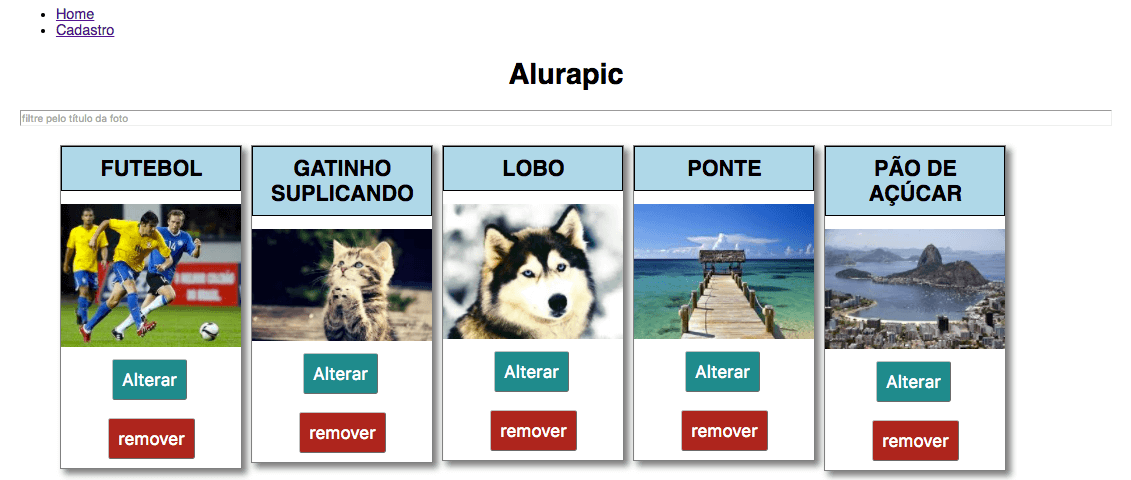
Curso 1

Bem-vindo, meu nome é Flávio Almeida (twitter @flaviohalmeida) e muito provavelmente já nos encontramos em outros treinamentos aqui do Alura.

**Conhecendo um pouco da nossa aplicação**

Tenho aqui o projeto deste curso completo. Este projeto se chama **Alurapic**, um sistema de gerenciamento de imagens, permitindo que o usuário cadastre fotos e busque por aquelas que seguem determinado critério.



Tenho certeza que se eu lhe mostrasse esta aplicação fora do contexto desse treinamento, você ficaria pensando qual tecnologia foi utilizada na sua construção, mas como estamos em um treinamento de Vue.js, você já sabe a resposta.

**Apresentando o Vue.js**

Vue.js é um framework para criação de componentes reutilizáveis, inclusive para criação de Single Page Applications, aquele tipo de aplicação que nunca recarrega durante seu uso. Um dos pontos de destaque desse framework é seu core minimalista de aproximadamente 17kb e a utilização de DOM virtual para realizar transformações no DOM, sem falar que o desenvolvedor não precisa se preocupar em aplicar técnicas para otimizar sua aplicação e, quando houver necessidade, ela será mínima.

Digamos que o Vue.js incorpora muitos dos conceitos do Angular 1.X da Google ao mesmo tempo que adota a estratégia de virtual DOM do React do Facebook tentando acompanhar a facilidade de uso do primeiro e a performance do segundo.

**Exercício obrigatório**

Agora que você já tem uma visão geral do nosso projeto, faça o [primeiro exercício obrigatório deste capítulo](https://cursos.alura.com.br/course/vue/task/23201). Lá você terá instruções de como configurar a infraestrutura mínima em seu computador pessoal para poder realizar este curso, inclusive orientações de qual editor utilizar.

Espero que sua experiência seja tão excepcional quanto a minha durante o tempo de elaboração deste treinamento. Um abraço do Flávio Almeida e bom estudo.

**Olá! Este curso esta dividido em dois módulos para melhorar sua experiência. O primeiro é este que você esta assistindo. Quando terminar todo o curso, você poderá cursar o segundo módulo com assuntos que não foram abordados aqui, completando assim sua jornada em Vue.js. Um abraço do Flávio Almeida! (twitter @flaviohalmeida).**

Para poder desenvolver de maneira profissional utilizando Vue, você precisa da plataforma Node.js instalada. O [Node.js](https://nodejs.org/) é um ambiente JavaScript multiplataforma disponível para Linux, Mac e Windows. Para instalá-lo, siga as instruções abaixo referentes a sua plataforma:

**Versão do Node.js**

É necessário no mínimo o Node.js em sua versão 6.X. Se você não é um desenvolvedor experiente com solução de problemas, evite instalar versões ímpares do Node.js, pois não são LTS ([long term support](https://pt.wikipedia.org/wiki/Suporte_de_longo_prazo))!

**Caso você já tenha o Node instalado**

Se você já tem o Node instalado em sua máquina, verifique se ele está pelo menos na versão **6.X**através do comando node -v no seu terminal, pois precisamos de uma versão atualizada do mesmo para que não ocorra erros na sua máquina durante o projeto.

**Instalação Node.js no Linux (Ubuntu)**

No Ubuntu, através do terminal (permissão de administrador necessária) execute o comando abaixo:

sudo apt-get install -y nodejs

**ATENÇÃO:** em algumas distribuições Linux, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Neste caso específico, no lugar do binário ser node, ele passa a se chamar **nodejs**. Isso gera problemas, pois a instrução **npm start** não funcionará, pois ela procura o binário node e não nodejs. Para resolver no Ubuntu

sudo ln -s /usr/bin/nodejs /usr/bin/node depois o comando npm start funcionará conforme esperado.

É uma pena haver essa discrepância, mas fica aqui essa dica!

**Instalação Node.js no Windows**

Baixe o instalador clicando no grande botão install, diretamente da página do [Node.js](https://nodejs.org/). Durante a instalação, você apenas clicará nos botões para continuar o assistente. Não troque a pasta padrão do Node.js durante a instalação, a não ser que você saiba exatamente o que está fazendo.

**Instalação Node.js no MAC**

O [homebrew](http://brew.sh/) é a maneira mais recomendada para instalar o Node.js em sua máquina, através do comando:

brew update

brew install node

Não usa homebrew? Sem problema, baixe o instalador clicando no grande botão install, diretamente da página do [Node.js](http://brew.sh/).

**Editor recomendado: Visual Studio Code (Gratuito)**

Recomendo extremamente o uso do Visual Studio Code (VSCode) um editor gratuito criado pela Microsoft disponível claro, para Windows, Linux e MAC. É esse que utilizarei ao longo do treinamento.

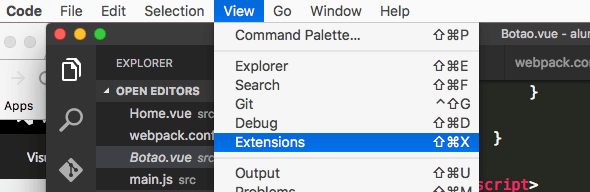
Você pode baixá-lo no endereço <https://code.visualstudio.com/download>.

Se você utiliza IDE's como Eclipse ou NetBeans, sugiro abandonar essas IDE's pois você encontrará problemas desde acentuação até mensagens de erros falso positivo. Caso você insista no uso delas, faça primeiro o curso usando o editor indicado e depois utilize sua IDE. Isso evitará que você poste no fórum dúvidas a respeito desses editores que não aconselho utilizar na parte de front-end.

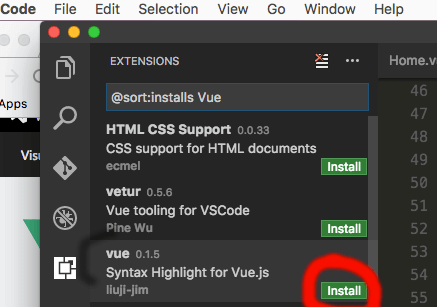
Caso você não queria usar o Visual Studio, pode utilizar outros editores como Atom ou Sublime3. Contudo, sugiro fortemente o Visual Studio Code para que você tenha uma paridade visual comigo ao longo do curso.

**Sintaxe colorida (sintaxe highlight) com Visual Studio Code e Vue**

Caso seu VSCode não esteja com marcação de cores, recomenda-se instalar para melhor visualização de arquivos .vue. Com o VSCODE aberto vá até o menu **views -> extension**:



Isso fará com que o campo de busca de extensões seja exibido. Digite **vue** e escolha a opção **vue** do Liujim-jim como demonstrado na imagem abaixo:



Depois de instalar, basta reiniciar o VSCODE para que a extensão seja carregada.

Vamos dar início ao nosso projeto. No entanto, "dar início" a um projeto pode ser algo um tanto demorado e complicado. Precisamos baixar os scripts que serão usados, decidir a estrutura de pastas e até configurar um servidor web. Imagine fazer todo esse processo antes de escrevermos sequer uma linha de código da nossa aplicação!

## Automatizando a construção do projeto com Vue CLI

A boa notícia é que o Vue possui um CLI (command line interface) que nada mais é do que uma ferramenta que automatiza a construção da infraestrutura do projeto. Mas de onde baixaremos o CLI do Vue? Acessaremos algum site? Com certeza não, aliás, baixar dependências de sites é uma prática cada vez mais extinta no mundo front-end. Utilizaremos o gerenciador de pacotes do Node.js, o **npm**, para baixar o Vue CLI, inclusive todas as dependências da aplicação.

Agora você deve estar entendendo o motivo pelo qual o Node.js é pré-requisito obrigatório para desenvolver em Vue, no entanto essa obrigatoriedade existe apenas em ambiente de desenvolvimento. Quando terminamos uma aplicação, o uso do Node.js não é necessário, pois cabe ao desenvolvedor escolher qual servidor utilizar para hospedá-la, seja ele um servidor Php, Java, .Net entre outros. E claro, nada o impede de querer usar o Node.js como servidor da sua aplicação.

