitura do simpático post *How to install Node.js without admin rights*, de Abdel Olakara (2014).

Instalando Node.js no Ubuntu

A instalação no Ubuntu se dará pelo gerenciador de pacotes *apt* (*Advanced Package Tool*). O site oficial do Node.js mostra os detalhes da instalação. Vou reproduzir os passos aqui. Você pode realizar a instalação com qualquer usuário, porém a senha de root será solicitada.

Primeiramente, entre em um console e execute o comando abaixo.

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Esse comando serve para atualizar o endereço dos repositórios

cujo *apt* procura pelos pacotes.

Olhando mais de perto: o comando `curl` permite obter dados da URL indicada. As diretivas `-sL` são para, respectivamente, fazer o processo em modo silencioso (sem exibir nada na tela), e redirecionar a chamada caso o site aponte para outro endereço.

O `|` (pipe) encaminhará o download para o comando `sudo` (*Super User Do*), que, por sua vez, permitirá executar outros comandos como root. Ele manterá as diretivas do usuário logado (`-E`) e executará o script sem efetivamente salvá-lo no disco (`bash -`). Note que https://deb.nodesource.com/setup_6.x é um script. Não vamos adentrá-lo.

Após o comando, o sistema baixará várias referências e, no final, exibirá algo similar à figura adiante.

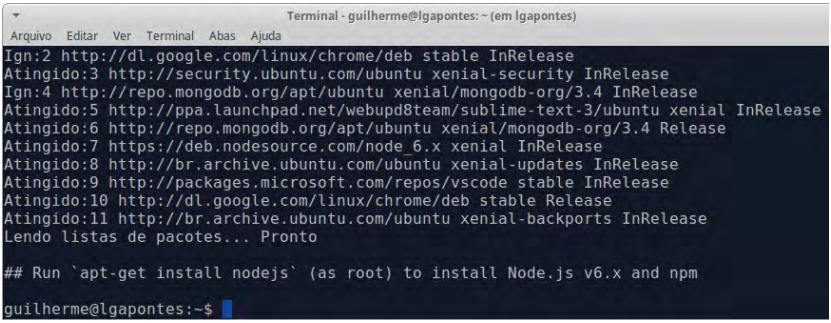

Figura 3.7: Atualizando as referências do apt

O próximo passo é executar a instalação com o comando abaixo. A diretiva `-y` manterá a instalação silenciosa (sem perguntar se você realmente deseja baixar e instalar o pacote `nodejs`).

```
sudo apt-get install -y nodejs
```

O sistema vai baixar tudo e realizar a instalação. Isso pode demorar alguns minutos. Concluído este processo, momento em que você terá o terminal liberado, estamos prontos para executar os comandos para testar a instalação, conforme adiante.

```
node -v
npm -v
```

Eles devem mostrar algo semelhante ao screenshot a seguir:

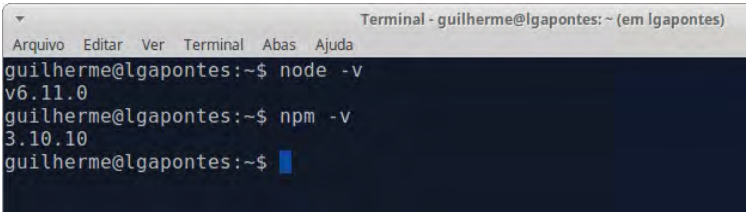

Figura 3.8: Testando a instalação do Node.js

Pronto. Se você não tiver a senha do root, há a possibilidade de executar uma instalação alternativa. Não vou retratá-la aqui, pois já existe um roteiro simples na bíblia dos desenvolvedores: https://stackoverflow.com/questions/31025159/installing-nodejs-without-sudo-in-ubuntu.

Estamos prontos para começar a configuração do projeto.

3.4 CONFIGURAÇÃO MANUAL DO PROJETO

Iniciaremos agora a parte mais importante deste capítulo. Vamos configurar nosso ambiente de desenvolvimento e entender todos os passos de sua arquitetura. Eu realmente acho importante este estudo, mas (há um spoiler aqui) caso você prefira o jeito fácil, vá à penúltima seção deste capítulo. Lá mostrarei uma *alternativa*

pronta para configurar um projeto React (`create-react-app`).

Criando a estrutura básica

Todos os passos executados são compatíveis com os consoles UNIX-like (Ubuntu, Mac OS etc.) ou Windows. Abra seu terminal, acesse um diretório de sua preferência e vamos começar. Particularmente costumo guardar meus projetos em `/opt/repositories/github` . Não se preocupe se o seu for diferente.

Crie e acesse o diretório do projeto com os seguintes comandos:

```
mkdir behappywith.me
cd behappywith.me/
```

Este será o diretório principal de nossa aplicação progressiva. Você verá que, além do código React, vamos esbarrar em outras configurações futuras. Por conta disso, não vamos construir a parte do React na raiz. Vamos criar outros dois diretórios chamados `front-end` e `back-end` , conforme comandos adiante. A parte de *back-end* será abordada no futuro. Nosso foco neste e nos próximos capítulos será o *front-end*.

```
mkdir front-end
mkdir back-end
```

Com os principais diretórios criados, sugiro que você abra a pasta `behappywith.me` pela opção *Open Folder* do seu editor. Veja a seguir um exemplo do projeto no VSCode.

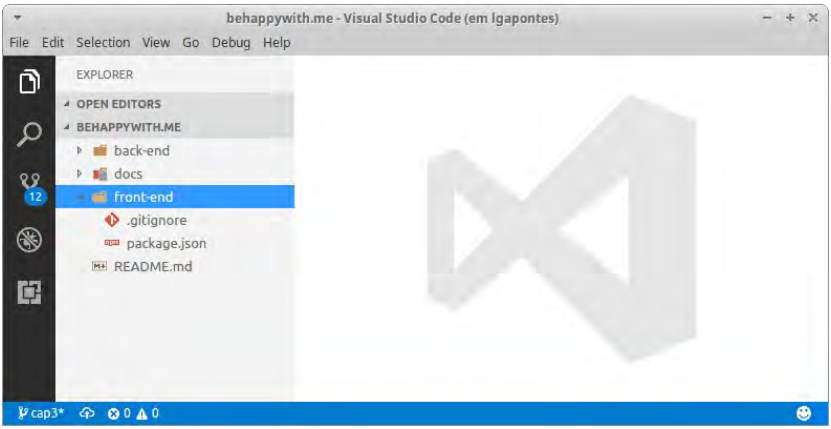

Figura 3.9: Visual Studio Code

Não se apegue aos arquivos `README.md` e `.gitignore`, e à pasta `docs`, todos exibidos na figura. A PWA publicada no GitHub possui tais recursos. Caso você pense em publicar seu projeto em algum repositório git, sugiro que crie um arquivo `.gitignore` semelhante ao publicado no GitHub deste livro (https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/.gitignore). Ele deve ficar dentro da pasta `front-end` para evitar a publicação de códigos de distribuição e outros arquivos de tempo de desenvolvimento.

Criando o arquivo package.json

Vamos à configuração inicial de um projeto Node.js. O NPM oferece um conjunto de comandos para manutenção de um projeto. Um deles é o `init`. Seu comportamento padrão é fazer algumas perguntas inerentes ao projeto e, posteriormente, criar o arquivo `package.json`. Este arquivo mantém todos os dados necessários ao projeto. Execute os comandos a seguir.

```
cd front-end/
npm init –y
```

O parâmetro `-y` diz ao NPM para criar uma configuração default (respondendo *sim* para as perguntas). O console exibirá algo semelhante à figura apresentada a seguir.

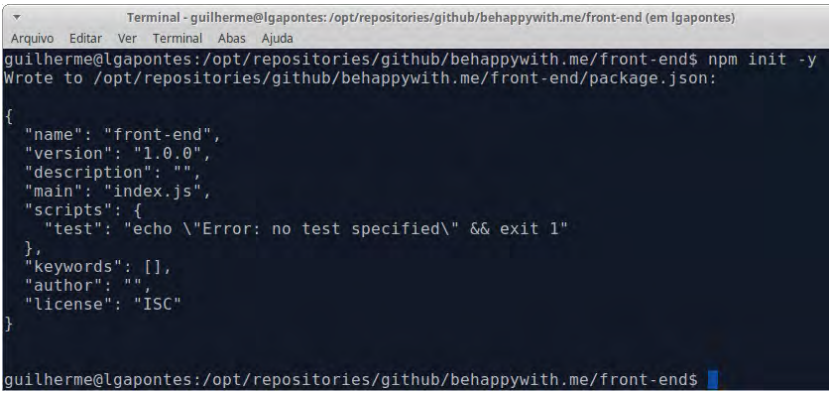

Figura 3.10: package.json criado por default

Abra o arquivo `package.json` criado dentro do diretório `behappywith.me/front-end`. Vamos ajustá-lo pontuando detalhes teóricos quando necessário.

Os campos `"name"`, `"version"` e `"description"` são autodescritivos. O valor `"front-end"` da propriedade `"name"` foi obtido do diretório corrente. Troque-o para `"behappywith.me"`. A versão e a descrição foram definidas com o valor default do comando `init`. Não vamos precisar do campo `"description"`, portanto, apague-o.

O elemento `"main"` serve para especificar o principal script do seu projeto. Ele é muito útil quando vamos publicar o projeto como uma biblioteca no NPM, para que outros desenvolvedores

possam importá-la. Em nosso caso particular, este item não será necessário. Vamos removê-lo.

O item `"scripts"` engloba todos os scripts que serão executados via comando `npm`. Por enquanto, temos apenas um comando chamado `"test"`. Aqui podemos declarar quaisquer scripts necessários ao projeto.

Para executá-los, o `npm` aceita os parâmetros `run nome_script`. Por exemplo, para rodar o script `test`, poderíamos executar:

```
npm run test
```

Por enquanto, se tentarmos executá-lo, receberemos a mensagem de erro: *Error: no test specified*. Isso apenas significa que não temos nenhum script de teste configurado. Um outro detalhe: o NPM já reconhece os comandos `start` e `test` sem a necessidade do prefixo `run`. Então, se executarmos o comando a seguir, o resultado seria o mesmo.

```
npm test
```

Mas de quais scripts vamos precisar? Apenas dois:

1. Um comando `start` para executar o projeto em tempo de desenvolvimento;
2. E o comando `build` para distribuição do projeto para ambiente de produção.

Faremos a configuração de ambos ainda neste capítulo, logo após realizarmos a instalação de todas as dependências necessárias. Por enquanto, apenas remova a referência ao script `"test"`.

Prosseguindo, o item `"keywords"` mantém um array de strings que ajudam na localização de seu projeto no site do NPM. Não vamos precisar disso neste projeto. Remova-o.

O item `"author"` também é autodescritivo. Podemos definilo com uma única sintaxe ou por meio de uma estrutura JSON. Em sintaxe única, deve-se utilizar a notação `"nome <email> (site)"`, conforme exibido adiante.

```
"Guilherme Pontes <email@lgapontes.com> (http://lgapontes.com)"
```

A notação de JSON é semelhante, apenas segregando os valores em campos diferentes. Neste livro, vamos adotar a notação exposta anteriormente. Fique à vontade para alterar esses dados.

Por fim, temos a `"license"`, que, por padrão, foi valorada como `"ISC"`. Não faz parte do escopo deste livro evoluir nesse assunto, mas basicamente a ISC é uma simplificação do texto das licenças MIT e BSD. Caso queira, altere esse valor a seu critério. Como leitura opcional, recomendo um breve comparativo entre as licenças em: https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses.

Terminamos parcialmente. Por enquanto, nosso arquivo `package.json` ficou igual ao código adiante.

```
{
    "name": "behappywith.me",
    "version": "1.0.0",
    "scripts": {},
    "author": "Guilherme Pontes <email@lgapontes.com> (http://lga
pontes.com)",
    "license": "ISC"
}
```

Configurando as dependências

Agora vamos instalar as dependências. Começaremos pelo React. De forma idêntica à execução pelo browser, precisaremos importar as bibliotecas React e React DOM no projeto executado pelo Node.js. Execute o comando a seguir.

```
npm install --save react@15.6.1 react-dom@15.6.1
```

Olhando de perto, o parâmetro `install` serve para baixar e instalar uma ou mais dependências no projeto. Tais dependências podem ser instaladas globalmente ou localmente. As dependências locais ficam centradas em um diretório chamado `node_modules`, localizado na raiz do projeto, que em nosso caso é `behappywith.me/front-end`.

Apesar de esta pasta ser essencial para as coisas funcionarem, não é saudável salvá-la em sua ferramenta de versionamento por dois motivos: ela é gigante (muitos megabytes) e facilmente recuperada através do comando `npm install` (que pode ser executado dentro do diretório do projeto). No caso do git, podemos acrescentar uma referência no `.gitignore` e não precisaremos mais nos preocupar com isso.

Mas se não salvarmos o `node_modules`, como nosso projeto funcionará em outro computador, quando outro usuário baixá-lo do repositório? Muito simples, além de salvar as dependências em `node_modules`, podemos acrescentar suas referências ao `package.json` para que qualquer um que baixe o projeto possa instalá-las pessoalmente pelo comando `npm install`.

Veremos agora onde o arquivo `package.json` guarda essas referências. Ele nos oferece duas propriedades:

1. `dependencies` : indica todas as dependências do projeto. Ou seja, a relação de pacotes apresentada neste array é considerada pré-requisito para seu funcionamento.
2. `devDependencies` : indica as dependências em tempo de desenvolvimento. De forma semelhante às dependências normais, este array guarda todas as dependências necessárias para que o projeto seja desenvolvido.

Mas como podemos acrescentá-las? Manualmente ou por meio de diretivas complementares ao parâmetro `install` . A diretiva `--save` serve para, além de baixar as bibliotecas para `node_modules` , acrescentar os pacotes indicados na propriedade `dependencies` .

Abra novamente o arquivo `package.json` e note que ele foi alterado. A propriedade `dependencies` ficou semelhante ao trecho a seguir:

```
"dependencies": {
    "react": "^15.6.1",
    "react-dom": "^15.6.1"
}
```

O acento circunflexo prefixado na versão da biblioteca indica que a *Minor Version* (versão menor, que na prática é o número do meio) da biblioteca poderá ser acrescida nas instalações futuras deste projeto. Por *acrescida*, entenda que, se executarmos o comando `npm update` , o NPM fará uma verificação das bibliotecas indicadas no `package.json` em busca de novas versões. Neste caso, ele vai respeitar a sintaxe que utilizarmos em `dependencies e devDependencies` .

Ou seja, se deixarmos como está, o NPM poderá atualizar as

bibliotecas `react` e `react-dom` a partir do segundo número. Essa atualização também ocorreria no caso de um usuário baixar o projeto e instalá-lo através de `npm install`.

O prefixo aplicado às dependências em `package.json` nos ajuda a controlar minuciosamente a atualização das bibliotecas. Novos *patches* (melhorias pequenas) serão aceitos se usarmos o `~`, e novas funcionalidades de uma mesma versão (*Minor Version*) serão aceitas com o uso do `^` – ambos prefixados antes da versão.

Existe também a permissividade de alteração da *Major Version*, que não oferece plena garantia de retrocompatibilidade. Recomendo fortemente *não* utilizá-la.

Figura 3.11: Semântica das versões das bibliotecas do NPM

Neste livro, em alguns casos, vamos adotar a versão pontual, sem permissão de atualização. Por conta disso, alteraremos o `package.json` com intuito de restringir o uso do React e React DOM para a versão exata. Altere o trecho de `dependencies` da seguinte forma:

```
"dependencies": {
  "react": "15.6.1",
  "react-dom": "15.6.1"
}
```

Voltando ao comando `npm install`, o sufixo `@15.6.1` aplicado ao nome dos pacotes serve para especificarmos a versão

explicitamente. Por padrão, é acrescentada a permissividade mais abrangente da Minor Version (o símbolo `^`). Para fixar a exata versão no `package.json`, você pode acrescentar a diretiva `--save-exact` em conjunto com a `--save`. Veja um exemplo adiante.

```
npm install --save --save-exact react@15.6.1 react-dom@15.6.1
```

Para fecharmos o entendimento do parâmetro `install`, resta-nos conhecer a diretiva `--save-dev`. Assim como a `--save`, a `--save-dev` vai alterar o `package.json`, com a diferença de que as referências serão acrescentadas à propriedade `devDependencies`.

Algumas bibliotecas só são necessárias enquanto estivermos desenvolvendo. Quer um exemplo? O transpiler e os presets do Babel. Falando neles, vamos instalá-los agora.

```
npm install --save-dev babel-core@6.25.0 babel-preset-es2015@6.24
.1 babel-preset-react@6.24.1 babel-loader@7.1.0
```

Note que fizemos uma instalação de dependências para tempo de desenvolvimento (`--save-dev`). Como vimos, esses pacotes estarão na propriedade `devDependencies` do seu `package.json`.

Mas afinal, para que serve cada um dos pacotes instalados? O **babel-core** é o responsável pela compilação do JavaScript. É ele que vai gerar nosso ECMAScript compatível com o navegador de acordo com as configurações que vamos definir com os presets.

Por *preset*, entenda como uma biblioteca do Babel para viabilizar um tipo especial de transpiler. O **babel-preset-es2015**, por exemplo, serve para transformar o ECMAScript 6 em

ECMAScript 5. Seguindo essa mesma lógica, fica fácil deduzir que o **babel-preset-react** é o responsável pela compilação do código JSX. O **babel-loader** é o artefato integrador entre o Webpack e o Babel.

Depois que o download terminar, você verá algo semelhante à figura a seguir.

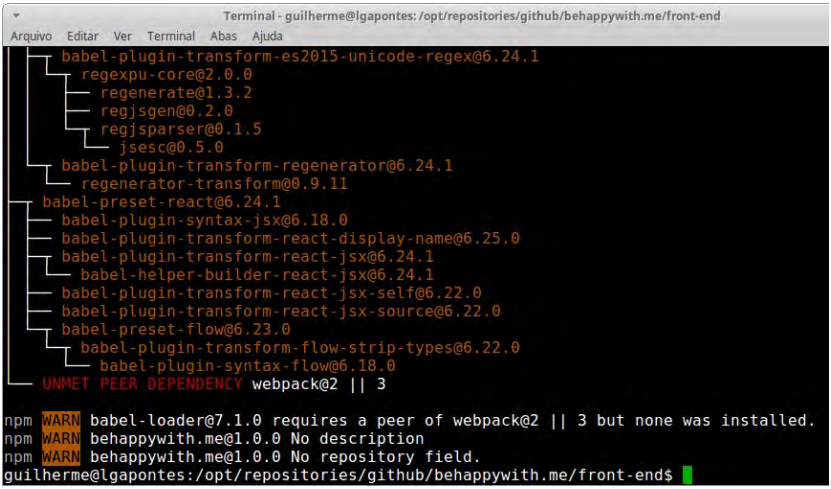

Figura 3.12: Resultado do comando NPM

Ficou assustado com a saída? Quem é habituado ao uso do NPM deve ter estranhado muito o resultado verboso e cheio de avisos. Vamos analisar aqui linha a linha para lhe deixar mais tranquilo.

As primeiras linhas são normais. Apenas indicam em formato de árvore os pacotes que instalamos. Não é necessário discutir todas essas dependências. A última linha da árvore assusta, até porque aparece em vermelho.

```
└── UNMET PEER DEPENDENCY webpack@2 || 3
```

Sua dedução está certa: isso é um erro (ou um aviso grave, por assim dizer). O que está acontecendo é o seguinte: o **babel-loader** é quem integrada o Babel com o Webpack. Ele funciona como um *Pacote Plugin* que precisa estar conectado aos *pacotes hosts* Babel e Webpack.

O NPM possui um tipo de dependência, conhecida como *Peer Dependencies* (dependência de pares), que nos permite especificar quais são os *pacotes hosts* (cuja tradução próxima seria *pacotes hospedeiros*) que um determinado *pacote plugin* está associado. Por *pacote plugin*, entenda aquele que **deve** ser *plugado* a um *pacote host* para funcionar corretamente. Então, ratificando, uma *Peer Dependency* não significa que o *pacote plugin* **necessita** de outro pacote, e sim que ele **deve** ser plugado a um *pacote host* de determinada versão.

O NPM **não** traz as *Peer Dependencies* automaticamente. Como começamos a instalação pelo Babel e, em momento algum, indicamos o Webpack, o NPM nos avisa que o **babel-loader** é um *pacote plugin* que ainda não está casado com seu *pacote host*.

Para evitar esse tipo de mensagem, você pode simplesmente instalar todos os *pacotes hosts* durante o mesmo comando `npm install` . Para fins didáticos, optei em instalar os pacotes e plugins separadamente.

Continuando nossa análise da saída do NPM, temos também a linha de aviso ratificando a necessidade do Webpack:

```
npm WARN babel-loader@7.1.0 requires a peer of webpack@2 || 3 but
 none was installed.
```

Por fim, as duas linhas finais nos avisam que nosso `package.json` não possui descrição e não contém o repositório cujo pacote está associado, respectivamente.

```
npm WARN behappywith.me@1.0.0 No description
npm WARN behappywith.me@1.0.0 No repository field.
```

Para fins da criação da PWA neste livro, ambas as propriedades podem ser desconsideradas. Vamos continuar com as dependências de desenvolvimento, seguindo agora com a instalação do Webpack.

```
npm install --save-dev webpack@3.0.0 webpack-dev-server@2.5.0
```

Como você já sabe, o Webpack nos ajudará a otimizar o código final, juntando arquivos e reduzindo (*minify*) tudo o que for possível. Ele também orquestrará as compilações do Babel e seus presets.

O **webpack-dev-server** é um simplório servidor web para rodar um projeto configurado no Webpack em tempo de desenvolvimento. Na prática, o que ele faz é compilar os arquivos e publicá-los na porta 8080. Obviamente não vamos utilizá-lo em produção.

Concluímos assim a instalação das dependências. O próximo passo é a configuração do Webpack.

Primeiros passos com o Webpack

A parte mais complicada do Webpack é a configuração. É claro, *complicada* em relação à instalação, que foi muito simples. Basicamente vamos configurá-lo através de um arquivo `webpack.config.js` na raiz do projeto (no nosso caso, na pasta

`front-end`). Para que todos os passos fiquem claros, vamos iniciar pela configuração simples e os embasamentos necessários.

Um projeto organizado pelo Webpack fará a compilação dos arquivos fontes para torná-los compatíveis com os navegadores. Efetivamente, vamos precisar das pastas `src` e `dist`, para respectivamente guardar os fontes e os arquivos de distribuição para produção.

Além de compilar, o Webpack também vai juntar os arquivos JavaScript, organizar as imagens, minificar o CSS e injetar scripts no HTML. Então, dentro de `src`, também precisaremos de pastas e arquivos específicos para fornecer esses insumos. Resumindo, o que queremos é uma estrutura conforme a figura destacada a seguir.

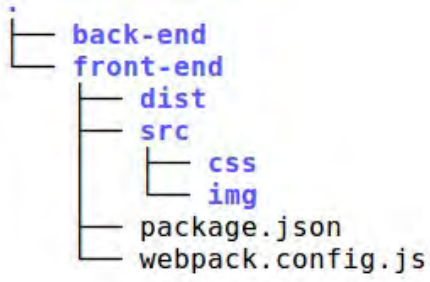

Figura 3.13: Diretórios e arquivos da PWA

Omiti o diretório `node_modules` para não poluir a representação. É preciso destacar que vamos evoluir essa estrutura ao longo do livro.

Um detalhe: observe as pastas `css` e `img` na raiz do `src`. Como sabemos, o React é um framework baseado em

componentes. Todos os componentes devem ser autossuficientes no que diz respeito à apresentação dos dados, inclusive considerando elementos de estilo (CSS) e imagens.

Os diretórios que criaremos aqui são para guardar as imagens e estilos globais do projeto. Eventualmente, teremos outras pastas `img` e `css` inerentes aos componentes. Então, no Linux, pelo console, podemos fazer isso apenas com o comando seguinte (supondo que você esteja em `behappywith.me/front-end/`).

```
mkdir -p src/css src/img dist
```

Se você estiver no Windows, vai precisar de, pelo menos, uns 10 cliques no mouse.

Com a estrutura pronta, vamos criar o arquivo `webpack.config.js` no diretório `front-end` do projeto (crie-o com sua IDE). Como vimos, este é o arquivo de configuração do Webpack. Ele segue um padrão de arquivo JavaScript muito comum em projetos Node.js. Caso não esteja acostumado com isso, não se preocupe. Vamos editá-lo passo a passo.

Comece-o com as linhas:

```
const webpack = require('webpack');
const path = require('path');
```

Estes comandos são utilizados em projetos Node.js para importar bibliotecas. Vou resumir o comportamento que a função `require()` faz, nesta sequência, para encontrar os pacotes:

1. Primeiramente, ela faz uma busca localmente;
2. Em seguida, no core do Node.js;
3. Depois, no diretório `node_modules` local;

4. Por fim, nos diretórios da instalação global do Node.js.

Nestas importações apresentadas, o `webpack` foi obtido da pasta `node_modules` e o `path` do core do Node.js. O `path` é uma biblioteca que nos ajudará na definição de caminhos no disco rígido.

Vamos acrescentar agora uma sintaxe para agrupar todas as configurações do arquivo. Inclua o trecho a seguir após os comandos de importação.

```
module.exports = {
};
```

A sintaxe `module.exports` é um recurso do Node.js usado para deixar todas as suas propriedades (que estiverem dentro do objeto `{}`) acessíveis de fora desse arquivo.

O próximo passo é informar ao Webpack o local onde estarão nossos arquivos-fontes. Na verdade, indicaremos um arquivo centralizador, e o Webpack encontrará todos os arquivos requisitados por ele. Isso é realizado pela propriedade `entry` , conforme visualizado a seguir.

```
module.exports = {
   entry: path.join(__dirname, 'src/index.jsx')
};
```

Note que estamos utilizando uma função `join()` da biblioteca `path` . Ela simplesmente junta o diretório corrente com a string `src/index.jsx` , resultando em uma indicação ao arquivo principal da aplicação localizado em `behappywith.me/front-end/src/index.jsx` . Já que citamos o arquivo, crie-o pela sua IDE e acrescente o conteúdo exposto no código a seguir.

```
import React from 'react'
import ReactDOM from 'react-dom'

ReactDOM.render(
   <h1>Bem-vindo ao React!</h1>,
   document.querySelector("#main")
)
```

Neste exemplo, simplificamos o trecho React. Em vez de explicitamente criar um componente, apenas retornamos uma sintaxe JSX no `render()` do ReactDOM. Há também pequenas diferenças no `id` do elemento cujo React será injetado (que, em vez de `exemplo`, como na execução pelo browser, passou a ser chamado de `main`) e na importação do React e do ReactDOM com a sintaxe do ECMAScript 6.

A nova versão do JavaScript redesenhou muito bem a sintaxe de importação dos arquivos. Até o ES5, os módulos Node.js precisavam da sintaxe `require()`. Agora, quando optamos em trabalhar com o ES6, podemos adotar a sintaxe simplificada dos comandos `import e from`.

MAS POR QUE USAMOS `REQUIRE()` NO `WEBPACK.CONFIG.JS`?

Usar `require()` no `webpack.config.js` não foi desleixo de nossa parte. O Babel é o componente que compila o ES6. Ele faz isso com ajuda do Webpack. Se colocássemos comandos ES6 no arquivo de configuração que o Webpack utiliza para chamar o Babel e compilar tudo, quem compilaria o arquivo de configuração? É um paradoxo!

Se configurássemos o Node.js para compilar o arquivo de configuração do Webpack e, somente depois, utilizá-lo para compilar os demais arquivos, resolveríamos o problema. Particularmente, prefiro o jeito simples: respiro fundo e escrevo 20 linhas em ECMAScript 5. O resto, fazemos com o ECMAScript 6.

Agora que já sabemos onde nosso código JSX será injetado (`#main`), que tal criar o arquivo HTML? Crie um arquivo chamado `index.html` na raiz da pasta `dist` e implemente o conteúdo a seguir.

```
<!DOCTYPE html>
<html>
   <head>
       <meta charset="utf-8">
       <link rel="shortcut icon" href="img/favicon.ico">
       <title>behappywith.me</title>
   </head>
   <body>
       <div id="main"></div>
       <script src="bundle.js"></script>
   </body>
</html>
```

Veja que estamos fazendo referência à imagem *favicon* da página. Caso não saiba, a imagem *favicon* é utilizada pelo navegador para customizar o ícone da aba cuja página estará aberta. Se você não especificar nenhuma, será usada uma imagem padrão, que geralmente é o próprio ícone do navegador. Veja na figura adiante um exemplo de como ficará nossa página e o ícone que vamos adotar.

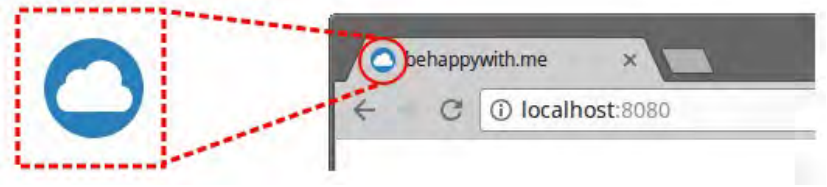

Figura 3.14: favicon circulado de vermelho

Baixe essa imagem pelo link http://behappywith.me/img/favicon.ico, e salve-a na pasta `behappywith.me/front-end/src/img`.

Mas espere um pouco, o arquivo HTML não deveria estar na pasta `src`? O Webpack fará a cópia do arquivo de `src` para `dist` somente após utilizarmos o plugin **html-webpack-plugin** (que será configurado adiante). Como o Webpack não foi *orientado* a copiar o arquivo, vamos temporariamente deixá-lo já publicado na pasta destino.

Note também que definimos uma chamada estática ao arquivo `bundle.js`. Isso não será necessário no futuro. Não se preocupe, tudo ficará claro em breve.

Voltando à configuração, o próximo passo é especificar o diretório final cujos arquivos compilados e minificados serão

salvos. Utilize a propriedade `output` para realizar essa configuração. Ajuste `webpack.config.js` conforme a seguir.

```
module.exports = {
   entry: path.join(__dirname, 'src/index.jsx'),
   output: {
       path: path.join(__dirname, 'dist'),
       filename: 'bundle.js'
   }
};
```

Dentro da propriedade `output`, temos outras duas propriedades:

1. `path`: local onde de fato o arquivo agrupador será salvo. O `join()` apontará para o diretório `dist` dentro de `front-end`.
2. `filename`: este é o nome do arquivo resultante da integração de todos os arquivos utilizados. Optei por chamá-lo de `bundle.js` (a tradução seria `agrupar.js`) para manter uma proximidade com a documentação oficial do Webpack, que utiliza exatamente este nome de arquivo.

A pasta `dist` manterá todos os arquivos necessários para a execução do projeto. Em outras palavras, será neste diretório que o Webpack vai jogar todos os arquivos (HTML, CSS, JavaScript etc.) visíveis aos usuários da PWA.

Já sabemos onde estão os arquivos-fontes e a pasta destino. Isso significa que todas as referências encontradas a partir do arquivo `index.jsx` serão consideradas como parte do sistema. Ou seja, quando criarmos nossos componentes, o Webpack vai esperar referências diretas aos arquivos JavaScript oriundas de `index.jsx` – algo como `import Componente from`

`'./componente'`, em que `componente` é um diretório que guarda um arquivo `index.js` em seu interior.

Por padrão, a importação automática, de arquivos `index.js`, espera apenas arquivos com extensão `.js`. Vamos configurar o Webpack para aceitar também extensões `.jsx`, que são mais adequadas aos arquivos React.

Para isso, podemos utilizar a propriedade `extensions`. Acrescente o trecho destacado a seguir logo após a sintaxe do `output`.

```
module.exports = {
   entry: path.join(__dirname, 'src/index.jsx'),
   output: {
       path: path.join(__dirname, 'dist'),
       filename: 'bundle.js'
   },
   resolve: {
       extensions: [".js",".jsx"]
   }
};
```

Pronto. Dessa forma, podemos referenciar diretórios com arquivos `.js` e `.jsx` de forma equivalente.

Vamos agora definir as regras de transformação utilizadas. Em nosso caso, trabalharemos com o **babel-loader**, que engloba o Babel e seus presets do React e ECMAScript 2015 (ECMAScript 6). Acrescente a configuração sob a propriedade `module`, como mostrado a seguir. As propriedades `entry` e `output` foram omitidas.

```
module.exports = {
   // código omitido ...
   module: {
       rules: [
```

```
        {
            test: /.jsx?$/,
            exclude: /node_modules/,
            include: path.join(__dirname, 'src'),
            use: [
            {
                loader: 'babel-loader',
                options: {
                    presets: ['es2015', 'react']
                }
            }
            ]
        }
        ]
    }
};
```

O `module` é um objeto agrupador dos módulos cujas regras são especificadas pelos `rules`. Por enquanto, temos uma regra para compilar todos os arquivos JSX. As propriedades `test` e `include` recebem expressões regulares para informar ao Webpack a extensão e o diretório dos arquivos origem, respectivamente.

A propriedade `exclude` também recebe uma expressão regular, porém para evitar arquivos ou diretórios. Se as condições forem atendidas, a propriedade `use` realizará os passos da transformação de acordo com as especificações. Para este caso, usaremos o `loader` do Babel com os presets `es2015` e `react`.

Já temos as configurações iniciais prontas. Resta-nos informar ao **webpack-dev-server** que o código final estará no diretório `dist`. Para isso, inclua o trecho adiante após a propriedade `modules`.

```
module.exports = {
    // códigos omitidos ...
    devServer: {
```

```
        publicPath: "/",
        contentBase: "./dist"
    }
};
```

Finalmente chegamos à nossa primeira versão do `webpack.config.js`. Vamos rodar o projeto pelo comando a seguir. Como não instalamos o servidor de desenvolvimento globalmente, precisaremos acessá-lo internamente no `node_modules`. Você deve estar no diretório `front-end`.

```
node_modules/webpack-dev-server/bin/webpack-dev-server.js
```

Logo de cara, o console nos mostra algumas ações realizadas.

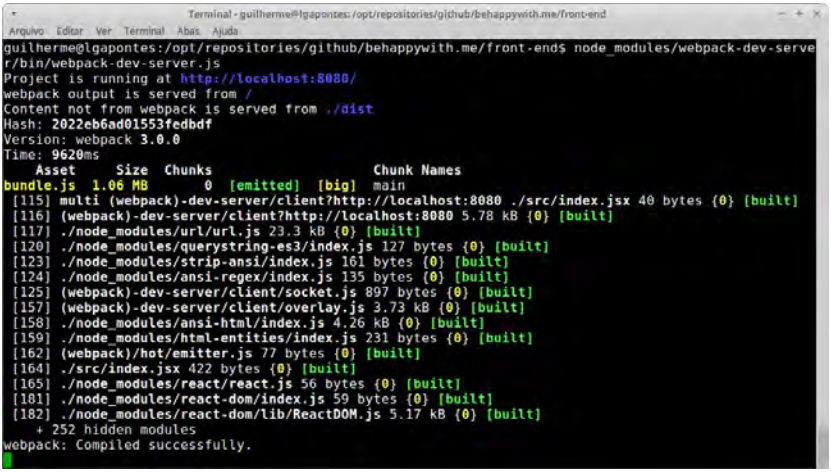

Figura 3.15: Resultado da execução no console

Esses foram os passos realizados pelo Webpack (internamente chamado pelo **webpack-dev-server**). A única observação aqui é o tamanho do `bundle.js` : **1.06 MB**. Não se preocupe, este `bundle.js` ainda não está otimizado. Vamos precisar acrescentar alguns plugins no arquivo de configuração do Webpack para

melhorar isso.

Por enquanto, apenas acesse o endereço http://localhost:8080 e veja nossa primeira execução do React via Node.js. Segue a screenshot da página carregada:

Figura 3.16: Resultado da primeira versão

Perfeito. Nossos próximos passos melhorarão a arquitetura proposta para o projeto, tornando os arquivos resultantes otimizados para uma PWA.

Segunda versão da configuração do Webpack

Lembra que colocamos o arquivo HTML direto no `dist`? Vamos agora tratar dessa má prática com a instalação de um plugin que criará o arquivo HTML. Aperte `Ctrl+C` (para finalizar o servidor) no console e, supondo que você ainda esteja na pasta `front-end`, digite o seguinte comando:

```
npm install --save-dev html-webpack-plugin@2.29.0
```

O **html-webpack-plugin** é responsável pela geração automática do arquivo HTML do projeto. Como vamos fazer

ajustes finos nesse arquivo no futuro (para atender aos requisitos básicos de uma PWA), não vamos deixar que o plugin crie um arquivo do zero. Em vez disso, informaremos um *exemplo* de arquivo HTML, e o Webpack simplesmente criará uma cópia na pasta `dist` com a injeção dos scripts necessários.

Mova o arquivo `index.html` da pasta `dist` para a pasta `src` e retire a chamada ao script `bundle.js`. No final, seu arquivo deve ficar igual ao apresentado no código a seguir.

```
<!DOCTYPE html>
<html>
   <head>
       <meta charset="utf-8">
       <link rel="shortcut icon" href="img/favicon.ico">
       <title>behappywith.me</title>
   </head>
   <body>
       <div id="main"></div>
   </body>
</html>
```

Além de não precisarmos apontar o script `bundle.js`, as coisas ficaram mais organizadas: todos os arquivos-fontes na pasta `src` e a pasta `dist` vazia.

Agora voltaremos ao arquivo de configuração do Webpack. Acrescentar plugins é muito fácil: basta importarmos o pacote e criarmos uma propriedade chamada `plugins`. O primeiro passo é simplesmente fazer a importação. Acrescente-a abaixo das outras importações, conforme exposto adiante.

```
const webpack = require('webpack');
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

// demais códigos ...
```

Agora vamos acrescentar o array `plugins`. Dentro dele, vamos instanciar um novo objeto `HtmlWebpackPlugin` com um parâmetro (JSON) de configuração. Veja a seguir o trecho deste código. Todas as demais propriedades foram omitidas.

```
module.exports = {
   // entry, output e resolve ...

   plugins: [
       new HtmlWebpackPlugin({
           filename: 'index.html',
           template: path.join(__dirname, 'src/index.html')
       })
   ],

   // module e devServer
};
```

A propriedade `filename` indica o nome do arquivo HTML que será gerado no `output` do Webpack (pasta `dist`). A propriedade `template` indica a localização do arquivo que o plugin usará como exemplo. Na prática, o Webpack vai copiar esse arquivo e acrescentar a chamada ao `bundle.js`.

Veja a seguir como está nosso arquivo `webpack.config.js`.

```
const webpack = require('webpack');
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
   entry: path.join(__dirname, 'src/index.jsx'),
   output: {
       path: path.join(__dirname, 'dist'),
       filename: 'bundle.js'
   },
   resolve: {
       extensions: [".js",".jsx"]
   },
   plugins: [
```

```
        new HtmlWebpackPlugin({
            filename: 'index.html',
            template: path.join(__dirname, 'src/index.html')
        })
    ],
    module: {
        rules: [
        {
            test: /.jsx?$/,
            exclude: /node_modules/,
            include: path.join(__dirname, 'src'),
            use: [
            {
                loader: 'babel-loader',
                options: {
                    presets: ['es2015', 'react']
                }
            }
            ]
        }
        ]
    },
    devServer: {
        publicPath: "/",
        contentBase: "./dist"
    }
};
```

Execute novamente o servidor de desenvolvimento.

```
node_modules/webpack-dev-server/bin/webpack-dev-server.js
```

Acesse a URL http://localhost:8080 e veja que tudo permanece funcional, com a diferença de que o HTML agora faz parte do fonte do projeto.

Criando os scripts de execução do package.json

Antes de instalar o próximo plugin, vamos facilitar um pouco as próximas execuções do servidor de desenvolvimento. Lembra da propriedade `scripts` do `package.json`? O NPM oferece-a para

casos como esse.

Muitas vezes, são necessários vários passos de execução para efetivamente publicar um projeto Node.js. Criar rotinas automatizadas no `package.json` é uma boa prática. Faremos isso agora.

Abra o arquivo `package.json` e inclua a seguinte linha. A sintaxe `start` deve ser utilizada para apontar para o script de inicialização da aplicação. No nosso caso, como isso será feito pela execução do `webpack-dev-server`, vamos simplesmente referenciá-lo.

```
"start": "node_modules/webpack-dev-server/bin/webpack-dev-server.
js"
```

O trecho `scripts` do arquivo `package.json` está exposto a seguir. As outras configurações foram omitidas.

```
{
 // ...
 "scripts": {
   "start": "node_modules/webpack-dev-server/bin/webpack-dev-serv
er.js"
 },
 // ...
}
```

Agora, sempre que quisermos rodar o servidor, precisamos apenas executar o comando a seguir.

```
npm start
```

Avante!

Terceira versão da configuração do Webpack

Até o momento, só executamos o projeto em tempo de desenvolvimento. Quando formos de fato publicar nossa PWA, precisaremos compilar e copiar os arquivos para o diretório `dist`. Observe, entretanto, que mesmo após rodarmos algumas vezes o **webpack-dev-server**, a pasta `dist` continua vazia. O servidor de desenvolvimento não vai contaminá-la com suas versões preliminares da compilação.

Para efetivamente produzirmos os arquivos finais, precisaremos executar o comando `webpack`. Supondo que você esteja na pasta `front-end`, execute o comando:

```
node_modules/webpack/bin/webpack.js
```

Não vou reproduzir a saída do console aqui, mas note que ela é muito próxima da que vimos na execução do servidor. Note também que o arquivo `bundle.js` já foi reduzido para 748 KB. Já é um progresso, mas vamos melhorar mais, no futuro.

Abra a pasta `dist` e veja que `bundle.js` e `index.html` foram copiados, mas não há nem um sinal da imagem `favicon.ico`.

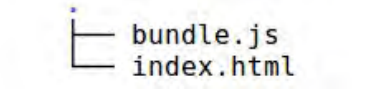

Figura 3.17: Conteúdo da pasta dist

O Webpack só vai copiar os arquivos de imagens e sua respectiva pasta `img` se o configurarmos para isso. O próximo passo resolverá esse impasse. Vamos instalar o plugin **file-loader**. Supondo que você esteja no diretório `front-end`, execute:

```
npm install --save-dev file-loader@0.11.2
```

A configuração deste tipo de plugin é um pouco diferente. Vamos criar uma regra que defina quais arquivos serão copiados de `src` para `dist` sem intervenção do Webpack. É isso mesmo: *sem intervenção*. Neste caso, o Webpack só precisa copiar os arquivos. A otimização dos arquivos de imagem em si deve ser tratada por nós. Faremos isso nos próximos capítulos.

A essência dessa regra gira em torno da identificação dos arquivos que desejamos copiar, tais como `.jpeg` ou `.png`, e a referência ao plugin `file-loader`. Isso pode ser feito com a seguinte sintaxe.

```
{
   test: /\.(jpe?g|ico|png|gif|svg)$/i,
   loader: 'file-loader?name=img/[name].[ext]'
}
```

Abra o arquivo `webpack.config.js` e acrescente este trecho após a regra do `babel-loader`:

```
module.exports = {
   // ...
   module: {
       rules: [
       {
           test: /.jsx?$/,
           exclude: /node_modules/,
           include: path.join(__dirname, 'src'),
           use: [
           {
               loader: 'babel-loader',
               options: {
                   presets: ['es2015', 'react']
               }
           }
           ]
       },
       {
           test: /\.(jpe?g|ico|png|gif|svg)$/i,
```

```
            loader: 'file-loader?name=img/[name].[ext]'
        }
        ]
    },
    // ...
};
```

Observe que não foi necessário realizar um `require()` explícito no início do arquivo. O Webpack sabe o que fazer por conta da sintaxe `file-loader` definida na propriedade `loader`. De forma semelhante à outra regra, há também a propriedade `test`. Essa regra captura (por expressão regular) todos os tipos de imagens que vamos utilizar e solicita ao Webpack que as salve como a mesma extensão no destino `img`.

A configuração está pronta, mas, se você executar o Webpack novamente, verá que a imagem ainda não será copiada. Isso se dá porque o Webpack parte da premissa que todos os arquivos usados no projeto serão chamados implícita ou explicitamente pelo `entry` configurado (que é o arquivo `index.jsx`). Como nosso arquivo não faz menção à imagem, o Webpack não a carregará e, consequentemente, não aplicará a regra exposta anteriormente.

Para resolver esse problema, apenas acrescente uma nova importação no arquivo `behappywith.me/front-end/src/index.jsx`.

```
import './img/favicon.ico';
```

Você nem precisa usá-lo. O simples fato de importá-lo já tornará a cópia possível. Feito isso, vamos mais uma vez executar o comando de compilação (a partir da pasta `front-end`).

```
node_modules/webpack/bin/webpack.js
```

Veja que nossa imagem foi copiada a contento.

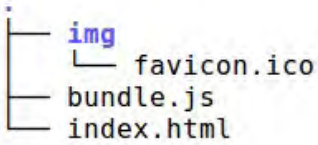

Figura 3.18: Conteúdo da pasta dist

A partir de agora, as imagens serão copiadas.

Configurando o build pelo package.json

De forma idêntica ao que fizemos com o **webpack-dev-server**, para facilitar as compilações futuras, vamos criar uma entrada `build` no `package.json`. Coloque-a abaixo da sintaxe `start` e defina a chamada do script `webpack.js`. Após esse ajuste, a seção de scripts no `package.json` ficará da seguinte forma:

```
{
 // ...
 "scripts": {
   "start": "node_modules/webpack-dev-server/bin/webpack-dev-serv
er.js",
   "build": "node_modules/webpack/bin/webpack.js"
 },
 // ...
}
```

Há um ponto de atenção aqui: o comando `npm build` **não** chamará o script que criamos. O NPM possui um comando interno chamado `build` para construir recursos nativos por meio do `node-gyp`. Não precisamos nos aprofundar, mas basicamente `npm build` é um comando padrão do NPM. No nosso caso, devemos executar o script `build` por:

```
npm run build
```

Agora temos nossos atalhos `start` e `build` prontos.

Quarta versão da configuração do Webpack

Com a configuração de cópia das imagens, resta-nos tratar dos arquivos CSS. Primeiramente baixe o core do Pure.css (arquivo `pure-min.css`) pelo link https://unpkg.com/purecss@1.0.0/build/pure-min.css e salve-o na pasta `css`.

Como vimos no capítulo anterior, o Pure.css é muito otimizado. Vamos utilizar seus estilos em todo o projeto, e por isso ele ficará neste diretório. Além do Pure.css, teremos também uma estilização própria. Crie um arquivo chamado `index.css` na pasta `css` e acrescente o conteúdo apresentado adiante.

```
#main {
   border: 0;
   padding: 0;
   margin: 0;
}
h1 {
   color: red;
}
```

Esse estilo apenas força a `<div id="main">` a zero pixels nas propriedades `border`, `padding` e `margin`. Acrescentamos também uma cor vermelha ao `<h1>`, para explicitar a aplicação do estilo no nosso próximo teste. No futuro, colocaremos definições globais nesse arquivo.

O próximo passo é configurar o Webpack para reconhecer esses arquivos. Para isso, vamos utilizar os plugins **style-loader** e **css-loader**. Instale-os pelo comando a seguir.

```
npm install --save-dev style-loader@0.18.2 css-loader@0.28.4
```

O pacote **css-loader** obtém os arquivos CSS pela funcionalidade `require()` e resolve sua sintaxe `import` e `url()`. O pacote **style-loader** trabalha com o CSS obtido e aplica-o na página. Esses pacotes têm suas responsabilidades separadas, mas devem ser utilizados em conjunto para efeito real no projeto.

Para fazê-los funcionarem, vamos acrescentar uma regra no `webpack.config.js`. De forma semelhante às outras `rules`, o primeiro passo é identificar quais arquivos serão capturados pela regra. Como devemos apontar apenas para os arquivos de estilo, de extensão `.css`, podemos tratar desta captura por meio da seguinte sintaxe:

```
test: /\.css$/,
```

Agora, de posse dos arquivos CSS, avisaremos ao Webpack para aplicar os plugins **style-loader** e **css-loader** com o seguinte trecho:

```
use: [
   { loader: "style-loader" },
   { loader: "css-loader" }
]
```

Com os conceitos esclarecidos, abra o arquivo `webpack.config.js` e adicione os trechos supracitados no array `rules`, conforme o código a seguir:

```
module.exports = {
   // ...
   module: {
       rules: [
       // ...,
       {
           test: /\.css$/,
           use: [
               { loader: "style-loader" },
```

```
                { loader: "css-loader" }
            ]
        }
        ]
    },
    // ...
};
```

O restante do arquivo foi omitido. A ideia é semelhante à das outras regras. Há uma expressão regular para pegar os arquivos e, caso seja bem-sucedida, distribuí-los juntamente com o projeto.

Vamos testar? Antes de executar o servidor de desenvolvimento, preciso ressaltar um detalhe importante: se houver arquivos na pasta `dist`, o **webpack-dev-server** não vai obter as atualizações feitas no `src`. Então, sempre apague o conteúdo de `dist` antes de voltar a nossa rotina de desenvolvimento, ok?

Agora sim, execute esses novos ajustes pelo servidor de desenvolvimento com o comando seguinte.

```
npm start
```

Vejamos o resultado em http://localhost:8080.

Figura 3.19: Exemplo de estilo que NÃO funcionou

Por que não ficou vermelho? Lembre-se de que Webpack faz referência explícita apenas aos arquivos configurados na propriedade `entry`, que, no nosso caso, é o `index.jsx`. Para ele encontrar os demais arquivos, teremos de importá-los dentro do `index.jsx` (da mesma forma que fizemos com o `favicon.ico`).

Vou aproveitar também para mostrar que o **webpack-dev-server** não precisa ser desligado para perceber alterações na pasta `src`. Como precisamos importar os arquivos de CSS para *dentro* do projeto, o simples fato de importá-los em `index.jsx` já será suficiente.

Vamos fazer essa pequena alteração **sem** interromper o servidor. Abra o arquivo `index.jsx` e altere-o conforme a seguir:

```
import './css/index.css';
import './css/pure-min.css';
```

Note que seu console mostrará mensagens de recompilação dos arquivos. Acesse novamente http://localhost:8080 e veja o resultado.

Figura 3.20: Exemplo de estilo que funcionou

Perfeito, a fonte do `<h1>` ficou vermelha! Feche o servidor (`Ctrl+C`) e vamos agora publicar os fontes na pasta `dist` com o comando adiante.

```
npm run build
```

Abra a pasta `dist` e note que o arquivo de estilo ainda não foi copiado.

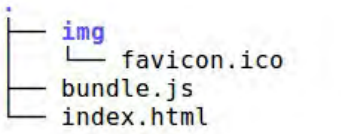

Figura 3.21: Pasta dist SEM o CSS

Mas por quê? Propositalmente, conduzi essa explicação por uma configuração mais simples para que você entenda o seguinte: o Webpack obteve os arquivos físicos do CSS porque os importamos dentro do *componente* `index.jsx` .

Como um componente é autossuficiente, o Webpack entende que o estilo deve ficar *inline* em sua apresentação. Neste caso, o *inline* fica dentro do `bundle.js` . Em outras palavras: o CSS está implícito lá dentro do código JavaScript.

Você deve estar achando isso muito bizarro. Certo, eu também acho. Mas independente do que achamos, na prática, ter CSS no JavaScript é bom ou ruim? Vamos analisar essa questão com muito cuidado.

Do ponto de vista do *overhead* da cópia dos arquivos do servidor web para o navegador, ter um arquivo um pouco maior é melhor do que dois arquivos separados. Ou seja, em vez de

aplicarmos um pouco de *overhead* para cada download, o browser só fará um download e pronto. Ok, esta lógica confere.

Entretanto, quando se trata de arquivos CSS embutidos dentro do JavaScript, o cenário muda um pouco. No desenvolvimento de páginas web, geralmente postergamos o download dos scripts para o final da página propositalmente para evitar que o navegador perca muito tempo nessa tarefa.

Não vamos entrar no mérito se isso é certo ou errado. Entenda apenas que baixar o JavaScript geralmente é mais custoso do que baixar o CSS. Isso acontece porque, entre outras minúcias, os arquivos de script são maiores do que os de CSS.

Voltando ao nosso caso, se assumirmos que o CSS absorvido pelo Webpack sempre ficará embutido no `bundle.js`, fatalmente estaremos obrigando os usuários a aguardarem o download completo do JavaScript, para somente então visualizarem a estilização dos elementos do DOM. Isso significa, como regra geral, que a *sensação* de lentidão da página seria superior do que nos casos em que o CSS é exposto separadamente.

Resumindo: para fins de publicação de nossa PWA, não é uma boa prática manter CSS e JavaScript juntos. Ok, mas então, como faremos? Baixaremos um outro plugin chamado **extract-text-webpack-plugin**. Estando na pasta `front-end`, execute o comando adiante.

```
npm install --save-dev extract-text-webpack-plugin@2.1.2
```

Este plugin de nome comprido serve exatamente para extrair todo o CSS aplicado aos componentes do React para um único arquivo de estilo na pasta `dist`. Para utilizá-lo, vamos alterar o

arquivo `webpack.config.js` em três pontos:

1. Incluir uma importação através de um `require()`;
2. Acrescentar o plugin no array de `plugins`;
3. Alterar a `rule` simplista dos pacotes `style-loader` e `css-loader` para trabalhar associada ao `extract-text-webpack-plugin`.

Como são muitas alterações, vou mostrar os trechos alterados em separado e, posteriormente, exibir o resultado. A importação do plugin é semelhante às outras que já fizemos.

```
const ExtractTextPlugin = require('extract-text-webpack-plugin');
```

No array de `plugins`, acrescente a sintaxe a seguir. Ela é bem simples. O importante é que você note que a criação do `ExtractTextPlugin` oferece um parâmetro string cujo valor é `style.css`. Esse será o nome do arquivo gerado na pasta `dist`.

```
new ExtractTextPlugin('style.css')
```

Por fim, ajuste a sintaxe associada ao `style-loader` e `css-loader` conforme o trecho adiante. A configuração antiga da regra de CSS foi totalmente substituída pelos parâmetros do objeto `ExtractTextPlugin`.

```
{
   test: /\.css$/,
   use: ExtractTextPlugin.extract({
       fallback: "style-loader",
       use: "css-loader"
   })
}
```

Pronto, agora veja os trechos alterados do `webpack.config.js`.

```
// ...
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
   // ...
   plugins: [
       new HtmlWebpackPlugin({
           filename: 'index.html',
           template: path.join(__dirname, 'src/index.html')
       }),
       new ExtractTextPlugin('style.css')
   ],
   module: {
       rules: [
       // ...
       {
           test: /\.css$/,
           use: ExtractTextPlugin.extract({
               fallback: "style-loader",
               use: "css-loader"
           })
       }
       ]
   },
   // ...
};
```

Para ver o resultado, execute mais uma vez o comando de compilação.

```
npm run build
```

Acesse o diretório `dist` e veja que tudo está conforme esperado.

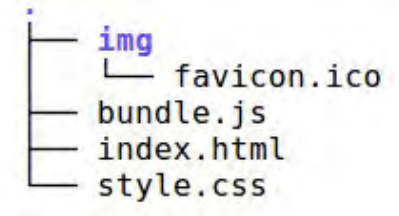

Figura 3.22: Pasta dist com o CSS

Próximo passo: a magia do *minify*!

Última versão da configuração do Webpack

Agora lhe convido a olhar um pouco mais de perto os arquivos gerados. Abra o arquivo `behappywith.me/front-end/dist/bundle.js` na sua IDE.

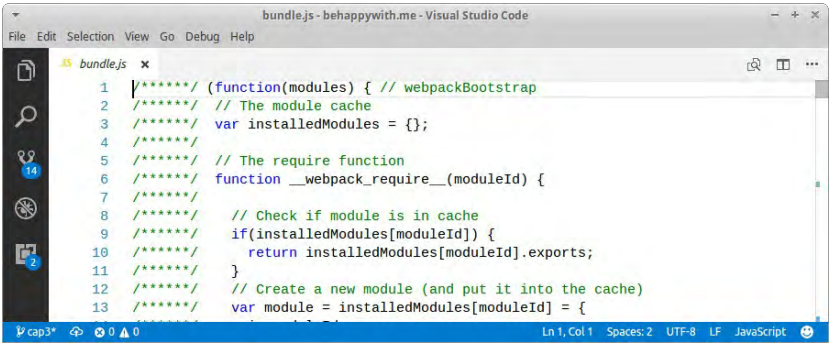

Figura 3.23: Arquivo bundle.js não minificado

Tudo parece estar em ordem. E realmente está em ordem, porém *para o ambiente de desenvolvimento*. Em produção, a prática é minificar todos os arquivos JavaScript e CSS.

MINIFICAR PRA QUÊ, SE EU JÁ SEI O QUE FAZER?

A ação de minificar consiste em remover tudo o que não for necessário para a execução do código, com a principal proposta de reduzir o tamanho físico dos arquivos. Nela geralmente são removidos espaços, comentários, tabulações, e há até uma refatoração que renomeia o nome das variáveis e funções para nomes mais curtos, como `a()`, `b()` etc. Então, quer arquivos menores? Tenha a prática do *minify* na cabeça!

Há estatísticas anônimas na internet que apontam que arquivos minificados são reduzidos em torno de 10%. Para uma aplicação real com muitos acessos simultâneos, isso é expressivo e deve ser considerado. Não faremos diferente. Aplicaremos um último plugin do Webpack chamado **uglifyjs-webpack-plugin**. Vamos instalá-lo:

```
npm install --save-dev uglifyjs-webpack-plugin@0.4.6
```

Com a instalação realizada, resta-nos configurar o arquivo `webpack.config.js`. Para isso, acrescente sua importação na parte superior do arquivo.

```
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
```

Mesmo sem repassarmos nenhum parâmetro, o plugin `uglifyjs-webpack-plugin` é capaz de identificar quais arquivos serão minificados, orquestrando tudo por meio de seu objeto `UglifyJSPlugin`. Isso significa que o simples fato de criá-lo (via

sintaxe `new UglifyJSPlugin()`) dentro do array `plugins` já é suficiente para que o código seja minificado corretamente. Então, conforme exposto adiante, acrescente ao array `plugins` a criação do objeto `UglifyJSPlugin` .

```
module.exports = {
   // ...
   plugins: [
       new HtmlWebpackPlugin({
           filename: 'index.html',
           template: path.join(__dirname, 'src/index.html')
       }),
       new ExtractTextPlugin('style.css'),
       new UglifyJSPlugin()
   ],
   // ...
};
```

Pronto. Execute novamente a compilação (para a pasta `dist`).

```
npm run build
```

Abra novamente o arquivo `bundle.js` na pasta `dist` e veja o resultado.

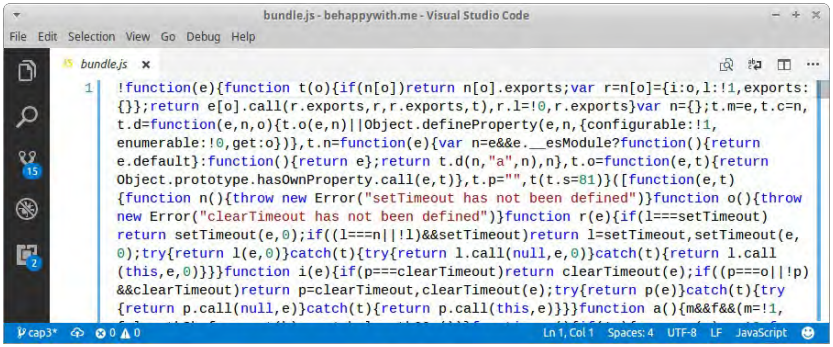

Figura 3.24: Arquivo bundle.js minificado

Nosso ambiente já está configurado para produção. Mas espere

um pouco: ao rodar em tempo de desenvolvimento, esse arquivo estará minificado também! Apague o conteúdo da pasta `dist`, execute o servidor de desenvolvimento e veja você mesmo.

```
npm start
```

A aplicação foi executada normalmente, porém note que há um *warning* explícito no console.

