Front-end com Vue.js

Da teoria à prática sem complicações

Casa do Código

LEONARDO VILARINHO

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-27-4

EPUB: 978-85-94188-28-1

MOBI: 978-85-94188-29-8

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

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

QUEM EU SOU?

Meu nome é Leonardo Vilarinho, sou estudante (ou formado, dependendo do ano em que você esteja lendo o livro) no curso superior de Análise e Desenvolvimento de Sistema do Instituto Federal do Triângulo Mineiro. Trabalho como freelancer há cerca de dois anos e, durante esse tempo, obtive muito sucesso na minha pequena carreira.

Como desenvolvedor, sou um belo cientista; vivo testando, pesquisando, reinventando e tendo ataques de loucuras, pois ser desenvolvedor não é só trabalhar com um código, mas sim entregar soluções, e entregar soluções é um trabalho árduo, porém maravilhoso. E a cada dia que passa, procuro soluções cada vez melhores e únicas, o que requer cada vez mais loucura.

Este livro veio como uma oportunidade ímpar de expandir meu conhecimento e passá-lo a outras pessoas.

PREFÁCIO

É de suma importância um sistema web conseguir entregar um serviço de qualidade para seu usuário. Dentro desse aspecto, temos alguns itens que podem se destacar e dar um diferencial à aplicação, seja um layout mais moderno ou interativo, um tempo de carregamento menor ou o funcionamento em qualquer plataforma.

> "Mas para conseguir algo assim, é necessário gastar muito tempo, HTML, CSS e JavaScript embolados e altas gambiarras, então prefiro entregar algo *meia-boca*". — **Desenvolvedor Qualquer**

Se você pensa como o *Desenvolvedor Qualquer*, então não conhece o **Vue.js**. Com ele, qualquer um pode entregar um projeto de qualidade, com as características citadas anteriormente e outras coisas.

O Vue é um *framework* feito em JavaScript, que tem como principal objetivo o reaproveitamento de código. Nele podemos criar aplicações web com maior qualidade e agilidade, e sua curva de aprendizagem é muito pequena. Embora ainda seja menos ativo no mercado, sua comunidade é bastante ativa, e a cada dia surgem novas extensões e recursos para serem aparelhados a ele, aumentando muito seu poder.

Este livro lhe dará um conhecimento abrangente sobre o Vue,

em sua versão 2.2. Passaremos por cada particularidade, com exemplos que mostram quais problemas podemos resolver com o que está sendo estudado. No final, teremos exercícios, com o objetivo de que o leitor termine a leitura desta obra podendo aplicar todo o seu conteúdo no mercado, sabendo usar as ferramentas adicionais e buscando mais conhecimento sobre a biblioteca.

Sugiro que o leitor saiba o básico em HTML5 e JavaScript, pois no decorrer da obra estruturas básicas dessas linguagens serão usadas no Vue para criar nossos sistemas, e a falta desse conhecimento prévio pode resultar em mau entendimento. Você pode aprender sobre esses temas em qualquer fonte de qualidade, como os livros: *HTML5 e CSS3: domine a web do futuro* e *Lógica de programação: crie seus primeiros programas usando JavaScript e HTML*, ambos à venda na Casa do Código.

Dedico esta obra a qualquer pessoa que pretenda ganhar conhecimento no desenvolvimento web, criando páginas mais rapidamente e com melhor qualidade, seja você um estudante, professor, profissional da área ou entusiasta, pois sabedoria nunca é demais, não é mesmo?

Sumário

Guia de consulta 1

1 Vue, por que usá-lo? 2

1.1 O que este livro abrange? 4

1.2 Suporte ao leitor 5

2 Preparando e iniciando o ambiente 7

2.1 Lapidando o Sublime Text 7

2.2 Instalando o Node.js 10

2.3 Extensões para o navegador 12

2.4 Primeiro projeto com vue-cli 15

3 Entendendo o funcionamento 19

3.1 A rainha das instâncias 20

3.2 Componentes, como peças de Lego 22

3.3 Reatividade, vendo tudo o que se altera 27

4 Criando e exibindo dados 29

4.1 O atributo data 31

4.2 Exibindo dados com diretivas 32

4.3 Teste seu avanço 40

5 Manipulando dados 43

5.1 Methods — Blocos de ações 44

5.2 Filters — Filtrando dados 46

5.3 Computed properties — Modificando dados para exibi-los 48

5.4 Watchers — Dados que podem esperar 51

5.5 Teste seu avanço 54

6 Componentes juntos são mais fortes 56

6.1 Teste seu avanço 67

7 Reutilizando componentes 69

7.1 Props — Recebendo atributos externos 70

7.2 Slot — Recebendo um bloco de código 75

7.3 Mixins — Estendendo um componente 79

7.4 Emit — Comunicação entre componentes 83

7.5 Teste seu avanço 88

8 Cada um segue seu caminho, com rotas! 90

8.1 Criando um cenário de exemplo 91

8.2 Configurando o Vue-router 92

8.3 Criando rotas 93

8.4 Trocando de página 96

8.5 Hash não é URL amigável 97

8.6 Criando sub-rotas 99

8.7 Enviando parâmetros 101

8.8 Teste seu avanço 105

9 Gerenciamento de estado com Vuex 107

9.1 Store — Criando nossa loja de dados 108
9.2 States — Declarando dados 109
9.3 Mutations — Alterando dados 112
9.4 Getters — Pegando dados 116
9.5 Actions — Executando mutações indiretamente 122
9.6 Modules — Organizando informações 124
9.7 Mapeando estado em componentes 127
9.8 Teste seu avanço 133
10 Criando e dividindo serviços 135
10.1 Teste seu avanço 141
11 Acrescentando funcionalidades 144
11.1 Criando diretivas customizadas 145
11.2 Criando seus próprios plugins 148
11.3 Publicando pacotes NPM 151
11.4 Teste seu avanço 153
12 Introdução a testes 155
12.1 Teste seu avanço 165
13 Alguns recursos escondidos 166
13.1 Manipulando teclas de atalho 166
13.2 Ciclo de vida dos componentes 168
13.3 Estendendo componentes com extends 170
13.4 Trabalhando com referências 172
13.5 Variável cifrão 172
13.6 Atualizando um componente 176
13.7 Diretiva v-pre 177

13.8 Acessando um índice no v-for 178
13.9 Modo history em produção 179
13.10 Concluindo 180

Projeto orientado **182**

14 Registro de usuários **183**

15 Autenticando um usuário **191**

16 Criando uma anotação **199**

17 Listando as notas criadas **205**

18 Apagando uma anotação **211**

19 Concluindo **214**

Versão: 21.6.5

Guia de consulta

Nesta primeira parte do livro, veremos uma documentação mais compacta do Vue.js, com palavreado e exemplos simples, tendo por objetivo passar e fixar o conhecimento a qualquer pessoa com domínio básico de HTML, CSS e JavaScript.

Nós veremos cada aspecto que o Vue disponibiliza na sua versão 2.2. Este livro pode se tornar um guia de consulta, por tratar de cada assunto separadamente, facilitando a pesquisa e a resposta sobre alguns dos principais fatores que são destaques no Vue atualmente.

No fim desta parte, você, leitor, terá conhecimento suficiente para criar qualquer tipo de aplicação com o Vue, aplicando os conceitos de reúso ensinados no decorrer do aprendizado, e na prática com exercícios.

Todos os exemplos vistos neste livro podem ser encontrados em https://github.com/leonardovilarinho/livro-vue.

Capítulo 1

VUE, POR QUE USÁ-LO?

O Vue.js trata-se de um *framework* progressivo que permite o desenvolvimento de interfaces de comunicação com o usuário. Ele tem o objetivo de ser incrementável ou montável, ou seja, divide-se em diversas peças que podem ser facilmente encaixadas umas nas outras.

Pronúncia: o nome Vue é de origem francesa, o que faz com que muitas pessoas pronunciem seu nome incorretamente. O certo é dizer *view*, como aquela palavra do inglês que significa *visão*.

Essa biblioteca, que você está prestes a amar, foi desenvolvida em JavaScript pelo japonês Evan You. Atualmente, mesmo sendo mantida apenas por seu criador e uma comunidade, ela se equipara ao poderoso Angular (do Google) e React (do Facebook). Isso se deve a diversos fatores, entre eles: curva de aprendizado muito menor; ser leve, vindo apenas com seu núcleo, com o resto sendo adicionado apenas quando surgir a necessidade; e ter uma performance superior à dos demais.

Apesar de não ter uma empresa por trás de si, a manutenção do projeto é muito grande. A cada dia surgem novas extensões e melhorias, deixando uma pessoa apaixonada por novas tecnologias (como eu) completamente saciada de conhecimento.

O crescimento do framework começou quando grandes empresas, como Alibaba e Baidu, começaram a usá-lo em diversos projetos de lojas virtuais e sites corporativos. Elas também ajudaram fazendo doações e um acordo para que Evan dedicasse todo o seu tempo ao projeto, recebendo um salário variante, maior que 7 mil dólares, pago por esses investidores, pela comunidade Vue e outros.

Mas sua popularidade no desenvolvimento web só deu um salto quando Taylor Otwell, o criador do Laravel — framework mais popular para PHP —, exaltou a criação do japonês em uma postagem no Twitter, dizendo:

> "*Current React learning status: overwhelmed. Learning @vuejs because it looks easy and has pretty website.*"
>
> "Status atual com a aprendizagem do React: sobrecarregado. Aprendendo @vuejs, porque parece fácil e tem um site bonito."

Depois disso, o crescimento foi instantâneo. Usuários do Laravel começaram a integrar suas aplicações com o Vue, o que deixou claro ser uma opção perfeita, pois ambos (Laravel e Vue) possuem uma sintaxe de código bem elegante.

Recomendações não faltam, mas em resumo, se você não quer ter de aprender quase tudo de desenvolvimento *front-end* para usar um framework do front, então o Vue é o que você deseja!

1.1 O QUE ESTE LIVRO ABRANGE?

Este livro aborda os principais aspectos do Vue, como usá-los e, no fim, prepara o leitor para implementar isso tudo no mercado de trabalho. É tudo muito simples de ser alocado na memória de qualquer pessoa. Embora tenha um escopo grande, cada tópico é coeso e não será difícil lembrar da totalidade do conteúdo. Porém, a prática é essencial para que isso se realize.

Procurei dividir a obra em duas partes, com objetivo de agradar um público maior. A primeira parte será um **guia** que vai passar por quase tudo o que o framework nos oferece, mostrando sua função e um exemplo, em que veremos: variáveis, métodos, filtros, observadores e muito mais. Já a segunda parte será o **projeto orientado**, na qual criaremos uma aplicação com o Vue.js, portanto, enquadrando-o em situações do mundo real.

No guia, além de descobrir tudo o que o núcleo do Vue nos oferece, veremos também como expandir a funcionalidade do framework, adicionando novos recursos e testes automatizados, fundamentais para criação de aplicações mais elaboradas. Daremos dicas de organização, entre outras.

No projeto orientado, vamos construir uma aplicação utilizando o que aprendemos da maneira correta e usando a prática do TDD para criar um sistema baseado em testes. Sempre nos lembraremos dos três pilares da programação orientada a

objetos, que na minha opinião são os três pilares de qualquer programação: a **coesão**, cada parte do sistema terá uma responsabilidade única; o **acoplamento**, mantendo o número de dependências de cada bloco igual a zero ou perto disso; e o **encapsulamento**, deixando cada bloco do sistema exibir para os demais apenas aquilo que realmente é necessário.

O Vue nos dá total liberdade de usá-lo como bem entendermos, logo você não ficará sobrecarregado (como Otwell), pois tudo funciona aqui, e funciona com elegância! Essa é a principal vantagem dessa biblioteca. Por que não fazer mais com menos? É essa a proposta, e é por isso que qualquer um deve utilizar o Vue.

1.2 SUPORTE AO LEITOR

Quero ampliar a conexão com o leitor. No repositório do livro (https://github.com.br/leonardovilarinho/livro-vue), você encontrará todos os exemplos que aparecem ao longo da obra. Cada exemplo terá um comentário adicional para expandir ainda mais seu conhecimento, e todas as dúvidas referentes ao trabalho serão respondidas quando colocadas nos *issues*.

Além do repositório, qualquer pessoa poderá acessar o blog do Vue.js Brasil (http://vuejs-brasil.com.br) para visualizar postagens de todos os autores da nossa comunidade. Esse blog é essencial para você continuar se especializando.

Por fim, mas não menos importante, a *Casa do Código* oferece um fórum (http://forum.casadocodigo.com.br) para dúvidas referentes aos livros editados por ela. Então, qualquer dúvida

postada lá, sobre esta obra, será respondida por mim.

CAPÍTULO 2

PREPARANDO E INICIANDO O AMBIENTE

Se você ainda programa no Bloco de Notas, ou em outro editor bem básico, sem configurações específicas para o uso de uma determinada ferramenta, esqueça o mercado de trabalho! Todas as empresas têm sempre um mesmo aspecto, que fala mais alto que os demais, a **produtividade**. Qualquer desenvolvedor pode ser muito bom no trabalho, criar códigos da NASA etc., mas se não for produtivo, não ganhará espaço.

Nessa situação, todo programador deve ter em sua máquina o local ideal para trabalho, com ferramentas que dão a maior produtividade possível e configurações que permitam testar códigos com agilidade. Ela também deve ser livre de travamentos e telas com carregamento demorado. Neste capítulo, instalaremos tudo o que é necessário, e veremos um código Vue pela primeira vez.

2.1 LAPIDANDO O SUBLIME TEXT

É de grande importância ter uma ferramenta robusta para criar seu código. Passei um tempo pesquisando opções que atingissem **agilidade**, **personalização** e **compatibilidade**. O Sublime foi a que

chegou mais perto de suprir totalmente esses três requisitos. Logo, vi a necessidade de compartilhar um pouco desse editor e sua configuração para o Vue.

> **Aviso**
>
> Não instale esse editor caso tenha outro favorito. Procure formas para suportar a sintaxe do Vue no editor de sua escolha.

Não entrarei em detalhes mais avançados de configuração e recursos, dado que alguns dos leitores já possuem um software com essa finalidade como um filho. Mostrarei os dois principais recursos para ter boa produtividade: **plugins** e **snippets**.

Plugins

Antes de tudo, é preciso ter o Package Control instalado no seu Sublime para realizar a instalação de qualquer plugin. Para isso, acesse o site https://packagecontrol.io/installation, e siga os passos de instalação para a sua versão do editor.

Os plugins visam expandir o funcionamento do editor para suprir um novo requisito. A seguir, temos uma lista com os principais plugins que nos ajudarão a escrever o código Vue no Sublime:

- Emmet — Possui uma sintaxe que lhe permite escrever códigos HTML em muito menos tempo. Por exemplo, ao escrever `ul>li*3` e pressionar a tecla

`Tab` , você verá que foi criada uma lista com três itens dentro dela.

- JavaScript Next — Facilita muito o reconhecimento da linguagem JavaScript, listando os métodos de cada objeto e até mesmo parte de sua documentação.
- SublimeCodeIntel — Um dos principais, faz algo parecido com o anterior, porém também reconhece nosso próprio código, tipos de variáveis etc.
- TrailingSpaces — Pode ser muito irritante para alguns programadores, pois mostra em destaque espaços inúteis no código, como espaços no final de uma linha.
- Vue Syntax Highlight — Destaca o código Vue e permite ver o seu fim ao selecionar uma tag ou abertura de bloco.
- Vuejs Complete — Possui alguns atalhos para escrever o código Vue. Basta iniciar o código e pressionar `Enter` para já obter todo o restante.
- Vuejs Snippets — Praticamente o mesmo que o anterior. É bom ter os dois para um suporte que complete mais códigos.
- AutoFileName — Quando tivermos de importar um arquivo, este plugin vai lhe mostrar os diretórios e arquivos na pasta atual, facilitando muito escrever o caminho do arquivo sem erro.

Para instalar qualquer desses plugins, com o Package Control já instalado, basta pressionar `Ctrl+Shift+P` , digitar `install` , e selecionar a opção `Package Control: Install Package` . Após isso, é só pesquisar pelo nome do plugin e clicar nele, que o

processo de instalação começará.

Snippets

Podemos defini-los como *atalhos de código*. Basicamente funcionam assim: se temos um código padrão que usamos em muitos arquivos, criamos um atalho para ele.

Para criar um novo snippet no Sublime, acesse o menu `Tools/Developer/New Snippet`, ou `Tools/New Snippet`. Será aberto um novo arquivo, no qual temos a tag `content` e `tabTrigger`, respectivamente, como o código que essa snippet vai criar e seu nome de atalho. Definimos assim:

```
<snippet>
    <content><![CDATA[
Hello, ${1:this} is a ${2:snippet}.
]]></content>
    <tabTrigger>hello</tabTrigger>
</snippet>
```

Salve o arquivo como `hello.sublime-snippet` e, em um novo arquivo, digite `hello` e pressione a tecla `Tab`. Note o resultado: **Hello, this is a snippet.**, exatamente aquilo que colocamos na tag `content`.

2.2 INSTALANDO O NODE.JS

A melhor alternativa para se trabalhar com Vue, assim como todas as ferramentas JavaScript, é usando o **Node.js** (uma plataforma de desenvolvimento back-end com JS) e o **NPM** (um gerenciador de pacotes JavaScript que depende do Node.js para ser instalado).

Isso se deve ao fato de termos em um projeto uma grande quantidade de código de terceiros, e podemos gerenciá-la com o **NPM**. Logo, não será necessário baixar manualmente e importar cada biblioteca nova, afetando altamente sua produtividade.

Existe uma outra opção para se trabalhar com Vue, usando o CDN oficial, ou baixando um arquivo minificado e importando-o no HTML normalmente com a tag `style`. Apesar de parecer mais simples, pois não depende de outro software, essa escolha pode lhe dar um pouco mais de trabalho. Isso porque todas as dependências seriam importadas manualmente, então teríamos também uma perda considerável de desempenho, dado que o Vue com Node gera um código minificado mais leve para ser usado em ambiente de produção.

No Windows

Assim como a maioria dos softwares, existe um executável para instalá-lo no Microsoft Windows. Para isso, basta entrar no site https://nodejs.org, escolher uma das opções em `Download for Windows` (sugiro a versão mais nova) e instalar com `Next, Next, Next ... Finish`.

No Linux

No Linux, temos de ir ao terminal para fazer a instalação. O exemplo citado aqui instalará a versão 7, que é a mais recente (no momento da escrita do livro). Caso tenha alguma versão posterior, instale-a, pois ela poderá conter uma série de melhorias referentes à anterior.

Em distribuições baseadas em Debian, fazemos:

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Em distribuições baseadas em Fedora, fazemos:

```
curl --silent --location https://rpm.nodesource.com/setup_7.x | b
ash -
yum -y install nodejs
```

No MacOS

Para instalar no sistema operacional da Apple, temos uma forma parecida com a do Windows. Basta acessar o site https://nodejs.org, baixar seu `.pkg`, executá-lo e seguir o fluxo `Next, Next, Next ... Finish`.

Testando a instalação

Para verificar se correu tudo bem, abra um prompt/terminal e digite dois comandos:

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

Se estiver tudo certo, a saída será o número da versão de cada um instalado no seu computador, como o exemplo:

```
C:\Users\leona>node -v
v7.5.0

C:\Users\leona>npm -v
4.1.2
```

2.3 EXTENSÕES PARA O NAVEGADOR

Para completar o ambiente de desenvolvimento, o Vue e a sua comunidade disponibilizam algumas extensões que dão um

feedback gráfico do código de sua aplicação. Elas são ideais para desenvolvedores debugarem o código e verem a estrutura e as mudanças que estão sendo apresentadas na tela naquele momento.

Entre essas extensões, citarei as duas principais, que têm objetivos parecidos, porém, quando usadas juntas, dão um suporte de grande qualidade. Ambas estão disponíveis apenas para o Google Chrome, mas alguns navegadores possuem extensões que permitem instalar outras, originalmente feitas para o Chrome, como é o caso do navegador que eu utilizo, o Opera.

Vue Devtools

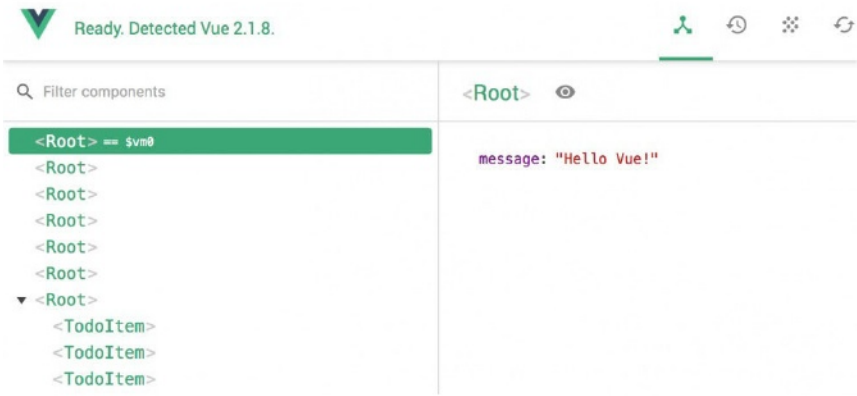

Figura 2.1: Vue Devtools

O link para download é https://github.com/vuejs/vue-devtools.

Como podemos ver na imagem, essa extensão nos mostra todos os elementos que compõem nossa tela. Além disso, ao selecionar um deles, podemos ver quais atributos ele possui e qual o valor atual dos atributos. Muito útil para depurar, não é mesmo?

DejaVue

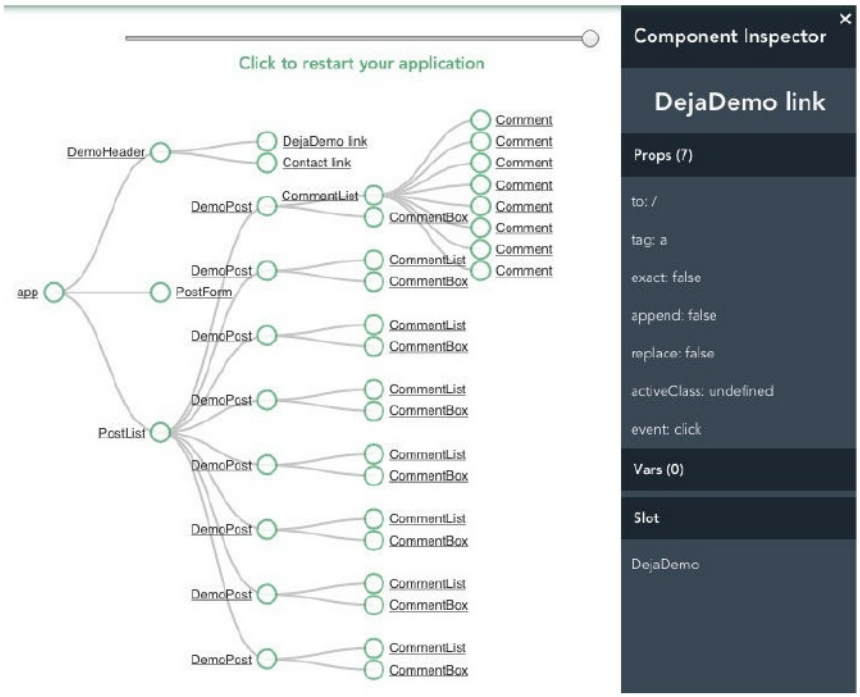

Figura 2.2: DejaVue

O link para download é https://github.com/MiCottOn/DejaVue.

Essa extensão nos mostra até um gráfico. Que beleza, não? No gráfico, podemos ter uma outra visão de todos os elementos listados no Vue Devtools. É uma visão mais complexa, em que podemos perceber o relacionamento de cada elemento, cada bloco que forma nossa aplicação. Ao selecionar um, é possível ver diversas coisas relacionadas a ele, como propriedades, variáveis e alocamentos.

Não se preocupe se não entender isso agora. No decorrer do livro, aprenderemos tudo isso e como usar essas ferramentas.

2.4 PRIMEIRO PROJETO COM VUE-CLI

Finalmente criaremos nosso primeiro projeto. Para isso, temos de iniciá-lo. No Vue, há três formas para se iniciar um projeto, sendo a primeira delas usando o CDN ou arquivo minificado, citada na seção anterior, onde alertamos sobre sua falta de produtividade.

Nossa segunda opção é instalar a biblioteca como uma dependência do NPM, e importá-la para nosso projeto seguindo normalmente com seu uso. Essa alternativa é um pouco melhor do que a outra, por conter um controle sobre tudo o que é usado no projeto. Mas ainda não é a melhor, pois não temos uma solução para minificar e executar nosso código.

Por fim, temos uma ferramenta disponibilizada pelo Evan, em que podemos baixar e iniciar projetos já configurados, com minificadores de código, testadores e outros. O `vue-cli` é um gerenciador de template. Com ele, podemos baixar e criar projetos com esqueletos de código predefinidos, tendo o objetivo de agilizar bastante a construção e a configuração de um projeto.

A escolha parece clara. Usaremos o `vue-cli` por nos dar uma suíte com tudo o que devemos ter para executar, minificar e testar uma aplicação. Vamos instalá-lo globalmente na máquina usando o NPM:

```
npm install -g vue-cli
```

Após a instalação, o comando `vue` será disponibilizado no

terminal. Agora você precisa escolher um diretório qualquer para armazenar seus projetos Vue, então deve entrar nele e iniciar uma nova aplicação com:

```
vue init webpack-simple helloworld
```

Nesse comando, usamos `vue init` para dizer que queremos criar um novo projeto, e em seguida definimos o template `webpack-simple`. Por fim, citamos o diretório no qual os arquivos serão colocados — no caso, em uma pasta chamada `helloworld`.

Templates

São usados para agilizar o processo de arquitetura de um projeto. Temos duas principais plataformas para gerenciar e empacotar nosso código, a *webpack* e a *Browserify*. Ambas possuem esse mesmo objetivo, então o uso de uma ou outra será definido pelo desenvolvedor. Eu particularmente prefiro a *webpack*, por ter mais conteúdo didático no mundo Vue, por isso vamos usá-la neste livro.

Por padrão, o `vue-cli` contém cinco templates, que podem ser vistos com o comando `vue list`:

- `browserify` — Usa o gerenciador citado e contém um ambiente mais potente, com verificador de código, testes de unidade e suporte ao Jade.
- `browserify-simple` — Um pouco mais simples que o anterior, tendo apenas o suporte ao Jade.
- `simple` — Não contém nenhum gerenciador, apenas com HTML e JS puro.
- `webpack` — Usando o webpack como empacotador, temos suporte a testes de unidade, extração de CSS e outros.
- `webpack-simple` — Usa o webpack como empacotador com um importador de arquivos mais potente.

O próximo passo é executar nossa aplicação, com o objetivo de testá-la. Para isso, entramos no diretório do projeto, usando o comando `cd` , e dizemos que queremos executar nosso código em ambiente de desenvolvimento:

```
cd helloworld
npm run dev
```

Depois de carregar os arquivos necessários, seu navegador padrão será aberto no http://localhost:8080, apresentando uma tela parecida com esta:

Figura 2.3: Vue.js 2

Esse é o sinal de que o Vue.js está devidamente configurado e funcionando. Se você fechar o terminal e recarregar, a página não mais funcionará. Isso porque o comando `npm run dev` cria uma *live* (ou servidor), que pode ser acessada no computador atual ou em sua rede, transmitindo todas as alterações em tempo real. Logo veremos que, se mudarmos algo nessa página, o navegador carregará a mudança no mesmo instante.

Capítulo 3

ENTENDENDO O FUNCIONAMENTO

O Vue pode ter diversas formas e fluxos de funcionamento, mas aqui veremos a forma mais correta, na qual temos um arquivo principal que requisita um bloco inicial e este, por sua vez, exibe os demais blocos que compõem nossa aplicação. Também veremos algo sobre a reatividade, um elemento fundamental para que o framework possa fazer tudo em tempo real com uma performance altíssima.

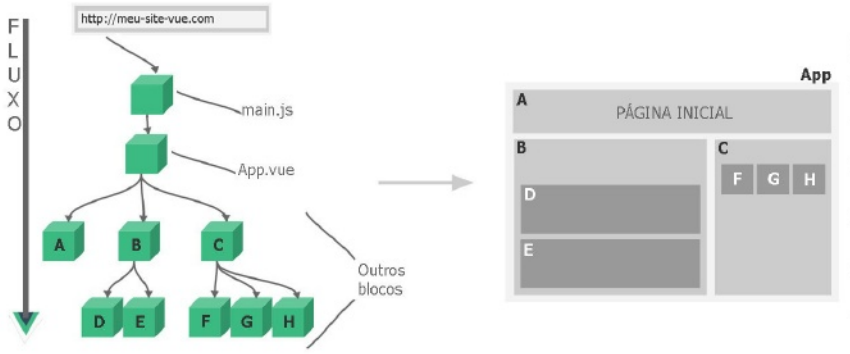

Figura 3.1: Fluxo de funcionamento do Vue

Na figura anterior, temos um exemplo de projeto no Vue. Podemos ver que, ao acessar esse site imaginário, obrigatoriamente

passaremos por dois arquivos: o `main.js`, onde temos o código iniciando a aplicação; e o `App.vue`, que será nosso layout, no qual chamamos os demais blocos, sejam eles A ou B.

Não se preocupe com esses dois arquivos, veremos mais sobre eles adiante. Neste momento, apenas observe que, depois do `App.vue`, temos uma árvore de blocos, e isso nos mostra o quão separado e organizado um site fica.

O resultado apresentado no browser é o nosso bloco representado por `App.vue`, no qual teremos os outros blocos que formavam a árvore, constituindo a interface do site. Como podemos ver: o A virou nosso cabeçalho, o B e o C são barras laterais e o restante dos blocos popula essas barras.

A ideia inicial é que você perceba que o reúso pode ser muito alto, dado que cada elemento da nossa interface está separado em blocos, e cada bloco tem seus blocos filhos e pai. Nas seções a seguir, veremos como é composto o arquivo `main.js` e os tão citados blocos, assim como o `App.vue`.

3.1 A RAINHA DAS INSTÂNCIAS

Esta é o nosso `main.js`, presente no diretório `src` e criado automaticamente quando se usa o `vue-cli`. Ele contém também uma configuração básica que nos permite executar o projeto. Trata-se do ponto inicial de qualquer aplicação feita em Vue, pois é o arquivo principal. Nele podemos estender as funcionalidades da biblioteca e fazer configurações básicas do projeto.

O exemplo mais básico desse arquivo se encontra utilizando apenas o Vue no projeto. Na instância do Vue, passamos um

objeto genérico JavaScript que pode ter diversas propriedades, como a propriedade `mounted` — um método executado assim que essa instância for montada. Ao acessar o navegador, você terá o alerta `Hello World!` .

```
import Vue from 'vue'

new Vue({
  mounted() {
      alert('Hello World!')
  }
})
```

Mas para fazer isso da maneira certa, como vimos na imagem, precisamos ter um arquivo que inicie um bloco, a partir do qual nossa aplicação se iniciará, transformando assim o `main.js` em nossa instância **root** (ou raiz). Para tanto, basta deixar o arquivo `main.js` conforme vindo na instalação. Nele, além do Vue, importamos um bloco padrão que terá o objetivo de iniciar nossa aplicação.

Depois, passamos dois novos atributos para o objeto do Vue:

- `el` — Responsável por indicar um elemento HTML que será a raiz do site. Então, qual será o objeto principal do nosso sistema, uma `div` , um `header` etc.? O Vue precisa saber disso para colocar o conteúdo visível para o usuário lá dentro, logo precisamos indicar esse elemento inicial com o `el` .
- `render` — Uma função que renderiza um bloco na tela. O Vue.js precisa saber o que ele começará desenhando nela e, para isso, usamos o atributo `render` . Ele é uma função que recebe um parâmetro `h` , responsável por desenhar o bloco na tela usando

uma tecnologia chamada VirtualDOM. Esta, por sua vez, nos dá um aumento de performance, dado que renderiza somente aquilo que realmente sofreu uma alteração.

```
import Vue from 'vue'
import App from './App.vue'

new Vue({
  el: '#app',
  render: h => h(App)
})
```

Mais tarde voltaremos à nossa raiz para adicionar extensões no framework e configurar novos aspectos do projeto. Por enquanto, basta deixá-lo assim para que o fluxo padrão funcione.