## Considerações sobre terminal/prompt de comando

Para utilizarmos o Vue CLI, é necessário um mínimo de traquejo no terminal (Linux/Mac) ou no prompt de comando (Windows) do seu sistema operacional. Não é exclusividade do Vue CLI ser utilizado através do terminal, e um conhecimento básico o ajudará bastante ao longo deste curso e dos demais da Alura que dependem do terminal. Bom, vamos começar os trabalhos.

## Instalando o CLI do Vue

Através do terminal e com o Node.js devidamente instalado, vamos instalar o Vue CLI através de um comando do npm. Mas atenção, é necessário ter privilégio de administrador para que o comando funcione corretamente. Dependendo de como seu sistema operacional esta configurado, talvez não seja necessário. Mas se houver algum problema durante a instalação você já sabe qual a possível causa.

No terminal (ou prompt de comando, Se você usa Windows), vamos executar o seguinte comando:

npm install -g vue-cli@2.7.0

Estamos instalando Vue CLI globalmente através do parâmetro -g para que possamos acessar o CLI de qualquer pasta através do terminal. Dentro de instantes tudo será baixado. Assim que a instalação terminar, vamos chamar o CLI e solicitar que seja impresso no console sua versão para sabermos se a instalação foi realizada com sucesso:

vue --version

Como instalamos a versão 2.7.0, será essa versão impressa no console. Aliás, peço que usem a mesma versão que estou usando neste treinamento. Ela já foi homologada por mim. É comum o aluno querer atualizar para a versão mais nova, um desejo justo. No entanto, novas versões podem causar bugs que podem atrapalhar o processo de aprendizagem do aluno. Sendo assim, depois que concluir o projeto e tudo estiver funcionando, se quiser atualizar para a versão mais nova e um erro acontecer, você saberá que o erro é da versão mais nova (bug, incompatibilidade) e não do seu código.

Excelente, agora que cliente de linha de comando esta instalado, podemos gerar nosso projeto.

## Novo projeto a partir de um template

Através do terminal, vou até a minha área de trabalho (Desktop) para em seguida e gerar o projeto alurapic através do comando:

vue init webpack-simple alurapic

Algumas perguntas serão feitas (nome do projeto, autor, versão) e podemos teclar ENTER para todas elas tranquilamente para adotarmos valores padrões. O exemplo acima usa como template o webpack-simple. Há outros templates mais simples e mais sofisticados, no entanto, este template é mais do que suficiente para o escopo da nossa aplicação.

O resultado do comando criará a pasta alurapic, no entanto, temos apenas a estrutura do projeto e uma lista de todas as suas dependências. Essas dependências não são baixadas automaticamente na construção do projeto. Precisamos entrar na pasta alurapic ainda no terminal e executarmos o comando:

npm install

Este comando baixará todas as dependências listadas no arquivo alurapic/package.json. Entenda esse arquivo como um catálogo de todos os recursos que nosso projeto precisa para funcionar. Aliás, como você já deve ter inferido, quem criou esse package.json automaticamente para nós foi o CLI.

## Levantando um servidor e acessando a aplicação

Quando todas as dependências forem baixadas, já podemos subir nossa aplicação através do comando:

npm run dev

O comando npm run dev executa um script criado em alurapic/package.json criado pelo próprio CLI. Há muita coisa envolvida nesse comando, mas o mais importante é saber por agora que ele levanta um servidor local servindo nosso projeto e abrirá automaticamente o navegador padrão do seu sistema operacional apontando para o endereço do projeto no servidor. Fantástico, não? Veremos uma página genérica criada pelo template que utilizamos ao criar nosso projeto.

Com o projeto criado, vamos verificar sua estrutura.

Dependências do Node.js são instaladas através do seu gerenciador chamado npm. Dependências podem ser instaladas localmente ou globalmente, esta última forma, através do parâmetro -g. Sendo assim, temos a instrução:

npm install -g vue-cli@2.7.0

O parâmetro -g pode vir no final, se assim desejarmos.

npm install vue-cli@2.7.0 -g

Dependências instaladas globalmente são acessíveis de qualquer pasta através do terminal do sistema operacional em uso.

Quando instalamos o Vue CLI globalmente podemos invocar o binário vue para criar a estrutura do projeto para nós. Por exemplo:

vue init webpack-simple calopsita

Neste caso o parâmetro init indica que estamos criando um novo projeto. No entanto, precisamos informar qual template usaremos em sua construtor, por isso usamos webpack-simple. Por fim, o último parâmetro é o nome do projeto, ou seja, a pasta na qual todos os arquivos criados pelo Vue CLI ficarão.

Sem precisarmos fazer nada, o comando npm run dev levantou um servidor http totalmente já configurado para que possamos acessar nossa aplicação, e mais, automaticamente abriu o navegador padrão do sistema operacional que carregou o arquivo alurapic/index.html. Mordomia assim só na casa da minha mãe!

Falando em mordomia, veremos que esse servidor voltado para o ambiente de desenvolvimento faz muito mais do que imaginamos, mas primeiro vamos entender a estrutura do projeto que foi criada, começando pelo conteúdo do arquivo alurapic/index.html:

<!-- alurapic/index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>alurapic</title>

</head>

<body>

<div id="app"></div>

<script src="/dist/build.js"></script>

</body>

</html>

Espere um pouco! Essa página apenas importa um script e possui uma div vazia. Como é possível que ela seja a página que estamos visualizando no navegador? E se eu disser para vocês que o conteúdo exibido no navegador, inclusive todos os arquivos da pasta alurapic/src foram transformados no script /dist/build.js? Quer dizer que nossa página se transformou em um script? Sim!

## Babel, Webpack e geração do bundle

Com a revelação que acabei de declarar, acredito que vocês estejam ansiosos para ver o conteúdo de /dist/build.js, mas sinto informa-los que esse arquivo não existe fisicamente, ainda. O conteúdo da pasta alurapic/src foi transformando em memória no arquivo build.js, por isso o arquivo não existe. Isso se dá assim para acelerar o tempo de desenvolvimento permitindo que o desenvolvedor veja o quanto antes o resultado de suas alterações no projeto. Aprendemos a gerar esse arquivo ainda neste curso, para que possamos distribuir nossa aplicação.

Voltando ao nosso arquivo build.js, eu disse que ele é o resultado da transformação dos arquivos da pasta alurapic/src, mas quem realiza essa transformação? Qual sua finalidade? Primeiramente, ocorrem duas transformações cada uma com ferramentas diferentes.

A primeira transforma através do processo de transcompilação todo o código escrito usando ES2015 para ES5 garantindo maior compatibilidade da nossa aplicação, mais notadamente em browsers desatualizados de smartphones. A segunda se encarrega de gerar um bundle para que seja carregado pelo navegador.

Para realizar as transformações anteriores que acabei de citar, o Vue CLI utiliza respectivamente [Babel](https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-3/task/20294) e [WebPack](https://webpack.github.io/). Por fim, vale ressaltar que Webpack vai mais além do que simplesmente criar um bundle, mas para início de conversa o que sabemos é suficiente para podermos continuar.

Agora que você já entendeu o motivo e como o build.js é gerado, vamos deixá-lo de lado e focar a pasta alurapic/srcpara entender o papel de cada arquivo que fará parte do bundle. Aliás, qual deles equivale ao conteúdo que é exibido em nosso navegador?

## Componentes declarados em Single File Templates

O arquivo que corresponde à página que estamos vendo no navegador é o alurapic/src/App.vue:

<!-- src/App.vue -->

<template>

<div id="app">

<img src="./assets/logo.png">

<h1>{{ msg }}</h1>

<h2>Essential Links</h2>

<ul>

<li><a href="https://vuejs.org" target="\_blank">Core Docs</a></li>

<li><a href="https://forum.vuejs.org" target="\_blank">Forum</a></li>

<li><a href="https://gitter.im/vuejs/vue" target="\_blank">Gitter Chat</a></li>

<li><a href="https://twitter.com/vuejs" target="\_blank">Twitter</a></li>

</ul>

<h2>Ecosystem</h2>

<ul>

<li><a href="http://router.vuejs.org/" target="\_blank">vue-router</a></li>

<li><a href="http://vuex.vuejs.org/" target="\_blank">vuex</a></li>

<li><a href="http://vue-loader.vuejs.org/" target="\_blank">vue-loader</a></li>

<li><a href="https://github.com/vuejs/awesome-vue" target="\_blank">awesome-vue</a></li>

</ul>

</div>

</template>

<script>

export default {

name: 'app',

data () {

return {

msg: 'Welcome to Your Vue.js App'

}

}

}

</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;

}

h1, h2 {

font-weight: normal;

}

ul {

list-style-type: none;

padding: 0;

}

li {

display: inline-block;

margin: 0 10px;

}

a {

color: #42b983;

}

</style>

No entanto, ele não é uma página, mas um **Single file template** (template de único arquivo) que equivale a um **módulo**que declara um **componente**. Muito coisa para um arquivo só, não? O que precisamos entender aqui é que o arquivo sendo um módulo, se quisermos usar o componente que ele declara precisamos importá-lo em outros módulos da aplicação que queira utilizá-lo.

Pense em um módulo como uma caixa preta que pode ter diversas funcionalidades e só aquelas que forem explicitamente exportadas podem ser utilizadas em outros módulos. Tanto isso é verdade que dentro da tag <script> de App.vue há a instrução export default permitindo que nosso componente seja importado por outros módulos. Aliás, falando em componente, que nada mais são do que um objeto que possui sua apresentação, dado e comportamento. É por isso que nosso componente é definido através de três grandes blocos: template (apresentação), script (comportamento e dados) e style (o estilo da apresentação)

Sabemos que esse arquivo será transformado automaticamente para algo que seja compreendido pelo navegador, mas como é feita a ligação desse componente com index.html, uma vez que o componente é exibido assim que nossa página é carregada. É isso que veremos.

Em index.html, quando nossa aplicação é carregada, a tag <div id="app"></div> é substituída pelo componente Appque acabamos de escrutinar, mas isso não ocorre por padrão. Alguém precisa explicitar isso na inicialização da nossa aplicação, a tarefa realizada por alurapic/main.js. Vamos verificar esse arquivo.