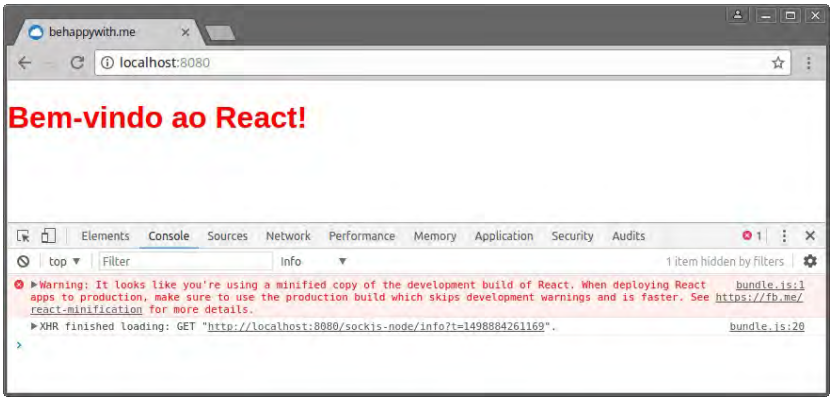

Figura 3.25: Aviso no console do browser

Esse aviso não é preocupante:

1. Ele está nos dizendo que estamos reduzindo os arquivos em tempo de desenvolvimento;
2. E que precisamos garantir que a compilação para produção seja realizada sem os avisos do ambiente de desenvolvimento.

O Webpack nos ajudará no item 2 com um plugin chamado **DefinePlugin**. Por meio dele, avisaremos ao Webpack o ambiente que estamos trabalhando. Só uma observação: **DefinePlugin** já está integrado ao Webpack, o que significa que não precisaremos

importá-lo.

O item 1 pode prejudicar nosso trabalho durante o desenvolvimento. Depurar código minificado no browser é uma tarefa insana! Para evitar esse problema, precisaremos desabilitar o plugin **UglifyJSPlugin**. Ok, óbvio. Mas como faremos isso apenas em tempo de desenvolvimento?

O Node.js possui um recurso conhecido como variáveis de *environment*, através das quais podemos capturar propriedades enviadas durante a chamada do processo. Em termos práticos, tudo parte da execução típica de uma aplicação Node.js, conforme adiante.

```
NODE_ENV=production node app.js
```

A suposta aplicação `app.js` poderá acessar o valor de `NODE_ENV` pela referência à propriedade de mesmo nome, fornecida no objeto de configuração `process.env`. Veja um exemplo.

```
if (process.env.NODE_ENV === 'production') {
   // Executar aqui alguma coisa para ambiente de produção
}
```

Isso se encaixa perfeitamente à nossa necessidade. O que precisamos fazer é chamar os plugins **DefinePlugin** e **UglifyJSPlugin** apenas quando estivermos publicando o código de produção. Ou seja, quando estivermos com o valor de `NODE_ENV` definido como `production`.

Existem várias formas de implementar isso. Faremos aqui a mais simples. Basicamente, vamos criar um array fora do `module.exports` com os plugins que coexistem em ambos os

casos. Isso pode ser feito com uma simples sintaxe JavaScript padrão, como no trecho a seguir.

```
plugins = [
   new HtmlWebpackPlugin({
       filename: 'index.html',
       template: path.join(__dirname, 'src/index.html')
   }),
   new ExtractTextPlugin('style.css')
];
```

Em seguida, através de uma condição simples, detectamos se estamos em produção e, se for o caso, acrescentamos os plugins **DefinePlugin** e **UglifyJSPlugin**. A dita *condição* nada mais é do que um `if` que vai verificar se o valor de `process.env.NODE_ENV` é igual a `'production'`. Se for, vamos acrescentar os plugins pelo método `push()`.

```
if (process.env.NODE_ENV === 'production') {
   plugins.push(new webpack.DefinePlugin({
       "process.env": {
           NODE_ENV: JSON.stringify(process.env.NODE_ENV)
       }
   }));
   plugins.push(new webpack.optimize.UglifyJsPlugin());
}
```

Note o uso do `===`, conhecido como *strict equal*, ou igualdade estrita. Este tipo de comparação no JavaScript só retornará `true` se os operandos possuírem o *mesmo valor* e o *mesmo tipo*.

A última alteração apenas aplicará o array `plugins` na propriedade de mesmo nome dentro da estrutura `module.exports`. Veja a seguir a reunião dos três trechos apresentados.

```
plugins = [
   new HtmlWebpackPlugin({
```

```
        filename: 'index.html',
        template: path.join(__dirname, 'src/index.html')
    }),
    new ExtractTextPlugin('style.css')
];

if (process.env.NODE_ENV === 'production') {
    plugins.push(new webpack.DefinePlugin({
        "process.env": {
            NODE_ENV: JSON.stringify(process.env.NODE_ENV)
        }
    }));
    plugins.push(new webpack.optimize.UglifyJsPlugin());
}

module.exports = {
    // ...
    plugins: plugins,
    // ...
};
```

Agora precisamos enviar os valores `development` e `production` pela variável `NODE_ENV` aos comandos `webpack-dev-server` e `webpack`, respectivamente. Faremos isso ajustando os scripts `start` e `build` do arquivo `package.json`. Veja como ficou o trecho `scripts` a seguir.

```
// ...
"scripts": {
    "start": "NODE_ENV=development node_modules/webpack-dev-server/bin/webpack-dev-server.js",
    "build": "NODE_ENV=production node_modules/webpack/bin/webpack.js"
},
// ...
```

Agora, ao executarmos o servidor de desenvolvimento, o **UglifyJSPlugin** não será chamado, os arquivos permanecerão intactos e o `warning` não será exibido. Por outro lado, se compilarmos o projeto, o `webpack` saberá que estamos publicação

a versão de produção por meio do plugin **DefinePlugin** (evitando a exibição do `warning`) e reduzirá os arquivos com o **UglifyJSPlugin**. Apague os arquivos da pasta `dist` e execute o comando para iniciar o servidor.

```
npm start
```

Abra o navegador, acesse o endereço http://localhost:8080 e veja o resultado.

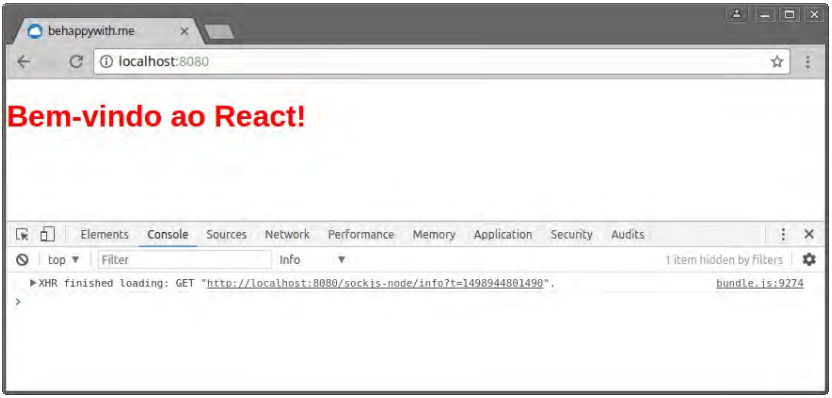

Figura 3.26: Resultado publicado pelo webpack-dev-server

Para confirmar que tudo deu certo, caso você esteja utilizando o Chrome, acesse a aba *Sources* (do *DevTools*), selecione o arquivo `bundle.js` e observe que o código não está minificado. Opções semelhantes existem nas ferramentas de desenvolvimento dos demais navegadores. Alternativamente, você pode acessar o código-fonte da página e clicar no link do `bundle.js`.

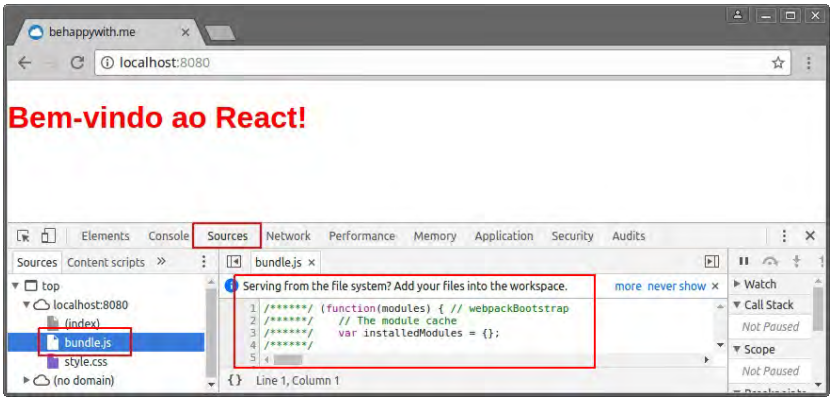

Figura 3.27: Arquivo bundle.js não minificado

Digite `Ctrl+C` para finalizar o servidor de desenvolvimento. Vamos agora compilar o projeto. Execute o comando adiante.

```
npm run build
```

Abra o arquivo `dist/bundle.js` e veja que ele está minificado.

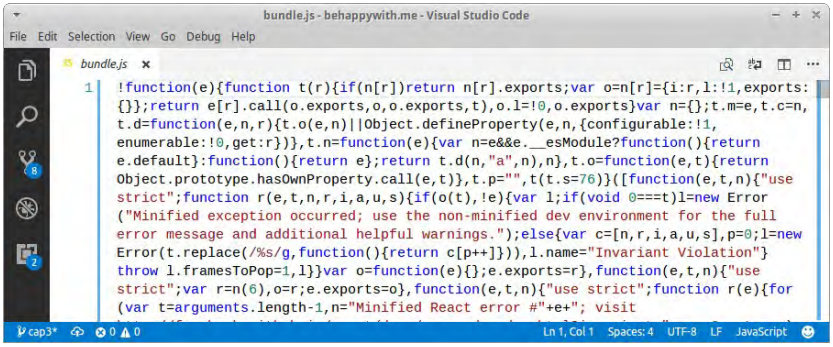

Figura 3.28: Arquivo bundle.js minificado

Abra agora o arquivo `dist/index.html` no seu navegador e veja que o `warning` também não apareceu. Neste caso, os

arquivos foram reduzidos, mas o **DefinePlugin** avisou ao React que estamos com os arquivos de distribuição de produção.

Figura 3.29: Arquivo index.html aberto direto no browser

Finalmente concluímos a construção do alicerce da nossa aplicação progressiva. A versão final do `webpack.config.js` está exposta a seguir. Você também pode baixá-la em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/webpack.config.js, do repositório GitHub deste projeto.

Só um detalhe: caso você baixe-o, perceberá que há outras dependências não discutidas neste capítulo. Elas serão tratadas no futuro, mas não trarão problemas em sua configuração.

```
const webpack = require('webpack');
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');

plugins = [
   new HtmlWebpackPlugin({
```

```
        filename: 'index.html',
        template: path.join(__dirname, 'src/index.html')
    }),
    new ExtractTextPlugin('style.css')
];

if (process.env.NODE_ENV === 'production') {
    plugins.push(new webpack.DefinePlugin({
        "process.env": {
            NODE_ENV: JSON.stringify(process.env.NODE_ENV)
        }
    }));
    plugins.push(new webpack.optimize.UglifyJsPlugin());
}

module.exports = {
    entry: path.join(__dirname, 'src/index.jsx'),
    output: {
        path: path.join(__dirname, 'dist'),
        filename: 'bundle.js'
    },
    resolve: {
        extensions: [".js",".jsx"]
    },
    plugins: plugins,
    module: {
        rules: [
        {
            test: /.jsx?$/,
            exclude: /node_modules/,
            include: path.join(__dirname, 'src'),
            use: [
            {
                loader: 'babel-loader',
                options: {
                    presets: ['es2015', 'react']
                }
            }
            ]
        },
        {
            test: /\.(jpe?g|ico|png|gif|svg)$/i,
            loader: 'file-loader?name=img/[name].[ext]'
        },
        {
```

```
            test: /\.css$/,
            use: ExtractTextPlugin.extract({
                fallback: "style-loader",
                use: "css-loader"
            })
        }
        ]
    },
    devServer: {
        publicPath: "/",
        contentBase: "./dist"
    }
};
```

Veja na figura adiante a estrutura final dos arquivos e diretórios. Nesta representação, os arquivos de `dist` foram excluídos.

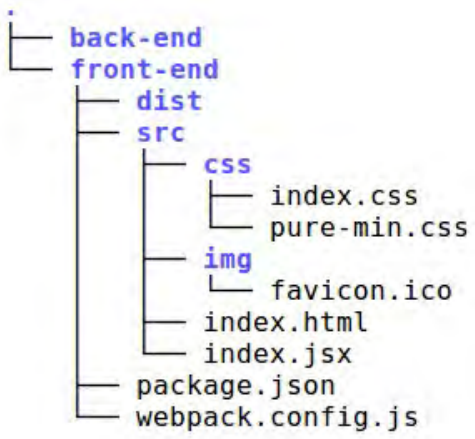

Figura 3.30: Estrutura do projeto React

É importante ressaltar que não esgotamos a instalação de todos os pacotes necessários. Acrescentaremos também outros diretórios para apoiar a arquitetura do projeto. Como são assuntos que merecem atenção absoluta, trataremos deles no momento

oportuno.

Não se esqueça de limpar seu diretório `behappywith.me/front-end/dist`. Do contrário, você terá a falsa impressão de que o servidor de desenvolvimento não está enxergando as alterações na pasta `src`.

3.5 CRIANDO O PROJETO COM CREATE-REACT-APP

Existe uma ferramenta publicada no GitHub que mantém uma série de configurações padronizadas para a criação de projetos React similares ao que discutimos na seção anterior. Esta ferramenta pode ser encontrada em https://github.com/facebookincubator/create-react-app.

A ideia do `create-react-app` é oferecer aos desenvolvedores React algo semelhante aos comandos `ember-cli` e `@angular-cli`, ferramentas para auxiliar a criação dos projetos Ember e Angular.

Imagino que você esteja se perguntando por que configuramos nosso projeto manualmente se existe uma ferramenta que faz isso por nós. Vou direto ao ponto. Veja nas figuras a seguir a diferença entre os arquivos de distribuição de produção.

A estrutura de diretórios à esquerda exibe a pasta `dist` da nossa configuração. A estrutura da direita mostra o resultado do `create-react-app`.

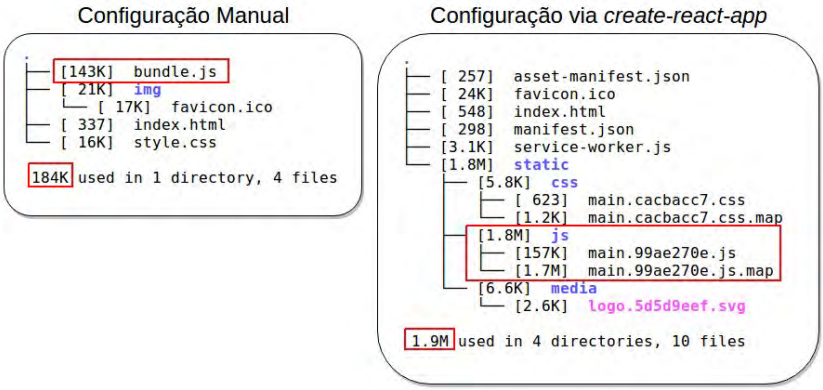

Figura 3.31: Comparação entre os arquivos bundle.js

De forma superficial, podemos ver o quão importante é fazer um ajuste fino na arquitetura de um projeto. Nossa arquitetura resultou em um diretório de distribuição de **184 KB**, contra os **1.9 MB** da geração automática. A intenção aqui não é dizer que a ferramenta `create-react-app` é ruim; muito pelo contrário, ela é excelente. A questão é avaliar se realmente precisamos de tudo que ela oferece.

Obviamente a diferença entre 1.9 MB e 184 KB vem com um bônus de muitos outros pacotes auxiliares às aplicações React. Não vamos retratar os plugins instalados em uma aplicação criada pelo `create-react-app`, mas você pode visualizar a lista completa em https://github.com/facebookincubator/create-react-app/.

Neste mesmo endereço, você também encontrará os passos de instalação. É tudo muito simples. Vou reproduzi-los aqui. Mas atenção, caso você opte em trabalhar com a arquitetura criada manualmente na seção anterior, não execute os próximos passos.

Saia do diretório `behappywith.me`. Em seguida, execute a

instalação global do pacote `create-react-app` através do comando a seguir.

```
npm install -g create-react-app
```

O comando de criação da estrutura do projeto já cria também uma pasta para guardar tudo. Logo, você não precisa criá-la manualmente. Simplesmente execute o comando adiante (ele pode demorar alguns minutos).

```
create-react-app test-create-react-app
```

Em seguida, acesse o diretório e inicialize a aplicação, respectivamente com os comandos expostos à frente.

```
cd test-create-react-app/
npm start
```

O comando `start` vai abrir automaticamente um navegador e acessar o endereço http://localhost:3000. Este vai exibir algo semelhante à figura a seguir.

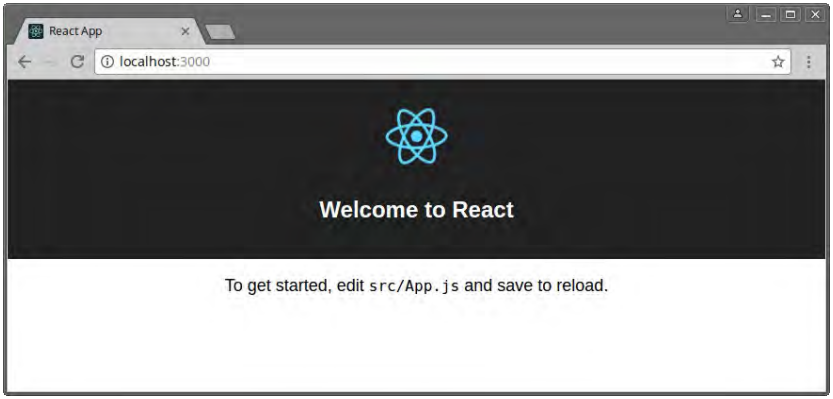

Figura 3.32: Aplicação de teste create-react-app

Para finalizá-la, basta clicar `Ctrl+C` no console. Assim como

nossa configuração manual, você será capaz criar os arquivos de distribuição de produção.

```
npm run build
```

Pronto. Você é livre para escolher qualquer uma das alternativas apresentadas neste capítulo. É claro, a versão que roda no browser é menos recomendada, por questões já discutidas. No mais, seja pela arquitetura manual ou automática, você conseguirá implementar as features apresentadas nos próximos capítulos normalmente.

Caso você opte pelo `create-react-app`, para injetar os componentes no DOM, você usará o arquivo `src/index.js`. Essa arquitetura centraliza o código que você produzirá no arquivo `src/App.js`. Ainda não tratamos do assunto, mas ela também já registra um *Service Worker* e contém um arquivo `manifest.json`. Veremos em detalhes, mas (spoiler) o Service Worker nos ajudará a guardar dados no navegador, e o `manifest.json` possui dados para que a PWA seja *instalada* no dispositivo do usuário.

3.6 CONCLUSÃO

O propósito principal do capítulo foi concluído com a utilização do NPM, do Babel e do Webpack. Por meio deles, fomos capazes de configurar todas as dependências da nossa PWA. Estudamos três possibilidades:

1. Via navegador, que só é aconselhada para estudo;
2. A configuração avançada, elaborada com uma sintonia fina a respeito de quais pacotes e plugins serão utilizados;

3. A forma automática, recomendada para quem não quer perder tempo com configuração e não se importar em ter muitos pacotes que talvez não sejam relevantes no resultado final.

Mas qual delas escolher? Você é a única pessoa capaz de responder essa pergunta! Há projetos e projetos. Se seu projeto merece um tratamento minucioso quanto a performance, evolução dos plugins e tamanho dos arquivos publicados, sugiro a configuração manual.

Por outro lado, se você não tiver uma equipe de desenvolvedores fixa (fábrica de softwares, por exemplo) e a configuração perfeita do projeto não for sua principal preocupação, sugiro usar a opção automática. No caso, por *falta de configuração perfeita*, entenda como um prejuízo de meros segundos no carregamento da solução e uma imperceptível sensação de *lentidão* (em relação à configuração manual). Esses dois elementos só serão realmente prejudiciais se você possuir milhares de acessos simultâneos.

Boa sorte em sua escolha! No próximo capítulo, vamos iniciar a codificação dos componentes da PWA, a começar pela tela de cadastro do usuário.

Capítulo 4

PRIMEIROS PASSOS DO DESENVOLVIMENTO

Iniciaremos a codificação. Para relembrá-lo, os requisitos da página que vamos desenvolver englobam a tela inicial do sistema, exibida aos usuários apenas durante o primeiro acesso à aplicação. Como serão nossos primeiros componentes React do projeto, vamos aprofundar bastante no funcionamento do framework, eventualmente desbravando também as melhorias oferecidas pelo ECMAScript 6.

Neste capítulo, serão construídos os componentes `Header` e `Label`, conforme exibido na imagem a seguir. Iniciaremos com apenas esses dois propositalmente por se tratarem de componentes sem estado e com lógica ínfima, os que nos dá um ambiente propício para introduzir elementos da arquitetura React ainda não discutidos. Faremos isso paulatinamente.

Figura 4.1: Épico 2

Os frameworks web contemporâneos nos levam a um paradigma interessante: desenvolver por componentes. O React também segue nessa linha. Criaremos componentes dentro de componentes para compor telas e, por fim, construir aplicação.

Aceita essa premissa, resta-nos decidir quais componentes e em qual nível de granularidade (tamanho dos componentes) vamos construí-los. A seção a seguir introduzirá essa discussão.

4.1 DEFINIÇÃO DOS COMPONENTES

A melhor forma de definir a composição e a hierarquia dos componentes é através do protótipo. Para facilitar, vou replicar o protótipo de baixa fidelidade da tela de novo usuário a seguir.

Figura 4.2: Protótipo da tela de novo usuário

A tela da esquerda é a primeira visão, cujo usuário poderá entrar com seu nome e gênero. O nome será um simples `<input>` do tipo `text`, enquanto o gênero será escolhido entre duas imagens. Se o usuário clicar no botão *Próximo* sem definir ambos os dados, o sistema deverá exibir uma mensagem de validação.

A tela da direita será exibida com o `<input>`, configurado como apenas leitura, e um *Image Scroller* para seleção do avatar do usuário. O componente Image Scroller será horizontal e é por natureza mais amigável aos smartphones.

Para deixá-lo compatível com desktop, haverá ícones para regredir ou avançar na lista de imagens. Na parte inferior, há os botões *Voltar* e *Salvar*, cujos objetivos são óbvios. É importante

ressaltar que as imagens de avatar vão variar de acordo com o gênero.

Analisando as telas, podemos distribuí-la em vários componentes, em diferentes níveis de interesse. Vamos começar com o esboço a seguir.

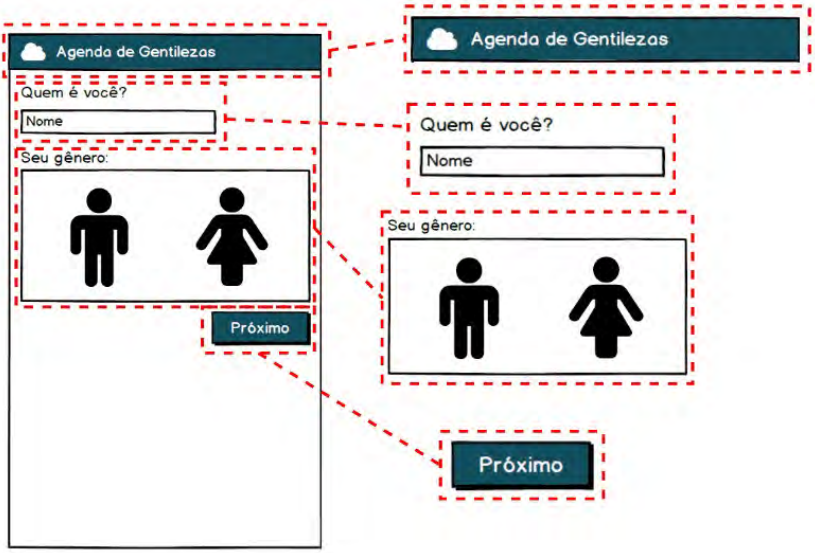

Figura 4.3: Esboço de organização dos componentes

A ideia aqui é *dividir para conquistar*. Cada componente é uma parte do todo. Na imagem anterior, dividimos a primeira tela em 4 componentes:

1. Cabeçalho;
2. A `<label>` e o `<input>` para receber o nome do usuário;
3. A `<label>` e as imagens para seleção do gênero do usuário;
4. O botão de próximo.

Podemos ir além. Por exemplo, veja que há possibilidade de dividir ainda mais o cabeçalho.

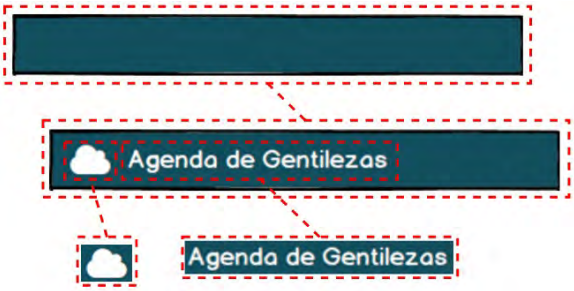

Figura 4.4: Divisão do cabeçalho

Aqui o cabeçalho foi organizado em outros três componentes:

1. A `<div>` azulada para agrupar os itens do cabeçalho;
2. O logotipo da nossa aplicação;
3. A `<label>` Agenda de Gentilezas.

O exagero nesta divisão é nítido. O problema é que nem sempre isso fica óbvio como no caso apresentado. Definir componentes muito pequenos é extremamente improdutivo. Em contrapartida, definir componentes muito grandes prejudica a usabilidade.

Mas afinal, como vamos definir a granularidade dos componentes? Não há uma regra fixa para isso. Há correntes que optam por organizá-los em componentes de apresentação (*Presentational*) e agrupadores (*Container*). Outras, orientam o uso de funções JavaScript para componentes mais simples e classes (ECMAScript 6) para componentes mais complexos. Há também a classificação de componentes sem estado (*stateless*) e com estado

(*stateful*). Enfim, tudo é possível.

Neste livro, vamos trabalhar com duas orientações:

1. **Airbnb React/JSX Style Guide**: existe um projeto no GitHub chamado Airbnb que mantém várias seções de boas práticas organizadas por tópicos. Lá estão listadas várias práticas para se construir aplicações React e código JSX de forma manutenível, organizada e padronizada. É uma excelente referência para quem deseja aperfeiçoar-se no desenvolvimento de aplicações *front-end*. Caso queira mais detalhes, acesse https://github.com/airbnb/javascript/tree/master/react/.

2. **Abordagem de Kirupa Chinnathambi**: boas práticas apresentadas no livro *Learning React: A Hands-On Guide to Building Maintainable, High-Performing Web Application User Interfaces Using the React JavaScript Library*. Nesta obra, Kirupa destaca algumas regras gerais, tais como *"um componente só pode ter um objetivo"* e *"deve ser passível de reutilização"* (por este, ou outros projetos).

Como consolidado dessas duas abordagens, temos as seguintes orientações:

1. **Objetivo único**: um componente só pode ter um objetivo. Componentes com mais de um objetivo são fadados à separação. Por exemplo, um formulário com três tags `<input>` de texto e um botão pode ser separado em cinco componentes distintos (um deles seria o próprio formulário, para agrupar os demais).
2. **Stateless ou Stateful**: um componente pode ter ou não

estado. Por *estado*, entenda que o componente precisar guardar valores dentro de si para viabilizar sua renderização. Componentes com estado (*stateful*) são mais complexos por natureza. Se você tiver um componente que agrupe pequenos componentes que necessitam de seus próprios estados, ele seria um bom candidato à separação. Por outro lado, um agrupador de componentes sem estado pode ser tratado como um único componente.

3. **Reutilização**: um componente muito grande é difícil de ser reutilizado. Com outra perspectiva, um componente muito simplório também é. Essa análise precisa ser feita com foco na aplicação. Ainda sobre reutilização, o fato de um componente ter parte de sua estrutura diferente não significa necessariamente a separação. Exemplo: um botão, que ora terá o texto *Cancelar*, ora o texto *Salvar*, **não** precisa ser dividido em dois botões.

Se tentarmos aplicar essas regras ao cabeçalho, é possível abstrair que seu principal objetivo é ficar na parte superior da página e agrupar seus elementos compositores. Cada um desses elementos é *stateless* e pode perfeitamente ser gerenciado como parte do componente maior. Ou seja, conclui-se que segregar o cabeçalho conforme exposto anteriormente é um exagero.

Essas discussões foram para fixarmos a ideia da componentização de uma aplicação React. Podemos agora finalmente apresentar uma alternativa válida para nosso primeiro protótipo. Veja na figura adiante como serão organizados nossos componentes.

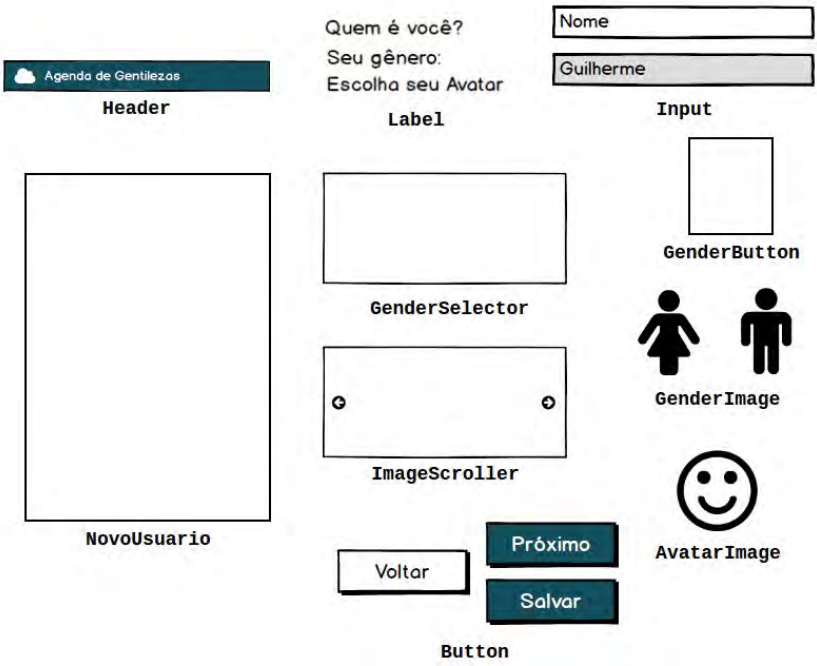

Figura 4.5: Componentes da tela de novo usuário

Discutiremos suas especificações particulares nas próximas seções.

Um pouco de UX

A escolha do avatar do personagem será realizada por meio de um *Image Scroller*, componente que permite avançar (ou retornar) para uma imagem com o evento swipe horizontal na tela do dispositivo. O *swipe* é um tipo de toque no qual o usuário arrasta o dedo horizontalmente para visualizar uma lista. Veja um exemplo de swipe ao lado esquerdo da figura adiante.

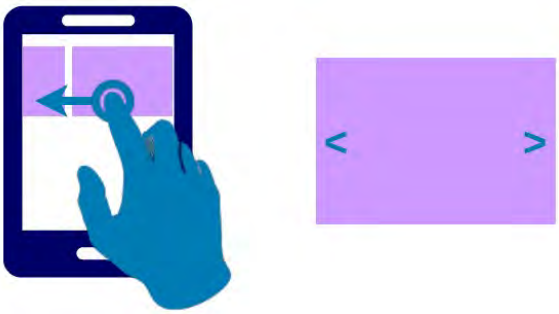

Figura 4.6: Exemplo de swipe

Além do swipe, é importante garantirmos que o componente também aceite o *flinging*, um tipo de toque semelhante ao swipe, porém executado com uma mecânica ligeiramente diferente, mais rápida, como se algo fosse *jogado* para o lado.

Gostaria de destacar que as setas laterais em cada imagem, além de facilitar os cliques de mouse dos usuários de desktop, ajudam na *affordance* do componente. Affordance é a capacidade que um componente tem de dizer aos usuários que ele é clicável, ou recebe algum tipo de ação específica. As setas *dizem* ao usuário que arrastar o dedo sobre a imagem vai fazê-la desaparecer para o lado, dando lugar a outra.

UX

O acrônimo UX advém do termo *User eXperience*, cuja tradução oficial é *Experiência do Usuário*. Este conceito engloba o estudo de elementos associados à interação do usuário com um sistema (ou *serviço*, na ideia mais abrangente), com o intuito de melhorar sua percepção. Apesar de ser muito empregado na criação do design das aplicações, o UX extrapola os aspectos de usabilidade e avança para um paradigma mais subjetivo, tratando também do *sentimento* que o usuário carrega após sua interação.

Em outras palavras, preocupar-se com UX durante o desenvolvimento de aplicações requer alinhar as necessidades dos usuários (e seus objetivos de negócio) às tecnologias disponíveis, projetando interfaces que proporcionem uma boa experiência nas interações.

Nosso componente Image Scroller será produzido para atender a esses requisitos. Eventualmente, faremos discussões iguais a essa para detalhar as soluções de UX adotadas na PWA.

4.2 CONSTRUINDO O COMPONENTE APP

Os principais componentes desta tela serão o cabeçalho e a `<div> NovoUsuario`. Este último fará a organização dos demais componentes e será responsável por alimentá-los com as propriedades e estados adequados.

Contudo, como o sistema completo terá outras `<div>` agrupadoras, o ideal é criarmos uma nova `<div>` mais abrangente, responsável por alimentar todas as demais `<div>` internas. Por questões de convenção das aplicações React, vamos chamar essa `super-<div>` de `App`.

Não se preocupe com esse *conceito* de passar propriedades e estados para os componentes. A partir de agora, vamos abordar esses detalhes do desenvolvimento.

Por meio de sua IDE, abra o arquivo `index.css`, cujo caminho é `behappywith.me/front-end/src/css/index.css`. Não vamos precisar mais da estilização do `<h1>` de vermelho, criada no capítulo anterior para exercitar a configuração do ambiente de desenvolvimento. Remova o trecho a seguir do arquivo.

```
h1 {
   color: red;
}
```

A versão final do arquivo ficará igual a:

```
#main {
   border: 0;
   padding: 0;
   margin: 0;
}
```

Uma boa prática em projetos React é organizar todos os componentes dentro de uma pasta chamada `components`. Crie-a dentro do diretório `src`. A seguir, estão os comandos necessários para a criação via terminal, mas você também poderá utilizar a IDE para esse procedimento.

```
cd behappywith.me/front-end/src/
```

```
mkdir components
```

Precisamos de um arquivo principal a partir do qual todos os demais componentes serão convocados. Por convenção do *Airbnb React/JSX Style Guide*, é comum chamá-lo de `App.jsx`. No geral, os componentes ficarão em subpastas com seus respectivos nomes, porém o componente `App` será um mero organizador, sem regras de renderização própria.

Crie o arquivo `App.jsx` dentro da pasta `components` para começarmos sua codificação. Na primeira linha, entre com a importação da biblioteca `React`, conforme o seguinte:

```
import React from 'react';
```

O próximo passo é criar a classe do componente. Por enquanto, vamos testar o uso do componente `App` no projeto. Por conta disso, o código deve apenas retornar um `<h1>` no método `render()`. Veja-o a seguir.

```
class App extends React.Component {
    render() {
        return (
            <h1>Componente App!</h1>
        );
    }
}
```

O componente está pronto. Todavia, da forma como ele está encapsulado no arquivo `App.jsx`, o componente `index.jsx` não poderá encontrá-lo. Precisamos de um mecanismos de *exportação* para deixar `App` disponível para outros componentes poderem importá-lo. Introduziremos tal assunto no próximo tópico.

O poder da exportação e importação

A partir do ECMAScript 6, ficou muito mais fácil trabalhar com recursos de importação e exportação de bibliotecas. Agora existem meios de exportação e importação explícita, respectivamente assimilados com a adoção dos comandos `export` e `import`.

A sintaxe `export` permite exportar vários trechos de um determinado arquivo. Existem duas formas de exportação:

1. a padrão, indicada pela sintaxe `export default`;
2. a nomeada, apenas sinalizada pela palavra `export`.

Vejamos um exemplo contemplando os dois casos.

```
// Arquivo componente.js
export var valorA = 'Valor de A';
export function funcaoB() {
   return 'Função B';
};
export default function funcaoPrincipal() {
   return 'Função Principal';
};

// Arquivo main.js
import funcaoPrincipal, { valorA, funcaoB } from "componente"

// Arquivo outro.js - outro exemplo
import funcaoPrincipal from "componente"
```

Primeiro veja a sintaxe do arquivo `componente.js`. Temos três opções de exportação, sendo apenas a última como a principal. O uso da notação `default` permite-nos importar o elemento exportado de forma semelhante ao exemplo exposto no arquivo `outro.js`, em que simplesmente fazemos referência ao elemento principal.

Caso precisemos importar as exportações nomeadas (sem o

`default`), faremos o uso de `{}` (chaves) ao redor das variáveis que receberão as propriedades/funções importadas. No caso do arquivo `main.js`, a variável `valorA` receberá a variável de mesmo nome, exportada em `componente.js`. A importação `funcaoB` funciona de forma equivalente.

A ideia da importação/exportação nomeada é facilitar a obtenção de trechos menos importantes dos arquivos separados pelo acesso às propriedades de mesmo nome. Como resultado final, temos um código mais enxuto e organizado.

Com esses conceitos estabelecidos, podemos voltar ao componente `App`. Para permitir que os outros arquivos o *encontrem*, vamos exportá-lo pela sintaxe `export default App`. Veja o código completo a seguir.

```
import React from 'react';

class App extends React.Component {
    render() {
        return (
            <h1>Componente App!</h1>
        );
    }
}

export default App;
```

Neste e na grande maioria dos casos, faremos somente a exportação padrão (`export default`). A importação nomeada, com as `{}` (chaves), é mais usada em bibliotecas que disponibilizam muitos recursos por meio do mesmo arquivo JavaScript.

Finalizando, com base na exportação do componente `App`, poderemos importá-lo diretamente no arquivo `index.jsx`,

conforme a simulação a seguir. Nota: este ajuste no código de `index.jsx` será revisado em breve. Vou evidenciá-lo aqui apenas para concretizar o nosso raciocínio.

```
// App.jsx
export default App;

// index.jsx
import App from './components/App.jsx'
```

Mas atenção, temos um detalhe importante! Devo ressaltar que a importação nomeada **não** é a mesma coisa do que o recurso *destructing* do ECMAScript 6. Vamos conversar sobre ele adiante.

Appetite for destructing

Primeiro ponto: apesar do uso das chaves, não confunda a importação nomeada com o *destructing* (desestruturamento), outro novo recurso do ECMAScript 6. Para atar as pontas soltas e não o confundir quando começarmos a usar o *destructing*, vamos estudá-lo agora.

O ES6 tem alguns objetivos principais: tornar o JavaScript menos verboso, mais fácil de entender e com melhor manutenibilidade. A funcionalidade de desestruturamento, ou simplesmente destructing, é uma nova opção que nos permite recuperar partes de um objeto ou array de forma direta e simples.

Vejamos um exemplo prático. Note que o simples uso de `{}` permite-nos obter as propriedades de `obj` sem informar qual delas estará salva nas variáveis.

```
var obj = {
   a: 'valor de a',
   b: 'valor de b'
};
```

```
// Exemplo 1
var {a,b} = obj;

// Exemplo 2
var {a} = obj;
```

Em outras palavras, o ES6 sabe que, decorando a variável com chaves, ele deve obter a propriedade de mesmo nome dentro do objeto à direita da atribuição. Isso pode ser realizado para pegarmos uma ou todas as propriedades do objeto.

Há diversas outras aplicações excelentes para o uso do destructing, mas o que vimos até aqui já é suficiente para utilizarmos em nosso projeto. Apenas ratificando, não confunda a sintaxe das importações nomeadas com a do *destructing*. Ambas usam `{}` para obter partes do todo, mas são recursos diferentes do ECMAScript 6.

Importando o componente App em index.jsx

Vamos agora refatorar o arquivo `index.jsx`. Ele será o arquivo que de fato vai injetar o código JSX através do ReactDOM. Veja que, em vez de explicitamente expor o código como exemplificado no capítulo anterior, ele chamará nosso novo componente `App`.

Há dois trechos para alterar:

1. Acrescentar uma importação ao componente `App`;
2. Alterar o método `render()` para incluir uma chamada à tag associada ao componente, que, no caso, é `<App />`.

Veja o código a seguir.

```
import React from 'react'
import ReactDOM from 'react-dom'

import './img/favicon.ico';
import './css/index.css';
import './css/pure-min.css';
import App from './components/App.jsx'

ReactDOM.render(
   <App />,
   document.querySelector("#main")
)
```

Salve tudo e execute o servidor de desenvolvimento. Nota: já reproduzimos o comando de execução do servidor várias vezes. Vou reproduzi-lo uma última vez, a seguir. Apenas se lembre de que o comando deve ser executado na pasta `front-end`. No geral, você pode deixá-lo sempre ligado. Quando for necessário desligá-lo, vou mencionar explicitamente.

```
npm start
```

Em seguida, acesse o endereço http://localhost:8080 e desfrute da exibição de nosso novo componente `App`.

Figura 4.7: Exibição do componente App

Componente principal criado. Adiante.

4.3 CRIANDO O COMPONENTE CABEÇALHO

O componente `App` será nosso ponto principal. Nele importaremos os componentes `Header` e `NovoUsuario`. Este último será responsável por organizar todos os demais componentes.

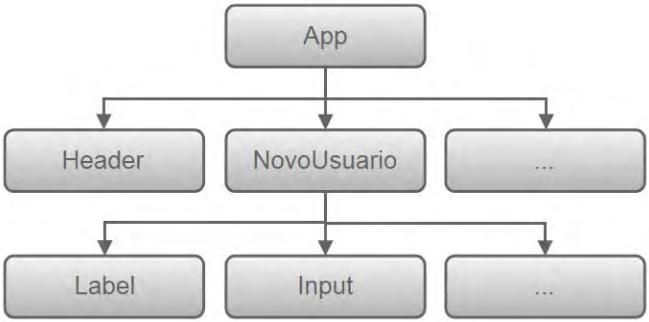

Figura 4.8: Representação de uso dos componentes

O `Header` não será necessário apenas nesta tela. Ele será útil em toda a aplicação. Isso é um comportamento muito comum em aplicações React. A reutilização dos componentes é essencial para a qualidade do sistema. Por conta disso, o próprio componente `App` vai desenhá-lo.

Arquivo JSX

Crie uma pasta `Header` dentro de `behappywith.me/front-end/src/components`. Segundo as boas práticas de codificação expostas no Airbnb React/JSX Style Guide, componentes devem ter seus diretórios nomeados com *PascalCase*.

PascalCase

Trechos criados com concatenação de palavras cuja primeira letra é escrita em caixa-alta. Exemplo: NomeCompleto.

camelCase

Não confundir PascalCase com camelCase! O camelCase tem sua primeira letra minúscula e todas as concatenações entre as palavras destacadas com letras maiúsculas. Exemplo: nomeCompleto.

Dentro do diretório, crie um arquivo chamado `index.jsx`. Essa configuração nos permitirá importar esse componente pela seguinte sintaxe:

```
// Header/index.jsx
export default Header;

// App.jsx
import Header from './Header'
```

Graças à propriedade `extensions` do `webpack.config.js`, aplicada no capítulo anterior, o `import` reconhecerá o `index.jsx` como sendo o principal arquivo do componente (diretório) importado. O `./` (ponto e barra) prefixado em `./Header` aponta para o diretório corrente.

Vamos então à criação do componente. Abra o arquivo

`Header/index.jsx` e codifique as importações necessárias logo nas primeiras linhas.

```
import React from 'react'
import './index.css'
import './img/logo.png'
```

A biblioteca `React` já é nossa conhecida. Note que importamos também um arquivo `index.css` localizado na raiz do componente (pasta `Header`) e uma imagem chamada `logo.png` dentro de uma pasta `img`. Ainda não temos nenhum dos dois arquivos. Esses pormenores serão resolvidos em breve.

Com as importações prontas, podemos criar o código do componente. As tags JSX envolvidas na criação do `Header` são bem simples. Teremos apenas uma `<div>` encapsulando um hyperlink `<a>` e um título `<h4>`.

A `<div>` será estilizada com algumas classes do Pure.css próprias para cabeçalhos e ajustadas para mantê-la sempre no topo da página. O hyperlink envolverá o logo da aplicação e o título mostrará o texto *Agenda de Gentilezas*. Veja a seguir o trecho de código relacionado.

```
<div className="header pure-menu pure-menu-horizontal pure-menu-f
ixed">
   <a href="/"><img className="logo" src="img/logo.png" /></a>
   <h4 className="label">Agenda de Gentilezas</h4>
</div>
```

Esse código ficará exposto dentro do método `render()` do componente. Poderíamos criá-lo agora, mas, antes disso, responda a uma pergunta: nosso componente `Header` precisa guardar dados da aplicação?

Figura 4.9: Protótipo do Header

Não. Essa é uma percepção importante, pois será através dela que vamos decidir se o componente será uma `class` ou uma `function`. A melhor forma de lhe mostrar esses conceitos é pela conclusão do código do `Header`. Veja-o adiante.

```
export default function Header() {
   return (
       <div className="header pure-menu pure-menu-horizontal pure
-menu-fixed">
           <a href="/"><img className="logo" src="img/logo.png" />
</a>
           <h4 className="label">Agenda de Gentilezas</h4>

       </div>
   );
}
```

Observe que esse componente possui notação totalmente diferente do `App`, discutido anteriormente. Por questões didáticas, vou replicar a declaração de ambos componentes a seguir, lado a lado.

Componente **App.jsx**	Componente **Header**
`class App extends React.Component {`	`export default function Header() {`
`}`	`}`

Figura 4.10: Comparação das declarações

A questão aqui gira em torno da diferença entre a sintaxe `function` e a `class`. As boas práticas de desenvolvimento React estabelecidas pelo *Airbnb React/JSX Style Guide* orientam-nos na

definição de dois tipos de componentes: com estado e sem estado. Por conveniência, adotaremos neste livro suas definições originais em inglês, respectivamente chamando-os de *stateful* e *stateless*.

Já sabemos que um componente React pode armazenar dados internos. Sabemos também que esses dados são guardados em um objeto chamado `this.state`. É uma boa prática criarmos componentes *stateful* (com dados na propriedade `state`) por meio da sintaxe `class`, apresentada à esquerda da figura anterior.

Como veremos nos capítulos futuros, o componente `App` será o ponto principal para obtenção e armazenamento dos dados da aplicação. Neste caso, por *armazenar*, entenda valores salvos no estado do objeto.

O objeto `this.state`, entretanto, não é obrigatório em todos os componentes. Existem casos de componentes simples, que só precisam exibir dados estáticos. Esses dados podem ser sempre os mesmos ou repassados através de *propriedades*. Componentes desse tipo podem ser implementados com uma declaração mais simples, de `function`, como apresentado à direita da figura anterior. Essa é uma boa prática e vamos adotá-la durante a criação de todos os componentes do livro.

Para finalizarmos, as propriedades dos componentes *stateless* devem ser passadas como parâmetro da função, enquanto as dos componentes *stateful* serão passadas no `constructor` da classe do componente. Retornaremos a este assunto em breve.

Esclarecida a declaração de componentes por `function`, vamos replicar adiante o código completo do componente `Header`. Codifique-o no arquivo `Header/index.jsx`.

```
import React from 'react'
import './index.css'
import './img/logo.png'

export default function Header() {
   return (
       <div className="header pure-menu pure-menu-horizontal pure
-menu-fixed">
           <a href="/"><img className="logo" src="img/logo.png" /
</a>
           <h4 className="label">Agenda de Gentilezas</h4>

       </div>
   );
}
```

Note que a exportação (`export default`) da função `Header` é feita diretamente em sua declaração. Se você observar bem a sintaxe JSX, notará também que há um atributo chamado `className` . Preciso compartilhar que este atributo **não** faz parte da especificação W3C. Então por que o usar dentro da tag `<div>` ?

Lembre-se: o JSX é muito próximo do HTML mas **não** é HTML. E esta é justamente uma de suas diferenças. O atributo `className` é equivalente ao atributo `class` das tags HTML comuns.

Infelizmente (ou felizmente), o JSX é uma sintaxe que roda sobre o JavaScript, que, por sua vez, tem a palavra `class` reservada. Por conta disso, o JSX nos oferece uma alternativa semelhante por meio do `className` .

Com nosso código JavaScript pronto, podemos concluir com a criação do arquivo de estilo (`index.css`) e o download do logotipo do sistema.

Logotipo do cabeçalho

Crie uma pasta `img` dentro de `Header`, baixe o arquivo PNG pelo endereço https://behappywith.me/img/logo.png e salve-o nesta pasta. Este arquivo contém o logotipo revestido de uma sombra que sobrepõe a cor do cabeçalho. Veja-o na figura a seguir.

Figura 4.11: Logotipo do header

A silhueta branca ficará perceptível sobre a cor de fundo do cabeçalho.

Arquivo de estilo

Agora crie um arquivo `index.css` dentro da pasta `Header`. Vamos evoluí-lo aos poucos. O primeiro passo é criar um trecho para a `<div>` de classe `header`. Ela terá uma cor de fundo azulada (`#2c80b9`) e um tamanho mínimo pela propriedade `min-width`.

```
div.header {
    background-color: #2c80b9;
    min-width: 320px;
}
```

A vantagem da propriedade `min-width` é que ela permite que o comprimento do cabeçalho fique elástico até o valor mínimo de 320 pixels. Ou seja, ele pode ter 2000 pixels, 1000 pixels, 500 pixels, ou qualquer outro valor que seja maior o igual ao tamanho mínimo de 320 pixels.

Tamanho ideal para uma aplicação mobile

A regra é simples: devemos construir designs adequados ao menor comprimento (*width*) de viewport disponível entre os aparelhos modernos. Como há vários aparelhos (iPhone, BlackBerry, Alcatel, LG, entre outros) cujo *width* da viewport é de 320 pixels, é importante estarmos compatíveis com eles.

Entretanto, é possível fixar um tamanho mínimo (`min-width`) para evitar renderizações grosseiras em casos de viewports de comprimento inferior. Para mais detalhes sobre o tamanho das viewports dos aparelhos, consule o site http://viewportsizes.com/.

A imagem do cabeçalho merece uma pequena estilização. O primeiro estilo (`div.header img`) será para generalizar quaisquer imagens utilizadas no cabeçalho. Esse excesso de zelo serve para evitar ajustes finos que possam surgir de outras propriedades, inclusive as do Pure.css. O segundo estilo (`div.header img.logo`) serve para o logo da aplicação ficar posicionado à esquerda e manter seu tamanho fixo.

```
div.header img {
    display: block;
    margin: 0;
}

div.header img.logo {
    float: left;
    height: 92px;
    width: 164px;
}
```

O texto *Agenda de Gentilezas* também necessita de um estilo. Vamos torná-lo branco (`#ffffff`), com altura (`line-height`) fixa, sem margem, sem estilização de negrito ou itálico (`font-weight`) e posicionado à esquerda (`float: left`).

```
div.header h4.label {
    color: #ffffff;
    font-weight: normal;
    line-height: 92px;
    float: left;
    margin: 0;
}
```

A parte mais importante desse arquivo são as *media queries*. A ideia básica aqui é reduzir a proporção que o cabeçalho ocupa da tela para viewport menores. A sintaxe de uma *media query* trabalha com o comando `@media` personalizado com uma condição que definirá se os estilos serão ou não aplicados.

A condição `(max-width: 600px)` , por exemplo, só aplicará os estilos internos à *media query* se o tamanho máximo da viewport for de 600 pixels. Nesse caso, vamos reduzir o tamanho do logo e do texto, provocando a redução geral do cabeçalho (a `<div>` do header é responsiva ao tamanho do seu conteúdo interno). Veja a seguir as duas *media queries* necessárias.

```
@media (max-width: 992px) {
    div.header img.logo {
        height: 78px;
        width: 139px;
    }
    div.header h4.label {
        line-height: 78px;
    }
}

@media (max-width: 600px) {
    div.header img.logo {
```

```
        height: 62px;
        width: 110px;
    }
    div.header h4.label {
        line-height: 62px;
    }
}
```

Assim concluímos o estilo do cabeçalho. Salve o arquivo `index.css`. Caso seja necessário, este código encontra-se no repositório do livro e pode ser baixado diretamente em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/components/Header/index.css.

Lembra-se das características de uma PWA tratadas no *Capítulo 1*? Na ocasião, conversamos sobre a importância de criar um site que se ajuste ao tamanho do navegador/dispositivo.

Figura 4.12: História 1.11

Estamos começando a resolvê-la. Ainda nos restam alguns detalhes, mas a ideia é usar recursos responsivos do Pure.css e próprios (pela sintaxe `@media`).

4.4 CRIANDO O COMPONENTE DO NOVO USUÁRIO

Vamos ao componente `NovoUsuario`. Crie uma pasta chamada `NovoUsuario` dentro de `behappywith.me/front-`

`end/src/components` . Dentro dessa pasta, crie um arquivo `index.jsx` . Não há nenhuma novidade em relação à sintaxe React neste componente.

Por enquanto, esse componente será muito simples. Na primeira linha, ele importa a biblioteca do React:

```
import React from 'react'
```

Em seguida, codifica a criação de uma classe na sintaxe ECMAScript 6 com um único método:

```
class NovoUsuario extends React.Component {
   render() {
       // ...
   }
}
```

Na última linha do arquivo, vamos colocar a exportação com a sintaxe `export default` .

```
export default NovoUsuario;
```

O método `render()` terá apenas uma tag `<div>` vazia com uma classe de estilo chamada `center` . Lembre-se de que, como a sintaxe `class` é reservada pelo ECMAScript 6, a referência à classe `center` se dá por meio do atributo `className` . Veja a seguir o código completo do componente.

```
import React from 'react'

class NovoUsuario extends React.Component {
   render() {
       return (
           <div className="center">
           </div>
       );
   }
}
```

```
export default NovoUsuario;
```

Vamos discutir agora um detalhe de usabilidade. O componente `NovoUsuario` servirá de invólucro para todos os componentes dessa tela. Em dispositivos mobile, ele será imperceptível. Praticamente tomará grande parte, se não toda área, da viewport.

Nos navegadores de desktop, entretanto, ele será fundamental para uma boa apresentação do layout. A classe de estilo CSS `center` terá atributos para manter a `<div>` centralizada horizontalmente na tela e com comprimento e altura elásticos (no sentido de crescer de acordo com o conteúdo interno).

Note que nossa PWA terá comportamento semelhante, sempre centralizada no browser. Seguindo a recomendação da documentação do React, resolveríamos este problema criando classes *inline* para cada tela. Aqui faremos diferente. Como todos os invólucros das telas precisam responder a uma mesma estilização, será mais elegante criarmos a classe `center` no arquivo `index.css` geral do projeto do que replicar estilos em arquivos `index.css` dentro de cada componente.

Abra o arquivo `index.css` principal do projeto, disponível no diretório `behappywith.me/front-end/src/css` . Vamos começar pelo estilo básico e a seguir aplicar *media queries*.

A `<div>` centralizada terá algumas propriedades que merecem destaque. A propriedade `box-sizing` definida como `border-box` significa que a `<div>` considera as propriedades `padding` e `border` como parte de seu `width` e `height` . Em outras palavras, agora um `width` de 100 pixels decorado com

`border` de 1 pixel continuará tendo 100 pixels em vez de 102 pixels. Isso ajudará na conservação do tamanho mínimo da página.

Haverá também uma propriedade `display` definida como `table`. Esta, em conjunto com a propriedade `margin` definida como `auto`, permitirá que a `<div>` fique centralizada na tela.

Um último detalhe da propriedade `margin`: ela é capaz de receber quatro valores, que respectivamente representam o tamanho das margens superior, direita, inferior e esquerda. Os valores que forem omitidos automaticamente assumem uma margem nula (igual a zero).

As demais propriedades não merecem destaque. Entre com o trecho a seguir no arquivo `index.css`.

```
div.center {
    box-sizing: border-box;
    padding: 5px;
    display: table;
    margin: 0 auto;
    min-width: 320px;
    margin-top: 92px;
}
```

Vamos acrescentar também um pequeno ajuste via *media queries*, que será necessário para reduzir a distância entre o topo da página em viewports menores. Coloque o trecho a seguir no final do arquivo `index.css`.

```
@media (max-width: 992px) {
    div.center {
        margin-top: 78px;
    }
}

@media (max-width: 600px) {
    div.center {
```

```
        margin-top: 62px;
    }
}
```

Lembre-se de que esse e outros códigos podem ser baixados do GitHub. Este, em especial, está disponível em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/css/index.css/.

Vamos agora ajustar o `App.jsx` para comportar os novos componentes.

Ajustando o componente App

Após a criação desses novos componentes, o código de `App.jsx` ficou obsoleto. Vamos ajustá-lo para simplesmente importar os componentes `Header` e `NovoUsuario`, e apresentá-los em seu método `render()`. Abra-o e entre com o código apresentado a seguir. A diferença está apenas nas importações e no retorno do método `render()`.

```
import React from 'react';
import Header from './Header';
import NovoUsuario from './NovoUsuario';

class App extends React.Component {
    render() {
        return (
            <Header />
            <NovoUsuario />
        );
    }
}

export default App;
```

Atenção: há um erro nesse código. O React tem uma limitação que proíbe duas ou mais tags na renderização do componente.

Vamos entendê-la e resolvê-la no próximo tópico.

Executando no ambiente de desenvolvimento

Ao executar o servidor de desenvolvimento, vamos nos deparar com um erro.

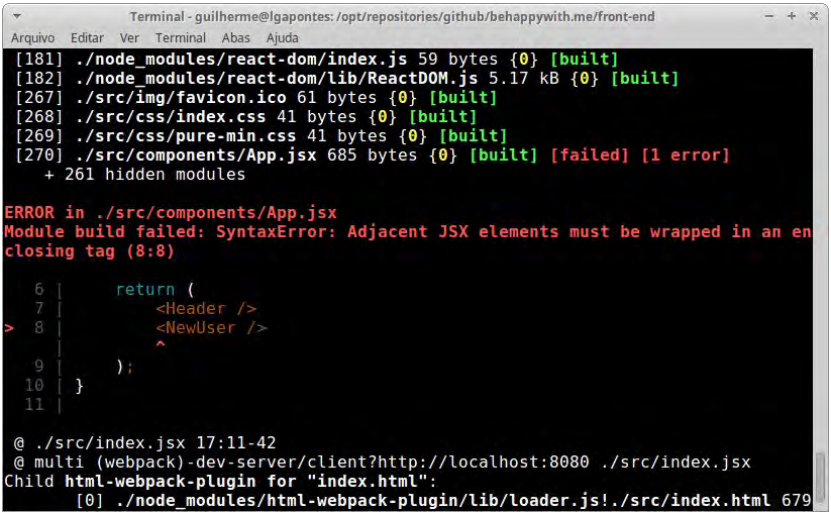

Figura 4.13: Erro no componente App

Vamos concentrar-nos no trecho do console reproduzido a seguir.

```
ERROR in ./src/components/App.jsx
Module build failed: SyntaxError: Adjacent JSX elements must be w
rapped in an enclosing tag (8:8)
```

Esta é uma das principais restrições do React: o código JSX só pode ter um elemento retornado por método `render()`. Para viabilizar trechos com mais de uma tag, precisamos encapsular os elementos dentro de uma `<div>` (ou `<span>`, ou `<h1>` etc.).