3.2 COMPONENTES, COMO PEÇAS DE LEGO

Todos os blocos e instâncias do Vue, tão citados anteriormente no livro, são na realidade **componentes**. Eles, como o próprio nome indica, são pedaços de códigos que compõem nosso sistema.

Um componente tem três formas de mostrar um conteúdo na tela, denominadas *render functions*. A primeira é usando o VirtualDOM puro, em que criamos elementos HTML na tela com o próprio JavaScript:

```
new Vue({
    el: '#app',
    render : (createElement) => {
        return createElement('header', { attrs: {id: 'elemento'}
}, [
            createElement('h1', 'Titulo do post'),
            createElement('p', 'Conteudo do post')
        ])
    }
```

```
})
```

Ao colocar esse conteúdo no arquivo `main.js`, serão exibidos na tela um título e um parágrafo, exatamente o que pedimos para fazer. Primeiramente criamos um `header` com um identificador denominado `elemento` e, dentro dele, criamos um `h1` e um `p` para mostrar um título e um parágrafo, respectivamente. O trabalho pode ser imenso caso tivermos mais elementos compondo nossa tela, logo não é comum utilizar esse método.

A segunda alternativa é um pouco parecida com a anterior, e utiliza uma biblioteca chamada JSX para misturar HTML dentro do JavaScript, criando o template a ser mostrado na tela. Para usarmos essa opção no Vue, precisaremos instalar alguns plugins:

```
npm install\
  babel-plugin-syntax-jsx\
  babel-plugin-transform-vue-jsx\
  babel-helper-vue-jsx-merge-props\
  --save-dev
```

E, por fim, é necessário alterar o arquivo `.babelrc` localizado na raiz do nosso projeto, deixando-o assim:

```
{
  "presets": ["es2015"],
  "plugins": ["transform-vue-jsx"]
}
```

Agora sim poderemos testar essa maneira, e usaremos o mesmo exemplo dado anteriormente. Com esse método, teremos o código:

```
new Vue({
    el: '#app',
    render (h) {
        return (
            <header>
```

```
                <h1>Titulo do post</h1>
                <p>Conteudo da postagem</p>
            </header>
        )
    }
})
```

Um pouco melhor, não? Mas, por exemplo, quando tivermos uma lista de dados para ser mostrada na tela, teremos uma complicação maior, pois o retorno do método `render` ficará mais complexo. O JSX nos oferece uma sintaxe bem difícil de ser trabalhada, além de misturar HTML e JavaScript, o que não é muito limpo.

Por fim, temos o formato de arquivo disponibilizado pelo próprio Vue, o `.vue`, que se divide entre três tags: `template`, a área visual do componente, onde entra o HTML; `script`, onde está todo o objeto do componente, com suas propriedades, métodos etc.; e `style`, o estilo CSS daquele bloco.

Um arquivo `.vue` deixa tudo mais separado e organizado. Em nenhum momento o HTML ou o CSS se mistura com o JavaScript. Essa é a grande vantagem desse último método, e o motivo para que a grande maioria dos profissionais o utilize. É ele que usaremos no livro. A seguir, veremos como é composto um componente usando esse método de definição:

```
<template>
  <div>
    <!-- conteúdo visual -->
  </div>
</template>

<script>
export default {
  // dados do componente, como nome, atributos e métodos
}
```

```
</script>

<style>
/* css com classes, id e outros seletores */
</style>
```

Mais uma vez, note como temos todo o conteúdo separado: o que será exibido na tela, o código que faz aquilo funcionar e o estilo para formatar tudo o que é mostrado.

Atenção ao template

A tag `template` de um componente possui uma especialidade. Nela, podemos ter apenas *um* elemento filho, ou seja, não podemos ter dentro dela dois botões como filhos diretos. Por exemplo:

```
<template>
    <button>Entrar</button>
    <button>Registrar</button>
</template>
```

O correto a se fazer para adicionar esses dois botões é colocá-los dentro de uma `div`, deixando o template com apenas a filha `div`:

```
<template>
    <div>
        <button>Entrar</button>
        <button>Registrar</button>
    </div>
</template>
```

Caso faça como no primeiro trecho de código, o seu projeto nem vai executar, dizendo que um componente pode ter apenas um elemento raiz. No exemplo dado, tal elemento foi a `div`.

Podemos nomear um componente, para que possamos identificá-lo em ferramentas como o *Vue Devtools* e o *DejaVue*. Para isso, usamos o atributo `name`. Bem sugestivo, não?

O uso de nomes não é obrigatório, mas é visto como uma boa

prática, pois identifica uma parte do código. Um componente sem nome, diferente de uma classe sem nome, até funcionaria. Entretanto, você não identificaria o que estivesse acontecendo ali, o que, em sistemas grandes, seria um caos.

Esse nome deve ser uma `String` com minúsculas ou maiúsculas, porém precisa ter a mesma regra de definição de variáveis, ou seja, sem caracteres especiais, espaços e números no início. Também é recomendável usar um prefixo para identificar o componente, pois nomes genéricos como `button` poderiam facilmente gerar confusão e se misturar com a tag do HTML. Além disso, isso poderia afetar a reusabilidade, pois seria muito comum ter componentes com nomes iguais. Então, defina um prefixo que esteja claro para você, como:

```
export default {
  name: 'lv-botao'
}
```

Todo componente tem um ciclo de vida, uma vida útil, e cada uma de suas etapas pode ser interceptada, como fizemos anteriormente ao usar o método `mounted`. Veremos no decorrer do livro os tipos de interceptadores do ciclo e em qual fase cada um é chamado.

3.3 REATIVIDADE, VENDO TUDO O QUE SE ALTERA

Esse é um dos aspectos nativos mais interessantes no Vue. Trata-se de algo um pouco complexo até mesmo para as pessoas mais experientes no assunto. Esta seção tem o objetivo de abordar o básico sobre a reatividade no Vue. Mas não se preocupe, logo a

veremos funcionando.

A reatividade define-se como uma lógica na programação do Vue, responsável por observar cada variável atualmente presente nele, vendo seu estado e valor. Quando uma variável é alterada, instantaneamente a reatividade trata de informar a modificação para todos os observadores de elementos que a usam. Assim, eles são renderizados novamente na tela, fazendo o usuário ver essas modificações em tempo real.

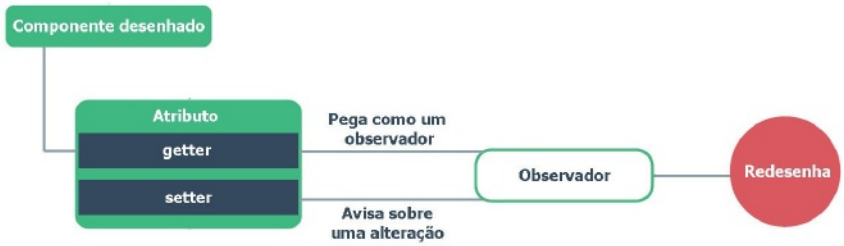

Figura 3.2: Reatividade alterando dados

A imagem anterior exemplifica mais ainda o conceito. Nela, quando um componente é desenhado na tela, todos os seus atributos são transformados em observadores e redesenhados.

A reatividade é algo da natureza do Vue, e seu uso é constante. Logo, não é possível desabilitá-la. É um recurso essencial para termos páginas mais interativas e bonitas, pois podemos ter qualquer conteúdo se alterando em tempo real, sem recarregamento de página, o que dá uma sensação de agilidade, algo primordial hoje em dia.

Capítulo 4

CRIANDO E EXIBINDO DADOS

Enfim chegou o esperado momento, a hora de praticar! Espero que entendam que a teoria vista até agora é uma parte essencial para a inicialização com o Vue. Agora vamos dar um *upgrade* no livro, tornando-o muito mais prático!

Separei este capítulo para mostrar especialmente como exibimos e criamos atributos. Vale prestar bastante atenção nessa parte e estudá-la, pois é o ponto principal de qualquer aplicação, já que, sem dados, não temos nada para o usuário ler. E mesmo com dados, se não soubermos exibi-los, acontece o mesmo problema.

Para isso, vamos criar um projeto usando o `vue-cli`, instalar suas dependências e executá-lo em ambiente de desenvolvimento:

```
vue init webpack-simple criando-exibindo-dados
cd criando-exibindo-dados
npm install
npm run dev
```

Lembre-se de que, para ver o resultado no navegador, devemos deixar o terminal aberto executando a *live* criada pelo comando `npm run dev`. Caso em algum momento você feche o terminal sem querer, será necessário executar apenas esse comando, pois já

teremos o projeto criado e suas dependências instaladas.

Todo o código deste capítulo será adicionado no arquivo `src/App.vue` do projeto que foi criado, pois como vimos antes, o `App` é o nosso componente principal.

Como exemplo, vamos tomar um componente que tenha de exibir um título, um subtítulo e uma lista de tarefas. Enquanto nossa lista não tiver tarefas, exibiremos uma mensagem "sem tarefas", com nossos campos e botões desabilitados; quando essa lista for carregada, habilitaremos tudo e mostraremos seus dados.

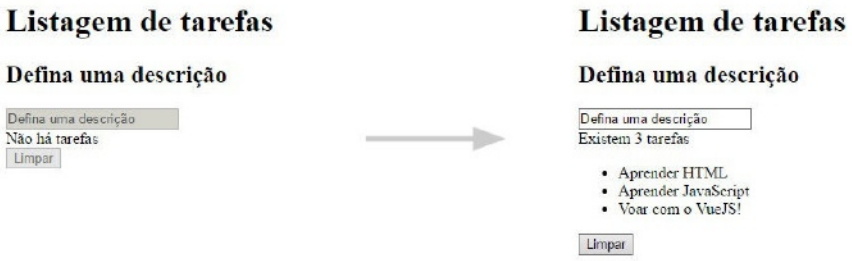

Figura 4.1: Exemplo do que faremos neste capítulo

Na imagem anterior, podemos ver o exemplo do capítulo. Nele, nosso `input` e o botão estarão desabilitados antes de carregar as tarefas e veremos uma mensagem alertando que não temos nenhuma. Ao carregá-las, o campo e o botão serão habilitados, e uma mensagem mostrará a quantidade de tarefas que temos e exibirá todas as da lista.

Dica

Após iniciar um novo projeto, execute-o com `npm run dev` e deixe o terminal aberto para ele continuar sendo transmitido para o navegador. Assim, toda alteração no código será mostrada instantaneamente no browser.

4.1 O ATRIBUTO DATA

Primeiramente, não é data de aniversário, e sim, do inglês, *dados*. No Vue, trata-se de um método em que retornamos todos os atributos que o componente atual terá, ou seja, todas as variáveis que usaremos em um bloco estarão no retorno dessa função.

Esses atributos do retorno de data se transformarão nos observadores da reatividade que vimos no capítulo anterior. Assim, quando um desses dados for alterado, veremos isso instantaneamente na tela, sem ter de recarregá-la ou algo do tipo.

Podemos definir na tag `script` do componente `App` , além do nome do nosso componente, as variáveis `titulo` , que armazenará o título da página, `subtitulo` e `tarefas` , que será uma lista. Essas variáveis compõem o objeto de retorno do método `data` . Declarando-as lá, será possível acessar esses dados em qualquer lugar do componente usando o `this` .

```
export default {
  name: 'lv-tarefas',
  data () {
```

```
    return {
      titulo: 'Listagem de tarefas',
      subtitulo: 'Defina uma descrição',
      tarefas: []
    }
  }
}
```

4.2 EXIBINDO DADOS COM DIRETIVAS

Diretivas são manipuladores de template, usados como atributos do HTML. Com elas, podemos alterar nosso DOM (árvore de elementos HTML) acrescentando ou omitindo informações e elementos. O Vue possui uma grande quantidade de diretivas padrões, mas nos últimos capítulos veremos que também podemos criar uma diretiva própria customizada.

v-text — Exibindo os valores dos dados

Listagem de tarefas

Defina uma descrição

Figura 4.2: Resultado do exemplo com v-text

Essa diretiva pode ser usada de duas maneiras. Usamos a `v-text` para mostrar o valor do nosso atributo `titulo` e a interpolação do Mustache para exibir nosso `subtitulo`. Veja que o funcionamento, nos dois casos, é o mesmo, apenas o modo de programar se altera.

```
<template>
  <h1 v-text="titulo"></h1>
  <h2>{{ subtitulo }}</h2>
</template>
```

Ao executar o projeto com o código feito até então, você encontrará uma tela em branco. Mas se abrir o console do navegador, verá que temos um erro: *"Component template should contain exactly one root element. If you are using v-if on multiple elements, use v-else-if to chain them instead"*.

Como podemos ver, a mensagem é bastante intuitiva. Ela está falando que o componente só pode ter **exatamente** um elemento raiz, ou seja, a tag `template` pode ser composta apenas por um único filho, como vimos anteriormente. Há uma exceção, quando temos condicionais `if/else`, mas deixaremos isso para depois.

Para resolver o erro, adicionaremos uma `div` como elemento raiz do componente:

```
<template>
  <div id="app">
    <h1 v-text="titulo"></h1>
    <h2>{{ subtitulo }}</h2>
  </div>
</template>
```

Note também que, nessa `div`, coloquei um atributo `id` com valor de `app`. Isso é necessário porque estamos programando no `App.vue`, que é nosso componente representante do layout (vide capítulo anterior). É obrigatório informar o seletor da nossa instância raiz definida no arquivo `main.js`, que por padrão é `#app`, para que o Vue consiga identificar onde a aplicação vai se iniciar.

Agora sim! No navegador, podemos ver o valor das variáveis sendo exibido dentro das tags de título do HTML. Note que nesse caso não precisamos citar o `this` para acessar o dado do componente. Isso acontece porque as diretivas já tentam pegar

esses dados no componente atual por padrão.

v-once — Mostrando dados apenas uma vez

Essa diretiva tem como objetivo desativar a reatividade em um elemento. Por exemplo, ao incluí-la em um parágrafo, este será carregado apenas uma vez, e não será alterado posteriormente.

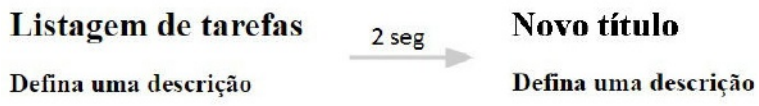

Figura 4.3: Resultado do exemplo acrescentando setTimeout

Note que o texto do `h1` será alterado de acordo com a variável se dentro do nosso componente definirmos um método `mounted`, que funciona como um construtor e será invocado quando o componente for exibido na tela. Ele trocará o valor do título dois segundos após o componente ser montado, com o `setTimeout`, fazendo assim um dos papéis da reatividade.

```
export default {
  ...
  data () {...},
  mounted() {
    setTimeout( () => this.titulo = "Novo título", 2000 )
  }
}
```

Figura 4.4: Resultado do exemplo acrescentando v-once

Em alguns casos, queremos que o texto do template não se

altere com a mudança da variável que ele representa, e é isso o que essa diretiva faz. Caso adicionemos o `v-once` ao lado da diretiva `v-text` do código anterior, a mensagem do `h1` não vai mudar, mesmo depois de dois segundos.

```
<h1 v-once v-text="titulo"></h1>
```

v-if — Controlando o que é mostrado

Muitas vezes usamos o `v-if` para definirmos o layout, ocultando ou mostrando informações para o usuário. Ele se define como um controlador do fluxo, no qual podemos dizer se o sistema seguirá o caminho X ou o Y a partir dele.

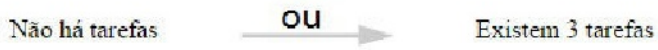

Figura 4.5: Resultado do exemplo com v-if

Não é necessário dizer o que o `if` faz, mas com ele podemos fazer com que, se o tamanho da lista de tarefas for zero, mostremos uma mensagem de que não há tarefas. Mas digamos que existam X tarefas:

```
<div v-if="tarefas.length <= 0">
  Não há tarefas
</div>

<div v-if="tarefas.length > 0">
  Existem {{tarefas.length}} tarefas
</div>
```

Caso você coloque um item dentro do `Array` de tarefas, essa mensagem mudará, informando quantas tarefas temos nessa lista.

v-for — Percorrendo dados

Imagine que temos uma lista com três dados e precisamos exibi-los na tela. Podemos simplesmente mostrar um por um, usando três linhas para isso, e cada linha exibindo um item da lista. Mas se tivermos uma lista dinâmica, com dez ou cinquenta itens? Então, usamos o `v-for` para percorrer automaticamente cada item dessa lista, desde o primeiro ao último elemento.

Você que já tem experiência em programação saberá o que é um `for`, então vamos colocar um dentro do segundo `if` do código anterior. Ele percorrerá toda nossa lista de tarefas, de modo que podemos ver uma lista HTML com cada um de seus objetos no navegador.

```
<div v-if="tarefas.length > 0">
  Existem {{tarefas.length}} tarefas

  <ul>
    <li v-for="tarefa in tarefas"> {{ tarefa }} </li>
  </ul>
</div>
```

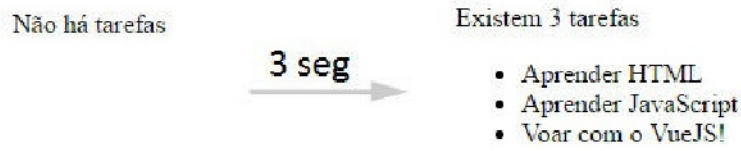

Figura 4.6: Resultado do exemplo com v-for

Dentro de `mounted`, vamos alterar nosso `setTimeout` para que, apenas depois de três segundos, seja carregada a lista de tarefas e exibida na tela. Isso nos possibilitará ver o `for` trabalhando, e também será necessário para as diretivas seguintes.

```
mounted() {
  setTimeout( () => {
    this.titulo = "novo titulo"
```

```
    this.tarefas = ['tarefa 1', 'tarefa 2', 'tarefa 3', 'tarefa 4
]
  }, 3000 )
}
```

v-model — Vínculo entre a interface e o atributo

Muitas vezes precisamos pegar os valores de um elemento da tela. Por exemplo, em um formulário, temos de pegar os dados digitados pelo usuário e executar uma ação com eles, correto? Fazemos isso com o `v-model`, que tem o objetivo de vincular o valor de um elemento com uma variável no seu código.

Ou seja, se o valor de um `input` for ABC, o da variável que passamos no `v-model` também será ABC. Vejamos isso funcionando na prática:

Figura 4.7: Resultado do exemplo com v-model

Essa diretiva é normalmente usada em campos de formulário para vincular o valor desse campo com o de um atributo no nosso `data`. Nós podemos manipular o valor do nosso subtítulo com um input de texto, assim a descrição digitada pelo usuário será o nosso subtítulo.

```
<input v-model="subtitulos" type="text" placeholder="Descrição">
```

Ao ver o navegador, teremos um novo `input` na tela. Porém, ao digitar algo nele, nada vai acontecer. Se observarmos no console, temos mais um erro: *"[Vue warn]: Property or method "subtitulos" is not defined on the instance but referenced during*

render".

Diferente do erro que vimos anteriormente, esse não compromete a execução do código, pois não é uma falha fatal, mas sim um aviso. Note que ele está nos informando que a propriedade (ou método) `subtitulos` não foi definida ou instanciada durante a renderização do componente.

Se prestarmos atenção no nosso `data`, veremos que realmente não temos um atributo com esse nome, e sim `subtitulo`. Basta modificarmos o nome dessa variável no `v-model` para obtermos o resultado esperado.

```
<input v-model="subtitulo" type="text" placeholder="Descrição">
```

Ao fazer isso, você já pode ver que o valor padrão do campo será igual ao valor da variável `subtitulo`. E ao alterar esse texto no `input`, a modificação será propagada no nosso subtítulo. Esse aspecto é o **data binding**, que faz parte da reatividade.

v-bind — Adicionando detalhes à página

O `v-bind` é responsável por mapear atributos HTML, assim você pode alterar valores de atributos dos elementos de acordo com alguma condição dada no seu script. Esses atributos podem ser personalizados ou os padrões do HTML: `id`, `class`, `placeholder` etc.

Figura 4.8: Resultado do exemplo com v-bind

Por exemplo, vamos desabilitar o `input` criado anteriormente até que a nossa lista de tarefas não seja apresentada na tela. Isto é, ele ficará desabilitado enquanto o comprimento da lista for zero.

```
<input v-bind:disabled="tarefas.length == 0" v-model="subtitulo"
type="text">
<!-- Ou usando apenas dois pontos -->
<input :disabled="tarefas.length == 0" v-model="subtitulo" type="
text">
```

Vale lembrar que essa diretiva possui argumentos, nos quais você poderá mostrar qual atributo do HTML será alterado de acordo com o valor entre aspas. Logo, como informado, argumento pode ser qualquer atributo HTML ou personalizado, não só o `disabled`, mas também o `maxlength`, `placeholder`, `autocomplete` entre outros.

v-on — Executando ações

A diretiva `v-on` é responsável por invocar eventos, e funciona como uma *escuta* para os elementos da nossa tela. Ou seja, caso tenhamos uma `div`, podemos declarar uma escuta do tipo `click`, assim, sempre que clicarmos nela, vamos executar o evento informado no `v-on`.

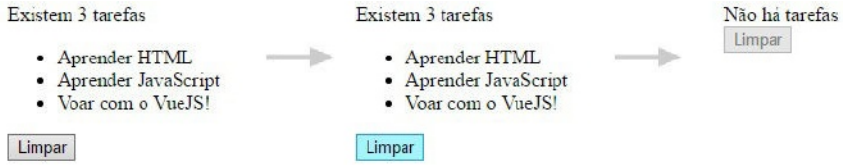

Figura 4.9: Resultado do exemplo com v-on

Criaremos um botão e, ao clicar nele, poderemos limpar nossa lista de tarefas. Limpando essa lista, o `input` também será desativado, pois o comprimento dele passa a ser zero. Com isso, veremos automaticamente a reatividade funcionando, porque o atributo `tarefas` foi alterado. Logo é enviada uma notificação para o observador do componente (que usa esse atributo) se recarregar.

```
<button v-on:click="tarefas = []">Limpar</button>
<!-- Ou usando apenas arroba -->
<button @click="tarefas = []">Limpar</button>
```

Mais uma vez, temos um argumento, nesse caso o `click` . Ele pode ser alterado para suprir outras necessidades, como `change` , `keyup` etc.

4.3 TESTE SEU AVANÇO

Neste capítulo, inauguro a seção *Teste seu avanço*, para já começarmos a praticar com o Vue. É essencial que você treine tudo o que aprendemos. Esta seção contém alguns exercícios para que você, leitor, possa praticar, e ela estará presente ao final de cada capítulo a partir de agora.

As respostas para os exercícios estão em um repositório separado do GitHub: https://github.com/leonardovilarinho/vue-

livro-avanco. Nele, além de comparar as respostas, você poderá tirar dúvidas comigo, abrindo um *issue*. Vamos começar a primeira lista de exercícios!

Memorize a teoria

1. Para que serve o atributo `data` de um componente?
2. O que são diretivas? Defina cada diretiva a seguir:
    - v-once
    - v-if
    - v-text
    - v-for
    - v-model
    - v-on (@)
    - v-bind (:)

Na prática

1. Com a diretiva que mostra um dado na tela, crie um atributo chamado `nome` com um valor inicial igual a `Mané Garrincha` e exiba-o na tela.
2. Usando o `setTimeout`, altere o conteúdo de `nome` para `Mineirão` quatro segundos após o componente ser montado.
3. Crie uma tag `small` mostrando o conteúdo da variável `nome` no seu estado inicial. Faça com que, mesmo depois dos quatro segundos, ela continue exibindo o nome `Mané Garrincha`.

4. Crie um botão que apague a tag `small` quando ela estiver na tela e mostre-a quando não estiver. Use a diretiva `v-if` e `v-on` para isso.

5. Habilite o botão criado anteriormente apenas após passarem os quatro segundos do nosso `setTimeout`.

6. Crie um `input` que lhe permita editar o conteúdo da variável `nome`.

7. Declare um `Array` chamado `cadeiras` e coloque os seguintes valores nele:

    - 30.000 brancas
    - 30.000 cinzas
    - 10.000 vips

    Mostre esses dados na tela para o usuário, dentro de uma lista `ul`.

Capítulo 5

MANIPULANDO DADOS

No capítulo anterior, vimos como declaramos variáveis no atributo `data` do Vue. Elas são usadas para armazenar dados que serão utilizados naquele componente. Agora, assim como em qualquer outro sistema, às vezes temos a necessidade de alterar os valores dessas variáveis, pois assim estaremos manipulando o estado atual do sistema, seja apagando uma mensagem de erro exibida na tela ou algo do tipo.

O Vue fornece diferentes maneiras de se manipular uma variável, e veremos todas elas no decorrer do capítulo. O importante agora é saber que cada uma delas tem o seu próprio objetivo, como: filtrar valores das variáveis; agrupá-las ou formatá-las; observar os valores das variáveis etc. Esse tratamento dividido possibilita um ganho na performance do sistema, pois a variável é tratada da forma que queremos, apresentando apenas o comportamento desejado.

Assim como fizemos anteriormente, teremos de iniciar um projeto para desenvolver os exemplos deste capítulo. A cada novo capítulo criaremos outro projeto, assim você terá todo o guia separado, no final, para consultá-lo sempre.

```
vue init webpack-simple manipulando-dados
cd manipulando-dados
```

```
npm install
npm run dev
```

Com o projeto em branco já criado, veremos quais são as formas que o Vue apresenta para realizar a manipulação de dados, quais são suas indicações de uso e um exemplo básico de cada uma. O objetivo é que você, ao término do capítulo, possa empregar cada maneira da forma mais correta, sabendo claramente quais as diferenças entre elas.

5.1 METHODS — BLOCOS DE AÇÕES

Os manipuladores de dados mais simples que o Vue nos apresenta são os métodos. Como em toda linguagem, métodos/funções são responsáveis por guardar um trecho de código, recebendo ou não parâmetros. Eles executam esse trecho de código, e retornam ou não um valor resultante quando invocado.

No capítulo anterior, já usamos métodos, mas você não percebeu. Na diretiva `v-on`, igualamos a lista de tarefas a um `Array` vazio.

```
<button @click="tarefas = []">Limpar</button>
```

Esse é um método anônimo de uma linha, que pode ser colocado direto sem precisar declará-lo.

Mas agora veremos um exemplo usando essa propriedade muito útil. Teremos dois botões e, ao clicar no `+`, o número do resultado será incrementado; clicando em `-`, esse número será decrementado.

Para isso, vamos ter no `template` do componente um

parágrafo `p` que mostrará o resultado armazenado na variável `total` e dois botões. O primeiro chamará o método `calcula` com o parâmetro `+`, para informar que devemos realizar a adição. Já o segundo chamará a mesma função, porém com o parâmetro `-`, indicando a subtração.

Por fim, declararemos o método `calcula` que acionará a soma ou a subtração. Com o código posterior, teremos nosso componente completo e funcionando.

```
<template>
  <div id="app">
    <p>{{ total }}</p>

    <button @click="calcula('-')"> - </button>
    <button @click="calcula('+')"> + </button>
  </div>
</template>

<script>
export default {
  name: 'lv-contador',
  data () {
    return {
      total: 10
    }
  },
  methods: {
    calcula( sinal ) {
      this.total = (sinal == '-') ? this.total - 1 : this.total +
1
    }
  }
}
</script>
```

Note que os métodos ficam dentro da propriedade `methods` de um componente e, com isso, temos um bloco de código separado dos demais, mostrando que ali dentro teremos apenas

métodos. Nessa propriedade, vamos declarar funções, como a `calcula`, da maneira como o JavaScript as disponibiliza, desde que cumpram o seu objetivo.

Com a *live* fornecida pelo comando `npm run dev` ainda ativa no terminal, o navegador vai nos apresentar o resultado da figura seguinte. Note que, se clicarmos no botão `+`, o total será aumentado, e se apertarmos `-`, será diminuído.

Figura 5.1: Resultado do exemplo de methods

5.2 FILTERS — FILTRANDO DADOS

Ao contrário do atributo que vimos na seção anterior, os filtros são pouco utilizados no Vue. Eles devem ser empregados apenas para formatar o dado que será exibido no template, e para nada mais do que isso.

Um exemplo muito usado para mostrar o papel do filtro é um cenário no qual temos de exibir o nome completo do usuário, mas formatando-o para que as primeiras letras fiquem maiúsculas.

No `template`, criaremos dois parágrafos `p`: o primeiro exibindo a variável `nome` e o segundo invocando o filtro passando a variável como parâmetro.

```
<p>Nome Iniciado: {{ nome }}</p>
<p>Nome Filtrado: {{ formataNome(nome) }}</p>
```

No `script` do nosso componente, vamos adicionar um atributo chamado `filters`. Assim como o `methods`, ele nos

fornecerá certa organização, dado que saberemos que todos os filtros estarão lá dentro.

Dentro dele, declaramos o método que será nosso filtro. Ele obrigatoriamente deve receber um parâmetro que será o valor a ser filtrado. Nesse método, cortaremos a `String` recebida e transformaremos a primeira letra de cada palavra em maiúscula.

```
export default {
...
data() {
  return {
    ...
    nome: 'jose antonio'
  }
}
...
filters: {
  formataNome( valor ) {
    console.log('executando filter')
    valor = valor.toLowerCase()
    let corta = valor.split(' ')
    let resultado = ''
    for (let nome of corta)
    resultado += nome.charAt(0).toUpperCase() + nome.slice(1) + '
 '
    return resultado
  }
}
```

Ao tentar ver o código funcionando no navegador, teremos um erro informando que o método `formataNome` não foi definido ou instanciado quando o componente foi renderizado. Isso se deve ao fato de se tratar de um filtro. Apesar de ser declarado como método, seu uso não é igual ao de um. O uso correto de filtros é feito dentro de interpolações Mustache, em que devemos informar primeiro a propriedade a ser filtrada, e depois o nome do filtro que vamos usar, utilizando o `pipe` para separar essas instruções.

```
<p>Nome Iniciado: {{ nome }}</p>
<p>Nome Filtrado: {{ nome | formataNome }}</p>
```

Agora, ao ir para o browser, teremos nosso exemplo funcionando! Será mostrado o valor da variável `nome` no seu estado inicial e, logo abaixo, o seu estado já filtrado, ou seja, com o nome capitalizado. Podemos ver na figura:

Nome Iniciado: jose antonio

Nome Filtrado: Jose Antonio

Figura 5.2: Resultado do exemplo com filters

5.3 COMPUTED PROPERTIES — MODIFICANDO DADOS PARA EXIBI-LOS

Computed properties (propriedades computadas) geram muita confusão, pois têm um papel parecido com o dos filtros, mas na verdade possuem algumas diferenças. Enquanto os filtros são métodos que formatam um dado recebido para que ele seja melhor apresentado na tela, as propriedades computadas representam um dado, modificando-o ou agrupando-o a outros dados, como se fosse um formatador apenas para uma variável, e não um método.

No exemplo desta seção, continuaremos aquele código usado nos filtros. Vamos acrescentar um parágrafo `p` para mostrar a propriedade computada e um `input` com o `v-model` do dado que ela representa.

```
...
<p>Nome Computado: {{ nomeFormatado }}</p>
<label>Input a computar</label>
<input v-model="nome" type="text">
```

Após isso, no `script`, criaremos o atributo `computed`. Dessa forma, o Vue identifica que tudo o que está ali dentro são propriedades computadas, assim como fizemos anteriormente.