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

new Vue({

el: '#app',

render: h => h(App)

})

O arquivo alurapic/src/main.js, assim como App.vue é um módulo do ES2015:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

// código posterior omitido

Sendo um módulo do ES2015, para termos acesso a outros artefatos de outros módulos da nossa aplicação, precisamos explicitar qual artefato de qual módulo desejamos importar. Como este é o arquivo responsável em exibir nosso componente App precisamos importá-lo, o que é feito na segunda instrução do arquivo. Como só temos um componente em App.vue e a sintaxe export default foi utilizada, usamos como nome do artefato a ser importado o mesmo nome do arquivo, sem a extensão .vue.

No entanto, antes de importar App, o módulo alurapic/main.js importa Vue (maiúsculo) do módulo vue. O módulo vue esta localizado dentro da pasta alurapic/node\_modules e o CLI já tem tudo configurado para que a pasta do módulo seja enxergada pela instrução import. O artefato Vue é o **Global View Object**, um objeto especial do Vue.js, seu core.

## View Instance

Para que possamos carregar, ou melhor, renderizar nosso componente App em <div id="app"></div> de index.html, precisamos criar uma **view instance** com auxílio do global view object que importamos:

// alurapic/src/main.js

// código anterior omitido

// criando uma view instance

new Vue({

/\* código omitido \*/

})

Uma instância de view é uma ponte entre nossos componentes e view. Em nosso caso, nossa view é o index.html. Mas a view instance não fez "curso mãe Diná" para saber qual componente deve carregar e o local onde deve inseri-lo em index.html, sendo necessário passar essa informação para a ela.

## Renderizando um template

Vejamos nossa view instance devidamente configurada:

// alurapic/src/main.js

// código anterior omitido

new Vue({

el: '#app',

render: h => h(App)

})

A propriedade el recebe como parâmetro o seletor do elemento que será substituído pelo nosso componente, já na função render passamos o componente que desejamos renderizar.

Agora que já entendemos como as coisas se encaixam em nossa aplicação, vamos voltar para nosso componente Apppara compreender um conceito importante logo neste primeiro capítulo.

Voltando para a declaração do nosso componente App, na parte que declara seu template, ou seja, sua apresentação temos a seguinte tag h1:

<!-- alurapic/src/App.vue -->

<!-- código anterior omitido -->

<h1>{{ msg }}</h1>

<!-- código posterior omitido -->

Se fizermos um "de, para" deste trecho do nosso template com o que visualizamos no browser, em seu lugar é exibido o texto "Welcome to Your Vue.js App". Como isso é possível? Para que possamos compreender essa mágica, vou apelar para uma analogia com o trabalho de um advogado.

Um advogado quando cria um memorando ou coisa parecida ele não faz do zero, ele usa um template do documento que deseja redigir personalizando apenas algumas lacunas desse documento. Não é à toa que o nome da nossa tag se chama template. Nela, lacunas são expressas através de {{ }} e a informação entre as chaves duplas é a informação que o template necessita para ficar completo. No caso, precisamos da informação msg, ou melhor, do dado msg. No caso de App, é o próprio componente que disponibiliza o dado de que seu template precisa através da função data lá dentro da tag <script>. Vejamos:

// alurapic/src/App.vue

// código anterior omitido

data () {

return {

msg: 'Welcome to Your Vue.js App'

}

}

// código posterior omitido

Em App.vue, disponibilizamos dados para o template através da função data. Essa função sempre retorna um objeto e as propriedades desse objeto são acessíveis no template do componente usando a sintaxe {{ }}.

## Interpolação e data binding

O resultado da sintaxe especial {{ }} com o nome da propriedade que desejamos ler é chamada de **interpolação**. Dizemos que o dado msg foi interpolado no template. Por fim, essa interpolação segue uma regra: os dados fluem sempre da sua origem para o template e nunca o caminho contrário.

Tecnicamente falando, o que a interpolação faz é uma associação de dados unidirecional chamada data binding. Aliás, uma característica dessa associação é que qualquer mudança no dado gera automaticamente uma atualização no template do componente. Como o componente é exibido dentro de uma view, no caso index.html, podemos dizer que mudanças nos dados acarretam uma atualização da view.

Podemos fazer um teste alterando o valor da propriedade msg por outro qualquer: primeiro módulo caso você queria se familiarizar ainda mais com a sintaxe apresentada com logo do curso.

<!-- alurapic/src/App.vue -->

<!-- código anterior omitido -->

<script>

export default {

name: 'app',

data () {

return {

msg: 'Seja bem-vindo à sua Vue.js App'

}

}

}

</script>

<!-- código posterior omitido -->

Veja que ao alterarmos o valor de msg do objeto retornado pela função data do nosso componente, mudamos o que é apresentado para o usuário. Mas espere um pouco, essa mudança entrou em vigor sem precisarmos recarregar nossa página. Como isso é possível?

## Live reloading

Qualquer alteração que fizemos nossos arquivos do nosso projeto gerará um novo bundle em memória e fará com que o navegador recarregue automaticamente para refletir nossas últimas alterações. Isso só é possível porque o servidor criado pelo Vue CLI suporta LiveReloading.

Agora que já temos uma visão geral de como a aplicação é estruturada, inclusive alguns conceitos importantes do Vue.js, já podemos adequar nossa aplicação para ser tornar realmente a Alurapic!

Neste capítulo aprendemos:

* instalar e utilizar o Vue CLI
* criar do zero um novo projeto
* estrutura fundamental do projeto
* arquivos de template .vue
* estrutura de um template
* relação entre index.html, main.js e App.vue
* como renderização de um template é feita
* data binding unidirecional através de interpolação
* o mecanismo de live reloading

Agora que temos uma visão geral de como cada parte do projeto se encaixa, inclusive a aprendizagem do conceito de data binding através de interpolação, podemos começar a dar ao projeto a cara da aplicação que desejamos construir.

## Template e view root

Vamos apagar o conteúdo do template de alurapic/src/App.vue deixando apenas o esqueleto necessário para criarmos nossa primeiro componente:

<!-- alurapic/src/App.vue -->

<template>

</template>

<script>

export default {

}

</script>

<style>

</style>

Nosso componente App, o primeiro a ser exibido na view index.html exibirá uma lista de fotos. Nesse sentido, vamos adicionar uma tag img no template de alurapic/src/App.vue para exibir uma imagem qualquer da nossa escolha, inclusive você pode pesquisar essa imagem no próprio Google e utilizar seu endereço.

Alterando src/App.vue:

<!-- alurapic/src/App.vue -->

<template>

<img src="https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It" alt="Cachorro">

</template>

<script>

export default {

}

</script>

<style>

</style>

Pelo mecanismo de live reloading, assim que salvarmos o arquivo nosso navegador será carregado exibindo nossa imagem.

Faltou o título da página, vamos colocar uma tag h1:

<!-- alurapic/src/App.vue -->

<template>

<h1>Alurapic</h1>

<img src="https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It" alt="Cachorro">

</template>

<script>

export default {

}

</script>

<style>

</style>

Surpresa! Nada é exibido no navegador. Se olharmos o terminal temos a seguinte mensagem:

ERROR in ./~/vue-loader/lib/template-compiler.js?id=data-v-6a8232ee!./~/vue-loader/lib/selector.js?type=template&index=0!./src/App.vue

template syntax error Component template should contain exactly one root element:

<h1>Alurapic</h1>

<img src="https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It" alt="Cachorro">

If you are using v-if on multiple elements, use v-else-if to chain them instead.

@ ./src/App.vue 11:23-143

@ ./src/main.js

@ multi main

Extraindo dessa mensagem de erro a mensagem mais importante:

App.vue

template syntax error Component template should contain exactly one root element:

Isso acontece, porque se quisermos exibir mais de um elemento dentro da tag template, os elementos devem estar dentro de um elemento pai. Nesse caso, uma tag div resolve esse problema:

<!-- src/App.vue -->

<template>

<div>

<h1>Alurapic</h1>

<img src="https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It" alt="Cachorro">

</div>

</template>

<script>

export default {

}

</script>

<style>

</style>

Perfeito, conseguimos exibir uma imagem, mas se fosse para exibir a imagem assim como fizemos não precisaríamos do Vue. Queremos exibir a imagem dinamicamente baseada nos dados da foto que lhe forem passados. Mas antes de passarmos os dados para a a tag img, vamos fazer o seguinte. Forneceremos para o template de App.vue o título da página.

## Data binding unidirecional através de interpolação

Aprendemos que é a função data que deve retornar um objeto JavaScript com os dados de que o template da nossa view precisa:

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1>{{ titulo }}</h1>

<img src="https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It" alt="Cachorro">

</div>

</template>

<script>

export default {

data() {

return {

titulo: 'Alurapic'

}

}

}

</script>

<style>

</style>

Excelente, recarregando nossa página o título Alurapic continua sendo exibido. Se trocarmos por outro, automaticamente o navegador recarregará e usará a nova informação exibindo-a para o usuário. Vale ressaltar que interpolações com {{ }} realizam uma associação unidirecional da fonte de dados para o template sempre nesse sentido. Sendo assim, mudanças nos dados acarretam mudanças no template do nosso componente.

Agora, vamos retornar os dados da foto. No caso, vamos usar os mesmos que usamos antes:

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1>{{ titulo }}</h1>

<img src="" alt="">

</div>

</template>

<script>

export default {

data() {

return {

titulo: 'Alurapic',

foto: {

url: 'https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It',

titulo: 'Cachorro'

}

}

}

}

</script>

<style>

</style>

Veja que movemos o valor de src e de alt da tag img para a propriedade do objeto foto retornando também pela função data. E agora, como acessaremos essas informações no template. Será que podemos utilizar interpolação assim como fizemos com o titulo?

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1>{{ titulo }}</h1>

<img src="{{ foto.url }}" alt="{{ foto.titulo }}">

</div>

</template>

<script>

export default {

// código omitido

}

</script>

<style>

</style>

Nossa página esta em branco no navegador! Olhando no terminal temos a seguinte mensagem:

ERROR in ./~/vue-loader/lib/template-compiler.js?id=data-v-6a8232ee!./~/vue-loader/lib/selector.js?type=template&index=0!./src/App.vue

template syntax error src="{{ foto.url }}": Interpolation inside attributes has been removed. Use v-bind or the colon shorthand instead. For example, instead of <div id="{{ val }}">, use <div :id="val">.