Ou seja, para corrigir o erro de `App.jsx`, coloque `<Header>` e `<NovoUsuario>` dentro de uma `<div>`. Observe que não é necessário desligar o servidor de desenvolvimento. Veja a alteração no trecho a seguir (omitimos as importações e a sintaxe de exportação).

```
class App extends React.Component {
    render() {
        return (
            <div>
                <Header />
                <NovoUsuario />
            </div>
        );
    }
}
```

Problema resolvido. Execute o servidor de desenvolvimento (caso esteja desligado) e acesse a URL http://localhost:8080. Você verá algo semelhante a:

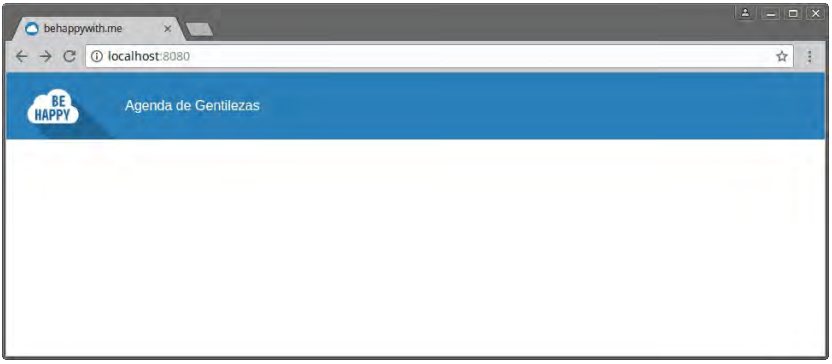

Figura 4.14: Cabeçalho da PWA

Tudo parece correto – para usuários de desktop. Mas lembre-se do mobile-first: como será que está nossa aplicação em um aparelho móvel? Para testarmos isto, não é necessário acessá-la em

um celular. Podemos simplesmente ativar a ferramenta de desenvolvimento do Google Chrome (ou qualquer outro navegador) e exibir a página com a representação reduzida (mobile).

Para isso, clique em `F12` e pressione o ícone *Toggle Device Toolbar* no canto superior esquerdo da ferramenta de desenvolvimento.

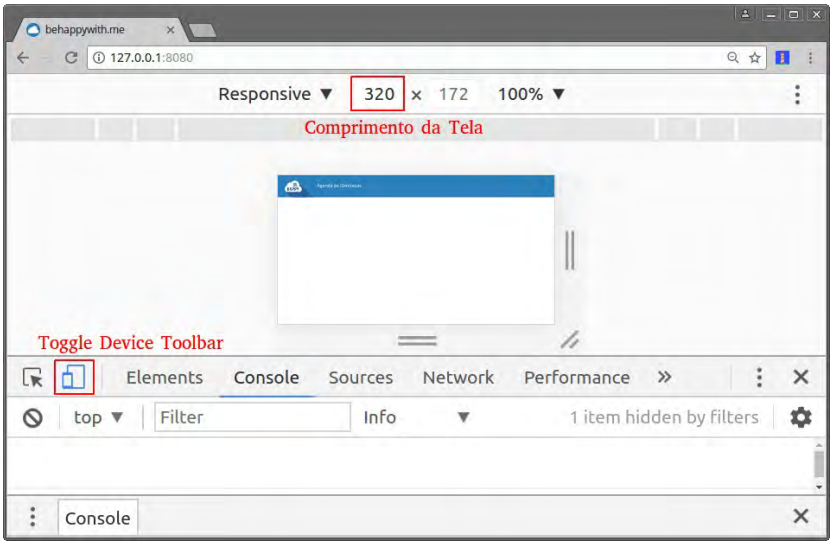

Figura 4.15: Toggle Device Toolbar e comprimento da tela

Na parte superior da página, nos campos *Responsive*, ajuste o comprimento da tela (primeiro campo de texto) para 320 pixels. Esta será a configuração mínima aceita em nossa PWA. Veja que o cabeçalho encolheu a ponto de ficar ilegível.

Na verdade, isso não aconteceu só com o cabeçalho. Toda a página foi reduzida. Isso é culpa do React? Como vamos resolver?

Veremos ambas respostas no próximo tópico.

Ajustando o index.html

A redução da página não é *culpa* do React. Na verdade, por default, os navegadores móveis renderizam as páginas em uma viewport maior do que a tela. Isso ocorre para que não seja necessário reduzir os layouts em uma pequena janela e quebrar a maioria dos sites não responsivos.

O problema é que nosso site **é** responsivo. A representação aplicada pelo *Toggle Device Toolbar* em uma tela de 320 pixels nos mostra o que ocorreria em um dispositivo mobile. Como vamos resolver?

Existe uma meta tag para controlar o comportamento da viewport dos navegadores. Ela serve justamente para fixar o tamanho e a escala da viewport. Esta metatag deve entrar no `<head>` do arquivo `index.html`. Altere-o acrescentando a sintaxe abaixo no início da `<head>`.

```
<meta name="viewport" content="width=device-width, initial-scale=
1.0">
```

O atributo `name` indica qual metatag será configurada (no nosso caso, a `viewport`). O atributo `content` está definido com as configurações `width e initial-scale`. O `width` definido como `device-width` avisa ao navegador que nossa PWA se ajustará ao dispositivo. O `initial-scale` basicamente define a estala (zoom) inicial do site. O valor `1.0` indica que um pixel estilizado (CSS) será equivalente a um pixel da viewport.

Com ambos os parâmetros, o comportamento inicial da PWA

em dispositivos móveis será adequado. A configuração que fizemos aqui reflete um dos requisitos das aplicações progressivas, este que registramos na história a seguir.

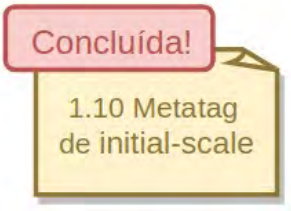

Figura 4.16: História 1.10

Problema de renderização resolvido. Salve os arquivos, acesse novamente o endereço http://localhost:8080 com o Toggle Device Toolbar ativado, e veja o resultado da metatag viewport.

4.5 CRIANDO O COMPONENTE LABEL

O objetivo do componente `Label` é simplesmente exibir um texto auxiliar às tags de entrada da aplicação. Como se trata de um componente novo, crie um diretório chamado `Label` dentro de `behappywith.me/front-end/src/components` e, dentro dele, crie um novo arquivo chamado `index.jsx`.

Vamos começar importando a biblioteca do React.

```
import React from 'react'
```

Assim como o `Header`, o componente `Label` será *stateless*. Isso nos direciona à criação de uma `function`. Por enquanto, ela receberá um parâmetro chamado `props`, que representa as propriedades do componente.

```
export default function Label(props) {
   // ...
}
```

Nomeamos o parâmetro como `props` para respeitar a convenção do objeto (JSON) `props` criado automaticamente pelo React nos componentes criados com `class` (todo o componente `class` especializado de `React.Component` possui um atributo chamado `props`).

Mas como exatamente passamos propriedades para os componentes? Através de atributos na tag do componente. Vejamos um exemplo:

```
<Label qualquerAtributo="qualquerValor" />
```

Na chamada ao componente `Label`, o atributo `qualquerAtributo` será um elemento presente no objeto `props` passado como parâmetro da `function Label()`. O parâmetro `props` é um típico objeto JavaScript, cujos atributos poderão ser acessados com `props.qualquerAtributo`.

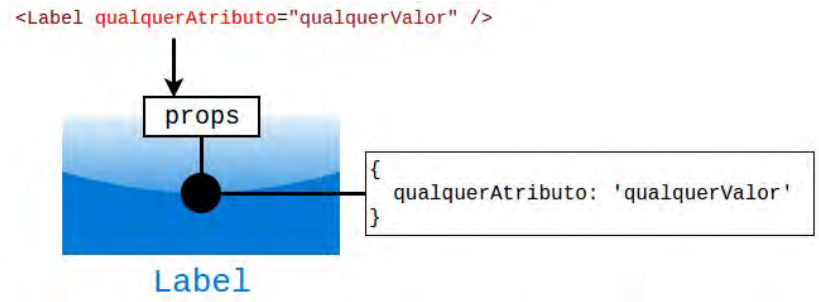

Figura 4.17: Passagem de propriedades

Um dos objetivos das propriedades é usá-las dentro do código JSX. Para acessar valores de qualquer variável dentro do JSX,

precisamos simplesmente envolvê-las com `{}` (chaves). Se quiséssemos, por exemplo, acessar o valor da propriedade `qualquerAtributo` dentro da função `Label`, faríamos um código semelhante ao disposto a seguir.

```
export default function Label(props) {
   return (
       <label>{props.qualquerAtributo}</label>
   );
}
```

Primeira versão do Label

Agora que conhecemos a dinâmica da passagem de propriedades, precisamos decidir quais são importantes para o componente `Label`. Essa decisão é tomada com base no *objetivo* do componente, que neste caso simplesmente encapsulará a tag `<label>`. Isso nos dá uma noção de que precisaremos passar no mínimo os mesmos atributos recebidos pela tag `<label>`, que, *a priori*, são o `for` e o valor do texto da `<label>`.

O atributo `for` é um recurso de usabilidade que auxilia no clique (ou toque) do componente associado. Quando definimos `for` com o `id` de uma tag `<input>`, ao clicar sobre o texto da `<label>`, o foco será direcionado ao componente `<input>`.

O valor do texto da `<label>` deve ficar entre as tags de abertura e fechamento. A primeira `Label` do formulário receberá como valor a `string` *Quem é você?*. Vamos chamar essa propriedade de `texto`.

```
export default function Label(props) {
   return (
       <label for={props.for}>{props.texto}</label>
   );
```

```
}
```

O componente `NovoUsuario` vai agrupar todos os elementos de um `<form>` (formulário) de cadastro de usuários. O `Label`, apesar de não armazenar dados para o cadastro, faz parte desse formulário.

Com base nisso, precisamos então ajustar o código de `NovoUsuario`. Vamos começar importando o componente `Label`.

```
import Label from '../Label'
```

Agora, no método `render()`, vamos acrescentar à tag `<form>` e, dentro dela, uma chamada ao novo componente `Label`.

```
import React from 'react'
import Label from '../Label'

class NovoUsuario extends React.Component {
    render() {
        return (
            <div className="center">
                <form>
                    <Label for="nome" texto="Quem é você?" />
                </form>
            </div>
        );
    }
}
```

O valor da propriedade `for` ficou como `nome`, porque este será o `id` da `<input>` associada. Execute o servidor de desenvolvimento e observe que o código é compilado com sucesso. Contudo, ao acessar http://localhost:8080, vamos nos deparar com o seguir erro:

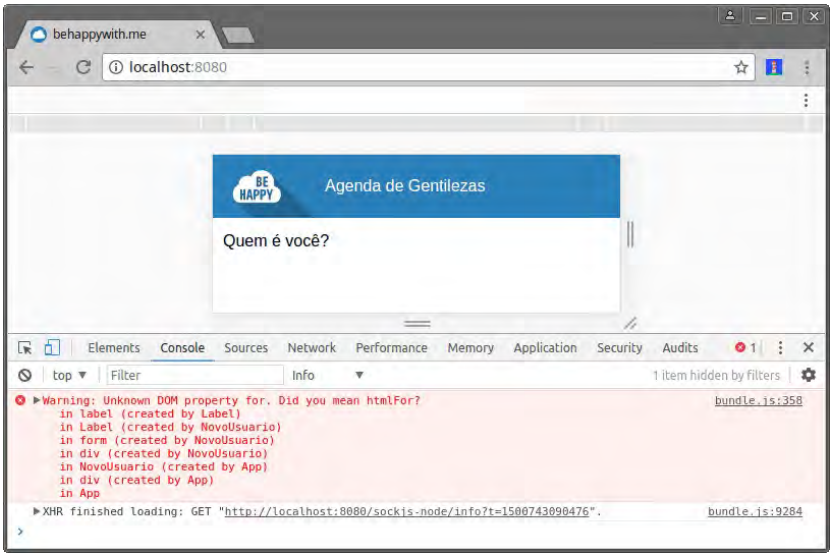

Figura 4.18: Erro no uso do for

O erro (na verdade, *warning*) está reproduzido a seguir:

```
Warning: Unknown DOM property for. Did you mean htmlFor?
    in label (created by Label)
    in Label (created by NovoUsuario)
    in form (created by NovoUsuario)
    in div (created by NovoUsuario)
    in NovoUsuario (created by App)
    in div (created by App)
    in App
```

Você se lembra da adaptação que o React fez com a sintaxe `className`? Assim como o `class`, o comando `for` é reservado pelo JavaScript. Para contornar, os componentes React adotaram um atributo chamado `htmlFor`. Quando renderizado, por baixo dos panos, o React vai transformá-lo no verdadeiro atributo `for` utilizado pela tag `<label>`. Vamos corrigir este problema.

Segunda versão do Label (sem erro)

A correção do problema é simples: dentro do código do componente `Label`, basta alterarmos a sintaxe `for` para `htmlFor`.

```
export default function Label(props) {
    return (
        <label htmlFor={props.htmlFor}>{props.texto}</label>
    );
}
```

De forma semelhante, vamos ajustar o código de `NovoUsuario`. No código com erro, estávamos passando uma propriedade chamada `for`. Agora, o componente `Label` aguarda a propriedade `htmlFor`.

```
class NovoUsuario extends React.Component {
    render() {
        return (
            <div className="center">
                <form>
                    <Label htmlFor="nome" texto="Quem é você?" />
                </form>
            </div>
        );
    }
}
```

Só uma observação: chamamos a propriedade do componente `Label` de `htmlFor` só para facilitar no entendimento de seu uso. Poderíamos chamá-la de `nomeDaPropriedadeFor`. O mesmo não é válido para a tag `<label>` interna no componente, pois ela espera uma propriedade `htmlFor`. Se passássemos `nomeDaPropriedadeFor`, teríamos um erro (`<label>` não conhece essa propriedade).

Acesse novamente o endereço http://localhost:8080 e veja que

o erro foi resolvido.

Estilização inline dos componentes

Nosso `Label` já possui propriedades para o `htmlFor` e seu valor. Vamos agora acrescentar uma propriedade para indicar se o campo associado foi ou não preenchido corretamente. Por exemplo, o componente `Label` de *Quem é você?* estará associada ao `<input id="nome">`. Se este campo não for preenchido, marcaremos ambos (`<input>` e `<label>`) de vermelho.

Já vimos como aplicar estilos pela definição da classe CSS. Entretanto, o React oferece uma outra alternativa de estilo inline com o atributo `style`. Diferentemente do atributo `style` da especificação original do HTML, as tags do React recebem um objeto JSON cujas propriedades são semelhantes às de uma classe CSS – com exceção de que aqui, em vez do uso do `-` (hífen) para conectar as chaves com mais de uma palavra, usaremos o camelCase.

Resumindo, a chave chamada `background-image` do código CSS comum deverá ser escrita como `backgroundImage`. Na prática, é comum criarmos um objeto JavaScript com as propriedades do CSS em camelCase e chamá-lo na sintaxe JSX. Veja um exemplo a seguir.

```
render() {
    const estilo = {
        color: '#ff0000'
    };

    return (
        <h1 style={estilo}>Título vermelho</h1>
    );
}
```

Note que o objeto `estilo` é referenciado no JSX normalmente pelo uso de `{}` (chaves). Com esse recurso, podemos passar outro parâmetro para o componente e, dependendo de seu valor, atribuir ou não o estilo inline na `<label>`.

É bem simples, mas, por acaso, você notou a palavra `const` na declaração da variável? Conversaremos sobre essa sintaxe agora.

Declaração de variáveis no ECMAScript 6

A nova sintaxe do JavaScript trouxe-nos duas novas alternativas para declaração de variáveis: `let` e `const`. A definição `var` continua presente, mas sua coexistência com o `let` tornou-a obsoleta. O `var` possui o que chamamos de *escopo de função*. Isso é de certa forma incômodo aos programadores de linguagens como Java ou C#, que estão habituados à declaração por *escopo de blocos*.

Em poucas palavras, o *escopo por função* é problemático, principalmente quando as declarações com `var` são realizadas fora das funções – em um escopo global. Isso significa que qualquer variável declarada *fora de uma função* terá visibilidade em todas as funções e blocos do código, causando problemas de manutenibilidade e erros de variáveis não definidas ou com valor sobrescrito.

Com o `let`, esse comportamento é contornado, pois, como ele trabalha com *escopo de bloco*, se tentarmos acessá-lo fora do bloco, o JavaScript vai retornar um erro `ReferenceError: nomeVariavel is not defined`. Além disso, se tentarmos declarar a mesma variável mais de uma vez, vamos nos deparar

com o erro `SyntaxError: Identifier 'nomeVariavel' has already been declared`.

A sintaxe `const` também possui essas restrições, porém, como a própria abreviação esclarece, o valor torna-se uma constante e não poderá ser alterado. Se tentarmos alterar o valor de uma `const`, o sistema retornará `TypeError: Assignment to constant variable`.

Resumindo, o `var` deve ser desencorajado em relação ao `let` e, sempre que houver necessidade de constantes, deve-se usar o `const`.

Terceira versão do Label (estilização)

Vamos agora alterar o código do componente `Label` para receber uma propriedade chamada `valorInvalido`. Assim como o HTML, a sintaxe JSX aceita propriedades do tipo `boolean` que podem simplesmente ser declaradas sem a passagem explícita de um valor.

Na prática, isso significa que, quando declaramos `<Label valorInvalido />`, o valor de `props.valorInvalido` dentro do componente será `true`. Em contrapartida, se ele não for definido (`<Label />`), a propriedade `props.valorInvalido` não terá nenhum valor definido (`undefined`). Isso nos permite fazer uma condição simples com o valor de `props.valorInvalido` para definir ou não a estilização vermelha para a `<label>`.

Abra o arquivo `index.jsx` dentro da pasta `Label`. Se `props.valorInvalido` for `true`, ele será vermelho (`#d50000`); do contrário, será cinza (`#444444`). Isso pode ser realizado com

uma condição ternária dentro da constante `estilo`.

Vamos incluir esse código no início da `function` do componente e, em seguida, aplicá-la à tag `<label>` pela propriedade `style`.

```
export default function Label(props) {
    const estilo = {
        color: props.valorInvalido ? '#d50000' : '#444444'
    };

    return (
        <label
            style={estilo}
            htmlFor={props.htmlFor}>
            {props.texto}
        </label>
    );
}
```

Para testar, altere o código do arquivo `index.jsx`, do componente `NovoUsuario`, incluindo a propriedade `valorInvalido` na chamada do componente.

```
<Label htmlFor="nome" texto="Quem é você?" valorInvalido />
```

Isso resultará na seguinte renderização:

Figura 4.19: Label sem o erro do for

4.6 REAJUSTANDO O COMPONENTE NOVOUSUARIO

Para finalizarmos, vamos acrescentar estilos do Pure.css no formulário do componente `NovoUsuario`. Usaremos duas classes: `pure-form` e `pure-form-stacked`. A classe `pure-form` define o estilo básico dos formulários. Por sua vez, a classe `pure-form-stacked` realiza a disposição das tags uma abaixo da outra (o `<input>` ficará sob a `<label>`).

Abra o arquivo `index.jsx` do `NovoUsuario` e acrescente a chamadas dessas classes, conforme o código exibido a seguir. Note que a propriedade `valorInvalido` foi retirada. No futuro, criaremos uma lógica para acrescentá-la em casos de falha no preenchimento do formulário.

```
class NovoUsuario extends React.Component {
    render() {
        return (
            <div className="center">
                <form className="pure-form pure-form-stacked">
                    <Label htmlFor="nome" texto="Quem é você?" />
                </form>
            </div>
        );
    }
}
```

O bom de usar um framework CSS é que não precisamos nos preocupar com o código por trás dessas classes. O Pure.css oferece essas classes a formulários, e nós simplesmente usamos.

4.7 CONCLUSÃO

Aprendemos como a prototipação auxilia na definição dos requisitos de um projeto React e nas boas práticas para definir a granularidade dos componentes. Discutimos sobre a experiência do usuário associada ao componente Image Scroller e por que é importante focar em dispositivos mobile sem nos esquecermos dos dispositivos não mobile.

Construímos gradativamente os componentes `App`, `NovoUsuario`, `Header` e `Label`. O componente `App` ficou como principal mantenedor da aplicação, por enquanto apenas responsável por renderizar os componentes `Header` e `NovoUsuario`. Durante a criação do cabeçalho, ajustamos também a renderização da aplicação pela metatag viewport.

Por fim, desenvolvemos o componente `Label`, considerando o uso das novas declarações `let` e `const` do ECMAScript 6 e a estilização inline. Observamos como passar propriedades para um

componente e a limitação de uma única tag retornada pelo método `render()`.

Todos esses conceitos aqui apresentados serão necessários para construirmos os próximos componentes, mais sofisticados, tratados massivamente nos próximos capítulos.

CAPÍTULO 5

COMPONENTES COM ESTADO E FLUXO DE EVENTOS

No capítulo anterior, concluímos a criação do componente `Label`. Iniciamos também com o código dos componentes `NovoUsuario` e `App`. Digo *iniciamos* porque muitos ajustes ainda serão feitos.

Neste capítulo, vamos desenvolver a entrada do nome (`<input>`) e a `div` de escolha do gênero. Na prática, isso significa construir os componentes `Input`, `GenderButton` (botão do gênero), `GenderImage` (imagem do gênero) e `GenderSelector` (seletor do gênero). Veja a prototipação adiante.

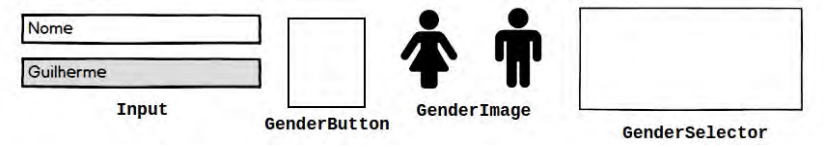

Figura 5.1: Componentes deste capítulo

Vamos começar esclarecendo como o React trabalha com os

estados dos componentes.

5.1 ESTADOS DOS COMPONENTES

Neste capítulo, precisaremos alcançar um novo patamar da componentização: a utilização dos estados. Através do estado (`state`), o componente armazena dados internamente, com intuito de utilizá-los para ajustar sua renderização apropriadamente.

Estado inicial de um componente

Lembra do código `Exemplo` exposto no *Capítulo 2*? Vamos agora retratá-lo na íntegra. Como nada será acrescentado ao arquivo `exemplo.html` , cujo esqueleto HTML (código React no navegador) está implementado, não vamos reproduzi-lo aqui.

O componente é um simples `<h1>` que troca sua cor de fundo de acordo com um contador e recebe como parâmetro (propriedade) o `texto` que será escrito. Os valores das variáveis `cor` e `contador` serão armazenados no estado do componente.

A variável `contador` será acrescida sempre que o usuário tocar (ou clicar) no `<h1>` . Se o resultado de `(contador % 2)` for igual a zero, `cor` será `red` ; do contrário, será `blue` .

Tais detalhes nos direcionam à criação de um componente *stateful*, ou seja, com a sintaxe `class` . Componentes sem estado (`function`) recebem suas propriedades como parâmetros da função. No caso de componentes com estado (`class`), passaremos as propriedades no `constructor` do objeto. Como o objeto `React.Component` também espera recebê-las, logo no

início utilizaremos a sintaxe `super()` para chamar o `constructor` da *superclasse*.

```
class Exemplo extends React.Component {
    constructor(props) {
        super(props);
    }
}
```

Se vamos utilizar o estado do componente para renderizar a cor do `<h1>`, é importante definirmos um estado inicial. Faremos isso no `constructor`, conforme indicado a seguir.

```
constructor(props) {
    super(props);
    this.state = {
        cor: 'red',
        contador: 0
    };
}
```

Veja que `state` recebe um objeto JavaScript diretamente através de uma atribuição. Esta é uma **exceção**. No geral, devemos considerar `this.state` como um elemento imutável, disponível apenas para leitura. Na maioria dos casos, vamos atualizar o estado do componente com o método `this.setState()`.

Um detalhe importante a considerar é que o método `this.setState()` é assíncrono, o que significa que ele não altera instantaneamente o valor de `this.state`. Na prática, ler o valor de `this.state` imediatamente após a atualização pode retornar o valor do estado antes da atualização.

O MUNDO DAS CALLBACKS

O JavaScript é uma linguagem de I/O não bloqueante. Na prática, isso significa que, se você fizer uma operação de entrada e saída – tal como acessar uma API RESTful de forma assíncrona –, a linha de execução não será interrompida. Às vezes, entretanto, se faz necessário aguardar o retorno de uma dessas funções assíncronas para posteriormente executar uma ação no código. A função JavaScript que será executada quando a operação não bloqueante for concluída é convencionalmente chamada de *callback*.

```
function funcaoNaoBloqueante(callback) {
    // API assíncrona de consulta ao servidor
    // Cria uma variável resultado
    callback(resultado);
}

funcaoNaoBloqueante(function(resultado) {
    // Faz alguma coisa com resultado
});
```

Neste caso, a `function` anônima passada por parâmetro para a função `funcaoNaoBloqueante()` é uma *callback*. Este exemplo é bem simplório, mas mostra uma boa representação de como as funções *callback* são usadas.

Se você precisar de uma leitura síncrona, há uma assinatura do método `this.setState()` que recebe uma função *callback* como segundo parâmetro. Neste caso, dentro da *callback*, o valor do estado estará devidamente atualizado.

Com os conceitos por trás do `this.setState()` introduzidos, vamos aplicá-los ao componente de exemplo.

Alterando o estado de um componente

Exploraremos agora um caso de leitura e escrita do estado fora do `constructor`. Como vimos, o valor de `contador` influenciará no valor de `cor`, que, por sua vez, será utilizado para definir a cor de fundo da tag `<h1>`. Esta lógica será codificada no método `trocarCor()`, pelo qual obtemos o valor de `contador` de `this.state`, realizamos a verificação de seu valor e atribuímos o estado ao componente.

```
class Exemplo extends React.Component {
    // ...
    trocarCor(e) {
        let contador = this.state.contador;
        let cor = ( ++contador % 2 ) == 0 ? 'red' : 'blue';
        this.setState({
            cor: cor,
            contador: contador
        });
    }
}
```

Resta-nos o método `render()`. No início, especificaremos a constante `estilo` com dois elementos: `padding` e `background`. O que nos interessa aqui é a cor de fundo. Em vez de colocarmos algo estático, vamos obter a variável `cor` de `this.state` para defini-la.

O retorno de `render()` conterá a tag `<h1>`. Além da estilização com a constante `estilo` e do valor definido a partir da propriedade `texto`, faremos a declaração do evento `onClick` apontar para o método `trocarCor()`.

```
class Exemplo extends React.Component {
    // ...
    render() {
        const estilo = {
            padding: 10,
            background: this.state.cor
        }
        return (
            <h1
                style={estilo}
                onClick={this.trocarCor.bind(this)}
            >
                {this.props.texto}
            </h1>
        )
    }
}
```

Há dois detalhes importantíssimos neste código. *A priori*, note que a sintaxe é `onClick`, com *camelCase* em vez da especificação original `onclick` da W3C. O outro detalhe é a necessidade do uso do `bind()`. Só para reforçar, o `bind()` é uma *bound function*, útil para permitir repassar o objeto corrente (`this`) para uma função.

USO DO MÉTODO `BIND()`

Primeiro precisamos entender que, no JavaScript, cada função executada é associada a um *contexto de execução*. O uso da palavra reservada `this` permite-nos acessar este contexto. Quando acessamos o `this` dentro de objetos iguais ao componente `Exemplo`, o contexto de execução é o próprio objeto (`Exemplo`).

Então por que precisamos do `bind()`? Se você está habituado a trabalhar com JavaScript, sabe que a sintaxe `onClick={this.trocarCor}` atribui a função *callback* `trocarCor()` ao evento `onClick` **sem** repassar o contexto de execução do objeto `Exemplo`. Na prática, isso significa que, sem o `bind()`, o `this` dentro do método `trocarCor()` seria `undefined`.

Como regra geral, para que as funções *callback* tenham acesso ao contexto de execução do objeto, precisamos decorá-las com a chamada ao método `bind()`.

Vamos agora evoluir um pouco nossa discussão sobre uma alternativa ao uso do `bind()`, as *arrow functions*, outro belo recurso do ECMAScript 6.

Arrow functions

As *arrow functions* oferecem uma nova forma de codificar funções na linguagem JavaScript, muito mais enxuta e com escopo

de execução mais direto.

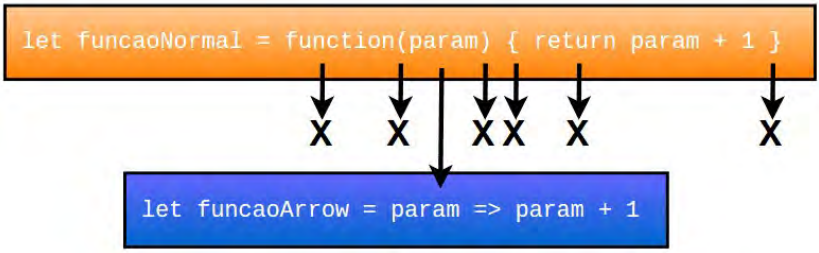

Figura 5.2: Sintaxe das arrow functions

A representação anterior representa bem sua melhoria sob o ponto de vista da sintaxe, que pode ser resumida na assinatura sem a palavra `function` e os `()` (parênteses) ao redor dos parâmetros (em casos em que exista apenas um parâmetro), a retirada das sintaxe `{}` (chaves) e do `return` quando a função possui um único comando.

Há também a melhoria sob a forma de execução, com a qual o comando `this` deixa de ser definido dinamicamente (comportamento padrão do JavaScript) e torna-se uma referência estática ao objeto no qual a função foi declarada. Como essa regra também se aplica às *callback*, e as *arrow functions* são efetivamente aplicadas em funções desse tipo, o ganho sob essa perspectiva do escopo de execução é muito relevante.

Na prática, isso nos dá a liberdade de refatorar o método `trocarCor()` com *arrow functions* e dispensar o uso do `bind()`. Neste caso, o método `trocaCor()` receberá uma *arrow function*, e o evento `onClick` do método `render()` não precisará mais da sintaxe `bind()`.

```
class Exemplo extends React.Component {
```

```
    // ...
    trocarCor = (e) => {
        let contador = this.state.contador;
        let cor = ( ++contador % 2 ) == 0 ? 'red' : 'blue';
        this.setState({
            cor: cor,
            contador: contador
        });
    }
    render() {
        // ...
        return (
            <h1
                style={estilo}
                onClick={this.trocarCor}
            >
                {this.props.texto}
            </h1>
        )
    }
}
```

Há infinitas possibilidades com as *arrow functions*, mas, para o escopo desta obra, o que tratamos aqui já é suficiente para continuarmos.

5.2 CRIANDO O COMPONENTE INPUT

De volta aos componentes do projeto, o componente `Input` será bem simples porém robusto. Seguindo as boas práticas, precisamos nos preocupar em deixá-lo reutilizável para todas as nossas necessidades. Em termos técnicos, ele será composto por:

1. Um mecanismo para que outros componentes obtenham o valor digitado na tag `<input>` .
2. Um atributo `id` da tag `<input>` ;
3. Do valor do `placeholder` (texto exibido dentro da tag até o momento em que o usuário digitar);

4. Do tamanho máximo de caracteres;
5. Um indicativo para sinalizar se ele é ou não `read-only` ;
6. Do estilo para indicar se ele está ou não corretamente preenchido;
7. De um valor default;

Vamos construí-lo aos poucos a partir de agora.

Input precisa manter um estado?

Esta é uma das discussões mais importantes deste capítulo. Parece uma decisão simples: como a tag `<input>` será um elemento interno ao componente `Input` , o fluxo mais lógico seria armazenar um estado guardando o valor digitado e, posteriormente, fornecê-lo aos demais componentes. Em outras palavras, poderíamos construir algo semelhante à representação a seguir.

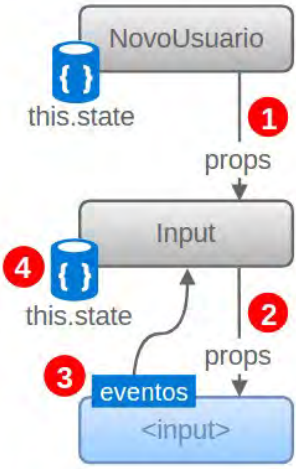

Figura 5.3: Fluxo de dados entre NovoUsuario e Input

O item 1 da imagem expõe a passagem das propriedades entre `NovoUsuario` e o componente `Input`, que, por sua vez, vai enviá-los à tag `<input>` (item 2). O item 3 trata-se de um evento do tipo `onChange` aplicado à tag `<input>`. Sempre que o usuário alterar o valor de `<input>`, vamos tratar do evento e atualizar o estado do componente.

Guardar o valor da tag `<input>` no estado do componente (item 4) seria uma tarefa relativamente simples, que veremos na criação dos próximos componentes. O que precisamos discutir aqui é como resgatar esse valor em `NovoUsuario`. E para entendermos a melhor forma de fazer isso, precisamos manter o foco na forma como o React se propõe a trabalhar.

React é um framework da camada de apresentação. Ponto. Admitir isso é o primeiro passo para entender que todos os seus recursos foram oferecidos para possibilitar a construção desta camada de forma organizada e manutenível. Os estados dos componentes podem guardar momentaneamente um valor de negócio para personalizar a renderização, mas *a aplicação* em si é a verdadeira responsável por isso.

Olhando de outro ângulo, se o método `render()` precisar de valores salvos em `this.state` para decidir como expor seus elementos, é bem provável que o componente seja *stateful*. Por outro lado, se estivermos guardando dados em `this.state` para em algum momento provê-los para a aplicação, *talvez* devêssemos ter um componente *stateless*.

O `nome` do usuário é uma informação necessária para toda a aplicação, ou ela pode ser descartada quando o usuário acessar outra tela (e o `Input` for destruído)? Obviamente que é uma

informação para a PWA funcionar. O componente `NovoUsuario`, por questões semânticas, deve ser capaz de criar uma estrutura (JSON) para organizar todos os dados do usuário. E é por isso que ele precisa acessar o valor do `Input`.

No *Capítulo 2*, citamos brevemente que a passagem de parâmetros para os componentes, via `props`, pode ser realizada para *dados* ou *funções*. Até o momento, repassamos apenas *dados*. Veremos agora como a passagem de funções *callback* para os componentes filhos pode nos ajudar a organizar melhor o **local correto** de armazenamento dos estados dos componentes.

Se o componente `NovoUsuario` é o responsável por criar os dados do usuário, por que não deixá-lo armazenar esses dados em seu estado?

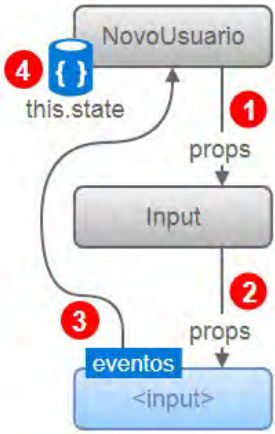

Figura 5.4: Obter valor de Input por eventos

Todas as `props` passadas no item 1 serão repassadas à tag `<input>` (item 2). Uma dessas propriedades poderia ser, por

exemplo, uma função *callback* para o evento `onChange` da tag `<input>`. Dessa forma, sempre que o valor for alterado, o evento vai disparar a função *callback* do `NovoUsuario`, obter o valor da tag `<input>` e guardá-lo no estado do componente (item 4).

Se fixarmos essa lógica, fica fácil compreender que, na verdade, o componente `Input` será *stateless*, cuja implementação por `function` será iniciada no próximo tópico.

Primeira versão do Input

De forma semelhante aos demais componentes, `Input` também precisará de um diretório. Crie-o (com o nome `Input`) dentro de `behappywith.me/front-end/src/components`. Em seguida, crie um arquivo chamado `index.jsx` dentro desse diretório. Começaremos pelo básico, definindo as importações, a sintaxe da função `Input` e sua exportação.

```
import React from 'react'

export default function Input(props) {
}
```

A função retornará apenas uma tag `<input type="text">` com algumas propriedades. Vamos começar com a recepção do `id`, do `maxLength` (note a sintaxe *camelCase*) e do `placeholder` (não há *camelCase* porque esta é uma palavra única). Por enquanto, vamos repassá-las **uma a uma** à tag `<input>`.

Como observação, a disposição dos campos de forma indentada após a abertura `<input` e fechamento `/>` da tag é uma boa prática de codificação, segundo às regras do Airbnb React/JSX Style Guide.

```
export default function Input(props) {
    return (
        <input
            type="text"
            id={props.id}
            placeholder={props.placeholder}
            maxLength={props.maxLength}
        />
    )
}
```

Por enquanto, possuímos apenas três propriedades, mas perceba que seria inviável se tivéssemos de repassar um número muito grande de parâmetros. Veremos no próximo tópico uma alternativa mais adequada.

Operador spread

Estudaremos agora outro recurso do ECMAScript 6: o *Spread Operator* (operador de propagação). A ideia principal é propagar (ou expandir) uma estrutura de dados ao passá-la para um método receptor. Um array `a` definido com `['valor1',2,'tres',4]` poderia ser automaticamente expandido para métodos que esperam vários argumentos (como o `console.log()`, por exemplo) pela adoção da sintaxe `...` (três pontos) antes do nome da estrutura de dados (`...a`)

Esta situação assemelha-se muito ao caso vivenciado no componente `Input`. As propriedades poderiam ser repassadas diretamente, sem relacioná-las uma a uma. Se reescrevermos o retorno com a sintaxe do *spread operator*, além de o código ficar mais elegante, podemos acrescentar outras propriedades sem impactar o código interno do componente.

```
export default function Input(props) {
    return (
```

```
        <input
            type="text"
            {...props}
        />
    )
}
```

É claro que, neste caso, as propriedades esperadas pela tag `<input>` devem ser literalmente as mesmas informadas na chamada de `Input`. Em outras palavras, ao usar o *spread operator*, se passarmos um atributo `id`, uma propriedade de mesmo nome será repassada ao próximo componente. Se a tag `<input>` não *conhecer* todas as propriedades, receberemos um erro.

Segunda versão do Input

Vamos agora acrescentar as propriedades `readOnly`, um estilo para destacá-lo quando o valor estiver inválido, e um valor default. Começaremos por `readOnly` e `defaultValue`.

Assim como a `maxLength`, essas propriedades foram convertidas em um formato *camelCase* e podem ser repassadas diretamente por meio do *spread operator*. Em outras palavras, não precisamos alterar **nada** na chamada à tag `<input>`. Para usá-las, precisaremos apenas indicá-las no código de `NovoUsuario`, conforme veremos em breve.

O próximo passo é criar uma flag informando se o campo está ou não inválido. A exemplo do componente `Label`, estudado no *Capítulo 4*, passaremos uma propriedade chamada `valorInvalido` e, a partir dela, ajustaremos o estilo da tag `<input>`.

No que tange aos atributos de estilização, precisamos apenas pintar o fundo de vermelho claro (`#ffcdd2`) e a borda de vermelho (`#d50000`). Podemos organizá-los em uma classe de estilo inline exatamente antes do retorno da função.

```
export default function Input(props) {
    const estilo = {
        borderColor: props.valorInvalido ? '#d50000' : '#cccccc',
        backgroundColor: props.valorInvalido ? '#ffcdd2' : '#ffff
ff'
    };

    return (
        <input
            type="text"
            style={estilo}
            {...props}
        />
    )
}
```

Tudo parece correto, mas se alterássemos o componente `NovoUsuario` e executássemos o servidor de desenvolvimento, esta versão de código do componente `Input` provocaria o seguinte aviso (*warning*):

Figura 5.5: Warning da propriedade valorInvalido

Qual o motivo deste *warning*? Lembre-se de que estamos repassando as propriedades através do *spread operator*. Isso significa que **tudo** o que for definido na chamada de `Input` será repassado. A tag `<input>` do React, todavia, não sabe o que fazer

com a propriedade `valorInvalido`. Este é o motivo do *warning*.

Então, pela lógica, poderíamos simplesmente excluir a propriedade indesejada com o comando `delete` (que permite excluir um atributo de um objeto). *Poderíamos*, mas infelizmente isso não é tão simples. Se tentássemos excluí-la, com a sintaxe `delete props.valorInvalido;`, iríamos nos deparar com outro problema.

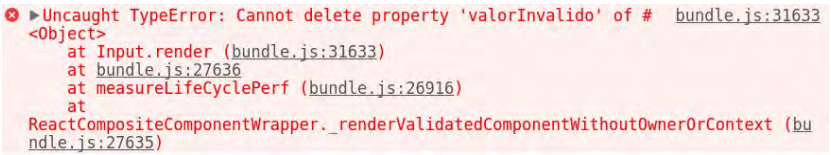

Figura 5.6: Erro ao excluir valorInvalido

Isso acontece porque o React define o objeto `props` como imutável. Não podemos excluir ou alterar seus atributos. Vamos corrigir esse problema na próxima versão do componente, tratada adiante.

Terceira versão do Input

Para resolver esse impasse, vamos copiar o objeto `props` para uma variável local e, em seguida, alterá-la, excluindo a propriedade `valorInvalido`. A cópia de `props` será realizada por meio de um recurso chamado `Object.assign()`. O método `assign()` é usado para copiar todos os atributos de um ou mais objetos para outro objeto de destino, obtido através do retorno do método.

Em outras palavras, a função `Input` definirá a constante `estilo`, copiará `this.props` para uma variável chamada `propriedades`, apagará o atributo

`propriedades.valorInvalido` e repassará o objeto ajustado integralmente à tag `<input>`. Veja a seguir o código completo do componente `Input`.

```
import React from 'react'

export default function Input(props) {
    const estilo = {
        borderColor: props.valorInvalido ? '#d50000' : '#cccccc',
        backgroundColor: props.valorInvalido ? '#ffcdd2' : '#ffff
ff'
    };

    let propriedades = Object.assign({},props);
    delete propriedades.valorInvalido;

    return (
        <input
            type="text"
            style={estilo}
            {...propriedades}
        />
    )
}
```

Essa mesma lógica poderá ser aplicada para quaisquer propriedades que comprometam a passagem indireta através do *spread operator*. É claro que, se o número de propriedades excluídas for superior ao número de propriedades repassadas, descartaríamos o *spread operator* em prol da declaração explícita.

Ajustando o componente NovoUsuario

Concluímos a criação do componente `Input`. Vamos agora ajustar o código do componente `NovoUsuario` para referenciá-lo. Abra o arquivo `index.jsx` dentro do diretório `behappywith.me/front-end/src/components/NovoUsuario`. No início do arquivo, faça a devida importação.

```
import React from 'react'
import Label from '../Label'
import Input from '../Input'
// ...
```

Em seguida, ajuste o método `render()` para convocar o componente `Input`.

```
render() {
    return (
        <div className="center">
            <form className="pure-form pure-form-stacked">
                <Label htmlFor="nome" texto="Quem é você?" />
                <Input
                    id="nome"
                    placeholder="Digite seu nome"
                    maxLength="40"
                    readOnly
                    valorInvalido
                    defaultValue="Guilherme"
                />
            </form>
        </div>
    );
}
```

Esse código fará com que a tag `<input>` seja apenas de leitura, e que esteja marcada de vermelho (`valorInvalido`) e com o valor `Guilherme` preenchido por default. É importante destacar que, assim como o atributo `readonly` original da W3C, precisamos apenas declarar a sintaxe `readOnly` na tag. O mesmo ocorre para o atributo `valorInvalido`.

Em ambos os casos, porém, se passarmos uma string `"false"` estaticamente para as propriedades (`readOnly="false"` `valorInvalido="false"`), a tag `<input>` vai continuar se mantendo como somente leitura e com a borda avermelhada. Isso acontece porque o React considera a string `"false"` como um valor **não** `undefined`, por assim dizer. Isso significa que o

simples fato de colocar esse tipo de propriedade *booleana* fará com que o campo assuma a característica.

As únicas maneiras de desativá-las são:

1. Obviamente, não a declarar;
2. Atribuir `false`, `null` ou `undefined` como um valor JavaScript (entre chaves).

Vamos optar pela segunda, já que isso nos dá a flexibilidade de criarmos variáveis booleanas no estado do componente e atribuí-las às propriedades. Faremos isso agora.

Começaremos com um estado para validar o preenchimento do nome pela propriedade `valorInvalido`. Se o nome estiver correto, `valorInvalido` será verdadeiro; do contrário, falso. Note que isso se assemelha muito ao `valorInvalido` do componente `Label`, e que ambos precisam caminhar juntos em termos da validação, o que nos leva a tratá-los de forma conjunta.

Tudo fica mais claro visualizando o código. Assim como fizemos no início do capítulo, vamos inicializar o estado declarando o valor de `this.state` no `constructor` do objeto. Abra o arquivo `index.jsx` do componente `NovoUsuario` e altere-o. Por enquanto, vamos mantê-lo como `true`.

```
class NovoUsuario extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            nomeInvalido: true
        };
    }
}
```

O próximo passo é repassar esse estado aos componentes

`Label` e `Input`. Isso pode ser realizado buscando seu valor pela sintaxe `this.state.nomeInvalido`. Vamos ajustar o método `render` para comportar essa alteração.

A propriedade `readOnly` possui a mesma semântica, mas, como ela será necessária apenas no futuro, vamos apenas mantê-la estaticamente definida como `false` por enquanto. Veja a seguir o código do método `render()`.

```
class NovoUsuario extends React.Component {
    // constructor ...
    render() {
        return (
            <div className="center">
                <form className="pure-form pure-form-stacked">
                    <Label
                        htmlFor="nome"
                        texto="Quem é você?"
                        valorInvalido={this.state.nomeInvalido}
                    />
                    <Input
                        id="nome"
                        placeholder="Digite seu nome"
                        maxLength="40"
                        readOnly={false}
                        valorInvalido={this.state.nomeInvalido}
                        defaultValue="Guilherme"
                    />
                </form>
            </div>
        );
    }
}
```

Com esse código, já podemos visualizar a aplicação. Execute o servidor de desenvolvimento, acesse o endereço http://localhost:8080/, e veja um resultado semelhante à figura a seguir.

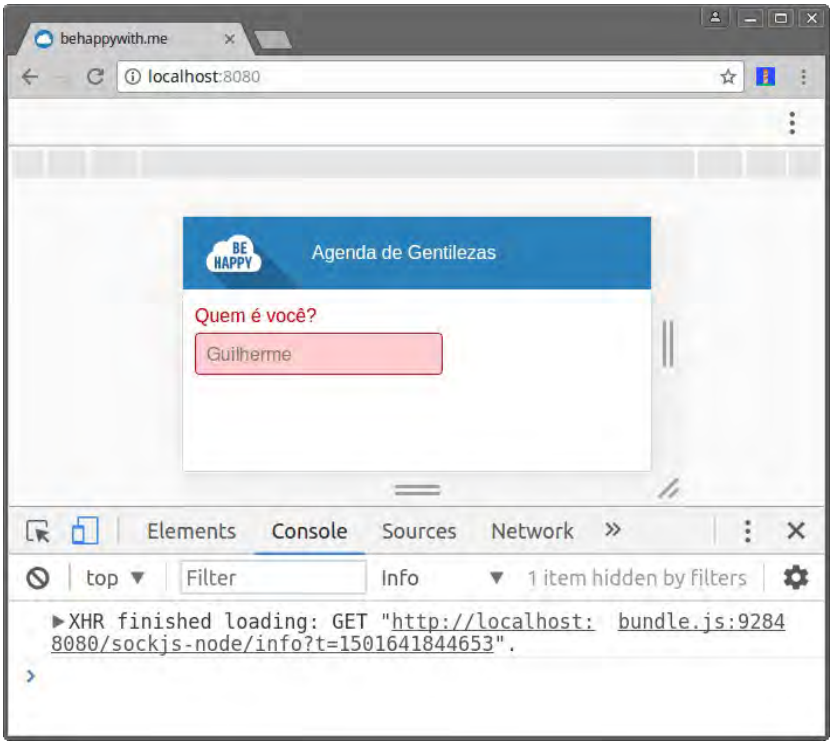

Figura 5.7: Exibição dos componentes Label e Input

Continuando, como `NovoUsuario` deve armazenar o valor digitado no componente `Input` em seu estado, vamos precisar reorganizar `this.state` do `constructor` em duas estruturas principais: `usuario` e `validacao`. Dentro delas, teremos os atributos `nome` e `nomeInvalido`, respectivamente. Note que alteramos `nomeInvalido` para `false`.

```
this.state = {
    usuario: {
        nome: ''
    },
    validacao: {
        nomeInvalido: false
```

```
    }
}
```

Agora, vamos implementar um método para receber as alterações realizadas na tag `<input>` dentro do componente `Input`. Como sabemos, `Input` repassará todas as propriedades para sua tag interna, inclusive *funções*. Isso significa que, se criarmos um método `atualizarNome()` e o repassarmos através de uma propriedade chamada `onChange` (cuja tradução poderia ser *em mudança*), a tag `<input>` vai registrá-lo e, sempre que houver mudança em seu valor, este método será executado.

Mas como vamos obter o valor de `<input>`? Se temos um evento que reage à mudança do elemento (`onChange`) e que, por sua vez, invoca uma função para tratá-lo (`atualizarNome()`, neste caso), podemos recuperar o *dado alterado* do próprio evento.

Na prática, o que faremos será obter o novo valor que *provocou* o evento `onChange`. Isso pode ser realizado por meio da sintaxe `e.target.value` (supondo que `e` seja o evento).

De volta ao código de `NovoUsuario/index.jsx`, crie um método `atualizarNome()` para pegar o valor de `e.target.value` e salvar em `this.setState()`. Veja o código.

```
atualizarNome(e) {
    let usuario = this.state.usuario;
    usuario.nome = e.target.value;
    this.setState({
        usuario: usuario
    });
}
```

Há um ponto muito importante aqui! O método `this.setState()`, como corretamente exposto na documentação oficial do React, faz um *merge* ajustando apenas os

itens passados, sem afetar os demais. Isso significa qu,e se tivermos um objeto com dois atributos e repassarmos apenas um deles pelo `this.setState()`, o outro ficará intacto.

Então, por que o método `atualizarNome()` não pode usar a sintaxe `this.state.usuario.nome` diretamente? Porque o *merge* **não** percorre toda a hierarquia dos atributos internos. Em outras palavras, ele faz o *merge* dos atributos da raiz do estado.

Se atualizarmos `usuario` conforme o código adiante, quando tivermos outros atributos (como o gênero e o avatar), eles serão perdidos no momento da atualização do `nome`. Ou seja, com o código exibido adiante, teríamos um erro na lógica de definição do `nome`.

```
atualizarNome(e) {
    // Este código possui um problema
    // Ele sobrescreve os demais atributos de usuario
    this.setState({
        usuario: {
            nome: e.target.value
        }
    });
}
```

Então, lembre-se: quando o estado comportar objetos internos, vamos precisar resgatá-los, atualizá-los e finalmente salvá-los com `this.setState()`. E o que acontece com o objeto `validacao`? Como ele está na raiz, a atualização do objeto `usuario` não vai comprometê-lo.

De volta ao nosso código, o último passo é ajustar o método `render()`. Acrescente a propriedade `onChange` apontando para o método `atualizarNome()`. Aqui utilizamos o `bind()`. Se quiser, altere para uma *arrow function*.

Aproveite também para alterar a sintaxe explícita da string `Guilherme` pelo valor de `this.state.usuario.nome` em `defaultValue`, e ajustar a origem do booleano `valorInvalido` de `Label` e `Input` – que agora deve ser obtido de `this.state.validacao.nomeInvalido`. Veja o panorama completo da chamada dos componentes `Label` e `Input` no trecho adiante.

```
render() {
    return (
        <div className="center">
            <form className="pure-form pure-form-stacked">
                <Label
                    htmlFor="nome"
                    texto="Quem é você?"
                    valorInvalido={this.state.validacao.nomeInval
ido}
                />
                <Input
                    id="nome"
                    placeholder="Digite seu nome"
                    maxLength="40"
                    readOnly={false}
                    valorInvalido={this.state.validacao.nomeInval
ido}
                    defaultValue={this.state.usuario.nome}
                    onChange={this.atualizarNome.bind(this)}
                />
            </form>
        </div>
    );
}
```

Salve tudo e execute o servidor de desenvolvimento. Ao acessar http://localhost:8080/, você verá o resultado esperado.

5.3 CRIANDO O COMPONENTE DE SELEÇÃO DE GÊNERO

Vamos conversar sobre a lógica deste componente. Veja sua organização:

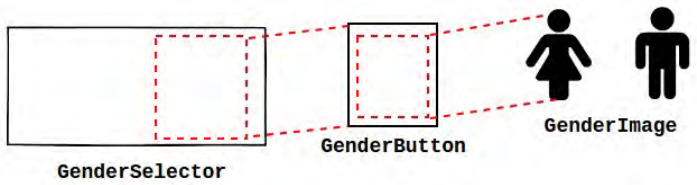

Figura 5.8: Organização dos componentes

O `GenderSelector` será o responsável por organizar as opções de seleção do gênero (masculino ou feminino) lado a lado. Como as duas opções não podem ser mutuamente selecionadas, o `GenderSelector` conterá uma lógica para sinalizar ao `GenderButton` se ele está ou não selecionado. Para o caso em que ele não estiver, haverá também uma borda na cor vermelha quando a propriedade `valorInvalido` for `true`.

O componente `GenderButton` servirá como ponto de clique (ou toque) no qual o usuário selecionará seu gênero. Na prática, ele será um link `<a>` que, quando clicado, ficará com uma cor destacando sua seleção.

De forma semelhante ao componente `Input` exposto no tópico anterior, o evento `onClick` repassado à tag `<a>` terá origem no componente `NovoUsuario`. Isso significa que o resultado do evento salvará o valor `'m'` ou `'f'` no `this.state` do `NovoUsuario`.

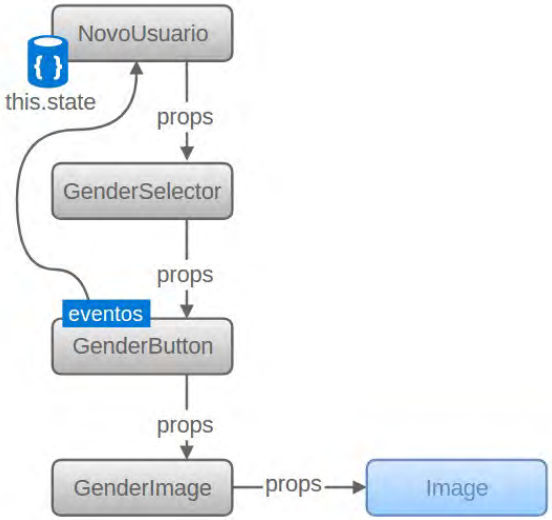

Figura 5.9: Manipulação do componente GenderSelector

Falta-nos discorrer sobre como trabalharemos com as imagens do gênero. O mais comum é utilizar a tag `<img>` . Isso se aplica perfeitamente bem quando temos poucas imagens a serem apresentadas. Na nossa aplicação, entretanto, além das 2 imagens do gênero, teremos 46 imagens de avatar, 6 imagens dos destinatários das gentilezas, 7 tipos de gentileza e outras minúcias relacionadas ao logo e aos botões.

Seria um pecado mortal se essas imagens fossem grandes (tamanho do arquivo). Por nossa sorte, elas são bem pequenas e comprimidas através do site CompressPNG (http://compresspng.com/). Considerando uma média de 10kb por arquivo, teremos algo em torno de 600kb só de imagens.

É um tamanho razoável, mas eventualmente usuários com conexão mais fraca serão prejudicados. E este é um problema que

não podemos abafar. O requisito que trata do carregamento rápido da página em conexões 3G existe para mantermos o compromisso com esses usuários. O valor mínimo aceito é de 10 segundos de carregamento, e vamos reduzi-lo para 3 (*three seconds and go*).

Figura 5.10: História 1.6

A taxa de download de uma conexão de banda larga 3G típica varia entre 400 Kbps a 4 Mbps. No Brasil, o 3G fica na média de 1 Mbps, sendo que as prestadoras de serviço podem fornecer no mínimo 10% deste valor. Isso significa que, no caso mais crítico, um usuário que estiver conectado via 3G terá uma velocidade de conexão de 100 Kbps.

Com uma conta rápida no site http://downloadtimecalculator.com/, podemos concluir que este usuário superlimitado levará 49 segundos só para baixar as imagens de nossa PWA. Isso mesmo: uma eternidade de 49 segundos!

Mesmo sabendo que os cálculos foram realizados com um cenário extremo e sem considerar o uso da compressão `gzip` pelo Nginx (que veremos no futuro), não podemos simplesmente ignorar o problema. Na verdade, aplicações construídas sob a teoria de *progressive enhancement* **não devem** ignorá-lo.

Há uma feitiçaria de webdesigners conhecida como *Image*

Sprites que vai nos ajudar. Não há uma tradução adequada para o termo *Sprites* quando utilizado neste contexto, mas, na prática, ele significa colocar todas as imagens lado a lado em um único arquivo. Essa técnica surgiu no século passado quando os desenvolvedores de jogos otimizavam a renderização dos personagens na tela do computador colocando todos os movimentos (imagens) em um único arquivo.

Figura 5.11: Sprites de um jogo

A ideia de criar imagens com Sprites na web também está relacionada à performance, mas não exatamente ao tamanho dos arquivos, e sim à quantidade de vezes que o browser vai precisar ir ao servidor para baixar os arquivos.

Quanto mais arquivos tivermos, mais tempo o browser levará para realizar todas as solicitações ao servidor HTTP. Como cada solicitação consome banda da conexão com a internet, se organizarmos nossas imagens em um único arquivo (em Sprites), vamos reduzir a quantidade de solicitações e, consequentemente, reduzir o consumo da largura de banda.

Pode parecer ridículo e demasiadamente trabalhoso, mas há ganho. Se não existisse, grandes players como Twitter, Amazon etc. não usariam esse recurso.

Se quisermos aprofundar mais a discussão, é possível ainda citar que imagens agrupadas provocam menor impacto no tráfego

por terem maior probabilidade de compactação. A lógica de compactação nos leva a crer que, quando colocamos imagens semelhantes em um único arquivo, o resultado será mais efetivo.

No nosso caso, por exemplo, se compactarmos as 2 imagens de gênero soltas, teremos algo em torno de 6.7 kbytes. Se essas imagens fossem colocadas em único arquivo, após a compactação, elas terão 5 kbytes – uma diferença de 25%. Se acrescentarmos esse comportamento ao overhead padrão do protocolo HTTP (versão 4), que é de 2% em relação ao tamanho dos arquivos, cada kbyte economizado no agrupamento das imagens trará, sim, um ganho razoável.

Depois deste testemunho nerd, é fácil de entender o motivo pelo qual criaremos um componente específico para encapsular a lógica de exibição das imagens.

Lógica do componente GenderImage

No caso do nosso componente de gênero, poderíamos ter apenas duas imagens, uma ao lado da outra.