Dentro desse atributo, criamos um método que retorna o dado formatado. Note que não passamos esse dado por parâmetro, pois essa propriedade só pode representar um único dado, e retorná-lo computado e formatado.

```
export default {
  ...
  data () {
    return {
      ...
      nome: 'jose antonio'
    }
  },
  ...
  computed: {
    nomeFormatado() {
      console.log('executando computed')
      return this.nome.toUpperCase()
    }
  }
}
```

Teremos um resultado assim:

Figura 5.3: Resultado do exemplo com computed properties

As propriedades computadas possuem melhorias de desempenho quando comparadas a métodos e filtros. Para vermos isso funcionando, abra o console e recarregue a página. Veremos que, ao iniciar o componente, tanto o filtro quanto a propriedade

computada são chamados; e ao alterar o atributo `nome`, ambos também são. Isso acontece porque a *computed* representa esse dado, e temos um elemento no qual esse atributo usa o filtro `formataNome`, mas ao pressionarmos o botão de `+` — usado no exemplo de `methods`, que não tem nada a ver com o atributo `nome` —, nosso filtro ainda continuará sendo executado.

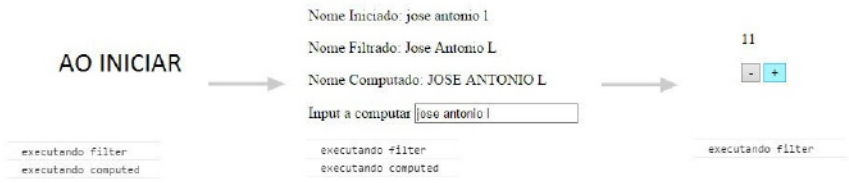

Figura 5.4: Verificando a teoria da performance

Podemos concluir que os filtros são sempre reexecutados, enquanto as *computed properties* são chamadas apenas quando o atributo que elas representam sofre alguma alteração. Logo, elas são melhores no desempenho.

Mas não é só isso, as propriedades computadas permitem fazer ainda mais. Também podemos manipular *setters*: ao trocarmos o valor da propriedade `nomeFormatado`, por exemplo, lemos apenas seus três primeiros caracteres.

Para criar um *setter*, teremos de mudar a estrutura da propriedade criada anteriormente. Ela não será mais um método, e sim um objeto comum. Dentro desse objeto, vamos ter os métodos `get` e `set`. O primeiro será executado quando pegarmos o valor dessa propriedade e o segundo assim que o valor de `nomeFormatado` for alterado.

```
nomeFormatado: {
```

```
  get: function () {
    console.log('executando computed')
    return this.nome.toUpperCase()
  },
  set: function (novoValor) {
    this.nome = novoValor.substring(0, 3)
  }
}
```

Ao trocar o `model` do nosso campo para `nomeFormatado`, você poderá ver o `set` funcionando, pois agora nosso campo de texto representará o valor já computado, e não sua origem (a variável `nome`).

```
<input v-model="nomeFormatado" type="text">
```

Com a live criada pelo `npm run dev` ainda ativa, em qualquer navegador, vamos entrar no link *localhost:8080*. Note que, ao digitar o conteúdo no `input`, apenas as três primeiras letras serão lidas, pois nosso `set` está sendo executado.

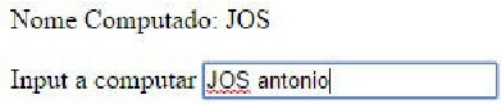

Figura 5.5: Colocando um set na propriedade e testando com o input

5.4 WATCHERS — DADOS QUE PODEM ESPERAR

Watchers (ou observadores) funcionam basicamente como as propriedades computadas da seção anterior, pois são observadores de propriedades e mudam seu valor de acordo com o dado que representam. Nesse caso, temos acesso ao histórico do valor da propriedade que está sendo observada, ou seja, a cada alteração, podemos capturar o novo e o antigo valor da variável, tomando

ações customizadas a partir disso.

Isso é útil para executar operações assíncronas, como requisição à API externa, já que precisamos observar se a API já retornou os dados e, a partir daí, mexer no layout da aplicação para mostrar os dados recebidos e alterar o estado de botões e outros elementos.

Podemos ter um campo de texto e, enquanto o usuário estiver digitando, exibir uma resposta "Aguardando o término da digitação...". Quando ele parar de digitar, apresentamos outra, dizendo "Terminou de digitar...".

Figura 5.6: Funcionamento de um watch

Para isso, no nosso template, criaremos o campo `input` que representa a variável `busca`, responsável por pegar o texto do usuário. Também precisaremos de um parágrafo para exibir a mensagem de resultado, então criamos uma tag `p` exibindo o valor da variável `resultado`.

```
...
<input v-model="busca" type="text">
<p v-text="resultado"></p>
```

Agora temos de declarar essas variáveis no nosso atributo `data`. Feito isso, criaremos o método `recolheResposta` para simular uma API. Nele, armazenaremos o valor digitado, esperando meio segundo para ver se o usuário terminou ou não de escrever e, por fim, mostrar o resultado adequado na tela.

Assim, vamos criar o atributo `watch`, onde ficarão todos os nossos observadores personalizados, e dentro dele, uma função com o mesmo nome da variável que ele espionará — nesse caso, a `busca`. Esse método pode receber dois parâmetros: o primeiro representa o valor que a variável `busca` vai receber, e o segundo, o valor antigo que estava guardado nela.

Nesse caso, apenas mudaremos o valor do `resultado` e chamaremos o método criado anteriormente.

```
data () {
  return {
    ...
    resultado: '',
    busca: ''
  }
},
....
watch: {
  busca: function (novoValor, valorAntigo) {
    this.resultado = 'Aguardando o término da digitação...'
    this.recolheResposta()
  }
},
methods: {
  ...
  recolheResposta() {
    let valor = this.busca
    setTimeout( () => {
      if(valor == this.busca)
        this.resultado = 'Terminou de digitar...'
    }, 500)
  }
}
```

Armazene na mente

Note que nosso `watch` e a variável que queremos observar possuem o mesmo nome. Isso é um requisito para que possamos ter sucesso na operação, pois assim o Vue vincula tal variável a tal observador, fazendo então o citado observador personalizado.

5.5 TESTE SEU AVANÇO

Chegou a hora de, mais uma vez, testar se você está adquirindo o conhecimento esperado até então. Caso não consiga realizar alguns exercícios, releia o capítulo, busque entender a resposta do exercício no GitHub e me pergunte. Vamos lá!

Memorize a teoria

Para as perguntas a seguir, responda se você faria algo usando *métodos*, *filtros*, *propriedades computadas* ou *observadores*.

1. Vamos imaginar um componente no qual você tenha as variáveis `nome` e `sobrenome`. Qual maneira vista para manipular dados você usaria para exibir o *nome completo* na tela? Explique-se.

2. Caso você tenha uma variável do tipo `double`, deve formatá-la e exibi-la como reais (R$). O que você usaria para formatar esse dado? Justifique.

3. Imaginando que temos uma variável que representa a hora

do sistema, quando ela se alterar, automaticamente você terá de ver se os minutos atuais são 00 ou 30, para então tomar uma ação. O que você usaria para solucionar esse problema? Justifique.

4. Temos uma tag `a` em um componente e, ao passar o mouse em cima dela, o sistema deve exibir um alerta na tela. O que utilizaríamos para exibir esse alerta? Por quê?

Na prática

1. Crie um `input`, que represente a variável `nome`, e abaixo dele uma tag `small`, que exiba o valor da variável `status` iniciada valendo `'Digite seu nome acima'`.

2. Faça com que, ao apertarmos um botão, seja exibido um `alert 'Olá, '` concatenado com o valor da variável `nome`.

3. Com um observador, faça com que, ao estarmos digitando no `input`, o valor da variável `status` seja `'Recebendo seu nome...'`. Ao terminar a digitação, faça com que o valor da variável `status` seja vazio.

4. Após o término da digitação, exiba o `nome` digitado com letras maiúsculas.

5. Também após a digitação, faça com que seja exibida a quantidade de caracteres do `nome` digitado.

Capítulo 6

COMPONENTES JUNTOS SÃO MAIS FORTES

Já falamos muito sobre componentes no decorrer desta obra. São a parte principal do Vue, já que o framework é inteiramente composto por eles. Na verdade, o Vue nada mais é do que um gerenciador de componentes, em que você tem um componente raiz e um inicial para ser renderizado, e a partir dele chamamos outros para compor nossa aplicação.

Neste capítulo, veremos como dividir um componente em vários. Antes tínhamos um só, o `App`, e ele era responsável por fazer diversas coisas, como listar tarefas, trocar um subtítulo e apagar a lista. Note que isso atinge um dos pilares da programação, a **coesão**, pois, para um código ser coeso, ele deve ter apenas uma responsabilidade.

Esse pilar deve ser levado a sério quando você está usando o Vue, pois componentes muito grandes serão de extrema complexidade e de fraca manutenção. Ao quebrar um componente em vários, superamos esse problema e, de bônus, aumentamos o reúso do nosso sistema, já que partes com apenas um objetivo podem ser usadas para compor outras N partes.

Elaboraremos um novo projeto. Caso ainda tenha dificuldade nessa parte, relembre o capítulo *Preparando e iniciando o ambiente*. Neste projeto, teremos um componente que representa um time de futebol, cujo perfil será formado por `nome` e `estadio`. Além disso, teremos um campo de busca, no qual, ao digitar o nome de um time, retornaremos como resultado o de seu rival.

Para originar um novo componente, devemos entrar na pasta `src` e criar um arquivo `.vue` com o nome do nosso componente, por exemplo `LVTime.vue`. Devemos escrever nesse arquivo a estrutura padrão de um componente.

```
<template>
    <div>
    </div>
</template>

<script>
export default{
    name: 'LVTime',
    data() {
        return {

        }
    },
}
</script>

<style>
</style>
```

Note que já defini um nome para ele seguindo os padrões ensinados no capítulo *Preparando e iniciando o ambiente*. Coloquei também uma `div` como seu elemento raiz e declarei um atributo `data` vazio.

Para automatizar esse processo, podemos criar um *snippet* no

Sublime, com o código seguinte:

```
<snippet>
    <content><![CDATA[
<template>
    <div>
    </div>
</template>

<script>
export default{
    name: '${1:component-name}',
    data() {
        return {

        }
    },
}
</script>

<style>
</style>
]]></content>
    <tabTrigger>component</tabTrigger>
</snippet>
```

Após salvá-lo, digite `component` em algum arquivo aberto no Sublime e pressione a tecla `Tab`. Você verá que já temos o esqueleto do novo componente criado, e o cursor do mouse já estará no local do `name`. Essa dica evidencia que o segundo capítulo não foi em vão, pois isso lhe ajudará a ser produtivo.

No `script` do componente criado, iniciaremos o projeto. Nele teremos o perfil de um time e um campo de busca, no qual poderemos consultar rivais de diversas equipes.

Inicialmente, precisamos declarar no `data` os dados de nosso time, no caso `nome` e `estadio`. Assim teremos os atributos que compõem o perfil da equipe já prontos para serem usados.

Também precisaremos de uma variável para armazenar o nome da equipe que estamos buscando, então declaramos a `busca`. Por fim, vamos criar um método que recebe o nome de um time e retorna o nome de seu rival.

```
...
data() {
  return {
    nome: 'Cruzeiro',
    estadio: 'Mineirão',
    busca: '',
  }
},
  methods: {
    rival(time) {
      switch( time.toLowerCase() ) {
        case 'corinthians':
          return 'palmeiras'
        case 'santos':
          return 'são paulo'
        case 'flamengo':
          return 'vasco'
        case 'cruzeiro':
          return 'em minas é marmelada'
        default:
          return 'não encontrado'
      }
    }
  }
...
```

Com isso, já temos todo o nosso `script` pronto, com os dados referentes ao perfil de um time e o método que realiza a busca de um rival. Assim, suprimos os requisitos do projeto especificados no início do capítulo.

No `template`, vamos colocar um `input` para realizar a busca do rival de um time qualquer, representado pela variável `busca`. Também acrescentaremos um parágrafo `p` para mostrar

qual é o rival daquele time e, por fim, exibir o perfil do nosso, mostrando os dados `nome` e `estadio`.

```
...
<h3>Buscando rivais</h3>
<label>Time:</label>
<input type="text" v-model="busca">
<p>Rival: {{ rival(busca) }}</p>

<h3>Perfil</h3>
<p>Time: {{ nome }}</p>
<p>Estádio: {{ estadio }}</p>
...
```

Figura 6.1: Tela inicial do Vue.js

Ao rodar o projeto com `npm run dev`, note que a tela anterior será exibida. Isso é mostrado porque o componente inicial da aplicação por padrão é o `App.vue`. Vamos abri-lo, apagar seu conteúdo e importar o componente `LVTime` para ele, assim nosso novo componente será exibido na tela. O `App` ficará assim:

```
<template>
  <div id="app">
    <LVTime></LVTime>
  </div>
```

```
</template>

<script>
import LVTime from './LVTime.vue'

export default {
  name: 'app',
  components: {LVTime}
}
</script>
```

Note que temos um novo atributo no `script`, um objeto chamado `components`. Ali definimos quais componentes serão importados para o atual. Após importá-lo, podemos usar a tag `<componente-importado>` para exibi-lo na tela, assim como fizemos na linha `<LVTime> </LVTime>`.

Ao ver novamente o navegador, nosso exemplo já estará funcionando. O resultado será o seguinte:

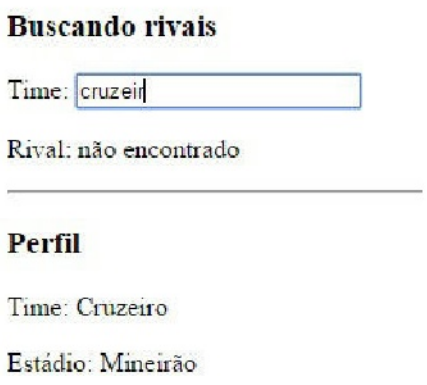

Figura 6.2: Componente LvTime já funcionando

Já será possível visualizar as informações do perfil do time e também consultar por rivais. Por curiosidade, duplique a linha em

que exibimos nosso componente, deixando-a assim:

```
...
 <LVTime></LVTime>
 <LVTime></LVTime>
...
```

Após isso, esse componente será renderizado duas vezes na tela, porque o chamamos essas duas vezes.

Voltando ao assunto principal, perceba que nosso componente não está coeso. Ele lida com o perfil do time e também com a busca de um rival, logo temos duas responsabilidades distintas. Isso é errado, pois o componente fica totalmente complexo e não será reusável. Dessa maneira, sempre que importarmos o perfil do time, teremos obrigatoriamente de importar também o campo de busca por rivais, e quase nenhum sistema tem uma tela sempre com esses dois aspectos.

Para solucionar esse problema, podemos dividi-lo em dois componentes distintos. Criaremos dentro do diretório `src` os arquivos `LVBusca.vue` e `LVPerfil.vue`, colocando o esqueleto de componente em ambos.

No componente `LVBusca`, vamos colocar apenas aquilo que se refere à busca de um rival, ou seja, o `input`, o parágrafo com seu rival, a variável `busca` e o método `rival`. Ele ficará assim:

```
<template>
    <div>
        <h3>Buscando rivais</h3>
        <label>Time:</label>
        <input type="text" v-model="busca">
        <p>Rival: {{ rival(busca) }}</p>
    </div>
</template>
```

```
<script>
export default{
    name: 'lv-busca',
    data() {
        return {
            busca: ''
        }
    },
    methods: {
        rival(time) {
            switch( time.toLowerCase() ) {
                case 'corinthians':
                    return 'palmeiras'
                case 'santos':
                    return 'são paulo'
                case 'flamengo':
                    return 'vasco'
                case 'cruzeiro':
                    return 'em minas é marmelada'
                default:
                    return 'não encontrado'
            }
        }
    }
}
</script>
```

Fazemos o mesmo para o componente `LVPerfil`, colocando dentro dele apenas aquilo relacionado ao perfil de um time. Ou seja, declaramos as variáveis `nome` e `estadio` no `data`, e no `template` vamos exibir os seus valores:

```
<template>
    <div>
        <h3>Perfil</h3>
        <p>Time: {{ nome }}</p>
        <p>Estádio: {{ estadio }}</p>
    </div>
</template>

<script>
export default{
    name: 'lv-perfil',
```

```
    data() {
        return {
            nome: 'Cruzeiro',
            estadio: 'Mineirão',
        }
    },
}
</script>
```

Note que nossos componentes individualmente ficaram menores e respectivamente coesos, pois um componente apenas busca o rival de um time e o outro apenas exibe o perfil de um. Essa decisão também afeta o reúso do código.

Com o `LVTime` tal como visto antes, não poderíamos usá-lo em diversas páginas, mas agora, com ele separado, podemos reutilizar um mesmo componente. Um exemplo é o `LVBusca` que, por ser um tipo de busca, deveria estar presente em todas as páginas do site, enquanto o `LVPerfil` seria uma página de perfil.

Por fim, temos de exibir esses componentes na tela. Usaremos novamente o componente inicial `App` para fazer isso.

```
<template>
  <div id="app">
    <lvBusca></lvBusca>
    <hr>
    <lv-perfil></lv-perfil>
  </div>
</template>

<script>
import lvBusca from './LVBusca.vue'
import lvPerfil from './LVPerfil.vue'

export default {
  name: 'app',
  components: {lvBusca, lvPerfil}
}
</script>
```

Note que agora temos duas tags personalizadas, pois importamos dois componentes diferentes. Perceba também que, na importação, usei o padrão de nomenclatura *camelCase*, que usa a primeira palavra da variável em letras minúsculas e, a partir da segunda, coloca a primeira letra do termo em maiúscula.

Quando seguimos esse padrão, o Vue nos fornece dois tipos de tag: uma usando o *camelCase*, utilizada para importar o `lvBusca`, com `<lvBusca></lvBusca>`; e outra com o *kebab-case*, para importar o `lvPerfil` com `<lv-perfil></lv-perfil>`.

Essa nomenclatura é pessoal, você não é obrigado a usá-la. Mas a indicação é que empregue apenas um tipo no mesmo projeto, pois, como vimos no exemplo anterior, ficará um pouco estranho ter tags com padrões de nome diferentes.

Para finalizar o capítulo, vamos relembrar o passado. Você se lembra da imagem *Fluxo de funcionamento do Vue*, apresentada no capítulo *Entendendo o funcionamento*? Nela tínhamos uma árvore de componentes que acabava formando uma interface web, e cada componente seria um elemento na tela do usuário.

Com as extensões instaladas no capítulo *Preparando e iniciando o ambiente*, podemos ver essa árvore no nosso projeto atual. Para isso, no navegador, pressione `Ctrl+Shift+i` para abrir o inspecionador de elementos. Nele teremos algumas abas, como: `Element`, `Console` e `Source`. Nenhuma dessas nos interessa agora, vamos selecionar primeiramente a aba `DejaVue`, e veremos esta árvore:

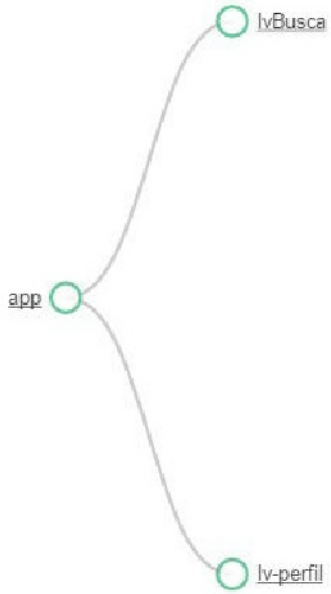

Figura 6.3: Árvore de componentes DejaVue

Podemos também selecionar a aba Vue , na qual, à esquerda, teremos um menu. Ao abri-lo, verá que ele forma outro tipo de árvore:

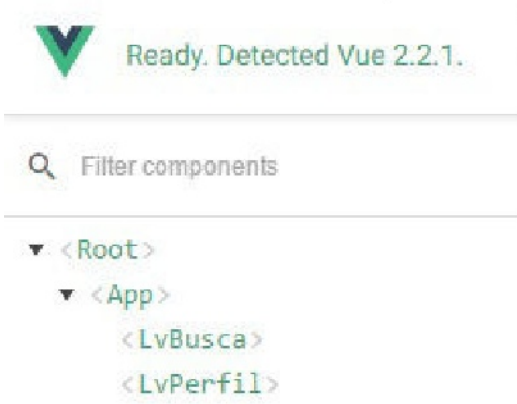

Figura 6.4: Árvore de componentes VueDevTools

Essas ferramentas nos ajudam a ter uma visão geral da aplicação. Caso você selecione alguma linha ou bolinha da árvore, será apresentado um painel com as informações daquele componente. Por exemplo, veja o componente `LvBusca` :

Figura 6.5: Inspecionando um componente

Note que temos nosso `data` no canto direito e, caso digitemos outra coisa no `input` da tela, ele também será trocado em tempo real.

Para que você memorize tudo o que foi aprendido até aqui, use a imaginação, crie algum pequeno projeto. Com isso que aprendemos, já podemos fazer muita coisa legal no Vue. Que tal um pequeno jogo de perguntas e respostas? Ou até mesmo um cronômetro com marcadores de tempo?

6.1 TESTE SEU AVANÇO

Vamos mais uma vez testar seu conhecimento. Um capítulo pequeno exige um número menor de exercícios, por isso faremos apenas dois.

Memorize a teoria

1. Em um cenário fictício, temos um sistema pequeno de vendas, no qual a página inicial deve apresentar: lista de categorias; produtos em destaque; últimos comentários;

menu superior; barra de busca. Levando isso em consideração, como você separaria esses elementos? Quais componentes você criaria?

Na prática

1. Faça uma tela apresentando uma lista de postagens imaginárias e, ao clicar em uma delas, exiba seu conteúdo na tela. Fique à vontade para usar a imaginação, pois o objetivo desse exercício é que você crie os componentes dessa tela com a maior coesão possível.

Capítulo 7

REUTILIZANDO COMPONENTES

Agora que já sabemos como podemos criar, importar e usar mais de um componente, focaremos em sua reusabilidade. Anteriormente criamos componentes mais coesos, veremos agora como reaproveitá-los.

Muitas vezes nos pegamos copiando e colando linhas de código de um arquivo para outro, e sabemos que isso é errado. Teremos como resultado um código duplicado, o que pode torná-lo feio, mais complexo e com manutenções difíceis.

Para evitar isso, o Vue.js nos oferece alguns recursos para que separemos e aproveitemos nosso código. Por exemplo, podemos criar componentes sem design e importar seus atributos para outros, como também deixar um componente mais aberto, passando atributos externos. E, para fechar, temos uma maneira simples de comunicação com outros componentes, integrando-os.

É isso o que veremos a seguir. Para cada exemplo, será utilizado um projeto em branco, assim posso mostrar bem separadamente cada aspecto que o Vue nos oferece para reusar.

7.1 PROPS — RECEBENDO ATRIBUTOS EXTERNOS

Inicio esta lista com as `props`, pois é algo que você realmente usará muito! Aqui nós podemos defini-las como variáveis imutáveis, isto é, que não podem ser alteradas, e serão recebidas na hora de declarar o componente, assim como os parâmetros de um método. Isso ajuda bastante na hora de reusar.

Usamos o atributo `props` de um componente para receber quaisquer dados externos, e esses dados serão usados no nosso componente. No GitHub, por exemplo, temos diversas coleções de componentes que utilizam as `props` como seu grande aspecto, usando-as para alterar o design do componente ou seu conteúdo.

Por exemplo, temos um sistema que possui diversas telas com listas de produtos, categorias etc., e precisamos de algumas listas em ordem alfabética e outras não. Muitos desenvolvedores fariam isso na mão, criando um `Array` ordenado para cada lista que deve ser exibida alfabeticamente e mostrando-o dentro de um componente. Na hipótese mais errada, teríamos algo assim:

```
...
<h4>Lista um</h4>
<ul>
  <li v-for="item in ['feijão', 'arroz', 'carne']">{{ item }}</li

</ul>
<br>
<ul>
  <li v-for="item in ['arroz', 'carne', 'feijão']">{{ item }}</li

</ul>
<br>

<h4>Lista dois</h4>
```

```
<ul>
  <li v-for="item in ['leite', 'biscoito', 'bombom', 'aveia']">{{
 item }}</li>
</ul>
<br>
<ul>
  <li v-for="item in ['aveia', 'biscoito', 'bombom', 'leite']">{{
 item }}</li>
</ul>
..
<style>
li {
    display: inline;
    list-style-type: none;
    padding-right: 20px;
    float: left;
}
</style>
```

Note que pela primeira vez usei um CSS no componente. Ele serve apenas para formatar nossa lista, exibindo-a em uma só linha. Observe também que estou mostrando algumas listas, ordenando-as manualmente. O resultado será igual ao da figura:

Lista um

feijão arroz carne

arroz carne feijão

Lista dois

aveia biscoito bombom leite

leite biscoito bombom aveia

Figura 7.1: Resultado do exemplo

Apesar de funcionar, nós temos muitos códigos duplicados e

muita programação manual, o que afetará nossa produtividade. Solucionaremos isso com um componente que representa uma lista, que receberá os dados em um `Array` aleatório e um atributo indicando se ela será ou não ordenada. Note que, no final, teremos um componente que pode representar qualquer lista do sistema, ordenando-a ou não, e exibindo-a com o resultado igual ao anterior.

Para melhorar nosso código, vamos criar um componente no diretório `src`, chamado `LvLista.vue`. No seu `script`, temos o novo atributo `props`. Ele é um `Array` em que declaramos quais propriedades externas estaremos recebendo, no caso será `lista` e `ordena`. A primeira representa nossa lista de dados a ser exibida, e a segunda, se ela será ou não ordenada.

```
export default{
    name: 'LvLista',
    props: ['lista', 'ordena'],
    created() {
        if(typeof this.ordena != 'undefined')
            this.lista.sort()
    }
}
```

Veja que também temos outro novo atributo no componente, o `created`, que é um método. Ele é chamado assim que o componente é invocado e antes de ser desenhado na tela, logo é executado antes do `mounted` que já usamos. Nesse método, vamos verificar se foi passado ou não o argumento `ordena`; caso tenha sido passado, organizaremos nossa lista em ordem alfabética automaticamente.

No nosso `template`, exibiremos nossa lista recebida, e no `style` teremos o CSS para formatar aquela lista.

```
...
<ul>
    <li v-for="item in lista">{{ item }}</li >
</ul>
...
```

No `App.vue` , vamos importar o nosso componente que representa uma lista. Para passar os atributos para ele, fazemos do mesmo jeito que no HTML. Porém, neste caso, temos um atributo dinâmico, que é a lista, então devemos usar o `v-bind` para passá-la, ficando assim:

```
<template>
    <div id="app">
        <h4>Lista um</h4>
        <lv-lista :lista="['feijão', 'arroz', 'carne']"></lv-list
>
        <br>
        <lv-lista :lista="['feijão', 'arroz', 'carne']" ordena></
lv-lista>

        <br>

        <h4>Lista dois</h4>
        <lv-lista :lista="['leite', 'biscoito', 'bombom', 'aveia'
]" ordena></lv-lista>
        <br>
        <lv-lista :lista="['leite', 'biscoito', 'bombom', 'aveia'
]" ></lv-lista>

    </div>
</template>

<script>
import LvLista from './LvLista.vue'

export default {
    name: 'app',
    components: {LvLista}
}
</script>
```

Cada vez que chamo o componente, estou passando uma lista diferente. Note que minhas listas nem estão ordenadas em ordem alfabética, mas quando passo o atributo `ordena`, o componente por si só já a exibirá ordenada. Isso se deve ao método `created` que vimos antes.

Por fim, obtivemos o mesmo resultado que no exemplo anterior, porém reutilizamos muito código, e não temos mais nada duplicado. Essa é a ideia principal do atributo `props`, mas ele vai além, já que podemos validar nossos dados externos.

Para fazer a tal validação, devemos trocar nosso `Array` `props` por um objeto, assim ele pode receber parâmetros. Aqui temos alguns deles, como o tipo, o padrão e se é ou não obrigatório. Podemos alterar a `props` do componente `LvLista` deixando-a assim:

```
props: {
    ordena: null,
    lista: {
    required: true,
        type: Array,
        default: []
    }
},
```

Deixamos o `ordena` sem nenhuma validação. Já nossa `lista` deve ser uma propriedade obrigatória, pois sempre devemos ter uma para exibir nesse componente. Ela também deve ser um `Array` e, por padrão, estar vazia.

Com isso, caso um desenvolvedor externo use seu componente de forma errônea, por exemplo, passando uma `String` no lugar do `Array` da lista, o Vue exibirá alertas e erros no console. Ele informará que o tipo de dado que o atributo `lista` do

componente espera receber está diferente do que lhe foi passado, logo teremos mais segurança e menos bugs.

7.2 SLOT — RECEBENDO UM BLOCO DE CÓDIGO

Anteriormente vimos como fazer para receber em um componente alguns atributos, mas no Vue também temos a possibilidade de receber um código HTML para encorpar nosso componente. Para isso, usamos `slots`, que são declarações básicas que indicam um espaço para receber um código HTML.

Um exemplo muito real para se usar `slot` é um *card* básico, no qual temos de exibir informações distintas para o usuário. Um exemplo é termos um título e um conteúdo em todos os cartões.

A princípio, para se fazer isso, poderíamos *codar* tudo na unha, criando cartão por cartão e trocando seu conteúdo na mão, copiando e colando esse código para fazer mais um cartão. Por exemplo:

```
...
<div class="cartao">
    <h2>Preços</h2>
    <p>Aqui você encontra preços <strong>baixos</strong></p>
</div>
<div class="cartao">
    <h2>Qualidade</h2>
    <p>Temos os produtos com mais <strong>qualidade</strong> no m
ercado</p>
</div>
...
```

Porém, esse código tende a se expandir ao infinito. Ali temos um cartão simples, até legível, mas imagine se tivéssemos um card

do Bootstrap? O código ficaria muito mais complexo, pois teria mais classes do CSS e elementos do HTML.

Podemos resolver isso com as `props` vistas na seção anterior, então, vamos criar um componente chamado `LvCartao.vue`. No seu `script`, vamos apenas declarar as `props` que receberemos, no caso será o `titulo` e o `conteudo`.

```
export default {
    ...
      props: [ 'titulo', 'conteudo' ],
  ...
}
```

Já no seu `template`, vamos colocar o código HTML de apenas um cartão, exibindo as propriedades que serão recebidas. Note que, para exibir a propriedade `conteudo`, estou usando a diretiva `v-html`, pois essa propriedade pode conter tags HTML, e elas devem ser interpretadas como linguagem e não um texto plano.

```
<template>
    <div class="cartao">
        <h2>{{ titulo }}</h2>
        <p v-html="conteudo"></p>
    </div>
</template>
```

Vamos também colocar algo no `style` do componente, já que temos a classe `cartao`: uma borda e um espaçamento.

```
<style>
.cartao { border: 1px solid #000; padding: 10px; }
</style>
```

Agora, na declaração do componente, teríamos de passar nosso `titulo` e `conteudo`, ficando algo assim:

```
<lv-cartao
    titulo="Preço"
    conteudo="Aqui você encontra preços <strong>baixos</strong>">
</lv-cartao>
```

Tivemos uma melhora no código, mas o atributo `conteudo` me incomoda um pouco, pois estamos passando uma `String` que mistura texto com código HTML, e isso não é um trabalho muito limpo. E se tivermos mais conteúdo ou uma `div` com botões dentro do nosso cartão? Essa `String` viraria um verdadeiro inferno, então é para isso que temos o `slot` .

Vamos alterar nosso componente `LvCartao` , retirando a propriedade `conteudo` e, no seu lugar, colocando apenas uma tag chamada `slot` :

```
<h2>{{ titulo }}</h2>
<slot></slot>
```

Essa tag será substituída pelo conteúdo informado dentro da declaração do nosso componente, logo poderemos declará-lo no `App.vue` assim:

```
<lv-cartao titulo="Preço">
   Aqui você encontra preços <strong>baixos</strong>
</lv-cartao>
```

Veja que agora temos um código muito mais legível e sem duplicação. Dentro da tag `lv-cartao` , poderíamos criar botões ou outros elementos HTML sem muita complexidade, pois agora temos um `slot` .