@ ./src/App.vue 11:23-143

@ ./src/main.js

@ multi main

ERROR in ./~/vue-loader/lib/template-compiler.js?id=data-v-6a8232ee!./~/vue-loader/lib/selector.js?type=template&index=0!./src/App.vue

template syntax error alt="{{ foto.titulo }}": Interpolation inside attributes has been removed. Use v-bind or the colon shorthand instead. For example, instead of <div id="{{ val }}">, use <div :id="val">.

@ ./src/App.vue 11:23-143

@ ./src/main.js

@ multi main

A parte que nos interessa é:

template syntax error src="{{ foto.url }}": Interpolation inside attributes has been removed. Use v-bind or the colon shorthand instead. For example, instead of <div id="{{ val }}">, use <div :id="val">.

E agora?

## A diretiva v-bind

Não podemos usar interpolação em atributos! Precisamos fazer de outra maneira, aliás essa maneira possui duas formas. A primeira é usarmos v-bind:

<template>

<div>

<h1>{{ titulo }}</h1>

<img v-bind:src="foto.url" v-bind:alt="foto.titulo">

</div>

</template>

<script>

export default {

// código omitido

}

</script>

<style>

</style>

Assim que realizamos a alteração, nossa página volta a ser exibida, inclusive a tag img com os dados da foto. Usamos a sintaxe v-bind:nomeDoAtributo. O valor é atribuído diretamente, sem { {}} como v-bind:src="foto.url". É o Vue que fará a interpolação por debaixo dos panos.

O que acabamos de ver em ação é o uso de uma **diretiva** do Vue. Diretivas nada mais são do que um código interpretado pelo Vue que encapsula determinada funcionalidade ensinando novos truques para o navegador. Tanto isso é verdade que no mundo HTML não existe v-bind, logo, esta diretiva esta sendo interpretada pelo Vue.

## Um atalho elegante para v-bind

No entanto, pode parecer um tanto verbo usar a sintaxe v-bind para realizar uma associação unidirecional que vai da fonte de dados para a view. Nesse caso podemos trocar v-bind pelo seu atalho dois pontos:

<template>

<div>

<h1>{{ titulo }}</h1>

<img :src="foto.url" :alt="foto.titulo">

</div>

</template>

<script>

export default {

// código omitido

}

</script>

<style>

</style>

É uma sintaxe muito mais enxuta equivalente ao v-bind. Aliás, se não quisermos usar interpolação para o conteúdo de uma tag, podemos usar a diretiva v-text, em nosso exemplo ficaria <h1 v-text="titulo"></h1>. No entanto, eu usarei interpolação até o final do treinamento para fornecer o conteúdo de tags.

Estamos quase chegando ao final do capítulo, mas vem aquela pergunta: e se quisermos exibir mais de uma imagem, qual alteração será necessária em nosso código? Veja que a pergunta é totalmente válida pois App será a responsável pela exibição de todas as fotos da nossa aplicação.

Primeiro, vamos renomear a propriedade foto para foto1 em nossa função data. Em seguida, vamos adicionar a nova propriedade foto2 que aponta para outra imagem com outro nome. Com isso, vamos adicionar mais uma tag img no template e apontar para a nova propriedade usando o atalho para v-bind, o famoso dois pontos. Vamos também adotar a boa pratica de exibir uma lista não ordenada através da tag ul, onde cada imagem será o conteúdo de uma li:

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1>{{ titulo }}</h1>

<ul>

<li>

<img :src="foto1.url" :alt="foto1.titulo">

</li>

<li>

<img :src="foto2.url" :alt="foto2.titulo">

</li>

</ul>

</div>

</template>

<script>

export default {

data() {

return {

titulo: 'Alurapic',

foto1: {

url: 'https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It',

titulo: 'Cachorro'

},

foto2: {

url: 'https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It',

titulo: 'Cachorro2'

},

}

}

}

</script>

<style>

</style>

Excelente, temos duas imagens sendo exibidas. Mas se tivéssemos trinta? Teríamos as propriedades foto3, foto4 e assim por diante, sem falar que teríamos que adicionar em App uma tag img para cada foto. Isso não precisa ser assim. Para revolucionarmos nosso código, primeiro vamos seguir a boa prática de disponibilizar os dados da foto em um array. Com isso, deixaremos de ter várias propriedades, uma para cada foto no objeto retornado pela função data:

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1>{{ titulo }}</h1>

<ul>

<li>

<img :src="foto1.url" :alt="foto1.titulo">

</li>

<li>

<img :src="foto2.url" :alt="foto2.titulo">

</li>

</ul>

</div>

</template>

<script>

// agora temos apenas a propriedade `fotos` que é um array que possui dois objetos que possuem as propriedades `url` e `titulo`, cada um com seu valor.

export default {

data() {

return {

titulo: 'Alurapic',

fotos: [

{

url: 'https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It',

titulo: 'Cachorro'

},

{

url: 'https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTOhmlmzV4-Sifx5BIc2SXeA-1CtZJf8jb8V\_vPZyKbXIQJKU-rkxGO6OM',

titulo: 'Gato'

}

]

}

}

}

</script>

<style>

</style>

Excelente, e agora?

## A diretiva v-for

Mudamos apenas a fonte de dados, mas e o nosso template, como ficará? A ideia é a seguinte, para cada item do nosso array fotos queremos construir uma tag li coma imagem da foto que esta sendo iterada no momento. Queremos algo como um forEach em JavaScript. Para isso, o próprio Vue nos disponibiliza a diretiva v-for que permite realizar laço dentro do nosso template, vejamos seu uso:

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1>{{ titulo }}</h1>

<ul>

<li v-for="foto of fotos">

<img :src="foto.url" :alt="foto.titulo">

</li>

</ul>

</div>

</template>

<script>

export default {

// código omitido

}

</script>

<style>

</style>

Incrível! Para cada item do nosso array fotos uma tag li é criada. Na diretiva v-for indicamos qual lista estamos varrendo, em nosso caso, fotos, que faz parte do objeto retornado pela função data.

Mas como ter acesso a cada foto da lista? Precisamos ter acesso ao elemento para podermos passar os dados para a tag img. Para isso, damos um apelido para cada item da lista, em nosso caso, escolhemos o apelido foto. Como temos duas fotos em nossa lista, a diretiva v-for será repetida duas vezes e através de interpolação com {{ }} podemos ter acesso aos dados da foto, por exemplo, com {{ foto.titulo }} acessamos seu título.

Agora, imagine se no lugar de termos dados fixos esses dados viessem de um servidor, de um API na web? Exibiríamos instantaneamente todas as fotos de uma só vez com pouquíssimo esforço. Aliás, esse será o assunto do próximo capítulo.

A diretiva v-for pode ser usada com **of** ou **in**. A primeira é mais perto da sintaxe dos iterators do JavaScript. Ao longo do curso usarei os dois, mas por enquanto ficaremos com o **of**. Exemplos:

v-for="nome of nomes"

v-for="nome in nomes"

Neste capítulo aprendemos:

* data binding unidirecional em atributos através da diretiva v-bind
* atalho para v-bind
* a diretiva v-for

Nossa aplicação consegue exibir uma lista de fotos fixa, mas sabemos que em uma aplicação real os dados das fotos devem vir de um servidor, ou melhor, de uma API disponibilizada por um servidor. Uma API nada mais é do que um endereço único na Web que é capaz de executar uma série de ações e também receber e enviar dados.

Baixe o arquivo [api.zip](https://s3.amazonaws.com/caelum-online-public/vue/api.zip) que contém a API do curso feita em Node.js. Depois de baixar, descompacte o arquivo na sua área de trabalho. Em seguida, abra um **novo terminal** . Neste terminal, levante o servidor que disponibiliza a API utilizada pelo nosso projeto através do comando:

npm start

Se tudo correr bem, será exibida a seguinte mensagem no console:

API escutando na porta: 3000

Por fim, para verificar se a API esta retornando dados acesse o endereço http://localhost:3000/v1/fotos. Isso fará com que uma lista de fotos seja exibida em seu navegador.

Quando usamos o endereço anterior, o navegador automaticamente utiliza o método GET. Nesse sentido, nosso API esta preparada para quando o endereço for este e o método for GET responder com uma lista de fotos. Veremos ao longo do curso que nossa API suporta outros métodos para cadastrar novos fotos, inclusive para apagá-las.

Veja que até o final do curso trabalharemos com dois terminais. O primeiro, que torna acessível nossa aplicação para o navegador e o outro que disponibiliza uma API para ser consumida por essa aplicação.

## Um pouco sobre a separação do cliente e sua API

Quando criamos uma SPA, é prática comum do mercado hospedar o front-end em um servidor em separado da API consumida por esse servidor. Aliás, uma mesma aplicação SPA criada com Vue pode consumir uma ou mais API's criadas nas mais diversas linguagens, sejam elas Python, Java, C# entre outras. O segredo dessa compatibilidade é que essas API's devem seguir o padrão REST, um padrão fortemente baseado no protocolo HTTP, por isso universal.

Essa separação entre front e da API permite que a mesma API seja consumida por outros tipos de clientes que não sejam um navegador como aplicativos feitos em Android ou até mesmo uma engenhoca construída com Arduino que envia dados de sensores para a API.

Este curso foca a parte de front-end com Vue e não da API. Confira os treinamentos da Alura sobre API REST e escolha aquele com a que você se sentir mais à vontade em programar.

Agora que já temos nossa API disponibiliza localmente, já podemos integrar nossa aplicação Alurapic com esta API. Poderíamos fazer essa integração realizando requisições Ajax através do XmlHttpRequest, aquele objeto do mundo JavaScript que permite realizar requisições Ajax ou até mesmo realizar esse tipo de requisição através do jQuery. No entanto, o Vue possui um módulo que se integra perfeitamente com todo o seu ecossistema. Este módulo se chama VueResource.