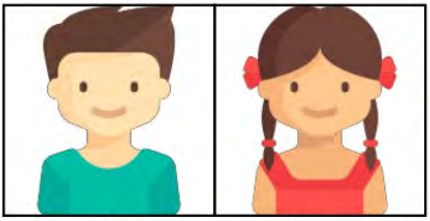

Figura 5.12: Imagens do gênero

Essas imagens, entretanto, também compõem o *pull* de

imagens do avatar do personagem. Com isso, seria mais razoável condensá-las em um único arquivo a partir do qual criaremos os componentes de seleção de gênero e seleção de avatar. Como também existirá uma lógica que exibirá avatares distintos para cada gênero, podemos organizar os Sprites em duas linhas: avatares masculinos e avatares femininos.

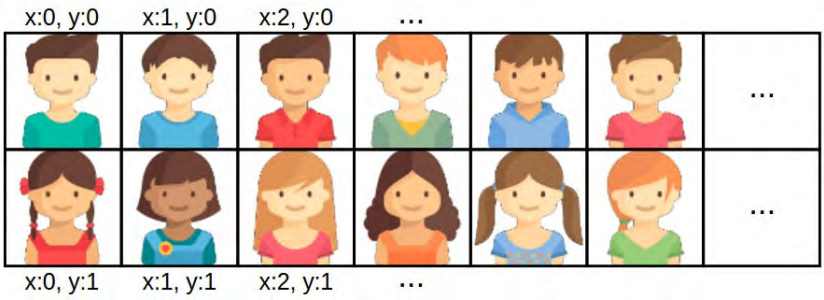

Figura 5.13: Imagens dos avatares

Nossa imagem `avatars.png` terá essa configuração, com Sprites de 170 por 170 pixels. Como observação, apesar de o tamanho físico dos Sprites serem de 170 pixels, o componente vai exibi-los com comprimento e altura iguais a 140 pixels. Essa diferença de tamanho foi necessária para melhor aproveitar as mesmas imagens em todas as telas do sistema.

Por estarem organizadas em uma matriz, podemos imaginá-las dispostas em eixos X e Y. E para que isso funcione, em vez de tags `<img>`, precisaremos de tags `<div>` estilizadas com a propriedade `background` posicionada especificamente para cada Sprite.

A propriedade `background` de uma classe de estilo CSS possui algumas variações. Uma delas (`background-image`) é

usada para especificar o nome do arquivo de imagem que será atribuída ao plano de fundo de um elemento. Veja no código a seguir um estilo que, quando aplicado à `<div class="imagem"> </div>`, exibirá um quadrado de 140 pixels.

```
div.imagem {
    background-image: url(avatars.png);
    background-size: auto 280px;
    width: 140px;
    height: 140px;
    display: block;
}
```

Como sabemos, a imagem `avatars.png` possui uma disposição de Sprites com 170 pixels, e nosso sistema deve exibi-las com 140 pixels. A propriedade `background-size` serve justamente para *forçar* uma altura de 280 pixels, que equivale a duas imagens de 140 pixels – uma sobre a outra.

A propriedade `background-size` recebe dois argumentos: o comprimento e a altura do plano de fundo. O valor default para ambos é `auto`, que, como o próprio nome já diz, renderizará automaticamente o tamanho da `background-image` de acordo com o tamanho real da imagem.

Nossa imagem de gêneros/avatares tem 3910 pixels horizontais (são 23 avatares por gênero) e 340 pixels verticais (2 gêneros). Esses serão os valores representados pela sintaxe `auto` indicada no exemplo a seguir.

```
background-size: auto auto;
```

Se trocarmos explicitamente a referência vertical de `background-size` para `170px` (o segundo valor `auto`), a altura da imagem será reduzida pela metade (de `340px` para `170px`) e

o CSS vai automaticamente redimensionar o comprimento, implicitamente alterando `3910px` para `1955px`.

```
background-size: auto 170px;
```

Isso reduzirá o tamanho da imagem no browser, porém, como não ajustamos o tamanho da `<div>`, serão apresentados 4 Sprites em vez de apenas 1.

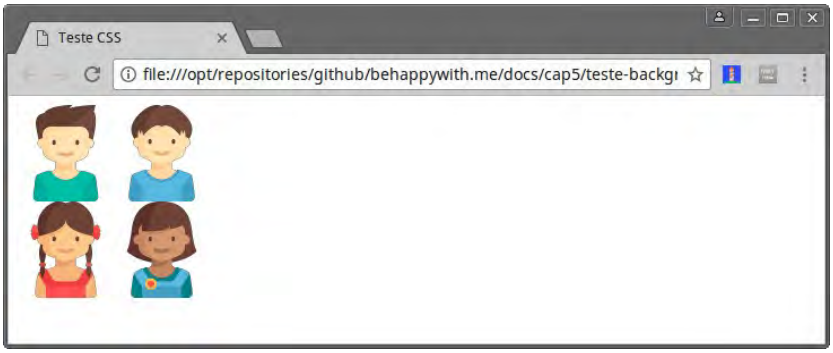

Figura 5.14: Exibição errada do redimensionamento da imagem

Para ajustar esse impasse, **sempre** precisaremos acompanhar o redimensionamento do plano de fundo com o comprimento e altura da `<div>`, respectivamente, ajustando as propriedades `width` e `height`. Como reduzimos o tamanho da imagem pela metade, o correto será ajustar o tamanho da `<div>` na mesma proporção (de `170px` para `85px`).

```
div.imagem {
    background-image: url(avatars.png);
    background-position-y: 0;
    background-position-x: 0;
    background-size: auto 170px;
    width: 85px;
    height: 85px;
    display: block;
```

```
}
```

Com esse ajuste, somente 1 Sprite redimensionado será exibido no navegador. Essa mesma lógica será aplicada ao caso da redução das imagens de 170 para 140 pixels: `background-size` para redimensionar e `width` e `height` para exibir a `<div>` com o tamanho correto.

A teoria sobre redimensionamento da *Image Sprites* está concluída. Vamos conversar agora sobre o algoritmo de exibição de apenas um Sprite de forma indexada. Veja o código a seguir.

```
div.imagem {
    background-image: url(avatars.png);
    background-size: auto 280px;
    width: 140px;
    height: 140px;
    display: block;
}
```

Se aplicarmos essa classe CSS a uma tag `<div>`, automaticamente estaremos apresentando a imagem cujos eixos imaginários X e Y são iguais a zero.

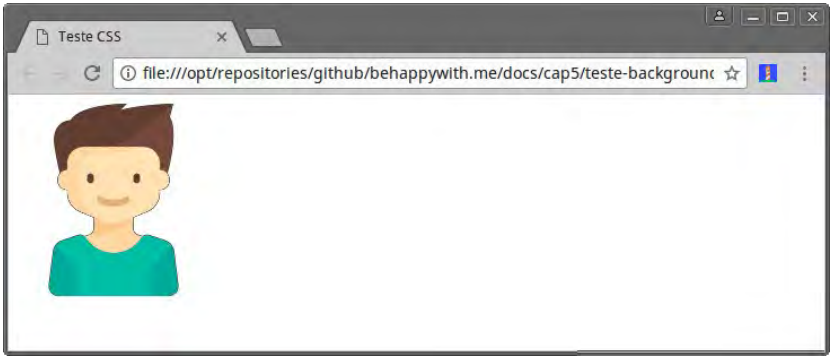

Figura 5.15: Exibição da com estilo

Isso acontece porque `background-image`, por default, renderiza a imagem a partir dos primeiros pixels (pixel de índice 0), de forma horizontal e vertical no canto superior da `<div>`. E como seu tamanho está sendo ajustado com `background-size`, apenas o primeiro Sprite será exibido.

Se, por outro lado, informarmos à `<div>` que a imagem deve ser exibida a partir do pixel 140 verticalmente, ela esconderá a imagem do menino e mostrará a imagem da menina, posicionada no suposto eixo X=0 e Y=1.

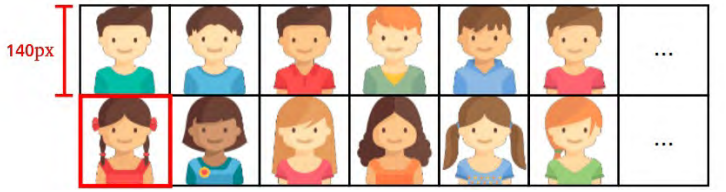

Figura 5.16: Exibição a partir do pixel 140 (vertical)

É como se deslocássemos a imagem 140 pixels para cima e encaixássemos o segundo Sprite que começa exatamente a partir do pixel vertical de número 140 e pixel horizontal de número 0 (zero). A ideia é essa, mas como vamos deslocar o `background`? Através da sintaxe `background-position-y`, conforme o código a seguir.

```
div.imagem {
    background-image: url(avatars.png);
    background-size: auto 280px;
    background-position-y: -140px;
    width: 140px;
    height: 140px;
    display: block;
}
```

Essa pequena alteração provocará o seguinte efeito. Note o uso do sinal `-` (menos) antes do `140px`.

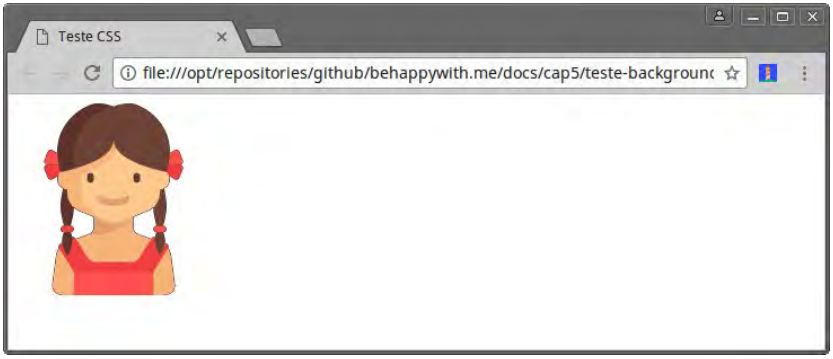

Figura 5.17: Exibição da `<div>` com estilo

A mesma lógica pode ser aplicada ao atributo `background-position-x` para deslocarmos a imagem `avatars.png` horizontalmente para a esquerda, e assim exibi-la a partir de um certo pixel.

A magia da *Image Sprites* pode então ser resumida em criar uma `<div>` com um estilo que informe uma imagem através da propriedade `background-image`, faça um redimensionamento com `background-size`, e desloque-a para a esquerda e para cima ajustando a posição inicial por meio das propriedades `background-position-x` e `background-position-y`, respectivamente.

Se estivéssemos fadados ao uso de uma estilização estática, reproduzir as duas imagens do gênero implicariam em criar dois estilos com posicionamento distinto do plano de fundo. O pior seria no caso do componente de escolha do avatar: teríamos de criar 46 classes de estilo. A manutenibilidade seria péssima.

Sorte nossa que faremos a estilização dinâmica dentro de um componente React. Esse componente de imagem receberá duas propriedades, chamadas `eixoX` e `eixoY`, que, quando multiplicadas pelo tamanho do Sprite, serão capazes de exibir corretamente a imagem desejada. Este é o verdadeiro poder da estilização inline via código.

Você já deve estar imaginando como ficará o código do componente `GenderImage`. Antes de começar, porém, deixe-me colocar uma breve questão: seria adequado reproduzir a mesma lógica de posicionamento do `background` nos componentes `GenderImage` e `AvatarImage`? Não.

Então, para evitar essa duplicidade, criaremos um componente `Image` para ser o verdadeiro mantenedor dessa inteligência. Os componentes `GenderImage` e `AvatarImage` serão películas, por assim dizer, que passarão os parâmetros corretos de exibição do Sprite ao componente `Image`.

Ainda no viés de reutilização dos componentes, outras telas do sistema vão exibir essas mesmas imagens em tamanhod diferenciadod. A mesma técnica obtida com `background-size` nos exemplos anteriores será aplicada ao componente `Image` para redimensioná-las.

Com as lógicas de exibição e redimensionamento devidamente explicadas, vamos ao código real do componente.

Criando o componente Image

Crie uma pasta chamada `Image` dentro do diretório `behappywith.me/front-end/src/components`. Dentro de

`Image`, crie um arquivo chamado `index.jsx`. Apesar de este componente ser *stateless* e as boas práticas do *Airbnb React/JSX Style Guide* orientarem no uso da sintaxe `function`, neste caso, vamos criá-lo a partir de uma `class`.

Isso se faz necessário porque codificaremos métodos particulares para cada trecho da lógica de renderização das imagens. Poderíamos simplesmente colocá-la integralmente dentro da `function`, mas isso criaria um empecilho sob o ponto de vista de manutenção.

Abra o arquivo `index.jsx` e crie a estrutura básica da classe `Image`, conforme o código:

```
import React from 'react'

 class Image extends React.Component {
    constructor(props) {
        super(props);
    }
}

export default Image;
```

O próximo passo será criar as funções para calcular o deslocamento nos eixos X e Y. Como conversamos, eles serão obtidos a partir do produto entre as propriedades `eixoX` e `eixoY` com o tamanho do Sprite.

É claro que, para reaproveitar esse componente em outras situações, seu tamanho não poderá ser fixo em 140 pixels. Ele também receberá outras duas propriedades, `width` e `height`, que, além do cálculo do deslocamento, serão utilizadas na classe inline do estilo. Assim, crie os métodos `calcularPosicaoX()` e `calcularPosicaoY()` conforme o trecho adiante.

```
class Image extends React.Component {
    // constructor ...

    calcularPosicaoX() {
        return (this.props.eixoX * this.props.width * (-1)) + 'px
;
    }
    calcularPosicaoY() {
        return (this.props.eixoY * this.props.height * (-1)) + 'p
x';
    }
}
```

Veja que estamos multiplicando o produto dos eixos e dimensões (comprimento e altura) por `(-1)`. Isso fará com que a imagem desloque-se para a esquerda ou para cima, dependendo do método.

O texto `'px'` é necessário porque o retorno dos métodos será atribuído diretamente às propriedades do estilo. Nesse código, inclusive, estamos concatenando os trechos com a sintaxe `+`. Isso pode ser facilmente substituído pelos *Template Strings* do ECMAScript 6, muito mais elegante e manutenível.

O *Template Strings* é um recurso bem simples que permite integrar variáveis e expressões JavaScript ao mesmo trecho de código em que o texto é definido. Vejamos como ficaria o método `calcularPosicaoX()` se substituirmos a concatenação pelo *Template Strings*.

```
calcularPosicaoX() {
    return `${this.props.eixoX * this.props.width * (-1)}px`
}
```

A diferença mais importante é que, em vez do uso da sintaxe com `'` (aspas simples) ou `"` (aspas duplas), o *Template Strings* trabalha com a `` ` `` (crase). Para exibir as expressões, devemos

circundá-las com `{}` (chaves) antecedidas de um `$` (cifrão).

Isso já é suficiente para o uso da nossa aplicação, mas outro excelente recurso do *Template Strings* é permitir a quebra de linhas sem o uso da sintaxe `\n`, deixando-as diretamente expostas como se estivéssemos escrevendo um texto livre. O código adiante, por exemplo, é permitido.

```
console.log(`Este é
um texto
com quebra
de linhas`);
```

Pronto, finalizamos com o *Template Strings*. Então, se refatorarmos os métodos para calcular a posição, teremos o seguinte código:

```
calcularPosicaoX() {
    return `${this.props.eixoX * this.props.width * (-1)}px`
}
calcularPosicaoY() {
    return `${this.props.eixoY * this.props.height * (-1)}px`
}
```

Vamos agora criar um método para redimensionar o tamanho da `background-image`. Como vimos na seção anterior, isso será feito pela propriedade `background-size`. Crie um método chamado `calcularTamanho()` que retorne `auto` para o comprimento e um valor baseado na propriedade `backgroundHeight` para a altura. Veja o código adiante.

```
class Image extends React.Component {
    // constructor ...
    // calcularPosicaoX ...
    // calcularPosicaoY ...

    calcularTamanho() {
        return `auto ${this.props.backgroundHeight}px`
```

```
    }
}
```

De posse destes cálculos, vamos criar outro método que retornará o objeto de estilo inline que será aplicado à `<div>`. Além do que já discutimos, vamos aplicar também as propriedades `display` e `margin` para centralizar a `<div>` no elemento superior, seja este qual for. Vejamos o código. Os demais métodos e a declaração da classe `Image` foram omitidos.

```
obterEstilo() {
    return {
        backgroundImage: `url(${this.props.arquivo})`,
        backgroundPositionX: this.calcularPosicaoX(),
        backgroundPositionY: this.calcularPosicaoY(),
        backgroundSize: this.calcularTamanho(),
        width: `${this.props.width}px`,
        height: `${this.props.height}px`,
        display: 'table',
        margin: '0 auto'
    }
}
```

Lembre-se de que a estilização inline do React transforma as propriedades separadas por `-` (sinal de menos) na sintaxe *camelCase*. Note também que a passagem do nome do arquivo por meio da `this.props.arquivo` se faz necessária porque obviamente esse componente deve ser capaz de exibir qualquer imagem (além da `avatares.png`).

Como organizamos toda a complexidade em métodos particulares, o método `render()` em si será muito simples. Ele apenas retornará uma `<div>` com o estilo inline retornado pelo método `obterEstilo()`.

```
class Image extends React.Component {
    // constructor ...
    // calcularPosicaoX ...
```

```
    // calcularPosicaoY ...
    // obterEstilo ...

    render() {
        return (
            <div style={this.obterEstilo()}>
            </div>
        )
    }
}
```

Com isso, concluímos o `Image`. Este é o componente mais complexo deste capítulo. Veja o seu código na íntegra:

```
import React from 'react'

class Image extends React.Component {
    constructor(props) {
        super(props);
    }

    calcularPosicaoX() {
        return `${this.props.eixoX * this.props.width * (-1)}px`
    }
    calcularPosicaoY() {
        return `${this.props.eixoY * this.props.height * (-1)}px`
    }
    calcularTamanho() {
        return `auto ${this.props.backgroundHeight}px`
    }

    obterEstilo() {
        return {
            backgroundImage: `url(${this.props.arquivo})`,
            backgroundPositionX: this.calcularPosicaoX(),
            backgroundPositionY: this.calcularPosicaoY(),
            backgroundSize: this.calcularTamanho(),
            width: `${this.props.width}px`,
            height: `${this.props.height}px`,
            display: 'table',
            margin: '0 auto'
        }
    }
```

```
    render() {
        return (
            <div style={this.obterEstilo()}>
            </div>
        )
    }
}

export default Image;
```

Vamos agora ao `GenderImage`.

Criando o componente GenderImage

Como `Image` necessita de muitos parâmetros, a ideia é encapsular essa complexidade em componentes de imagem particulares para cada caso. Nesta seção, vamos criar o `GenderImage`, responsável pela exibição das duas figuras associadas aos gêneros.

Crie uma pasta `components/GenderImage` e, dentro dela, um arquivo `index.jsx`. É importante lembrar que o nosso arquivo `avatars.png` terá o Sprite do gênero masculino no suposto eixo `X:0 e Y:0`, e do feminino em `X:0 e Y:1`.

O único componente que precisa conhecer essa regra é o `GenderImage`. O componente `NovoUsuario` apenas sabe que o usuário terá o atributo do gênero definido como `'m'` quando for masculino, e `'f'`, quando feminino.

Para transformar o domínio de `'m'` e `'f'` na propriedade `eixoY`, criaremos uma condição ternária que retornará `0` quando o valor da propriedade `genero` for `'m'`; caso contrário, `1`. Isso significa criar um trecho de código semelhante ao apresentado a seguir.

```
eixoY={(props.genero === 'm') ? 0 : 1}
```

Além disso, as propriedades `width` e `height` serão definidas com 140, e `backgroundHeight` com 280. Estes valores dizem respeito ao tamanho de exibição desejável da imagem `avatars.png`.

Como `GenderImage` é *stateless*, vamos manter o padrão de especificação por `function`. Veja o código completo adiante.

```
import React from 'react'
import Image from '../Image'

export default function GenderImage(props) {
    return (
        <Image
            eixoX={0}
            eixoY={(props.genero === 'm') ? 0 : 1}
            width={170}
            height={170}
            backgroundHeight={340}
            arquivo="img/avatars.png"
        />
    )
}
```

Guardando o arquivo avatars.png corretamente

Já terminamos toda a parte de código dos componentes `Image` e `GenderImage`. Falta-nos apenas baixar e discutir onde salvaremos a imagem `avatars.png`.

Quando se trabalha com projetos componentizados, como este, o mais natural seria criar uma pasta `img` dentro do componente `GenderImage` e salvar `avatars.png` nela. Se fizéssemos isso, entretanto, o futuro componente `AvatarImage` teria de replicar o arquivo físico e a importação de `avatars.png`. Se essa duplicação

não for feita, o componente `AvatarImage` ficaria dependente de `GenderImage`, o que não cheira muito bem.

Para não precisar replicar arquivos e importações, vamos centralizar as imagens no próprio componente `Image`. Apesar de ele ser teoricamente imparcial a esses detalhes técnicos, usá-lo como ponto único de importação e armazenamento dos arquivos será mais adequado para o nosso caso.

Abra o arquivo `behappywith.me/front-end/src/components/Image/index.jsx` e inclua a importação à imagem `avatars.png`, conforme o código adiante.

```
import React from 'react'
import './img/avatars.png'

class Image extends React.Component {
    // ...
}
```

Por fim, crie um novo diretório `img` em `behappywith.me/front-end/src/components/Image/`, baixe a imagem `avatars.png` em https://behappywith.me/img/avatars.png e salve-a nele.

Apenas para visualizar esses novos componentes em ação, abra o arquivo `index.jsx` de `NovoUsuario`, importe o componente `GenderImage` e acrescente sua chamada no método `render()`. Note que ele espera receber uma propriedade `genero` com o valor `'m'` ou `'f'`.

```
// outros imports ...
import GenderImage from '../GenderImage'

class NovoUsuario extends React.Component {
    // constructor e métodos ...
```

```
    render() {
        return (
            <div className="center">
                <form className="pure-form pure-form-stacked">

                    {/* Componentes Label e Input */}

                    <GenderImage
                        genero="f"
                    />
                </form>
            </div>
        );
    }
}
```

Só como nota, veja que comentários no JSX exigem o uso de `{}` (chaves) ao redor de `/*` e `*/`, que é sintaxe padrão JavaScript. Se você executar o servidor de desenvolvimento e acessar http://localhost:8080/, verá algo semelhante à imagem a seguir.

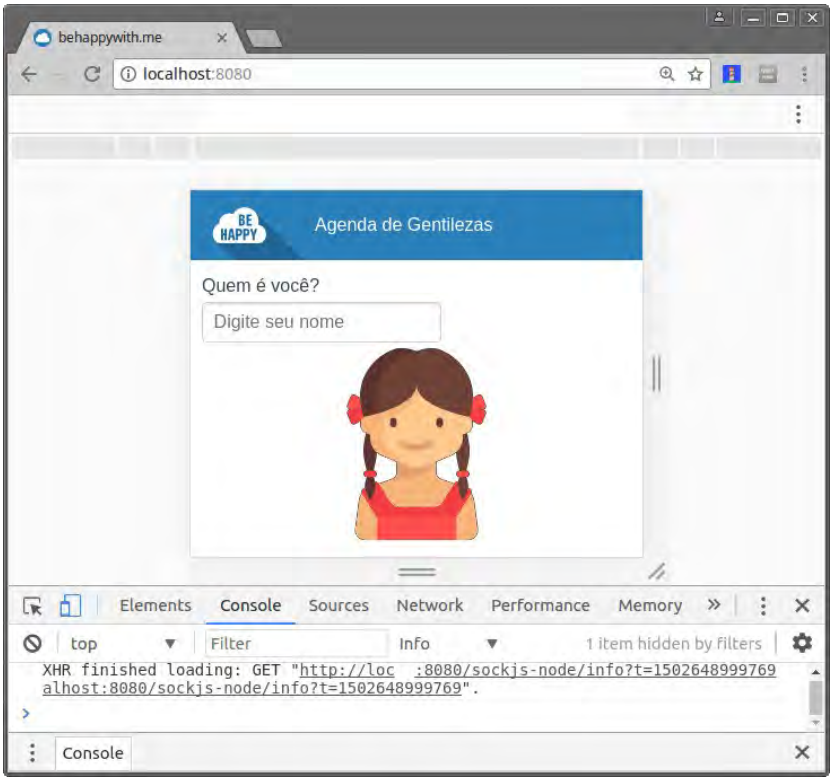

Figura 5.18: Teste dos componentes Image e GenderImage

Se você alterou o seu código para visualizá-los, **apague** esses dois ajustes (importação e chamada ao componente `GenderImage`) em `NovoUsuario`. Na prática, isso ficará encapsulado no componente `GenderButton`, como veremos na próxima seção.

Criando o componente GenderButton

A ideia principal deste componente é deixar evidente se o usuário selecionou ou não uma das opções do gênero com um

clique. Por questões semânticas e de praticidade, o mais apropriado neste caso é representá-lo por meio de um hyperlink `<a>`. Ele também repassará o gênero para `GenderImage` e será o responsável por capturar o evento `onClick`.

Vamos começar pela estilização. O objeto JavaScript de estilização inline **não** nos oferece nativamente a opção `hover`, muito comum nas classes CSS de hyperlink. Apesar de pouco útil para dispositivos mobile, o efeito `hover` é ótimo para o *affordance* (indica se o componente é clicável) das tags `<a>` nos dispositivos desktop.

Se quiséssemos criar o efeito `hover` via estilo inline do React, seria necessário tratar os eventos `onMouseEnter` e `onMouseLeave`, realizados sobre a tag `<a>` para alterar um estado interno do componente exibindo-o de forma diferenciada. O componente seria muito mais complexo!

Em vez disso, podemos simplesmente utilizar classes CSS com a sintaxe `hover` nativa e indicá-las na renderização do componente. Vamos implementar esta última opção.

Crie um diretório chamado `GenderButton` dentro de `behappywith.me/front-end/src/components/`. Dentro dele, crie um arquivo chamado `index.css`.

Primeiramente, vamos criar a estilização padrão pela classe `a.gender-button`. Nela teremos nosso conhecido `box-sizing`, que integra os pixels usados na `padding` e `border` juntamente com as propriedades `width` e `height`. Especificaremos uma borda com curvatura nas pontas com `border-radius`, uma `margin` padrão de 3 pixels e um `padding-top` com `5px`, que

distancia o elemento interno em 5 pixels a partir do topo.

```
a.gender-button {
    box-sizing: border-box;
    width: 180px;
    height: 180px;
    border-radius: 10px;
    margin: 3px;
    padding-top: 5px;
    display: inline-block;
}
```

A propriedade `display` é um recurso que ajuda na forma de exibição dos elementos. Neste caso, por meio do valor `inline-block`, eles serão organizados em linha, um ao lado do outro. O trecho de código adiante definirá o comportamento do `hover`, que será responsável por aplicar uma cor cinza (`#bdbdbd`) quando o mouse estiver sobre a tag `<a>`.

```
a.gender-button:hover {
    background-color: #bdbdbd;
}
```

Resta-nos apenas os estilos para representar o clique ou toque do usuário. Para isso, criaremos outras duas definições registrando o plano de fundo como verde (`#00C853`) independente de o mouse estar ou não sobre a tag.

```
a.selected-gender-button {
    background-color: #00C853;
}

a.selected-gender-button:hover {
    background-color: #00C853;
}
```

Neste caso, foi necessário criar outro efeito `hover` com a mesma cor (`#00C853`) apenas para descartar o `hover` da classe anterior. Afinal, não seria agradável desvanecer o verde para o

cinza quando ocorrer o `houver` depois que a seleção do gênero já tivesse sido realizada.

Assim, finalizamos o CSS. Por ser muito simples, não vou replicá-lo integralmente aqui. Caso precise, ele pode ser obtido em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/components/GenderButton/index.css.

Agora vamos ao código do componente. Crie um arquivo chamado `index.jsx` dentro da pasta `GenderButton`. Como ele será *stateless*, comece pelas importações e pelo esqueleto da `function`, devidamente exportado. Note a importação do `index.css` e do componente `GenderImage`.

```
import React from 'react'
import './index.css'
import GenderImage from '../GenderImage'

export default function GenderButton(props) {
}
```

Esse componente receberá três propriedades, a saber:

1. Um booleano `selecionado`, cuja utilização será reservada à escolha das classes CSS apropriadas. Se ele for `true`, aplicaremos as classes `gender-button` e `selected-gender-button`. Do contrário, apenas aplicaremos `gender-button`.
2. Uma função `atualizarGenero()`. Como veremos nos próximos parágrafos, a atualização dos dados do usuário ocorrerá no componente `NovoUsuario`. De forma semelhante à que fizemos no `Input`, a *callback* do método `atualizarGenero()` de `NovoUsuario` será passada para `GenderSelector`, e repassada para `GenderButton` tratá-

la.

3. Uma string `genero`, que será repassada para o componente `GenderImage` e para a função `atualizarGenero()` identificar qual hyperlink de gênero foi clicado.

Com a lógica introduzida, continuaremos a implementação do código aos poucos. A definição da classe de estilo, realizada pela propriedade `className`, será feita com uma condição ternária direto na tag `<a>`. O trecho de código associado ficará da seguinte forma:

```
className={
    props.selecionado ?
    "gender-button selected-gender-button" :
    "gender-button"
}
```

Como já conversamos, essa tag `<a>` recepcionará o toque (ou clique) do usuário. Isso significa declarar o evento `onClick`. Como estamos em um componente *stateless* e o comportamento dessa função é simplesmente chamar a *callback* `atualizarGenero()`, adotaremos um código por *arrow functions*, conforme exposto adiante. Note a passagem de `e` (evento) e da propriedade `props.genero` por parâmetro.

```
onClick={e => props.atualizarGenero(e,props.genero)}
```

Só nos resta chamar o componente `GenderImage` entre a abertura e o fechamento da tag `<a>`, e reprimir o comportamento padrão do hyperlink (acessar outra página) por meio da sintaxe `href="#!"`. Aplicando as regras estabelecidas, chegaremos ao seguinte código. As importações foram omitidas.

```
export default function GenderButton(props) {
    return (
```

```
        <a
            className={
                props.selecionado ?
                "gender-button selected-gender-button" :
                "gender-button"
            }
            href="#!"
            onClick={e => props.atualizarGenero(e,props.genero)}
        >
            <GenderImage
                genero={props.genero}
            />
        </a>
    )
}
```

Desta forma, finalizamos nosso componente. Sua lógica simples vai nos proporcionar um efeito representado na imagem a seguir.

Figura 5.19: Exibição do GenderButton

Adiante!

Criando o componente GenderSelector

Este componente vai renderizar uma `<div>` e trabalhará com três propriedades:

1. O booleano `valorInvalido`, para marcar sua borda com vermelho (`#d50000`);

2. A string `genero`, para indicar qual `GenderButton` estará selecionado;
3. A função `atualizarGenero()`, que simplesmente será repassada para `GenderButton`.

Ele será *stateless* e terá um estilo inline. Vamos começar pelo esqueleto da `function`, suas importações e sua exportação. Crie um diretório `GenderSelector` dentro do diretório em `behappywith.me/front-end/src/components/`. Em seguida, crie um arquivo chamado `index.jsx` para comportar o código, conforme adiante.

```
import React from 'react'
import GenderButton from '../GenderButton'

export default function GenderSelector(props) {
}
```

A regra para transformar a propriedade `props.genero` nas flags que definirão qual `GenderButton` estará selecionado é simples. Criaremos duas constantes booleanas chamadas `masculino` e `feminino`. Se o valor de `props.genero` for `'m'`, a constante `masculino` será `true`; do contrário, será `false`.

De forma semelhante, o valor `'f'` de `props.genero` vai determinar se a constante `feminino` será `true` ou `false`. Traduzindo essa lógica para o código, teremos o seguinte:

```
export default function GenderSelector(props) {
    const masculino = props.genero==='m';
    const feminino = props.genero==='f';
}
```

O próximo passo é criar a constante de estilo. Vamos precisar de apenas cinco atributos, a saber: `boxSizing`, `border`,

`borderRadius`, `padding` e `paddingBottom`. São todos elementos exclusivamente usados para tornar a exibição da `<div>` mais elegante. O único ponto de atenção é o comportamento do atributo `border`, que terá a cor da borda alternada entre cinza (`#cccccc`) e vermelho (`#d50000`) de acordo com o valor de `props.valorInvalido`.

Veja o código:

```
import React from 'react'
import GenderButton from '../GenderButton'

export default function GenderSelector(props) {
    // constantes masculino e feminino ...

    const cor = props.valorInvalido ? '#d50000' : '#cccccc';
    const estilo = {
        boxSizing: 'border-box',
        border: `1px solid ${cor}`,
        borderRadius: '5px',
        padding: '3px',
        paddingBottom: '0'
    };
}
```

Note o uso da crase, do recurso *Template Strings*, para a definição da cor da borda. Agora só nos resta a sintaxe `return`. Devemos definir uma `<div>` com a propriedade `style` apontando para a constante `estilo` e dois componentes `GenderButton` internos, um para cada gênero.

Lembre-se de que `GenderButton` recebe três propriedades:

1. O booleano `selecionado`, que receberá o valor da constante `masculino` no primeiro `GenderButton`, e o valor de `feminino`, no segundo.
2. A string `genero`, que deve receber `'m'` ou `'f'`,

dependendo do caso.

3. A função `atualizarGenero()`, que `GenderSelector` apenas repassará.

Veja adiante como isso se traduz em sintaxe JSX.

```
export default function GenderSelector(props) {
    // constantes masculino e feminino ...
    // constante estilo ...

    return (
        <div style={estilo}>
            <GenderButton
                selecionado={masculino}
                genero={'m'}
                atualizarGenero={props.atualizarGenero}
            />
            <GenderButton
                selecionado={feminino}
                genero={'f'}
                atualizarGenero={props.atualizarGenero}
            />
        </div>
    )
}
```

Pronto, o componente está concluído. Veja seu resultado:

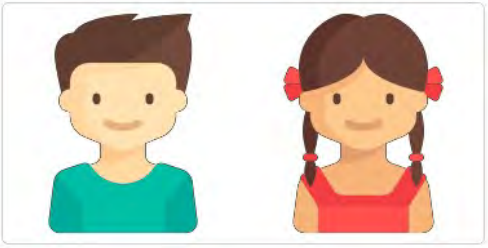

Figura 5.20: Exibição do GenderSelector

Este foi um exemplo mais simples, com o valor da propriedade

`props.valorInvalido` como `false` e nenhum componente `GenderButton` selecionado. Na próxima seção, veremos o comportamento completo.

Alterando o componente NovoUsuario

É muito importante que esteja claro o fluxo de funcionamento entre os componentes `NovoUsuario`, `GenderSelector` e `GenderButton`. Este último é quem de fato captura o evento de clique (ou toque) do usuário e simplesmente envia-o para que `NovoUsuario` atualize seu estado.

Com `this.state` devidamente atualizado, o componente `NovoUsuario` repassa-o para `GenderSelector`, que, pela lógica das constantes `masculino` e `feminino`, define qual `GenderButton` estará selecionado. Neste momento, `GenderSelector` será capaz de desselecionar, por assim dizer, um `GenderButton` no caso de o usuário ficar alterando entre as opções de gênero.

Vamos agora ajustar o código de `NovoUsuario`. O primeiro passo é importar o componente `GenderSelector`. Coloque-o no início do arquivo `index.jsx` do diretório `behappywith.me/front-end/src/components/NovoUsuario`.

```
import React from 'react'
import Label from '../Label'
import Input from '../Input'
import GenderSelector from '../GenderSelector'

// class NovoUsuario ...
```

Vamos agora acrescentar os atributos `genero` e `generoInvalido` no estado inicial de `NovoUsuario`. O atributo

`genero` deve ser definido no objeto `usuario` com o valor inicial igual a `''` (string vazia). Já o atributo `generoInvalido` deve ser criado em `validacao` e ter seu valor inicial como `false`. Essas modificações serão dispostas no `constructor` do componente, conforme o trecho de código adiante.

```
constructor(props) {
    super(props);

    this.state = {
        usuario: {
            nome: '',
            genero: ''
        },
        validacao: {
            nomeInvalido: false,
            generoInvalido: false
        }
    };
}
```

De forma semelhante à que fizemos no implementação do fluxo de `Input`, vamos criar um método chamado `atualizarGenero()` para fazer o tratamento do `onClick` realizado em `GenderButton`. Vou reproduzir adiante a *arrow function* criada em `GenderButton` apenas para relembrá-lo de que o método `atualizarGenero()` deve receber como parâmetro o evento e o `genero` associado ao clique.

```
onClick={e => props.atualizarGenero(e,props.genero)}
```

Por meio do parâmetro `genero`, seremos capazes de obter os valores do gênero e guardar no estado de `NovoUsuario`. Lembre-se também de que será necessário obter o valor de `this.state.usuario` na íntegra para atualizá-lo e, somente em seguida, salvá-lo no estado.

Mas para que precisaremos do evento `e`? Veremos agora.

```
atualizarGenero(e,genero) {
    e.preventDefault();
    let usuario = this.state.usuario;
    usuario.genero = genero;
    this.setState({
        usuario: usuario
    });
}
```

E para que serve `e.preventDefault()`? Quando codificávamos código JavaScript diretamente sobre o evento `onclick` do hyperlink padrão do W3C, era muito comum retornarmos um booleano `false` para evitar que a tag `<a>` carregasse a nova página.

```
<a href="#!" onclick="executaAlgumaCoisa(); return false">
  Opção da página
</a>
```

Não temos como fazer isso explicitamente através da sintaxe React. Por conta disso, os eventos sintéticos provocados pelas tags da JSX são decorados com a função `preventDefault()`, que tem justamente o mesmo efeito. Então, sempre que quisermos inibir o comportamento padrão das tags, faremos uma chamada explícita à `preventDefault()` nos métodos de tratamento.

Resta-nos apenas atualizar o método `render()`. Além do componente `GenderSelector`, criaremos também outra referência ao componente `Label`, desta vez com o texto `"Seu gênero:"` e sem a propriedade `htmlFor`. A sintaxe `htmlFor` pôde ser omitida, pois um clique sobre a `Label` não teria efetividade, já que `GenderSelector` não tem nenhum `<input>` para evidenciá-lo.

A chamada ao `GenderSelector` receberá três parâmetros:

1. O booleano `valorInvalido`, oriundo do estado de `NovoUsuario`;
2. A string `genero`, também obtida do estado;
3. E o método `atualizarGenero()`, com o recurso da *bound function* definido pela sintaxe `bind()`.

No trecho a seguir, omitimos as declarações dos componentes `Label` e `Input` relacionados ao nome do usuário para destacar o novo código. A própria sintaxe do `render()` e do `return` também foram omitidas.

```
<div className="center">
    <form className="pure-form pure-form-stacked">
        {/* Componentes Label e Input do nome */}

        <Label
            texto="Seu gênero:"
            valorInvalido={this.state.validacao.generoInvalido}
        />
        <GenderSelector
            valorInvalido={this.state.validacao.generoInvalido}
            genero={this.state.usuario.genero}
            atualizarGenero={this.atualizarGenero.bind(this)}
        />
    </form>
</div>
```

Assim, concluímos o último ajuste de código deste capítulo. Execute o servidor de desenvolvimento e manipule a tela por completo, clicando nas opções de gênero e preenchendo o campo de nome. Se você quiser simular o comportamento de sinalização de invalidade do nome ou gênero, altere para `true` os atributos `nomeInvalido` e `generoInvalido`, respectivamente. A figura adiante só mostrar o comportamento típico.

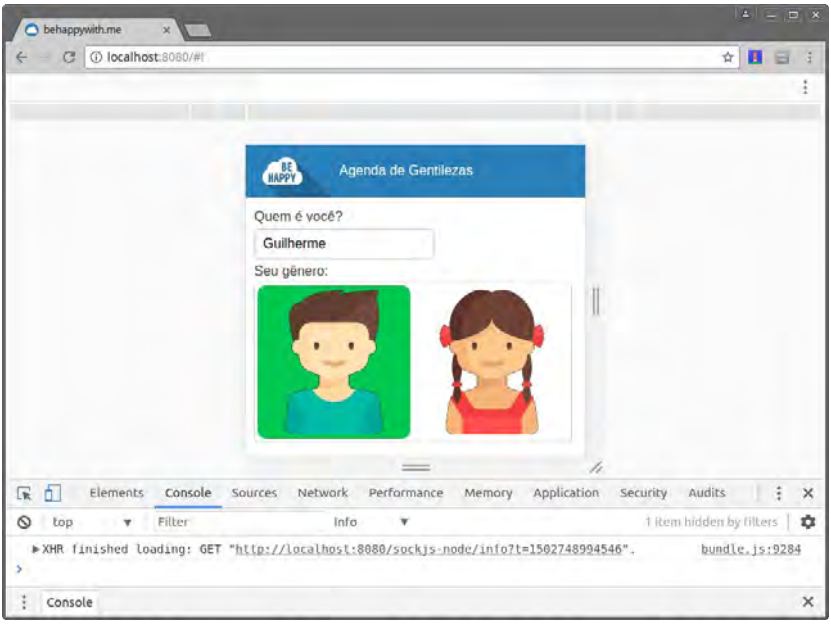

Figura 5.21: Versão final dos componentes Input e GenderSelector

Visualize-a também sem o recurso `Toggle device toolbar`, ícone no topo esquerdo das ferramentas de desenvolvimento do Google Chrome (em outras palavras, com o navegador em seu comportamento padrão de desktop). Assim você poderá ver o efeito cinza do `hover` sobre os `GenderButton` em ação.

5.4 CONCLUSÃO

Este capítulo nos mostrou todo o poder de fogo do React. Analisamos de perto como trabalhar com componentes *stateful* e as boas práticas para manipulação e acesso aos dados. Vimos que o React preconiza que o fluxo de dados na hierarquia ocorra a partir dos componentes pais para os componentes filhos. O fluxo dos

filhos para os pais deve ser realizado por meio de funções *callback*.

Concluímos com uma prática de exibição de imagens por Sprite que favoreceu a performance e a criação dos componentes associados à captura do gênero do usuário. Por enquanto, não há nenhum tratamento para os dados informados, e os valores de `nomeInvalido` e `generoInvalido` são estáticos. No próximo capítulo, vamos aprimorar esse comportamento e também elaborar os componentes `Button` e `ImageScroller`.

Capítulo 6

COMPONENTES COMPLEXOS E DOMÍNIO DA APLICAÇÃO

Neste capítulo, concluiremos os últimos componentes do livro, mais especificamente os componentes `Button` , `ImageScroller` e alguns desdobramentos. Veja na figura adiante uma representação do trabalho a ser realizado.

Figura 6.1: Ajustando protótipos

Observe que, diferentemente do `GenderImage` , não vamos criar um componente de imagem para os avatares. Teremos apenas um método `renderizarImagem()` interno ao `ImageScroller` . Precisaremos também de outro componente implícito chamado

`Toast` para exibir as mensagens de sucesso e falha do componente `Button`.

6.1 O DOMÍNIO DA APLICAÇÃO

Fizemos uma introdução relevante do padrão arquitetural MVC (*Model-View-Controller*) no *Capítulo 2*. Ressaltamos também que o React é um framework centrado na View. Na prática, isso quer dizer que a estrutura de seus componentes foi concebida para suportar a construção de elementos para a visão do projeto.

Entretanto, isso não implica na proibição do uso do *Model* (ou, em português, *modelo de domínio*). Apenas significa que o React não se importa se usarmos ou não um *Model*. Certo, mas o que exatamente *faz* um *Model*? Depende do nível de abstração que quisermos atribuir a ele.

Vamos agora evoluir essa discussão com a efetiva criação do *modelo de domínio* (ou Model). A palavra *domínio* no contexto do desenvolvimento de software evoca uma ideia de representação do conhecimento que é mantido pela aplicação. Em outras palavras, ele é uma abstração do mundo real, representada em objetos *modelados*, específica e especialmente para tornar a informação presente dentro do sistema. Às vezes, ela é enriquecida com regras de negócio, validação de dados, persistência de dados, entre outros elementos. O quão próximo a representação está do objeto real depende justamente do *domínio* que a aplicação pretende absorver.

Há duas formas de representar um domínio: *Anemic Domain Model* (modelo de domínio anêmico) ou *Rich Domain Model*

(modelo de domínio rico). O primeiro, mais simplório, apenas contempla objetos com *getters* e *setters* para manter os dados. O segundo mantém regras de negócio dentro dos objetos, com o objetivo de concentrar a inteligência da aplicação nesta camada. Em nossa PWA, vamos trabalhar com um modelo de dados rico.

Primeira versão da classe Usuario

Nosso objeto do modelo será implementado por meio de uma classe chamada `Usuario`. Primeiramente, crie o diretório `behappywith.me/front-end/src/models`. Dentro dele, crie um arquivo chamado `Usuario.js`. Vamos começar por seu esqueleto e sua exportação.

```
class Usuario {
}

export default Usuario;
```

Os atributos serão organizados pelo `constructor` da classe. Para inicializá-los, precisamos apenas adotar uma sintaxe `this.nomeAtributo`.

```
class Usuario {
    constructor() {
        this.nome = ''
        this.genero = ''
    }
}
```

Agora vamos definir os métodos de validação `validarNome()` e `validarGenero()`, a iniciar pela validação do nome. Apesar de termos definido a propriedade `maxLength` para o componente `Input`, que evita strings maiores do que 40 caracteres, vamos replicá-la aqui por se tratar de uma das

validações atribuídas ao nome. Além disso, é necessário verificar se o valor de `this.nome` é uma `string` e se está preenchida (`length != 0`).

```
validarNome() {
    if (
        typeof this.nome === 'string' &&
        this.nome.length != 0 &&
        this.nome.length <= 40) {
        return true;
    }
    return false;
}
```

A validação do gênero é bem similar, com a diferença de que os valores devem ser `'m'` ou `'f'` . Há várias formas de implementá-lo. Vamos utilizar o método auxiliar `some()` , outro excelente recurso do ECMAScript 6.

O método `some()` recebe uma função *callback* que é capaz de comparar todos os valores do array e retornar `true` se, pelo menos, um deles atender à condição explícita.

```
validarGenero() {
    return ['m','f'].some(param => {
        return this.genero === param
    })
}
```

A primeira versão da classe de modelo `Usuario` está pronta. O próximo ajuste será aplicado ao componente `NovoUsuario` .

Alterando o componente NovoUsuario

No capítulo anterior, concluímos a definição do estado de `NovoUsuario` com duas propriedades principais:

1. `usuario` , para guardar os dados do usuário;

2. `validacao`, para manter os booleanos associados à validação dos dados. Inicialmente, eles foram definidos como `false`.

A partir de agora, em vez de deixar a definição dos campos da propriedade `usuario` por conta do componente `NovoUsuario`, vamos importar a classe `Usuario`, instanciá-la e guardá-la no estado `this.state.usuario`.

De início, acrescente a importação do arquivo `Usuario.js` conforme o trecho adiante. Como `NovoUsuario.jsx` encontra-se na pasta `behappywith.me/front-end/src/components/`, e a pasta `models` está sob `src`, a importação precisou retornar dois diretórios (`../../`) para, em seguida, entrar em `models`.

```
// outras importações ...
import Usuario from '../../models/Usuario'

class NovoUsuario extends React.Component {
    // ...
}
```

O próximo e último passo é ajustar o `constructor` para, em vez de definir os atributos `nome` e `genero` explicitamente, instanciar um novo objeto `Usuario` e guardá-lo no estado de mesmo nome. O restante do código foi omitido.

```
constructor(props) {
    super(props);

    this.state = {
        usuario: new Usuario(),
        validacao: {
            nomeInvalido: false,
            generoInvalido: false
        }
    };
}
```

Os métodos `atualizarNome()` e `atualizarGenero()` não precisam ser ajustados. Eles continuam resgatando o estado `this.state.usuario` e atualizando-o conforme os valores recebidos pelos eventos `onChange` e `onClick` – respectivamente disparados pelos componentes `Input` e `GenderSelector`. Como eles acessam os atributos `nome` e `genero`, e ambos estão disponíveis no objeto de modelo `Usuario`, tudo permanecerá funcionando como antes. O grande diferencial, como veremos na próxima seção, será na validação desses dados.

6.2 CRIANDO O COMPONENTE BUTTON

`Button` será um componente `stateless` com algumas propriedades para auxiliar na definição do estilo e um evento `onClick` gerenciado por componentes de hierarquia mais alta. Quanto ao estilo, basicamente vamos utilizar as classes `pure-button` e `pure-button-primary` do Pure.css, e um estilo customizado para definir as cores e tamanhos adequados para nossa PWA.

UX IDEAL PARA BOTÕES MOBILE

Há estudos completos sobre quais são o tamanho e o espaçamento ideais para botões de uma aplicação mobile (DANDEKAR; RAJU; SRINIVASAN, 2013). Para atender às boas práticas, vamos fixar o tamanho padrão do componente `Button` com uma altura de 38 pixels e comprimento de 120 pixels. Vamos também manter os botões secundários (*Cancelar*, *Voltar* etc.) totalmente à esquerda, e o principal (*Salvar*, que geralmente é único no formulário), à extrema direita.

Crie uma pasta chamada `Button` em `behappywith.me/front-end/src/components/` . Dentro dela, crie um arquivo `index.jsx` e codifique o esqueleto do componente *stateless*, conforme adiante.

```
import React from 'react'

export default function Button(props) {
}
```

Vamos agora criar uma simples lógica ternária para aplicar ou não a classe `pure-button-primary` a uma constante `classes` (que posteriormente será atribuída ao atributo `className`). Como o próprio nome esclarece, a classe `pure-button-primary` do Pure.css é utilizada para os botões primários. Ela deve ser acrescida à classe `pure-button` .

A decisão de ser ou não primário estará baseada na propriedade booleana `props.principal` . Codifique o trecho a

seguir logo no início da função `Button`.

```
export default function Button(props) {
    const classes = props.principal ?
        'pure-button pure-button-primary' : 'pure-button';
}
```

Vamos agora criar uma constante `estilo` para incorporar outras qualidades à tag `<button>`. As propriedades `width`, `height` e `marginTop` serão necessárias para explicitamente melhorar a usabilidade. No caso de `marginTop`, afastar minimamente os botões do topo (componente `GenderSelector`) também é uma boa prática (a fim de evitar que o usuário erre o toque).

O trecho mais importante está relacionado às cores de fundo (`backgroundColor`) e ao posicionamento (`float`). Ambos estão ligados ao valor do booleano `props.principal`. Se for `true`, o botão será azul (`#2c80b9`) e ficará posicionado à direita (`float: 'right'`). Do contrário, a cor será cinza (`#e6e6e6`) e flutuante à esquerda (`float: 'left'`). Vejamos o código.

```
import React from 'react'

export default function Button(props) {
    // const classes ...

    const estilo = {
        boxSizing: 'border-box',
        backgroundColor: props.principal ? '#2c80b9' : '#e6e6e6',
        float: props.principal ? 'right' : 'left',
        marginTop: '10px',
        width: '120px',
        height: '38px'
    }
}
```

Falta-nos o retorno da função. Além da propriedade

`principal`, teremos também as propriedades `onClick` e `texto` para, respectivamente, repassar o tratamento do evento de clique e definir a *label* interna do botão. Veja adiante apenas o trecho de `return`.

```
return (
    <button
        className={classes}
        style={estilo}
        onClick={props.onClick}
    >
        {props.texto}
    </button>
)
```

Com isso, concluímos o componente `Button`. No próximo tópico, faremos os ajustes necessários em `NovoUsuario`.

6.3 AJUSTANDO O COMPONENTE NOVOUSUARIO

A principal ideia aqui é evocar os métodos de validação `validarNome()` e `validarGenero()` do objeto `Usuario` através da captura do evento `onClick` e, a partir do resultado encontrado, avançar para a próxima visão da tela de cadastro ou exibir os campos inválidos para o usuário. Vamos começar com a renderização do componente `Button` no método `render()` de `NovoUsuario`.

A única propriedade de `Button` que exige maior cuidado é `onClick`. Vamos defini-la por meio de um método chamado `validar()`, que, por enquanto, apenas vai imprimir um texto qualquer no console do navegador. Para evitar qualquer comportamento inesperado, colocaremos também a chamada ao

`preventDefault()`.

Abra o arquivo `NovoUsuario/index.jsx` e codifique o método `validar()` conforme o trecho de código adiante:

```
validar(e) {
    e.preventDefault();
    console.log('O botão Próximo foi clicado...');
}
```

Passaremos esse método através da *bound function* `bind()`. A propriedade `texto` receberá a string `'Próximo'`, e `principal` pode ser passada apenas como uma propriedade sem valor. Lembre-se também de importar o componente `Button` no início do arquivo. Veja a seguir os trechos da importação e da chamada de `Button` no método `render()`.

```
// Outras importações ...
import Button from '../Button'

class NovoUsuario extends React.Component {
    // Outros métodos ...
    render() {
        return (
            <div className="center">
                <form className="pure-form pure-form-stacked">
                    {/*
                        Label do nome ...
                        Input do nome ...
                        Label do gênero ...
                        GenderSelector ...
                    */}

                    <Button
                        principal
                        texto="Próximo"
                        onClick={this.validar.bind(this)}
                    />
                </form>
            </div>
        );
```

```
    }
}
```

Na figura adiante, à esquerda, temos o layout da primeira visão de cadastro de usuários após a inclusão do botão de próximo. À direita, veja que o console do navegador exibiu a string `'O botão Próximo foi clicado...'` após um toque realizado no botão.

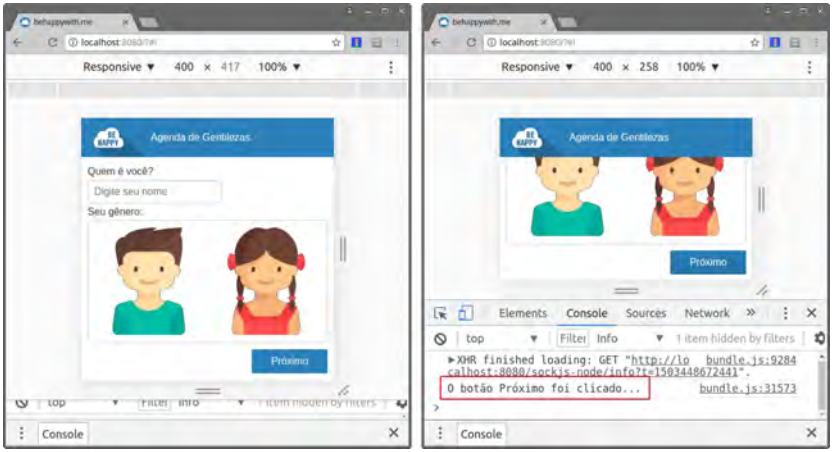

Figura 6.2: Botão Próximo

Com o evento `onClick` funcional, vamos agora validar os dados do `nome` e `genero`. Se estiverem válidos, exibiremos a próxima tela (para seleção do avatar). Dentro do método `validar()`, precisamos obter o objeto `Usuario` salvo em `this.state.usuario` e convocar seus métodos de validação dos campos.

O resultado de cada método será salvo no objeto `validacao`, também obtido no estado do componente. Fica mais fácil de compreender a lógica pelo código. Abra o arquivo `NovoUsuario/index.js` e altere o método `validar()`:

```
validar(e) {
    e.preventDefault();
    let usuario = this.state.usuario;
    let validacao = this.state.validacao;
    validacao.nomeInvalido = ! usuario.validarNome();
    validacao.generoInvalido = ! usuario.validarGenero();

    this.setState({
        validacao: validacao
    });
}
```

Do ponto de vista da organização dos componentes React criados até o momento, essa pequena alteração já provoca um efeito interessante. Veja adiante uma figura em que o botão *Próximo* foi clicado com os campos em branco.

Figura 6.3: Campos inválidos

As duas próximas alterações serão para definir uma mensagem

de erro adequada de acordo com o preenchimento inválido dos campos, e criar um mecanismo para o componente avançar ou não para a próxima tela de cadastro.

Mensagens com o componente Toast

Durante o surgimento dos aplicativos mobile, ganhamos também alguns componentes particulares à usabilidade de toque. Um deles é o componente de mensagem *Toast* que, segundo as lendas, teve seu nome baseado naqueles pães torrados com torradeiras (*toasts*). Esse tipo de torradeira, depois que termina seu processo de torrar o pão, executa um movimento abrupto para jogá-lo para fora. Este mesmo movimento é percebido nas mensagens de aviso Toast.

Neste tópico, veremos como *baixar* e configurar o componente *React Toastify*. Este é um projeto publicado no NPM (https://www.npmjs.com/package/react-toastify) que oferece uma excelente opção de Toast totalmente adaptado ao React. Além de muito simples de ser usado, ele possui cinco variações de Toast e a possibilidade de customização.

Vamos começar com a instalação do pacote necessário através do NPM. Em seu console (ou prompt de comando), entre no diretório `behappywith.me/front-end` e execute o comando a seguir.

```
npm install --save react-toastify@2.0.0-rc.3
```

A sintaxe de uso do Toast é muito simples, mas, para não poluirmos os demais componentes com código *alienígena*, criaremos um componente para encapsular a lógica de execução das mensagens. Com sua IDE, crie uma pasta chamada `Toast` no

diretório `behappywith.me/front-end/src/components/` e, dentro dela, crie o arquivo `index.jsx` com o código adiante.

```
import React from 'react';
import { ToastContainer, toast } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.min.css';

class Toast extends React.Component {
}

export default Toast;
```

O próximo passo é criar os métodos de exibição do *Toast*. Para evitar que os componentes externos conheçam a sintaxe de exibição do tipo da mensagem, vamos criar métodos especialistas que vão convocar os métodos originais `success`, `info` e `error` para os casos apropriados. A mensagem a ser exibida será informada pelo parâmetro `msg`.

```
sucesso(msg) {
    toast.success(msg)
}
info(msg) {
    toast.info(msg)
}
erro(msg) {
    toast.error(msg)
}
```

O método `render()` possui algumas particularidades intrínsecas ao React Toastify. Podemos resumi-las nas seguintes definições:

1. A propriedade `position` indica em qual trecho da tela a mensagem será exibida.
2. O Toast, por padrão, é fechado (some da tela) automaticamente. O tempo em milissegundos é definido pela propriedade `autoClose`. No nosso caso, teremos o

valor 5.000 milissegundos (5 segundos).

3. O booleano `hideProgressBar` permite esconder a barra de progresso. Particularmente, prefiro escondê-la (`true`), pois deixa a interface mais leve. Fique à vontade para alterar, se quiser.
4. A propriedade booleana `closeOnClick` permite fechar o Toast com um clique.
5. Por fim, a propriedade `pauseOnHover` interrompe a contagem do tempo para fechamento do Toast quando o mouse estiver sobre ele.

Com isso, temos o seguinte código:

```
render() {
    return (
        <ToastContainer
            position="bottom-center"
            autoClose={5000}
            hideProgressBar={true}
            closeOnClick
            pauseOnHover
        />
    );
}
```

Assim, finalizamos o componente `Toast` .

Possível problema de compatibilidade

Ao clonar este projeto do GitHub e executar o comando `npm start`, é possível que você enfrente problemas de compatibilidade do *React Toastify*, mesmo tendo restringido a versão `2.0.0-rc.3`.