E se tivéssemos mais um `slot` ? Por exemplo, se todos os cartões tivessem um rodapé além do seu conteúdo? Ao tentar declarar outra tag `slot` , note que o Vue exibirá um erro no console dizendo que encontrou a presença duplicada do `slot`

padrão no componente, o que pode causar falha na renderização.

Calma, podemos resolver isso. O Vue permite que criemos `slots` com nomes, assim você mostrará para ele que o conteúdo X deve ser colocado no `slot` padrão, e o conteúdo Y no `slot` `rodape`. Para fazer isso, logo após nossa tag `slot`, colocaremos um novo `slot` chamado `rodape`.

```
<hr>
<slot name="rodape"></slot>
```

Agora temos dois `slots` no componente `LvCartao`: o padrão, que representará o conteúdo e não tem um `name`; e o `rodape`, que representa nosso rodapé, que pode ser qualquer código HTML que será exibido depois de um `hr`.

Dentro da declaração do componente, podemos informar o conteúdo do `rodape`. Basta colocar o atributo `slot` em qualquer elemento. Os elementos sem esse atributo serão colocados dentro do `slot` padrão, que no caso representa o corpo do cartão.

```
<lv-cartao titulo="Preço">
    Aqui você encontra preços <strong>baixos</strong>

    <div slot="rodape">
        Meu conteúdo do rodapé
    </div>
</lv-cartao>
```

Ao executar, o conteúdo dentro dessa `div` será exibido logo após o `hr`, que foi onde declaramos nosso `slot` `rodape`. O resultado será parecido com:

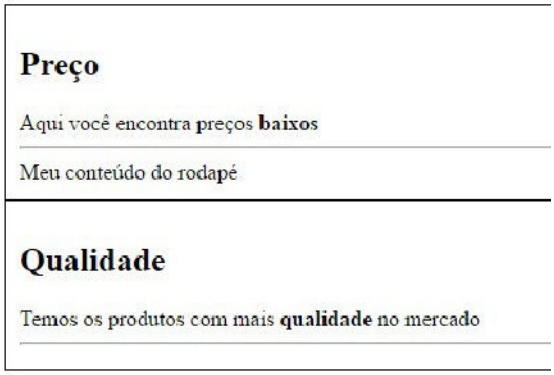

Figura 7.2: Resultado do exemplo de slot

Aqui, usei duas vezes meu componente `LvCartao`, como no exemplo dado no início da seção. A única diferença foi que, no primeiro cartão, também declarei um `slot` rodapé, já no último não há nenhum rodapé.

7.3 MIXINS — ESTENDENDO UM COMPONENTE

Às vezes, podemos ter métodos, variáveis, filtros e outras coisas que são usadas em diversos componentes. É papel do `mixin` fazer com que possamos reusar esses atributos em outros componentes, estabelecendo uma espécie de herança entre eles.

Em um sistema de exemplo, temos um blog, onde os títulos das postagens devem ser sempre exibidos em *uppercase*. Deveríamos usar um filtro para formatar o título, mas em um blog teremos diversos componentes nos quais aparece o título da postagem, por exemplo: na lista de postagens; na busca por postagens; na leitura do post; no campo de comentários sobre o post; no espaço de RSS

do blog; entre outros. Para não precisarmos ficar declarando o filtro em diversos componentes e termos um código duplicado, podemos usar um `mixin` para resolver o problema.

Para isso, vamos criar um arquivo chamado `postagem.js` dentro do diretório `src`. Nele vamos exportar um objeto qualquer com os atributos que queremos compartilhar. No caso, compartilharemos um filtro que transforma o valor filtrado em maiúsculo.

```
export default {
    filters: {
        maiusculo: (valor) => {
            return valor.toUpperCase()
        }
    }
}
```

Vamos criar os componentes `LvLeitor` e `LvComentarios` apenas para exemplificar um blog, no qual um componente representa a leitura de um post e o outro, o bloco de comentários daquele post.

Após criarmos o esqueleto de cada componente, devemos importar o novo `mixin` no `script` de ambos. Para isso, usaremos o `import` do ES2015, e declararemos um atributo do tipo `Array` chamado `mixins`, passando a variável importada para ele:

```
import postagem from './postagem'

export default{
    ...
    mixins: [postagem],
    ...
}
```

No `template` do componente `LvComentarios`, vamos apenas criar um `h4` mostrando um texto com o título do post e filtrando-o com o filtro `maiusculo`:

```
<div>
  <h4>Comentários de {{ 'artigo X' | maiusculo }}</h4>
</div>
```

No componente `LvLeitor`, faremos algo parecido, porém o `h4` deve dar lugar ao `h1`, assim teremos um título maior:

```
<div>
  <h1>Lendo {{ 'artigo X' | maiusculo }}</h1>
  ..........
</div>
```

Por fim, no `App.vue`, devemos importar esses dois componentes para ver o funcionamento. O resultado será algo parecido com:

Figura 7.3: Resultado do exemplo de mixin

Você verá que, mesmo não declarando um filtro com nome `maiusculo` nos componentes `LvLeitor` e `LvComentarios`, o código e o filtro vão funcionar sem erros. Isso se deve ao fato de já importarmos o `mixin` que tem esse filtro por padrão.

Note que aumentamos o reúso do código. Para finalizar, vale lembrar de que esses `mixins` têm o mesmo conceito de herança.

Por exemplo, se declararmos no componente `LvLeitor` um filtro chamado `maiusculo` que transforma o valor em maiúsculo e concatena com a `String` `postagem`, o resultado será o nome do post concatenado com a `String`. Isso porque o Vue priorizará o filtro lido no próprio componente, ignorando o declarado no `mixin`, assim como acontece na herança.

Lendo ARTIGO X postagem

.....

Comentários de ARTIGO X

Figura 7.4: Resultado do exemplo de mixin e herança

Higher-Order Components

Na comunidade do Vue.js, há um novo conceito surgindo, o *Higher-Order Components*, em que um componente pode pegar atributos específicos de outro. Isso funciona como um `mixin`, porém personalizado e mais prático, pois você pode ter um componente A e um B, podendo reaproveitar atributos de um no outro.

O motivo para o surgimento desse conceito é o mesmo que sua vantagem: você pode escolher o que quer importar para o seu componente, caso deseje apenas importar filtros ou métodos, aumentando assim a performance.

A desvantagem que vejo é a organização e o acoplamento. Você pegará atributos de outro componente e não de um `mixin`, logo, se aquele componente depender de outros, seu acoplamento subirá e seu código ficará mais complexo.

Caso tenha interesse em saber mais sobre o assunto, no blog do Vue.js Brasil, Pablo Henrique escreveu um post sobre isso: http://vuejs-brasil.com.br/higher-order-components-em-vue-js-2.

7.4 EMIT — COMUNICAÇÃO ENTRE COMPONENTES

Em alguns momentos na nossa aplicação, quando usamos muitos componentes divididos (como estamos aprendendo),

precisamos comunicar a um componente que outro foi alterado, pois com essa alteração ele pode ou não ter de mudar algo em si mesmo. Para isso, temos o `emit`, no qual podemos fazer com que um componente emita um sinal para outro, podendo até mesmo enviar dados.

Por exemplo, em um blog temos um componente que filtra os posts que recebem mais curtidas. Então, o componente `curtida` deve notificar ao que filtra os posts mais curtidos que uma postagem teve mais um *like*, assim nossa lista de mais curtidos pode mudar em tempo real.

Para vermos isso funcionando, escolhi criar um microchat, em que teremos um componente que representa um usuário. Quando esse usuário submeter uma nova mensagem, o componente a enviará para outro, e ela será exibida na tela principal do projeto.

Vamos iniciar criando um componente chamado `LvUsuario.vue` na pasta `src`, com seu esqueleto pronto. Editaremos seu `script` colocando no `data` a variável `mensagem` que armazenará o texto digitado pelo usuário. Também vamos receber uma propriedade chamada `nome`, que nada mais é do que o nome do usuário daquele chat.

```
export default{
    name: 'lv-usuario',
    data() { return { mensagem: '' } },
    props: ['nome'],
}
```

Por fim, adicionaremos o método `envia`, que enviará uma notificação chamada `novaMsg` com a mensagem que o usuário acabou de digitar. Após isso, ele limpará o campo de mensagem para que a próxima seja digitada.

```
...
methods: {
  enviar() {
    this.$emit('novaMsg', this.nome + ': ' + this.mensagem)
    this.mensagem = ''
  }
}
...
```

No template do componente, devemos informar o nome do usuário que teremos no chat com a tag `small` . Vamos também colocar um `input` que representa nossa variável `mensagem` e um botão para que ele possa executar o método `enviar` para mandá-la aos outros componentes.

```
<div>
  <small>{{ nome }}:</small>
  <input type="text" v-model="mensagem">
  <button @click="enviar">Enviar</button>
</div>
```

Temos o componente pronto, e ele nada mais é do que a interface de um usuário do chat. Agora devemos importá-lo para nosso `App.vue` . Ao declará-lo, passamos a propriedade `nome` e também uma escuta para a notificação `novaMsg` , com a diretiva `v-on` ou simplesmente o `@` , assim o Vue reconhecerá que, sempre que o componente enviar a notificação `novaMsg` , o método `escrever` do `App` será executado.

```
<template>
  <div id="app">
    <lv-usuario nome="Filomena" @novaMsg="escrever"></lv-usuario>
  </div>
</template>

<script>
import LvUsuario from './LvUsuario.vue'

export default {
```

```
  name: 'app',
  components: { LvUsuario },
}
</script>
```

Ainda não temos o método `escrever`, então vamos declará-lo. Ele deve receber um parâmetro, que será a mensagem enviada pela notificação `novaMsg` emitida pelo componente `LvUsuario`. Vamos pegar essa mensagem e concatená-la com uma variável chamada `chat` que armazenará todas as mensagens enviadas pelos usuários declarados.

```
...
data () { return { chat: '' } },
methods: {
    escrever(mensagem) {
        this.chat = mensagem + '<hr>' + this.chat
    }
}
...
```

Note que declarei no `data` a variável `chat`. Agora devemos exibi-la na tela e, para isso, no final do nosso `template`, coloque:

```
<br>
<div v-html="chat"></div>
```

Assim, todas as mensagens enviadas pelo componente `LvUsuario` serão exibidas separadas por uma linha `hr`. No navegador, digite alguma mensagem no `input` e clique em enviar. O resultado será algo como:

Filomena: Mais uma msg Enviar

Filomena: Digitando outra msg

Filomena: Estou digitando uma mensagem

Figura 7.5: Enviando mensagem para outro componente

Note que, ao enviar uma mensagem, ela é transmitida pelo `$emit` em uma notificação chamada `novaMsg`. É lida pelo método `escrever` do componente `App` e, por sua vez, concatenada com as recebidas anteriormente e exibida na tela.

Caso você declare mais uma vez o componente `LvUsuario` no `App`, teremos a simulação de dois usuários logados no nosso microchat. Faça o teste acrescentando no seu `template`:

```
<lv-usuario nome="Amarildo" @novaMsg="escrever"></lv-usuario>
```

Serão exibidos na tela dois campos para escrevermos. A primeira mensagem representará a Filomena, já a segunda, o Amarildo. Isso muda nosso resultado para:

Filomena: Enviar
Amarildo: Estou ótimo! Enviar

Filomena: Olá, tudo bem?

Amarildo: Oi

Figura 7.6: Chat com dois componentes lv-usuario

Concluímos que o `emit` tem por objetivo lançar notificações

de alterações de estado do componente, assim, a partir do componente pai, podemos captar mudanças que ocorreram no filho.

7.5 TESTE SEU AVANÇO

Vamos fazer alguns exercícios sobre o que aprendemos?

Memorize a teoria

1. Por que devemos reutilizar nossos componentes? O que isso nos traz de ganho?

   Responda as questões a seguir, informando se você utilizaria: `props`, `slots`, `mixins` ou `emit`, para criar o item pedido. Além disso, justifique sua escolha.

2. Em um componente chamado `slide`, devemos receber um conteúdo HTML com a imagem e o texto que o slide vai apresentar.

3. Devemos informar ao componente `Resultado` que o valor total da conta presente no componente `Calcula` se alterou.

4. Temos de receber o nome do título do componente `Postagem`. Além disso, também receberemos o nome do seu autor.

5. O sistema já tem um componente chamado `Pessoa`, com dados, filtros etc., e queremos criar o componente `Gerente`. Este vai possuir praticamente tudo o que o outro componente apresenta, logo, devemos reaproveitar o código.

Na prática

1. Crie um componente que represente um painel de confirmação. Ele receberá um título e um conteúdo personalizado, e esse conteúdo pode ser um código HTML.

2. No componente, crie um botão que emita uma notificação quando for clicado.

3. No `App.vue`, importe o componente criado e faça com que, quando seu botão for pressionado, apareça uma tag `p`, informando que o painel de confirmação foi fechado.

4. Faça com que tanto o título do painel de confirmação quanto a mensagem informando que o fechamos no `App.vue` sejam exibidos de trás para a frente. Para isso, use essa expressão do JS:

```
valor.split('').reverse().join('')
```

Capítulo 8

CADA UM SEGUE SEU CAMINHO, COM ROTAS!

Você deve ter notado que, até agora, usamos sempre a mesma tela do site e não trabalhamos com um exemplo que trocasse a página atual que estamos acessando. O Vue, por si só, não tem essa capacidade, mas o próprio Evan You nos oferece um plugin chamado **Vue-router**. Com ele, podemos criar aplicações com mais de uma página, dividindo nosso site, assim como a maioria dos sites atuais.

O conceito desse plugin, como o seu nome já diz, é criar rotas. Uma rota é um link do seu site. Ao entrar em qualquer site, como https://vuejs.org, seremos direcionados para a primeira rota, que é a barra (`/`). Note que, se clicarmos no menu `Guide` do site, o link será alterado para: https://vuejs.org/v2/guide/. Logo, agora estamos na rota `/v2/guide` , e assim segue-se para todos os links do site.

Resumidamente, uma rota é cada caminho/URL que seu sistema pode ter. Ter um sistema de rotas em um projeto é essencial para manipular todos os links do seu site com uma agilidade incrível. Como já foi dito, no Vue precisamos de algo externo para adquirir essa funcionalidade, pois o Vue é apenas um

gerenciador de componentes, e esse algo externo é o Vue-router.

O Vue-router é um componente/plugin (como preferir) que faz o gerenciamento das rotas de um sistema. Com ele, podemos definir quais links nosso site terá e o que será mostrado na tela do usuário quando aquele link for acessado.

Podemos instalá-lo de duas formas. A primeira é usando um arquivo `.js` fornecido no seu próprio site (https://router.vuejs.org), bastando apenas baixá-lo e importá-lo para o `index.html`. Mas como vimos nos primeiros capítulos, gerenciar manualmente as dependências do projeto pode se tornar um labirinto quando o sistema crescer, por isso o Vue-router nos disponibiliza a instalação via NPM. Esta é a mais indicada, pois teremos um gerenciador de dependências automatizado.

Vamos seguir o método mais indicado, com um projeto novo e um terminal aberto em sua pasta. Digite o comando:

```
npm install vue-router --save
```

8.1 CRIANDO UM CENÁRIO DE EXEMPLO

Com a extensão já instalada, antes de começar a configurá-la, montaremos um pequeno cenário para servir de demonstração de uso do Vue-router. Nele vamos ter um sistema com um menu superior, no qual podemos ir para a **página inicial** ou **lista de usuários** e, nesta última, teremos um botão para registrar um **novo usuário**.

Quando registrarmos um usuário, enviaremos uma **notificação** dizendo que ele foi registrado. Lógico que de fato o sistema não vai cadastrar um usuário, pois ainda não vimos como

fazer isso, vamos apenas simular a navegação de páginas no Vue.

Analisando a descrição dada anteriormente, note que temos quatro objetos, ou seja, quatro componentes, aqueles que estão em **negrito**. Criaremos esses componentes no diretório `src`, iniciando aleatoriamente pelo componente `Inicial`, que representará a página inicial do sistema. Por enquanto, apenas colocamos um nome para ele e, no seu `template`, um `h1` dizendo que estamos na *Home*.

```
<template>
    <div>
        <h1>Inicial</h1>
    </div>
</template>

<script>
export default{
    name: 'lv-inicial',
}
</script>
```

Vamos criar os componentes `Usuarios`, `NovoUsuario` e `Notificacao`, todos com o mesmo código do anterior, apenas alterando o seu `name` e o texto no `h1`. Assim podemos distingui-los uns dos outros.

8.2 CONFIGURANDO O VUE-ROUTER

Agora chegou a hora de configurar o Vue para que ele reconheça o novo plugin. Para isso, no início do arquivo `src/main.js`, vamos importar o `VueRouter`.

```
import Vue from 'vue'
import VueRouter from 'vue-router'
```

Após isso, temos de falar para o Vue: *"olha aqui amigão, vamos usar o Vue-router aí!"*, e é exatamente para fazer isso que o Vue disponibiliza um método chamado `use`. Então, logo abaixo das importações, utilizaremos o `use` passando como parâmetro o `VueRouter`, assim dizemos que vamos empregá-lo no projeto.

```
Vue.use( VueRouter )
```

O próximo e último passo, antes de criar as rotas, é criar uma instância do `VueRouter`. Será nela que vamos configurar as nossas rotas mais tarde. Por enquanto, crie uma constante chamada `router` que recebe a instância citada, e esta recebe um objeto vazio. Por fim, passamos essa constante no componente root do `Vue`, assim ele saberá quais rotas existirão no sistema.

```
const router = new VueRouter({})

new Vue({
    el: '#app',
    router,
    render: h => h(App)
})
```

8.3 CRIANDO ROTAS

Chegamos enfim à parte de começar a criar novas rotas. Para isso, logo após do `import` do Vue e Vue-router, devemos importar nossos componentes `Inicial` e `Usuario`:

```
import Inicial from './Inicial.vue'
import Usuarios from './Usuarios.vue'
```

Com eles já importados, podemos vinculá-los às novas rotas. Para isso, temos de editar o objeto vazio da instância `VueRouter` que declaramos antes. Nela vamos ter um atributo chamado

`routes` que representa um `Array` com todas as rotas do sistema. Deixe-o assim:

```
const router = new VueRouter({
    routes: [
        {path: '/',             component: Inicial},
        {path: '/usuarios',     component: Usuarios},
    ],
})
```

Note que, dentro do `Array` de rotas, cada rota é um objeto. Inicialmente, temos dois atributos: o `path` indica qual o caminho do site que essa rota vai representar, já o `component` corresponde ao componente que será montado na tela quando o link indicado no `path` for acessado.

Para testar, temos de realizar uma edição no `App.vue`, pois, como componente principal, ele deve informar quais outros componentes serão exibidos na tela. O Vue-router disponibiliza uma tag chamada `router-view`, e seu funcionamento é apresentar o componente que representa a rota atual. Ou seja, se a rota acessada for `/usuarios`, o `router-view` vai chamar nosso componente `Usuario` e exibi-lo na tela. Vamos deixar nosso `App.vue` assim:

```
<template>
  <div id="app">
    <router-view></router-view>
  </div>
</template>

<script>
export default {
  name: 'app',
}
</script>
```

Vá ao navegador com a live do `npm run dev` ativa. Nele você

verá que foi exibido o conteúdo do componente `Inicial.vue`. Isso funciona, pois o colocamos para representar nossa rota `/`. Caso você mude o link para http://localhost:8080/#/usuarios, o conteúdo do componente `Usuarios` é o que será exibido. Logo, já temos uma aplicação com mais de uma página.

Para exemplificar mais o funcionamento que acabamos de ver, abra o Vue Devtools. Note que, inicialmente, dentro do componente `App`, temos nosso `Inicial` e que, ao lado dele, aparece a tag `router-view`, pois esse componente está sendo exibido por ela.

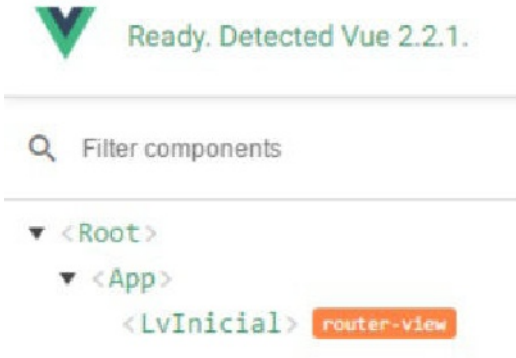

Figura 8.1: A tag router-view apresenta o componente Inicial

Quando acessarmos a rota `/usuarios`, veja que o componente representado pela tag `router-view` agora é o nosso `Usuarios`, e que, além da tag, temos a nossa rota atual. Basicamente a tag `router-view` foi trocada pelo conteúdo da rota atual.

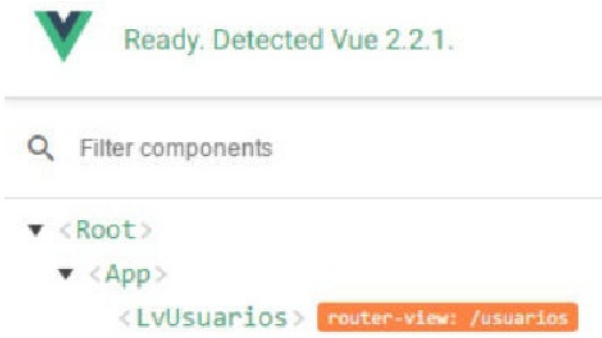

Figura 8.2: A tag router-view apresenta o componente Usuarios

8.4 TROCANDO DE PÁGINA

Veremos agora que o Vue-router nos dá a possibilidade de trocar a rota atual do site via programação. Por exemplo, ao clicar em um link, podemos ir para uma nova página ou voltar para a anterior.

Com as duas rotas do exemplo anterior já funcionando, vamos criar nosso menu superior no `App.vue` . Para isso, o Vue-router disponibiliza uma tag chamada `router-link` , que cria um link `a` na tela. Quando clicado, ele redirecionará para a rota informada no atributo `to` .

Criamos dois links, um vai para a página inicial e outro, para a página de usuários. Coloque o seguinte código antes do `router-view` :

```
<router-link to="/">Inicial</router-link>
<router-link to="/usuarios">Usuarios</router-link>
```

Ao testar no navegador, verá que, clicando em `Usuarios` , somos redirecionados para a rota `/usuarios` e, ao clicar em

`Inicial`, vamos para a rota `/`, como exemplificado na figura a seguir:

Figura 8.3: Navegando entre as rotas '/' e '/usuarios'

Note que o menu nunca sai da nossa tela, porque o colocamos dentro do componente `App`. Como nosso `router-view` está lá, logo todo o conteúdo adicional do componente será concatenado com a página que representa a rota atual. Isso é ótimo para criar layouts sem dificuldade.

Temos diversas formas para redirecionar para outra página do sistema. Veremos isso nas próximas seções.

8.5 HASH NÃO É URL AMIGÁVEL

Até então, você deve ter percebido que sempre temos uma *hashtag* no nosso link. A estrutura padrão de URL do Vue-router é: domínio + `/` + hash + rota.

No Vue-router, temos essa *hash* com o objetivo de trocar de página sem a necessidade de recarregá-la. Note que, diferentemente de outros sites, se clicarmos no menu `Usuarios`, a página será trocada, mas não recarregada, pois o navegador interpreta-a como uma âncora. Esse é um aspecto sensacional por dar uma interatividade muito grande.

Essa hash pode se tornar um problema, principalmente para a prática do SEO — conjunto de técnicas para aparecer melhor

ranqueado em buscadores —, pois normalmente os buscadores ignoram o que está depois da hash. Assim, eles lerão sempre apenas a página inicial do sistema, deixando as outras fora de seus resultados. Além disso, não é usual, pois o usuário não está acostumado com hashs em URLs.

Solucionar isso é bastante simples, graças à API *history* (definição a seguir) do HTML5. No Vue-router, podemos habilitar o modo `history` e, a partir daí, ele utilizará essa API da linguagem. Como consequência, essa hash sumirá, continuando a manter a troca de página sem recarregamento.

DEFINA 'API HISTORY'

É uma API fornecida pelo HTML5. Ela permite que, via JavaScript, manipulemos diretamente o histórico do navegador, adicionando links e trocando de URL sem o recarregamento (onde a mágica acontece).

Para trocar o modo de URL, na instância do `VueRouter` no `main.js`, vamos acrescentar mais um atributo. Ao lado de `routes`, esse atributo se chama `mode`, e nele temos de informar que vamos usar o modo `history`. Assim o Vue-router passa a usar a API do HTML5.

```
const router = new VueRouter({
    mode: 'history',
    routes: [
        ...
    ],
})
```

Vá ao navegador e note que a hash sumiu da nossa URL. Agora sim temos uma URL de fato amigável!

Mas para que isso funcione em ambiente de produção, temos de adicionar um arquivo de configuração no servidor. Posteriormente, neste livro (capítulo *Alguns recursos escondidos*), veremos como publicar nosso site na internet. Lá você encontrará o tal arquivo de configuração, já que agora ele não é importante para nós.

8.6 CRIANDO SUB-ROTAS

Para uma melhor organização, normalmente vemos níveis diferentes de rotas, ou seja, rotas dentro de rotas, ou sub-rotas. Funciona como vimos no site do Vue, quando acessamos a rota `/v2/guide`, em que `v2` é nossa rota e `guide` nossa sub-rota.

Usamos essas sub-rotas para manter a URL o mais legível possível. Como nosso sistema de exemplo tem a opção de cadastrar um novo usuário, a rota para essa página seria `/usuarios/novo`, pois `usuarios` representa um objeto do nosso sistema, logo todas as ações referentes a esse objeto devem ser suas sub-rotas.

Para começar a criar a sub-rota `/usuarios/novo`, vamos criar um `router-link` dentro do componente `Usuarios`. Ele funcionará como um botão: ao clicar nele, será exibido o componente de registro de um novo usuário. Para exibir esse componente, usaremos outra tag `router-view`.

```
...
<h1>Usuarios</h1>
<router-link to="/usuarios/novo">Novo usuario</router-link>
<router-view></router-view>
```

```
...
```

Quando temos a tag `router-view` dentro de um componente secundário, como em `Usuarios`, o Vue-router automaticamente ligará essa tag a uma sub-rota, buscando a atual e exibindo seu conteúdo dentro dessa tag. Caso não encontre uma sub-rota, ele exibirá apenas o componente secundário, no caso, o `Usuarios`.

O próximo passo é declarar essa sub-rota. Para isso, no `main.js`, teremos de importar nosso componente `NovoUsuario`. No objeto da rota `/usuarios`, vamos colocar um atributo chamado `children`; ele indica um `Array` de sub-rotas daquela rota.

```
...
import NovoUsuario from './NovoUsuario.vue'
...
const router = new VueRouter({
    routes: [
        ...
        {
            path: '/usuarios',
            component: Usuarios,
            children: [
                {path: 'novo', component: NovoUsuario},
            ]
        },
    ],
    ...
})
...
```

Dentro desse `Array`, há objetos de rotas, no caso, apenas a rota `novo`. Como ela é uma sub-rota, herdará também o `path` da rota pai, logo seu verdadeiro `path` será `/usuarios/novo`.

Quando executarmos isso no browser, veremos tudo funcionando devidamente, com o menu superior que criamos

antes. Agora na tela de usuários, temos um link para registrar um novo e, quando clicarmos nele, o componente `NovoUsuario` será desenhado na tela, como na figura:

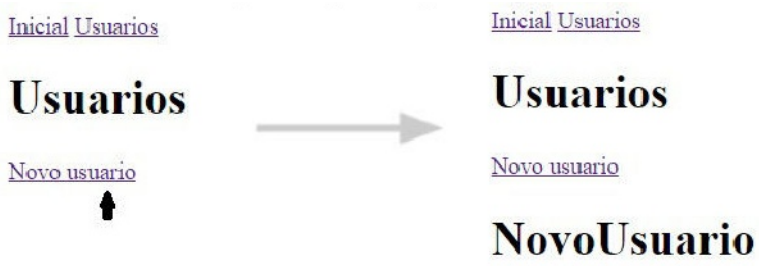

Figura 8.4: Ao clicar em novo usuário, a sub-rota aparece

Abrindo o Devtools, podemos ver novamente isso funcionando. Antes de clicar em `Novo usuario`, o componente `Usuarios` tem apenas um filho, o `router-link` e, ao clicar nesse link, veremos que foi adicionado o filho `NovoUsuario`. A ele está vinculado o `router-view: /usuarios/novo`, pois ele representa uma sub-rota.

Figura 8.5: Sub-rota aparecendo no Devtools

8.7 ENVIANDO PARÂMETROS

O último aspecto básico desse plugin que precisamos ver para criar qualquer aplicação é como enviar parâmetros dinâmicos em rotas. Normalmente usamos esses parâmetros para criar páginas dinâmicas, como perfis, pois o parâmetro indicaria o nome de

usuário ou identificador. Assim, poderíamos pegar os dados daquele usuário e exibi-los, criando o seu perfil.

Como ainda não temos uma base de dados, vamos fazer um exemplo menor. No componente `NovoUsuario` , crie dois botões: um executando o método `falhou` e o outro, o método `beleza` . Em cada método, vamos redirecionar a rota para `/usuarios/algumacoisa` , onde `algumacoisa` será nosso parâmetro. Este indicará se houve erro ou sucesso ao cadastrar um usuário; no sucesso, enviaremos o nome do usuário como parâmetro.

```
...
<h1>NovoUsuario</h1>
<button @click="falhou">Falha</button>
<button @click="beleza">Sucesso</button>
...
methods: {
  falhou() {
      this.$router.replace('/usuarios/erro')
  },
  beleza() {
      this.$router.replace('/usuarios/leonardo')
  }
}
...
```

Perceba que usamos outro meio para trocar de página. Antes apenas tínhamos visto o `router-link` , agora vimos o `replace` , que apenas troca a rota do site via JavaScript.

O próximo passo é registrar essa rota. Para isso, no `main.js` , vamos importar o componente `Notificacao` . Criamos uma sub-rota para `/usuarios` , e seu `path` será apenas o parâmetro `msg` . Temos de colocar dois pontos (`:`) antes do nome do parâmetro para que o plugin saiba que aquilo é uma variável.

```
...
import Notificacao from './Notificacao.vue'
...
routes: [
  {path: '/',             component: Inicial},
  {
    path: '/usuarios',
    component: Usuarios,
    children: [
      {path: ':msg', component: Notificacao, props: true},
      {path: 'novo', component: NovoUsuario},
    ]
  },
],
...
```

Note que usamos um novo atributo na rota, o `props`. Ele indica que o parâmetro da rota se tornará uma propriedade do componente `Notificacao`, assim podemos pegar seu valor sem nenhum trabalho.

Vamos pegar esse valor agora. No `Notificacao`, declaramos a propriedade `msg` e, com a diretiva `v-if`, exibiremos uma mensagem de erro ou saudação.

```
...
<h1>Notificação</h1>
<p v-if="msg == 'erro'">Erro ao cadastrar usuário</p>
<p v-if="msg != 'erro'">Olá, {{ msg }}</p>
...
props: ['msg'],
...
```

Ao tentar testar no navegador, ocorre um erro lógico. Quando clicarmos em `Novo usuario`, não iremos para o componente `NovoUsuario`, mas sim para `Notificacao`. Isso se deve à ordem na qual declaramos as rotas.

Ao declarar a rota que recebe um parâmetro antes da rota

`novo`, o Vue-router interpretará que a `String` `novo` da URL se trata de um parâmetro `msg`, e não de outra sub-rota. Para solucionar isso, devemos trocar a ordem das declarações para:

```
{path: 'novo', component: NovoUsuario},
{path: ':msg', component: Notificacao, props: true},
```

Assim, priorizamos a rota `/usuarios/novo`, identificando que a `String` `novo` é uma sub-rota, e não um parâmetro `msg`.

Agora sim estará tudo funcionando. Ao clicar no botão `Falha`, o componente `Notificacao` é exibido com a mensagem de erro; já se clicarmos em `Sucesso`, teremos a mensagem de saudação com o nome do usuário que supostamente teria sido criado, como na figura:

Figura 8.6: Casos de falha e sucesso, com parâmetro enviado

Com isso, temos os parâmetros funcionando, passando-os entre rotas, e já podemos criar páginas dinâmicas no Vue.