O VueResource não vem habilitado por padrão em nosso projeto, inclusive precisamos baixá-lo através do npm. Para isso, vamos acessar o terminal, parar nossa aplicação caso esta rodando e executar dentro da pasta alurapic o comando:

npm install vue-resource@1.0.3 --save

Por mais que o npm baixe o módulo, ele ainda não esta acessível pela nossa aplicação. Precisamos registrá-lo no global view object do Vue. Como sua ativação é feita no objeto de view global. Para isso, vamos alterar o arquivo alurapic/src/main.js, aquele primeiro arquivo carregado pela nossa aplicação:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

// importando o módulo

import VueResource from 'vue-resource';

new Vue({

el: '#app',

render: h => h(App)

})

Veja que a primeira coisa feita foi importar o módulo VueResource vue-resource. Veja também que adicionei um ponto e vírgula no final da instrução, o que é uma boa prática em JavaScript. No entanto, se vocês repararem, todo o código gerado pelo CLI não usa ponto e vírgula. Estaria o CLI adotando uma má prática?

Como o código que escrevemos passa por uma série de transformações até ser carregado pelo navegador, durante esse processo os pontos e vírgulas são adicionados, no entanto, como eu não programo só em Vue, mas também em JavaScript puro e outro frameworks, eu adicionarei os pontos e vírgulas no final de todas as instruções para não ter que adotar uma postura diferente em cada projeto.

Agora que foi elucidado o uso do ponto e vírgula, não pasta importarmos o módulo VueResource, veja, apenas importamos o módulo dentro de App.vue, mas precisamos ainda registrá-lo. Fazemos isso através do global view object que já temos importado:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

// registrando o módulo/plugin no global view object

Vue.use(VueResource);

new Vue({

el: '#app',

render: h => h(App)

})

Excelente, agora que já temos o módulo ativo em nossa aplicação, vamos utilizá-lo em alurapic/src/App.vue para obter a lista de fotos da nossa API e exibí-las em nosso aplicação.

Vamos voltar para alurapic/src/App.vue. Com nosso módulo VueResource baixado e registrado já temos condições de realizar requisições para nosso API, mas quando faremos isso? Quando o usuário clicar em um botão? Com certeza não.

Queremos que a requisição seja feita assim que nosso componente App tiver sido criado. Para nossa alegria, todo componente do Vue possui ganchos em seu clico de vida (Lifecycle Hooks) que permite executar um código em determinada fase do seu ciclo. No caso, a fase que nos interessa é a created. Vamos aproveitar e remover a lista de fotos:

<template>

<div>

<h1>{{ titulo }}</h1>

<img v-for="foto in fotos" :src="foto.url" :alt="foto.titulo">

</div>

</template>

<script>

export default {

data () {

return {

titulo: "Alurapic",

fotos: [

{

url: 'https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It',

titulo: 'cachorro'

},

{

url: 'https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcTwV4kVzT5McBdGSgqlVeRzubrNH\_mOrrkKseDOGFURq20HmsrelEfMU7It',

titulo: 'cachorro2'

}

]

}

},

created() {

// buscaremos nossas fotos aqui

}

}

</script>

<style>

</style>

Você pode acessar a própria documentação do Vue para ver todos os ganchos suportados no ciclo de vida de um componente neste endereço https://vuejs.org/v2/api/#Options-Lifecycle-Hooks.

Aliás, não faz mais nenhum sentido termos fixa a lista de fotos. Vamos ter apenas um array vazio que receberá os dados vindos da nossa API:

<template>

<div>

<h1>{{ titulo }}</h1>

<img v-for="foto in fotos" :src="foto.url" :alt="foto.titulo">

</div>

</template>

<script>

export default {

data () {

return {

titulo: "Alurapic",

fotos: []

}

},

created() {

// buscaremos nossas fotos aqui

}

}

</script>

<style>

</style>

Para termos certeza que o código em create é chamado assim que nosso componente é criado, vamos colocar temporariamente um alerta. Em seguida, vamos subir novamente nossa aplicação através do CLI para que seja possível acessa-la no navegador:

<template>

<div>

<h1>{{ titulo }}</h1>

<img v-for="foto in fotos" :src="foto.url" :alt="foto.titulo">

</div>

</template>

<script>

export default {

data () {

return {

titulo: "Alurapic",

fotos: []

}

},

created() {

alert('Funcionou!');

}

}

</script>

<style>

</style>

Excelente, o alerta é exibido. Chegou a hora de substituí-lo pelo código que busca nossas fotos da nossa API.

Como ativamos o módulo VueResource no global view object, todo componente terá acesso ao objeto $http através de this. É este o objeto responsável pela realização de requisições ajax que permitem o consumo de API's. Alterando App.vue:

// alurapic/src/App.vue

// código anterior omitido

created() {

this.$http

}

// código posterior omitido

Vimos que se digitarmos em nosso navegador o endereço http://localhost:3000/v1/fotos o navegador por padrão realiza uma requisição com o método GET e retorna uma lista de fotos no formato texto. Agora, precisamos fazer a mesma coisa com o mesmo endereço e método através de $http. A ideia é atribuirmos a lista de fotos retornada à propriedade fotos do nosso componente. Como há um data binding entre essa propriedade e a diretiva v-for, mudanças nessa propriedade farão com que a a view seja atualizada para refletir os dados da nossa lista.

// alurapic/src/App.vue

// código anterior omitido

created() {

this.$http.get('http://localhost:3000/v1/fotos')

}

// código posterior omitido

Agora, estamos pedindo para $http realizar uma requisição do tipo get através o método get que recebe como parâmetro o endereço da API que será consumida. O retorno será uma **promise**.

Uma promise nada mais é do que o resultado futuro de uma ação que esta por acontecer. Sendo assim, podemos escrever nosso código da seguinte maneira:

// alurapic/src/App.vue

// código anterior omitido

created() {

let promise = this.$http.get('http://localhost:3000/v1/fotos');

}

// código posterior omitido

O retorno é uma promise, ainda não é o valor que temos interesse, no caso, a lista de fotos. Requisições ajax são assíncronas e podem demorar milissegundos ou até mesmo segundos dependendo das condições da rede. Sendo assim, quando ela for realmente efetivada, é através do método then da nossa promise que temos acesso à resposta vinda do servidor:

// alurapic/src/App.vue

// código anterior omitido

created() {

let promise = this.$http.get('http://localhost:3000/v1/fotos');

promise .then(function(res) {

console.log(res);

});

}

// código posterior omitido

O código que escrevemos esta usando sintaxe do ES5. Podemos simplificar nosso código usando arrow function do E2015:

// alurapic/src/App.vue

// código anterior omitido

created() {

let promise = this.$http.get('http://localhost:3000/v1/fotos');

promise .then(res => console.log(res));

}

// código posterior omitido

Olhando no console do navegador vemos a mensagem:

Response {url: "http://localhost:3000/v1/fotos", ok: true, status: 200, statusText: "OK", headers: Headers…}

Veja que res não é ainda nossa lista de fotos, mas um objeto que contém, além da lista, uma série de informações de controle da resposta. Para obtermos a lista de fotos resultante, precisamos converter a lista no formato texto para uma lista de objetos em JavaScript através de res.json():

// alurapic/src/App.vue

// código anterior omitido

created() {

let promise = this.$http.get('http://localhost:3000/v1/fotos');

promise

.then(res => res.json());

}

// código posterior omitido

Neste ponto, estamos solicitando que para a resposta, em nosso caso, res, que converte os dados recebidos em objetos JavaScript. Sendo objetos, podem ser manipulados facilmente pelo nosso código. No entanto, toda promise possui uma característica peculiar. Tudo que é retornado pelo método then é acessível através da próxima chamada ao then. Como estamos usando arrow functions desta forma, há um retorno implícito, no caso, o retorno de res.json():

// alurapic/src/App.vue

// código anterior omitido

created() {

let promise = this.$http.get('http://localhost:3000/v1/fotos');

promise

.then(res => res.json())

.then(fotos => this.fotos = fotos);

}

// código posterior omitido

Na próxima chamada à then, usamos como nome de parâmetro fotos, mas poderia ser qualquer nome. Mas é através desse parâmetro que temos acesso as fotos retornadas pela nossa API. O que fazemos é atribuir essa nova lista à propriedade this.fotos. Como a lista foi atualizada, o sistema de data binding do Vue se encarregará de atualizar a view com os novos dados.

No entanto, não é comum trabalhar com a variável promise ou qualquer outra intermediária. Vamos alterar nosso código e deixando-o mais enxuto.

// alurapic/src/App.vue

// código anterior omitido

created() {

this.$http.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos);

}

// código posterior omitido

Mas se um erro acontecer? É por isso que a função then recebe dois parâmetros. A função que será chamada no sucesso e aquela chamada quando algo der errado. Inclusive temos acesso ao erro através do parâmetro recebido por ela que chamaremos de err:

// alurapic/src/App.vue

// código anterior omitido

created() {

this.$http.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

// código posterior omitido

Por enquanto vamos apenas logar a mensagem de erro no console.

Neste capítulo aprendemos:

* a subir uma API para ser consumida pela nossa aplicação
* a entender a separação entre cliente e API
* a baixar e registrar o módulo VueResource
* o suporte à lifecycle hooks que todo componente possui
* que recursos no global view object ficam disponíveis para todas as outras view
* consumir uma API através de $http
* o conceito de promise
* enxergar ainda melhor o poder do data binding

Até agora não lidamos com a parte estética da nossa aplicação e sabemos que no mundo web, o CSS é aquela tecnologia responsável pela aplicação de estilos. Nela, através da tag link importamos folhas de estilos que podem estar em um ou mais arquivos. No entanto, nossa aplicação utilizará uma outra abordagem. Vejamos.

Como Vue trabalha com o conceito de componente, a ideia é definirmos os estilos no próprio componente. Sendo assim, qualquer desenvolvedor que olhe o código do componente compreenderá os estilos que operam sobre ele. Vamos olhar o arquivo alurapic/src/App.vue.