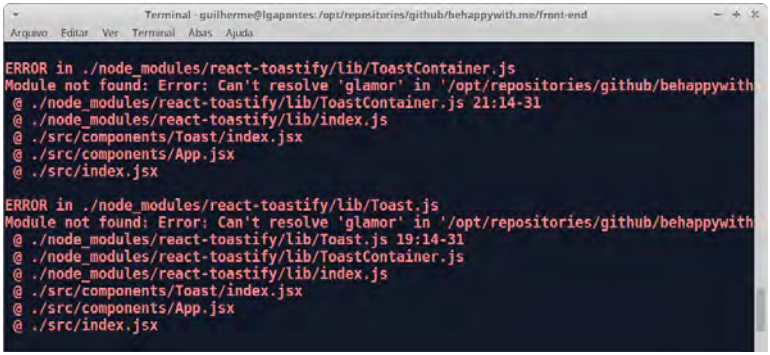

Figura 6.4: Possível problema de compatibilidade

Se você se deparar com erros como este, execute separadamente o comando de instalação do *React Toastify*, apresentado no início deste tópico, e tente novamente. Outra alternativa é retirar o `^` no início da versão do componente no arquivo `package.json` e reinstalar todos os pacotes.

Caso queria se aprofundar, existem outros recursos disponíveis na API do React Toastify que podem ser observados na página do projeto (https://fkhadra.github.io/react-toastify/).

Ajustando o componente App

Todas as mensagens *toast* serão exibidas diretamente a partir do componente `App`. A lógica será a seguinte: vamos declarar o componente `<Toast />` no JSX de `App`. Quando um componente precisar usufruir dele, vamos repassar propriedades com funções *callback* que internamente convocarão os métodos `sucesso()`, `info()` ou `erro()`, dependendo de cada caso.

Abra o arquivo `front-end/src/components/App.jsx`. O primeiro passo é importar o componente `Toast`.

```
import React from 'react';
import Header from './Header';
import NovoUsuario from './NovoUsuario';
import Toast from './Toast';

class App extends React.Component {
    // ...
}
```

Segundo as boas práticas de codificação da documentação oficial do React, como o *React Toastify* é uma biblioteca de terceiros, podemos convocar seus métodos através das *referências* (`this.refs`). A sintaxe `refs` é um recurso React disponível para acessarmos um determinado componente diretamente (como fazíamos com o método `getElementById()`, muito utilizado no JavaScript padrão). Ele nos permitirá usar a sintaxe `this.refs.toast.sucesso()` dentro das *callbacks* sem repassar eventos ao componente `ToastContainer`.

Vamos então ajustar o método de renderização de `App` em dois pontos:

1. Repassar uma propriedade chamada `erro` para o componente `NovoUsuario` com uma *arrow function* que invocará o método `erro()` da referência

`this.refs.toast`.

2. Acrescentar a tag `<Toast />` com a propriedade `ref` definida com a string `"toast"`. Este é o *ID* da referência que podemos acessar pela sintaxe `this.refs.toast`.

Veja a seguir como ficará o código de `App.render()`.

```
class App extends React.Component {
    render() {
        return (
            <div>
                <Header />
                <NovoUsuario
                    erro={msg=>this.refs.toast.erro(msg)}
                />
                <Toast ref="toast" />
            </div>
        );
    }
}
```

Salve as alterações no arquivo `App.jsx`. Voltaremos agora ao componente `NovoUsuario` para alterar o método `validar()` e indiretamente invocar o novo componente `Toast`.

Ajustando o componente NovoUsuario

Para concluir as alterações de `NovoUsuario` associadas aos componentes `Button` e `Toast`, vamos precisar de: uma nova variável de estado (`primeiraVisaoCompleta`) para indicar se a primeira visão está concluída; uma lógica para elaborar a mensagem de validação; e o efeito de transição entre as visões da tela.

Iniciaremos alterando o `constructor` para definir o valor padrão de `primeiraVisaoCompleta`. Esta variável será um

atributo do estado do componente, como pode ser visualizado no código a seguir.

```
constructor(props) {
    super(props);

    this.state = {
        usuario: new Usuario(),
        validacao: {
            nomeInvalido: false,
            generoInvalido: false
        },
        primeiraVisaoCompleta: false
    };
}
```

Trataremos agora da composição do texto da mensagem. Se o atributo `nome` for inválido, a mensagem deve ser `'Seu nome está inválido!'`. De forma semelhante, devemos ajustar esse texto no caso de o gênero ou ambos os campos estiverem inválidos. Isso será possível através de uma simples lógica de verificação das variáveis `nomeInvalido` e `generoInvalido` no método `validar()`.

Vamos aproveitar o resultado dessa validação para definir o valor final da variável `primeiraVisaoCompleta` e utilizá-la como gatilho para a chamada da mensagem de erro. Como a *callback* `erro()` foi repassada ao `NovoUsuario`, em caso de falha na validação, precisamos apenas convocá-la pela sintaxe `this.props.erro(mensagem)`. Veja o código do método `validar()` a seguir.

```
validar(e) {
    e.preventDefault();
    let usuario = this.state.usuario;
    let validacao = this.state.validacao;
    validacao.nomeInvalido = ! usuario.validarNome();
    validacao.generoInvalido = ! usuario.validarGenero();
```

```
    let mensagem = '';
    let primeiraVisaoCompleta = false;
    if (validacao.nomeInvalido && validacao.generoInvalido) {
        mensagem = 'Os campos nome e gênero estão inválidos!'
    } else if (validacao.nomeInvalido) {
        mensagem = 'Seu nome está inválido!'
    } else if (validacao.generoInvalido) {
        mensagem = 'Selecione seu gênero!'
    } else {
        primeiraVisaoCompleta = true;
    }
    if (!primeiraVisaoCompleta) {
        this.props.erro(mensagem);
    }

    this.setState({
        validacao: validacao,
        primeiraVisaoCompleta: primeiraVisaoCompleta
    });
}
```

Por fim, resta-nos tratar da lógica de quais campos exibir em cada visão. Como sabemos, a tela de cadastro possui duas visões: entrada dos campos nome e gênero e, posteriormente, a entrada do avatar.

Podemos sintetizar as diferenças nos seguintes pontos:

1. A primeira visão deve exibir os campos de nome (aberto para preenchimento), de gênero, e o botão *Próximo*.
2. A segunda visão continua exibindo o campo nome, porém já preenchido com o atributo `nome` do objeto `Usuario`. Além disso, temos o campo de seleção dos avatares, um botão *Voltar* para exibir a primeira visão, e um botão *Salvar* que efetivamente salvará o cadastro.

Mas como faremos para o método `render()` exibir as visões

adequadas? Simples: podemos criar condições para exibir um ou outro componente acordando com o estado `this.state.primeiraVisaoCompleta`. Por questões de boa manutenibilidade, é mais agradável manter esse tipo de lógica dentro de métodos próprios (como `renderizarBotoes()`, por exemplo) e depois chamá-los no método `render()`.

Adotando essa ideia para os componentes associados ao nome (`Label` e `Input`), podemos condicionar a propriedade `Input.readOnly` com o valor contrário de `this.state.primeiraVisaoCompleta`. Ou seja, a primeira visão deve permitir que o usuário digite seu nome, enquanto a segunda deve exibi-lo apenas como leitura.

Vamos encapsular isso em um novo método chamado `renderizarNome()`. Codifique-o com o trecho a seguir:

```
renderizarNome() {
    return (
        <section>
            <Label
                htmlFor="nome"
                texto="Quem é você?"
                valorInvalido={this.state.validacao.nomeInvalido}
            />
            <Input
                id="nome"
                placeholder="Digite seu nome"
                maxLength="40"
                readOnly={this.state.primeiraVisaoCompleta}
                valorInvalido={this.state.validacao.nomeInvalido}
                defaultValue={this.state.usuario.nome}
                onChange={this.atualizarNome.bind(this)}
            />
        </section>
    )
}
```

Há um detalhe importante: você deve ter notado o uso de uma tag `<section>` ao redor dos componentes `Label` e `Input`. Isso se fez necessário por conta daquela restrição do JSX que vimos no passado, que nos obriga a retornar apenas **um** componente por vez. O método `renderizarNome()` retorna um código JSX, logo, também é regido por essa restrição.

Mantendo a mesma lógica, crie um método chamado `renderizarGenero()`. Nele, teremos uma condição que, a partir do valor de `this.state.primeiraVisaoCompleta`, retornará `null` (outra restrição do JSX) ou os componentes `Label` e `GenderSelector`. De forma semelhante ao método `renderizarNome()`, vamos encapsular tudo em uma tag semântica `<section>`.

```
renderizarGenero() {
    if (this.state.primeiraVisaoCompleta) {
        return null
    } else {
        return (
            <section>
                <Label
                    texto="Seu gênero:"
                    valorInvalido={this.state.validacao.generoInv
alido}
                />
                <GenderSelector
                    valorInvalido={this.state.validacao.generoInv
alido}
                    genero={this.state.usuario.genero}
                    atualizarGenero={this.atualizarGenero.bind(th
is)}
                />
            </section>
        )
    }
}
```

Vamos agora à definição do método `renderizarBotoes()`. Além do botão *Próximo*, aplicado à primeira visão, precisaremos retornar os botões *Voltar* e *Salvar* na segunda visão. O comportamento do botão *Salvar* será tratado no futuro. Por enquanto, vamos deixá-lo sem evento `onClick`.

O botão *Voltar* simplesmente vai alterar o estado `this.state.primeiraVisaoCompleta` para `false`, fazendo com que a tela de cadastro exiba novamente a primeira visão. E, é claro, por se tratar de um `<button>`, a chamada à função `preventDefault()` também é necessária. Por ser muito simples, vamos implementá-lo com ajuda das *arrow functions*. Veja o código a seguir.

```
renderizarBotoes() {
    if (this.state.primeiraVisaoCompleta) {
        return (
            <section>
                <Button
                    texto="Voltar"
                    onClick={e => {
                        e.preventDefault();
                        this.setState({
                            primeiraVisaoCompleta: false
                        });
                    }}
                />
                <Button
                    principal
                    texto="Salvar"
                />
            </section>
        )
    } else {
        return (
            <section>
                <Button
                    principal
                    texto="Próximo"
```

```
                    onClick={this.validar.bind(this)}
                />
            </section>
        )
    }
}
```

A segregação da renderização dos componentes em métodos fará com que o método `render()` seja extremamente simples. Precisamos apenas convocar os métodos na ordem correta, e pronto. Veja o resultado a seguir.

```
render() {
    return (
        <div className="center">
            <form className="pure-form pure-form-stacked">
                {this.renderizarNome()}
                {this.renderizarGenero()}
                {this.renderizarBotoes()}
            </form>
        </div>
    );
}
```

Salve o arquivo `NovoUsuario/index.jsx`, execute o servidor de desenvolvimento e desfrute da agradável sensação de ver as coisas funcionando. Na imagem adiante, à esquerda, temos os campos nome e gênero preenchidos. Ao clicar em *Próximo*, conforme esperado, o componente `Input` tornou-se `readOnly`, o componente `GenderSelector` foi omitido e os botões *Voltar* e *Salvar* tomaram seu lugar na parte inferior da página.

Figura 6.5: Renderização das visões da tela de cadastro

Por outro lado, caso os campos da primeira visão estejam inválidos, ao clicar no botão *Próximo*, você vai se deparar com algo semelhante à imagem a seguir.

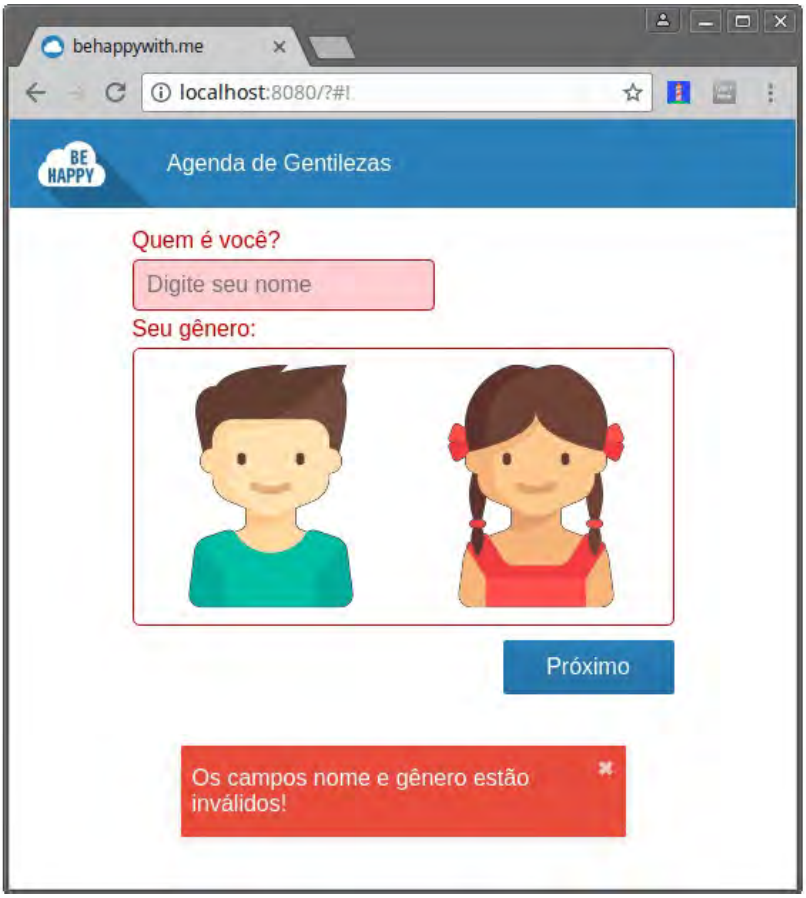

Figura 6.6: Renderização das visões da tela de cadastro

E assim concluímos a criação dos componentes `Button` e `Toast`. Concluímos também os ajustes da estrutura dos componentes `App` e `NovoUsuario` para suportar a transição entre as visões da tela de cadastro e exibir a mensagem em caso de falha no preenchimento dos campos.

6.4 CRIANDO O COMPONENTE IMAGESCROLLER

A partir de agora, desbravaremos o componente mais complexo do livro. Além de exigir uma organização mais sofisticada, no sentido de manter vários métodos para dividir sua lógica, precisaremos nos aprofundar nos eventos capturados no DOM e recebidos pelo código React. Ele terá alguns componentes internos:

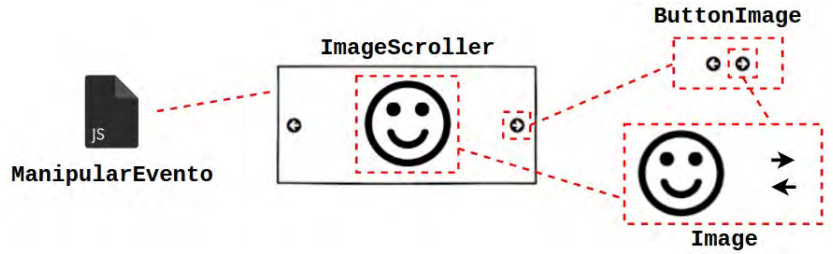

Figura 6.7: Componentes internos do ImageScroller

O `ImageSroller` em si será uma tag `<div>` altamente rebuscada. Como sua lógica de deslocamento das imagens internas não é trivial, usaremos uma classe JavaScript chamada `ManipularEvento.js` só para essa função.

A imagem dos avatares e as setinhas serão componentes `Image` . No caso especial das setas, criaremos ainda um novo componente chamado `ButtonImage` para encapsular a lógica e exibição especializada das imagens sob uma `<div>` clicável, conforme estudaremos a partir de agora.

Codificando o componente ButtonImage

Vamos começar pelo nível mais baixo da hierarquia. Como já implementamos o componente `Image`, resta-nos sobrepô-lo com uma `<div>` capaz de receber os eventos do `ImageScroller`. Assim como fizemos com o `GenderSelector`, o componente `ButtonImage` trabalhará com *Image Sprites*, organizando as imagens de 48 pixels lado a lado. Como adendo, note que, além das setas, esta imagem terá outros botões disponíveis na aplicação como um todo, o que totaliza um total de 336 pixels horizontais.

Figura 6.8: Imagens do ButtonImage

Baixe o arquivo `botoes.png` no link https://behappywith.me/img/botoes.png e salve-o dentro do diretório `behappywith.me/front-end/src/components/Image/img`. Agora será necessário importá-lo peloo componente `Image`. Abra o arquivo `src/components/Image/index.jsx` e acrescente a importação de `botoes.png` conforme o trecho de código a seguir.

```
import React from 'react'
import './img/avatars.png'
import './img/botoes.png'

class Image extends React.Component {
    // ...
}
```

Iniciaremos o código do componente pelo CSS. Crie uma pasta chamada `ButtonImage` em `behappywith.me/front-end/src/components/` e crie o arquivo `index.css`. Ele terá dois

estilos: a da `<div>` e o da `<div>` com `hover`. A estilização da `<div>` aplicará uma borda totalmente arredondada com a propriedade `border-radius`, um `padding` de `4px`, uma cor de fundo cinza (`#bdbdbd`) e uma configuração de `margin` adequada ao `ImageScroller`.

A estilização do `<div>` com o `hover` apenas aplicará uma cor mais escura de cinza (`#8d8d8d`), dando ao usuário de desktop o *affordance* mais apropriado. Veja a seguir o código.

```
div.option-image-scroller {
    border-radius: 50%;
    padding: 4px;
    background-color: #bdbdbd;
    margin: 62px 2px 0 2px;
}

div.option-image-scroller:hover {
    background-color: #8d8d8d;
}
```

Agora vamos ao JSX. Crie um arquivo chamado `index.jsx` em `src/components/ButtonImage`; inclua as importações do React, do CSS, do componente `Image`; e codifique a estrutura básica da `function`.

```
import React from 'react'
import './index.css'
import Image from '../Image'

export default function ButtonImage(props) {

}
```

Vamos agora criar três variáveis que serão importantes para o funcionamento do componente: um objeto `estilo`, `index` (para guardar o valor do `eixoX` aplicado ao componente `Image`) e uma constante `tamanho`. Os valores de `estilo` e `index` vão

variar de acordo com uma propriedade chamada `posicao`.

Quando `posicao` assumir o valor `direita`, precisamos exibir a imagem da seta para a direita (com `index` igual a 1) e posicioná-la com `float = 'right'`. Caso contrário, teremos `index` igual a `0` e `estilo` populado com `float = 'left'`. Isso é facilmente construído com uma condição simples baseada no valor de `props.posicao`.

A constante `tamanho` vai receber 30 para facilitar sua propagação às propriedades `width`, `height` e `backgroundHeight` do componente `Image`. Vejamos o código.

```
export default function ButtonImage(props) {
    let estilo = {};
    let index = 0;
    if (props.posicao === 'direita') {
        estilo.float = 'right'
        index = 1
    } else {
        estilo.float = 'left'
        index = 0
    }
    const tamanho = 30;
}
```

Agora que temos todas as variáveis definidas corretamente, podemos copiar `props` para uma variável local chamada `propriedades` e excluir a propriedade `posicao` – já que não podemos repassá-la à tag `<div>`.

```
export default function ButtonImage(props) {
    // Definição das variáveis ...

    let propriedades = Object.assign({},props);
    delete propriedades.posicao;
}
```

Por fim, resta-nos o código JSX em si. Dentro do `return` da função `ButtonImage()`, crie uma `<div>` estilizada com o objeto `estilo` e a classe `option-image-scroller` (respectivamente pelas propriedades `style` e `className`), que receba `{...propriedades}` e mantenha um componente `Image` em seu interior. Não há nada de especial a acrescentar a respeito do componente `Image`.

As propriedades `eixoX` e `eixoY` vão receber respectivamente o valor de `index` e `0`. O valor da variável `tamanho` será aplicada à `width`, `height` e `backgroundHeight`, e a propriedade `arquivo` vai receber o valor estático `"img/botoes.png"`. Veja como ficará esse trecho a seguir.

```
export default function ButtonImage(props) {
    // Definição das variáveis ...
    // Cópia de props para propriedades ...

    return (
        <div
            style={estilo}
            className='option-image-scroller'
            {...propriedades}
        >
            <Image
                eixoX={index}
                eixoY={0}
                width={tamanho}
                height={tamanho}
                backgroundHeight={tamanho}
                arquivo="img/botoes.png"
            />
        </div>
    )
}
```

Assim concluímos o componente `ButtonImage`. A partir dele,

estamos aptos a programar os mecanismos de transição entre as imagens do `ImageScroller` para os usuários desktop. Falta-nos discorrer sobre como o código do `ImageScroller` tratará dos mecanismos de *swipe* e *flinging*, próprios aos usuários mobile. Faremos isso a partir de agora.

Baixando o arquivo ManipularEvento.js

A lógica de um componente de *Image Scroller* é complexa porque construir o efeito do *deslocamento* das imagens internas à `<div>` principal requer o manuseio de variáveis próprias ao estilo CSS. Para simplificar e favorecer a manutenibilidade, vamos separar essa inteligência em dois elementos: no próprio componente `ImageScroller` e em um objeto `ManipularEvento.js`.

Isso é adequado para segregar a lógica de captura dos eventos e a renderização das tags JSX (particulares ao componente React) dos cálculos envolvidos na exibição das imagens na tela (concentrados em `ManipularEvento.js`). Na figura adiante, há uma boa representação de como o componente *ImageScroller* realiza sua mágica.

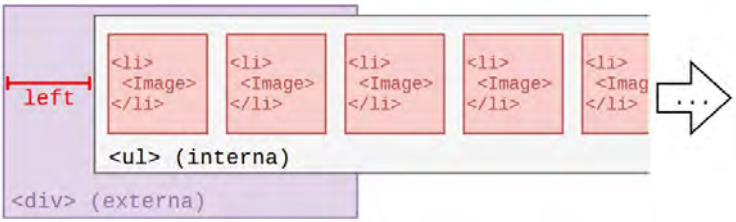

Figura 6.9: Representação do ImageScroller

A `<div>` externa terá um comprimento e uma altura fixos e,

obrigatoriamente, portará uma propriedade CSS chamada `overflow` com o valor `hidden` para evitar que apareçam barras de rolagem horizontais ou verticais. A `<ul>` interna vai encapsular tags `<li>`, que vão conter componentes `Image`.

A estilização da tag `<ul>` que é a parte principal. Ela conterá uma propriedade CSS chamada `position`, cujo valor será `relative`. Essa sintaxe do `position: 'relative'`, quando definida em conjunto com as propriedades `top`, `right`, `bottom` e `left`, vão distanciar a `<ul>` da `<div>` externa. No nosso caso, precisaremos apenas da propriedade `left`, conforme exibido na imagem.

Se `left` for positiva, a `<ul>` interna ficará deslocada para a direita. Por outro lado, se `left` for negativa, a `<ul>` interna pode ter seu início a partir de um ponto horizontal anterior à borda da `<div>` externa, dando o efeito de que as imagens estão deslocadas para a esquerda.

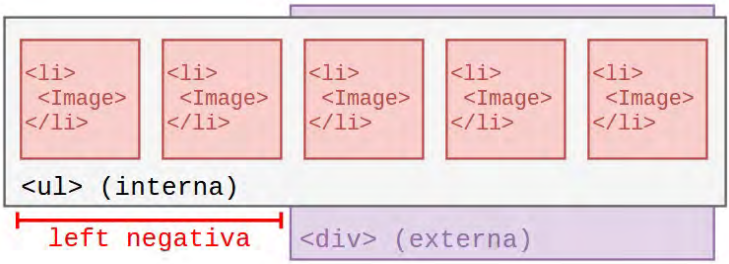

Figura 6.10: Representação do ImageScroller

Dito isso, o objeto `ManipularEvento.js` será o responsável por *calcular* o valor da propriedade `left` da `<ul>` interna de acordo com os eventos de toque (ou clique) disparados pelo usuário. Por exemplo, um evento `onTouchMove` cujo toque inicial

é menor do que o final significa que o usuário deseja visualizar as imagens que estão *escondidas* à esquerda da exibição atual – ou seja, o valor de `left` deve aumentar.

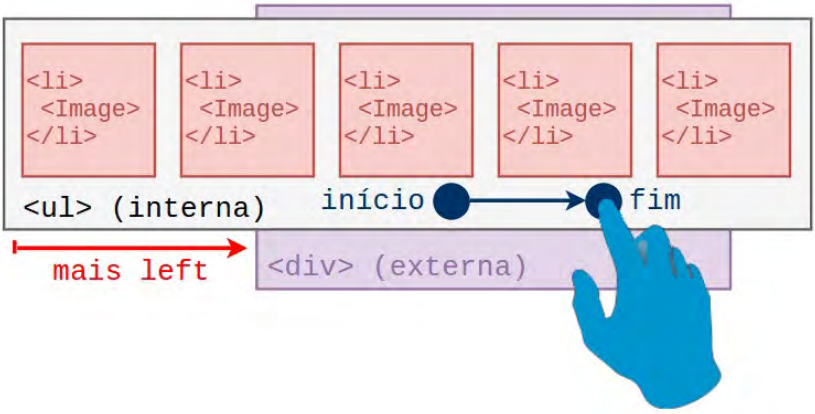

Figura 6.11: Representação do ImageScroller

De forma semelhante, se o movimento for ao contrário, o `left` vai reduzir. Essa é a *ideia* básica por trás da lógica. Mas fazer isso acontecer não é tão simples. Além dos cálculos relacionados ao `left`, `ManipularEvento.js` precisa impedir que o usuário movimente a `<ul>` interna para além dos limites inferiores e superiores.

Na prática, se uma imagem de avatar tem 140 pixels, 23 imagens terão 3220 pixels. O usuário **não** pode deslocar a `<ul>` interna de forma que a propriedade `left` fique, por exemplo, com um valor negativo de 5000 pixels. Isso faria todas as imagens ficarem ocultas na tela. Uma lógica semelhante será necessária para o deslocamento à direita.

Outro problema que precisa ser tratado é a aceitação dos

movimentos de *swipe* e *flinging*. Como vimos no *Capítulo 4*, *swipe* é o movimento de deslizar o dedo pela tela do celular, de forma contínua e natural. O *flinging* tem a mesma mecânica, porém é feito como se o usuário estivesse *jogando* alguma coisa para o lado.

O efeito *swipe* precisa de um pequeno ajuste no cálculo do `left` para reposicionar a imagem que estiver mais próxima no centro da `<div>` externa. Em outras palavras, isso vai melhorar a usabilidade. Quando o usuário fizer um *swipe* menor que o deslocamento necessário para ajustar uma imagem qualquer no centro da `<div>`, o objeto `ManipularEvento.js` precisa reajustar o valor de `left`.

O *flinging*, por outro lado – como é caracterizado por uma diferença mínima entre os toques iniciais e finais do usuário –, o deslocamento será sempre inferior ao tamanho de uma imagem. Nesse caso, o sistema vai perceber que um *flinging* foi feito e ajustar o `left` de acordo com a direção do movimento.

E o os eventos do *mouse*? Também precisam ser tratados. É ruim deslocar um seletor de imagens horizontalmente com um efeito semelhante ao *drag-and-drop* (clicar, arrastar e soltar) do mouse. Por conta disso, colocaremos pequenas setinhas nas extremidades da `<div>`. Um usuário desktop só poderá deslizar as imagens para esquerda ou direita por meio desses botões.

Para viabilizar esse efeito, o objeto `ManipularEvento.js` também é capaz de avançar ou retroceder as imagens a partir da definição de um índice. Clicar na seta para a esquerda deslocará a `<ul>` interna para a direita (aumentando o valor de `left`), enquanto clicar na seta para a direita provocará o comportamento contrário.

E, é claro, índices mínimos e máximos também são necessários para evitar que o usuário desloque a `<ul>` para além da quantidade de imagens disponíveis. Mas, o mais importante aqui é a compreensão dessa lógica.

Não precisamos nos aventurar pelas profundezas desse código. Agora que você já sabe como ele funciona, vamos baixá-lo e salvá-lo no projeto. Crie uma pasta chamada `ImageScroller` em `behappywith.me/front-end/src/components` e, dentro dela, salve o arquivo baixado no link https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/components/ImageScroller/ManipularEvento.js.

O uso correto de sua interface pública será apresentado no momento oportuno, quando o código do componente `ImageScroller` estiver sendo construído. Vamos agora nos debruçar sobre o sucinto objeto de modelo *Avatar*, responsável por guardar as propriedades relacionadas aos avatares exibidos na aplicação.

Codificando a classe Avatar

Não há nenhuma lógica sofisticada associada aos avatares disponíveis para o usuário selecionar. Um avatar só terá dois atributos: um `index`, para indicar a imagem a ser obtida do arquivo `avatars.png`; e uma descrição, para indicar um nome para a imagem selecionada. Além desses dois atributos, teremos também dois métodos: `toString()` e `obterTodos()`.

O método `toString()` será um protocolo que vamos adotar para todos os objetos usados como elementos do componente `ImageScroller`. Por meio do `toString()`, o `ImageScroller`

exibirá a descrição da imagem selecionada na parte inferior do componente, como no exemplo a seguir, cuja imagem destacada possui a descrição *Avatar 8*.

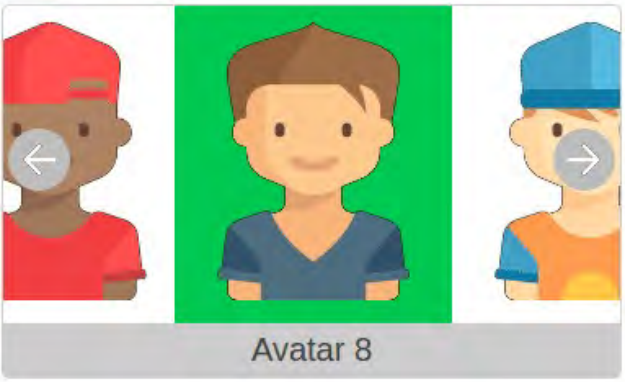

Figura 6.12: Componente ImageScroller

O método `obterTodos()` é mais rebuscado, tanto sob o ponto de vista de código quanto na arquitetura. Vamos começar pela discussão arquitetural. Nossos objetos de domínio (*Usuario*, *Avatar* etc.), além de conter a essência do negócio, também serão codificados como *Active Record* (cuja tradução não oficial é registro ativo).

Active Record é um padrão de projeto muito comum em linguagens como Ruby ou Python. Ele foi catalogado por Martin Fowler em seu excelente livro *Patterns of Enterprise Application Architecture* (Padrões de Arquitetura de Aplicações Corporativas).

A ideia do *Active Record* é criar um objeto híbrido com informações do negócio e uma inteligência para persisti-los na infraestrutura apropriada. Em outras palavras, um objeto aderente ao padrão *Active Record* conterá métodos como `inserir()` ,

`atualizar()`, `excluir()` e `consultar()`.

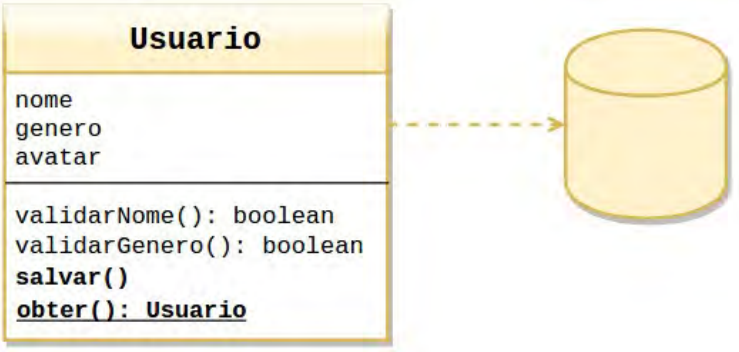

Figura 6.13: Exemplo de Active Record

No exemplo exposto nessa imagem, o método `salvar()` persistirá os dados do objeto no banco de dados. Se tivéssemos os métodos `atualizar()` ou `excluir()`, a lógica seria a mesma: a partir dos dados correntes, a persistência seria realizada adequadamente.

O método `obter()` é ligeiramente distinto. Por questões de fluência na utilização da API do objeto `Usuario`, ele pode ser convocado diretamente a partir da classe (método estático), sem a necessidade de instanciar um objeto. Na prática, ele retornará uma instância do próprio objeto `Usuario` e pode receber parâmetros para auxiliar na busca do objeto correto.

A lógica da persistência em si, relacionadas às *queries* SQL de bancos relacionais ou invocação de métodos nos bancos *noSQL* (bancos não relacionais), geralmente não são implementadas internamente nos métodos inerentes aos *Active Record*.

O que acontece na prática é que os frameworks que utilizam

esse padrão fornecem outros objetos próprios de infraestrutura para realizar este trabalho de baixo nível, deixando o objeto do domínio livre para delegar tais tarefas. Aqui, neste livro, nós também adotaremos essa estratégia. No próximo capítulo nós criaremos um objeto chamado *Repository* para centralizar o salvamento no cache do navegador.

Com esses conceitos alinhados, resta-nos concluir a explicação do método `obterTodos()`. A ideia desse método é aproximar o objeto `Avatar` do padrão *Active Record*. Com ele, retornaremos todos os avatares existentes na aplicação. Neste caso, como eles não estarão persistidos, precisaremos apenas retornar um array com as opções possíveis.

Agora, vejamos as nuances do código. Primeiro crie um arquivo chamado `Avatar.js` no diretório `behappywith.me/front-end/src/models/`. Em seguida, codifique-o com a classe `Avatar`, sua exportação, e seu `constructor` recebendo e guardando os parâmetros `index` e `descricao`.

```
class Avatar {
    constructor(index,descricao) {
        this.index = index;
        this.descricao = descricao;
    }
}

export default Avatar;
```

O método `toString()` só retornará o valor do atributo `this.descricao`. Neste caso, seu código é bem simples.

```
toString() {
    return this.descricao;
}
```

O método `obterTodos()` terá três novidades:

1. O uso da sintaxe `static` prefixada antes do nome para determinar que ele é um método estático da classe `Avatar`. Esta é outra novidade do ECMAScript 6.
2. O uso do método `fill()`, usado para atribuir um mesmo valor a todos os elementos de um array com 23 posições.
3. O método `map()`, também concebido pelo ES6 e utilizado para iterar pelos elementos do array e ajustá-los conforme função passada por parâmetro.

Cada elemento do array de avatares será um objeto do tipo `Avatar`, com seu `index` (idêntico ao índice do array) e sua `descricao` definida com a string `Avatar ${i+1}`. Veja seu código adiante.

```
static obterTodos() {
    return Array(23).fill(0).map((entry,i) => {
        return new Avatar(i,`Avatar ${i+1}`)
    })
}
```

Assim, concluímos o objeto `Avatar`. Resta-nos importá-lo e usá-lo dentro do objeto `Usuario` que, por default, deve ser inicializado com a primeira imagem de avatar – o elemento de índice `0` (zero) do array retornado pelo método `Avatar.obterTodos()`. Para implementar essa premissa, abra o arquivo `models/Usuario.js`, faça a importação do objeto `Avatar` e acrescente o atributo `this.avatar` em seu `constructor`, conforme o trecho de código seguinte.

```
import Avatar from './Avatar'

class Usuario {
    constructor() {
```

```
        this.nome = ''
        this.genero = ''
        this.avatar = Avatar.obterTodos()[0]
    }
    // ... outros métodos
}
```

Como o componente `ImageScroller` sempre trará o primeiro elemento do array já selecionado e limitará a seleção de uma das imagens disponíveis, o usuário não terá oportunidade de definir um valor inválido – fato este que nos dá o luxo de não precisar de um método do tipo `validarAvatar()`.

Codificando o componente ImageScroller

Vamos agora ao *core* do `ImageScroller`. Mesmo tendo parte da lógica encapsulada no objeto `ManipularEvento.js`, este componente ainda compreende uma certa complexidade que precisamos conhecer para concluirmos nosso entendimento sobre o framework React e seu uso efetivo em aplicações progressivas.

Ao código! Vamos começar pelas importações e pela sintaxe da classe. Crie um arquivo chamado `index.jsx` no diretório `front-end/src/components/ImageScroller` e codifique o trecho a seguir.

```
import React from 'react'
import Image from '../Image'
import ButtonImage from '../ButtonImage'
import ManipularEvento from './ManipularEvento'

class ImageScroller extends React.Component {
}

export default ImageScroller;
```

Quanto às importações, o componente `Image` será usado para

exibir as opções disponíveis do `ImageScroller`. O `ButtonImage` são as pequenas setas laterais para interação por mouse. O `ManipularEvento` é o objeto da lógica que baixamos e o `react` é a essência dos nossos componentes.

O componente `ImageScroller` vai receber cinco parâmetros, a saber:

1. `arquivo`, que representa o nome do arquivo cujas Sprites das imagens serão retiradas. Poderíamos generalizar esse componente para viabilizar Sprites de arquivos de imagens diferentes. Como em nossa aplicação só será necessário resgatar Sprites de um único arquivo, vamos mantê-lo com essa compatibilidade mais restritiva.
2. `eixoY` para simplesmente repassá-lo aos componentes `Image` internos. Só para relembrar, em `Image`, essa propriedade serve para escolher o eixo Y imaginário e provocar um deslocamento vertical no arquivo da imagem para cima.
3. O array `elementos`, que deve ser composto por objetos que contenham uma propriedade `index` e um método `toString()` (como é caso do objeto `Avatar`).
4. `selecionado`, que contém o objeto selecionado. Deve ser um dos itens presentes em `elementos`.
5. O método `onChange()`, que será o *callback* para obtermos o objeto associado à imagem selecionada. Sempre que uma imagem for posicionada no centro do `ImageScroller`, seja pelos eventos de toque ou pelo mouse, esse método será invocado com o respectivo objeto de `elementos` passado por parâmetro.

Esses atributos serão usados em vários trechos do componente. No `constructor`, entretanto, precisaremos apenas das propriedades `elementos` e `selecionado`. Elas serão necessárias para a nossa primeira interação com o objeto `ManipularEvento.js`.

Em termos práticos, esse objeto precisará de duas informações: a quantidade de imagens e o índice da imagem selecionada. No momento da criação de `ManipularEvento.js`, vamos salvá-lo no estado inicial do componente.

```
class ImageScroller extends React.Component {
    constructor(props) {
        super(props)

        this.state = {
            manipularEvento: new ManipularEvento(
                this.props.elementos.length,
                this.props.selecionado.index
            )
        }
    }
}
```

Um comportamento muito comum em `ImageScroller` será a descoberta da imagem selecionada. Para descobri-la, precisaremos obter o atributo `index` do objeto `ManipularEvento.js` e, a partir dele, retornar o respectivo elemento de `this.props.elementos`. Como adendo, a propriedade `manipularEvento.index` retorna o índice do avatar selecionado pelo usuário. Essa lógica ficará encapsulada no método `obterSelecionado()`.

```
obterSelecionado() {
    return this.props.elementos[
        this.state.manipularEvento.index
    ]
```

```
}
```

Agora vamos desbravar os pequenos trechos de JSX que serão necessários para compor o `ImageScroller` que desejamos. Eles serão organizados em métodos, cada um com estilos e código JSX próprios. Veja a seguir uma representação deles.

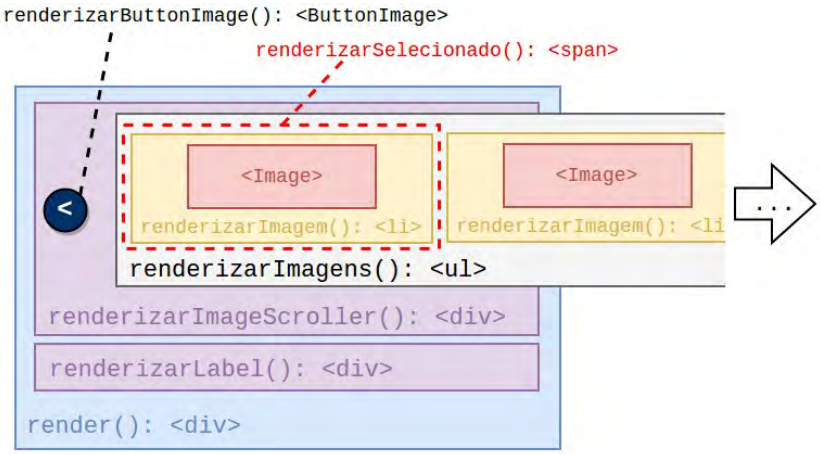

Figura 6.14: Métodos internos do ImageScroller

Vamos olhar todos de perto, a começar pelo `renderizarImagem()`.

Codificando o método renderizarImagem()

Como é possível identificar na imagem anterior, esse método será o responsável por renderizar o `<li>` cujas imagens disponíveis para seleção serão listadas. Ele receberá dois parâmetros: `entry` e `index`. O `index` é o índice ao qual o `entry` (um objeto do array `this.props.elementos`) estará associado.

A primeira lógica do método `renderizarImagem()` é assegurar um valor válido para `this.props.eixoY`. Como vimos, o parâmetro `eixoY` vai ser repassado ao componente `Image` e, no caso especial das imagens de avatares, servirá para exibir imagens do gênero masculino (`eixoY` igual a `0`) ou feminino (`eixoY` igual a `1`).

Por isso, precisamos garantir que, caso ele seja omitido, seu valor padrão seja `0` (zero). Uma simples condição ternária, como no código a seguir, é suficiente.

```
renderizarImagem(entry,index) {
    let eixoY = this.props.eixoY ? this.props.eixoY : 0;
}
```

Agora vamos ao `return` do método, que aqui será apresentado com uma sintaxe alternativa para a definição do estilo. Em vez de declarar uma variável para guardar a estrutura das propriedades, vamos colocar a estrutura diretamente na propriedade `style`. Porém, em vez de envolver a variável com um par de chaves (`{}`), precisaremos de dois pares (`style={{}}`). Isso se faz necessário porque a estrutura também possui seu próprio par de chaves.

Para começarmos, crie um `<li>` com a propriedade `style`, conforme o trecho a seguir:

```
renderizarImagem(entry,index) {
    let eixoY = this.props.eixoY ? this.props.eixoY : 0;
    return (
        <li style={{
        }}>
        </li>
    )
}
```

Quanto às propriedades CSS, há algumas que merecem destaque. A primeira delas é o famigerado `z-index`, cuja sintaxe React é `zIndex`, em *camelCase*. O posicionamento absoluto de itens na viewport na maioria das vezes compreende apenas a manipulação dos eixos X e Y.

Entretanto, existe a possibilidade de distanciar a tag exibida da viewport em um imaginário eixo Z. Isso causa o efeito de sobreposição de elementos, como se algumas tags estivessem sobre as outras. Na prática, essa *pilha* de tags é chamada de *Stack Level* (nível da pilha).

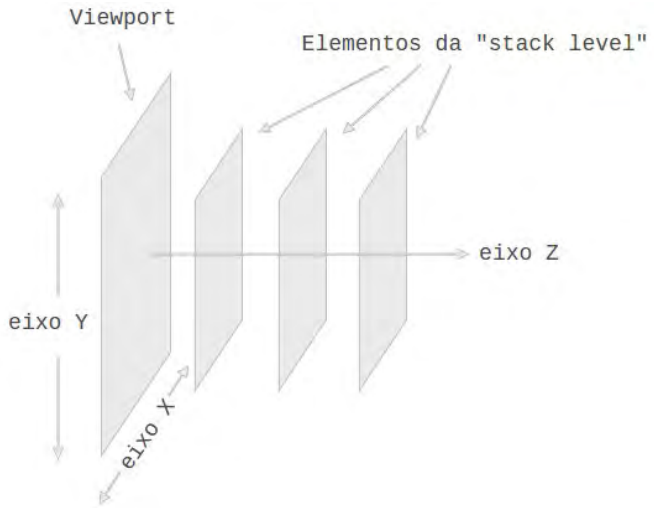

Figura 6.15: Representação da Stack Level

O posicionamento da tag na *Stack Level* é ajustada pela propriedade `zIndex`. Por default, essa propriedade assume o valor `auto`, que na prática significa que ela vai herdar o `zIndex` da sua tag pai. Se todas as tags pais também forem suprimidas, assumindo também o valor `auto`, essa herança vem diretamente

da tag `<html>`, que, por sua vez, possui o `zIndex` igual a `0` (zero).

Se alterarmos o `zIndex` positiva ou negativamente, a tag associada vai ser renderizada acima ou abaixo da viewport, respectivamente. Em outras palavras, se `zIndex` for `-10`, ela ficará abaixo de todos elementos que possuírem a propriedade `zIndex` maior que `-10`. De forma semelhante, se atribuirmos um `zIndex` como `10`, ele ficará sobre outros que tiverem valor inferior.

Mas atenção: a propriedade `zIndex` **só funciona** se estiver acompanhada da propriedade `position` explicitamente definida como `absolute`, `fixed` ou `relative`. O valor default dessa propriedade é `static`. Vamos declará-la explicitamente para evitar problemas na hora de renderizar os elementos.

Voltando ao caso do `<li>`, como a expectativa é que ele fique abaixo (sob o ponto de vista da Stack Level) dos outros componentes, vamos definir seu `zIndex` como `-1`. E para que ele tenha o efeito esperado, vamos também aportar a propriedade `position` com o valor `'absolute'`.

Como a partir de agora as tags `<li>` terão seu posicionamento absoluto, precisaremos criar uma margem diferenciada para cada elemento do array de imagens para apresentá-los um ao lado do outro. Se não fizermos assim, eles ficarão sobrepostos e não teremos o efeito esperado.

Para isso, calcularemos o produto entre `index` e o comprimento das imagens (140 pixels), guardando seu resultado na propriedade `marginLeft`. Veja a seguir como isso será

codificado.

```
<li style={{
    paddingTop: '8px',
    position: 'absolute',
    zIndex: '-1',
    marginLeft: `${index * 140}px`
}}>
</li>
```

Note que acrescentamos também um pequeno distanciamento de `8px` do topo pela propriedade `paddingTop`. Isso afastará as imagens da borda do `ImageScroller`.

Com a `<li>` pronta, resta-nos o componente `Image` interno. Sua invocação será semelhante às realizadas até o momento. A única observação importante aqui é que a propriedade `eixoX` **não** receberá o `index` que identifica a posição no array de imagens. Ele vai receber o atributo `index` do objeto presente no array. Veja o trecho a seguir.

```
<Image
    eixoX={entry.index}
    eixoY={eixoY}
    width={140}
    height={140}
    backgroundHeight={280}
    arquivo={this.props.arquivo}
/>
```

Veja que `eixoY` recebeu a variável local de mesmo nome, definida no início do método. O nome do arquivo foi repassado da propriedade `this.props.arquivo` do componente.

Agora que concluímos a lógica de renderização associada às tags `<li>`, temos espaço para introduzir uma outra característica do código JSX que aqui será primordial para terminarmos esse

método: a propriedade `key`. Se continuássemos com o desenvolvimento dos próximos métodos até a conclusão do componente, ao tentar renderizá-lo no navegador, nos depararíamos com a seguinte mensagem:

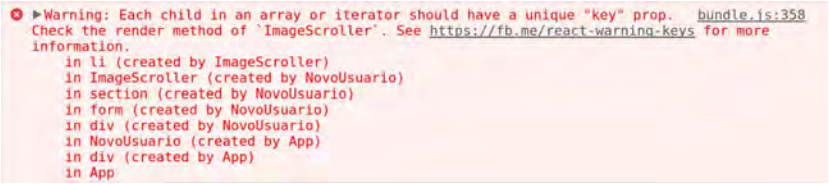

Figura 6.16: Erro da propriedade key

Essa mensagem do React é para nos avisar que estamos renderizando uma série de elementos dinamicamente através de um *loop*, porém eles não possuem uma propriedade `key`. Como vimos no *Capítulo 2*, o React possui um algoritmo para atualizar no DOM do navegador somente os elementos que foram efetivamente alterados no *Virtual DOM*.

Internamente, para auxiliar nessa lógica, o React *marca* os componentes renderizados com uma identificação única e, através dela, consegue perceber se houve ou não modificações passíveis de atualização no DOM real. Quando utilizamos elementos dinâmicos, ele necessita que nós, desenvolvedores, criemos programaticamente uma identificação para cada elemento renderizado. Esse trabalho extra consiste em declarar uma propriedade `key`.

Como `renderizarImagem()` será invocado várias vezes (para cada imagem renderizada), na prática estaremos criando vários `<li>`, fato este que se encaixa na obrigatoriedade do `key`. Então, para que esse *warning* não ocorra, vamos acrescentar um

propriedade `key` na tag `<li>` com uma concatenação do parâmetro `index` e do valor de `entry.toString()`. Isso vai garantir que o valor de `key` seja único. Veja adiante o código final do método `renderizarImagem()`, já com essa propriedade `key` estabelecida.

```
renderizarImagem(entry,index) {
    let eixoY = this.props.eixoY ? this.props.eixoY : 0;
    return (
        <li style={{
            paddingTop: '8px',
            position: 'absolute',
            zIndex: '-1',
            marginLeft: `${index * 140}px`
        }} key={index + entry.toString()}>
            <Image
                eixoX={entry.index}
                eixoY={eixoY}
                width={140}
                height={140}
                backgroundHeight={280}
                arquivo={this.props.arquivo}
            />
        </li>
    )
}
```

Na próxima seção, vamos construir o bloco de iteração para renderizar as tags `<li>` apresentadas aqui.

Codificando o método renderizarImagens()

O objetivo deste método é criar um array de tags `<li>` e renderizá-lo dentro de uma tag `<ul>`. Vamos iniciá-lo nos debruçando sobre o estilo CSS aplicado à `<ul>`.

Quando um usuário efetivar o toque para alguma direção, o objeto `ManipularEvento.js` guardado no estado fará a parte dos

cálculos. Através dele, obteremos o atributo `left` que deve ser atribuído à propriedade CSS de mesmo nome da tag `<ul>`. Como o próprio nome diz, esse atributo indica o quanto o bloco `<ul>` vai se distanciar da margem do `ImageScroller`.

Isso já é suficiente para deslocar as imagens para a esquerda e a direita, conforme direção do toque. O que não ficará agradável é que um pequenino toque pode provocar um deslocamento muito grande, gerando uma sensação de atraso ou defeito no componente. No bom português, seu movimento ficará esquisito.

Há duas possibilidades para resolvermos este problema:

1. Captar a velocidade e direção do movimento e, quando ocorrer o evento `onTouchEnd`, invocar a função nativa do JavaScript `setInterval()` para prolongar o movimento. A função `setInterval()` recebe uma função callback (primeiro parâmetro) que será executada em intervalos regulares de tempo (segundo parâmetro,definido em milissegundos).
2. Usar a propriedade CSS `transition-duration`, que indica quantos segundos (`s`) ou milissegundos (`ms`) um efeito de transição leva para ser concluído.

A segunda opção (`transition-duration`) é mais elegante, porém existe uma pequena questão de compatibilidade que vamos contornar acrescentando prefixos específicos para algumas versões antigas dos browsers. O problema gira em torno das versões 8 e 9 do Internet Explorer e todas as do Opera Mini (versão simplificada do Opera mobile).

Para garantir, vamos cobrir as sintaxes dos navegadores Safari,

Chrome, Internet Explorer, Firefox, Opera, além da sintaxe nativa da W3C. Veja na imagem a seguir o prefixo utilizado para cada navegador.

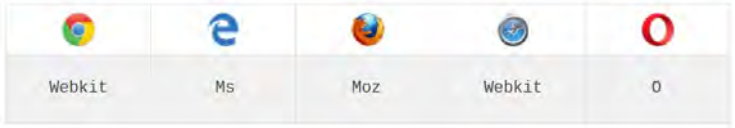

Figura 6.17: Prefixos por navegador

Observe que a imagem apresentou propositalmente os prefixos aderentes à notação *camelCase* do JSX, que na prática serão aplicados à propriedade `transitionDuration`. Entretanto, há uma importante diferença neste caso: veja que, em vez de iniciar com letras minúsculas, esses prefixos são iniciados com maiúsculas.

Em outras palavras, a sintaxe deles não adere ao *camelCase* padrão. Diferente de todas as propriedades nativas do W3C, cuja sintaxe será iniciada com letras minúsculas, as propriedades específicas de navegadores serão iniciadas com maiúsculas. Por exemplo, a propriedade `transitionDuration` específica para os navegadores Chrome e Safari é `WebkitTransitionDuration`, com o `W` maiúsculo.

Resumindo: se a propriedade for específica de um navegador, inicie-a com letras maiúsculas. Do contrário, utilize a sintaxe *camelCase* padrão.

Agora que já conversamos sobre a solução via `transitionDuration`, vamos tornar seu efeito *mais agradável*. Enquanto o evento `onTouchMove` estiver ocorrendo, podemos

aplicar meros 100 milissegundos de atraso na transação. Se o evento `onTouchEnd` for detectado, vamos aumentar o atraso para 800 milissegundos. Este sutil ajuste tornará a movimentação do toque mais atraente, deixando o delay após o término do toque um pouco mais lento.

Para tornar isso possível, precisamos resgatar o valor do atributo `toqueEmExecucao` do objeto `ManipularEvento.js`. Esse atributo assume o valor `true` caso o método `manipularEvento.iniciar()` seja executado, e `false` após a chamada do método `manipularEvento.atualizarToque()`.

Como o próprio nome sugere, o método `iniciar()` serve para avisarmos ao `ManipularEvento.js` que um evento de toque (que é contínuo enquanto o usuário estiver com o dedo na tela) foi iniciado. O método `atualizarToque()` serve para tratar os eventos `onTouchEnd` e `onClick`, que servem respectivamente para indicar o fim do toque (ou do evento de clique).

Certo, vejamos então como ficará a condição para definição do atraso e a definição de todas as propriedades `transitionDuration`, no trecho adiante.

```
renderizarImagens() {
    const ms = this.state.manipularEvento.toqueEmExecucao
        ? '100ms' : '800ms'

    const estilo = {
        WebkitTransitionDuration: ms, /* Safari e Chrome */
        MsTransitionDuration: ms, /* IE */
        MozTransitionDuration: ms, /* Firefox */
        OTransitionDuration: ms, /* Opera */
        transitionDuration: ms, /* Nativa do W3C */
    }
}
```

Com essas propriedades CSS definidas, vamos agora concluir a definição da constante `estilo` acrescentando as propriedades restantes. Por default, as tags `<li>` são decoradas com *bullets* (bolinhas) que prejudicam o layout e o posicionamento no nosso caso. Para omiti-las, vamos incluir a propriedade `listStyleType` com o valor `none`.

Para proporcionar o efeito do deslocamento via `left`, vamos acrescentar também a propriedade `position: 'relative'`. Por fim, vamos limpar qualquer vestígio de espaçamento entre os elementos internos e externos anulando as propriedades `margin` e `padding`.

```
const estilo = {
    WebkitTransitionDuration: ms, /* Safari e Chrome */
    MsTransitionDuration: ms, /* IE */
    MozTransitionDuration: ms, /* Firefox */
    OTransitionDuration: ms, /* Opera */
    transitionDuration: ms, /* Nativa do W3C */

    listStyleType: 'none',
    margin: '0',
    padding: '0',
    position: 'relative'
}
```

Restam-nos as duas propriedades CSS mais importantes da `<ul>`:

- `width`, para definir o tamanho total da estrutura que vai encapsular as imagens. Este campo será definido dinamicamente pelo produto entre a quantidade de elementos e o tamanho default das imagens (140 pixels).
- `left`, que vai efetivamente deslocá-lo para a esquerda e a direta de acordo com o atributo `left` do objeto `this.state.manipularEvento`, guardado no estado do

componente.

Veja o resultado final da constante `estilo`:

```
const estilo = {
    WebkitTransitionDuration: ms, /* Safari e Chrome */
    MsTransitionDuration: ms, /* IE */
    MozTransitionDuration: ms, /* Firefox */
    OTransitionDuration: ms, /* Opera */
    transitionDuration: ms, /* Nativa do W3C */

    listStyleType: 'none',
    margin: '0',
    padding: '0',
    position: 'relative',
    width: `${this.props.elementos.length * 140}px`,
    left: `${this.state.manipularEvento.left}px`
}
```

O próximo passo é criar uma lista de tags `<li>`, e faremos isso usando as boas práticas do React e do EcmaScript 6. A lista será dinamicamente gerada pelo uso do método `map()`. Este está disponível nos arrays e serve para percorrer todos os elementos e, se for necessário, modificá-los. Ele recebe como parâmetro uma *callback* que retorna o novo valor a ser atribuído ao elemento atual da repetição (primeiro argumento) e o índice da iteração atual (segundo argumento).

E é com a facilidade que o `map()` nos oferece que vamos invocar o método `renderizarImagem()` para cada elemento presente no array `this.props.elementos`. Após todas as iterações, vamos guardar o resultado na constante `lista`. Veja o código.

```
const lista = this.props.elementos.map(
    (entry,index) => this.renderizarImagem(entry,index)
);
```

Com a lista de tags `<li>` pronta, resta-nos envolvê-la em uma `<ul>` estilizada com a estrutura `estilo` no `return` do método.

```
return (
    <ul style={estilo}>
        {lista}
    </ul>
)
```

Assim concluímos o método `renderizarImagens()`. Veja seu código na íntegra adiante.

```
renderizarImagens() {
    const ms = this.state.manipularEvento.toqueEmExecucao
        ? '100ms' : '800ms'

    const estilo = {
        WebkitTransitionDuration: ms, /* Safari e Chrome */
        MsTransitionDuration: ms, /* IE */
        MozTransitionDuration: ms, /* Firefox */
        OTransitionDuration: ms, /* Opera */
        transitionDuration: ms, /* Nativa do W3C */

        listStyleType: 'none',
        margin: '0',
        padding: '0',
        position: 'relative',
        width: `${this.props.elementos.length * 140}px`,
        left: `${this.state.manipularEvento.left}px`
    }

    const lista = this.props.elementos.map(
        (entry,index) => this.renderizarImagem(entry,index)
    );

    return (
        <ul style={estilo}>
            {lista}
        </ul>
    )
}
```

Codificando o método renderizarSelecionado()

Este método vai apenas renderizar uma tag `<span>` para servir como referencial da imagem que está selecionada. Ele ajuda muito na usabilidade do componente. Veja adiante sua exposição na versão final do componente `ImageScroller`.

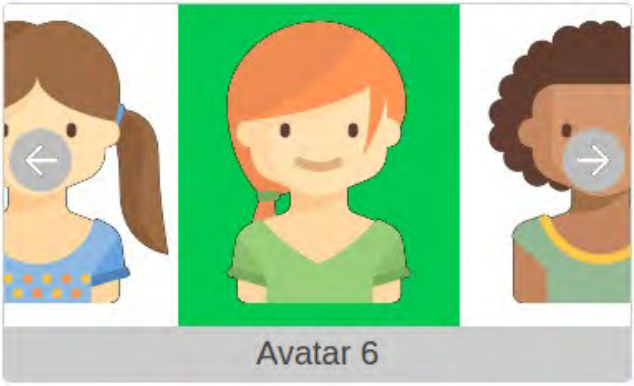

Figura 6.18: Imagem selecionada destacada

A imagem selecionada está destacada no centro do componente. Além das estilizações de posicionamento que não merecem destaque, teremos um fundo na cor verde (`#00C853`), a propriedade `position` definida como `relative` , e o `zIndex` definido com `-2` . Veja o código completo do método a seguir.

```
renderizarSelecionado() {
    return (
        <span
            style={{
                float: 'left',
                width: '140px',
                height: '160px',
                marginLeft: '42px',
                backgroundColor: '#00C853',
                position: 'relative',
```

```
                zIndex: -2
            }}
        ></span>
    )
}
```

O número dois negativo não foi arbitrário. Ele precisa assumir um valor que seja menor que o `zIndex` das tags `<li>` (que assumiram o valor `-1`). Isso provocará o efeito exposto na imagem: o `<span>` será exibido atrás da imagem centralizada.