Passamos pelo básico desse plugin, mas ele é um pouco mais amplo do que isso. Por exemplo, podemos ter observadores de rota para fazer um tipo de controle de acesso, dizendo que o usuário X não pode acessar a rota Y. Como esse é um assunto avançado e nem todo sistema tem controle de usuário, deixo-o fora do livro. Para aprender mais sobre o Vue-router, consulte:

https://router.vuejs.org.

8.8 TESTE SEU AVANÇO

Chegamos mais uma vez ao momento em que você testa o que aprendeu, tentando fazer coisas diferentes e concretizar o conhecimento adquirido.

Memorize a teoria

1. O que são rotas? Quando devemos usá-las?
2. Quais são as desvantagens de se usar a hash em URLs de um sistema?
3. Quando devemos usar sub-rotas?

Na prática

Para realizar esses exercícios, crie um novo projeto, e instale e configure o Vue-router.

1. Ao acessar a página inicial do sistema, faça com que o componente `Home` seja chamado e exiba um título com o texto `Bem-vindo ao blog` .
2. Faça com que a rota `/postagens` exiba um componente chamado `Lista` . Ele deve ter uma lista de links de rotas indo para as rotas `/postagens/id` , em que `id` é um número que identifica um post.
3. Faça um menu superior, trocando entre `/` e `/postagens` .

4. Ao clicar em um dos links do componente `Lista` , vamos para a rota `/postagens/id` . Faça com que ela represente um componente chamado `Postagem` , e neste exiba o título *Postagem* + *`id`* . Nele também teremos um link de rota indo para `/postagens/comentarios` .

5. Faça com que a rota `/postagem/comentarios` mostre um componente chamado `Comentarios` com uma lista qualquer de comentários. Fique atento à ordem da declaração de rotas.

   No final, teremos uma página inicial com menu superior indo para uma listagem de postagens. Ao selecionar uma postagem da lista, na mesma página será incluído um componente responsável por mostrar o título da postagem e um link para seus comentários e, ao clicar para ver os comentários, veremos uma lista aleatória.

Capítulo 9

GERENCIAMENTO DE ESTADO COM VUEX

Vamos mais uma vez falar de um plugin externo do Vue, também criado por Evan You. O *Vuex* é um gerenciador global de dados, no qual você pode ter uma loja de dados que todos os componentes poderão acessar e manipular. Seu objetivo é facilitar a troca de dados, mantendo a organização, a segurança e uma única fonte.

Centralizando a fonte de dados da aplicação, você garantirá que não haja dados duplicados e nem distintos quando deveriam ser iguais. Além disso, no Vuex, temos o gerenciamento de estado, em que você pode ver com facilidade a alteração da interface guiada pelo estado de seus dados. Isto é, quando eles mudam, a interface também muda, e o Vuex tem a capacidade de deixar todos os estados armazenados para futura consulta, como para debugar ou ver qual é o valor de uma variável de estado antes e depois de ser alterada.

Neste capítulo, vamos conceber um pequeno projeto de lista de tarefas, em que será possível criá-las e apagá-las, além de pesquisar uma tarefa da lista. Note que, como elemento principal, temos uma lista de tarefas. Ela deve ser manipulada em diversos componentes,

como: lista, pesquisa, botão de apagar e criar. Logo, devemos usar o Vuex para armazenar a lista, assim não corremos o risco de manipulá-la de forma errônea e afetar o funcionamento da aplicação.

Antes de tudo, vamos instalar o Vuex, usando o comando:

```
npm install vuex --save
```

Antes de codificar, temos de dizer que o Vuex é dividido em cinco partes. Temos o `state`, usado para a definição de dados que a loja possui; os `getters`, que são os transmissores desses dados, utilizados para pegar dados da loja; as `mutations`, que são os manipuladores de dados; as `actions`, cuja única função é submeter mutações; e por fim, os `modules`, grupos de tudo o que vimos anteriormente, assim priorizamos a organização. A seguir, passaremos por cada parte do Vuex.

9.1 STORE — CRIANDO NOSSA LOJA DE DADOS

Vamos iniciar criando nossa loja, ou `store`, onde nossos dados compartilhados ficarão. Para melhor organização, o Vuex indica a criação de muitos arquivos, mas vamos por partes. Para começar, no diretório `src`, criaremos uma pasta `loja` e, dentro dela, um arquivo chamado `raiz.js`.

Nesse arquivo, faremos a elaboração e a exportação da nossa loja, mas antes de tudo devemos importar o Vue e o Vuex, registrar o plugin no Vue com o método `use` e, só então, criar nossa `store`. Caso não siga essa ordem, teremos um erro, pois devemos importar o Vuex para o Vue antes de tudo.

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
})
```

Note que exportamos uma instância de `Store` e nela passamos um objeto vazio. Ali teremos nossos dados mais tarde, pois esse código já é o bastante para criação de uma loja vazia. O próximo passo é informarmos para nossa aplicação atual que usaremos essa loja. Para isso, no `main.js`, vamos importar nossa loja e enviá-la como o atributo `store` para a instância raiz do Vue.

```
import loja from './loja/raiz'

new Vue({
    store: loja,
    ...
})
```

Com isso, já temos uma loja pronta para ser usada no nosso site. A seguir, veremos como começar a criar novos dados compartilhados.

9.2 STATES — DECLARANDO DADOS

Para ter dados dentro da nossa loja, precisamos primeiramente criar algumas variáveis para guardar esses dados. Esse é o único papel do `state`.

Mantendo a organização e o padrão do Vuex, dentro do diretório `loja`, vamos criar o arquivo `estado.js`. Nele exportaremos um objeto JavaScript, e já vamos declarar nosso

`Array` de `tarefas` para iniciar o exemplo do capítulo.

```
export default {
    tarefas: [],
}
```

Após exportar, vem a parte de importar. Precisamos dizer para nossa `store` que vamos usar esses dados de estado e, para isso, no arquivo `raiz.js`, temos de importar o objeto criado anteriormente e passado como o atributo `state` da instância da loja:

```
import estado from './estado'

export default new Vuex.Store({
    state: estado,
})
```

Ao iniciar nossa aplicação com o `npm run dev`, note que nada mudou na interface do site. Mas se abrirmos o Vue Devtools e irmos à aba chamada `Vuex`, podemos ver no canto direito a nossa lista de tarefas, que por enquanto está vazia, como mostra a figura a seguir.

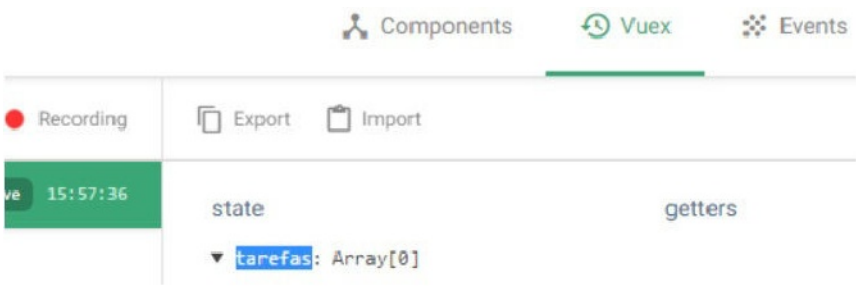

Figura 9.1: Variável de estado está criada

A loja do sistema pode ser acessada em qualquer componente pela variável `$store`. Você pode fazer um teste criando outra

variável de estado, por exemplo, `nome` com valor `VueJS`, e no componente `App` abri-la usando `{{ $store.state.nome }}`. O resultado será `VueJS`.

Não é indicado alterar o valor dessas variáveis diretamente pelo `state`. Isso seria um erro, dado que o Vuex não registra essa alteração na loja. O resultado será a variável alterada na `store`, mas sem deixar um rastro, logo não teremos o funcionamento correto.

Para testar isso, aproveitando a variável `nome` criada anteriormente, altere seu valor ao montar o componente, usando o método `mounted`:

```
mounted() {
  setTimeout(() => {
    this.$store.state.nome = 'Novo nome'
  }, 1500)
},
```

Figura 9.2: Variável de estado não é alterada

Ao executar isso, note que depois de um segundo e meio, o texto na tela é de fato alterado para `Novo nome`. Porém, se olharmos no Vue Devtools, o valor da variável `nome` do `state` não foi modificado. Você pode fazer muita confusão com isso, pois os outros componentes não serão informados sobre o novo valor, por isso veremos o jeito certo de alterar dados, na próxima seção.

9.3 MUTATIONS — ALTERANDO DADOS

Os dados presentes na `store` do sistema provavelmente serão alterados em algum momento. Para fazer isso, temos de usar as `mutations`. É com elas que temos a possibilidade de alterar corretamente um valor da loja.

Vamos adicionar tarefas a nossa lista. Temos de criar um arquivo para armazenar nossas `mutations`, então, dentro de `loja`, criaremos o arquivo `mutacoes.js`. Nele podemos declarar métodos que têm a única responsabilidade de alterar dados do `state` e, por convenção, o nome das mutações deve estar sempre em maiúsculas.

No arquivo `mutacoes.js`, vamos exportar um objeto com o método `ADD_TAREFA`. Todo método de mutação deve receber dois parâmetros: o primeiro se refere ao `state` e o segundo, ao valor que será armazenado nele. Nesse método, vamos apenas pegar o `valor` passado e adicionar no final da nossa lista de tarefas.

```
export default {
    ADD_TAREFA: (estado, valor) => {
        estado.tarefas.push(valor)
    },
}
```

Precisamos registrar essas mutações na nossa loja. Para isso, no arquivo `loja/raiz.js`, temos de importar o objeto criado anteriormente e passá-lo para a instância da `Store` como um atributo chamado `mutations`.

```
...
import mutacoes from './mutacoes'

export default new Vuex.Store({
    ...
```

```
    mutations: mutacoes,
}
```

Com isso, esse método estará disponível para todos os componentes do sistema. Testaremos criando um componente chamado `AddTarefa.vue` e, no seu `template`, vamos ter um `input` que representa a variável `tarefa` e um botão que executa o método `adicionar`.

```
<input type="text" v-model="tarefa">
<button @click="adicionar">Adicionar</button>
```

No `script`, adicionaremos a variável `tarefa` no `data` e criaremos o método `adicionar`. Nele, temos de verificar se o valor está preenchido, e então executar uma mutação para adicionar a tarefa à nossa lista. Na variável `$store`, temos um método chamado `commit` que se responsabiliza por executar mutações da loja, e precisamos usá-lo para executar o `ADD_TAREFA` enviando a variável `tarefa` como valor.

```
export default{
    name: 'lv-addtarefa',
    data() { return { tarefa: '' } },
    methods: {
        adicionar() {
            if(this.tarefa != '')
                this.$store.commit('ADD_TAREFA', this.tarefa)
            this.tarefa = ''
        }
    },
}
```

Feito isso, temos de importar e registrar o componente no nosso `App`, para que ele possa ser exibido na interface do usuário:

```
import LvAddtarefa from './AddTarefa.vue'

export default {
    name: 'app',
```

```
    components: {LvAddtarefa},
    ...
}
```

Lembre-se de colocá-lo no `template` com a tag correspondente ao seu registro anterior:

```
<lv-addtarefa></lv-addtarefa>
```

Com isso, ao testar no navegador, não veremos nada demais. Mas caso abra o Vue Devtools, note que, ao clicar em `Adicionar`, a lista de tarefas no nosso `state` será alterada, passando a ter a última tarefa adicionada, conforme a figura a seguir.

Figura 9.3: Adicionando tarefas à lista da loja

Note também que, na lista à esquerda do Devtools, novos itens foram adicionados. Cada item dessa lista representa um estado do sistema e, no último, temos uma tag nomeada `active`. Ela mostra qual é o estado atual do sistema.

Você pode selecionar qualquer outro item dessa lista também. Veja que, ao fazer isso, o valor da nossa lista de tarefas vai se alterar e apresentará o valor exato do estado em que foi selecionado.

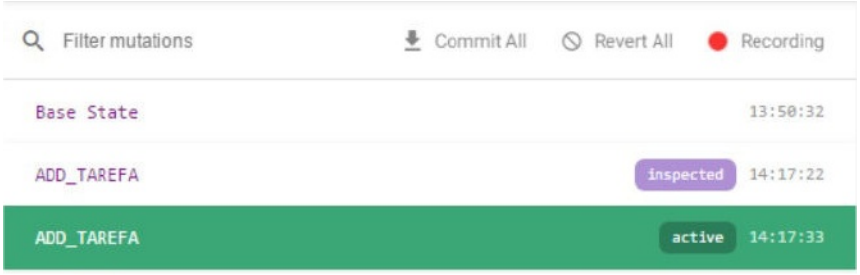

Figura 9.4: Lista de estados do sistema

Veja na figura anterior que, além dos estados, podemos ter um poderoso debugger, no qual vemos quais mutações foram executadas, além de reverter algumas e controlar se a loja está ou não lendo mutações.

Para encerrar, criaremos mais uma mutação, que será responsável por apagar uma tarefa da nossa lista. Voltando ao arquivo `mutacoes.js` , abaixo do método `ADD_TAREFA` , crie um método com o nome `DEL_TAREFA` . Ele apenas consultará qual a `posicao` da tarefa que foi passada por parâmetro na nossa lista do `state` , empregando o método `indexOf` , e então vai apagá-la, usando o `splice` .

```
DEL_TAREFA: (estado, valor) => {
  let posicao = estado.tarefas.indexOf(valor)

  if(posicao > -1)
    estado.tarefas.splice(posicao, 1)
},
```

Executaremos essa mutação na próxima seção, pois precisamos da lista de tarefas sendo exibida na tela para podermos apagar uma tarefa. Por enquanto, deixaremos a ação de exclusão pronta, apenas criando as mutações e um componente chamado `DelTarefa` . No seu `template` , temos um botão que executa o

método `apagar` quando clicado.

```
<button @click="apagar()">D</button>
```

No seu `script`, vamos receber uma propriedade chamada `tarefa`, que nos dirá qual devemos apagar. Além disso, teremos o método `apagar`, que será executado quando o botão for clicado. Ele apenas fará um `commit` na mutação criada anteriormente, passando como parâmetro a `tarefa` recebida na propriedade.

```
export default{
    name: 'lv-deltarefa',
    props: {
        tarefa: {
            required: true,
            type: String
        }
    },
    methods: {
        apagar() {
            this.$store.commit('DEL_TAREFA', this.tarefa)
        }
    },
}
```

Assim já temos um componente que representa um botão e, ao clicar nele, vamos apagar uma tarefa da lista — o que faremos na seção seguinte, pois nela aprenderemos como pegar dados corretamente da `store`.

9.4 GETTERS — PEGANDO DADOS

Até então, sabemos como declarar e alterar dados da loja, agora veremos como pegar esses dados e exibi-los na interface. Para isso, assim como em qualquer linguagem de programação, temos os métodos `getters`.

Como exemplo, vamos primeiramente exibir nossa lista de tarefas, mas vamos reordená-la, fazendo com que as recém-adicionadas fiquem em primeiro lugar e as antigas mais abaixo. Para isso, precisaremos criar nossos `getters`. Na pasta `loja`, crie um arquivo chamado `getters.js`.

No arquivo de getters, vamos exportar um objeto com um método chamado `listaTarefas`, que vai tirar uma cópia da nossa lista com o método `slice` do JavaScript, e então retorná-la ao contrário, usando o método `reverse`.

```
export default {
    listaTarefas: estado => {
        let lista = estado.tarefas.slice()
        return lista.reverse()
    },
}
```

Assim como fizemos para registrar as `mutations` e o `state`, temos de voltar ao arquivo `raiz.js` para importar nossos `getters` e passá-los como atributo para a instância da `Store`. Só assim seremos habilitados para usar esses métodos em qualquer componente.

```
...
import getters from './getters'

export default new Vuex.Store({
    ...
    getters: getters,
})
```

Agora, no browser, caso vá ao Devtools, você verá que, ao lado do nosso `state`, temos o quadro de `getters`, e nele já constará o método que acabamos de criar. Caso adicione alguns itens à nossa lista, verá que o retorno desse método será a mesma lista,

embora ao contrário, como podemos ver na figura:

Figura 9.5: Getters formatam e retornam um elemento do state

O objetivo de um `getter` é parecido com o das propriedades computadas. Ele deve formatar e retornar um dado formatado. Com um exemplo de `getter` já pronto, vamos exibir nossa lista na interface.

Para isso, crie o componente `Tarefas` e, no seu `template`, teremos uma lista `ul` com seus itens `li` percorrendo uma variável chamada `tarefas`. Dentro de cada item da lista, estarão o botão de apagar (criado na seção anterior) e o nome daquela tarefa.

```
<ul>
  <li v-for="tarefa in tarefas">
    <lv-deltarefa :tarefa="tarefa"></lv-deltarefa>
    {{ tarefa }}
  </li>
</ul>
```

Note que, no componente do botão de exclusão, estou passando a propriedade `tarefa`, que valerá o nome da nossa tarefa. Assim cada item da lista terá um botão de exclusão que, quando pressionado, excluirá aquele item.

No `script` do componente `Tarefas`, vamos simplesmente importar o botão para apagar a tarefa, e criar uma propriedade computada. Seu papel é representar o `getter` da lista, que vai

retorná-la já ao contrário.

```
import LvDeltarefa from './DelTarefa.vue'
export default{
    name: 'lv-tarefas',
    components: {LvDeltarefa},
    computed: {
        tarefas() {
            return this.$store.getters.listaTarefas
        }
    },
}
```

Para de fato ver essa lista funcionando, precisamos chamar esse componente no `App` . Vamos importá-lo e registrá-lo, assim:

```
...
import LvTarefas from './Tarefas.vue'

export default {
    ...
    components: {LvAddtarefa, LvTarefas},
    ...
}
```

Agora, no `template` do `App` , não podemos esquecer de usar a tag do componente de lista. Declare-a depois do formulário de criação de tarefa:

```
<lv-addtarefa></lv-addtarefa>
<lv-tarefas></lv-tarefas>
```

Ao executar o código no browser, teremos primeiramente nossa lista em branco, sendo exibido apenas o formulário de cadastro de tarefas, mas basta cadastrarmos uma nova tarefa para vermos a mágica já acontecendo. Observe que já temos, também sendo exibido, o botão de exclusão e, ao pressioná-lo, nossa tarefa é excluída:

Figura 9.6: Apagando uma tarefa da lista

Obtivemos um simples `getter` funcionando, mas `getters` podem ir além e receber parâmetros. Isso é fantástico para quando temos de realizar buscas em dados da `store`.

Aproveitando isso, vamos fazer nosso formulário de busca de tarefas. Para tanto, criaremos mais um `getter` no arquivo `getters.js`. Ele se chamará `buscaTarefas` e receberá um parâmetro nominado `termo`, e então percorrerá nossa lista, pesquisando pelas tarefas que contêm o `termo` buscado no seu nome. No fim, retornará um `Array` com todos os resultados.

```
buscaTarefas: estado => termo => {
  let resultado = []
  if(termo != '') {
    for (let i = 0; i < estado.tarefas.length; i++) {
      if(estado.tarefas[i].indexOf(termo) > -1)
        resultado.push(estado.tarefas[i])
        }
  }

  return resultado
}
```

Note que agora temos duas funções encarrilhadas. A primeira está recebendo o parâmetro que representa nosso `state`, já a segunda recebe nosso `termo`. Precisamos fazer assim para transformar o `getter` em um método, porque até então ele era um simples atributo, tanto que estávamos chamando-o sem parênteses, assim: `this.$store.getters.listaTarefas`.

O próximo passo é criar o componente `Busca`. No `template`, declare um `input` que representa a variável `termo`. Abaixo disso, vamos exibir o resultado da busca, o que pode ser uma lista, então faremos uma lista `ul` e seus itens `li` percorrendo e exibindo os itens da variável `resultado`.

```
<label>Buscar por</label>
<input type="text" v-model="termo" >
<ul>
    <li v-for="tarefa in resultado">{{ tarefa }}</li >
</ul>
```

No `script`, vamos declarar a variável `termo` no `data`. Já a variável `resultado` será uma propriedade computada, pois ela executará nosso `getter`, criado anteriormente, enviando o `termo` buscado como parâmetro. O retorno será um `Array` com os resultados da busca.

```
export default{
    name: 'lv-busca',
    data() { return { termo: '' } },
    computed: {
        resultado()
        {
            return this.$store.getters.buscaTarefas(this.termo)
        }
    },
}
```

Importe e registre esse componente no `App.vue`, assim ele será exibido na tela. Ao verificar no navegador, além da lista e do nosso formulário de criação, teremos o formulário de busca que já estará funcionando. Ao digitar um nome de tarefa, será retornada uma lista com as tarefas que possuem aquele nome, como podemos ver na figura:

Buscar por cru

- torcer para cruzeirao

Figura 9.7: Buscando por uma tarefa da lista

A demonstração citada no início do capítulo está pronta, criando, excluindo e buscando por tarefas. Mas nem tudo está da forma mais correta. Veremos nas seções posteriores o que há de contraindicado e como melhorar o processo.

9.5 ACTIONS — EXECUTANDO MUTAÇÕES INDIRETAMENTE

As ações são muito parecidas com as mutações, porém existem diferenças. Ações podem ser assíncronas — executadas juntamente com outra tarefa — e têm o único papel de executar `commits`, enquanto mutações alteram o estado de um dado.

O uso de ações no Vuex não é algo obrigatório e viria a ser melhor aproveitado caso tivéssemos, junto com a `store`, uma API externa ou dados muito complexos que demorassem para serem mutados, assim podendo fazer uso do assincronismo.

Como cada ação é usada para rodar um `commit`, temos como objetivo desta seção eliminar todos os `commits` que fizemos anteriormente, mas antes devemos criar as ações que os executam. No diretório `loja`, crie o arquivo `acoes.js`, e nele vamos armazenar nossas ações.

No arquivo de ações, exporte um objeto com os métodos `adicionarTarefa` e `removerTarefa`. O papel de cada um será

apenas executar o `commit` na mutação correspondente. Veja que recebemos um `contexto` e um `valor` como parâmetros, que respectivamente indicam a `store` e o `valor` passado.

```
export default {
    adicionarTarefa(contexto, valor) {
        contexto.commit('ADD_TAREFA', valor)
    },

    removerTarefa(contexto, valor) {
        contexto.commit('DEL_TAREFA', valor)
    },
}
```

Para executar essas ações, temos de trocar todos os `commits` pelo método `dispatch`. Este recebe como parâmetro a ação que devemos executar. Então, no `AddTarefa.vue`, troque nosso `commit`:

```
this.$store.commit('ADD_TAREFA', this.tarefa)
```

Por uma linha que chame a ação criada:

```
this.$store.dispatch('adicionarTarefa', this.tarefa)
```

Faça o mesmo no componente `DelTarefa.vue`, porém executando a ação `removerTarefa`. Agora, ao testar no navegador, podemos ter o mesmo resultado. No funcionamento nada muda, mas agora não iniciamos uma mutação diretamente no componente, mas sim na própria loja, via `action`.

Com o fim desta seção, vimos todo o fluxo do Vuex e tudo o que ele nos proporciona para compartilhar estados. Podemos ver esse fluxo no diagrama tirado da documentação e traduzido logo adiante:

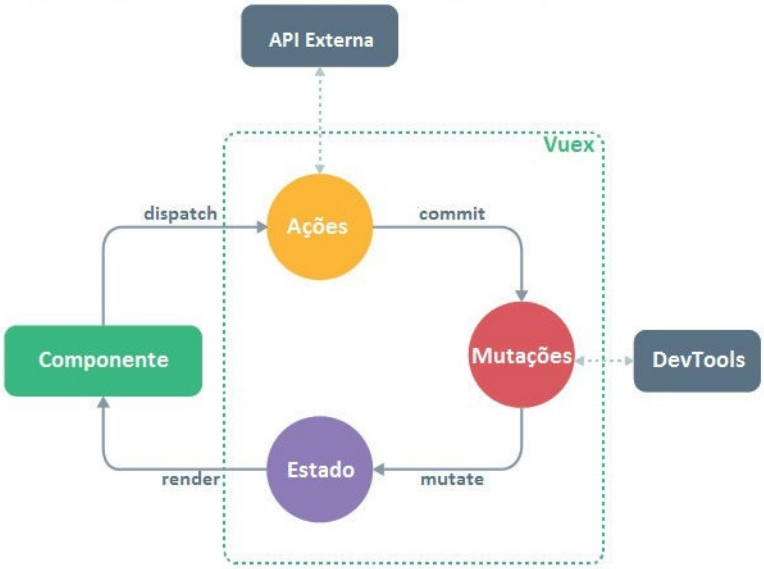

Figura 9.8: Fluxo do Vuex

Note que, na figura anterior, temos nossa `store` do Vuex e, dentro dela, ações, mutações e estado (além dos `getters` omitidos na imagem). Um componente qualquer pode solicitar uma ação, e essa ação pode ou não fazer uso de uma API externa, mas no fim ela sempre executa um `commit` que, por sua vez, faz uma mutação que altera o estado da aplicação. Quando esse dado é alterado, a reatividade trata de renderizá-lo na tela usando ou não o `getter`.

9.6 MODULES — ORGANIZANDO INFORMAÇÕES

Como podemos ver, o Vuex demanda a criação de muitos arquivos para ter uma pequena organização, mas não acabamos

por aqui. Para aqueles que gostam de algo bem separado, a biblioteca disponibiliza os módulos.

Um módulo é um grupo composto de `state`, `actions`, `getters` e `mutations`. No fim, podemos adicionar diversos módulos na nossa `store` e, tendo esses arquivos separados por módulos, nossa loja fica mais bem dividida.

Atualmente, temos na `store` apenas nossa lista de tarefas. Ela já demanda muitos métodos, mas imagine se tivéssemos o estado de uma agenda de contatos nesse mesmo projeto? As coisas ficariam mais confusas, pois teríamos, por exemplo, no arquivo de mutações, além dos métodos `ADD_TAREFA` e `DEL_TAREFA`, os `ADD_CONTATO` e `DEL_CONTATO`. Logo, isso tende a crescer ao infinito.

Mas agora, vamos poder separar isso muito bem, bastando criar dois módulos: um guardaria as coisas referentes às `Tarefas` e outro seria responsável pelos `Contatos`. Como exemplo, vamos apenas criar um módulo chamado `Tarefas`. Ele representará toda a loja usada na nossa lista de tarefas, deixando suas `ações`, `mutações`, `getters` e `estado` separados dos demais.

Para fazer isso com um grau maior de organização, criaremos o diretório `modulos` dentro de `loja` e, dentro desse diretório, outra pasta chamada `tarefas`. Vamos copiar os arquivos `mutacoes.js`, `estado.js`, `getters.js` e `acoes.js` para lá, assim teremos todos os dados relacionados a tarefas em uma mesma pasta.

Ainda dentro da pasta `tarefas`, precisamos de um arquivo para unir todas as partes desse módulo, então criaremos um

chamado `raiz.js`, no qual vamos importar essas partes e exportá-las em um objeto JavaScript.

```
import estado from './estado'
import mutacoes from './mutacoes'
import getters from './getters'
import acoes from './acoes'

export default {
    state: estado,
    mutations: mutacoes,
    getters: getters,
    actions: acoes
}
```

Antes de tudo, perceba que nosso módulo ficou um pouco errado. Se reparar no arquivo `loja/modulos/tarefas/estado.js`, ainda temos a variável `nome` que tínhamos criado para testar o `state`. Precisamos retirá-la daí, pois ela não tem nenhuma ligação com a lista de tarefas. Por isso, dentro do diretório `loja`, crie um arquivo chamado `estado.js` e exporte essa variável como um objeto.

```
export default {
    nome: 'VueJS',
}
```

Note que passamos a ter dois estados: um do módulo `Tarefas` e outro no escopo global da loja. Com isso temos algo mais bem-estruturado e dividido.

O último passo é refazer o arquivo `loja/raiz.js`, pois já não temos aqueles arquivos de mutações etc. dentro da `loja`, dado que os movemos para o módulo. Logo, precisamos apenas importar nosso `state` global para pegar a variável `name` e, por fim, importar nosso módulo `Tarefas` para registrá-lo na `store` usando o atributo `modules`.

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

import estado from './estado'
import Tarefas from './modulos/tarefas/raiz'

export default new Vuex.Store({
    state: estado,
    modules: {
        Tarefas
    }

})
```

Ao ver no navegador, nada mudou. Nosso sisteminha continua com o funcionamento normal, mas o Devtools se alterou um pouco. Agora no seu `state`, podemos ver o módulo `Tarefas` e, só dentro dele, temos nossa lista, como podemos ver a seguir:

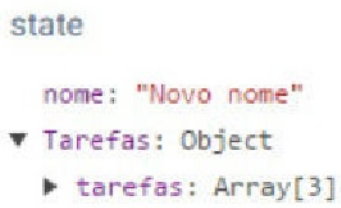

Figura 9.9: Módulo Tarefas no Devtools

9.7 MAPEANDO ESTADO EM COMPONENTES

Você pode ter notado que o uso do Vuex deixou o código dos nossos componentes feio — com linhas grandes e repetidas —, pois sempre que queremos fazer algo relacionado à `store`, temos de acessar a variável de instância `this.$store`. Isso afeta a elegância do código.

Porém, podemos ocultar o acesso a essa variável. Para isso, precisamos mapeá-la transformando seus atributos em atributos do componente, como `methods` ou `computed`.

Mapeando nossos dados da loja, teremos um código muito mais bonito e muita praticidade, sem termos de acessar `store` toda hora. Para fazer isso, precisamos de um plugin do Babel, já que o mapeamento é feito através de *spread operator* — operador que transforma um `Array` em atributos/ parâmetros.

Logo, é preciso instalar o plugin que dá suporte a esse operador. Vamos executar este comando na pasta do projeto:

```
npm install --save-dev babel-plugin-transform-object-rest-spread
babel-preset-latest
```

Com o plugin instalado, precisamos dizer que vamos usá-lo no projeto. Alteraremos o arquivo `.babelrc`, presente na raiz do projeto. Registrando o plugin, seu arquivo ficará parecido com este:

```
{
  "presets": [
    ["latest", {
      "es2015": { "modules": false }
    }]
  ],
  "plugins": ["transform-object-rest-spread"]
}
```

Já temos tudo instalado e configurado, agora vamos começar a mapear nossa `store`!

Mapeando states

Podemos transformar nossas variáveis de estado em

propriedades computadas do componente. Para testar isso, vamos importar a constante `mapState` do `vuex` no componente `App`. No atributo `computed`, chamaremos o método importado usando o `spread`, e passaremos como parâmetro um objeto com os nomes das variáveis de estado que queremos pegar. No caso, pegaremos apenas a variável `nome`, pois só ela é usada no componente `App`.

```
import {mapState} from 'vuex'

export default {
    ...
    computed: {
        ...mapState(['nome'])
    },
}
```

Note que, pela primeira vez no livro, o `...` no código não quer dizer 'qualquer coisa', e sim representa o `spread operator`, dado que precisamos usá-lo ao chamar o método `mapState`.

Agora no `template` desse componente, podemos tirar o acesso direto à `store` mudando onde estava sendo exibido `$store.state.nome` para apenas `nome`, porque agora nossa variável de estado é uma propriedade computada. Veja que eliminei o método `mounted` do componente, já que ele estava alterando um valor diretamente pelo `state`, o que é errado, pois para alterar algo, deveríamos usar mutações.

Mapeando getters

O processo de mapear `getters` da loja é muito parecido com o anterior. Nossos `getters`, como geralmente são dados, devem ser representados localmente em componentes por propriedades

computadas.