Pode parecer estranho em um primeiro momento, mas App.vue é um componente. Dessa forma, páginas em uma aplicação com vue são todas componentes. Nesse sentido, precisamos realizar os seguintes ajustes em nosso componente:

* Ajustes:
  + centralizar o título
  + adicionar uma margem direita e esquerda de mesmo tamanho na página inteira
  + utilizar a fonte Helvetica
  + remover o bullet que aparece antes de cada item da lista
  + cada foto deve ir ao lado da outra

Primeiro, vamos adicionar as classes "corpo", "centralizado" e "lista-fotos", 'lista-fotos-item" nas tags div, h1 , ul e lirespectivamente.

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotos">

<img :src="foto.url" :alt="foto.titulo">

</li>

</ul>

</div>

</template>

<script>

// código omitido

</script>

<style>

</style>

Agora, dentro da tag style vamos adicionar os estilos que atendem cada um dos nosso requisitos de design:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotos">

<img :src="foto.url" :alt="foto.titulo">

</li>

</ul>

</div>

</template>

<script>

// código omitido

</script>

<style>

.centralizado {

text-align: center;

}

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

</style>

Usamos tags para não ficarmos amarrados à determinada estrutura do HTML. Por exemplo, se mais tarde alguém trocar nosso título de h1 para h2, como estamos usando um seletor de classe, nosso estilo ainda continuará a ser aplicado.

Como livereloading, a alteração do nosso componente App.vue é refletida instantaneamente em nosso navegador. Temos o título centralizado entre outros estilos aplicados. Mas como o estilo definido em App.vue funciona, se estilos devem ser declarados ou definidos na tag <head>. Aliás, nosso componente nem possui essa tag!

Isso funciona, os estilo definido na tag style do componente é inserida em index.html assim que abrimos nossa aplicação no navegador. Não precisamos nos preocupar com isso, a própria infra do Vue se encarregará de fazer isso para nós.

Ainda não terminamos com App.vue. Recebemos uma recomendação de exibir cada foto dentro de um painel e esse painel deve possui um título. Uma estrutura que podemos usar é a seguinte:

<!-- EXEMPLO DA ESTRUTURA, NÃO ENTRA AINDA EM NOSSO CÓDIGO -->

<div class="painel">

<h2 class="painel-titulo">Titulo do painel aqui</h2>

<div class="painel-corpo">

Conteúdo do painel aqui

</div>

</div>

Agora, vamos incorporar essa estrutura em nosso App.vue:

<!-- alurapic/src/App.vue -->

<template>

<div>

<h1 class="centralizado">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotos">

<div class="painel">

<h2 class="painel-titulo">{{foto.titulo}}</h2>

<div class="painel-corpo">

<img :src="foto.url" :alt="foto.titulo">

</div><!-- fim painel-corpo -->

</div><!-- fim painel -->

</li>

</ul>

</div>

</template>

<script>

// código omitido

</script>

<style>

/\* código omitido \*/

</style>

Veja que usamos interpolação para exibir o título da foto na tag que exibe o título do painel. Vale lembrar que isso é possível porque a variável foto declarada na diretiva v-for é acessível por todos elementos filhos do elemento na qual foi utilizada.

É claro que a visualização do nosso painel deixa a desejar, é por isso que vamos adicionar um estilo para o painel:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotos">

<div class="painel">

<h2 class="painel-titulo">{{foto.titulo}}</h2>

<div class="painel-corpo">

<img :src="foto.url" :alt="foto.titulo">

</div><!-- fim painel-corpo -->

</div><!-- fim painel -->

</li>

</ul>

</div>

</template>

<script>

// código omitido

</script>

<style>

.centralizado {

text-align: center;

}

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

/\* estilo do painel \*/

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

</style>

O problema é que a imagem vaza para fora do painel, para isso, precisamos fazer com que a imagem tenha como largura 100% do elemento pai na qual esta inserida. Aliás, é muito comum queremos este tamanho para imagens responsivas que se adaptam ao tamanho de diferentes telas. Vamos adicionar a classe imagem-responsiva na tag img e criar esse estilo:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="titulo">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotos">

<div class="painel">

<h2 class="painel-titulo">{{foto.titulo}}</h2>

<div class="painel-corpo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</div><!-- fim painel-corpo -->

</div><!-- fim painel -->

</li>

</ul>

</div>

</template>

<script>

// código omitido

</script>

<style>

.titulo {

text-align: center;

}

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

.imagem-responsiva {

width: 100%;

}

/\* estilo do painel \*/

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

</style>

Excelente, o visual da nossa lista de fotos está bem melhor do que antes, mas tem algo que deixa a desejar em nosso código. É isso que veremos no próximo vídeo.

As chances de querermos utilizar o mesmo painel em outras páginas da nossa aplicação não são pequenas. E para que isso seja possível, somos obrigados a copiar o código HTML e o CSS. Mesmo em uma aplicação Web tradicional na qual podemos importar o mesmo CSS em várias páginas, a marcação HTML terá que ser refeita.

Durante esse processo, essa ou aquela classe pode ser omitida ou essa ou aquela tag pode não ter sido fechada o que pode ocasionar problemas. Mais ainda, se a estrutura do painel mudar, seremos obrigados a alterar em todos os lugares. A boa notícia é que isso não precisa ser assim.

Podemos tornar nosso painel um componente de Vue e reutilizá-lo em qualquer outro local da nossa aplicação. Dizemos que nosso painel será um componente shared, compartilhado. O mesmo não ocorre com App.vue que é específico da nossa aplicação e não faz sentido ser reutilizado por outra aplicação ou ainda dentro da mesma aplicação.

## Criando um shared component

Dessa forma, vamos criar a pasta alurapic/src/components/shared/painel. Todos nosso componentes que criarmos a partir de agora ficarão dentro da pasta components. Além disso, todos aqueles que forem componentes reutilizáveis e compartilháveis ficarão dentro da subpasta shared. Criaremos outra subpasta por componente e dentro dela teremos o arquivo .vue. Sendo assim vamos seguir a convenção de criar arquivos de componente começando em pascal case:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

</template>

<script>

export default {

}

</script>

<style>

</style>

Agora que temos o esqueleto básico do nosso componente constituído pelas tags template, script e style, vamos mover a marcação do painel e seu estilo para dentro do componente:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo"></h2>

<div class="painel-conteudo">

</div>

</div>

</template>

<script>

export default {

}

</script>

<style>

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

</style>

Nosso componente ainda não esta pronto, mas assim que estiver poderemos utilizá-lo em App.vue da seguinte maneira:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotos">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<!-- código posterior omitido -->

Veja como ficou simples a marcação do de App.vue. No caso, estamos usando o componente Painel.vue como meu-painel da definição do template de App.vue. Além disso, veja que o componente recebe seu título na propriedade titulo. Aliás, esse é um ponto que precisamos nos debruçar.

Todo componente em Vue é uma unidade de código que pode encapsular sua marcação, estilo e comportamento, este último, ações que podem ser realizadas com ele. Por enquanto, só disponibilizamos dados para o template e não executamos nenhum comportamento, algo que veremos ainda nesse curso.

Para que seja possível se comunicar com um componente passando dados para ele, precisamos adicionar a propriedade titulo na lista de propriedades recebíveis do componente App.vue. Alterando o componente:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<div class="painel-conteudo">

</div>

</div>

</template>

<script>

export default {

props: ['titulo']

}

</script>

<style>

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

</style>

Veja que na parte scripts, temos a propriedade props. Nela podemos passar uma lista de propriedades que podem ser recebidas pelo componente. Essas propriedades podem ser acessadas no template do componente através de interpolação. É por isso que dentro da tag que representa o título do componente, usamos {{ titulo }}.

Com essa última alteração, nada será exibido em nosso navegador. Isto porque precisamos importar o componente Painel.vue em App.vue para poder utilizá-lo:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotos">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

// importando nosso Painel

import Painel from './components/shared/painel/Painel.vue';

export default {

// código omitido

}

</script>

<style>

/\* estilos omitidos \*/

</style>

Importar nosso Painel ainda não é suficiente. Precisamos indicar em App.vue como iremos referenciar o componente em seu template. Podemos escolher qualquer nome, no caso, vamos escolher meu-painel. É através da propriedade components que associamos o nome meu-painel ao componente:

<!-- alurapic/src/App.vue -->

<script>

import Painel from './components/shared/painel/Painel.vue'

export default {

components: {

'meu-painel': Painel

},

data () {

return {

titulo: 'Alurapic',

fotos: []

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

/\* estilos omitidos \*/

</style>

Excelente! Quando nosso página recarrega, temos um painel para cada foto. No entanto, apenas o título do painel é exibido. Onde está o seu conteúdo, no caso, nosso foto? Entenderemos o que houve no próximo vídeo.

Quando o Vue renderiza nosso componente Painel em App, ele não entende que deve preservar tudo aquilo que esta entre as tags <meu-painel>. O Vue manipula aquela parte do DOM trocando-a pela renderização do nosso componente Painel. Para que isso seja possível, precisamos indicar no template de Painel a área que queremos considerar como um slot, ou seja, aquela área que recebera tudo aquilo que tiver dentro da tag <meu-painel>. Para isso, vamos alterar o template alurapic/src/components/shared/painel/Painel.vue e trocar a div conteúdo pelo componente slot. Nosso componente final fica assim:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<slot class="painel-conteudo">

</slot>

</div>

</template>

<script>

export default {

props: ['titulo']

}

</script>

<style>

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

</style>

Criamos nosso primeiro componente que encapsula marcação HTML, dados e seu estilo. Mas que tal vermos como ficará cada item dentro do nosso painel com sombra? É algo que o design não pediu, mas que podemos fazer facilmente usando o seletor universal do CSS.