Codificando o método renderizarButtonImage()

O próximo método será o `renderizarButtonImage()`. Seu único objetivo é renderizar o componente `ButtonImage` e repassar corretamente as propriedades para seu posicionamento e tratamento dos eventos.

Como vimos na criação do componente `ButtonImage`, ele deve receber a propriedade `posicao` definida como `'direita'` ou `'esquerda'`. Este valor pode inclusive ser informado como um parâmetro do método `renderizarButtonImage()`, com intento de torná-lo mais flexível. Veja o trecho de código a seguir.

```
renderizarButtonImage(posicao) {
    return (
        <ButtonImage
            posicao={posicao}
        />
    )
}
```

Conversaremos a partir de agora sobre o tratamento dos eventos (`onTouchStart`, `onTouchMove`, `onTouchEnd` e `onClick`) que serão enviados ao `ButtonImage` por meio do *spread operator*.

Para isso, todavia, é importante que você compreenda alguns conceitos sobre o *SyntheticEvent*, um *wrapper* (invólucro) genérico que encapsula todos os eventos W3C para o React. Internamente, o *SyntheticEvent* contém propriedades e métodos comuns a todos os tipos de eventos, sendo especializado com propriedades específicas quando ele é disparado pelas tags React.

Na prática, o *SyntheticEvent* possui muitas propriedades comuns aos eventos nativos da W3C e algumas especializadas do framework. Não vamos desbravá-lo na íntegra aqui, mas, caso queira, você pode obter mais detalhes em sua documentação oficial: https://facebook.github.io/react/docs/events.html#form-events.

Com o *SyntheticEvent* devidamente apresentado, precisamos investigar as propriedades `bubbles` e `eventPhase` e o método `stopPropagation()`.

O `bubbles` é um booleano que serve para indicar se o evento faz o *bubbles* pelo DOM. Em poucas palavras, os eventos do DOM, quando disparados, primeiramente fazem um trajeto da tag mais alta da hierarquia (`<html>`) até a tag cujo evento foi originado (um `<button>` , por exemplo). Essa fase é conhecida como *Capturing Phase* (fase de captura).

Chegando à tag principal, além da *Target Phase* (fase do alvo), que indica que o evento atingiu a tag que o disparou, alguns tipos de eventos são capazes de progredir de baixo para cima na hierarquia, através da fase conhecida como *Bubbling Phase*, cuja tradução não significativa é *fase borbulhante*.

A critério de exemplo, supondo as tags `<html>` (item 1 da

imagem a seguir), `<body>` (item 2) e `<button>` (item 3), a fase de *bubbles* percorrerá a sequência do botão até a tag mais alta. Então, resumindo, a propriedade `bubbles` será `true` se o evento disparado possuir a *Bubbling Phase*; caso contrário, `false` .

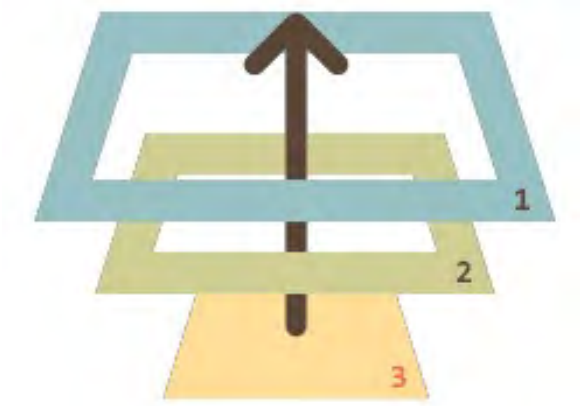

Figura 6.19: Efeito bubbles dos eventos

O `eventPhase` é um número inteiro que assume seu valor de acordo com as fases de propagação. Os valores possíveis são:

- `0` (zero), quando não há evento sendo processado;
- `1` , que indica a *Capturing Phase* (do elemento de maior hierarquia para baixo);
- `2` , quando o evento alcança o elemento que o disparou (*Target Phase*, ou fase do alvo);
- `3` , para indicar a *Bubbling Phase* (do elemento que disparou o evento para o de mais alta hierarquia).

Na prática, as fases *Capturing Phase* e *Target Phase* não são muito influentes no comportamento esperado dos eventos em uma página. Por outro lado, o que é muito comum acontecer é

explicitamente interromper o fluxo da *Bubbling Phase* de forma a evitar que o evento seja propagado para os componentes mais altos da hierarquia.

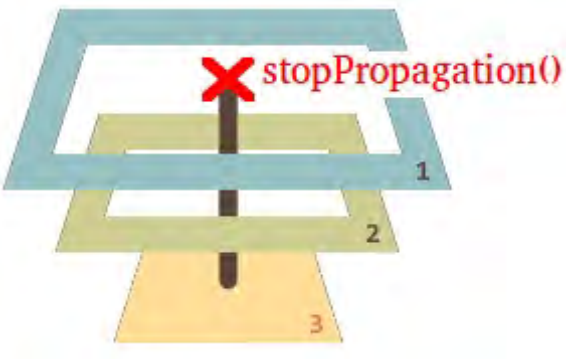

Figura 6.20: Interrompendo a Bubbling Phase

Assim como os eventos W3C, o *SyntheticEvent* também possui o método `stopPropagation()`, cujo objetivo é interromper a propagação natural do evento nas fases *Capturing Phase* e *Bubbling Phase*. Em outras palavras, se estivermos na fase de *bubbles*, um evento `onClick` poderia ser capturado em diferentes tags ao longo da hierarquia. Ao convocar `stopPropagation()`, esse fluxo é interrompido.

Mas para que precisamos interromper a propagação dos eventos? Para alcançarmos o entendimento, vou usar como base o evento `onTouchStart`, que ocorre quando o usuário inicia um toque na tela do dispositivo mobile. O componente `ButtonImage` nada mais é do que uma tag `<div>` bastante sofisticada. E, como sabemos, `ButtonImage` será um componente interno do `ImageScroller` (ou seja, `ImageScroller` está hierarquicamente mais alto do que `ButtonImage`).

Quando um usuário mobile estiver selecionando uma imagem, a expectativa é de que o evento `onTouchStart` seja disparado da `<div>` externa do `ImageScroller`. Por outro lado, se o usuário mobile quiser usufruir dos botões laterais, ao tocar neles, devemos explicitamente **evitar** que um efeito de deslocamento também seja aplicado pela `<div>` externa. Em outras palavras, se `ButtonImage` for clicado, ele deve ser o único responsável pela seleção das imagens.

Para conquistarmos esse comportamento, precisamos capturar o evento `onTouchStart` e, a partir dele, interromper a propagação para a `<div>` externa (e qualquer outra tag de hierarquia superior). Esse tipo de lógica é possível com a simples invocação do método `stopPropagation()`, conforme o exemplo a seguir.

```
onTouchStart={e => e.stopPropagation()}
```

Essa mesma lógica deverá ser aplicada aos eventos `onTouchMove` e `onTouchEnd`, o que nos proporcionará um código semelhante ao apresentado adiante.

```
renderizarButtonImage(posicao) {
    return (
        <ButtonImage
            posicao={posicao}

            onTouchStart={e => e.stopPropagation()}
            onTouchMove={e => e.stopPropagation()}
            onTouchEnd={e => e.stopPropagation()}
        />
    )
}
```

Como agora todos os eventos de toque não serão propagados, precisamos implementar explicitamente o evento `onClick` para

viabilizar o deslocamento das imagens aos usuários desktop.

Em primeiro plano, `onClick` deve evitar o comportamento default pelo método `preventDefault()`. Em seguida, após recuperar o objeto `manipularEvento` do estado do componente, ele poderá decrementar ou acrescentar um atributo `index` respectivamente se o valor do parâmetro `posicao` for `'esquerda'` ou `'direita'`. Só para reforçar: um clique na seta da esquerda vai exibir a próxima imagem à esquerda da posição atual – o que na prática significa movimentar a `<ul>` para a direita.

O próximo passo é chamar os métodos `manipularEvento.definirIndex()` (repassando o valor de `index`) e `manipularEvento.atualizarClique()` (para concretizar o novo índice). Com `manipularEvento` ajustado, vamos atualizar o estado do componente `ImageScroller` com `this.setState()` e, somente após a atualização ser concluída, executar `this.props.onChange` repassando o objeto que está selecionado pela chamada de `this.obterSelecionado()`.

Veja a seguir o tratamento completo do evento `onClick`, já exposto com todo o código do método `renderizarButtonImage()`.

```
renderizarButtonImage(posicao) {
    return (
        <ButtonImage
            posicao={posicao}

            onTouchStart={e => e.stopPropagation()}
            onTouchMove={e => e.stopPropagation()}
            onTouchEnd={e => e.stopPropagation()}

            onClick={e => {
```

```
                e.preventDefault();
                let manipularEvento = this.state.manipularEvento;
                let index = manipularEvento.index;
                if (posicao == 'esquerda') {
                    index += -1;
                } else {
                    index += 1;
                }
                manipularEvento.definirIndex(index);
                manipularEvento.atualizarClique();

                this.setState({ manipularEvento: manipularEvento
},() => {
                    this.props.onChange(this.obterSelecionado());
                });
            }}
        />
    )
}
```

Veja que utilizamos a sintaxe de `this.setState()`, que, além de receber o valor do estado, executa uma *callback* após a efetiva atualização do estado do componente `ImageScroller`. Lembre-se de que `this.setState()` é assíncrono. O uso da *callback* para garantir o momento correto da chamada de `onChange` se faz necessário.

Com isso, concluímos o método `renderizarButtonImage()`. Vamos agora avançar para o método mais importante: `renderizarImageScroller()`.

Codificando o método renderizarImageScroller()

Este será o principal método do componente. Ele será o ponto de invocação de quase todos os métodos criados, com exceção apenas do `renderizarLabel()`. Além disso, ele também renderizará a tag `<div>` responsável por capturar os efetivos

eventos de toque.

Vamos começar pelo mais simples: o estilo CSS. Neste método, teremos um estilo inline definido através de uma constante chamada `estilo`. A primeira propriedade já é nossa conhecida `boxSizing`, cujo valor `'border-box'` integra `padding` e `border` ao tamanho.

Até então, as definições de bordas criadas foram concebidas pela propriedade `border`, que recebe em uma única string a espessura, o tipo (estilo) e a cor da borda. Neste componente, a aparência ideal vai nos exigir uma configuração mais refinada pelas propriedades:

- `borderWidth` para a espessura (geralmente em pixels);
- `borderBottomWidth` para a espessura específica da parte inferior da `<div>`;
- `borderStyle` para o estilo (`solid`, `dotted`, `dashed`, `double`, entre outros);
- `borderColor` para a cor da borda;
- `borderRadius` para a definição do arredondamento dos cantos da `<div>` de maneira uniforme;
- `borderBottomLeftRadius` e `borderBottomRightRadius` para definição do arredondamento específico dos cantos inferiores esquerdo e direito, respectivamente.

A maior alteração está na necessidade de utilizar propriedades tão pontuais como `borderBottomWidth`, `borderBottomLeftRadius` e `borderBottomRightRadius`. Elas serão úteis porque a `<div>` do `ImageScroller` em si será acoplada à `<div>` da descrição da imagem, que ficará exposta na

parte inferior.

A ideia dessas propriedades é omitir a borda e o arredondamento da parte inferior da `ImageScroller`, de forma a deixá-la *sem adereços*, por assim dizer. A mesma lógica será aplicada à `<div>` da descrição, porém nas propriedades da borda superior.

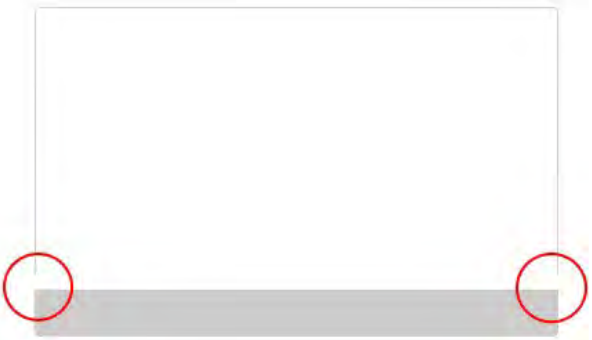

Figura 6.21: Exemplo do encaixe dos elementos

Outra particularidade dessa estilização é que, em vez de a `<div>` ser elástica de acordo com seu conteúdo, vamos definir tamanhos fixos para as propriedades `width` e `height`. Os valores adotados – que são `310px` para comprimento e `160px` para altura – coincidem perfeitamente com o tamanho da PWA na viewport.

A última propriedade necessária será o `overflow`, cuja tradução literal é *transbordar*. Quando os componentes internos de uma `<div>` são maiores que a própria `<div>`, o CSS vai ativar barras de rolagens verticais e horizontais. Esta propriedade nos ajuda a controlar esse comportamento default.

No nosso caso específico, devemos omitir as barras e permitir que o valor interno *transborde* o tamanho da `<div>` externa. Isso é possível com o valor `hidden`.

Com todas as propriedades detalhadas, podemos agora visualizar a estrutura `estilo` na íntegra.

```
const estilo = {
    boxSizing: 'border-box',
    borderWidth: '1px',
    borderBottomWidth: '0',
    borderStyle: 'solid',
    borderColor: '#cccccc',
    borderRadius: '5px',
    borderBottomLeftRadius: '0',
    borderBottomRightRadius: '0',
    width: '310px',
    height: '160px',
    overflow: 'hidden'
};
```

O outro trecho de código desse método simplesmente retornará a tag `<div>`. Como vimos, ela será responsável pela captura dos eventos `onTouchStart`, `onTouchMove` e `onTouchEnd`. Vamos apresentar o código dos métodos efetivos em breve. Na renderização da `<div>` em si, só vamos convocá-los através da *bound function* `bind()` repassando o componente `ImageScroller` (referenciado por `this`).

Além disso, dentro da `<div>`, vamos convocar todos os métodos criados até o momento, na seguinte ordem:

1. `renderizarButtonImage('esquerda')`, para exibir o botão da esquerda;
2. `renderizarSelecionado()`, para exibir a `<span>` de objeto selecionado;

3. `renderizarImagens();`
4. `renderizarButtonImage('direita')`, para o botão da direita;

Como os componentes possuem posições relativas, a sequência de chamadas serve mais para manter uma coerência lógica dos trechos JSX internos.

Desta forma, concluímos o método `renderizarImageScroller()`. Veja-o na íntegra a seguir.

```
renderizarImageScroller() {
    const estilo = {
        boxSizing: 'border-box',
        borderWidth: '1px',
        borderBottomWidth: '0',
        borderStyle: 'solid',
        borderColor: '#cccccc',
        borderRadius: '5px',
        borderBottomLeftRadius: '0',
        borderBottomRightRadius: '0',
        width: '310px',
        height: '160px',
        overflow: 'hidden'
    };

    return (
        <div
            style={estilo}
            onTouchStart={this.onTouchStart.bind(this)}
            onTouchMove={this.onTouchMove.bind(this)}
            onTouchEnd={this.onTouchEnd.bind(this)}
        >
            {this.renderizarButtonImage('esquerda')}
            {this.renderizarSelecionado()}
            {this.renderizarImagens()}
            {this.renderizarButtonImage('direita')}
        </div>
    )
}
```

Para que ele realmente tenha o comportamento esperado, vamos criar os métodos de tratamento dos eventos, a começar pelo `onTouchStart()`. Como toda a lógica do deslocamento da `<ul>` já está sendo tratada pelo objeto `ManipularEvento.js`, esses métodos serão bem simples.

Uma informação básica que você precisa ter em mente é que, quando trabalhamos com eventos de toque, múltiplos eventos podem ocorrer ao mesmo tempo (vários toques de uma vez). Para guardar propriedades específicas de cada um deles, a equipe da especificação original W3C organizou-os em um objeto chamado `TouchList`.

Na prática, vamos obtê-lo de um evento de toque qualquer a partir da sintaxe `e.targetTouches`. Como `TouchList` é um array, ele guardará em seus índices os toques na superfície iniciados no elemento alvo do evento `e`. Cada índice terá um objeto com várias propriedades, mas, para nós, somente a `clientX` será necessária.

As propriedades `clientX` e `clientY` servem para guardar as coordenadas X e Y do ponto de toque relativos à viewport, sem observar o deslocamento da tela pelo scroll (*scroll offset*). Como nosso componente só precisa capturar o movimento horizontal, não vamos precisar da propriedade `clientY`.

Regressando ao código, obteremos a propriedade `clientX` do primeiro toque (de índice `0`) a partir do evento `e`. De posse deste dado, vamos obter o `ManipularEvento.js` do estado do componente, convocar seu método `iniciar()` repassando `clientX` como parâmetro e, por fim, atualizar o estado do `ImageScroller` através de `this.setState()`. Apenas quatro

linhas de código JavaScript serão necessárias para concluirmos esse primeiro método.

```
onTouchStart(e) {
    let clientX = e.targetTouches[0].clientX;
    let manipularEvento = this.state.manipularEvento;
    manipularEvento.iniciar(clientX);
    this.setState({ manipularEvento: manipularEvento });
}
```

Na sequência, os métodos `onTouchMove()` e `onTouchEnd()` seguem a mesma base de `onTouchStart()`, diferenciando-se apenas pelos outros métodos que serão invocados de `ManipularEvento.js` e, no caso especial de `onTouchEnd()`, na convocação do método `onChange()` para identificar a imagem selecionada. Veja seus códigos a seguir.

```
onTouchMove(e) {
    let clientX = e.targetTouches[0].clientX;
    let manipularEvento = this.state.manipularEvento;
    manipularEvento.mover(clientX);
    this.setState({ manipularEvento: manipularEvento });
}
onTouchEnd(e) {
    let manipularEvento = this.state.manipularEvento;
    manipularEvento.atualizarToque();
    this.setState({ manipularEvento: manipularEvento },() => {
        this.props.onChange(this.obterSelecionado());
    });
}
```

Perceba que o tratamento `onTouchMove()` convoca o método `manipularEvento.mover()`, e `onTouchEnd()`, o método `manipularEvento.atualizarToque()`. Este último provocará o ajuste do atributo `this.state.manipularEvento.left` utilizado no método `renderizarImagens()` e, consequentemente, deslocará a `<ul>` das imagens.

Concluímos assim os métodos `renderizarImageScroller()`, `onTouchStart()`, `onTouchMove()` e `onTouchEnd()`. Vamos agora ao `renderizarLabel()`, responsável por renderizar a descrição da imagem selecionada.

Codificando o método renderizarLabel()

Não precisamos discorrer detalhes profundos sobre este código. Ele basicamente cria propriedades CSS para cores e tamanho da fonte, da cor de fundo, e da estilização apropriada na parte superior da `<div>` (para que ela se encaixe perfeitamente na `<div>` da `ImageScroller`).

```
renderizarLabel() {
    const estilo = {
        boxSizing: 'border-box',
        borderWidth: '1px',
        borderStyle: 'solid',
        borderTopWidth: '0',
        borderColor: '#cccccc',
        borderRadius: '5px',
        borderTopLeftRadius: '0',
        borderTopRightRadius: '0',
        backgroundColor: '#cccccc',
        color: '#444444',
        fontSize: '20px',
        textAlign: 'center',
        padding: '5px',
        width: '380px'
    };

    return (
        <div style={estilo}>
            {this.obterSelecionado().toString()}
        </div>
    )
}
```

Outro detalhe que merece destaque é o uso do método `toString()` do objeto selecionado. Esse método que está codificado dentro dos avatares apenas retornará o nome das imagens.

Concluindo o componente ImageScroller

Por fim, chegamos ao método `render()`. Ele será um mero centralizador da invocação dos métodos `renderizarImageScroller()` e `renderizarLabel()`. Veja o código.

```
render() {
    return (
        <div>
            {this.renderizarImageScroller()}
            {this.renderizarLabel()}
        </div>
    )
}
```

Este foi o último recurso de codificação do componente `ImageScroller`, que, por sua vez, é o último componente do livro. Durante a construção dos seus métodos, você foi submetido ao que temos de mais rebuscado na codificação do framework React.

Como o código é extenso, não vou reproduzi-lo na íntegra aqui. Caso queira conferir com o seu, baixe o arquivo completo em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/components/ImageScroller/index.jsx.

Na próxima seção, vamos ajustar o componente `NovoUsuario` para tratar os eventos `onChange` e efetivamente renderizar o `ImageScroller`.

6.5 AJUSTANDO OS COMPONENTES APP E NOVOUSUARIO

Concluiremos este capítulo com os pequenos ajustes no componente `NovoUsuario` e na visualização do resultado final da segunda tela de cadastro de usuários.

Ajustes periféricos

Abra o arquivo `index.jsx` da pasta `front-end/src/components/NovoUsuario` e comece com a inclusão da importação do `ImageScroller`, conforme o trecho destacado a seguir.

```
import React from 'react'
import Label from '../Label'
import Input from '../Input'
import GenderSelector from '../GenderSelector'
import Usuario from '../../models/Usuario'
import Avatar from '../../models/Avatar'
import Button from '../Button'
import Toast from '../Toast'
import ImageScroller from '../ImageScroller'

class NovoUsuario extends React.Component {
    // ...
}
```

Se os avatares são diretamente influenciados pelo gênero, é justo sempre *limpar* o índice do avatar selecionado para a primeira opção disponível. Isso será realizado ajustando o método `atualizarGenero()` para redefinir o atributo `avatar` para seu valor inicial, que é `Avatar.obterTodos()[0]`.

```
atualizarGenero(e,genero) {
    e.preventDefault();
    let usuario = this.state.usuario;
```

```
    usuario.genero = genero;
    usuario.avatar = Avatar.obterTodos()[0];
    this.setState({
        usuario: usuario
    });
}
```

Até o momento, definimos o JSX de renderização dos botões *Salvar* e *Voltar*, sendo que, para este último, temos um breve *arrow function* atualizando o valor de `primeiraVisaoCompleta` para `false`. De forma semelhante à que fizemos no método `atualizarGenero()`, vamos zerar o conteúdo de `usuario.avatar` sempre que o usuário regressar para a primeira tela. Isso pode evitar problemas de renderização caso o usuário fique clicando nos botões *Próximo* e *Voltar* várias vezes antes de concluir o cadastro.

Para aplicar essa alteração, modifique a renderização do componente `Button` presente no método `renderizarBotoes()`, conforme o trecho de código adiante.

```
<Button
    texto="Voltar"
    onClick={e => {
        e.preventDefault();
        let usuario = this.state.usuario
        usuario.avatar = Avatar.obterTodos()[0];
        this.setState({
            usuario: usuario,
            primeiraVisaoCompleta: false
        });
    }}
/>
```

Ajustando o botão Salvar

É sabido que o componente `NovoUsuario` é responsável pela captura dos dados e pelo preenchimento do objeto `usuario`.

Entretanto, ele será completamente indiferente à persistência desses dados no cache do navegador. Independente da ciência envolvida nesta *persistência*, que será tratada no próximo capítulo, já podemos estabelecer que `NovoUsuario` delegará essa responsabilidade para o componente `App`.

Na prática, essa delegação se dará por meio de uma função *callback* repassada à `NovoUsuario`, função esta que será invocada quando o usuário clicar no botão *Salvar*. Para manter certa proximidade com os termos comumente usados em aplicações web, vamos nomear essa função de `onSubmit`.

Abra o arquivo `NovoUsuario/index.jsx`. A partir de agora, o botão *Salvar* repassará o objeto `usuario` guardado no estado do componente ao método `this.props.onSubmit()`. Note que `preventDefault()` também será necessário para evitar o comportamento natural do botão. Altere o trecho de *Salvar* localizado no método `renderizarBotoes()` para que ele fique igual ao código a seguir.

```
<Button
    principal
    texto="Salvar"
    onClick={e => {
        e.preventDefault()
        this.props.onSubmit(this.state.usuario)
    }}
/>
```

Resta-nos agora alterar o componente `App` para repassar o `onSubmit`. Abra o arquivo `front-end/src/components/App.jsx` e altere seu método `render()` conforme o trecho de código a seguir. Por enquanto, sua *arrow function* apenas exibirá um *toast* de sucesso com uma mensagem

personalizada com o gênero e o nome do usuário.

```
render() {
    return (
        <div>
            <Header />
            <NovoUsuario
                onSubmit={usuario => {
                    let genero = usuario.genero == 'm' ? 'o' : 'a
'
                    this.refs.toast.sucesso(
                        `Seja bem-vind${genero} ${usuario.nome}!`
                    )
                }}
                erro={msg=>this.refs.toast.erro(msg)}
            />
            <Toast ref="toast" />
        </div>
    );
}
```

As alterações básicas já estão prontas. Nos próximos tópicos, concentraremos nossos esforços ao componente `NovoUsuario`, mais especificamente na invocação do componente `ImageScroller` e sua renderização no método `render()`.

Criando o método renderizarAvatar()

A primeira coisa que vamos estabelecer é que, de forma oposta à condição do método `renderizarGenero()`, o método `renderizarAvatar()` só vai renderizar os componentes `Label` e `ImageScroller` se o valor do estado `primeiraVisaoCompleta` for verdadeiro. Do contrário, o método retornará `null`.

```
renderizarAvatar() {
    if (this.state.primeiraVisaoCompleta) {
        // ... demais códigos
    } else {
        return null
```

```
    }
}
```

A `Label` será muito simples, contendo apenas o valor `"Escolha seu avatar:"` no atributo `texto`. Como não há possibilidade de escolher um valor inválido de avatar, não será necessário definir o atributo `valorInvalido`. Assim como fizemos com os outros métodos de `NovoUsuario`, vamos circundar os valores retornados em uma tag `<section>`.

```
return (
    <section>
        <Label
            texto="Escolha seu avatar:"
        />
    </section>
)
```

Agora, abaixo da `Label`, vamos acrescentar a tag do componente `ImageScroller`. A propriedade `arquivo` deve receber a imagem a partir da qual os avatares serão retirados, ou seja, a string `"img/avatars.png"`. A propriedade `eixoY` deve receber o valor `0` (zero) se o usuário for do gênero masculino, ou `1` se for feminino. Isso será implementado com uma condição ternária básica.

A propriedade `elementos` deve receber a lista de objetos relacionados aos Sprites exibidos, que, neste caso, será obtida diretamente do método estático `obterTodos()` da classe `Avatar`. A propriedade `selecionado` será definida com `this.state.usuario.avatar`, que recupera o *avatar* do usuário.

Resta-nos a propriedade `onChange`, que deve repassar a função *callback* a ser executada sempre que uma nova imagem for selecionada. A única coisa com que precisamos nos preocupar aqui

é salvar o *avatar* passado por parâmetro na *callback* e guardá-lo no estado do componente `NovoUsuario`. Por ser um código simples, vamos mantê-lo encapsulado em uma *arrow function*.

Veja a seguir o código completo do método `renderizarAvatar()`.

```
renderizarAvatar() {
    if (this.state.primeiraVisaoCompleta) {
        return (
            <section>
                <Label
                    texto="Escolha seu avatar:"
                />
                <ImageScroller
                    arquivo="img/avatars.png"
                    eixoY={(this.state.usuario.genero == 'm' ? 0
 1)}
                    elementos={Avatar.obterTodos()}
                    selecionado={this.state.usuario.avatar}
                    onChange={avatar => {

                        let usuario = this.state.usuario;
                        usuario.avatar = avatar;
                        this.setState({

                            usuario: usuario

                        });
                    }}
                />
            </section>
        )
    } else {
        return null
    }
}
```

Método pronto!

Ajustando o método render()

Por fim mas não menos importante, vamos acrescentar a invocação do método `renderizarAvatar()` no JSX do método `render()`, conforme código a seguir.

```
render() {
    return (
        <div className="center">
            <form className="pure-form pure-form-stacked">
                {this.renderizarNome()}
                {this.renderizarGenero()}
                {this.renderizarAvatar()}
                {this.renderizarBotoes()}
            </form>
        </div>
    );
}
```

Assim, finalmente concluímos toda a lógica React da tela de cadastro de usuários. Caso você queria ver o código do arquivo `NovoUsuario/index.jsx` na íntegra, acesse o repositório do projeto em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/components/NovoUsuario/index.jsx.

Salve todas as alterações e vamos checar como a PWA está se comportando até agora.

Acessando o componente NovoUsuario

Já temos condições de utilizar plenamente a tela de cadastro de usuários. Para tornar os testes mais fidedignos, em vez de simplesmente executarmos o servidor de desenvolvimento pelo comando `webpack-dev-server`, vamos publicar a aplicação (ainda como desenvolvimento) para podermos acessá-la através de um celular.

Isso é possível porque `webpack-dev-server` possui uma diretiva `--host` pela qual podemos definir o IP do nosso computador. Abra o console (ou prompt), acesse o diretório `behappywith.me/front-end/` do projeto e execute o comando a seguir, alterando o endereço IP *192.168.0.18* para o seu caso.

```
NODE_ENV=development node_modules/webpack-dev-server/bin/webpack-
dev-server.js --host 192.168.0.18
```

Agora, através de qualquer dispositivo conectado à sua rede, você poderá acessar nossa PWA pelo endereço http://192.168.0.18:8080/.

Veja a seguir os prints reais realizados em um dispositivo Android. As três primeiras imagens são relacionadas à primeira visão do cadastro, respectivamente, com ambos campos em branco, com o *nome* e o *gênero* inválidos e, finalmente, com *nome* igual à *Guilherme* e gênero inválido.

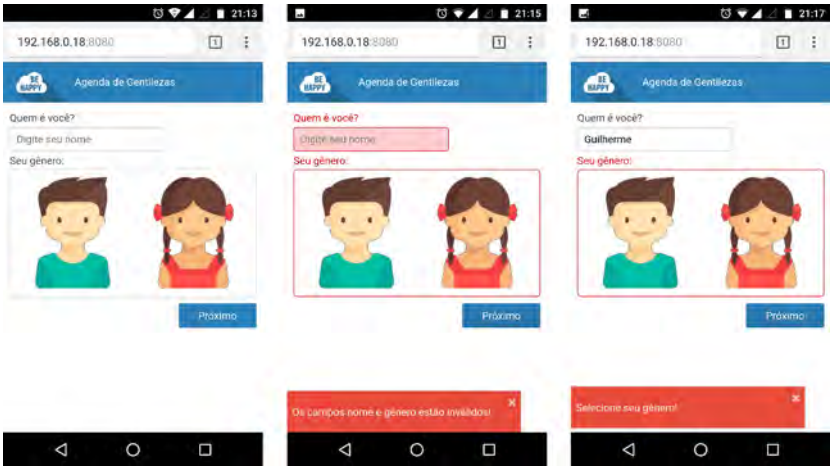

Figura 6.22: Screenshots do celular

Após o preenchimento correto dos nomes, ao clicarmos no botão *Próximo*, poderemos selecionar o avatar e, se clicarmos no botão *Salvar*, teremos a mensagem *Seja bem-vindo Guilherme*.

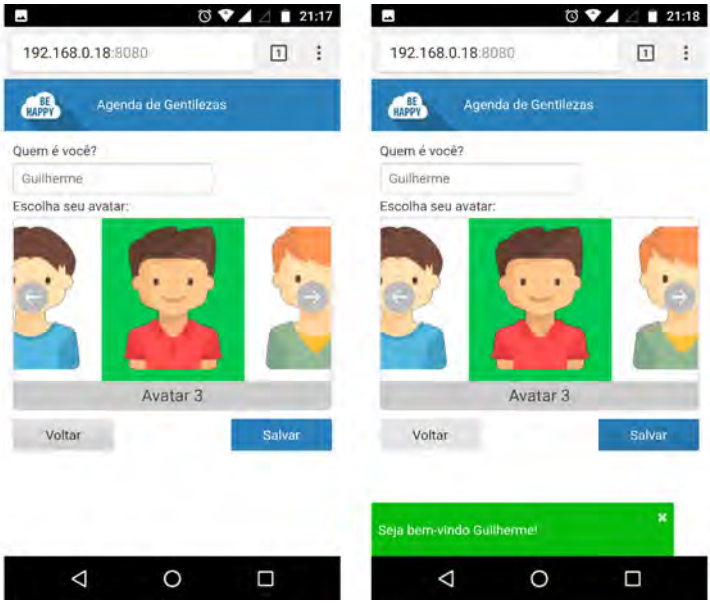

Figura 6.23: Screenshots do celular

Veja a seguir os exemplos da segunda sessão da tela de cadastro dos usuários, agora com o gênero feminino selecionado. Note que a mensagem *Seja bem-vinda Laura*, além de exibir o nome, converteu *bem-vinda* para o gênero correto.

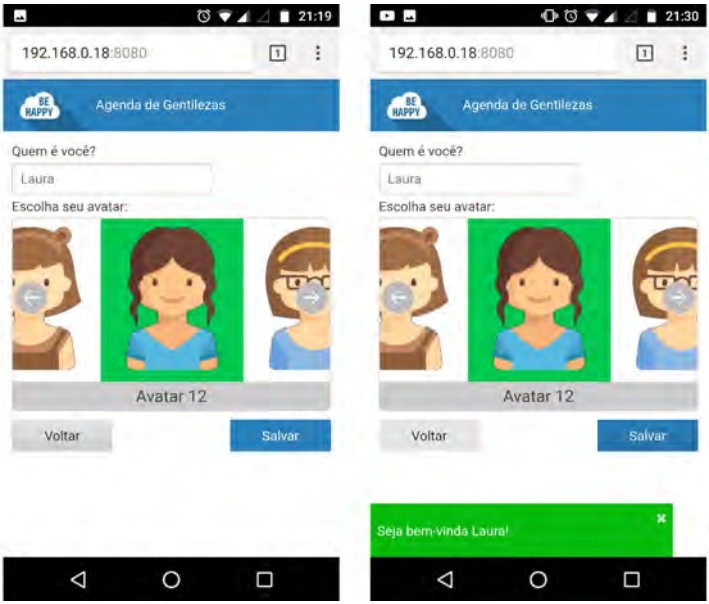

Figura 6.24: Screenshots do celular

Concluímos nossos testes com o celular. De volta ao console (ou prompt), clicando a sequência `Ctrl+C`, você desativará o servidor de desenvolvimento.

Só queria deixar claro que, apesar de possibilitar acesso remoto, o servidor `webpack-dev-server` **não** deve ser utilizado para ambiente de produção. Essa restrição, apontada pela documentação oficial (https://github.com/webpack/webpack-dev-server), existe porque internamente o `webpack-dev-server` não passa de um código construído no framework Express com Node.js rodando na porta 8080.

Como o combo Express+Node.js é próprio para atividades dinâmicas (como executar código JavaScript no *back-end*, por

exemplo), deixá-lo responsável por fornecer apenas conteúdo estático (que é o caso de nossa aplicação PWA) seria um desperdício de tempo de CPU. Em outras palavras, o Node.js é um recurso *mais custoso* do que um servidor estático comum.

No futuro, faremos outros testes iguais a esse através do poderoso servidor *Nginx*, adequado para o caso real de produção.

6.6 CONCLUSÃO

Desta forma, concluímos todo o conteúdo relacionado ao framework React sob o ponto de vista de aplicações progressivas. Discutimos desde as boas práticas da implementação do modelo de domínio de uma aplicação, ao avançado padrão de projeto *Active Record* para viabilizar ações de persistência de forma mais organizada.

Neste capítulo, em especial, criamos os componentes *Button*, *Toast* e *ImageScroller*, cada um com particularidades específicas do ponto de vista de desenvolvimento. No caso do *Button*, simplesmente adotamos a estilização do *Pure.css* e pronto. O *Toast* foi ainda mais simples. A parte complexa do código foi inteiramente reutilizada a partir do componente *React Toastify* disponível no NPM.

Com o *ImageScroller*, o cenário foi totalmente o oposto. Construímos a tag `ButtonImage` e toda a complexidade dos efeitos e usabilidade de um carrocel de imagens nativo no React, sem usar qualquer artifício JQuery ou de bibliotecas externas.

No próximo capítulo, vamos concentrar nossos esforços no entendimento das possibilidades de salvamento de dados no

navegador e no fechamento dos requisitos mais simples relacionados à PWA.

Capítulo 7

CODIFICANDO OS REQUISITOS PROGRESSIVOS

Começaremos este capítulo com duas seções importantíssimas para as aplicações progressivas, mais especificamente para armazenar dados e arquivos no navegador do usuário. Para realizar o salvamento dos dados da aplicação, discutiremos três propostas: *Web SQL Database*, *Indexed Database* e *Web Storage* – sendo esta última a opção utilizada neste livro.

Quanto tratarmos do salvamento dos arquivos, abordaremos as APIs *Service Worker* e *Application Cache*, que, na prática, precisarão coexistir para aumentar a compatibilidade da nossa aplicação com os browsers atuais.

Trabalharemos também com os últimos requisitos PWA que necessitam de alteração no código-fonte do sistema. Para se aproximar do relatório do Lighthouse (ferramenta que quantifica a progressividade de uma aplicação), estes requisitos serão tratados em dois tópicos distintos: com validação manual e com validação automática.

Os requisitos de validação manual poderão ser tidos como

resolvidos ainda neste capítulo. Aqueles com validação automática (que dependem da geração do relatório do Lighthouse) serão verificados no *Capítulo 8*, quando estivermos com todos os requisitos finalizados – inclusive aqueles relacionados à publicação da aplicação no ambiente de produção.

7.1 SALVANDO OS DADOS NO NAVEGADOR

Agora que temos todo o conteúdo proposto para o framework React construído, resta-nos salvar os dados do novo usuário no navegador. Para isso, existem três grandes vertentes: *Web SQL Database*, *Indexed Database* e *Web Storage*. Veremos a partir de agora uma breve introdução a cada uma delas e o embasamento técnico da escolha apropriada para nosso projeto.

O *Web SQL Database*, mais conhecido como *WebSQL*, foi concebido com o intuito de fornecer uma forma eficiente de manipulação de dados via linguagem SQL no navegador. Na prática, trata-se de um SQLite embarcado no navegador. Sua grande vantagem é a utilização do SQL (`select` , `insert` , `delete` etc.).

Como desvantagens, podemos destacar sua incompatibilidade com browsers famosos como o Firefox e o Internet Explorer e, principalmente, o fato de a W3C ter interrompido sua padronização com a justificativa de que outras implementações (além do SQLite) deveriam ter sido realizadas. Por isso, vamos descartá-lo. Para saber mais, veja https://www.w3.org/TR/webdatabase/.

O *Indexed Database*, carinhosamente chamado de *IndexedDB*,

é uma forma de armazenamento *NoSQL* capaz de guardar grandes quantidades de dados estruturados. Além da boa performance, outra grande vantagem é que sua API é assíncrona. Uma tela de consulta com muitos itens, por exemplo, não precisa ficar *congelada* aguardando a obtenção de todos os dados. Ela poderia ir carregando paulatinamente, tornando a interação com o usuário mais agradável.

Como desvantagens, sua API é tida como complexa, e sua portabilidade está na faixa de 87% (https://caniuse.com/#search=indexed) dos browsers atuais – fato preocupante sob a ótica do *Progressive Enhancement*. Para mais detalhes, consulte https://www.w3.org/TR/IndexedDB-2/.

Por fim, resta-nos conhecer o *Web Storage*. Assim como o *IndexedDB*, o *Web Storage* é um mecanismo de armazenamento *NoSQL* cuja especificação W3C (https://www.w3.org/TR/webstorage/) é fortemente ativa na comunidade de software. Sua compatibilidade de 95% com os browsers (https://caniuse.com/#search=storage) é um importante atrativo para aplicações que devem alcançar o maior número de usuários possíveis. Outra vantagem é sua API, mais amigável e simplificada quando comparada com seu concorrente direito (*IndexedDB*).

Como desvantagens, temos seu comportamento síncrono (de I/O bloqueante) e um espaço de armazenamento máximo de 5 MB. Certamente haverá casos em que tais limitações serão determinantes. Na nossa PWA, entretanto, como a quantidade de dados será ínfima, ambas as restrições não trarão problemas para a implementação.

Com essa breve introdução, já temos condições de ratificar a escolha *Web Storage*. A partir de agora, vamos entender como funciona sua arquitetura.

Utilizando o localStorage

O *Web Storage* possui duas maneiras de armazenar os dados: *sessionStorage* e *localStorage*. O primeiro deles só mantém os dados disponíveis durante a mesma sessão do navegador (ao fechá-lo, os dados serão perdidos). O *localStorage*, por sua vez, mantém os dados mesmo que o navegador seja fechado. Nosso foco neste estudo será o *localStorage*.

Vamos usá-lo para salvar os dados de nossa aplicação, com o intuito de persistir o objeto `Usuario` e deixá-lo acessível mesmo nos casos em que a aplicação estiver *offline*. O primeiro passo para se trabalhar com o *localStorage* é recuperar o objeto `Storage` pela sintaxe `window.localStorage` . Esse objeto vai nos fornecer os métodos para definir, recuperar ou apagar os dados do *localStorage*, respectivamente pelos comandos a seguir (o primeiro comando apenas recupera o objeto `Storage`).

```
var storage = window.localStorage;
storage.setItem('nome', 'Guilherme');
let nome = storage.getItem('nome');
storage.removeItem('nome');
```

Um ponto a observar é que o comportamento *key-value* deste banco é limitado ao uso de strings. Para contornar essa limitação, podemos simplesmente converter um objeto JSON com a função `JSON.stringify()` , disponível no JavaScript. No exemplo anterior, se quiséssemos salvar um pequeno objeto com o atributo `nome` , poderíamos empregar o seguinte código:

```
var storage = window.localStorage;
var json = { 'nome': 'Guilherme' };
storage.setItem('json',JSON.stringify(json));
```

Para comprovar que os dados estão sendo salvos, crie um simples HTML com o código anterior, abra-o no Chrome e observe a aba *Application* do DevTools. Nela existe uma sessão chamada *Storage* (à esquerda) composta por alguns mecanismos de armazenamento. Ao clicar na descrição da sua página, dentro do item *Local Storage*, você visualizará (à direita) a chave `json` populada com o objeto `{ 'nome': 'Guilherme' }`.

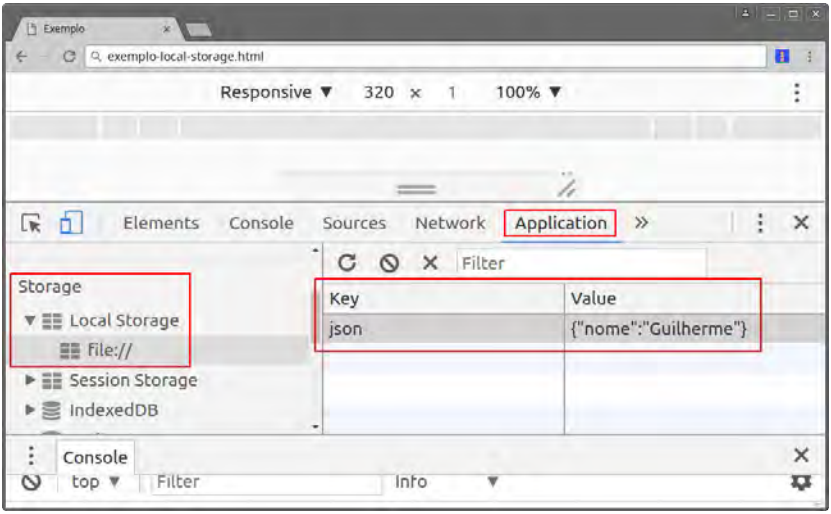

Figura 7.1: Dados salvos no navegador

Como o objeto foi salvo em formato de texto, sua recuperação envolve convertê-lo para JSON através da sintaxe `JSON.parse()`. Veja o código a seguir.

```
var storage = window.localStorage;
var json = JSON.parse(storage.getItem('json'));
console.log(json.nome);
```

Concluímos assim o entendimento da sintaxe do *Web Storage*. No próximo tópico, vamos criar um ponto único (`Repository.js`) de acesso aos dados da aplicação, encapsulando assim toda a *complexidade* inerente à API de forma a escondê-la dos demais componentes e favorecer a manutenibilidade.

Criando o objeto Repository.js

Crie uma pasta chamada `infrastructure` no diretório `front-end/src/` . Dentro dela, crie um arquivo chamado `Repository.js` . Este será composto pelo `constructor` e pelos métodos `salvar()` e `obter()` .

O `constructor` será simplório, servido apenas para definir o nome da *key* usada no *localStorage*. Vamos chamá-la de `behappy-user` . Para começarmos, dentro do arquivo `Repository.js` , crie a estrutura da classe, sua exportação e o `constructor` .

```
class Repository {
    constructor() {
        this.key = 'behappy-user';
    }
}

export default Repository;
```

O método `salvar()` vai receber dois parâmetros: a estrutura de dados, que será salva no *localStorage*, e uma *callback* para retornar o fluxo de execução ao objeto que convocará o método. Como `localStorage.setItem()` é síncrono, a chamada externa ao método `salvar()` não vai bloquear a linha de execução.

Para garantir que o próximo comando seja executado apenas **após** o salvamento no *localStorage*, o objeto convocador deverá

trabalhar dentro da *callback*, algo semelhante ao código a seguir:

```
salvar(json,() => {
    // Neste momento o json já foi salvo
})
```

O método em si vai converter o objeto passado por parâmetro em uma string (através da sintaxe `JSON.stringify()`), salvá-la no *localStorage* e convocar a `callback()`. Veja o código adiante.

```
salvar(json,callback) {
    let data = JSON.stringify(json);
    window.localStorage.setItem(this.key,data);
    callback();
}
```

O método `obter()` será um pouco mais sofisticado. Primeiramente, ele vai obter a string salva no *localStorage* (com auxílio da propriedade `this.key`) e convertê-la em uma estrutura JSON. Um ponto muito importante é que o *localStorage* **não** salva os métodos do objeto.

Em outras palavras, quando o objeto `Usuario` for convertido para string e salvo no navegador, seus métodos `validarNome()` e `validarGenero()` serão perdidos. Isso pode ser contornado **copiando** os dados do JSON para um novo objeto `Usuario`. Faremos isso em breve.

De posse da estrutura JSON, usaremos **duas** *callbacks* para dar sequência ao fluxo de execução no ponto em que o método `obter()` será convocado. Por conveniência semântica, vamos nomear as *callbacks* de `sucesso` e `falha`.

A função `sucesso` será convocada apenas se houver dados resgatados do *localStorage*, enquanto a função `falha` será exatamente o contrário. Essa lógica permitirá tratar o resultado da

chamada de `obter()` com dois fluxos distintos, conforme o exemplo a seguir.

```
obter(json,() => {
    // Dado obtido com sucesso
}, () => {
    // Nenhum dado foi encontrado
})
```

O código a seguir materializa essa ideia na real implementação do método `obter()`.

```
obter(sucesso,falha) {
    let data = window.localStorage.getItem(this.key);
    let json = JSON.parse(data);
    if (json)
        sucesso(json);
    else
        falha();
}
```

Na nossa PWA, não vamos construir uma lógica de exclusão dos dados do *localStorage*. Se fosse o caso, precisaríamos apenas encapsular a sintaxe `storage.removeItem()` em outro método.

Assim, concluímos o `Repository.js`. Caso queira, você poderá visualizar o código completo em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/infrastructure/Repository.js.

Ajustando o model Usuario

Como estudamos no *Capítulo 6*, a classe `Usuario` adotará uma estrutura de *Rich Domain Model* e implementará o padrão *Active Record*. Na prática, isso significa que ele possui métodos de negócio e métodos de persistência. A parte da lógica de negócio já foi implementada. Resta-nos agora codificar os métodos para

salvar e obter os dados.

Como a técnica de persistência em si está centralizada no `Repository.js`, os métodos de `Usuario` servirão apenas como uma película para realizar as transações. Os primeiros passos serão importar e instanciar a classe `Repository`.

Abra o arquivo `Usuario.js`, localizado em `front-end/src/models/`, e inclua em seu início o seguinte código. Note que vamos instanciar `Repository` como uma constante que poderá ser utilizada apenas no escopo interno do código `Usuario.js` (ela não será exportada e não está interna ao escopo da classe `Usuario`).

```
import Avatar from './Avatar'
import Repository from '../infrastructure/Repository';
const repository = new Repository();

class Usuario {
    // ...
}
```

Vamos agora ao método `salvar()`. Ele será bem simples, apenas repassará a instância corrente do objeto `Usuario` e a *callback* recebida como parâmetro para o método `repository.salvar()`.

```
salvar(callback) {
    repository.salvar(this,callback);
}
```

O método `obter()` será mais rebuscado. A primeira questão a ser destacada é que ele terá um escopo de classe (será estático). Como já discutimos, métodos desse tipo costumam apresentar uma fluência cujo acesso se dá diretamente com a sintaxe `Usuario.obter()`, sem recorrer a nenhuma instância. Isso será

implementado por meio do uso da palavra `static` antes da declaração do método.

Outro detalhe a ser observado é a necessidade da *conversão* da estrutura JSON resgatada de `repository.obter()` em um objeto `Usuario` real. Faremos isso criando uma nova instância de `Usuario` e populando-o com os dados obtidos (inclusive com um novo objeto `Avatar`). Esse trecho só será aplicado à *callback* `sucesso`, visto que a *callback* `falha` não precisa retornar nenhum dado.

```
static obter(sucesso,falha) {
    repository.obter(json => {
        let usuario = new Usuario();
        usuario.nome = json.nome;
        usuario.genero = json.genero;
        usuario.avatar = new Avatar(
            json.avatar.index,
            json.avatar.descricao
        );
        sucesso(usuario);
    },falha);
}
```

Com esses três ajustes, cobrimos todas as alterações necessárias. Para fins de testar as operações de persistência, vamos acrescentar também um simples método `toString()`. Ele será utilizado apenas para facilitar a exibição dos dados do usuário resgatado do *localStorage* na tela. A versão final da aplicação **Be happy with me** não utilizará esse método.

```
toString() {
    return `${this.nome}, ${this.avatar.toString()}`
}
```

Desta forma, concluímos o código. Você poderá visualizá-lo na íntegra no repositório do projeto, ou pontualmente em

https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/src/models/Usuario.js/.

Ajustando o componente App

O último ajuste associado a este tópico de persistência será aplicado ao componente `App`. Atualmente, ele renderiza o componente `NovoUsuario` de forma estática. Vamos atribuir a ele a responsabilidade de decidir se a aplicação precisa ou não realizar o cadastro do usuário. Se ele não encontrar dados do usuário no *localStorage*, ele vai renderizar o componente `NovoUsuario`, que, por sua vez, fará o cadastro e salvamento dos dados no navegador.

Por outro lado, se `App` encontrar os dados em *localStorage* (o que significa que o usuário já fez o seu cadastro), ele simplesmente vai exibi-los na tela do sistema. Essa lógica se tornará possível porque, além de viabilizar a persistência e o resgate dos dados salvos no navegador, o componente `App` também vai mantê-los em seu estado interno.

Sequenciando os passos, temos:

1. A aplicação iniciará com uma busca no *localStorage*.
2. Caso nada seja encontrado, o estado de `App` permanecerá vazio e o componente `NovoUsuario` será exibido.
3. Ao final do cadastro, `App` salvará os dados no *localStorage* e em seu estado.
4. Como o método `setState()` implica em uma nova renderização, `NovoUsuario` deixará de ser exibido, dando lugar a uma simples `<div>` com os dados do usuário cadastro.
5. No futuro, quando o navegador for reaberto, `App`

encontrará os dados em *localStorage* e continuará sem exibir `NovoUsuario`.

Na prática, o componente `NovoUsuario` só será exibido uma vez por aplicação (exceto no caso de o usuário apagar o *Web Storage* do navegador). O primeiro ajuste para implementar essa lógica será importar o modelo `Usuario` em `App`. Abra o arquivo `App.js` localizado em `front-end/src/components/` e acrescente a importação, conforme adiante.

```
import React from 'react';
import Header from './Header';
import NovoUsuario from './NovoUsuario';
import Toast from './Toast';
import Usuario from '../models/Usuario'

class App extends React.Component {
    // ...
}
```

A obtenção dos dados do *localStorage* e a criação do estado do componente será realizada no constructor. O uso do método estático `Usuario.obter()` e suas *callbacks* de `sucesso` e `falha` permitem-nos segregar o código em: salvar `usuario` em `this.state` na chamada de `sucesso`; ou deixá-lo explicitamente `undefined` na chamada de `falha`.

```
constructor() {
    super()
    Usuario.obter(usuario => {
        this.state = {
            usuario: usuario
        };
    },() => {
        this.state = {
            usuario: undefined
        };
    });
```

```
}
```

Agora que temos condições de persistir os dados, a *callback* `onSubmit` repassada ao `NovoUsuario` precisará ser refatorada. Até então, ela só exibia um *Toast* de sucesso. Agora ela sequencialmente salvará o usuário no *localStorage*, atualizará o estado de `App` e, por fim, exibirá a mensagem.

Como o aninhamento dessas *callbacks* tornará o código mais complexo, vamos separar a lógica de exibição desse *Toast* em um método à parte, chamado de `msgNovoUsuario()`. Veja seu código a seguir.

```
msgNovoUsuario(usuario) {
    let genero = usuario.genero == 'm' ? 'o' : 'a';
    this.refs.toast.sucesso(
        `Seja bem-vind${genero} ${usuario.nome}!`
    )
}
```

Diante das mudanças citadas, a renderização do `NovoUsuario` poderá ser realizada por um trecho de código semelhante ao exibido a seguir. Note o aninhamento das *callbacks* de `salvar()` e `setState()`.

```
<NovoUsuario
    onSubmit={usuario => {
        usuario.salvar(() => {
            this.setState({
                usuario: usuario
            }, () => {
                this.msgNovoUsuario(usuario)
            })
        });
    }}
    erro={msg=>this.refs.toast.erro(msg)}
/>
```

Como `NovoUsuario` só será exibido no caso de não existir

dados no *localStorage*, podemos concentrar essa lógica em um método separado, que aqui chamaremos de `renderizarNovoUsuario()`. Se o usuário existir, vamos exibir seus dados em uma `<div>`. Do contrário, retornaremos o trecho supracitado do `NovoUsuario`. Veja a seguir o código completo de `renderizarNovoUsuario()`.

```
renderizarNovoUsuario() {
    let usuario = this.state.usuario;
    if (usuario) {
        return (
            <div style={{marginTop: '140px', textAlign: 'center'}
}>
                <b>Usuário obtido do <i>localStorage</i></b><br />

                {usuario.toString()}
            </div>
        )
    } else {
        return (
            <NovoUsuario
                onSubmit={usuario => {
                    usuario.salvar(() => {
                        this.setState({
                            usuario: usuario
                        }, () => {
                            this.msgNovoUsuario(usuario)
                        })
                    });
                }}
                erro={msg=>this.refs.toast.erro(msg)}
            />
        )
    }
}
```

Apenas reforçando, a `<div>` de exibição do usuário cadastrado **não** faz parte da versão final da aplicação. Vamos utilizá-la aqui para fins didáticos e para a validação desse requisito. A aplicação real publicada em https://behappywith.me exibirá a

lista de *gentilezas* do usuário cadastrado.

A construção dessa listagem não faz parte do escopo deste livro, pois você já consegue fazer tudo sozinho com o que aprendeu aqui. Se quiser fazer como exercício, pode conferir seu resultado em https://github.com/lgapontes/behappywith.me/tree/master_en-US.

Por fim, para finalizar o código de `App`, como a renderização de `NovoUsuario` ficou concentrada em `renderizarNovoUsuario()`, o método `render()` poderá ser reescrito como uma simples convocação dos elementos `Header`, `renderizarNovoUsuario()` e `Toast` – todos circundados com uma `<div>`. Veja seu código adiante.

```
render() {
    return (
        <div>
            <Header />
            {this.renderizarNovoUsuario()}
            <Toast ref="toast" />
        </div>
    );
}
```

Concluindo esse ajuste, salve todas as alterações, execute o servidor de desenvolvimento e veja o resultado. No seu primeiro acesso, a aplicação exibirá a tela de cadastro. Ao preenchê-la, a aplicação fará o salvamento e, a partir de então, a `<div>` com os dados do usuário sempre será exibida.

Figura 7.2: Salvando dados do usuário

Da esquerda para a direita, as duas primeiras telas representam o cadastro de um usuário, e a terceira tela é a exibição de seus dados após a persistência. Mesmo que você finalize o navegador, a partir de agora somente a terceira tela será exibida. Caso você queira limpar o *localStorage*, acesse a aba *Application* do DevTools e use a opção *Clear storage*.

Assim concluímos o último requisito funcional da nossa PWA, responsável pela persistência dos dados.

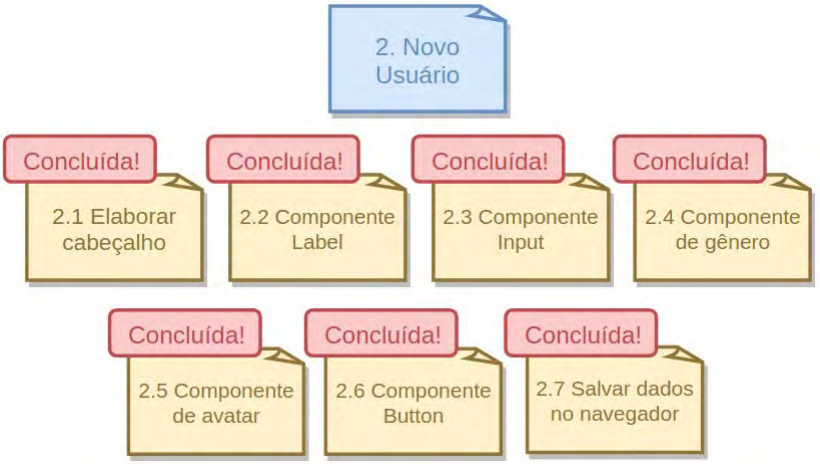

Figura 7.3: Requisitos funcionais da aplicação

Nos próximos tópicos, concluiremos quase todos os requisitos PWA restantes, a começar pelo registro de um *Service Worker*.

7.2 SALVANDO ARQUIVOS NO NAVEGADOR

Ratificando a introdução realizada no *Capítulo 1*, uma aplicação progressiva precisa fornecer a capacidade de funcionar mesmo que o usuário esteja sem acesso à internet. Há duas formas de implementar tal recurso: pelo *Application Cache* ou pelo registro de um *Service Worker* (para salvar os arquivos no *Cache Storage*).

Como nota às questões de compatibilidade, apesar de *Service Worker* ser um recurso poderoso e exigido como requisito das aplicações progressivas, as estatísticas do site https://caniuse.com/#feat=serviceworkers mostram claramente que ele ainda não é compatível com uma grande quantidade de

navegadores.

Em contrapartida, apesar de o *Application Cache* possuir uma especificação obsoleta, ele é aceito em 94% dos navegadores (https://caniuse.com/#feat=offline-apps). Por isso, uma boa prática é utilizar ambas as especificações. Se o usuário estiver com um navegador que não suporta o *Service Worker*, ele utilizará o *Application Cache* e poderá usufruir dos recursos *offline* como qualquer outro usuário.

Configurando o Application Cache

O *Application Cache*, também conhecido como *AppCache*, é uma API declarativa de alto nível na qual podemos informar ao navegador quais arquivos ele precisa armazenar simplesmente relacionando-os em um arquivo estático conhecido como `manifest`. Uma vez especificado, o `manifest` será carregado pela tag `<html>` e, no primeiro acesso à página, salvará os arquivos relacionados no *AppCache*.