Como parece que queremos remover todos os acessos à variável `$store` dos nossos componentes, temos de editar dois deles, `Tarefas` e `Busca`, uma vez que ambos acessam os `getters` da loja. Vamos iniciar pelo componente `Tarefas`, importando a constante `mapGetters` do `vuex` e usando-a com o `spread operator`, para transformar nosso getter `listaTarefas` em uma propriedade computada chamada `tarefas`.

```
import {mapGetters} from 'vuex'

export default{
    ...
    computed: {
        ...mapGetters({
            tarefas: 'listaTarefas'
        })
    },
}
```

Note que fizemos praticamente o mesmo processo: mapeamos os `getters` que foram informados dentro do objeto passado para o `mapGetters`. Nesse caso, queríamos renomeá-lo, então criamos um atributo com o nome desejado, e colocamos seu valor igual ao `getter` que ele vai representar.

Vamos para o componente `Busca`. Ele é um pouco distinto, pois seu `getter` recebe um parâmetro, mas nada muda no processo de mapeamento. Vamos mapeá-lo como o anterior, usando as propriedades computadas.

```
import {mapGetters} from 'vuex'

export default{
    ...
```

```
    computed: {
        ...mapGetters({
            resultado: 'buscaTarefas'
        })
    },
}
```

A diferença vem na hora de usar a `computed` `resultado`. Note que antes a usávamos como um método `this.$store.getters.resultado(this.termo)`, então, para fazê-la funcionar, basta indicar no `for` que ela é um método e passar o parâmetro `termo` para ele, deixando-a assim:

```
<li v-for="tarefa in resultado(termo)">{{ tarefa }}</li >
```

Agora sim temos nossos `getters` mapeados, recebendo ou não parâmetros.

Mapeando actions

Por fim, falta mapearmos nossas ações de criar e excluir uma tarefa. No livro, ficaremos devendo o mapeamento de mutações, pois estamos seguindo o fluxo indicado do Vuex, usando `actions`. Porém, o processo para mapear mutações é o mesmo que o usado para mapear ações, apenas alterando a variável importada para `mapMutations`.

Como ações são métodos, é necessário mapeá-las como métodos do componente, e não mais como propriedades computadas. Mas o processo ainda assim é bem parecido com o anterior.

Note que estamos usando ações tanto no componente `AddTarefa` quanto no `DelTarefa`. Começaremos pelo `AddTarefa`, importando a constante `mapActions` do `vuex`,

usando-a no atributo `methods` para mapear a ação `adicionarTarefa` e vinculando-a a um método chamado `adicionar`.

```
import {mapActions} from 'vuex'

export default{
    ...
    methods: {
        ...mapActions({
            adicionar: 'adicionarTarefa'
        }),
        submeterTarefa() {
            if(this.tarefa != '')
                this.adicionar(this.tarefa)
            this.tarefa = ''
        }
    }
}
```

Note que criei outro método, o `submeterTarefa`, porque temos uma validação e uma ação antes e depois de executar a ação da `store`. Por isso penas troquei o acesso direto à `store` por um método mapeado. Outra mudança foi no `template`: no botão `Adicionar`, executamos o método `submeterTarefa`, e não mais o `adicionar`.

Chegou o momento de eliminar o último acesso feito à variável `$store`. Ele se encontra no componente `DelTarefa`, e o processo será praticamente igual ao feito no componente `AddTarefa`. Entretanto, não temos validação e ações extras, logo podemos mapear a ação da loja e usá-la diretamente no `template`.

```
import {mapActions} from 'vuex'

export default{
    ...
```

```
    methods: {
        ...mapActions({
            apagar: 'removerTarefa'
        })
    }
}
```

No código anterior, vinculamos a ação `removerTarefa` com o método `apagar` do componente. Como nossa ação recebe um parâmetro, precisamos enviá-lo para a ação no `template` :

```
<button @click="apagar(tarefa)">D</button>
```

Com isso, acabamos com o acesso à variável 'feia' `$store` , e deixamos nosso código muito mais elegante, abordando quase todo o escopo do Vuex.

9.8 TESTE SEU AVANÇO

Enfim, vamos terminar este longo capítulo? Não poderia deixar de lado alguns exercícios, não é mesmo?

Memorize a teoria

1. Por que e quando devemos usar o Vuex?
2. Qual é a melhor maneira de alterar dados de estado do Vuex? Por quê?
3. Como podemos melhorar a organização da nossa `store` quando temos diversos dados de estado, mutações etc.?
4. Caso um desenvolvedor queira eliminar o acesso ao `$store` , o que ele deve fazer?

Na prática

1. Continue o código feito no capítulo e, ao lado da lista de tarefas, faça uma lista de contatos. Nela vamos poder salvar nome e e-mail de uma pessoa, listar essa agenda na tela e apagar contatos.

   Sinta-se à vontade para usar o `vue-router` e separar tudo isso em telas diferentes, caso queira.

   Obs.: ainda fui bonzinho, podia ter pedido para buscar e editar dados, e outra série de coisas. ;)

Capítulo 10

CRIANDO E DIVIDINDO SERVIÇOS

Já com uma boa base de Vue, rotas e estados, podemos começar a captar dados externos para a nossa aplicação. Para isso, temos centenas ou milhares de maneiras diferentes, variando de desenvolvedor para desenvolvedor. Independente disso, um serviço sempre terá o mesmo papel, que é enviar e receber dados para/de ambientes externos, normalmente via *Ajax*.

Aqui no livro, usaremos a biblioteca *Axios*. Como dito antes, essa é uma opção pessoal, prefiro o Axios por ter uma sintaxe mais amigável e sua divisão de serviços ser feita facilmente. Além dele, há a famosa *Vue Resource*, porém ela não possui uma boa organização de código, logo a descartei.

Neste capítulo, vamos simular o recolhimento de dados externos em uma aplicação de exemplo. Normalmente, aplicações coletam dados externos para exibi-los, e esses dados podem vir de uma API de serviços, como o Facebook, e têm o único objetivo de agregar dados do back-end no front-end.

Como exemplo, usaremos o método `GET`. Não se preocupe, vamos fazer algo mais complexo na parte de projeto orientado do

livro. Por enquanto, para nos familiarizarmos com o Axios, faremos apenas um sistema que carrega dois JSONs locais e os mostram em uma lista, na qual simulam uma API externa.

Prepare-se

Caso você não conheça Ajax e requisições `GET`, `POST`, `PUT` e `DELETE`, pesquise por *protocolo HTTP* para continuar a leitura com um melhor entendimento. Veja a página na Wikipédia: https://pt.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

Em resumo, o protocolo HTTP tem em seu escopo requisições e respostas. Ele é responsável pelo transporte de dados em sistemas distintos. Requisições são ações que enviam solicitações para um sistema externo, e essas solicitações podem ou não ter dados embutidos, como o login de um usuário. Já a resposta é aquilo que o sistema externo retorna para nossa aplicação, geralmente retornando dados solicitados ou mensagens de erro.

`GET`, `POST`, `PUT` e `DELETE` são as formas que uma requisição pode tomar. Quando você quer apenas pegar dados externos sem qualquer validação, use `GET`. Se precisa enviar dados seguros e também dados para criar um novo registro, utilize o `POST`. Quando precisamos enviar dados para editar um registro no sistema externo, usamos o `PUT`. Já para solicitar a exclusão de um registro nesse sistema externo, utilizamos o `DELETE`.

Mesmo com esse resumo, se você nunca ouviu falar sobre

isso, é melhor estudar um pouco ou pular essa parte, pois geralmente é aqui que começamos a migrar do front para o back-end.

Antes de tudo, para servir de exemplo deste capítulo, vamos criar um projeto. Nele devemos instalar o plugin do Axios com o seguinte comando:

```
npm install axios --save
```

Com o plugin instalado, vamos criar um diretório chamado `servicos` dentro do `src`, no qual teremos um arquivo nominado `configuracoes.js`, que exportará a configuração do Axios. Nele vamos colocar a base da nossa API, ou seja, em qual link buscaremos os dados — nesse caso, no próprio projeto. Também definiremos um tempo máximo de resposta de 10 segundos.

```
import axios from 'axios';

export const http = axios.create({
  baseURL: '/',
  timeout: 10000,
});
```

Ainda dentro da pasta `servicos`, vamos criar os arquivos `carros.js` e `avioes.js`, que serão nossos serviços. Exportaremos os objetos `Carro` e `Aviao` e, dentro deles, adicionamos nossos métodos. Por enquanto, declararemos em cada um o método `lista`, e nesse método apenas retornaremos a resposta de uma requisição `GET` para um JSON que ainda vamos criar, usando a configuração do Axios já feita.

```
import {http} from './configuracoes'
```

```
export default {
    lista: () => { return http.get('/dados/carros.json') }
}
```

O arquivo `avioes.js` será igual ao anterior, apenas trocando `carros.json` para `avioes.json`. Para o exemplo, vamos criar esses arquivos, que representam as tabelas do nosso banco de dados, de onde os dados vêm. Criamos o diretório `dados` dentro da raiz do projeto e, dentro dele, teremos dois arquivos, o `carros.json`:

```
[
    {"nome": "Panarema", "marca": "Porsche"},
    {"nome": "Macan", "marca": "Porsche"},
    {"nome": "Virage", "marca": "Aston Martin"},
    {"nome": "Rapide", "marca": "Aston Martin"},
    {"nome": "E-Type", "marca": "Jaguar"},
    {"nome": "Mark 2", "marca": "Jaguar"}
]
```

E o `avioes.json`:

```
[
    {"nome": "Gazelle", "marca": "Aérospatiale"},
    {"nome": "Impala", "marca": "Atlas"},
    {"nome": "Bonanza", "marca": "Beechcraft"}
]
```

Agora no `App.vue`, podemos simplesmente importar nosso serviço e exibir os dados do arquivo JSON. Para testar, vamos chamar nossos métodos `lista` no método `mounted` do componente:

```
import Carro from './servicos/carros'
import Aviao from './servicos/avioes'

export default {
  ...
  mounted() {
```

```
    Carro.lista().then(dado => console.log(dado.data) )

    Aviao.lista().then(dado => console.log(dado.data) )
  }
}
```

Ao iniciar a live no navegador, podemos abrir o console e ver nossos dados sendo exibidos:

```
▼ [Object, Object, Object, Object, Object, Object]
  ▶ 0: Object
  ▶ 1: Object
  ▶ 2: Object
  ▶ 3: Object
  ▶ 4: Object
  ▶ 5: Object
    length: 6
  ▶ __proto__: Array(0)
▶ [Object, Object, Object]
```

Figura 10.1: Dados recebidos via AJAX

Já que temos esses dados, por que não os exibir na tela para o usuário? Vamos criar um componente chamado `Lista`, e nele receberemos: uma `lista`, que deve ser um `Array` ou `Object`; e um `titulo`. Mostraremos esse `titulo` e uma lista HTML que percorre a `lista` que foi passada.

```
...
<h4>Lista de {{ titulo }}</h4>
<ul>
  <li v-for="item in lista">
    {{ item.nome }} de <i>{{ item. marca }}</i>
  </li >
</ul>
...
<script>
export default{
    ...
    props: {
        titulo: null,
        lista: {
```

```
            required: true,
            type: Array | Object
        }
    }
}
</script>
```

Agora com esse componente que exibe qualquer lista com `nome` e `marca` para nós, vamos fazer com que nosso componente `App` exiba os dados pegos no serviço. Para isso, no `script`, declararemos no `data` as variáveis `carros` e `avioes` para armazenar a lista recebida de cada JSON. Além disso, vamos importar e registrar o componente `Lista`, e alterar o `mounted` para armazenar nosso resultado.

```
import Carro from './servicos/carros'
import Aviao from './servicos/avioes'
import LvLista from './Lista.vue'

export default {
  name: 'app',
  data() { return { carros: {}, avioes: {} } },
  components: {LvLista},
  mounted() {
    Carro.lista().then(dado => this.carros = dado.data)

    Aviao.lista().then(dado => this.avioes = dado.data)
  }
}
```

No `template`, basta chamarmos o componente `Lista` enviando uma lista e um título, para que ele possa exibir nossos dados. Vamos usá-lo duas vezes, pois temos duas listas.

```
<lv-lista titulo="Carros" :lista="carros"></lv-lista>
<lv-lista titulo="Aviões" :lista="avioes"></lv-lista>
```

O resultado disso será nossas duas listas com os dados pegos do JSON sendo exibidas na interface do usuário, como na figura a

seguir:

Lista de Carros

- Panarema de *Porsche*
- Macan de *Porsche*
- Virage de *Aston Martin*
- Rapide de *Aston Martin*
- E-Type de *Jaguar*
- Mark 2 de *Jaguar*

Lista de Aviões

- Gazelle de *Aérospatiale*
- Impala de *Atlas*
- Bonanza de *Beechcraft*

Figura 10.2: Lista exibida com dados do JSON

Tratando-se de uma biblioteca externa e simples de ser usada, este capítulo ficou bem curto. Para ganhar mais conhecimento sobre a biblioteca, acesse sua documentação (https://github.com/mzabriskie/axios). Com ela, basicamente temos a possibilidade de enviar e receber dados, expandindo consideravelmente o poder de uma aplicação. Como já citei, veremos mais sobre ela no final da obra.

10.1 TESTE SEU AVANÇO

Chegamos a mais um momento de testar seus conhecimentos.

Memorize a teoria

1. Quando devemos usar serviços em uma aplicação?

2. Deve-se usar sempre a biblioteca Axios para criar serviços? Por quê?

Na prática

Ao final destes exercícios, teremos um pequeno sistema de buscar por CEP, e para isso não usaremos mais o JSON local, e sim uma API pública.

1. Crie a configuração do Axios com a URL da nossa API, apontando para https://viacep.com.br/ws/, com tempo limite de 5 segundos.

2. Construa um serviço chamado `cep`. Nele teremos um método chamado `busca`, que recebe um parâmetro e retorna uma requisição `GET` para `cep + /json`, em que `cep` é o valor da variável recebida por parâmetro.

3. No componente `App`, importe o serviço, e crie um `input` e um `button`. Faça com que, ao clicarmos no botão, busquemos o CEP informado no `input`, usando o método `busca` do serviço.

4. Mostre os dados que foram recebidos na tela. O retorno da API é um objeto JSON, com este padrão:

```
{
  "cep": "38300-970",
  "logradouro": "Avenida Nove",
  "complemento": "670",
  "bairro": "Centro",
  "localidade": "Ituiutaba",
  "uf": "MG",
  "unidade": "",
```

```
  "ibge": "3134202",
  "gia": ""
}
```


Para acessar um atributo do resultado, use, por exemplo, `dado.data.bairro`.

Capítulo 11

ACRESCENTANDO FUNCIONALIDADES

Ao longo do livro, vimos que podemos usar outras bibliotecas e extensões para aumentar o poder do Vue. Neste capítulo, trataremos sobre como podemos criar nossas próprias extensões.

A grande vantagem de criar e disponibilizar partes externas para o Vue é o reaproveitamento. Hoje, existem diversos tipos de bibliotecas, como coleções de componentes, rodadores de testes, acrescentadores de funções e outros. Ao criar, podemos ter sempre um código com alguma função que pode ser adicionado a qualquer outro projeto e a qualquer momento.

Para isso, temos três formas de disponibilizar extensões. A primeira não veremos aqui, pois não se enquadra no Vue, que são as libs NPM — pacotes de código que são desenvolvidos em JavaScript para expor funcionalidades para qualquer outro sistema em JavaScript.

Já para o Vue, temos duas possibilidades: as diretivas customizadas, que são feitas para formatação e manipulação de elementos HTML, assim como vimos com `v-text` e outros; e os plugins, que estendem diversas funcionalidades do Vue, assim

como o Vue-router e o Vuex.

11.1 CRIANDO DIRETIVAS CUSTOMIZADAS

Essas diretivas são atributos HTML. No Vue, podemos criar e registrar novas diretivas. Elas normalmente são ligadas diretamente a elementos do HTML, e devem manipular seus atributos para suprir uma necessidade.

Já vimos várias diretivas padrões do Vue no capítulo *Criando e exibindo dados*, como a diretiva `v-on` que tem como objetivo executar uma ação — ou seja, colocar uma escuta em algum evento do elemento, como o `click`. Logo, podemos executar algum código JavaScript quando o elemento que possui essa diretiva for clicado.

Antes de criarmos nossa primeira diretiva, temos de entender um pouco o seu funcionamento. No Vue, uma diretiva pode ter três tipos de atributos que manipulam a ação que ela deve tomar, e um desenvolvedor pode declarar esses atributos junto com a declaração de uma diretiva qualquer.

Por exemplo, na diretiva `v-text`, passamos uma variável para ser exibida, assim: `v-text="variavel"`. Ela trata-se de um atributo chamado `value`, que indica valores reativos que serão passados para serem usados pela diretiva.

Além do `value`, podemos usar argumentos para desviar caminhos que a diretiva pode fazer, como usamos em diretivas como `v-on` e `v-bind`. Ao usar `:click`, estamos usando `args` para alterar o destino da diretiva, por exemplo: `v-on:click="metodoClick"`.

Por fim, podemos utilizar modificadores para alterar o comportamento de uma diretiva. É o caso do `prevent` que usamos no evento de `click`. Ele é empregado para evitar que a página se recarregue ao dar submit em um formulário, assim: `v-on:click.prevent="metodoClick"`.

Como exemplo, podemos criar uma diretiva que formate o texto elemento HTML em letras maiúsculas, minúsculas ou capitalizadas. Normalmente, esse é um papel dos filtros, porém podemos usar isso para exemplificar uma diretiva, dado que normalmente essa também formata dados a serem exibidos.

Para começar, criaremos um projeto para exemplo. Vamos criar o arquivo `src/formato.js` para armazenar a diretiva criada, e nele exportaremos um objeto que tem o atributo `bind`. Esse é um método que será executado sempre que essa diretiva for criada em um elemento HTML.

Esse método recebe três parâmetros: `elemento`, o elemento HTML no qual a diretiva foi registrada; `informacao`, que possui um objeto com os valores passados para a diretiva; e `vnode`, que representa o nó virtual criado pelo compilador Vue. Dentro desse método, temos um `switch` que transforma o case do texto do `elemento` de acordo com o modificador que foi passado na diretiva.

```
export default{
  bind: function (elemento, informacao, vnode) {
    switch( Object.keys( informacao.modifiers )[0] ) {
      case 'maiusculo':
        elemento.innerHTML = elemento.innerHTML.toUpperCase()
      break
      case 'minusculo':
        elemento.innerHTML = elemento.innerHTML.toLowerCase()
      break
```

```
      case 'capitalizado':
        let txt = elemento.innerHTML.split(' ')
        elemento.innerHTML = ''

        for (var i = 0; i < txt.length; i++) {
          elemento.innerHTML += txt[i].substring(0, 1).toUpperCas
e() + txt[i].substring(1) + ' '
        }
      break
    }

  }
}
```

No `main.js`, vamos importar esse objeto e registrá-lo como uma diretiva. Usaremos o método global `directive`, que recebe dois parâmetros: o nome da diretiva e o objeto de métodos que ela possui.

```
import VFormato from './formato'
Vue.directive('formato', VFormato)
...
```

Agora, basta usarmos nossa diretiva em qualquer componente. No `App`, por exemplo, podemos criar alguns elementos `div` com um texto qualquer que chama a nossa diretiva. No caso, temos de passar um modificador para ela, que pode ser igual a `maiusculo`, `minusculo` ou `capitalizado`.

```
<div v-formato.minusculo>
  Meu textinho
</div><br>
<div v-formato.maiusculo>
  Meu textinho
</div><br>
<div v-formato.capitalizado>
  Meu textinho
</div>
```

Note que o nome que demos para a diretiva anteriormente foi

`formato`. Porém, por padrão, o Vue acrescenta como prefixo um `v-`, apenas para identificar que aquele atributo do HTML é uma diretiva do Vue.

Ao executar no navegador com `npm run dev`, teremos um resultado como esperado:

meu textinho

MEU TEXTINHO

Meu Textinho

Figura 11.1: Diretiva formatando textos

11.2 CRIANDO SEUS PRÓPRIOS PLUGINS

Os plugins são mais complexos, e não é à toa que temos alguns com diversos poderes, como é o caso do Vuex, visto no capítulo *Gerenciamento de estado com Vuex*. Em plugins, podemos manipular diversos aspectos de uma aplicação Vue e, por meio deles, registrar diretivas personalizadas, criar métodos globais e de componentes, assim como definir variáveis e mixins.

Só pela descrição dada anteriormente, você deve ter notado que plugins por si só são uma maravilha. Com eles, é possível expandir o reúso de um sistema para diversos outros, pois podemos apenas copiar o arquivo em que criamos o plugin para outro projeto, ou publicá-lo na loja do NPM.

Aqui vamos criar um exemplo bem simples, pois normalmente plugins são bem complexos e usados para criar ações genéricas e novas *features*; por exemplo, eles podem dar a possibilidade de

gerenciar rotas, idiomas, permissões, entre outros. Isso depende muito da imaginação do desenvolvedor.

Nosso plugin será capaz de se infiltrar em componentes, adicionando a eles um método `created` que apenas exibe uma mensagem no console. Ele disponibilizará para todos os componentes um método chamado `$ola` e uma diretiva `v-ola`. Por fim, também registramos um método global chamado `ola`.

Vamos fazer isso tudo em um arquivo chamado `src/plugin.js`:

```
const OlaMundo = {}

OlaMundo.install = function (Vue, options) {
  Vue.ola = () => {
    console.log('Olá mundo do Vue com método global')
  }

  Vue.prototype.$ola = (options)  =>{
    console.log('Olá mundo do Vue com método de instância')
  }

  Vue.directive('ola', {
    bind (el, binding, vnode, oldVnode) {
      el.innerHTML = 'texto do plugin'
    }
  })

  Vue.mixin({
    created: () => {
      console.log('created executado pelo plugin')
    }
  })
}
export default OlaMundo
```

Note que primeiro criamos um objeto que representa nosso plugin e, no final de tudo, o exportamos. Assim podemos importá-

lo para o Vue, o que faremos a seguir. Veja também que definimos um método `install`, que obrigatoriamente deve receber um parâmetro `Vue`, pois precisamos dele para poder registrar diretivas ou qualquer outra coisa na aplicação Vue.

Com isso, no `main.js`, já podemos importar o plugin para qualquer projeto em Vue, e registrá-lo com o método `use`, assim como fizemos no Vue-router no capítulo *Cada um segue seu caminho, com rotas!*:

```
import Plugin from './plugin'
Vue.use(Plugin)
Vue.ola()
```

A partir daí, já podemos usar os seus recursos. Note que já utilizei um deles, o método global `ola` que exibirá no console a mensagem *Olá mundo do Vue com método global.*

Em qualquer componente, como o `App`, podemos usar nossos outros recursos, como o método `$ola` e a diretiva `v-ola`:

```
<template>
  ...
    <div v-formato="'maiusculo'" v-ola>
      Meu textinho {{ $ola() }}
    </div><br>
  ...
</template>

<script>
export default {
  name: 'app',
}
</script>
```

Ao executar isso no navegador, o texto da `div` será trocado para *texto do plugin* por causa da diretiva `v-ola`. Além disso, teremos as seguintes mensagens no console do browser:

```
[HMR] Waiting for update signal from WDS...
Olá mundo do Vue com método global
created executado pelo plugin
created executado pelo plugin
Olá mundo do Vue com método de instância
You are running Vue in development mode
```

Figura 11.2: Console exibe resultado de métodos do plugin

A primeira é executada pelo método global que chamamos no `main.js`, já a segunda e a terceira são do mixin criado no nosso plugin. A primeira foi executada pelo método `created` do componente raiz, já a segunda pelo mesmo método, mas presente no componente `App`. Por fim, temos o resultado do método `$ola`, que foi registrado como método de instância.

Como podemos ver, um plugin pode ser bastante útil e ajuda a aumentar muito o poder dos nossos componentes. Para finalizar o capítulo, veremos na próxima seção como torná-los um pacote NPM. Assim ele estará disponível na nuvem e na comunidade open-source, para que você e seus colegas possam baixá-lo e usá-lo em diferentes projetos, sem dificuldade nenhuma.

11.3 PUBLICANDO PACOTES NPM

Esta seção não se enquadra diretamente no Vue, mas achei válido deixar algumas linhas do livro para se referir ao assunto, já que a possibilidade de publicarmos nosso código de extensões em uma loja online é simplesmente sensacional.

Como dito no final da seção anterior, aqui faremos com que nosso plugin seja instalável em qualquer aplicação pelo comando `npm install meu-plugin`. Assim teremos esse código na nuvem

e acessível em qualquer parte do mundo. Porém, temos alguns requisitos antes de começar:

- Ter o Git instalado no seu computador;
- Ter uma conta no GitHub.

Para iniciar, vamos criar um diretório em qualquer lugar do computador e copiar o arquivo `plugin.js`, criado anteriormente, para lá. No prompt/ terminal, entre nessa pasta e execute os comandos a seguir para criar um repositório Git e um pacote NPM:

```
git init
npm init
```

Ao executar o último comando, serão feitas algumas perguntas. A mais importante ali é o `name`, pois ele será o nome do seu pacote NPM. Caso coloque `plugin-demo-lv`, nosso pacote será instalado pelo comando `npm install plugin-demo-lv`.

No final, teremos um novo arquivo chamado `package.json` na raiz da pasta do plugin, e seu conteúdo deve ser parecido com o seguinte:

```
{
  "name": "plugin-demo-lv",
  "version": "1.0.0",
  "description": "",
  "main": "plugin.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

A seguir, devemos criar um repositório no GitHub para

armazenar esse código. Após criá-lo, copie o link do repositório gerado e, trocando `LINK_DO_REPOSITORIO` pelo link gerado por seu GitHub, execute os seguintes comandos na pasta do nosso plugin:

```
git add .
git commit -m "first commit"
git remote add origin LINK_DO_REPOSITORIO
git push -u origin master
```

Basicamente adicionamos todos os arquivos que criamos em um `commit`, vinculamos esse repositório local com o que geramos no GitHub e publicamos dando o `push`. Nesse último, talvez seja preciso informar seu login do GitHub.

Com o repositório já na nuvem, basta publicá-lo como um pacote do NPM. Para isso, execute o comando `npm publish`. Caso você nunca tenha feito isso, você precisará criar uma conta antes, usando o comando `npm adduser`. Se tudo correr bem, você verá algo parecido com `+ plugin-demo-lv@1.0.0` em seu terminal.

Agora você já pode instalar seu plugin em qualquer aplicação do Vue. Basta executar o comando `npm install plugin-demo-lv` e, no arquivo `main.js` do projeto, importá-lo e registrá-lo usando:

```
import PluginDemo from 'plugin-demo-lv'
Vue.use(PluginDemo)
```

Vale lembrar que `plugin-demo-lv` é o nome do pacote NPM que você registrou no `package.json`.

11.4 TESTE SEU AVANÇO

Com o término deste capítulo, adquirimos conhecimento para expandir ainda mais nosso código. Vamos fazer alguns exercícios para ver como você está se saindo?

Memorize a teoria

1. Qual a maior vantagem em criar diretivas customizadas e plugins?
2. O que os plugins podem manipular em uma aplicação Vue?

Na prática

1. Faça uma diretiva customizada que receba um `Array` com dois números como valores. Ela deve transformar o `innerHTML` do elemento na soma desses dois números.
2. Na mesma diretiva anterior, adicione o modificador `subtrai`. Caso ele seja enviado para a diretiva, faça com que o resultado seja a subtração desses dois valores do `Array`.
3. Crie um plugin que forneça um método de instância chamado `saudacao`. Esse método deve receber um parâmetro denominado `nome`, e apenas retornar um `alert` com o texto `Bem-vindo, + nome`.
4. No mesmo plugin anterior, crie um `mixin` que forneça um método nomeado `chamei`. Este deve executar um `alert` com o texto `Chamei meu pai`. No componente `App`, execute esse método para ver se tudo ocorreu bem.

Capítulo 12

INTRODUÇÃO A TESTES

De acordo com o crescimento de uma aplicação, podemos implementar módulos que afetam o funcionamento de outros. Quando isso acontece, seu sistema quebra e qualquer alteração em um componente pode deixar outro sem funcionar. Para evitar esse contratempo, temos os testes automatizados.

Testes automatizados são um conjunto de arquivos de código que invocam a lógica de negócio do seu sistema, realizando uma série de ações rotineiras e comparando o resultado de todas elas. Assim podemos garantir a integridade do sistema, pois caso obtenhamos resultados distintos, o teste não passará e informará o erro em uma parte do seu código.

Poderíamos ficar um tempão falando sobre esse assunto, e não é à toa que temos vários livros e pesquisas nessa área. Mas apenas farei uma propaganda gratuita aqui para dois livros do melhor autor da Casa do Código, Maurício Aniche:

- *Testes automatizados de software: Um guia prático* — Mostra diversos tipos de testes em sistemas Java usando o framework de testes JUnit, entrando um pouco na prática do TDD.
- *Test-Driven Development: Teste e Design no Mundo Real* — Aprofunda na prática do TDD, com edições feitas para Java, C# e PHP. A leitura não vale só para aprender sobre testes, mas também sobre atributos que aumentam a qualidade de um código.

Apesar de não ser o foco deste livro, na *Parte II* vamos utilizar a prática do TDD para criar um sistema em Vue da melhor maneira possível.

Entrando novamente no ambiente Vue, neste capítulo vamos realizar um exemplo bem simples, pois testes nasceram para serem simples. Caso já tenha trabalhado com testes automatizados no JavaScript, já deve ter conhecido Mocha e Jasmine. Ambos são os rodadores de testes mais famosos do mercado dessa linguagem.

Mas como eu gosto de ser um pouco diferente, aqui no livro usaremos um *runner* de testes chamado AVA. Ultimamente ele vem ganhando mercado por alguns motivos, como:

- Ser mais leve;
- Demandar menos configurações;
- Rodar testes paralelamente;
- Sintaxe simples.

Além disso, temos uma documentação bem simples, em português e com poucas dependências para instalação. Vamos primeiramente instalá-lo em um novo projeto do Vue:

```
npm install browser-env ava vue-node --save-dev
```

Note que, além do pacote `ava`, instalamos também o `browser-env`, responsável por um navegador pequeno para rodar nossos testes, e o `vue-node` que permite que carreguemos componentes Vue.

Com esses pacotes instalados, precisamos realizar algumas configurações para de fato rodar os testes automatizados. A primeira delas será criar um diretório na raiz do projeto chamado `test`, para armazenar os arquivos de testes. Nele criaremos o arquivo `webpack.config.js`, que será responsável por definir como carregar arquivos `.js` e `.vue` dos testes:

```
module.exports = {
  module: {
    loaders: [
      {
        test: /\.vue$/,
        loader: 'vue-loader'
      },
      {
        test: /\.js$/,
        loader: 'babel',
        exclude: /node_modules/
      }
    ]
  },
```

```
  resolve: {
    extensions: ['.js', '.vue']
  }
};
```

Com nosso `loader` de arquivos já pronto, vamos criar ainda na pasta `test` um arquivo chamado `.setup.js`. O objetivo dele é iniciar o navegador de testes e configurar a importação de arquivos `.vue` e `.js` no AVA:

```
const browserEnv = require('browser-env');
const hook = require('vue-node');
const { join } = require('path');

browserEnv();
hook(join(__dirname, './webpack.config.js'));
```

Chegamos à última configuração. No arquivo já existente `package.json`, vamos criar dentro de `scripts` um novo comando, chamado `test`. Ele vai executar o AVA, informando-o de executar os arquivos de testes dentro do diretório `test`. Além disso, vamos declarar um atributo chamado `ava`, e nele acrescentaremos uma extensão de configuração ao AVA, dizendo que ele deverá incluir o arquivo `.setup.js` antes de rodar seus testes:

```
{
  ...
  "scripts": {
    ...
    "test": "ava test/*.test.js"
  },
  "ava": {
    "require": [
      "./test/.setup.js"
    ]
  },
  ...
}
```

Ao final disso, já temos nosso ambiente configurado. Se executarmos o comando `npm run test` que criamos anteriormente, vamos ver uma mensagem em vermelho:

```
1 exception

× Couldn't find any files to test
```

Como ela diz, não temos arquivos de testes no nosso sistema. Embora isso seja um erro, percebemos que é uma confirmação de que já temos o AVA trabalhando.

Para começarmos a fazer os testes, escolhi um exemplo bem simples. Tudo será feito no componente `App`, assim podemos tirar a complexidade do código da aplicação e focar apenas no código dos testes automatizados. Neste capítulo, construiremos um contador com dois botões, um para incrementar seu valor e outro para decrementá-lo. Com esse cenário, temos um ambiente perfeito para muitos testes automatizados, dando uma bela introdução ao assunto.

Para fazer o contador, temos de usar uma variável chamada `total` para mostrar nosso resultado, e dois botões, um para `subtrair` e outro para `somar` o `total`:

```
<template>
  <div id="app">
    <p>Total: {{ total }}</p>
    <button @click="subtrair"> - </button>
    <button @click="somar"> + </button>
  </div>
</template>

<script>
export default {
  name: 'app',
  data () { return { total: 0 } },
```

```
  methods: {
    subtrair() { this.total -- },
    somar() { this.total ++ }
  }
}
</script>
```

Você pode testar se está tudo funcionando ao executar o `npm run dev`, mas isso não é necessário.

Agora é preciso criar nosso teste e abordar todas as possibilidades de uso da aplicação, simulando o decremento e o incremento do total, por exemplo. Assim, garantimos que a aplicação nunca nos trará uma surpresa no resultado. Pensei em três casos que devemos testar para termos a certeza do correto funcionamento do sistema:

1. Quando o componente é montado, `total` deve ser `0`;
2. Quando `subtrair` é chamado, o `total` passará a ser `-1`;
3. Quando `somar` é chamado, o `total` voltará a ser `0`.

Tendo todos os casos já listados, é hora de implementar nosso teste. Crie o arquivo `test/App.test.js`, e nele importe o `ava` que, por convenção de seus criadores, recebe o nome `test`. Também devemos importar o componente que vamos testar, nesse caso o `App`.