Alterando Painel.vue:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<slot class="painel-conteudo">

</slot>

</div>

</template>

<script>

export default {

props: ['titulo']

}

</script>

<style>

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

/\* colocando todos os elementos do painel com box shadow \*/

\* {

box-shadow: 5px 5px 5px;

}

</style>

## Quando estilos globais dão problema

Temos um problema. Não só os elementos do painel ficaram com sombra, mas o painel em si, inclusive o título da de App.vue que não tem nada a haver com a história. O problema é que aplicamos um seletor que afetará todos conteúdo da página, não apenas o componente. Dizemos que o estilo do componente Painel vazou do componente para o mundo externo. Podemos resolver isso facilmente especificando nosso seletor:

.painel \* {

box-shadow: 5px 5px 5px;

}

## Estilos com escopo de componente

Mas se tivermos algum outro elemento da nossa página que faça uso da classe painel? Corremos o risco do estilo do nosso componente ser aplicado indevidamente. Se quisermos que os estilos definidos em styles do nosso componente afete apenas o componente, basta adicionarmos a propriedade scoped na tag style. Vamos adicionar esse atributo, inclusive utilizar o seletor universal que usamos anteriormente:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<slot class="painel-conteudo">

</slot>

</div>

</template>

<script>

export default {

props: ['titulo']

}

</script>

<style scoped>

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

\* {

box-shadow: 5px 5px 5px;

}

</style>

É claro que não usaremos esse estilo tosco que acabamos de criar, mas veja que as sombras agora afetam apenas o nosso componente. É uma boa prática usar um escopo de componente quando criamos nossos componentes reutilizáveis, evitando assim que os estilos de um componente interfiram no estilos de outro.

Ah, e não esqueça de remover o último seletor, ok?

É possível termos mais de um slot por componente, por exemplo, para inserirmos conteúdo em locais diferentes do nosso componente. Para isso existe o named slot. Vejamos um exemplo:

<!-- ComponenteQualquer.vue -->

<template>

<div>

<slot name="cabecalho" class="header" ></slot>

<hr>

<slot class="body"></slot>

<hr>

<slot name="rodape" class="footer"></slot>

</div>

</template>

<script>

export default {}

</script>

Veja que nosso componente possui três slots. Dois nomeados e outro não. Agora, quando ele for utilizado em outro componente podemos fazer:

<componente-qualquer>

<div slot="cabecalho">

<h1>Bem-vindo!</h1>

</div>

<p>Seja bem-vindo à Alura!</p>

<div slot="rodape">

<p>copyright 2017</p>

</div>

</componente-qualquer>

As tag's divs que receberam a propriedade slot e seu nome, serão incluídas dentro do seu respectivo slot. Já o parágrafo Seja bem-vindo à Alura será inserido no slot padrão, aquele que não recebeu um nome em nosso componente.

Neste capítulo aprendemos:

* a estilizar um componente
* boa prática na organização do projeto
* a criar um componente reutilizável
* como passar parâmetro para um componente (comunicação)
* o conceito de slot e preservação de conteúdo
* importação de um componente em outro
* a definir e delimitar o escopo de estilos ao componente no qual foram declarados

Ainda não somos capazes de incluir novas fotos em nossa aplicação, mas com certeza o número de fotos pode ser bem grande e ajudar o usuário a encontrar uma foto por parte do título é de grande valia.

A primeira coisa que precisamos fazer é adicionar um input que capturará o título para que possamos utilizá-lo como critério de filtro da nossa lista. Inclusive, já vamos adicionar um estilo para ele na tag style do nosso componente App.vue:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<input type="search" class="filtro" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotos">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

// código omitido

</script>

<style>

/\* código anterior omitido \*/

.filtro {

display: block;

width: 100%;

}

</style>

Excelente, agora precisamos arrumar uma forma de conseguirmos termos acesso ao valor digitado pelo usuário a cada dígito. Temos uma situação diferente. Antes, aprendemos a criar uma associação de dados, no inglês data binding que fluía dos dados para a view, ou seja, para o template do nosso componente. Um exemplo é a propriedade fotosretornada pela nossa função data. Quando ela é modificada pelo retorno de $http os dados fluem para a view fazendo com que ela seja atualizada. Queremos algo agora um pouco diferente, queremos que o dado flua da view para dentro do nosso componente.

Primeiro, vamos adicionar no objeto retornado pela função data do nosso componente a propriedade filtro:

<!-- alurapic/src/App.vue -->

<template>

<!-- código do template omitido -->

</template>

<input class="filtro" placeholder="filtre pelo título da foto">

<script>

import Painel from './components/shared/painel/Painel.vue'

export default {

components: {

'meu-painel': Painel

},

data () {

return {

titulo: 'Alurapic',

fotos: [],

filtro: ''

}

},

created() {

// código omitido

}

}

</script>

<style>

/\* código omitido \*/

</style>

## A diretiva v-on e mais um tipo de data binding

Agora, precisamos fazer uma associação unidirecional que flua da view para a fonte de dados, no caso, para a propriedade filtro. Queremos que a propriedade filtro seja atualizada a cada dígito no campo. Sabemos que no mundo JavaScript há o evento input disparado toda vez que algum valor é inserido no campo.

Vamos alterar a tag input do template e adicionar a diretiva **v-on:input**:

<!-- alurapic/src/App.vue -->

<!-- código anterior omitido -->

<input type="search" class="filtro" v-on:input placeholder="filtre pelo título da foto">

<!-- código posterior omitido -->

É através da diretiva v-on: que podemos elaborar uma resposta para eventos do JavaScript. No caso, adicionamos o nome do evento logo após os dois pontos. No entanto, precisamos associar algum código para este evento. Completando o código:

<!-- alurapic/src/App.vue -->

<!-- código anterior omitido -->

<input class="filtro" v-on:input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<!-- código posterior omitido -->

Quando digitamos no campo, a expressão adicionada entre aspas será executada. Veja que ela atribui à filtro o valor de $event.target.value. Quando um evento em JavaScript é disparado, há um objeto especial chamado event que detém um monte de informação sobre o evento disparado. No caso usamos $event pois esse é um objeto especial do Vue. Dizemos que é um event original encapsulado pelo Vue. É através dele, assim como o event padrão que podemos ter acesso ao alvo do evento, no caso, o seu target. No caso o target é o próprio input. É por isso que do target podemos fazer .value para obter o valor do input.

Se vocês ainda desconfiam que o filtro esteja recebendo o valor do digitado por nós, coloque a seguinte interpolação logo abaixo do input:

<!-- alurapic/src/App.vue -->

<!-- código anterior omitido -->

<input type="search" class="filtro" v-on:input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

{{ filtro }}

<!-- código posterior omitido -->

Veja que a cada dígito no local da interpolação será exibido o valor que digitamos no input. Isso é fantástico, pois com pouquíssimo esforço estamos capturando o valor digitado pelo usuário e exibindo-o automaticamente na tela.

Por fim, vale ressaltar que v-onrealiza um data binding unidirecional que flui da view para os dados e a interpolação ou v-bind realiza uma associação unidirecional que flui dos dados para view.

Agora que já aprendemos a capturar o filtro do usuário, que fazemos uso dele para filtrarmos a nossa lista?

Agora que já temos o valor digitado pelo usuário podemos utilizá-lo para filtrar nossa lista de fotos.

Pensem comigo. Alguma lógica terá que ser aplicada no dado fotos para que apenas as fotos que tenham parte do título sejam consideradas. Certo? Mas não podemos simplesmente sair removendo os itens da lista, porque se o usuário desistir de um filtro e utilizar outro temos que exibir o item. Outro ponto, se o campo estiver em branco vamos exibir todas as fotos para ele. E agora, como lidaremos com essa lógica?

O Vue oferece uma solução elegante chamada de *computed property*, em português, propriedade computada. Hoje temos um dado que é fotos, mas essa lista precisará ser computada podendo retornar uma lista diferente da original. Sempre que tivemos que realizar algum cálculo ou aplicar alguma lógica dinamicamente podemos usar computed property.

Vamos adicionar em nosso componente App a propriedade computed e nela passar um objeto. Cada propriedade do objeto é obrigatoriamente uma função. Não poderia ser diferente, já que executaremos uma lógica:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<input type="search" class="filtro" v-on:input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotos">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from './components/shared/painel/Painel.vue'

export default {

components: {

'meu-painel': Painel

},

data () {

return {

titulo: 'Alurapic',

fotos: [],

filtro: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

// filtra a lista, por enquanto vamos retornar uma lista em branco

return [];

} else {

// se o campo estiver vazio, não filtramos, retornamos a lista

return this.fotos;

}

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

/\* código omitido \*/

</style>

Uma *computed property* pode ser acessada como uma propriedade em nossa view. Sendo assim, na diretiva v-forvamos usar fotosComFiltro no lugar de fotos:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<input type="search" class="filtro" v-on:input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComfiltro">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from './components/shared/painel/Painel.vue'

export default {

components: {

'meu-painel': Painel

},

data () {

return {

titulo: 'Alurapic',

fotos: [],

filtro: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

// filtra a lista, por enquanto vamos retornar uma lista em branco

return [];

} else {

// se o campo estiver vazio, não filtramos, retornamos a lista

return this.fotos;

}

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

/\* código omitido \*/

</style>

Faça um teste. Se o campo estiver em branco, tudo será exibido. Se qualquer coisa for digitado no campo do filtro, nada será exibido. Chegou a hora de aplicarmos a lógica que retorna a lista filtrada pelo que digitamos. Para isso precisamos saber um pouquinho de expressão regular e o uso da poderosa função filter que todo array possui:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<input type="search" class="filtro" v-on:input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from './components/shared/painel/Painel.vue'

export default {

components: {

'meu-painel': Painel

},

data () {

return {

titulo: 'Alurapic',

fotos: [],

filtro: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

// criando uma expressão com o valor do filtro, insensitivo

let exp = new RegExp(this.filtro.trim(), 'i');

// retorna apenas as fotos que condizem com a expressão

return this.fotos.filter(foto => exp.test(foto.titulo));

} else {

return this.fotos;

}

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

/ \* código omitido \*/

</style>

Veja que dentro da nossa *computed property* podemos acessar dado filtro através de this. Isso é possível porque Vue internamente aplica sua mágica para que o this tenha acesso a todos as propriedade definidas na função data, o que é excelente para o desenvolvedor.