Começaremos tal configuração alterando a tag `<html>` do template para avisar ao browser que utilizaremos o *AppCache*. Esse ajuste precisar ser aplicado em todas as páginas HTML do sistema. Como temos uma *Single-Page Application* (SPA), só precisamos alterar o arquivo `front-end/src/index.html`, conforme o código a seguir.

```
<!DOCTYPE html>
<html manifest="behappy.appcache">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initia
l-scale=1.0">
        <link rel="shortcut icon" href="img/favicon.ico">
```

```
        <title>behappywith.me</title>
    </head>
    <body>
        <div id="main"></div>
    </body>
</html>
```

Optei em chamá-lo de `behappy.appcache`, mas poderíamos chamá-lo de qualquer outro nome se mantivermos a extensão `.appcache`. Antigamente, a extensão `.manifest` também era aceita. Devemos evitá-la, pois ela pode conflitar com um *MIME type* não registrado da Microsoft, chamado `application/manifest`.

> **MIME TYPE**
>
> MIME é um acrônimo de *Multipurpose Internet Mail Extensions* (Extensões Multifunção para Mensagens de Internet), que é uma padronização do formato dos arquivos. Ela é configurada nos servidores `http` para indicar ao navegador que tipo de ação deve ser executada de acordo com o tipo do arquivo.

Uma questão importante a ser considerada é que o arquivo `manifest` deve ser provido pelo servidor `http` como sendo do *MIME type* `text/cache-manifest`. Em tempo de desenvolvimento, não precisaremos nos preocupar com esse detalhe, mas, durante a configuração do *Nginx*, no *Capítulo 8*, trataremos dessa restrição.

O código-fonte da nossa aplicação não precisa *conhecer* este

arquivo. Vamos criá-lo diretamente no diretório de distribuição (`front-end/dist`). Crie um arquivo de texto vazio chamado `behappy.appcache` nesse diretório e entre com o trecho a seguir.

```
CACHE MANIFEST
# v0.0.1
```

Como primeira regra, note que um arquivo de `manifest` deve ser inicializado com a sintaxe `CACHE MANIFEST` . O símbolo `#` (hashtag) indica comentários, em que aqui aproveitamos para definir uma versão `v0.0.1` . Esta é uma boa prática porque, uma vez carregado pelo navegador, o arquivo `manifest` só será interpretado novamente se ele possuir alguma diferença.

Em uma situação em que seus arquivos fossem alterados no servidor, para forçarmos uma atualização do lado cliente, podemos simplesmente alterar esse comentário (para `v0.0.2` , por exemplo). Com isso, o navegador saberá que seu `manifest` foi alterado e, consequentemente, atualizará o *Application Cache*.

O próximo passo é populá-lo com o nome dos arquivos que desejamos salvar no navegador. Esse arquivo pode ser subdividido em três seções: `CACHE:` , que indica os arquivos que devem ser armazenados após o primeiro download; `NETWORK:` , para indicar quais arquivos precisam ser baixados do servidor; e `FALLBACK:` , seção opcional que indica páginas substitutas no caso de algum recurso não ser encontrado.

Em nossa aplicação, listaremos todos os arquivos no `CACHE:` e colocaremos um `*` (asterisco) no `NETWORK:` para indicar que os arquivos não relacionados devem ser baixados. Veja a seguir a versão final do arquivo `behappy.appcache` .

```
CACHE MANIFEST
```

```
# v0.0.1

CACHE:
bundle.js
style.css
img/avatars.png
img/botoes.png
img/favicon.ico
img/logo.png

NETWORK:
*
```

Note que o arquivo `index.html` **não** está explicitamente listado. Isso não é necessário porque a página que faz referência ao `manifest` é salva automaticamente. Apesar de não ser um comportamento padronizado entre os navegadores, deve-se evitar listar o próprio arquivo `behappy.appcache` na seção `CACHE:`, pois isso pode prejudicar sua atualização (mesmo quando alterarmos o número da versão `v0.0.1`).

Com essas alterações realizadas, execute o servidor de desenvolvimento, acesse a página http://localhost:8080/ e veja no `console` as ações de salvamento. Para comprovar que de fato os arquivos foram salvos no *AppCache*, acesse a aba *Application*, seção *Application Cache*. Neste item, você encontrará relacionados ao `behappy.appcache` (à direita da imagem a seguir) todos os arquivos guardados (à esquerda).

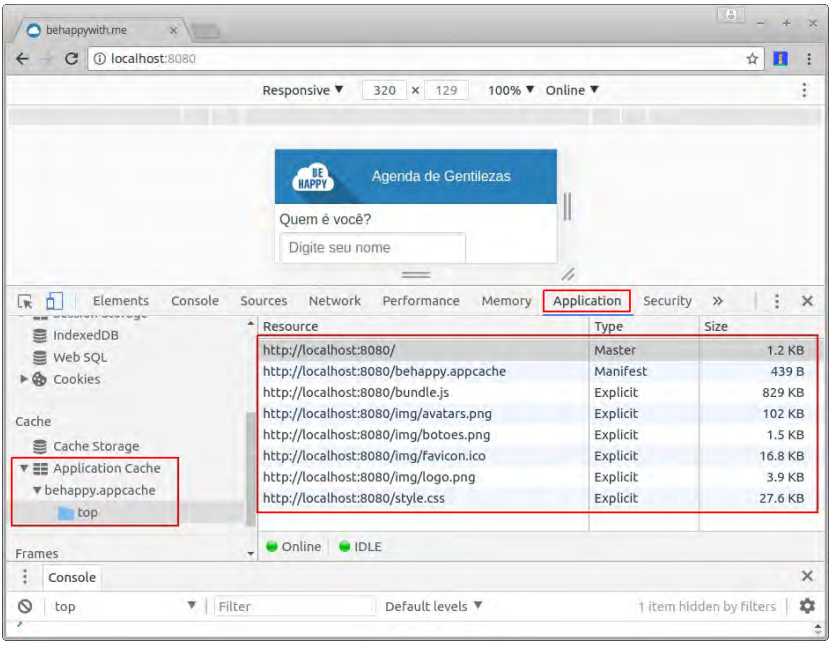

Figura 7.4: Visualização dos arquivos do AppCache

A verdadeira mágica acontecerá se você desligar o `webpack-dev-server` e atualizar a página no navegador. Veja que a aplicação continuará disponível para uso. A exceção é o próprio arquivo `behappy.appcache`, que, como pode ser visualizado no console, não pôde ser encontrado.

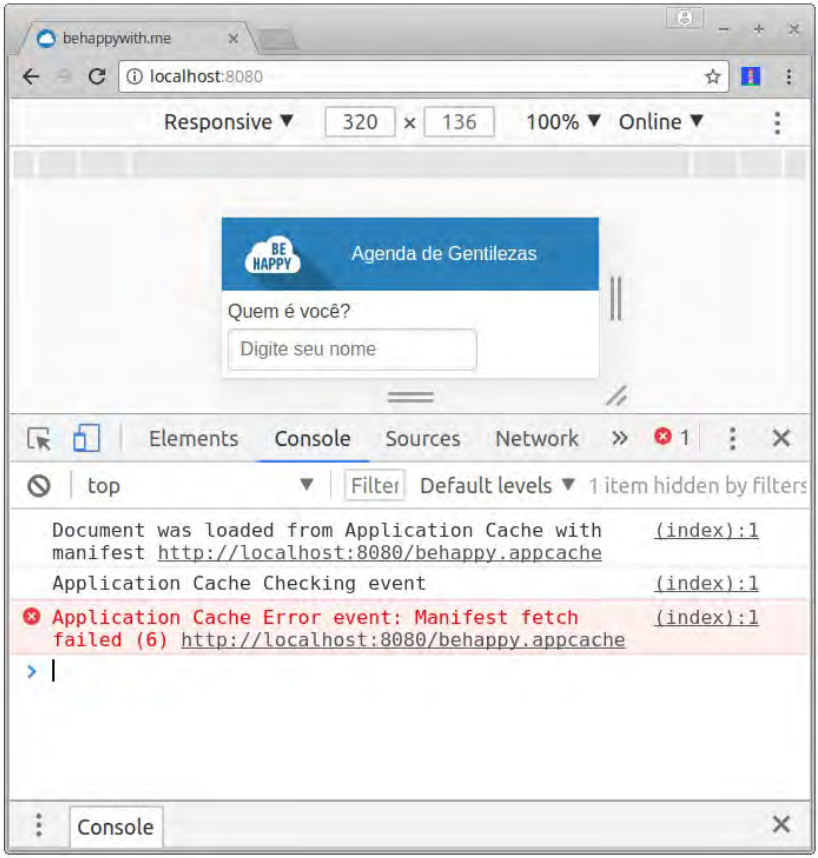

Figura 7.5: Falha na obtenção do manifest

Esse comportamento particular do *Google Chrome* informa que o navegador não conseguiu obter uma nova versão do arquivo `behappy.appcache`. Ele sempre tentará baixar um novo arquivo para garantir que o cache seja atualizado se o arquivo obtido for diferente do armazenado localmente. Essa mensagem não será exibida em outros navegadores.

Com isso, concluímos os testes do *Application Cache*,

permitindo que o navegador continue trabalhando normalmente mesmo com o `webpack-dev-server` desligado. Como adendo, caso você não queira ficar desligando o servidor toda hora, uma alternativa interessante para ser utilizada em tempo de desenvolvimento é o recurso `Offline` do navegador *Google Chrome*.

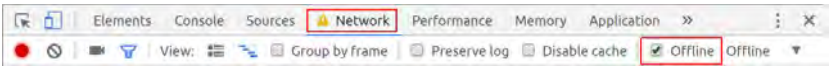

Figura 7.6: Recurso Offline do Chrome

Esse recurso está disponível nas abas *Network* e *Application* do DevTools. Se marcado, o navegador vai simular uma situação cujo servidor estará indisponível (inclusive o `localhost`).

Registrando o Service Worker

A especificação *Service Worker* tem um objetivo semelhante ao *Application Cache*, porém com recursos mais sofisticados que tornarão nossa aplicação realmente funcional em um ambiente *offline*.

Conceitualmente, *Service Worker* é uma API de baixo nível pela qual podemos capturar eventos de busca realizados pela aplicação (item 1, da representação a seguir) e armazenar as respostas do servidor (item 2) no cache do navegador (item 3).Em um segundo momento, podemos interceptar tais solicitações e retornar seus arquivos diretamente do cache – inclusive quando a aplicação estiver *offline*.

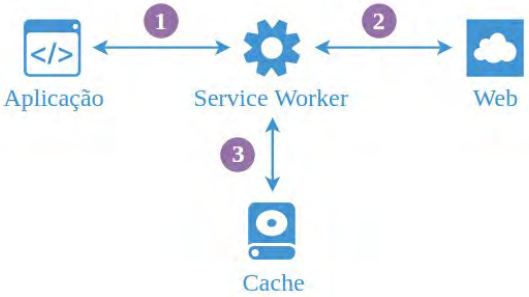

Figura 7.7: Fluxo do Service Worker

Em termos técnicos, o *Service Worker* é um componente JavaScript *event-driven* (conduzido por eventos) registrado no navegador que fica em execução permanentemente. Por meio de seu evento de ativação (marcado pela sintaxe `activate`), estaremos aptos a salvar requisições (`Request`) no *Cache Storage* do navegador. O *Cache Storage* é uma interface assíncrona disponível pela sintaxe `caches`, que organiza *locais* para guardar o resultado das requisições (arquivos) em uma estrutura de chave/valor.

O uso do termo *locais* no plural não é arbitrário. Através do *Cache Storage*, podemos criar vários caches. Poderíamos ter um cache para as imagens, outro para os scripts, e assim por diante. Neste projeto, vamos criar um único cache chamado `behappy.sw-X`, em que o sufixo `X` será substituído por uma variável de versão utilizada para forçarmos uma atualização do *Cache Storage* (no caso de alterações na nossa PWA).

Antes de avançarmos para o código, precisamos nos aprofundar um pouco na especificação da API, a começar pelas etapas do ciclo de vida. O ciclo de vida do *Service Worker* tem por

finalidade garantir que o gerenciamento do cache de uma página seja realizado pelo mesmo script, que, por sua vez, só poderá assumir controle após o término da execução da versão anterior. Em outras palavras, só haverá uma versão do *Service Worker* sendo executado por vez – mesmo em abas diferentes do navegador.

Figura 7.8: Ciclo de vida do Service Worker

O item *Registrando* do fluxo anterior consiste em baixar o script da rede e registrá-lo no navegador através do método `navigator.serviceWorker.register()`. Vamos tratar desta etapa no próprio arquivo `index.html` (localizado em `front-end/src/`).

Como há navegadores que não suportam esse recurso, acrescentaremos uma condição para verificar se o objeto `serviceWorker` existe e, se for caso, faremos os passos de registro. Para acompanharmos a configuração com acuidade, vamos adicionar também mensagens no console do navegador para exibir o sucesso ou falha das etapas. Acrescente no `<head>` do `index.html` uma tag `<script>` com o seguinte conteúdo:

```
<script>
    if('serviceWorker' in navigator) {
        navigator.serviceWorker.register('behappy.sw.js')
            .then(function(registration) {
                console.log('ServiceWorker registrado com sucesso
!');
            }).catch(function(err) {
                console.log('Erro no registro: ', err);
        });
    }
</script>
```

Veja que nomeamos o arquivo do *Service Worker* de `behappy.sw.js` – no trecho `register('behappy.sw.js')`. De forma semelhante ao que fizemos com o *AppCache*, o *webpack* não precisa conhecer esse arquivo em tempo de desenvolvimento. Vamos criá-lo diretamente na pasta `front-end/dist`.

Seguindo o ciclo de vida, o primeiro estado do *Service Worker* é conhecido como *Instalando*. Ele parte da captura do evento `install` e, na prática, significa que estamos na fase de instalação (ainda não completada). Em nosso caso, vamos apenas exibir no console a conclusão desta etapa, mas poderíamos utilizar esse momento para guardar arquivos em cache.

A captura dos eventos se dará pelo método `window.addEventListener()`, que recebe dois parâmetros: o nome do evento e a função de tratamento do evento. Um ponto de atenção é que o script do *Service Worker* **não** possui acesso ao objeto `window`. Teremos de utilizar sua referência `self` – que, na prática, é uma propriedade apenas de leitura, exatamente igual ao `window`. Veja a seguir como ficará nosso tratamento à fase de *Instalando*.

```
function instalarServiceWorker() {
    console.log('ServiceWorker instalado com sucesso!');
```

```
}

self.addEventListener("install", instalarServiceWorker);
```

O escopo do método `instalarServiceWorker()` representa a fase *Instalando* do fluxo. Um *Service Worker* instalado está pronto para entrar na fase de *Ativando*, porém só fará isso quando sua versão antiga (se houver) estiver sido finalizada em todas as abas do navegador.

Vamos tratar agora do evento `activate`. De forma semelhante à instalação, o processo de ativação foi subdividido em duas fases: *Ativando* e *Ativado*. Durante a fase *Ativando*, que será implementada na função `ativarServiceWorker()`, aproveitaremos para salvar nossos arquivos no cache e eventualmente apagar versões antigas desses arquivos (caso existam). Esta prática é muito importante durante a evolução da aplicação, momento em que novos arquivos (CSS, scripts, imagens e o próprio `behappy.sw.js`) podem ser publicados.

Como vimos, a codificação dessa regra de atualização do cache terá como base a string `behappy.sw`, sufixada com uma variável de versão. A partir desta versão, criaremos dois identificadores do cache: `idAtual` e `idAnterior`. Veja como ficará:

```
var identificador = 'behappy.sw';
var versao = 1;
var idAtual = identificador + '-' + versao;
var idAnterior = identificador + '-' + (versao - 1);
```

De posse do identificador do cache atual, vamos repassá-lo como parâmetro ao método `caches.open()` para registrá-lo (caso não exista) e abri-lo. Após este passo, `caches.open()` vai nos retornar uma `Promise` (uma *promessa* de objeto para processamento assíncrono) que fornece um método `then()` ao

qual passaremos uma callback para tratamento do cache.

Aproveitaremos este momento para informar ao navegador uma lista de arquivos que devem ser *cacheados*. Na prática, isso significa manter uma lista das URLs de todos os arquivos. Isso pode ser realizado com a criação de um array simples, como no trecho a seguir.

```
var urls = [
    '/',
    'bundle.js',
    'style.css',
    'img/avatars.png',
    'img/botoes.png',
    'img/favicon.ico',
    'img/logo.png',
];
```

Note que não deixamos explícito o arquivo `index.html`. Ele será *cacheado* por meio da *Request* à URL `/`. Tendo a lista de arquivos, vamos acrescentá-los ao cache pela sintaxe `cache.addAll(urls)`, em que `cache` é o parâmetro da callback repassada ao `caches.open(idAtual).then()`.

O método `addAll()` também retornará uma `Promise` (com o método `then()` implementado) pela qual excluiremos o cache antigo (`idAnterior`). Veja a seguir como ficará nosso arquivo `behappy.sw.js` após essa codificação. A captura e o tratamento do evento `install` foram omitidos.

```
var identificador = 'behappy.sw';
var versao = 1;
var idAtual = identificador + '-' + versao;
var idAnterior = identificador + '-' + (versao - 1);

var urls = [
    '/',
    'bundle.js',
```

```
    'style.css',
    'img/avatars.png',
    'img/botoes.png',
    'img/favicon.ico',
    'img/logo.png',
];

function ativarServiceWorker() {
    caches.open(idAtual).then(cache => {
        console.log('Cache Storage ' + idAtual + ' foi ativado co
m sucesso!');

        cache.addAll(urls)
            .then(function(){
                caches.delete(idAnterior)
                console.log('Cache Storage ' + idAnterior + ' foi
 excluído!');
            })
    })
}

self.addEventListener("activate", ativarServiceWorker);
```

Ao final da execução do método `ativarServiceWorker()`, finalmente chegaremos ao estado *Ativado*, que na prática é onde a mágica acontecerá. Neste momento, o *Service Worker* ficará ocioso aguardando por uma requisição (evento `fetch`).

Ao capturarmos este evento, faremos uma busca do arquivo no *Cache Storage* com a sintaxe `caches.match(event.request)`. Se o arquivo for encontrado, retornaremos ao navegador sem a necessidade de ir à internet. Caso contrário, permitiremos o comportamento normal (ir à internet) através do método `fetch(event.request)`. Veja a seguir a codificação dessa lógica.

```
function buscarArquivos(event) {
    event.respondWith(
        caches.match(event.request).then(function(arquivoCache) {
            return arquivoCache ? arquivoCache : fetch(event.requ
est);
```

```
        })
    )
}

self.addEventListener("fetch", buscarArquivos);
```

Perceba que nosso método `buscarArquivos()`, apesar de funcional, é bem específico para a nossa aplicação. Arquivos que não estiverem armazenados no cache simplesmente serão enviados ao navegador sem nenhum tratamento específico.

Em outros tipos de aplicação nas quais os usuários poderiam baixar fotos da internet com intuito de apresentá-las nas páginas, precisaríamos de uma dinâmica mais sofisticada em que arquivos não *cacheados* seriam salvos pela sintaxe `cache.put(event.request, response)`. Veja esta lógica no trecho a seguir. Não a codifique; ficará apenas como referência.

```
function buscarArquivosComSalvamento(event) {
    event.respondWith(
        caches.match(event.request).then(function(arquivoCache) {

            if (arquivoCache) {
                return arquivoCache;
            }

            var cloneDoRequest = event.request.clone();
            return fetch(cloneDoRequest).then(function(response)
{
                var cloneDoResponse = response.clone();
                caches.open(idAtual).then(function(cache) {
                    cache.put(event.request, cloneDoResponse);
                });
                return response;
            });
        })
    );
}
```

Observe que, neste método, clonamos os objetos `request` e

`response`. Isso foi necessário porque ambos são *streams* que só podem ser consumidos uma única vez. O `request` foi utilizado para o `fetch()` e para o `cache.put()`, enquanto o `response` para o `cache.put()` e para o próprio navegador (em `return response;`).

Com isso, finalizamos a tratamento dos eventos do *Service Worker* para nossa aplicação. Você poderá baixar a versão completa deste arquivo em https://raw.githubusercontent.com/lgapontes/behappywith.me/master/front-end/dist/behappy.sw.js/, se necessário.

Para concluirmos o fluxo, o estado *Redundante* representa o momento em que o *Service Worker* foi substituído por uma nova versão. Ele é útil para o navegador manter a consistência dos scripts, mas não trará nenhum desdobramento para nossa aplicação.

Execute o servidor de desenvolvimento para testarmos o *Service Worker*. Ele funcionará mesmo com o *AppCache* configurado. Porém, caso você queira testar seu funcionamento de forma individual, substitua a abertura `<html manifest="behappy.appcache">` por `<html>` no arquivo `index.html`. Isso evitará a busca pelo arquivo `behappy.appcache` e deixará o *Service Worker* como o único responsável pelo funcionamento *offline*.

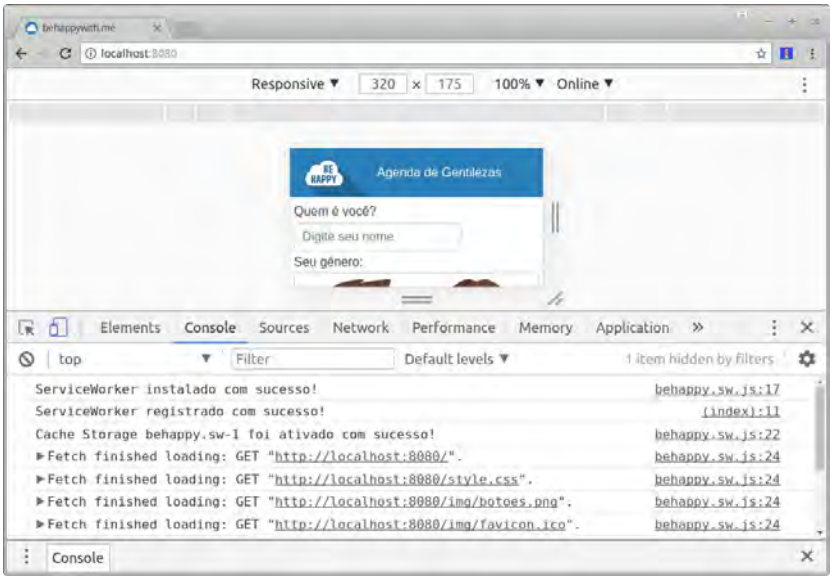

Figura 7.9: Testando o Service Worker

Acessando a página, você verá no console as operações realizadas pelo script `behappy.sw.js`. Para conferir se tudo ocorreu bem, acesse a aba *Application* do DevTools e veja o *Service Workers* ao lado direito da tela.

Uma dica interessante é deixar sempre marcada o checkbox *Show all*. Ele vai ajudá-lo a detectar versões obsoletas do seu script que, por ventura, tenham ficado presas em outras abas do navegador.

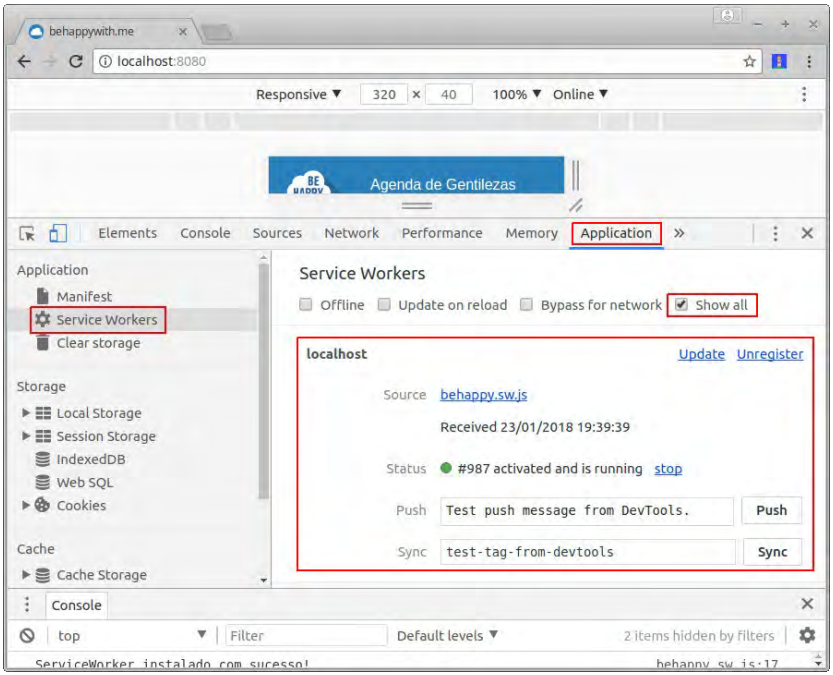

Figura 7.10: Testando o Service Worker

Como o *Service Worker* está no estado *Ativado*, a função `ativarServiceWorker` (do evento `activate`) já salvou os dados no *Cache Storage*. Veja-os nesta mesma aba, um pouco mais abaixo no DevTools.

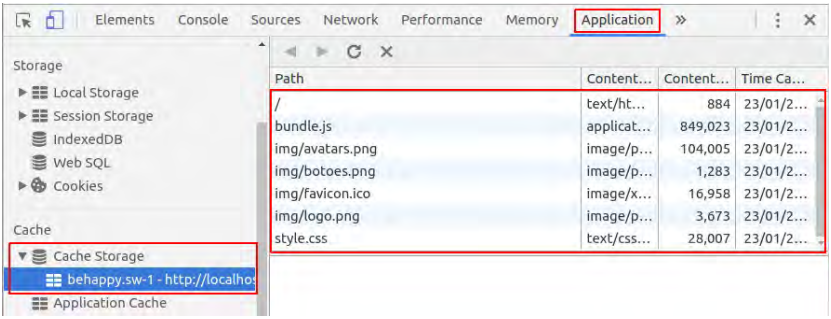

Figura 7.11: Arquivos salvos no Cache Storage

Como última etapa do teste, desligue o servidor de desenvolvimento (ou marque a checkbox *Offline* desta mesma seção *Service Workers*) e recarregue a página. Neste momento, o script `behappy.sw.js` buscará os arquivos no *Cache Storage* e os retornará ao navegador com uma resposta 200 padrão do HTTP, conforme pode ser confirmado na aba *Network* do navegador.

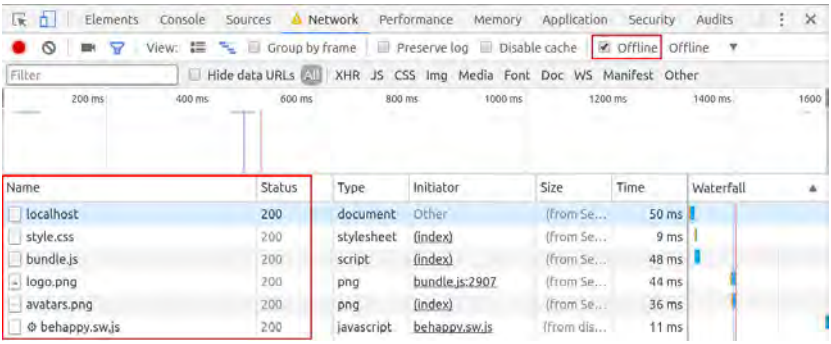

Figura 7.12: Resposta 200 mesmo off-line

Se você estiver usando o Google Chrome, após este último teste, é possível que você se depare com um ruído de erro no console do *Service Worker*. Veja a seguir um exemplo de como

seria tal situação.

Figura 7.13: Arquivos salvos no Cache Storage

Isso pode ocorrer porque o navegador não conseguiu obter o arquivo `behappy.sw.js`. Como **não** é recomendado salvá-lo explicitamente no *Cache Storage* pelo *Service Worker* (ou seja, ele não está listado na variável `urls`), a aplicação não pôde obtê-lo e registrou tal fato por meio da mensagem exibida na imagem anterior.

Este erro não afetará o funcionamento do seu *Service Worker* e não ocorrerá no ambiente de produção. Você pode se deparar com ele por uma particularidade do provimento de arquivos do `webpack-dev-server`, que, por padrão, não configura nenhum limite de tempo máximo para armazenamento dos arquivos.

Caso você queira contornar essa situação (em tempo de desenvolvimento), abra o arquivo de configuração do *webpack* em `front-end/webpack.config.js` e acrescente a propriedade `Cache-Control` como padrão no cabeçalho das respostas do `webpack-dev-server`, conforme trecho a seguir.

```
// ...
module.exports = {
    // ...
```

```
    devServer: {
        publicPath: "/",
        contentBase: "./dist",
        headers: { "Cache-Control": "max-age=600" }
    }
};
```

Esta propriedade diz ao navegador que os arquivos deverão permanecer no cache por 600 segundos. Note que isto não afetará o *Application Cache* ou o *Cache Storage*. Ele serve para incrementar o tempo de permanência dos arquivos estáticos no bom e velho cache do navegador.

Após essa configuração, reinicie o servidor `webpack-dev-server` , apague os dados do navegador (aba *Application*, opção *Clear storage*), e veja que o ruído de erro não ocorrerá. Como adendo, note que isso afetará a atualização dos arquivos, causando a falsa impressão de que as atualizações do código-fonte não estão sendo publicadas. Para evitar esta confusão, utilize seu navegador em *modo privativo* (janela anônima).

Logo, concluímos outros dois requisitos das aplicações progressivas, mais especificamente aqueles que indicam o registro de um *Service Worker* e um tipo de retorno com código 200 para os *Requests*, realizados enquanto a aplicação estiver *offline*.

Figura 7.14: Conclusão de requisitos PWA

Um último ponto intrinsecamente associado à finalização

desses requisitos é que os recursos do *Service Worker* só podem ser usados localmente (via acesso ao `http://localhost`), ou por meio de uma conexão segura (via HTTPS). Esta é uma regra da especificação para proteger os usuários de scripts maliciosos que porventura poderiam ser instalados em seu navegador. Como a utilização do HTTPS é um dos outros requisitos das aplicações progressivas que vamos cobrir em breve, não teremos problemas com essa restrição.

7.3 REQUISITOS COM VERIFICAÇÃO MANUAL

Neste tópico, trataremos dos requisitos cuja ferramenta Lighthouse não é capaz de avaliar automaticamente. Ou seja, são aqueles que vamos validar manualmente a partir de uma série de orientações oficiais, mantidas no endereço https://developers.google.com/web/progressive-web-apps/checklist#site-works-cross-browser/. Vamos começar verificando se a nossa aplicação é *cross-browser*.

O site deve ser cross-browser

Segundo essa especificação, uma aplicação *cross-Browser* deve ser testada nos navegadores *Google Chrome*, *Microsoft Edge*, *Mozilla Firefox* e *Safari*.

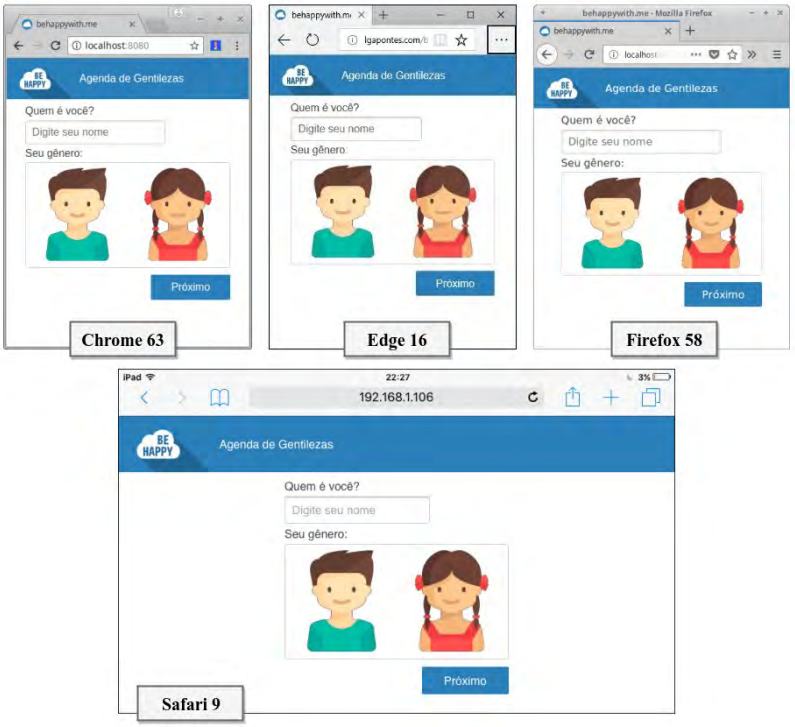

Figura 7.15: PWA utilizada em navegadores distintos

Os testes realizados nos navegadores cujas versões estão expostas na imagem anterior não apresentaram problemas de compatibilidade. Testes mais robustos poderiam contemplar uma gama maior de versões e softwares, mas, para o escopo deste trabalho, esses quatro browsers já são suficientes.

Com isso, segundo a documentação oficial do Lighthouse, podemos considerar que nossa aplicação progressiva é *cross-browser*. Só como adendo, o Firefox 58 destacou em seu console que o recurso do *AppCache* está depreciado e que, em breve, pode deixar de ser disponibilizado. Isso não será problemas para nós,

porque o Firefox também suporta o *Service Worker*.

As transições entre páginas não devem ser sensíveis à velocidade de conexão

Ainda sob as regras do Lighthouse, para que as páginas de uma *Single-Page Application* (SPA) não sejam consideradas sensíveis à velocidade de conexão, as transições entre elas podem ocorrer instantaneamente ou, em casos de download de muito conteúdo, devem fornecer um ícone de *loading*. Como nossa aplicação é uma SPA e não possui carregamento de conteúdo pesado, já podemos considerá-la aderente a essa especificação.

Esta avaliação está intrinsecamente relacionada ao requisito 1.6, que trata do carregamento das páginas em conexões 3G. Páginas não SPA seriam extremamente prejudicadas em casos de pouca banda de internet, em que as transições entre as páginas seriam afetadas pela velocidade de conexão. O requisito 1.6 será efetivamente testado no próximo capítulo, quando publicarmos a aplicação em ambiente de produção.

Cada página deve ter uma URL

Por fim, o último item de verificação manual orienta a configuração de URLs únicas para cada página da aplicação, ou seja, as URLs devem ser *bookmarkable* (passíveis de salvamento no histórico do navegador). O React não oferece recursos nativos para tratamento desta situação, porém, como alternativa oficial, ele indica o uso do *React Router*.

O uso completo deste framework não faz parte do escopo deste livro, mas trabalharemos aqui pequenos exemplos de sintaxe que

poderiam ser usados em *Single-Page Application* com muitas páginas. Sua instalação pode ser realizada através do NPM. Por meio do console, entre no diretório `behappywith.me/front-end` e execute o comando a seguir.

```
npm install --save react-router@4.2.0 react-router-dom@4.2.2
```

O *React Router* trabalha com dois módulos principais, chamados *React Router* e *React Router DOM*, cujas respectivas responsabilidades são manter o core do framework e manter os componentes utilizados no DOM da aplicação.

Usar roteamento das URLs no lado do cliente (navegador) é um comportamento muito comum nas aplicações web contemporâneas. Este paradigma oferece vantagens, mas, ao mesmo tempo, exige uma certa cautela. A principal vantagem é que, após o carregamento da página (mais especificamente do JavaScript que faz o roteamento), quando o usuário clicar em um de seus links, a própria aplicação fará o tratamento da rota sem a necessidade de se comunicar com o servidor HTTP. Ou seja, o overhead – associado ao envio da solicitação ao servidor, do processamento da rota e retorno para o navegador – é poupado com uma simples rotina JavaScript que roda localmente.

A desvantagem ocorre quando o usuário digita a URL no navegador. Como o JavaScript que faz a rota ainda não foi carregado, quem de fato vai interpretar a solicitação será o servidor. E para que suas URLs sejam direcionadas para a página correta, ele também precisa *conhecer* a configuração dessas rotas.

Manter essa lógica do roteamento duplicado no servidor e no navegador pode gerar problemas de manutenibilidade e falha na aplicação. Uma alternativa praticada é fazer com que o servidor

sempre redirecione para a mesma página *independente* da URL acessada.

Em outras palavras, qualquer URL que o usuário acessar (http://pagina.com, http://pagina.com/lista, http://pagina.com/dados etc.) sempre retornará o mesmo arquivo `index.html` para o navegador.

Com esses conceitos esclarecidos, podemos agora ressaltar que, para o uso pleno dos recursos do *React Router*, vamos precisar fazer um pequeno ajuste no arquivo de configuração do `webpack-dev-server`, mais especificamente na seção `devServer`. Acrescente a sintaxe `historyApiFallback: true` para informar ao `webpack-dev-server` que todas as respostas 404 (página não encontrada) sempre devem retornar a página principal `index.html`. Veja como ficará a alteração deste trecho no código a seguir.

```
// ...

module.exports = {
    // ...
    devServer: {
        publicPath: "/",
        contentBase: "./dist",
        headers: { "Cache-Control": "max-age=600" },
        historyApiFallback: true
    }
};
```

Com esta etapa de configuração realizada, resta-nos a criação de um código de exemplo. Para não poluir o código construído nos capítulos anteriores, sugiro a criação de um novo `index.jsx` dentro da pasta `behappywith.me/front-end/src/`. Atenção: não apague o arquivo `index.jsx` atual, pois o exemplo tratado

neste tópico não será integrado à nossa PWA.

A utilização do *React Router* é muito simples. O primeiro passo é importar os componentes `BrowserRouter`, `Link` e `Route` do `react-router-dom`. O `BrowserRouter` serve como organizador das rotas, ficando exposto como um agrupador dos outros componentes. O `Link` é usado para criar um hyperlink entre as rotas, e o `Route` é quem define o nome e o trecho JSX que será exibido nas rotas.

Para facilitar o uso do componente `Route`, é razoável criar funções com o código JSX que serão utilizadas dentro da sintaxe do `Route`.

```
import React from 'react'
import ReactDOM from 'react-dom'
import {
    BrowserRouter,
    Link,
    Route
} from 'react-router-dom'

const Principal = () => {
    return <h1>Página principal</h1>
}
const Dados = () => {
    return <h1>Dados de dados</h1>
}
```

Com as funções prontas, podemos associá-las às tags `Route`, que ficarão contidas dentro do componente `BrowserRouter`. Colocaremos esse código diretamente no método `ReactDOM.render()`, conforme pode ser visto adiante.

```
ReactDOM.render(
    <BrowserRouter>
        <div>
            <Route exact path="/" component={Principal} />
```

```
            <Route path="/dados" component={Dados} />
            <br /><br />
            <Link to="/">Principal</Link> - <Link to="/dados">Dad
os</Link>
        </div>
    </BrowserRouter>,
    document.querySelector("#main")
)
```

Analisando de perto, as sintaxes `path` e `component` de `Route` servem para definir a URL a partir da qual a página será exibida e o trecho JSX a ser exibido, respectivamente. O uso da palavra `exact` é necessário porque a string `/` também está contida em `/dados`. Sem o `exact`, o *React Router* reconheceria ambas configurações e apresentaria as duas rotas ao mesmo tempo. Por fim, o parâmetro `to` do componente `Link` serve para indicar qual rota deve ser exibida quando o hyperlink for clicado.

Supondo estarmos com o mesmo `index.html` do projeto (que possui uma `<div id="main"></div>` para injetar o conteúdo do código anterior), ao executar o servidor de desenvolvimento, é possível verificar o funcionamento das rotas com URLs únicas e exclusivamente coordenadas pelo JavaScript do navegador.

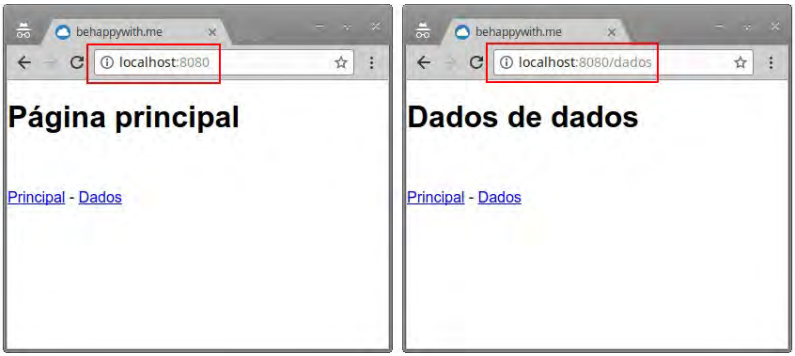

Figura 7.16: Rotas executadas pela aplicação

As duas páginas criadas neste exemplo são *bookmarkable*, atendendo plenamente esse requisito das aplicações progressivas. Além disso, temos o benefício do roteamento exclusivo do lado cliente a partir do *React Router*.

Com isso, concluímos o entendimento e a implementação dos três requisitos de verificação manual do Lighthouse.

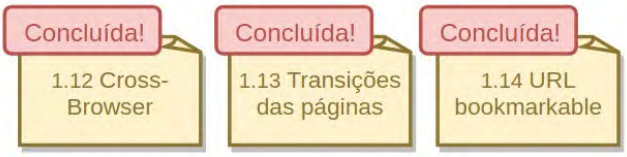

Figura 7.17: Conclusão de requisitos PWA

No próximo tópico, trataremos dos requisitos cuja validação se dará de forma automática.

7.4 REQUISITOS COM VERIFICAÇÃO AUTOMÁTICA

A partir de agora, vamos trabalhar nos últimos requisitos que necessitam de alteração no código-fonte do sistema. Apesar de serem considerados muito importantes, você perceberá que eles são pouco intrusivos, exigindo apenas pequenas alterações nos arquivos `index.html` e `index.css`, e a criação do arquivo de instalação da PWA.

Exibir conteúdo com o JavaScript desabilitado

Neste tópico, criaremos um mecanismo para exibir uma mensagem estática aos usuários que acessarem nossa aplicação com o JavaScript do navegador desabilitado. Esta alteração baseia-se no uso da tag `<noscript>`, que é capaz de apresentar elementos do DOM apenas nos casos em que o recurso de script do navegador estiver desativado.

Uma outra boa prática é esconder os trechos JavaScript com as sintaxes de comentário `<!--` e `//-->`, evitando que o browser tente exibi-los como texto simples. Em nossa aplicação, ambas técnicas serão aplicadas diretamente no arquivo `index.html`, localizado em `front-end/src/`.

A tag `<noscript>` pode ser acrescentada no `<body>` ou no `<head>`, sendo que neste último ela só poderia conter os elementos `<link>`, `<style>` ou `<meta>`. Como nossa intenção é apresentar conteúdo, vamos colocá-la no início do `<body>`. Os demais trechos do `index.html` foram omitidos.

```
<body>
    <noscript>
        <div>
            Desculpe, mas não é possível utilizar esta aplicação
com o recurso de scripts desabilitado.
            Verifique como habilitá-lo e tente novamente.
```

```
            <br />
            <img src="img/logo.png" />
        </div>
    </noscript>
    <div id="main"></div>
</body>
```

Note que, além do texto, estamos exibindo também um `<img>` apontando para o logo da aplicação. Falta-nos agora circundar o JavaScript com a sintaxe de comentário. O único trecho tratado será aquele que registra o *Service Worker* no navegador. Altere-o como no código a seguir.

```
<script>
    <!--
    if('serviceWorker' in navigator) {
        navigator.serviceWorker.register('behappy.sw.js')
            .then(function(registration) {
                console.log('ServiceWorker registrado com sucesso
!');
            }).catch(function(err) {
                console.log('Erro no registro: ', err);
        });
    }
    //-->
</script>
```

Para que a mensagem exposta em `<noscript>` não seja exibida em texto puro, sem formatação, podemos estilizá-la com duas classes CSS. Acrescente-as no arquivo `front-end/src/css/index.css`.

```
noscript div {
    background-color: #2c80b9;
    color: #ffffff;
    font-weight: bold;
    font-size: 16px;
    text-align: center;
    padding: 10px 10px 100px 10px;
    border: 0;
    position: relative;
```

```
}

noscript div img {
    position: absolute;
    right: 0;
    bottom: 0;
}
```

Com isso, concluímos esse requisito. Caso você queira simulá-lo, o Google Chrome oferece um recurso para desativar o JavaScript. Ele está disponível no ícone de mais opções do DevTools (três pontos), menu *Settings*, seção *Preferences*. Marque o checkbox *Disable JavaScript*, confome a imagem a seguir.

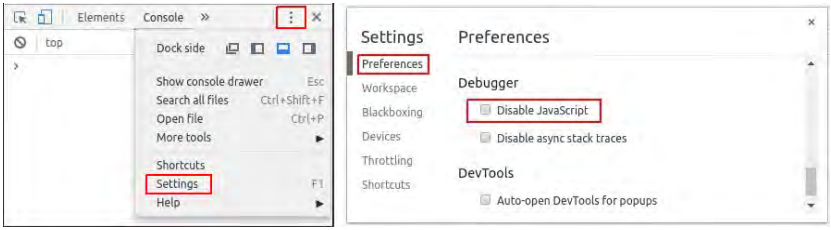

Figura 7.18: Desativar JavaScript do navegador

Ao acessar a página com o JavaScript desativado, você verá o conteúdo de `<noscript>` sendo apresentado.

Figura 7.19: Desativar JavaScript do navegador

No próximo tópico, vamos estilizar a barra de endereços com as cores de nossa PWA.

Estilizando a barra de endereço

Apesar de pouco relevante sob o ponto de vista de usabilidade, o simples fato de colorir a barra de endereço do browser com as cores primárias do site melhora bastante a apresentação, deixando a página com uma aparência mais próxima de uma aplicação

nativa. Para implementarmos tal recurso, basta aplicarmos uma metatag `theme-color` cujo conteúdo será igual à cor azul (`#2c80b9`) do cabeçalho da página.

Essa metatag, todavia, não é compatível com aparelhos mobile rodando Windows ou iOS. Por isso, vamos também acrescentar outras três metatags específicas para esses dispositivos. Abra o arquivo `front-end/src/index.html` e acrescente a sintaxe seguinte dentro da tag `<head>` .

```
<!-- Google Chrome e Mozilal Firefox -->
<meta name="theme-color" content="#2c80b9">
<!-- Windows Phone -->
<meta name="msapplication-navbutton-color" content="#2c80b9">
<!-- iOS Safari -->
<meta name="apple-mobile-web-app-status-bar-style" content="#2c80
b9">
<meta name="apple-mobile-web-app-capable" content="yes">
```

Como observação, há versões do iOS que não exibirão a cor a menos que você ative a tag `apple-mobile-web-app-capable` (que possibilita o modo `full screen` do sistema). Para visualizar essa alteração, você precisará acessar o servidor de desenvolvimento por meio de um dispositivo mobile. Execute-o com a sintaxe `--host` seguida do IP do seu computador, como fizemos no final do *Capítulo 6*. Você verá um resultado semelhante à imagem a seguir.

Figura 7.20: Barra de endereços colorida

No próximo tópico, trataremos conjuntamente dos requisitos de instalação e da *splash screen* de nossa PWA.

Permitir a instalação da aplicação

Estes dois últimos requisitos devem ser tratados conjuntamente porque a *splash screen* que vamos configurar é apenas um caso particular para aplicações mobile passíveis de

instalação no dispositivo. O termo *splash screen* em si é mais abrangente do que aquele proposto pelas aplicações progressivas. Ele também é utilizado, por exemplo, para indicar aqueles ícones de *loading* que bloqueiam o uso de páginas web durante o carregamento.

Nosso *splash screen* será aquele estritamente associado ao carregamento de um aplicativo no dispositivo mobile. Mais especificamente, nossa tela de abertura só funcionará quando for aberta a partir do *launcher* (ícone de execução pós-instalação) em navegadores Google Chrome de versão 47 (ou superior) instalados no Android.

Ela não funcionará em outros sistemas operacionais mobile (iOS e Windows) nem em browsers obsoletos. Apesar dessa forte limitação, o Google Chrome fará todo o trabalho, criando a *splash screen* automaticamente a partir dos dados que vamos configurar no arquivo `manifest.json`.

Não confunda o arquivo `manifest.json` com o arquivo de manifesto `behappy.appcache`. Este, conforme estudamos, está relacionado ao salvamento dos arquivos no *Application Cache*. O `manifest.json`, por outro lado, nos permite instalar nossa aplicação nos dispositivos mobile.

O arquivo `manifest.json` é um típico JSON disponível na raiz da página para ser interpretado pelo navegador. Nesta ocasião, o navegador apresentará ao usuário um diálogo questionando se ele deseja instalar a aplicação.

Entre outros campos que compõem esse arquivo, podemos destacar:

- `name` : um campo obrigatório que serve como identificador primário da aplicação. Como ele será exibido no diálogo de instalação, recomenda-se que tenha no máximo 45 caracteres. Vamos utilizar o valor `behappywith.me` .
- `short_name` : versão reduzida do nome da aplicação que será exibida no ícone do aplicativo (na tela do celular) e no título das abas do navegador. Ele não é obrigatório e, caso seja omitido, será substituído pelo campo `name` truncado. Recomenda-se um limite de 12 caracteres. No nosso caso, teremos `BeHappy` .
- `theme_color` : cor padrão da aplicação. Vamos utilizar o azul do cabeçalho (`#2c80b9`).
- `background_color` : este campo absorve a cor de fundo definido na estilização, porém, enquanto a página é carregada, torna a transição da cor padrão do browser para a cor de fundo do site mais suave. Manteremos o mesmo azul do campo `theme_color` .
- `display` : define o modo de exibição, aceitando os valores `fullscreen` (esconde totalmente o browser), `standalone` (apresentada em uma janela diferente, sem gerenciamento padrão de navegação do browser), `minimal-ui` (apresentada em uma janela diferente, sem alguns elementos de navegação do browser) e `browser` (apresentada no browser). Adotaremos o `fullscreen` .
- `orientation` : para definir com qual orientação (retrato ou paisagem) a aplicação vai se comportar. No nosso caso, vamos utilizar `portrait` .
- `scope` : para indicar qual escopo da página será caracterizado como uma aplicação *instalável*. Na prática, se

o usuário navegar para fora desse escopo, a aplicação volta a ser uma típica página web. Como toda nossa aplicação deve ser instalada, utilizaremos o valor `/`.

- `start_url`: indica qual página será aberta quando a aplicação for iniciada pelo ícone instalado no dispositivo. Vamos definir a página inicial (`/`).
- `icons`: um array de ícones usados pelo dispositivo para indicar a aplicação. Neste caso, é necessário mantermos ícones de tamanhos distintos para tornar nossa aplicação compatível com o maior número de sistemas operacionais possíveis. Os ícones serão baixados em breve.

Com isso, já temos subsídio para trabalhar em nosso arquivo `manifest.json`. Crie-o no diretório de distribuição da aplicação (`front-end/dist`) e entre com o código a seguir. Note que cada item do array de ícones é composto pelos campos `src`, `sizes` e `type`, que servem respectivamente para indicar o arquivo de imagem, seu tamanho e tipo.

```
{
  "name": "behappywith.me",
  "short_name": "BeHappy",
  "theme_color": "#2c80b9",
  "background_color": "#2c80b9",
  "display": "fullscreen",
  "orientation": "portrait",
  "scope": "/",
  "start_url": "/",
  "icons": [
    {
      "src": "img/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png"
    },
    {
      "src": "img/icon-96x96.png",
      "sizes": "96x96",
```

```
      "type": "image/png"
    },
    {
      "src": "img/icon-128x128.png",
      "sizes": "128x128",
      "type": "image/png"
    },
    {
      "src": "img/icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png"
    },
    {
      "src": "img/icon-152x152.png",
      "sizes": "152x152",
      "type": "image/png"
    },
    {
      "src": "img/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "img/icon-384x384.png",
      "sizes": "384x384",
      "type": "image/png"
    },
    {
      "src": "img/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

O próximo passo é baixar as imagens em https://behappywith.me/img/icons.zip. Descompacte as imagens e salve-as na pasta `front-end/src/img`. Para que o *webpack* faça a cópia dos arquivos, vamos importá-los no `front-end/src/index.jsx`. Inclua as linhas a seguir em `index.jsx`, de preferência logo abaixo da importação do `favicon.ico`.

```
import './img/icon-72x72.png';
import './img/icon-96x96.png';
import './img/icon-128x128.png';
import './img/icon-144x144.png';
import './img/icon-152x152.png';
import './img/icon-192x192.png';
import './img/icon-384x384.png';
import './img/icon-512x512.png';
```

Essa importação já é suficiente para que o `webpack` copie as imagens para a pasta de distribuição. Acrescente também uma referência nos arquivos `behappy.appcache` e `behappy.sw.js` para viabilizar o salvamento dessas imagens no *Application Cache* e no *Cache Storage*, respectivamente.

Resta-nos agora informar ao `index.html` que nossa aplicação contém um arquivo de manifesto para instalação em dispositivos mobile. Isso pode ser realizado acrescentando a sintaxe a seguir no cabeçalho (tag `<head>`) do arquivo `front-end/src/index.html`.

```
<link rel="manifest" href="/manifest.json">
```

Pronto! Por questões de segurança, o Chrome só permite realizar a instalação de PWAs fornecidas através de uma conexão HTTPS. Como os passos necessários para a criação do certificado pela entidade *Let's Encrypt* só serão apresentados no próximo capítulo, o teste real deste requisito será postergado.

Tendo um certificado autoassinado, você pode simular a instalação da aplicação via https://localhost:8080 no próprio desktop. Para isso, acesse o endereço `chrome://flags/#allow-insecure-localhost` no Google Chrome e ative o campo *Allow invalid certificates for resources loaded from localhost*. Este é um recurso avançado de desenvolvimento que o navegador oferece

para testarmos. Veja na imagem a seguir um teste realizado com este recurso.

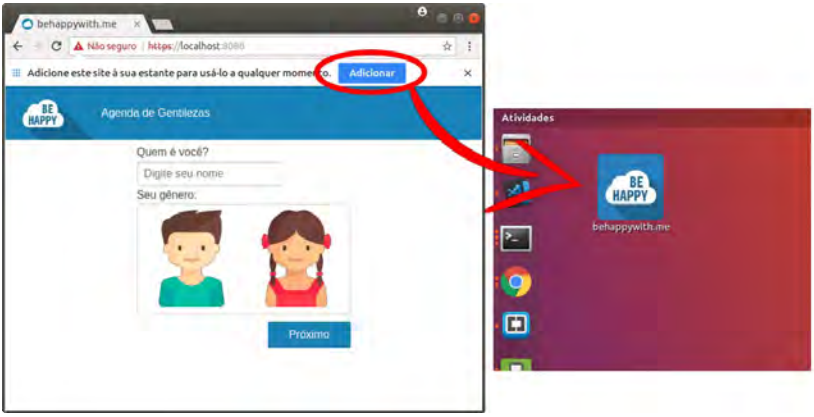

Figura 7.21: Instalação no desktop

Os passos para criar um certificado autoassinado e configurá-lo no *Nginx* serão expostos no próximo capítulo. Atenção: **desabilite** o `#allow-insecure-localhost` após os testes. Ele pode comprometer a segurança do seu computador.

A especificação da `manifest.json` ainda está evoluindo e existe uma tendência forte para padronização com os demais navegadores. Além dos itens de configuração apresentados aqui, você encontrará outros no endereço https://developer.mozilla.org/pt-BR/docs/Web/Manifest/.

Com isso, finalizamos todos os ajustes de código necessários para a construção de uma aplicação progressiva.

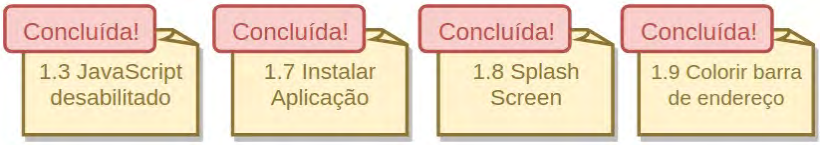

Figura 7.22: Conclusão de requisitos PWA

Os requisitos finais de carregamento rápido em internet 3G, uso do HTTPS e redirecionamento de HTTP para HTTPS serão tratados no próximo capítulo.

Arquivos de distribuição

Até então, temos trabalhado com os arquivos *compilados* em tempo de execução pelo `webpack-dev-server`. Com o código pronto, precisamos gerar os arquivos de distribuição na pasta `dist`. Pelo seu console (ou prompt), entre na pasta `behappywith.me/front-end` e execute o comando a seguir.