```
import test from 'ava'
import App from './../src/App.vue'
```

Vamos continuar, agora colocando nossa primeira abordagem e verificando se o primeiro valor de `total` será `0`:

```
test('total sendo iniciado com valor 0', t => {
    t.is( App.data().total, 0 )
```

```
})
```

Note que chamamos o método `test` importado do `ava`. Nele, passamos um nome para o teste atual e um método. Este método obrigatoriamente deve receber um parâmetro `t` que será nosso objeto de `assert`, pois, com ele, poderemos comparar os resultados finais com o que esperamos.

Ao executar o comando `npm run test` novamente, teremos a mensagem de sucesso: *1 passed*, informando que esse teste passou. Agora vamos realizar nosso segundo teste, no qual devemos chamar o método `subtrair` e, após isso, verificar se o valor da variável `total` se alterou para `-1`. Para tanto, abaixo do código anterior, coloque:

```
test('subtraindo um número do total', t => {
    App.methods.subtrair()
    t.is( App.data().total, -1 )
})
```

Veja que seguimos o padrão anterior, chamando o `test` com um nome e uma função. Nessa função, fizemos as ações desejadas e, por fim, comparamos os resultados com a variável de assert `t`. Mas ao executar novamente o `npm run test`, obtivemos um erro:

```
1 passed
1 failed

D:\vue-livro\exemplos\12-testes\test\App.test.js:10

        App.methods.subtrair()
  10:   t.is( App.data().total, -1 )
  11: })

Actual:

  0

Must be strictly equal to:

  -1
```

Figura 12.1: Erro do teste executado

Basicamente o AVA reclamou que o valor esperado era o `-1`, porém a saída foi igual a `0`. Mas por que isso aconteceu? Não chamamos o método `subtrair` que retira um ponto do `total`?

Sim, chamamos, porém nosso componente `App` não foi montado em nenhum momento, logo seu ciclo de vida não está ativo e as alterações de atributos não serão registradas. Sem montar um componente, poderemos apenas comparar seus valores iniciais, como fizemos no primeiro teste.

Mas como resolvemos isso, sendo que o próprio Vue monta esses componentes automaticamente? A resposta para essa pergunta é relativamente simples, usando Avoriaz!

O Avoriaz é uma biblioteca utilitária para usar componente Vue em testes automatizados. Ela permite que montemos esses componentes e chamemos seus atributos. Além disso, o Avoriaz nos dá a possibilidade de realizar testes de sistema/funcional, nos quais podemos simular cliques em elementos e outras coisas. Para

começar a utilizá-lo, precisamos instalá-lo:

```
npm install avoriaz@3.0.0 require-extension-hooks@0.3.0 require-e
xtension-hooks-vue require-extension-hooks-babel --save-dev
```

Antes de iniciar, temos de alterar o arquivo `.setup.js`, informando que estaremos lendo arquivos `vue` e `js`, usando a extensão nova. A partir de agora, o arquivo `webpack.config.js` torna-se dispensável.

```
require('browser-env')()

var hooks = require('require-extension-hooks')
hooks('vue').plugin('vue').push()
hooks(['vue', 'js']).plugin('babel').push()
```

No nosso teste, vamos importar o método `mount` presente nessa biblioteca, responsável por montar componentes do Vue. Vamos montar o componente `App` e armazená-lo em uma variável, a qual usaremos em todos os testes:

```
import test from 'ava'
import { mount } from 'avoriaz'
import App from './../src/App.vue'

var componente = mount(App)
...
```

Abaixo da linha em que declaramos `componente`, troque todo o uso do `App` para `componente`, assim vamos utilizar o componente já montado, e não o importado. Note que, mesmo assim, o erro persiste, porque o Avoriaz não suporta a chamada direta de métodos dos componentes, dado que eles só devem ser chamados por eventos em elementos HTML.

A solução para isso é usar os recursos de testes de sistema. No caso, vamos simular o clique no botão `-`, pois ele executa o

método `subtrair`, logo, supre a segunda abordagem de teste que foi definida:

```
test('subtraindo um número do total', t => {
    let button = componente.find('button')[0]
    button.trigger('click')

    t.is( componente.data().total, -1 )
})
```

Note que buscamos o primeiro elemento com a tag `button` presente no componente montado e, após isso, simulamos um clique nesse elemento. Esse clique, por sua vez, executou o método `subtrair` que manipulou o atributo `total`, assim ele passou a valer `-1`. Caso você execute os testes, teremos: *2 passed*.

Para abranger o último caso de teste, faremos algo parecido. Agora, vamos buscar o segundo `button` da tela, para simular o seu clique. No fim, a variável `total` deve ser `0`:

```
test('somando um número do total', t => {
    let button = componente.find('button')[1]
    button.trigger('click')

    t.is( componente.data().total, 0 )
})
```

Com isso, chegamos à meta e batemos nossos três casos de testes estipulados. O intuito do capítulo foi apenas mostrar que testes automatizamos podem ser bem simples de serem feitos com o Ava, possuem um código bonito e deixam até o desenvolvedor orgulhoso de criar algo como isso. Nos capítulos finais, veremos como associar isso à prática de TDD. Logo você verá não só os benefícios de verificar se está tudo funcionando, mas também uma melhora na codificação.

12.1 TESTE SEU AVANÇO

Este capítulo foi bem legal, ao menos escrevê-lo. Espero que os exercícios também sejam legais.

Memorize a teoria

1. Quais acontecimentos podem ser prevenidos com a codificação de testes automatizados? Por quê?

2. Por que sem o Avoriaz não conseguimos executar ações dentro de nosso componente?

Na prática

1. Seguindo o código escrito no capítulo, crie um componente chamado `Contato` , e declare três `buttons` nele. Faça com que:

    - Ao clicar no primeiro botão, transforme o texto de `valor` em minúsculo.
    - Ao clicar no segundo botão transforme o texto de `valor` em maiúsculo.

    - Ao clicar no terceiro botão, transforme o texto de `valor` em uma `string` vazia.

    No final, faça o teste automatizado de tudo isso, e pense em todos os casos de teste que devemos abranger, no caso serão quatro (segundo meus cálculos).

CAPÍTULO 13

ALGUNS RECURSOS ESCONDIDOS

Este capítulo finaliza a parte do guia deste livro. Até agora, o que vimos foi quase toda a documentação do Vue, explicada com exemplos distintos e palavras simples, além das bibliotecas extras dos capítulos anteriores. Agora conheceremos coisas bastante úteis, mas um pouco escondidas no Vue, que poucas pessoas acabam conhecendo e usando.

13.1 MANIPULANDO TECLAS DE ATALHO

O Vue nos permite configurar uma extensão para eventos de teclado. Nós podemos dar nomes a teclas e então vinculá-las a eventos, assim podendo fazer ações como: "ao pressionar `F2`, será inserido um texto na tela". Isso possibilita criar teclas de atalhos no nosso sistema.

Apesar de essa configuração ser simples, precisamos saber qual o código de cada tecla do teclado. Um jeito fácil de descobrir isso é criando um componente que apresenta um `input` com uma escuta no `keyup`. Ao pressionar uma tecla, teremos gravado esse evento na variável `e`, logo basta captarmos o `keyCode` do evento, assim ele será exibido no console:

```
<template>
  <input @keyup="alterou">
</template>

<script>
export default {
  methods: {
      alterou(e) { console.log(e.keyCode) }
  }
}
</script>
```

Ao executar o componente anterior, note no console que o código da tecla `F2` é `113`. Devemos registrar essa tecla na configuração de `keyCodes` do Vue, no `main.js`:

```
Vue.config.keyCodes = {
  'f-dois': 113,
}
```

Com a tecla registrada, podemos alterar o componente anterior, para que ele possa ocultar ou mostrar um texto quando a tecla for pressionada em qualquer local do componente. Para isso, criamos um parágrafo que será exibido ou não e uma variável `visivel` que nos mostrará o conteúdo. No evento `keyup`, passamos o modificador `f-dois`, assim o Vue vinculará a esse evento apenas a tecla `F2`. Nele, fazemos a troca de valor da variável `visivel`:

```
<template>
  <div id="app" @keyup.f-dois="visivel = !visivel">
      <p v-if="visivel">Você marcou a opção F2</p>
    ...
  </div>
</template>

<script>
export default {
  data() { return {visivel : false} },
  ...
```

```
}
</script>
```

13.2 CICLO DE VIDA DOS COMPONENTES

No decorrer do guia, usamos diversas vezes os métodos `mounted` e `created`. Eles são `hooks` de vida do componente, e cada um desses métodos é chamado em determinado momento, correspondente a uma fase de vida que o componente está passando. Além deles, podemos ter diversos outros menos usados. Podemos vê-los no diagrama a seguir, retirado da documentação oficial (https://vuejs.org/v2/guide/instance.html):

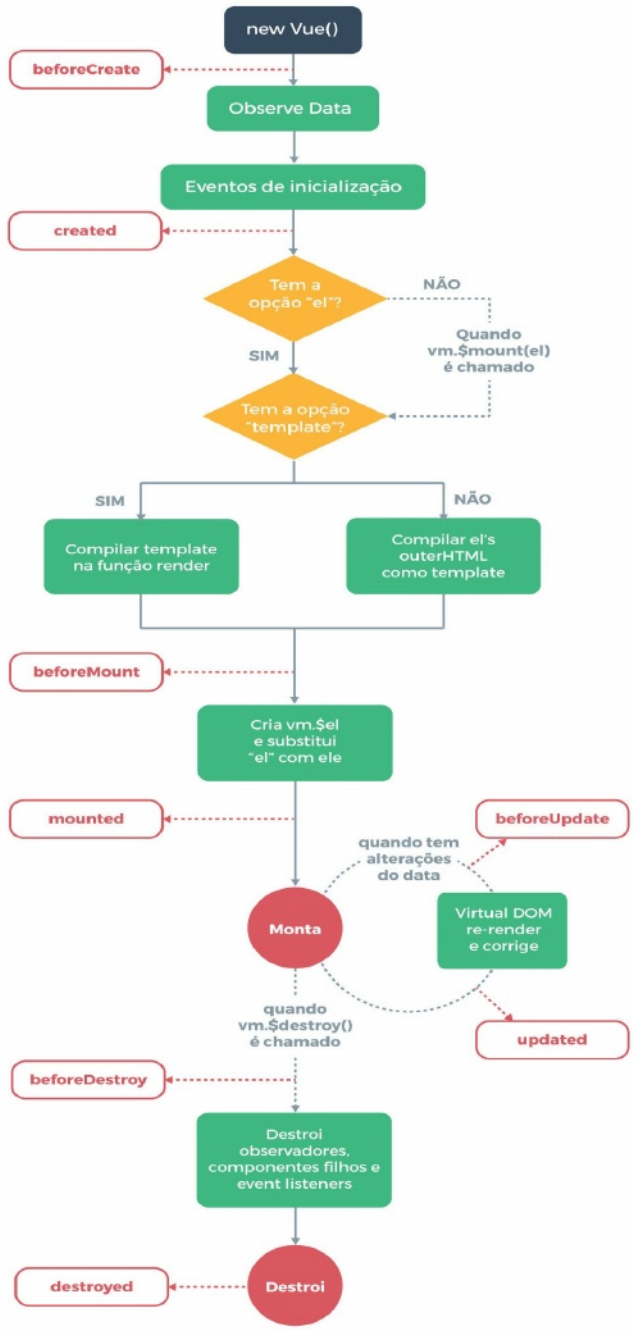
new Vue()
beforeCreate
Observe Data
Eventos de inicialização
created
Tem a opção "el"?
NÃO
Quando vm.$mount(el) é chamado
SIM
Tem a opção "template"?
SIM
NÃO
Compilar template na função render
Compilar el's outerHTML como template
beforeMount
Cria vm.$el e substitui "el" com ele
mounted
beforeUpdate
quando tem alterações do data
Monta
Virtual DOM re-render e corrige
quando vm.$destroy() é chamado
updated
beforeDestroy
Destroi observadores, componentes filhos e event listeners
destroyed
Destroi

Figura 13.1: Ciclo de vida de um componente Vue

No diagrama mostrado, podemos ver que temos: eventos executados antes da criação e da montagem de um componente (`beforeCreate` e `beforeMount`); eventos antes e quando o componente é atualizado (`beforeUpdate` e `updated`); e por fim, eventos quando um componente está prestes a ser destruído e quando é destruído (`beforeDestroy` e `destroyed`).

Com isso, a brincadeira pode ficar bastante complexa no Vue. Por exemplo, quando um componente de formulário de cadastro for destruído (ocultado), outros componentes podem aparecer nessa tela, mantendo uma interação de excelência com o usuário.

13.3 ESTENDENDO COMPONENTES COM EXTENDS

Além dos mixins que vimos no decorrer da obra, temos outra maneira de realizar heranças entre componentes, o `extends` , já citado no livro. Ele tem o mesmo objetivo que um mixin, porém a origem não precisa ser exatamente um `mixin` , mas sim qualquer outro componente. Ou seja, com `extends` , você poderá pegar métodos ou atributos diversos de outros componentes, reaproveitando-os em outro.

Por exemplo, no componente `A` , temos uma variável `titulo` e, no método de montagem `mounted` , formatamos o título para exibi-lo em maiúsculo. Também temos o componente `B` que, além de todas as características do componente `A` , deve ter uma descrição. Logo, podemos reaproveitar o conteúdo do componente

`A` em `B` usando `extends`. Para isso, devemos criar o componente `A`, com o `titulo` e o método `mounted` conforme indicado anteriormente:

```
<template>
    <h1>{{ titulo }}</h1>
</template>

<script>
export default{
    data() { return { titulo: 'meu titulo' } },
    mounted() { this.titulo = this.titulo.toUpperCase() }
}
</script>
```

O segundo passo é importar o componente `A` no `B` e usar o atributo `extends` para informar que todos os atributos do componente `A` estarão disponíveis em `B`. Assim já podemos usar o método `mounted` e a variável `titulo` aqui:

```
<template>
    <div>
        <h2>{{ titulo }}</h2>
        <small>{{ descricao }}</small>
    </div>
</template>

<script>
import LvA from './A.vue'
export default{
    extends: LvA,
    data() { return { descricao: 'Oi tudo bem?' } },
}
</script>
```

Para testar, basta importar ambos e exibi-los dentro do componente `App`. Teremos então os dois funcionando normalmente, com o componente `B` puxando a variável e o método do componente `A`, como esperado.

13.4 TRABALHANDO COM REFERÊNCIAS

Quando importamos um componente dentro de outro, no Vue, podemos criar uma variável para armazenar essa importação. Esse armazenamento se chama referência, e é muito útil para quando precisamos pegar dados do componente que foi importado.

Para registrar uma referência, usamos o atributo HTML `ref`. Podemos tomar como exemplo o mostrado na seção anterior: no componente `App`, registramos o componente `A` e o armazenamos em uma referência chamada `a`.

```
<lv-a ref="a"></lv-a>
```

A partir desse momento, temos uma variável chamada `a` dentro do atributo `$refs` do componente atual. Nele podemos, por exemplo, pegar o valor da variável `titulo` que foi declarada no componente `A`. Para isso, basta criar um método em `App`, assim:

```
mostraTitulo() {
    console.log(this.$refs.a.titulo)
}
```

Ao chamá-lo, veremos no console que o título foi exibido com sucesso. Isso se torna ainda mais útil para coleções de componentes, em que geralmente temos componentes de alerta e notificações que ficam ocultos na tela e são chamados através de referências.

13.5 VARIÁVEL CIFRÃO

Dentro de um componente do Vue, temos variáveis que

representam atributos, como `data` e `props`, desse mesmo componente. Todas essas variáveis começam com um cifrão (`$`).

Provemos de algumas variáveis iniciadas com o cifrão, e existem alguns tipos usados apenas para referências de valores, como é o caso das seguintes:

- `$data` : representa o retorno do método `data()`, ou seja, será a representação de todas as variáveis que criamos no `data` do componente.
- `$props` : parecido com o anterior, porém representa as propriedades externas que o componente deve receber, onde podemos ver as regras de cada uma e acessar os seus valores.
- `$slots` : objeto no qual podemos acessar os elementos de `slot` do componente. Quando declaramos algum tipo de `slot`, ele estará disponível no script a partir dessa variável.
- `$refs` : já a usamos na seção anterior, essa variável se trata de um `Array` contendo todos os elementos que foram carimbados com o atributo `ref`.

Além dessas referências do próprio componente, temos variáveis que referenciam o parentesco de um componente. Como vimos anteriormente no Vue Devtools, uma aplicação Vue é uma árvore de componentes e, por si só, cada componente possui alguns parentes:

- `$parent` : armazena parentes diretos do componente atual na árvore do DOM.

- `$root` : indica a instância raiz do Vue, aquela mesma que geralmente é colocada no arquivo `main.js` .
- `$children` : representa todos os filhos de um componente.

Em um exemplo básico, podemos definir um cenário no qual temos os componentes `App` , `Atendente` , `Fila` e `Cliente` , e temos `App` como ponto inicial. Nele declaramos os componentes `Atendente` e `Fila` e, já no componente `Fila` , temos alguns componentes `Cliente` . Logo podemos definir que:

- `Atendente` tem como `$parent` o componente `Fila` , e vice-versa, já que são parentes diretos, como se fossem irmãos;
- `Fila` tem como `$children` vários componentes `Cliente` , pois estes foram usados dentro do `Fila` . Assim temos também `Fila` e `Atendente` como `$children` do `App` .
- Por fim, temos o componente raiz do arquivo `main.js` como `$root` de todos os outros componentes.

O exemplo pode ser visto mais claramente na figura a seguir:

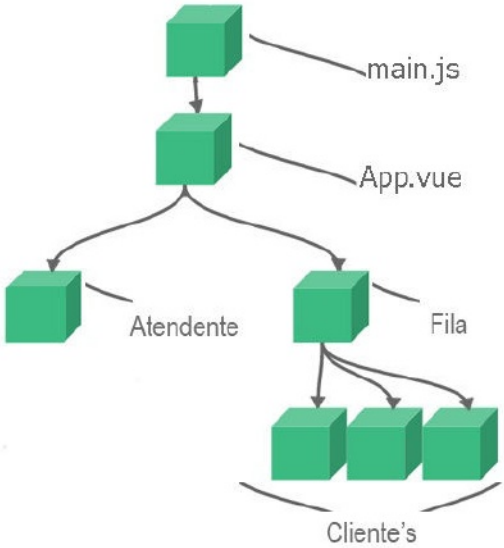

Figura 13.2: Árvore de parentesco

Ainda temos casos especiais, como as variáveis `$el` e `$options`. A primeira representa nosso componente como um elemento HTML, e é muito útil para alterar atributos e comportamentos diretos do HTML. Por exemplo, podemos deixar todo o texto de um componente com a cor vermelha, utilizando o código a seguir:

```
<template>
  <div id="app">
  Meu texto vermelho aqui
  </div>
</template>

<script>
export default {
  mounted () { this.$el.style.color = 'red' }
}
</script>
```

A variável `$options`, por sua vez, já armazena diferentes tipos de dados do nosso componente, mas sua principal característica é permitir a definição de variáveis de configurações de componentes. Por exemplo, podemos definir uma variável chamada `imagens` e, a partir do valor dela, habilitar ou não a exibição de imagens no componente.

```
export default {
  created() {
    this.$options.imagens = false
  },
  mounted () {
    if(!this.$options.imagens)
      console.log('Não carregar imagens')
  }
}
```

13.6 ATUALIZANDO UM COMPONENTE

Às vezes temos componentes que exibem diversos dados. Quando um novo dado é inserido, precisamos recarregar a listagem de acordo com a API externa. Atualizar isso manualmente pode ser trabalhoso, e uma alternativa é forçar a atualização do componente.

Chamando o método `$forceUptade()` em qualquer parte de um componente, ele será recarregado do zero, aliviando suas dores de cabeça com a manutenção de estados difíceis. Um exemplo simples é um componente que, por padrão, tem a fonte da cor vermelha, mas quando o atualizamos, deve passar a azul. Para que isso funcione de imediato e para que possamos ver a cor da fonte ser alterada, usamos o método `$forceUpdate()` para forçar a sua atualização:

```
<template>
```

```
  <div id="app">
    <p v-pre>{{ mensagem }}</p>
    <button @click="atualizar">Atualizar</button>
  </div>
</template>

<script>
export default {
  data() { return { cor: 'red' } },
  mounted () { this.$el.style.color = this.cor },
  updated() { this.$el.style.color = this.cor },
  methods: {
    atualizar() {
      this.cor = 'blue'
      this.$forceUpdate()
    }
  },
}
</script>
```

13.7 DIRETIVA V-PRE

A diretiva `v-pre` é usada para ignorar a interpolação do Mustache, que é usada para exibir o valor de uma variável ou expressão na tela. Com essa diretiva, podemos informar ao Vue que o conteúdo dentro das interpolações (`{{ }}`) não é um pedaço de código, mas que tudo aquilo é um texto que deve ser exibido exatamente como está.

Por exemplo, temos de exibir o texto `'{{ macarrao }}'` na tela e, ao tentar fazer isso, teremos um erro, dizendo que a variável `macarrao` não foi definida. Para burlar esse problema, usamos essa diretiva, assim o Vue interpreta todo o texto dentro do elemento HTML como string e não como código. Veja o exemplo:

```
<p v-pre> {{ macarrao }} </p>
```

13.8 ACESSANDO UM ÍNDICE NO V-FOR

Em programação, em geral as estruturas de repetições como o `for` possuem um contador, no qual podemos saber qual o elemento atual que está sendo repetido. No Vue, o `for`, de forma mais simples, não usa o índice atual do `Objeto` ou `Array` e, para fazer isso, precisamos declará-lo.

Em um exemplo simples, vamos declarar no `data` uma variável chamada `lista`. Nela teremos algumas pessoas, como:

```
lista: [
    {id: 1, nome: 'jose'},
    {id: 3, nome: 'maria'},
    {id: 4, nome: 'joao'},
    {id: 6, nome: 'humberto'},
    {id: 9, nome: 'cristiano'},
]
```

Para percorrer essa lista e mostrar qual a chave do elemento atual, mudamos um pouco a estrutura do `v-for`. Além do `item` percorrido, vamos declarar também o seu `indice`, deixando-o assim:

```
<ul>
  <li v-for="(item, indice) in lista">
    {{ item.id }} - {{ indice }}- {{ item.nome }}
  </li >
</ul>
```

O resultado será a lista exibida e também o índice de 0 a 4, lembrando que sempre o contador se inicia em zero.

Percorrendo objetos, podemos declarar três variáveis, além do `item` e `indice`. É possível também usar a `chave`, para termos a chave ou o nome do objeto que está sendo percorrido. Veja um exemplo:

```
<template>
  <ul>
    <li v-for="(item, chave, indice) in objetos">
      {{ chave }} - {{ indice }}- {{ item.nome }}
    </li >
  </ul>
</template>

<script>
export default {
  data() {
      return {
          objetos: {
              um: {nome: 'manoel'},
              dois: {nome: 'josefa'},
              tres: {nome: 'batista'},
              quatro: {nome: 'reinaldo'},
              cinco: {nome: 'zeze'},
          }
      }
  },
}
</script>
```

Teremos um resultado parecido com o anterior, porém a chave será o nome do objeto, ou seja, indo de `um` até `cinco`.

13.9 MODO HISTORY EM PRODUÇÃO

No capítulo *Cada um segue seu caminho, com rotas!*, havia prometido deixar uma parte aqui, mostrando como usamos o modo *history* em produção, ou seja, quando publicamos o sistema em uma hospedagem. Para eliminar a hash da nossa URL de uma vez por todas, precisamos de um arquivo de configuração no servidor.

Caso sua hospedagem use o Apache como servidor, crie ou edite o arquivo `.htaccess` na raiz do seu projeto, deixando-o

com o conteúdo a seguir:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.html [L]
</IfModule>
```

Nele criamos uma regra para reescrever nossa URL. Em uma explicação simples, todo o tráfego do sistema será reescrito para acessar o arquivo `index.html` do sistema. Na linha 4 do arquivo, informamos com cifrão (`$`) que tudo o que estiver após o `html` será enviado para a aplicação como uma string de dados; no nosso caso, informará qual a rota atual.

Caso sua hospedagem utilize o Nginx ou Node como servidor, para ver o arquivo de configuração correspondente, acesse https://router.vuejs.org/en/essentials/history-mode.html.

13.10 CONCLUINDO

Este capítulo mostrou um pouco de funcionalidades que nem todo mundo conhece. Tivemos o intuito de levar essa informação para que você, leitor e usuário do Vue, possa conhecer algo que talvez usará em projetos específicos.

Pelo fato de tratarmos de diversos assuntos, este capítulo não apresentará exercícios, dado também que usaremos algumas funcionalidades vistas aqui na próxima parte. Nela faremos um projeto prático para finalizar o livro.

Lembre-se de que todos os exemplos vistos neste livro podem ser encontrados em https://github.com/leonardovilarinho/livro-vue.

E todas as respostas dos exercícios vistos na Parte I foram disponibilizadas em https://github.com/leonardovilarinho/vue-livro-avanco.

Projeto orientado

Nesta segunda e última parte do livro, vamos colocar em prática um pouco do que vimos no guia. Temos o objetivo de mostrar ao leitor como realmente usar o que foi aprendido para criar uma aplicação utilizando o Vue.js.

Teremos uma aplicação considerada simples, na qual um usuário pode se registrar e logar no sistema, onde controlará suas anotações, podendo criar novas e apagá-las. Em um resumo, construiremos um *ToDo list online* que é multiusuário.

Durante o projeto, vamos usar uma API feita com o Laravel. Ela permitirá registrar novos usuários e notas, e consultar dados. Caso tenha curiosidade, você pode consultar o código no repositório: https://github.com/leonardovilarinho/api-vue-notes. Ao final, teremos uma aplicação parecida com a disponibilizada em: https://github.com/leonardovilarinho/vue-notes. Você pode usar esse repositório para análise do código ou solucionar dúvidas referentes ao projeto.

Antes de iniciar, vamos deixar uma aplicação já preparada para atender tudo o que usaremos. Após criar um projeto, instale o Vuex, o Vue-router e o Axios, com o seguinte comando:

```
npm install --save vuex vue-router axios
```

CAPÍTULO 14

REGISTRO DE USUÁRIOS

O ponto inicial da nossa aplicação é o registro de um usuário. Qualquer pessoa que acessar a rota `/registrar` do sistema poderá criar uma conta, informando seu nome, e-mail e senha, para posteriormente se logar no site e gerenciar suas anotações.

Primeiramente, como vimos no capítulo *Criando e dividindo serviços* da primeira parte, temos de criar nossos serviços. Eles vão conectar nosso front-end com a API previamente hospedada em http://vue-notes-api.azurewebsites.net. Antes de tudo, precisamos criar um arquivo de configuração do Axios. Para isso, crie o arquivo `src/servicos/configuracao.js`, e nele vamos importar o `axios` e criar um serviço apontando para a URL da API:

```
import axios from 'axios';

export const http = axios.create({
  baseURL: 'http://vue-notes-api.azurewebsites.net/',
  timeout: 10000,
  headers: {
    'Access-Control-Allow-Origin': '*',
  },
});
```

O próximo passo é criar o primeiro serviço. Ele armazenará todos os métodos relacionados ao objeto `usuario`. Crie o arquivo

`src/servicos/usuario.js`, no qual importaremos a configuração criada e exportaremos o serviço contendo um método chamado `registrar`. Este recebe os dados do usuário e envia-os com o método `POST` para a rota `usuario` da API:

```
import {http} from './configuracao'

export default {
    registrar: ({ nome, email, senha }) => {
        return http.post('usuario', { nome, email, senha });
    },
}
```

Com nosso serviço já criado, temos de fazer a tela de registro de usuários. Mas antes disso, note que em todo o sistema haverá diversos campos de texto, e praticamente todo campo e texto possuem um título, então vamos simplificar, criando um componente que representa um `slot` de texto.

Para iniciar, criaremos os testes desse componente. Siga a instalação mostrada no capítulo *Introdução a testes* e, quando seu ambiente de testes estiver pronto, crie o arquivo `test/Input.test.js`. Nele vamos testar o componente de campo de texto. Ele é bastante simples e recebe uma propriedade para representar o título e um `slot`, que será o campo.

Em resumo, montamos o componente com o Avoriaz, passando o `titulo` como propriedade. A seguir, testamos o valor recebido na tag `legend`:

```
import test from 'ava'
import { mount } from 'avoriaz'
import Input from './../src/componentes/LvInput.vue'

let componente = mount(Input, {
  propsData: {
      titulo: 'Nome'
```

```
  }
})

test('componente sendo inicializado com titulo Nome', t => {
    const titulo = componente.find('legend')[0]
    t.is( titulo.text(), 'Nome' )
})
```

Agora crie o componente em `src/componentes/LvInput.vue`. Caso executemos os testes com o componente `LvInput` vazio, teremos um erro, pois a asserção não teve resultado verdadeiro. Faremos esse erro sumir, colocando-o para receber a propriedade externa `titulo`. Por último, vamos declarar um `slot` para ele poder receber o `input` e dar um estilo para focalizá-lo no meio da tela:

```
<template>
    <fieldset style="width: 20%; margin-left: 40%">
        <legend>{{ titulo }}</legend>
        <slot />
    </fieldset>
</template>

<script>
export default{
    name: 'lv-input',
    props: ['titulo'],
}
</script>
```

Após criar esse componente reusável, vamos de fato começar a tela de registro. Como ela se conecta com a API externa para registrar o usuário, seu teste usará um mock para simular esse acesso, assim não sujamos o banco de dados em produção com os testes.

Aqui teremos apenas um teste (temos poucos testes para não estender muito o livro). Vamos criar os testes para o componente

`LvRegistro` . Ele representa a tela de registro do sistema, então precisamos primeiramente preencher os dados do formulário de registro que ficarão no atributo `data` do componente.