```
npm run build
```

Por meio do `build`, o *Webpack* compilou todos os arquivos e salvou-os em `dist` No próximo capítulo, vamos utilizá-los para publicar a aplicação no *Nginx*.

7.5 CONCLUSÃO

Neste capítulo, nos debruçamos sobre vários recursos *progressivos*. Estudamos as alternativas de salvamento de dados e salvamento de arquivos no navegador, tornando nossa página uma verdadeira aplicação *offline*. Validamos manualmente os requisitos de *cross-browser*, transições entre páginas e URLs *bookmarkable*, para mostrar que nossa aplicação está compatível com eles.

Por fim, ajustamos os últimos trechos no código-fonte para resolver os requisitos de estilização da barra de endereços, mensagem em casos de desativação do JavaScript e instalação da aplicação em dispositivos mobile (com *splash screen*). Isso nos leva à conclusão de nossa aplicação progressiva.

Para provê-la em um típico ambiente de produção seguro e com alta performance, e, ao mesmo tempo, concluir os requisitos associados ao HTTPS, no próximo capítulo, vamos configurar o poderoso servidor Nginx. Finalizaremos o livro executando o relatório do Lighthouse para confirmar que https://behappywith.me contempla todos os requisitos de uma aplicação progressiva.

Capítulo 8

PUBLICANDO A APLICAÇÃO EM PRODUÇÃO

Este capítulo pode ser dividido em duas seções: publicação da aplicação em ambiente de produção e validação dos requisitos PWA através do relatório do *Lighthouse*. A parte associada à publicação envolve a criação dos certificados (autoassinado e da entidade *Let's Encrypt*) e da instalação, e a configuração do *Nginx* para prover os arquivos de distribuição. Quando a aplicação estiver devidamente publicada, estaremos aptos a validá-la por meio do plugin *Lighthouse* (aba *Audits* do Google Chrome).

8.1 AMBIENTE DE PRODUÇÃO

O ambiente de produção de uma aplicação web pode ser sintetizado em publicar, via uma conexão segura (HTTPS), os arquivos estáticos e dinâmicos em um domínio válido. Por dinâmicos, entenda scripts ou linguagens que executam atividades no *server-side*, tais como PHP, Java, Go, JavaScript, entre outros.

Tais recursos geralmente estão associados a servidores de aplicação capazes de mantê-los *escutando* as solicitações dos

usuários. Nossa PWA **não** contém elementos de execução no servidor. Ela foi estruturada com arquivos estáticos que serão interpretados exclusivamente no *client-side*.

Então, para o nosso caso, precisamos focar em uma ferramenta que faça um provimento com alta performance e de forma segura. Como vimos no *Capítulo 2*, o servidor HTTP *Nginx* é extremamente performático e, quando combinado com um mecanismo de criptografia de chaves pública e privada, será perfeito para o provimento de https://behappywith.me.

Antes de tratarmos da instalação nos ambientes Windows e Linux, é importante destacar que o repositório do *Nginx* disponibiliza dois tipos estáveis de versão: *Mainline* e *Stable*. Em ambos os casos, pode-se considerar que a estabilidade do servidor será garantida.

A diferença é que a versão *Stable* garante compatibilidade com pacotes de terceiros. Caso você não tenha essa necessidade, poderá utilizar a versão *Mainline* sem problemas. Neste livro, vamos usar a *Stable*.

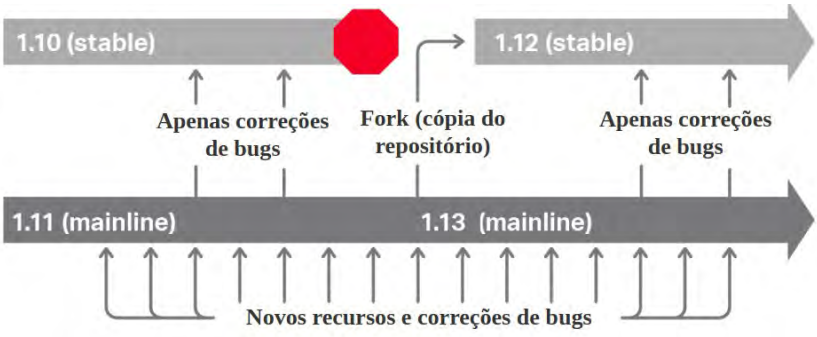

Figura 8.1: Versões do Nginx

Para realizar o provimento da aplicação de forma segura, trataremos também da criação de um par de chaves pública e privada. Serão apresentadas duas alternativas: o certificado autoassinado e o certificado real da entidade certificadora *Let's Encrypt*. Certificados autoassinados podem ser usados em ambientes de teste, porém são desencorajados em ambientes de produção.

Em termos simples, certificados desse tipo são criados sem o reconhecimento formal de uma entidade certificadora (terceiro confiável). Funciona assim: os navegadores *conhecem* uma gama de entidades certificadoras, responsáveis por certificar que uma URL provendo conteúdo HTTPS é fidedigna. Quando um navegador acessa um site seguro, ele obtém sua chave pública, valida-a com a entidade certificadora e, se tudo estiver correto, avisa o usuário de que a conexão é segura (geralmente exibindo os dados do certificado válido e um ícone do cadeado verde na barra de endereços).

Como o certificado autoassinado não possui este terceiro confiável, o navegador não é capaz de garantir sua idoneidade, mostrando ao usuário mensagens de aviso bastante agressivas, como no exemplo a seguir. Na prática, essa mensagem significa que o browser não foi capaz de confirmar que o fornecedor do site é realmente aquele cujo certificado está associado.

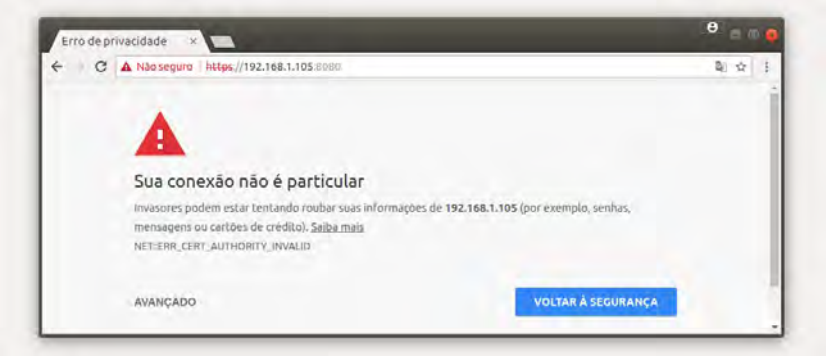

Figura 8.2: Erro de certificado HTTPS

Esse tipo de situação é contornado clicando na opção `Avançado`, e solicitando ao navegador para acrescentar o certificado em sua lista de certificados válidos. No entanto, o tratamento efetivo do problema envolve um terceiro confiável para confirmar aos browsers que nossa URL de fato está sendo provida por nós.

Existem várias entidades certificadoras de mercado consideradas válidas pelos navegadores. Neste livro, utilizaremos a *Let's Encrypt*, pela simples razão de ela prover esse tipo de serviço de maneira automatizada e gratuita. Veremos a criação dos certificados em breve, logo após os tópicos de instalação do *Nginx* nos sistemas Windows e Linux.

Instalação do Nginx no Windows

É importante começarmos este tópico ressaltando que a própria documentação oficial **não** recomenda utilizar o *Nginx for Windows* em ambientes reais de produção. Além de ser considerada uma *beta version*, é sabido que ela trabalha com um

único processo de 1024 conexões simultâneas e que a funcionalidade de *Proxy UDP* não é suportada.

Em outras palavras, o *Nginx* para Windows possui performance inferior ao do Linux e deve ser preterida. Para mais detalhes, consulte http://nginx.org/en/docs/windows.html.

Apesar das limitações expostas, a instalação do *Nginx* no Windows é bem trivial. Ela consiste apenas em baixar e descompactar o arquivo http://nginx.org/download/nginx-1.13.8.zip em um diretório de sua preferência. Aqui usaremos a pasta `C:\Users\Public\` .

Para o nosso caso, precisamos apenas saber que o executável `nginx.exe` e o arquivo de configuração `nginx.conf` estão disponíveis nos diretórios `nginx-1.13.8` e `nginx-1.13.8\conf` , nesta ordem.

Para inicializar o servidor, abra o prompt de comando, entre no diretório `C:\Users\Public\nginx-1.13.8` e execute o comando `nginx.exe` . Dependendo da versão do seu sistema, talvez seja necessário liberar o *firewall* para acessar a partir de outro dispositivo em sua rede.

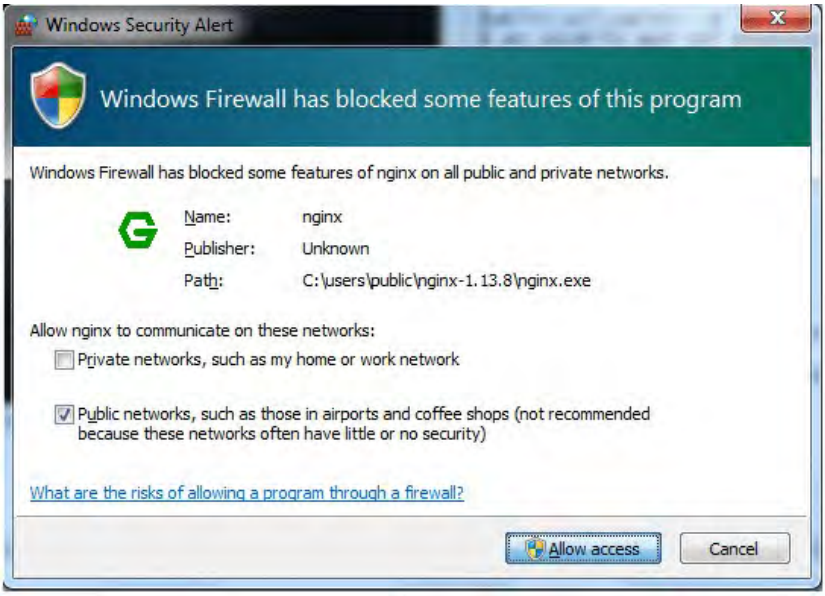

Figura 8.3: Firewall do Windows

A configuração padrão do *Nginx* publicará a página *Welcome to nginx* na porta 80 do *localhost*. Acesse-a pelo navegador para fins de validar sua instalação.

Figura 8.4: Página padrão do Nginx no Windows

Com isso, concluímos a configuração no ambiente Windows. Em breve, criaremos as configurações particulares à nossa aplicação e ajustaremos o arquivo `C:\Users\Public\nginx-1.13.8\conf\nginx.conf` utilizado neste teste.

Certificado autoassinado no Windows

O uso de certificados autoassinados **não** atende aos requisitos das aplicações progressivas. Todavia, para fins de testes locais na rede, eles podem ser úteis durante a construção do sistema. Ratifico esta informação porque, nesta obra, **não** vamos gerar um certificado real do *Let's Encrypt* para ambientes Windows.

Além das limitações já apresentadas do *Nginx for Windows*, a ferramenta de configuração *Certbot* (que é quem gera os certificados do *Let's Encrypt*) atualmente só oferece suporte à instalação em sistemas *UNIX-like*. Para mais detalhes, acesse o endereço https://certbot.eff.org/.

O Windows oferece nativamente mecanismos automatizados para gerenciamento de certificados (`certmgr.msc`), porém, como este conteúdo excede a proposta deste livro, utilizaremos o aplicativo `openssl` . No endereço http://gnuwin32.sourceforge.net/packages/openssl.htm/, você encontrará, além de outros recursos, um ZIP com os binários para execução.

Descompacte-o no diretório `C:\Users\Public\` . Como este binário não possui o arquivo de configuração do `openssl` , vamos baixar uma configuração padrão pelo endereço https://www.tbs-certificates.co.uk/openssl-dem-server-cert-thvs.cnf/. Salve o arquivo em `C:\Users\Public\openssl-0.9.8h-1-bin\` com o nome `openssl.cfg` . Cuidado porque o Windows costuma incluir uma extensão `.txt` ao final do arquivo. Se for o caso, retire-a.

Sugiro salvar as chaves pública e privadas respectivamente nos diretórios `C:\Users\Public\ssl\certs\` e `C:\Users\Public\ssl\private\` . Estreite os acessos da pasta `private` apenas ao administrador e ao usuário que vai executar o *Nginx*. Abra o prompt de comando, entre na pasta `C:\Users\Public\openssl-0.9.8h-1-bin\bin` e execute a sequência a seguir.

```
Set OPENSSL_CONF=C:\Users\Public\openssl-0.9.8h-1-bin\openssl.cfg
openssl.exe req -x509 -new -newkey rsa:2048 -nodes -keyout C:\Use
rs\Public\ssl\private\behappy-private.key -out C:\Users\Public\ss
l\certs\behappy-public.pem -days 90
```

Este comando vai questioná-lo sobre os dados de nossa aplicação. Por ser autoassinado, você pode responder todas as questões com o valor default (apenas digitando `ENTER`). Se tudo

der certo, você verá algo semelhante ao resultado a seguir.

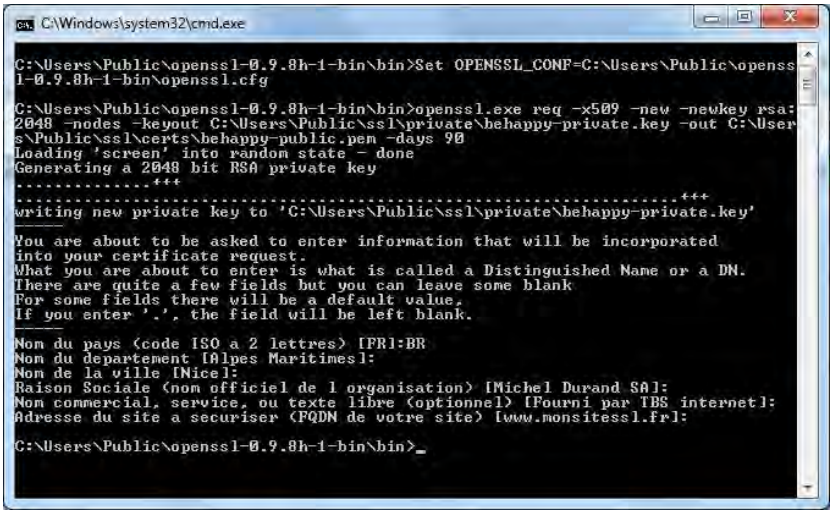

Figura 8.5: Criação do certificado autoassinado

Com isso, já temos nossas chaves `behappy-public.pem` e `behappy-private.key` com validade de 90 dias. Em breve, vamos referênciá-las na configuração *Nginx* para a nossa PWA.

Instalação do Nginx no Linux

Os sistemas operacionais *UNIX-like* são perfeitos para embarcar o *Nginx*, provendo recursos para deixá-lo performático e confiável. Para instalá-lo, vamos utilizar a ferramenta `apt-get`, começando pelo passo de assinatura do repositório do *Nginx*.

Em sistemas *Debian-like*, às vezes é necessário informar ao `apt-get` a chave usada para assinar determinados pacotes que serão baixados. Isso evita problemas e *warnings* durante a instalação. Como neste livro estamos trabalhando com o *Ubuntu*

(que é um sistema baseado no *Debian*), vamos precisar desses passos.

Seguindo as boas práticas, crie uma pasta `/opt/nginx` e, a partir dela, baixe a chave de assinatura do *Nginx* e adicione-a no `apt-get`. Note que, nos comandos a seguir, estamos explicitamente repassando a propriedade do diretório `/opt/nginx` para o usuário `guilherme`. Ajuste este trecho para o seu usuário.

```
sudo mkdir /opt/nginx
sudo chown guilherme:guilherme /opt/nginx/
cd /opt/nginx/
wget http://nginx.org/keys/nginx_signing.key
sudo apt-key add nginx_signing.key
```

Com a chave adicionada, vamos agora informar ao `apt-get` em quais repositórios ele encontrará o *Nginx*. Neste procedimento, você precisará informar também o `codename` da sua distribuição Ubuntu. Caso você não saiba, use o comando `lsb_release -a` no console para recuperá-la.

Veja um exemplo na imagem a seguir, caso em que o `codename` é `xenial`. Outros `codename` comuns à distribuição Ubuntu são `trusty` e `zesty`. Os endereços IP exibidos no terminal foram omitidos por questões de segurança.

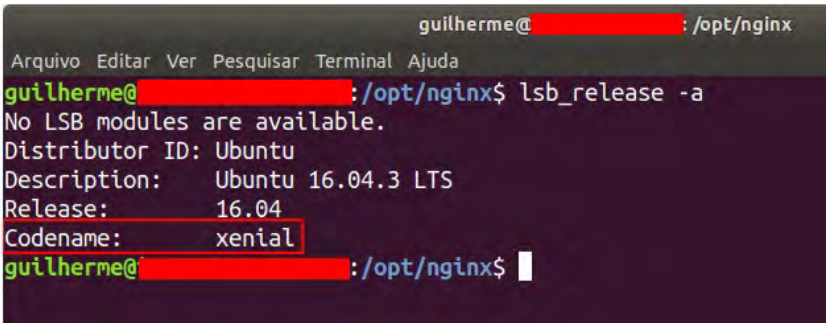

Figura 8.6: Verificando codename do Ubuntu

De posse do `codename` e com um editor de texto de sua preferência, adicione as linhas a seguir no final do arquivo `/etc/apt/sources.list` (lista de repositórios do `apt-get`). Atenção: altere as strings `xenial` do trecho a seguir pelo `codename` do seu sistema.

```
deb http://nginx.org/packages/ubuntu/ xenial nginx
deb-src http://nginx.org/packages/ubuntu/ xenial nginx
```

Com essas configurações prontas, resta-nos efetivamente instalar o *Nginx*.

```
sudo apt-get update
sudo apt-get install -y nginx
```

Pronto. Respectivamente, para inicializar e desligar o servidor HTTP *Nginx*, execute os comandos a seguir.

```
sudo service nginx start
sudo service nginx stop
```

Assim, terminamos a instalação do *Nginx* no servidor Linux. Como ainda não tratamos das configurações relacionadas à nossa PWA, ao acessarmos o servidor, veremos apenas a página padrão.

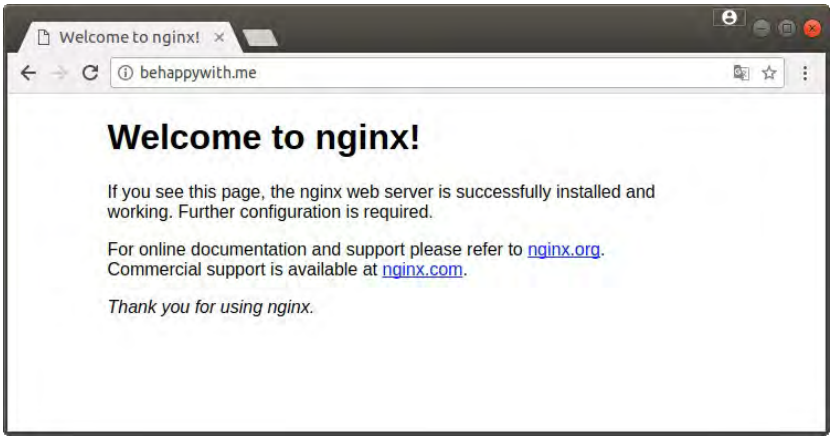

Figura 8.7: Página padrão do Nginx no servidor Linux

Nos próximos tópicos, criaremos os certificados HTTPS no sistema Linux.

Certificado autoassinado no Linux

Atente-se que, caso você esteja configurando um ambiente de produção real, sugiro fortemente a criação das chaves pela entidade *Let's Encrypt*, conforme exposto no próximo tópico. Por outro lado, se seu propósito é testar o uso do HTTPS em sua máquina, prossiga com os passos a seguir.

No ambiente Linux, as boas práticas indicam que a chave privada deve ser armazenada no diretório `/etc/ssl/private`, enquanto a chave pública no diretório `/etc/ssl/certs`. Abra seu terminal e execute o comando a seguir (como `root`).

```
openssl req -x509 -new -newkey rsa:2048 -nodes -keyout /etc/ssl/p
rivate/behappy-private.key -out /etc/ssl/certs/behappy-public.pem
 -days 90
```

Você precisará responder algumas perguntas sobre a organização que está provendo o site. Não há respostas inválidas em certificados autoassinados. Em certificados verdadeiros, todavia, você precisa responder tais perguntas com informações válidas. Note que esse certificado perderá sua validade em 90 dias.

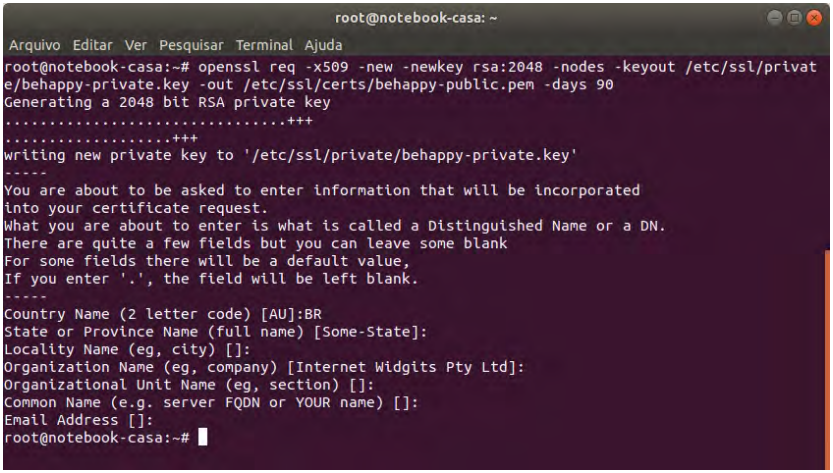

Figura 8.8: Criação do certificado autoassinado

Com os comandos a seguir, certifique-se de que o usuário executor do servidor *Nginx* deve possuir permissão de leitura.

```
sudo chmod o+rx /etc/ssl/private/
sudo chmod o+r /etc/ssl/private/behappy-private.key
sudo chmod o+r /etc/ssl/certs/behappy-public.pem
```

Nossas chaves públicas e privadas autoassinadas já estão devidamente criadas.

Certificado Let's Encrypt no Linux

Antes de você se debruçar sobre os parágrafos deste tópico,

lembre-se de que a criação de um certificado válido está condicionada a um domínio válido (por exemplo, https://behappywith.me) apontando para o endereço IP do servidor cuja aplicação está sendo provida. A critério de entendimento, nossa aplicação está publicada em um servidor na nuvem AWS da Amazon (https://aws.amazon.com/pt/), com o DNS (*Domain Name System*, responsável por redirecionar o domínio para o IP do servidor) e o próprio domínio registrados no Hostgator (https://www.hostgator.com.br/).

Caso você queira se aventurar com a criação de servidores web na nuvem, sugiro a leitura de *Amazon AWS: Descomplicando a computação na nuvem*, um livro de Jonathan Lamim Antunes publicado pela *Casa do Código* (https://www.casadocodigo.com.br/products/livro-amazon-aws).

A entidade *Let's Encrypt* tem como premissa viabilizar a criação dos certificados de forma automatizada. Ela trabalha com um aplicativo chamado *Certbot*, o cliente oficial que o *Let's Encrypt* oferece para obter e publicar os certificados SSL/TLS. Ele pode ser baixado no endereço https://dl.eff.org/certbot-auto/ e será nosso ponto de partida nesta configuração.

Seguindo as boas práticas, como se trata de um aplicativo de terceiros, crie um diretório `/opt/certbot` , altere o *dono* do diretório para seu usuário de trabalho no Linux, salve o `certbot-auto` e atribua a ele uma permissão de execução. No script a seguir, foi utilizado o usuário `guilherme` .

```
sudo mkdir /opt/certbot
sudo chown guilherme:guilherme /opt/certbot/
cd /opt/certbot/
wget https://dl.eff.org/certbot-auto
```

```
chmod a+x certbot-auto
```

O próximo passo é executá-lo. Atenção: o `certbot-auto` precisará das portas 80 e 443 disponíveis para validar seu domínio e criar os certificados. Caso você tenha outro servidor usando tais portas (por exemplo, o próprio *Nginx*), a execução do `certbot-auto` falhará.

Outro detalhe é que ambas as portas precisarão estar abertas no firewall do seu servidor, e acessíveis através dos endereços http://behappywith.me (porta 80) e https://behappywith.me (porta 443). Supondo que você esteja no diretório `/opt/certbot/`, execute o comando a seguir. Como o `certbot-auto` fará algumas instalações, o usuário utilizado precisar estar no grupo de *sudoers* (usuários que podem executar comandos como se fossem administradores) ou ser o próprio `root`.

```
./certbot-auto certonly --standalone -d behappywith.me
```

Durante o processo, você deverá informar um e-mail (para receber notificações sobre este certificado), aceitar os termos de uso (https://acme-v01.api.letsencrypt.org/directory), e permitir ou não o cadastro do seu e-mail na EFF (*Electronic Frontier Foundation*, fundação sem fins lucrativos responsável pela criação do `certbot-auto`). Não existe em aceitar ambos os termos, pois a EFF só vai enviar um e-mail explicando sua missão de criptografar as páginas da internet.

Ao final do processo, você verá uma tela semelhante à apresentada a seguir. O endereço IP do servidor foi omitido por questões de segurança.

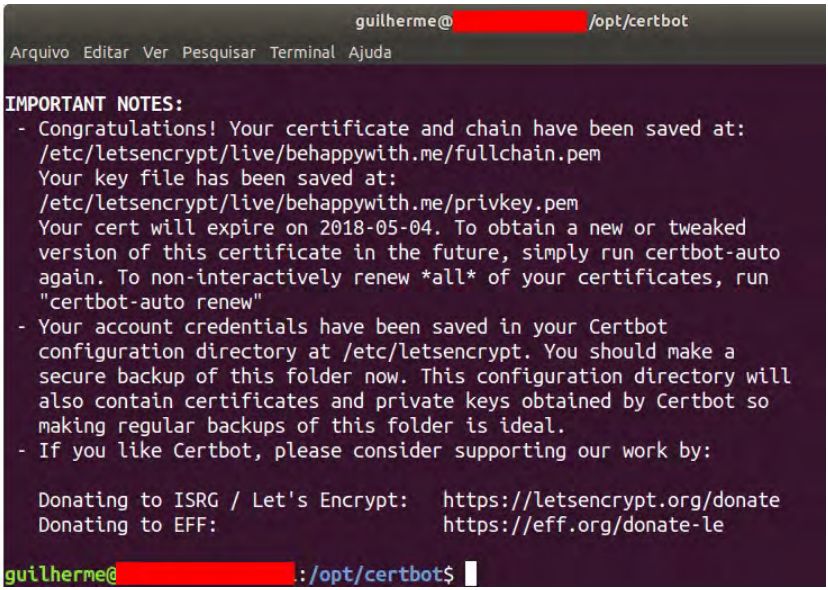

Figura 8.9: Criação do certificado Let's Encrypt

Como pode ser visualizado na imagem anterior, suas chaves pública (`fullchain.pem`) e privada (`privkey.pem`) foram criadas no diretório `/etc/letsencrypt/live/behappywith.me/` . Para seguir as boas práticas, vamos copiá-las para os `/etc/ssl/certs` e `/etc/ssl/private` , e renomeá-las para algo mais adequado ao projeto.

Como `root` , execute o trecho a seguir. Atenção: os comandos de cópia apresentados vão falhar caso você tenha gerado os certificados autoassinados no tópico anterior. Se quiser manter ambos, sugiro ajustar o nome dos arquivos conforme sua necessidade. Os três últimos comandos servem para permitir que *outros* usuários tenham acesso aos certificados.

```
cd /etc/letsencrypt/live/behappywith.me/
cp fullchain.pem /etc/ssl/certs/behappy-public.pem
cp privkey.pem /etc/ssl/private/behappy-private.key
chmod o+rx /etc/ssl/private/
chmod o+r /etc/ssl/private/behappy-private.key
chmod o+r /etc/ssl/certs/behappy-public.pem
```

Pronto, nossas chaves foram criadas. Uma observação importante é que este certificado possui uma validade de apenas 90 dias. Este é o prazo dos certificados do *Let's Encrypt*. Para contornar essa limitação, é comum configurar uma rotina automática no Linux (via `crond`) para executar o comando `certbot-auto renew` entre intervalos de tempo regulares.

Desta forma, sempre que o certificado estiver próximo do vencimento, a ferramenta `certbot-auto` fará seu trabalho de renovação. Esta configuração não faz parte do escopo deste livro. Para mais detalhes, acesse https://certbot.eff.org/docs/.

Configuração do Nginx

A última etapa da publicação de https://behappywith.me em produção tratará da criação do arquivo de configuração do *Nginx*. Como você perceberá ao longo dos próximos parágrafos, o mesmo arquivo pode ser utilizado nos ambientes Windows e Linux, salvo pequenas exceções que serão apontadas no momento oportuno.

Até então, todos os esforços foram concentrados na elaboração dos arquivos sob a pasta `front-end`. A partir de agora, como estaremos trabalhando na configuração do servidor, o código apresentado estará sob o diretório `back-end`.

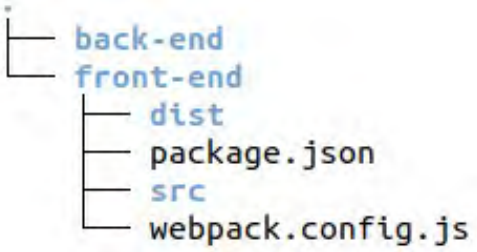

Figura 8.10: Estrutura de diretórios

O *Nginx* possui um portfólio muito extenso de sintaxes para configuração. Não é intenção deste livro abordar todas as possibilidades. Introduziremos apenas os comandos necessários à publicação da nossa PWA. A listagem a seguir apresenta alguns itens que precisamos perseguir:

- Redirecionamento do HTTP para o HTTPS;
- Utilização do HTTPS;
- Log dos acessos no servidor;
- Publicação dos arquivos estáticos;
- Definição do *MIME Type* específico para o arquivo `behappy.appcache` ;
- URLs não encontradas devem ser redirecionadas para a página principal.

Para relembrá-lo, esses dois últimos itens servem respectivamente para publicar corretamente o arquivo do *Application Cache* (que deve ter seu *MIME Type* como `text/cache-manifest`) e para viabilizar as URLs *bookmarkable* através do *React Router*. Apesar dessas particularidades intrínsecas às aplicações progressivas, o arquivo de configuração em si será bem simples – como veremos a partir de agora.

Na sua IDE, crie um arquivo chamado `default.conf` dentro de `behappywith.me/back-end`. Dentro dele, inclua dois trechos do bloco `server`, conforme a seguir.

```
server {
}

server {
}
```

Existem dois tipos de configuração no *Nginx*: *simple directives* (diretivas simples) e *block directives* (diretivas de bloco). As diretivas simples são aquelas usadas para definir propriedades com um nome e um valor. Esse tipo de diretiva pode aceitar um ou mais valores separados por espaços em branco e devem ser finalizadas com um `;` (ponto e vírgula).

As diretivas de bloco possuem uma sintaxe com `{}` (chaves) capaz de manter uma série de configurações de um mesmo contexto. A sintaxe `server` apresentada no trecho anterior é do tipo *block directives* e é comumente chamada de *Server Block*.

Como apresentado, o arquivo `default.conf` pode conter mais de um *Server Block*. Cada *Server Block* mantém as configurações de uma aplicação publicada no *Nginx*. Para que o servidor detecte qual delas deve ser usada, existem duas *simple directives* cujos nomes são `listen` e `server_name`.

O campo `listen` define o endereço IP e a porta que serão comparados aos da requisição (*Request*). Se todos os *Server Blocks* possuírem o mesmo endereço IP, podemos omiti-lo e utilizar somente a porta. O campo `server_name` identifica um ou mais *hostnames* (por exemplo, `behappywith.me`) do *Server Blocks*. Com ambos os campos, o *Nginx* será capaz de indicar qual *Server*

Blocks tratará da requisição do usuário.

No nosso caso, o primeiro *Server Block* tratará das solicitações ao domínio `behappywith.me` pela porta 443 (HTTPS), enquanto o segundo receberá as solicitações de `behappywith.me` na porta 80.

```
server {
    listen 443 ssl;
    server_name behappywith.me www.behappywith.me;
}

server {
    listen 80;
    server_name behappywith.me www.behappywith.me;
}
```

Note o uso da sintaxe `ssl` no primeiro bloco. Ela serve para garantir que este vai funcionar exclusivamente sob o protocolo SSL. Note também que definimos os domínios `behappywith.me` e `www.behappywith.me` para ambas configurações. Isso serve para capturar as requisições cujo prefixo `www` tenha sido informado.

Atenção: se você estiver configurando esse arquivo no sistema Windows, lembre-se de que o uso em ambiente de produção não é recomendado. Sugiro que você substitua o trecho `behappywith.me www.behappywith.me` apenas por `localhost`. Só assim você poderá publicar nossa PWA localmente pelo *Nginx*.

O segundo bloco, responsável pelas requisições à porta 80, servirá como ponto de redirecionamento ao bloco SSL. Vamos ajustá-lo para retornar ao browser a informação de que houve um **redirecionamento permanente** para o endereço protegido.

Como pode ser visualizado a seguir, isso será facilmente

configurado pelo código de status `301` (*301 Moved Permanently*) seguido do endereço correto. O *Server Block* da porta 443 foi omitido. Caso esteja utilizando o Windows, troque a sintaxe `return 301 https://behappywith.me` por `return 301 https://localhost;` .

```
server {
    listen 80;
    server_name behappywith.me www.behappywith.me;
    return 301 https://behappywith.me;
}
```

Com este *Server Block*, resolvemos o requisito 1.5, que trata do redirecionamento de HTTP para HTTPS. Daqui em diante, todas as configurações serão centradas no *Server Block* do SSL.

Faremos agora a configuração do HTTPS. Em primeiro plano, informaremos a localização das chaves pública e privada através das sintaxes `ssl_certificate` e `ssl_certificate_key` . Em seguida, vamos restringir os protocolos SSL aceitos com a sintaxe `ssl_protocols` , e o nível de criptografia usado com `ssl_ciphers` .

Este código está apontando para o diretório no Linux. Para adaptá-lo ao ambiente Windows, substitua os paths das chaves pública e privada por `C:/Users/Public/ssl/certs/behappy-public.pem` e `C:/Users/Public/ssl/private/behappy-private.key` , respectivamente. Observe que, mesmo no Windows, o *Nginx* trabalha com o separador `/` (barra) em vez de `\` (barra invertida).

```
server {
    listen 443 ssl;
    server_name behappywith.me www.behappywith.me;
```

```
    ssl_certificate /etc/ssl/certs/behappy-public.pem;
    ssl_certificate_key /etc/ssl/private/behappy-private.key;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers HIGH:!aNULL:!MD5;
}
```

O trecho `ssl_protocols` permite somente algumas versões do protocolo *Transport Layer Security* que, como vimos no *Capítulo 1*, é uma evolução da especificação *Secure Sockets Layer*, mais conhecida por seu acrônimo *SSL*. A sintaxe de `ssl_ciphers` obriga grandes chaves de criptografia pelo uso da string `HIGH`, proíbe conexões sem autenticação com `!aNULL`, e proíbe o uso do protocolo `MD5`.

Conexões sem autenticação são suscetíveis a ataques *man-in-the-middle*. O protocolo `MD5` é vulnerável ao ataque de força bruta e contém colisões conhecidas. Para mais detalhes, veja https://www.openssl.org/docs/manmaster/man1/ciphers.html#CIPHER_LIST_FORMAT.

Uma ação importante para servidores HTTP é *logar* os acessos em um arquivo. O *Nginx* trata desta configuração através da sintaxe `access_log` definida com o nome do arquivo de log. No Linux, arquivos como esse são salvos em um subdiretório de `/var/log` que, no nosso caso, será `/var/log/behappy/access.log`.

Com poder de `root`, crie o diretório `/var/log/behappy/` e repasse sua propriedade ao usuário `nginx` (todas as requisições tratadas pelo *Nginx* serão interpretadas com o usuário `nginx`):

```
sudo mkdir /var/log/behappy/
sudo chown nginx:nginx /var/log/behappy/
```

Com esses detalhes esclarecidos, a configuração em si é muito

simples. Ajuste o *Server Block* 443 como no trecho a seguir.

```
server {
    listen 443 ssl;
    server_name behappywith.me www.behappywith.me;

    ssl_certificate /etc/ssl/certs/behappy-public.pem;
    ssl_certificate_key /etc/ssl/private/behappy-private.key;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers HIGH:!aNULL:!MD5;

    access_log /var/log/behappy/access.log;
}
```

No Windows, a localização de arquivos de log pode variar de acordo com a versão do sistema – no 2008 Server: `C:\Users\user\AppData\Local\Programs\` , no 2003 Server: `C:\Documents and Settings\user\Application Data\` etc. Para simplificar, vamos criar uma pasta `log` no diretório `C:\Users\Public\behappywith.me\` . Isso significa que você deve substituir o caminho `/var/log/behappy/access.log` do código anterior por `C:/Users/Public/behappywith.me/log/access.log` .

As próximas configurações estarão relacionadas ao provimento dos arquivos estáticos. O *Nginx* criará uma correspondência entre a URL da solicitação e os blocos da sintaxe `location` de cima para baixo. Para calcular a correspondência (*matching*), esta sintaxe aceita strings com prefixos das URLs ou expressões regulares.

Conhecendo essas regras, entende-se que uma boa configuração para nossa PWA será tratar primeiramente das URLs mais restritivas (como o *MIME Type* para o arquivo `behappy.appcache`) e, em seguida, das URLs gerais. Isso fará

com que o fluxo de interpretação do *Nginx* tente primeiro resolver os casos mais pontuais antes de partir para o provimento dos arquivos comuns.

O primeiro passo é definir a diretiva `root`, que serve para especificar a localização dos arquivos publicados. Se a localização dos arquivos providos pelos blocos `location` fossem diferentes, teríamos de acrescentar uma diretiva `root` para cada situação.

No nosso caso, como todos os arquivos estão localizados em `front-end/dist`, criaremos uma única `root` no escopo do *Server Block*. Coloque a sintaxe a seguir abaixo do trecho do log.

```
root /opt/repositories/github/behappywith.me/front-end/dist;
```

Observe que, neste exemplo, a pasta `behappywith.me/ront-end/dist` está guardada em `/opt/repositories/github/`. Essa organização **não** é obrigatória. Altere-a para o seu caso. Não se esqueça, entretanto, que o usuário `nginx` precisa de acesso de leitura a esse diretório.

```
sudo chmod o+rx /opt/repositories/github/behappywith.me/front-end
/dist
```

Se você estiver no Windows, em vez da diretiva `root` apresentada anteriormente, utilize o trecho a seguir.

```
root C:/Users/Public/behappywith.me/front-end/dist;
```

Vamos acrescentar agora o caso menos abrangente do *MIME Type* para o arquivo `behappy.appcache`. Como essa `location` fará o *matching* a partir de uma URL fixa, vamos utilizar o símbolo `=` (igual) seguido de `/behappy.appcache`. Dentro do bloco, teremos apenas a sintaxe `default_type` para indicar o *MIME Type*. Inclua o trecho a seguir abaixo da sintaxe `root`. O resto do

arquivo foi omitido.

```
location = /behappy.appcache {
    default_type text/cache-manifest;
}
```

O último bloco de configuração `location` vai capturar todos os acessos direcionados ao `/` (raiz da aplicação). Ela terá duas instruções: `index` e `try_files`. O `index` será usado para informar o nome do arquivo principal do sistema. O `try_files` fará o redirecionamento das URLs não encontradas para o `index.html`. Isso evitará respostas do tipo *404 Not Found* e viabilizará a criação das rotas pelo *React Router*.

```
location / {
    index index.html;
    try_files $uri $uri/ /index.html;
}
```

Com isso, cobrimos todas as configurações necessárias à nossa PWA. No próximo tópico, que finalizará nossa conversa sobre o *Nginx*, veremos duas pequenas melhorias que podem ser aplicadas para otimizar o provimento do site: a compactação dos arquivos e o gerenciamento explícito do `favicon.ico`.

Últimos ajustes da configuração

Neste tópico, conversaremos sobre duas práticas bem quistas pela comunidade de software quando se trata do provimento de arquivos estáticos pelo *Nginx*. A saber:

- Compactação dos arquivos pelo *gzip*;
- Gerenciamento explícito do arquivo `favicon.ico`.

Compactar arquivos no *server-side* é uma excelente prática

para redução do *overhead* envolvido no download dos arquivos pelo browser. Há métricas na literatura (BUTLER, 2017) que apontam uma redução de até 70% no tamanho dos arquivos, fato mais que suficiente para adotarmos essa técnica mesmo sabendo que haverá um uso extra de CPU do servidor.

O *Nginx* já vem integrado com o *gzip*. Dessa forma, nossa única tarefa será baixar o arquivo de configuração `gzip.conf` no *GitHub* do projeto e salvá-lo na pasta `/etc/nginx/conf.d/` . **Atenção:** esta configuração **não** funcionará corretamente no sistema operacional Windows. Se for o seu caso, não a execute.

Supondo que você esteja em `behappywith.me/back-end` , execute os passos a seguir. Note que, em vez de copiarmos o arquivo, vamos criar um *hard-link* no Linux pelo comando `ln` .

```
wget https://raw.githubusercontent.com/lgapontes/behappywith.me/m
aster/back-end/gzip.conf
sudo chown o+r gzip.conf
sudo ln -f gzip.conf /etc/nginx/conf.d/gzip.conf
```

Pronto, isso já é suficiente para compactar os arquivos providos.

A última configuração diz respeito ao tratamento exclusivo do arquivo `favicon.ico` no arquivo de configuração do *Nginx*. Isso pode parecer um excesso de zelo, mas, em sites reais, o tratamento incorreto `favicon.ico` pode gerar uma série de problemas. O caso mais famoso foi o da semana de abertura do site *Instagram*, que, por falha no provimento deste ícone, gerou problemas de performance no `controller` do *Django* (um framework do *Python*, semelhante ao *Rails* do *Ruby*).

As regras para evitar esse tipo do problema são:

- Informar ao navegador o caminho correto do `favicon.ico` pela tag `<link>`;
- Desligar as mensagens de log do `GET` para `favicon.ico`.

O primeiro ponto foi resolvido no *Capítulo 4*. Para resolver o segundo, vamos acrescentar um bloco `location` no arquivo do *Nginx* (`default.conf`) desligando os logs de sucesso e falha por meio das sintaxes `access_log off` e `log_not_found off`, respectivamente.

Coloque este bloco logo abaixo da diretiva `root`, conforme apresentado a seguir. Como esta é nossa última configuração, vou apresentar o conteúdo completo do arquivo `default.conf` a seguir.

```
server {
    listen 443 ssl;
    server_name behappywith.me www.behappywith.me;

    ssl_certificate /etc/ssl/certs/behappy-public.pem;
    ssl_certificate_key /etc/ssl/private/behappy-private.key;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers HIGH:!aNULL:!MD5;

    access_log /var/log/behappy/access.log;

    root /opt/repositories/github/behappywith.me/front-end/dist;

    location /img/favicon.ico {
        access_log off;
        log_not_found off;
    }
    location = /behappy.appcache {
        default_type text/cache-manifest;
    }
    location / {
        index index.html;
        try_files $uri $uri/ /index.html;
    }
```

```
}

server {
    listen 80;
    server_name behappywith.me www.behappywith.me;
    return 301 https://behappywith.me;
}
```

E assim fechamos os conceitos envolvidos na publicação da PWA em produção.

Copiando o default.conf para o Nginx

Resta-nos copiar o `default.conf` criado para o *Nginx*. Caso precise, você encontrará duas versões deste arquivo preparadas para os ambientes Linux e Windows no *GitHub* do projeto. Veja os links adiante.

- Linux: https://raw.githubusercontent.com/lgapontes/behappywith.me/master/back-end/default.conf_linux
- Windows: https://raw.githubusercontent.com/lgapontes/behappywith.me/master/back-end/default.conf_windows

Se você estiver no Windows, baixe o arquivo, altere seu nome para `default.conf` e salve-o em `C:\Users\Public\behappywith.me\back-end\` . O *Nginx for Windows* possui um arquivo chamado `nginx.conf` , localizado em `C:\Users\Public\nginx-1.13.8\conf\` . Este já possui uma publicação default de página estática (aquela que visualizamos no tópico de instalação no Windows).

Para fazê-lo reconhecer a nossa PWA, vamos alterar esse

arquivo (`nginx.conf`) com o intuito de informar os *Server Blocks* criados nos tópicos anteriores. Abra o arquivo `nginx.conf` e, logo na primeira linha do bloco `http` , codifique a sintaxe a seguir.

```
# ...
http {
    include C:/Users/Public/behappywith.me/back-end/default.conf;

    # ...
}
```

Para que o nosso *Server Block* **não** conflite com o padrão definido nesse arquivo, exclua (ou comente) o `server` de `localhost` presente neste mesmo bloco `http` . Pronto! Supondo que a pasta `front-end` esteja localizada em `C:\Users\Public\behappywith.me\front-end\dist` , execute o arquivo `C:\Users\Public\nginx-1.13.8\nginx.exe` e veja nossa PWA rodando em `https://localhost` .

Se você estiver no Linux, baixe o arquivo a partir do diretório `behappywith.me/back-end/` , faça um backup do arquivo original e crie um *hard-link* do arquivo para `/etc/nginx/conf.d/` . Isso está traduzido nos comandos adiante.

```
wget https://raw.githubusercontent.com/lgapontes/behappywith.me/m
aster/back-end/default.conf_linux
mv default.conf_linux default.conf
sudo mv /etc/nginx/conf.d/default.conf /etc/nginx/conf.d/default.
conf_backup
sudo chown o+r default.conf
sudo ln -f default.conf /etc/nginx/conf.d/default.conf
```

Para inicializar e desligar o servidor *Nginx* no sistema operacional Linux, execute os respectivos comandos.

```
sudo service nginx start
```

```
sudo service nginx stop
```

Assim, concluímos os últimos requisitos de nossa PWA. Apesar de não explicitarmos o tratamento do requisito 1.6, as otimizações realizadas no `build` do *Webpack*, a performance do React, a construção de imagens em Sprites, e a otimização do *Nginx* com *gzip* são os fatores que contribuíram para sua resolução.

Na prática, nosso tratamento foi ainda mais agressivo do que o tempo de referência usado na validação do *Lighthouse* (10 segundos). Como vimos no *Capítulo 1*, nossa página está preparada para ser carregada em no máximo 3 segundos (*three seconds and go*). Isso pode ser confirmado na aba *Network* do Google Chrome, através de uma opção de simulação de velocidade. Como apresentado na imagem a seguir, ao utilizar a opção *Fast 3G*, a página foi carregada em 2.29 segundos.

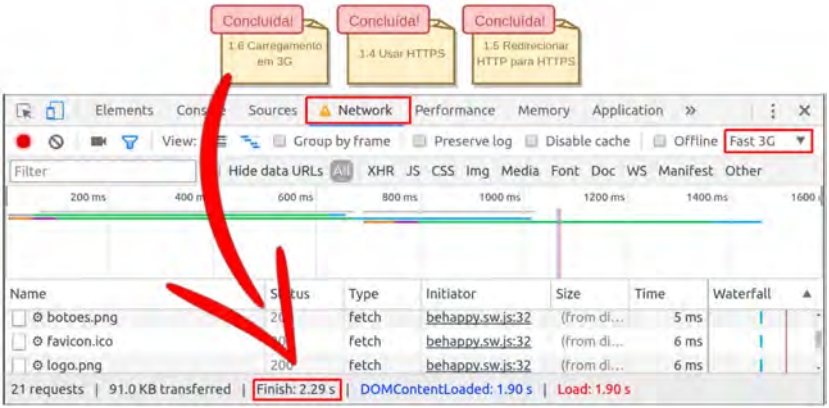

Figura 8.11: Requisitos concluídos

No próximo tópico, rodaremos o relatório do Lighthouse.

8.2 RELATÓRIO DO LIGHTHOUSE

Como estudamos no *Capítulo 1*, o Lighthouse é uma ferramenta de auditoria que nos oferece um relatório para quantificar o quanto uma página se aproxima dos requisitos progressivos. Há duas formas de instalá-lo: via uma extensão disponível até a versão 59 do Google Chrome, ou pela aba *Audits* disponível no Google Chrome a partir da versão 60.

Caso você esteja com uma versão anterior à 60 e por algum motivo não queira atualizá-la, acesse o menu de opções do Chrome (três pontinhos) no canto superior direito, submenu *Mais ferramentas*, opção *Extensões*. Na página de gerenciamento das extensões, clique em *Obter mais extensões*, consulte pela string *lighthouse* e, por fim, instale a extensão encontrada no navegador. Após esta instalação, a aba *Audits* estará disponível.

Figura 8.12: Extensões do Google Chrome

Segundo cenário: a partir da atualização 60 do Google Chrome, o Lighthouse já vem integrado através da aba *Audits*. Caso você esteja utilizando uma versão igual ou superior, não será necessário executar os passos de instalação supracitados.

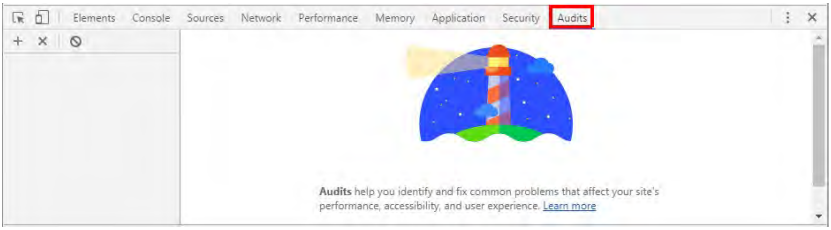

Figura 8.13: Lighthouse integrado ao Chrome

Para gerar o relatório, acesse a página de nossa aplicação publicada no servidor de produção (https://behappywith.me), entre na aba *Audits* do DevTools e clique no botão *Perform an audit*. Neste momento, o Lighthouse vai abrir um popup exibindo os tipos de validação para que você selecione.

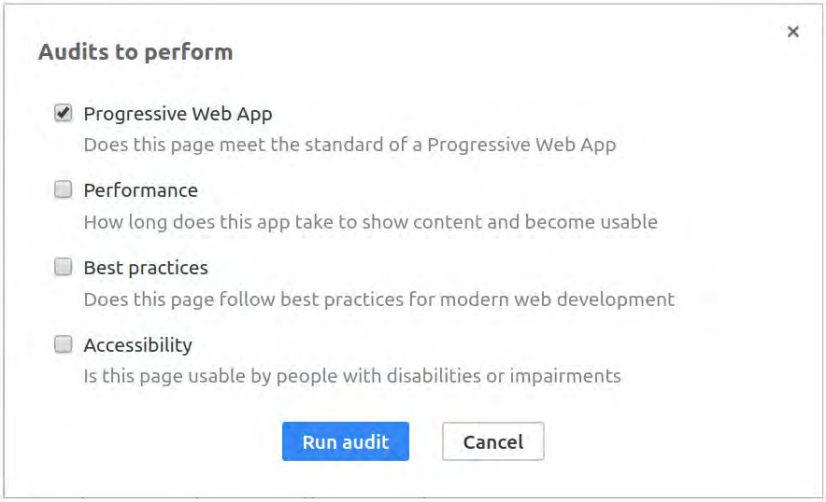

Figura 8.14: Opções de auditoria

Para o nosso teste, selecione a opção *Progressive Web App* e clique em *Run audit*. Não faz parte do escopo deste livro discutir as outras opções deste relatório. Todavia, caso queira mais detalhes,

acesse a documentação oficial em https://developers.google.com/web/tools/lighthouse/?hl=pt-br.

Após clicar no botão *Run audit*, aguarde alguns instantes até que a auditoria seja concluída. Note que Lighthouse executa uma espécie de emulador mobile internamente no navegador e destaca (abaixo do ícone de *loading*) quais requisitos PWA estão sendo validados no momento.

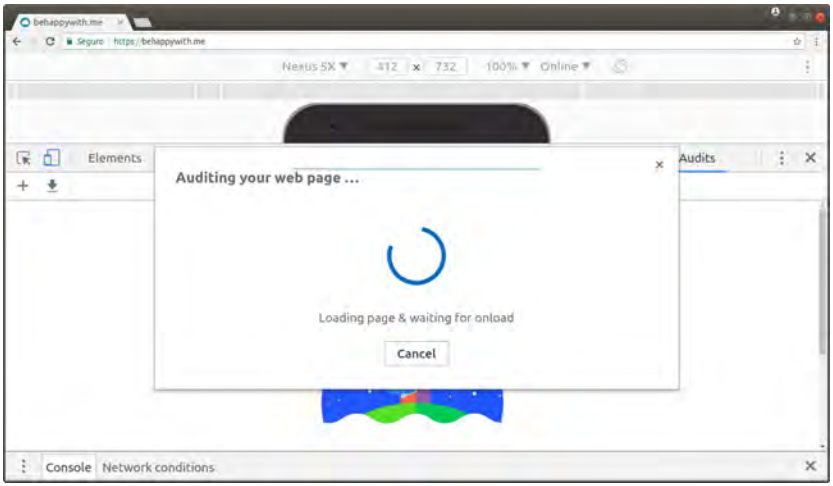

Figura 8.15: Lighthouse realizando sua auditoria

Após a execução, ele vai exibir um checklist com todos os requisitos validados, destacando-os com ícones de falha ou sucesso. No topo superior direito do relatório, será apresentado um percentual informando o quanto sua aplicação é tida como progressiva. Veja que nossa PWA https://behappywith.me foi quantificada com 100%.

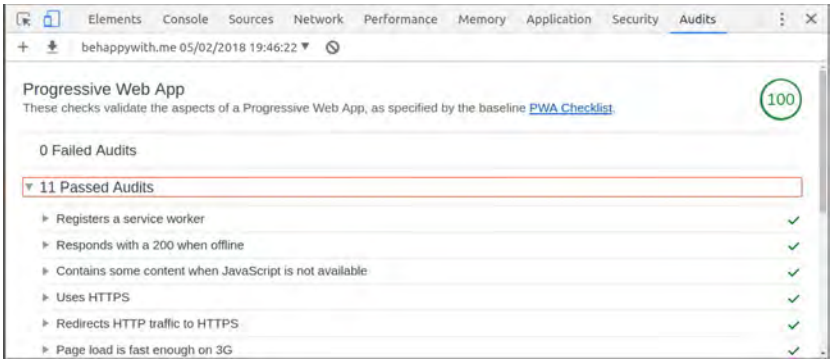

Figura 8.16: Relatório final do Lighthouse

Além desta avaliação técnica, é importante finalizarmos nosso estudo acessando o endereço https://behappywith.me a partir de um dispositivo mobile para visualizarmos os requisitos estudados e a componentização React em ação.

Da esquerda para a direita, a primeira tela exibe o HTTPS (validado pela entidade *Let's Encrypt*), a estilização do cabeçalho e a caixa de diálogo perguntando ao usuário se ele deseja instalar a aplicação localmente. A segunda imagem exibe o ícone instalado na área de trabalho do celular que, após clicado, exibe a *splash screen* (terceira tela) gerada a partir dos dados do arquivo `manifest.json`.

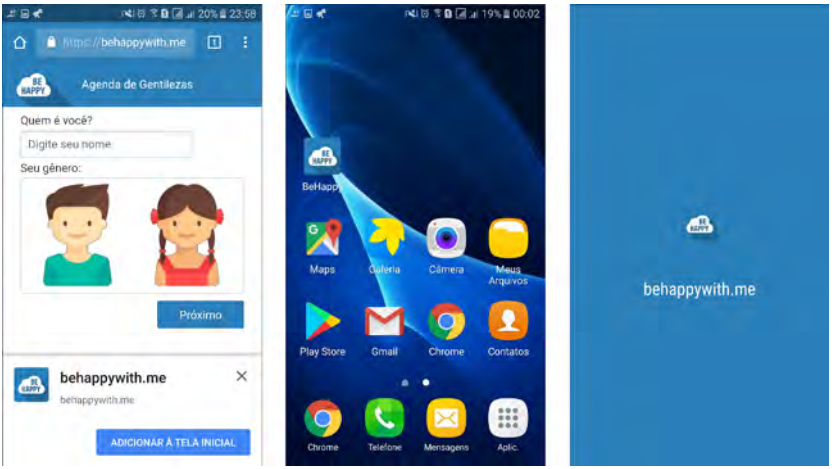

Figura 8.17: Acessando a aplicação BeHappy

Continuando o teste manual, a tela mais à esquerda mostra o cadastro de um usuário chamado *Guilherme* cujo avatar é *Avatar 15*. A tela seguinte exibe o *Toast* de sucesso do cadastro, já destacando a exibição dos dados do usuário. Por fim, a última tela mostra que, mesmo que você feche a aplicação e desligue a conexão com a internet, ao clicar no ícone do *BeHappy* na área de trabalho, a PWA será aberta exibindo os dados persistidos no `localStorage`.

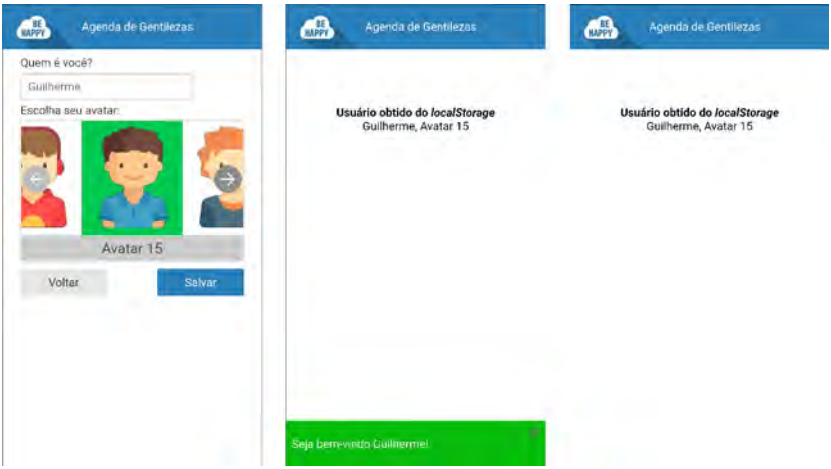

Figura 8.18: Acessando a aplicação BeHappy

Desta forma, concluímos nosso projeto! Uma última observação é que a página publicada em https://behappywith.me foi traduzida para o inglês, cujo código está no *branch* `master_en-US` (https://github.com/lgapontes/behappywith.me/tree/master_en-US). O *branch* `master` (https://github.com/lgapontes/behappywith.me) contém o mesmo código (em português) conduzido ao decorrer deste livro.

8.3 CONCLUSÃO

Caro leitor, a jornada proposta nesta obra chega ao final. Em uma rápida retrospectiva, abordamos as minúcias inerentes ao *Progressive Enhancement*, os detalhes de funcionamento do framework React, sua integração com os diversos pacotes necessários ao ambiente de desenvolvimento, a codificação dos componentes e requisitos PWA e, finalmente, a publicação da

aplicação em um ambiente real de produção.

Espero que, a partir do conteúdo apresentado, você esteja apto a implementar sua própria aplicação progressiva e acompanhar a evolução do tema, que, aliás, passa por uma constante evolução e padronização. Com o passar dos dias, mais e mais sistemas mobile e navegadores suportam os requisitos PWA, com uma expectativa real de que a *Progressive Enhancement* deixe de pertencer às *boas práticas* de desenvolvimento web e assuma o posto de *commodity*. Ou seja, no futuro, uma página web que não funcione *off-line* e não permita a instalação local será tão estranha aos usuários quanto uma televisão de tubo sem um controle remoto.

E vou além: quem sabe, em breve, as lojas mais famosas (*Google Play* e *App Store*) não se tornem aderentes aos recursos progressivos ao ponto de permitir a promoção de PWAs? A loja do Windows 10 permite tais publicações. A loja específica do Firefox também contém este recurso, mas sua descontinuidade já foi anunciada em 2018. Vou deixar esse tema no ar para que você, um desenvolvedor de aplicações progressivas, fomente a ideia.

Capítulo 9

REFERÊNCIAS BIBLIOGRÁFICAS

BUTLER, Tim. *Nginx cookbook*. Birmingham: Packt Publishing Ltd., 2017.

CARR, Richard. *Observer Design Pattern.* Jun. 2009. Disponível em: http://www.blackwasp.co.uk/Observer.aspx.

CHINNATHAMBI, Kirupa. *Learning React*: a hands-on guide to building maintainable, high-performing web application user interfaces using the React JavaScript library. Addison-Wesley Professional, 2016.

DANDEKAR, Kiran; RAJU, Balasundar I.; SRINIVASAN, Mandayam A. *3-D Finite-Element Models of Human and Monkey Fingertips to Investigate the Mechanics of Tactile Sense.* The Touch Lab, Department of Mechanical Engineering, and The Research, Laboratory of Electronics, Massachusetts Institute of Technology, v. 125, p. 682-691, out. 2003. Disponível em: http://touchlab.mit.edu/publications/2003_009.pdf.

FJORDVALD, Martin. *Instant Nginx Starter*. Birmingham: Packt Publishing Ltd, 2013.

FOWLER, Martin. *Analysis patterns*: reusable object models. Indianapolis: Addison-Wesley, 1997.

FOWLER, Martin. *Patterns of enterprise application architecture*. Crawfordsville: Addison-Wesley, 2002.

FOWLER, Martin. *Refatoração*: aperfeiçoando o projeto de código existente. Porto Alegre: Bookman, 2004.

GUSTAFSON, Aaron. *Adaptive Web Design*: crafting rich experiences with progressive enhancement. Chattanooga: Easy Readers, 2011.

IHRIG, Colin J. *Pro Node.js para desenvolvedores*. Rio de Janeiro: Editora Ciência Moderna, 2014.

LOPES, Sérgio. *Aplicações mobile híbridas com Cordova e PhoneGap*. São Paulo: Casa do Código, 2016.

OBJECT PARTNERS. *Comparing React.js performance vs. native DOM*. Nov. 2015. Disponível em: https://objectpartners.com/2015/11/19/comparing-react-js-performance-vs-native-dom/.

OLAKARA, Abdel Raoof. *How to install Node.js without admin rights*. Nov. 2014. Disponível em: http://abdelraoof.com/blog/2014/11/11/install-nodejs-without-admin-rights.

PASTOR, Pablo. *MVC for Noobs*. Mar. 2010. Disponível em: https://code.tutsplus.com/tutorials/mvc-for-noobs--net-10488.

PEREIRA, Caio Ribeiro. *Aplicações web real-time com Node.js*. São Paulo: Casa do Código, 2013.

PEREIRA, Caio Ribeiro. *Construindo APIs REST com Node.js*. São Paulo: Casa do Código, 2016.

PEYROTT, Sebastián. *More Benchmarks:* Virtual DOM vs Angular 1 & 2 vs others. Jan. 2016. Disponível em: https://auth0.com/blog/more-benchmarks-virtual-dom-vs-angular-12-vs-mithril-js-vs-the-rest/.

PRESSMAN, Roger S. *Engenharia de software:* uma abordagem profissional. 7. ed. Porto Alegre: McGraw Hill, 2011.

STEFANOV, Stoyan. *React:* up & running – Building web applications. Sebastopol: O'Reilly, 2016.