Após isso, criaremos um mock para o método `registrar` que chamará a API, assim garantimos que os dados de testes não sejam gravados no banco de dados. Esse mock apenas mudará o valor da variável `estado` , que, por sua vez, representa o texto do botão de registro. Quando ele for diferente de `Registrar` , o botão estará inativo, pois ficará aguardando uma resposta da API. Nosso `Registro.test.js` ficará com o seguinte teste:

```
import test from 'ava'
import Vue from 'vue'
import { mount } from 'avoriaz'
import Registro from './../src/componentes/LvRegistro.vue'
import LvInput from './../src/componentes/LvInput.vue'

Vue.component('lv-input', LvInput)

let componente = mount(Registro)

test('formulário de registro com mock de ação', t => {
    componente.setData({
        usuario: {
            nome: 'Leonardo',
            email: 'leonardo-i@outlook.com',
            senha: '12345678'
        },

    })
    componente.vm.registrar = function() {
        componente.setData({
            estado: 'Carregando...'
        })
    }
    componente.update()
    const botao = componente.find('input[type=submit]')[0]
    botao.trigger('click')
    t.is(botao.hasAttribute('disabled', 'disabled'), true)
```

```
})
```

Note que registramos o componente `lv-input`, pois, como ele é global e registrado no `main.js`, o teste acabaria não o reconhecendo caso ele não fosse declarado. Perceba também como é feita a criação de um mock, pois não havíamos visto isso antes. Para isso, usamos o atributo `vm` do componente, que nos permite acessar e editar qualquer propriedade do componente. No caso, estamos editando o valor do atributo `registrar` e reescrevendo-o com um novo comportamento. Após a concepção do mock, devemos atualizar o componente com o método `update`, assim ele carregará essa modificação.

Esse último teste (formulário de registro com mock de ação) também falhará, pois não temos o componente `LvRegistro` criado. Para fazê-lo ficar verde, vamos criar o componente e colocar nele o formulário de registro com os dados que definimos no teste, o botão de registro e o método `registrar`.

Na pasta `componentes`, crie o arquivo `LvRegistro.vue`. Nele devemos criar a interface de registro usando o `lv-input` três vezes, representando os campos de `nome`, `email` e `senha`, além do botão de registro. Logo, nosso `template` ficará assim:

```
<h2>Registre-se no sistema</h2>
<hr>

<lv-input titulo="Nome">
    <input type="text" v-model="usuario.nome">
</lv-input>

<lv-input titulo="Email">
    <input type="email" v-model="usuario.email">
</lv-input>

<lv-input titulo="Senha">
```

```
    <input type="password" v-model="usuario.senha">
</lv-input>

<br>
<input @click='registrar' type="submit" :value="estado" :disabled
"estado != 'Registrar'">
```

No `script`, devemos criar o método `registrar` que é invocado pelo botão. Ele chamará nosso serviço e, quando obtiver uma resposta positiva, vai exibir um alerta e redirecionar para a rota `/`. Usamos uma variável `estado` apenas para manipular o estado do botão e para desativá-lo enquanto esperamos a resposta da API:

```
import Usuario from './../servicos/usuario'
export default{
    name: 'lv-registro',
    data() { return { estado: 'Registrar', usuario : {nome: '', e
mail: '', senha: ''} } },
    methods: {
        registrar() {
            this.estado = 'Carregando...'
            Usuario.registrar( this.usuario ).then(resposta => {
                if(resposta.data.sucesso) {
                    alert('Registrado com sucesso')
                    this.$router.replace('/')
                }
                else
                    this.estado = 'Registrar'
            }).catch( e => console.log(e) )
        },
    }
}
```

Se tentarmos, ainda não conseguiremos ver o componente no navegador. Para isso, temos de criar a rota `/registrar` e informar que, ao acessá-la, é necessário exibir o componente `LvRegistro`. Vamos fazer isso criando o arquivo `src/rotas.php`, e nele faremos os passos vistos no capítulo *Cada*

um segue seu caminho, com rotas!, exportando uma instância do Vue-router com as rotas do sistema:

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import LvRegistro from './componentes/LvRegistro.vue'
Vue.use(VueRouter)

export default new VueRouter({
    mode: 'history',
    routes: [
        {path: '/registrar',     component: LvRegistro},
    ],
})
```

Para que o Vue-router e o componente `LvInput` funcionem em todo o sistema, devemos primeiramente importar o componente no `main.js` e registrá-lo globalmente no Vue, assim tiramos a necessidade de importá-lo em todos os componentes. Também precisamos importar o arquivo de rotas e registrá-lo na instância da raiz:

```
...
import LvInput from './componentes/LvInput.vue'
Vue.component('lv-input', LvInput)
import router from './rotas'

new Vue({
  ...
  router: router,
})
```

Por fim, devemos abrir o componente `App` e declarar o `router-view` para que o conteúdo das rotas seja exibido. Além disso, teremos um pequeno estilo apenas para centralizar todo o conteúdo no meio da tela:

```
<template>
    <div id="app">
        <router-view></router-view>
```

```
    </div>
</template>

<script> export default { name: 'app'} </script>
<style type="text/css">
    body, html, * { text-align: center }
</style>
```

Ao executar a *live*, com o comando `npm run dev`, e acessar a rota `/registrar`, veremos o formulário já funcionando. Se o registro é feito com sucesso, somos redirecionados para uma tela em branco, que será nossa tela de login, feita no próximo capítulo. Veja a tela de registro:

Figura 14.1: Tela de registro criada agora

CAPÍTULO 15

AUTENTICANDO UM USUÁRIO

Após registrar um usuário, geralmente temos de realizar o login com ele para ter acesso às informações registradas no sistema. Primeiramente, vamos criar o método `logar` no nosso serviço `usuario` para solicitar uma validação dos dados do usuário. Esse método chamará uma rota na API passando o `email` e a `senha` para validarmos se tal usuário de fato existe no sistema. Vamos adicionar esse método no arquivo `src/servicos/usuario.js`:

```
logar: ({ email, senha }) => {
    return http.post('usuario/login', {email, senha});
},
```

Vamos criar uma `Store` em `/src/vuex.js` para armazenar o usuário que será logado no sistema. Para criar a `Store`, seguimos os passos detalhados no capítulo *Gerenciamento de estado com Vuex*, criando apenas um dado no `state`, denominado `usuario`. Este armazena todas as propriedades de um usuário, tendo início com dados nulos. Ele conterá sua respectiva `mutation`, que altera o dado do `state` com um método, e a `action`, que invoca uma mutação, como o fluxo do Vuex recomenda:

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
    state: {
        usuario: { id: 0, nome: '', email: '' }
    },

    mutations: {
        LOGAR: (estado, { id, nome, email }) => {
            estado.usuario = { id, nome, email }
        },
    },

    actions: {
        logarUsuario(contexto, { id, nome, email }) {
            contexto.commit('LOGAR', { id, nome, email })
        },
    }
})
```

O próximo passo é importar e registrar a nossa loja na instância raiz da aplicação. Para isso, acrescente a `Store` no arquivo `main.js`:

```
...
import store from './vuex'

new Vue({
  ...
  store: store
})
```

Antes de criar o componente, criaremos o `Login.test.js` para testar a tela de login. Assim como na tela de registro, temos um formulário com as informações do login e um botão de ação para o componente `LvLogin` ver se tudo está funcionando como deveria. No teste, vamos setar os valores do `data` com os dados que serão preenchidos no formulário de login, fazer o mock do

método de `logar` que contatará nossa API para validar o usuário e, por fim, chamá-lo para ver se o botão de login foi desabilitado, simulando a espera pela resposta da API:

```
...
import Registro from './../src/componentes/LvLogin.vue'

Vue.component('lv-input',      LvInput)
Vue.component('router-link', {render: () => '<a></a>'})

let componente = mount(Registro)

test('formulário de login com mock de ação', t => {
    componente.setData({
        usuario: {
            email: 'leonardo-i@outlook.com',
            senha: '12345678'
        },

    })
    componente.vm.logar = function() {
        componente.setData({
            estado: 'Carregando...'
        })
    }
    componente.update()
    const botao = componente.find('input[type=submit]')[0]
    botao.trigger('click')
    t.is(botao.hasAttribute('disabled', 'disabled'), true)
})
```

Note que, além do `lv-input`, foi declarado o `router-link`, já que o componente `LvLogin` os utiliza. Como eles são globais, precisamos importá-los para que o teste não retorne erros por não achar esses componentes. No `router-link`, fizemos um mock bem simples, pois não temos o seu código na nossa pasta `src`.

Agora basta criar um componente `src/componentes/LvLogin.vue`. Este se parecerá bastante com o componente de registro, porém não terá o campo `nome`, e no

cabeçalho terá um link para a página de registro. Deixaremos o `template` assim:

```
<h2>Entre no sistema</h2>
<router-link to="/registrar">Registre-se</router-link>
<hr>

<lv-input titulo="Email">
    <input type="email" v-model="usuario.email">
</lv-input>

<lv-input titulo="Senha">
    <input type="password" v-model="usuario.senha">
</lv-input>

<br>
<input @click='logar' type="submit" :value="estado" :disabled="es
tado != 'Entrar'">
```

No `script`, vamos criar o método `logar`, invocando quando o botão é pressionado. Ele chamará o método `logar` do nosso serviço e, quando obtiver uma resposta positiva, vai armazenar os dados recebidos da API na `Store` do Vuex, e então redirecionará para a rota `/notas`. Note que ainda temos a propriedade computada que une os dados do formulário:

```
import Usuario from './../servicos/usuario'
export default{
    name: 'lv-login',
    data() { return { estado: 'Entrar', usuario: {email: '', senh
a: ''} } },
    methods: {
        logar() {
            this.estado = 'Carregando...'
            Usuario.logar( this.usuario ).then(resposta => {
                if(resposta.data.sucesso) {
                    this.$store.dispatch('logarUsuario', resposta
.data.usuario)
                    this.$router.replace('/notas')
                }
                else
```

```
                    this.estado = 'Entrar'
            }).catch( e => this.estado = 'Entrar' )
        },
    },
}
```

Vamos criar uma parte da tela inicial, para que possamos ter um sinal de que o login foi realizado com sucesso. Ela mostrará o usuário logado e um botão de logout, usando o Vuex para capturar os dados do usuário que acabou de se logar. Então, precisamos criar um mock que simule nossa `Store` do Vuex, já que os testes não se comunicam com a API. Temos de setar dados fictícios para o usuário logado.

Vamos usar esse mock em diversos testes, pois precisaremos dos dados do usuário logado em várias telas do sistema, por isso criaremos um arquivo solitário para ele. Na pasta `test`, crie o arquivo `mockvuex.js`, que exportará um objeto com um método para criar e montar um componente de teste com o Vuex.

```
import Vue from 'vue'
import Vuex from 'vuex'
import { mount } from 'avoriaz'

export default {
    vuex: function (comp) {
        Vue.use(Vuex)
        const store = new Vuex.Store({
          state: {
              usuario: {
                  nome: 'Leonardo',
                  id: 1
              }
          }
        })

        return  mount(comp, { store })
    }
}
```

Seguindo para os testes do nosso painel, crie o arquivo `Painel.test.js` usando o mock. Agora teremos dois testes:

- O primeiro verifica a mensagem de saudação do sistema, que será igual a `Painel logado com +` `nome_do_usuario` , assim podemos ver se o usuário realmente foi logado e se o componente está pegando os dados dele;

- O segundo teste criará um mock do método `sair` e simulará a saída do sistema, já que o Avoriaz não tem integração com o Vue-router. Simular isso é importante para ver se nosso botão realmente está invocando o método que retira os dados do usuário do Vuex.

```
import test from 'ava'
import Comp from './../src/componentes/LvPainel.vue'
import mockvuex from './mockvuex'

let componente = mockvuex.vuex(Comp)

test('verificando nome no painel', t => {
    const span = componente.find('span')[0]
    t.is(span.text(), 'Painel logado com Leonardo')
})

test('saindo do sistema com mock, verificando se botão está lá',
t => {
    let acao = ''
    componente.vm.sair = function() {
        acao = 'saiu'
    }
    componente.update()
    componente.find('button')[0].trigger('click')
    t.is(acao, 'saiu')
})
```

Agora vamos criar o componente `LvPainel`, chamado pela rota `/notas`, após o usuário entrar no sistema. No `template`, por enquanto, teremos apenas um texto informando o nome do usuário logado e um botão para sair do sistema:

```
<span>Painel logado com {{ $store.state.usuario.nome }}</span>
<button @click="sair">Sair</button>
```

No `script`, vamos declarar o método `logar`, que volta os dados do usuário para nulo e então redireciona a aplicação para a tela de login:

```
export default{
    name: 'lv-painel',
    methods: {
        sair() {
            this.$store.dispatch('logarUsuario', { id: 0, nome: '
, email: '' })
            this.$router.replace('/')
        }
    }
}
```

Por fim, teremos de registrar as rotas para que, quando as acessarmos, apresentemos o componente `Login` ou `Painel`. Para isso, crie a rota `/` declarando que ela representará o componente `Login`, e a rota `/notas` que vai para o componente `Painel`. No arquivo `rotas.js`, adicione-as assim:

```
import LvLogin from './componentes/LvLogin.vue'
import LvPainel from './componentes/LvPainel.vue'
...
{path: '/',             component: LvLogin},
{path: '/notas',     component: LvPainel},
```

Ao rodar a *live*, teremos a rota `/` funcionando. Você pode tentar entrar com um usuário inválido para ver a resposta negativa, e então entrar com outro que realmente criou, para ser

redirecionado ao painel que apresentará o nome desse usuário:

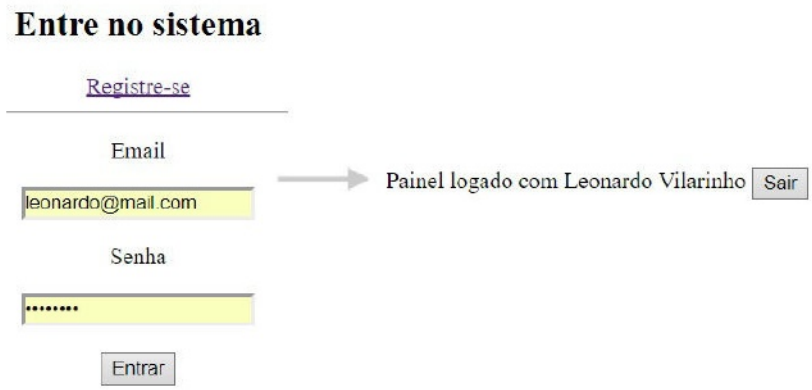

Figura 15.1: Tela de login, e após o login

Capítulo 16

CRIANDO UMA ANOTAÇÃO

Uma vez registrado e logado no sistema, o usuário poderá gerenciar suas anotações. A primeira coisa que devemos fazer é criar uma anotação, pois como gerenciar se não temos nenhuma nota?

Na API disponibilizada, existe uma rota chamada `notas` para realizar esse trabalho, então, no diretório `servicos`, é preciso criar um arquivo chamado `notas.js` para se comunicar com a API. Ele terá o mesmo molde do arquivo `usuario.js`, importando o arquivo de configuração do Axios e exportando um objeto com os métodos de manipulação do serviço.

Inicialmente, criaremos o método `criar`, que trata simplesmente de registrar uma anotação no banco de dados. Ele receberá: um objeto com um `conteudo`, que será o texto da nota; e o `id` do usuário logado, para fazer o vínculo dele com a anotação e retornar uma `Promise` para que o resultado da requisição seja tratado no componente que vamos criar.

```
import {http} from './configuracao'

export default {
    criar: ({ conteudo, usuario_id }) => {
```

```
        return http.post('notas', { conteudo, usuario_id });
    },
}
```

Antes de criar o componente para fazer uma nova anotação, conceberemos seu teste no arquivo `NovaNota.test.js`. Para criar uma anotação no banco de dados da API, precisamos do conteúdo da anotação e do `id` do usuário dono da nota, então temos de usar o mock do Vuex (criado no capítulo anterior) para pegar o `id` do usuário atual.

Faremos apenas um teste, igual aos testes anteriores, pois temos um formulário para criar a nota e um botão que contata a API sobre a nova anotação. No teste, iniciamos o componente com o conteúdo da anotação no `data`, e criamos um mock do método `criar` para que a API não seja contatada e crie a nota no banco. Esse mock vai mudar o estado do botão, então, após simular o clique nele, vemos se está desabilitado. Assim, simularemos todo o funcionamento desse componente, para ver se ele está totalmente funcional:

```
...

let componente = mockvuex.vuex(Comp)

test('formulário de criação com mock em ação', t => {
    componente.setData({
        conteudo: 'Ir ao mercado'
    })
    componente.vm.criar = function() {
        componente.setData({
            estado: 'Carregando...'
        })
    }
    componente.update()
    const botao = componente.find('input[type=submit]')[0]
    botao.trigger('click')
    t.is(botao.hasAttribute('disabled', 'disabled')        , true
```

```
)
})
```

O próximo passo é criar um componente para exibir o formulário de criação, chamar a API e coletar sua resposta. Para isso, o componente `LvNovaNota` é criado com o seguinte `template`:

```
<div>
    <input type="text" v-model="conteudo">

    <input @click='criar' type="submit" :value="estado" :disabled=
"estado != 'Criar'">
</div>
```

Nele, usa-se um botão que executa o método `criar` do componente. Já no `script` desse componente, é inserido o seguinte código:

```
import Nota from './../servicos/nota'
export default{
    name: 'lv-nova-nota',
    data() { return { estado: 'Criar', conteudo: '' } },
    methods: {
        criar() {
            if(this.conteudo != '') {
                this.estado = 'Carregando...'

                Nota.criar( this.campos ).then(resposta => {
                    if(resposta.data.sucesso) {
                        this.conteudo = ''
                        this.$bus.$emit('atualizacao')
                    }
                    this.estado = 'Criar'
                }).catch( e => this.estado = 'Criar' )
            }
        }
    },
    computed: {
        campos() {
            return {
                conteudo: this.conteudo,
```

```
                usuario_id: this.$store.state.usuario.id
            }
        },
    }
}
```

Além de informações básicas, como o nome do componente, a importação do serviço e a declaração das variáveis usadas nele, criamos a propriedade computada `campos`. Ela agrupa em um objeto o valor do campo de texto do formulário e o identificador do usuário armazenado na `Store` do Vuex.

No método `criar`, a validação de campo, feita nos outros componentes, é repetida, assim como o efeito no botão de ação. No núcleo do método, o método `criar` do serviço `Nota` é chamado, enviando os campos do formulário e recebendo uma `resposta`. Caso a resposta do servidor tenha sido positiva, o campo de texto é limpo e o botão volta ao normal, permitindo criar novas notas.

Note que, ali no método `criar`, usamos uma variável nova chamada `$bus`. Ela é um atalho que vamos criar para utilizar o `event bus`, que funciona como a comunicação entre componentes, porém uma comunicação global e não mais entre parentes.

No caso, estou emitindo um evento global chamado `atualizacao` para que qualquer outro componente do sistema possa ouvir e trabalhar no seu interior quando ele for disparado, ou seja, todos os componentes do sistema podem ouvir esse evento. Veremos como fazer isso mais tarde.

Para que essa variável funcione, no método `vuex` do `mockvuex` e no `main.js`, adicione a linha a seguir:

```
Vue.prototype.$bus = new Vue()
```

Para finalizar este capítulo e fazer de fato o formulário de cadastro aparecer após o usuário entrar no sistema, precisamos importar o componente criado no `LVPainel`, no capítulo anterior. Primeiro no `script`, é usado o `import` para incluir o arquivo, e o atributo `components` para usá-lo no componente atual:

```
import LvNovaNota from './LvNovaNota.vue'
export default{
    ...
    components: {
        LvNovaNota,
    },
    ...
}
```

Agora, no `template` do componente `LvPainel`, basta usar a tag do novo componente para fazê-lo aparecer na tela:

```
<template>
    ...
        <lv-nova-nota></lv-nova-nota>
    ...
</template>
```

Ao executar o `npm run dev`, o resultado será parecido com o da figura a seguir:

Figura 16.1: Tela inicial do sistema

Note que as notas ainda não são exibidas na tela, e este será o nosso próximo passo. Por enquanto, o indício de sucesso é que,

após criar a anotação, o campo de texto deve ser apagado.

Capítulo 17

LISTANDO AS NOTAS CRIADAS

No capítulo anterior, criamos nosso formulário para registrar uma anotação, mas ainda não podemos ver as anotações que temos cadastradas no sistema. Então, que tal exibir uma lista com elas?

Como sempre, temos de iniciar criando o método para se comunicar com a API, e isso sempre é feito no nosso arquivo de serviço. No arquivo `servicos/notas.js`, adicione o método `listar`, que enviará uma requisição `GET` para a API, e retornará uma `Promise` com o resultado da requisição:

```
...
export default {
    ...
    listar: (usuario_id) => {
        return http.get('notas?usuario_id=' + usuario_id);
    },
}
```

Note que recebemos o parâmetro `usuario_id`, pois queremos listar apenas as notas do usuário logado, então é preciso que seu identificador seja indicado.

Para exibir a lista, vamos criar os componentes `LvListaNotas` e `LvItemNota`. O primeiro representa a lista com

todas as notas e um método para atualizar toda a lista. Já o segundo representa um item daquela lista, ou seja, uma anotação que futuramente terá um botão para apagar a nota.

Vamos começar pelo item da lista. No teste `ItemNota.test.js`, apenas testamos criando o componente com uma propriedade padrão, assim nos certificamos de que o conteúdo da propriedade realmente foi parar no `template`:

```
import test from 'ava'
import Vue from 'vue'
import { mount } from 'avoriaz'
import Comp from './../src/componentes/LvItemNota.vue'

Vue.prototype.$bus = new Vue()

let componente = mount(Comp)

test('iniciando item da lista', t => {
    componente.setProps({
        nota: {conteudo: 'academia', id: 4}
    })
    const titulo = componente.find('span')[0]
    t.is(titulo.text(), 'academia')
})
```

Note que usamos o `event bus` aqui também, pois a lista de notas terá de escutar a emissão dada no elemento de nova nota.

Agora vamos criar o componente `LvItemNota` para que o teste criado neste capítulo fique verde. Esse componente será responsável por ser um item da lista de notas, e representará uma anotação criada pelo usuário. No `template`, teremos apenas um item de lista, com o conteúdo igual ao conteúdo da propriedade `nota`:

```
<li>
    <span>{{ nota.conteudo }}</span>
```

```
</li>
```

Já no `script`, serão criados apenas o nome do componente e a propriedade `nota`:

```
export default{
    name: 'lv-item-nota',
    props: {
        nota: {
            type: String,
            required: true
        }
    },
}
```

A propriedade `nota` é requerida, pois vamos receber a nota a ser exibida do componente da lista, no qual trabalharemos agora. O `ListaNotas.test.js` terá dois testes, e estes simularão os dois cenários que podemos ter no sistema:

- O primeiro seria quando o usuário ainda não cadastrou nenhuma anotação, então a lista teria 0 elementos;
- O segundo, quando o usuário já tem algumas anotações (no caso, duas), então o comprimento da lista será igual a 2.

Com esses dois casos, já poderemos ver se o componente funcionará em todo cenário possível:

```
...
let componente = mockvuex.vuex(Comp)

test('lista com 0 elementos', t => {
    const lista = componente.find('ul')[0]
    t.is(lista.find('li').length, 0)
})
```

```
test('lista com 2 elementos', t => {
    componente.setData({
        notas: [
            {conteudo: 'nota 1', id: 1},
            {conteudo: 'nota 2', id: 2},
        ]
    })
    const lista = componente.find('ul')[0]
    t.is(lista.find('li').length, 2)
})
```

No `template` do `LvListaNotas`, iniciamos a lista de notas usando a diretiva `v-for`, para percorrer o array de anotações recebidas da API. Para cada `nota`, um componente `LvItemNota` é criado e a `nota` é enviada para ele:

```
<ul style="text-align: left">
    <lv-item-nota v-for="(nota, ch) in notas" :key="ch" :nota="no
ta">
    </lv-item-nota>
</ul>
```

No `script`, temos muito trabalho a fazer. Além da importação do serviço `Nota` e do componente `LvItemNota`, temos a declaração do array de `notas`, a propriedade computada `campos` (assim como nos outros capítulos, ela recupera o identificador do usuário logado) e, por fim, o método `atualizar`. Este chama a requisição `listar` do serviço e, caso obtenha sucesso, pega a lista de notas do servidor e armazena-a no componente.

```
import Nota from './../servicos/nota'
import LvItemNota from './LvItemNota.vue'
export default{
    name: 'lv-lista-nota',
    components: {LvItemNota},
    data() { return { notas: {} } },
    mounted() {
        this.$bus.$on('atualizacao', this.atualizar)
```

```
        this.atualizar()
    },
    methods: {
        atualizar() {
            Nota.listar( this.campos ).then(resposta => {
                if(resposta.data.sucesso)
                    this.notas = resposta.data.notas
            }).catch( e => {} )
        }
    },
    computed: {
        campos() {
            return this.$store.state.usuario.id
        },
    }
}
```

Note que usamos o `event bus` aqui, pois o seu método `$on` é responsável por capturar os eventos globais emitidos. No caso, registramos que, sempre que uma `atualizacao` for emitida, o método `atualizar` será chamado.

Por fim, é preciso importar o componente de lista no nosso `LvPainel` para exibi-lo na tela inicial do sistema. Para isso, no componente `LvPainel`, importamos o arquivo `LvListaNotas.vue` e o registramos no atributo `components`:

```
...
import LvListaNotas from './LvListaNotas.vue'
export default{
    ...
    components: {
        ...
        LvListaNotas
    },
    ...
}
```

Após isso, o sistema já terá 90% do seu funcionamento, já que, além do registro e do login de usuário, podemos criar e ver as

anotações, como na figura a seguir:

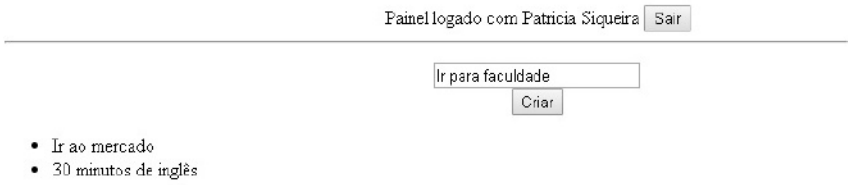

Figura 17.1: Tela inicial com lista de notas

O próximo passo é apagar nossas notas, o que será feito no próximo capítulo!

Capítulo 18

APAGANDO UMA ANOTAÇÃO

Chegamos ao último passo do sistema. Vamos fazer a remoção de anotações, assim o usuário pode deixar sua lista de tarefas sempre menor, sem ter aquela imensa lista inútil e bagunçada de notas.

Mais uma vez, a primeira coisa a se fazer é criar um método que realiza a requisição na API. No arquivo `servicos/notas.js`, criaremos o método `apagar`, que envia uma requisição do tipo `DELETE` para o servidor e retorna uma `Promise` para ser tratada no componente:

```
...
export default {
    ...
    apagar: (nota_id) => {
        return http.post('notas/' + nota_id);
    }
}
```

Note que existe o parâmetro `nota_id` que enviará o identificador da nota que queremos apagar, evitando que o servidor se confunda nessa tarefa.

Agora basta trabalhar no componente criado no capítulo

anterior. No `ItemNota.test.js`, adicionamos mais um teste para simular e testar o comportamento da aplicação ao apagar uma nota da lista. Nesse teste, vamos ter um mock no método `apagar`, que apagará a anotação. Depois, vamos fazer o mesmo que fizemos para testar o logout, definindo uma variável local `apagado` como `false` e verificando se, após apertar o botão para apagar o item, o seu valor foi mudado para `true`:

```
test('apagando o item da lista', t => {
    let apagado = false
    componente.vm.apagar = function() {
        apagado = true
    }
    componente.find('button')[0].trigger('click')
    t.is(apagado, true)
})
```

No `LvItemNotas`, precisamos ter um gatilho para acionar a remoção de uma anotação e, para isso, no `template` anterior, vamos adicionar um botão que executará o método `apagar`.

```
<template>
    <li>
        <button @click="apagar()">Apagar</button>
        <span>{{ nota.conteudo }}</span>
    </li>
</template>
```

Já no `script`, apenas importamos o serviço de `Nota` e criamos o método `apagar`:

```
import Nota from './../servicos/nota'
export default{
    name: 'lv-item-nota',
    props: ['nota'],
    methods: {
        apagar() {
            if(confirm('Deseja realmente apagar a anotação?'))
                Nota.apagar( this.nota.id )
                    .then(r => { this.$bus.$emit('atualizacao') }
```

```
)
                .catch(e => { alert('Erro ao apagar') })
        }
    }
}
```

Antes de apagar, é intuitivo confirmar se o usuário realmente pretende apagar aquela anotação, então usamos o método `confirm` para exibir um alerta na tela. Ao confirmar a exclusão, chama-se a requisição criada anteriormente.

Note que independentemente do resultado da requisição, nada é feito, pois o componente `LvListaNotas` trabalha em tempo real, e já trará um resultado da remoção para o usuário. Agora basta executar o `npm run dev` e ver tudo funcionando:

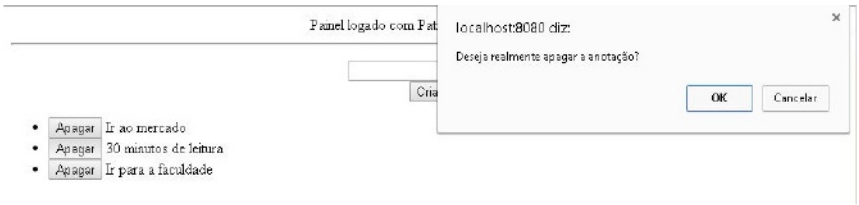

Figura 18.1: Tela inicial npm com a remoção de tarefa

Todo o projeto montado na Parte II pode ser encontrado em: https://github.com/leonardovilarinho/vue-notes.

E a API do projeto se encontra em (não é parte do livro, mas complementar): https://github.com/leonardovilarinho/api-vue-notes.

Capítulo 19

CONCLUINDO

Parabéns! Você acaba de finalizar a leitura do nosso livro. Espero que esta obra seja essencial para sua carreira no mundo do Vue.js. Com o aprendido aqui, você está apto a construir sistemas web dinâmicos, impressionando seus clientes com uma ótima usabilidade e fluidez.

Mas você deve saber que todo conhecimento ainda é pouco. Nunca saberemos tudo, porém cada conhecimento ganho é algo que conquistamos. Por isso, para que amplie ainda mais sua sabedoria, a seguir é exposta uma lista de links para desmistificar ainda mais o Vue.js:

- **Lista 'awesome'** (http://github.com/vuejs/awesome-vue): uma lista completa de exemplos, projetos que usam Vue.js, plugins e bibliotecas. Dê uma olhada nos plugins `vue-acl` e `vue-multilangue` (marketing caseiro).
- **Grupo no Slack** (http://vuejs-brasil.herokuapp.com): grupo Slack oficial da comunidade Vue no Brasil. Nele você poderá tirar dúvidas e saber tudo sobre a biblioteca.
- **Grupo no Telegram** (http://t.me/vuejsbrasil): grupo

Telegram oficial da comunidade Vue no Brasil, onde você poderá tirar dúvidas e saber tudo sobre a biblioteca.

- **Blog Vue Brasil** (http://vuejs-brasil.com.br): blog da comunidade brasileira sobre Vue, onde são encontrados diversos artigos da biblioteca.
- **Site oficial** (http://vuejs.org): no site oficial da biblioteca, você encontra o guia inicial, novidades das novas versões e toda a referência da API.
- **Guia de estilo de componente** (http://pablohpsilva.github.io/vuejs-component-style-guide): documento que serve como um guia para deixar seus componentes com estilos padronizados, com padrões sugeridos pela comunidade oficial.
- **Mapa mental** (http://mindmeister.com/pt/842448234/vue-js): mapa interativo para que você estude toda a API fornecida pelo Vue, ganhando mais conhecimento do que ela lhe oferece.

Explore e use principalmente o primeiro link da lista. Com ele, você se abrirá para um mundo de extensões e poderes. Crie alguns sistemas usando as bibliotecas que mais lhe chamaram atenção, seja por efeitos gráficos ou funcionalidades.

Caso tenha alguma dúvida ou feedback sobre o livro, use o fórum da Casa do Código para me contatar: http://forum.casadocodigo.com.br.