Nosso componente final fica assim:

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<input type="search" class="filtro" v-on:input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<img class="imagem-responsiva" :src="foto.url" :alt="foto.titulo">

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from './components/shared/painel/Painel.vue'

export default {

components: {

'meu-painel': Painel

},

data () {

return {

titulo: 'Alurapic',

fotos: [],

filtro: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

let exp = new RegExp(this.filtro.trim(), 'i');

return this.fotos.filter(foto => exp.test(foto.titulo));

} else {

return this.fotos;

}

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

.centralizado {

text-align: center;

}

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

.imagem-responsiva {

width: 100%;

}

.filtro {

display: block;

width: 100%;

}

</style>

Tornamos a experiência do usuário ainda melhor com a possibilidade de filtrar nossa lista de fotos. No entanto, que tal permitirmos que nosso componente Painel seja colapsável? Se implementarmos esse comportamento no componente, todos os lugares que ele for utilizado também terá o recurso.

## A diretiva v-show

Queremos esconder o conteúdo do painel, não removê-lo. Para isso há a diretiva v-show. Quando adicionada em um elemento, quando seu valor for true, o elemento será exibido, se for false, será ocultado. Por debaixo dos panos a diretiva realiza um display: none.

Vamos adicioná-la no slot de App. Lembrando que este é o local onde serão inseridos os conteúdos do painel.

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<slot class="painel-conteudo" v-show="false">

</slot>

</div>

</template>

<script>

export default {

props: ['titulo']

}

</script>

<style>

/\* código omitido \*/

</style>

Como atribuímos o valor false, nossos painéis devem exibir apenas o título do painel, mas isso não acontece. O painel e seu conteúdo continuam sendo exibidos.

## Pegadinha do v-show e slots

O problema é que não podemos usar a diretiva v-show diretamente na tag slot. Se quisermos fazer com que o slot e todo seu conteúdo desaparece, precisamos envolvê-lo em uma tag div e nela usar a diretiva. Faremos isso, inclusive vamos mudar a class painel-conteudo de slot para a nova div:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<div class="painel-conteudo" v-show="false">

<slot></slot>

</div>

</div>

</template>

<script>

export default {

props: ['titulo']

}

</script>

<style>

.painel {

padding: 0 auto;

border: solid 2px grey;

display: inline-block;

margin: 5px;

box-shadow: 5px 5px 10px grey;

width: 200px;

height: 100%;

vertical-align: top;

text-align: center;

}

.painel .painel-titulo {

text-align: center;

border: solid 2px;

background: lightblue;

margin: 0 0 15px 0;

padding: 10px;

text-transform: uppercase;

}

</style>

Perfeito! Todos os painéis são exibidos, mas apenas seu título. Se quisermos voltar a exibir seu conteúdo basta atribuirmos true ao valor de v-show. No entanto, queremos que essa troca do valor de v-show seja feita pelo usuário toda vez que ele clicar no título do painel.

O primeiro passo é adicionar no objeto retornando pela função data a propriedade visivel que começa como true e associá-la à diretiva v-show ao invés de deixarmos um valor fixo como true ou false:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo">{{ titulo }}</h2>

<div class="painel-conteudo" v-show="visivel">

<slot></slot>

</div>

</div>

</template>

<script>

export default {

props: ['titulo'],

data() {

return {

visivel: true

}

}

}

</script>

<style>

/\* código omitido \*/

</style>

Bom, ainda continuamos na mesma, a diferença é que fizemos usa associação de v-show com visivel da nossa função data. Agora é que vem o truque.

## Bind de eventos, mais uma vez

Aprendemos que através da diretiva v-on podemos executar um código a partir de um evento do JavaScript.Nesse caso, vamos adicionar um v-on:dblclick para responder ao evento click na tag <h2> que representa o título do nosso painel:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo" v-on:dblclick="visivel = !visivel">{{ titulo }}</h2>

<div class="painel-conteudo" v-show="visivel">

<slot></slot>

</div>

</div>

</template>

<script>

export default {

props: ['titulo'],

data() {

return {

visivel: true

}

}

}

</script>

<style>

/\* código omitido \*/

</style>

Veja que no evento dblclick estamos mudando o valor da propriedade visivel para o seu valor oposto. Se ela vale true, mudaremos no clique para false. Se clicarmos novamente, mudaremos para true.

Um novo teste demonstra que nossa estratégia funcionou. Veja que nosso painel agora possui seu primeiro comportamento e, não importa onde nosso componente Painel seja utilizado, esse comportamento o acompanhará.

## Atalho para v-on

Da mesma forma que aprendemos um atalho para v-bind, há um atalho para a diretiva v-on. No caso, basta adicionarmos o nome do evento com o prefixo @:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo" @dblclick="visivel = !visivel">{{ titulo }}</h2>

<div class="painel-conteudo" v-show="visivel">

<slot></slot>

</div>

</div>

</template>

<script>

export default {

props: ['titulo'],

data() {

return {

visivel: true

}

}

}

</script>

<style>

/\* código omitido \*/

</style>

Inclusive podemos alterar em App de v-on:input para @input.

Tudo muito bacana, mas que tal realizar esse processo de exibe e oculta com efeito, por exemplo, um fade? É isso que veremos no próximo vídeo.

Vue não faz mágicas no que diz respeito à animações. É o desenvolvedor que deve criar os estilos que devem ser aplicados no momento correto para que os efeitos aconteçam. O que o Vue faz é indicar a hora exata na qual nossas classes devem ser aplicadas.

No entanto, o Vue precisa delimitar a área do template no qual fará essas indicações. Como queremos realizar uma transição no conteúdo do nosso Painel, vamos envolver nossa div com a classe painel-conteudo pelo componente transition. Obrigatoriamente precisamos escolher um name para ele. No caso, usaremos painel-fade:

<!-- alurapic/src/components/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo" @dblclick="visivel = !visivel">{{ titulo }}</h2>

<transition name="painel-fade">

<div class="painel-conteudo" v-show="visivel">

<slot></slot>

</div>

</transition>

</div>

</template>

<script>

// código omitido

</script>

<style>

/\* código omitido \*/

</style>

Em primeiro lugar, transition só pode ser usado com um elemento, se tentarmos colocar mais um elemento como filho do nosso wrapper receberemos um erro.

Se clicarmos em nossa Painel, nada ainda acontece. Precisamos criar o estilo para painel-fade. O problema é que para realizarmos uma transição, precisamos aplicar estilos diferentes, por exemplo, quando o elemento esta sendo exibido ou quando ele esta sendo escondido.

É por isso que o componente transition adiciona dinamicamente nos seus elementos filhos as seguintes classes dinamicamente:

painel-fade-enter // antes do elemento ser incluído ou removido, o estado atual

painel-fade-enter-active // quando o elemento esta sendo incluído

painel-fade-leave-active // quando o elemento esta sendo removido

Sendo assim, podemos fazer o seguinte:

<!-- alurapic/src/componets/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo" @dblclick="visivel = !visivel">{{ titulo }}</h2>

<transition name="painel-fade">

<div class="painel-conteudo" v-show="visivel">

<slot></slot>

</div>

</transition>

</div>

</template>

<script>

// código omitido

</script>

<style>

/\* código anterior omitido \*/

.painel-fade-enter, .painel-fade-leave-active {

opacity: 0

}

</style>

Por fim, usamos os outros dois estilos para aplicar a transição:

<!-- alurapic/src/componets/shared/painel/Painel.vue -->

<template>

<div class="painel">

<h2 class="painel-titulo" @dblclick="visivel = !visivel">{{ titulo }}</h2>

<transition name="painel-fade">

<div class="painel-conteudo" v-show="visivel">

<slot></slot>

</div>

</transition>

</div>

</template>

<script>

// código omitido

</script>

<style>

/\* código anterior omitido \*/

.painel-fade-enter, .painel-fade-leave-active {

opacity: 0

}

.painel-fade-enter-active, .painel-fade-leave-active {

transition: opacity .4s

}

</style>

Façamos um teste agora. Quando ocultamos e exibimos no elemento um fade é realizado.

O componente Painel foi o primeiro componente reutilizável que criamos. No entanto, por mais ínfima que nossa tag img possa ser teremos que lembrar sempre de adicionar width: 100% para que ela se adapte a diversos tamanhos. Para evitar que esqueçamos desse detalhe, que tal criarmos o componente ImagemResponsiva que encapsulará a tag img com seu estilo?

Vamos criar o arquivo alurapic/src/components/shared/imagem-responsiva/ImagemResponsiva.vue:

<!--`alurapic/src/components/shared/imagem-responsiva/ImagemResponsiva.vue -->

<template>

<img class="imagem-responsiva" :src="url" :alt="titulo">

</template>

<script>

export default {

props: ['url', 'titulo']

}

</script>

<style scoped>

.imagem-responsiva {

width: 100%;

}

</style>

Agora, vamos importar nosso componente em App.vue e utilizá-lo no lugar da tag img. Aliás, não esqueça de remover o estilo de App que movemos para dentro do nosso componente:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<h1 class="centralizado">{{ titulo }}</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from './components/shared/painel/Painel.vue';

import ImagemResponsiva from './components/shared/imagem-responsiva/ImagemResponsiva.vue'

export default {

components: {

'meu-painel': Painel,

'imagem-responsiva': ImagemResponsiva

},

data () {

return {

titulo: 'Alurapic',

fotos: [],

filtro: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

let exp = new RegExp(this.filtro.trim(), 'i');

return this.fotos.filter(foto => exp.test(foto.titulo));

} else {

return this.fotos;

}

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos);

}

}

</script>

<style>

.centralizado {

text-align: center;

}

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

.filtro {

display: block;

width: 100%;

}

</style>

É claro que uma imagem responsiva pode ter outras propriedades, mas a boa notícia é que em todo lugar que nosso componente for utilizado, quando mudarmos sua definição, a alteração será aplicada em todos os lugares.

Neste capítulo aprendemos:

* a diretiva v-on e como lidar com eventos do JavaScript
* atalho para diretiva v-on
* diferença de v-on para v-bind
* a filtrar uma lista
* compreender o uso de computed property
* que podemos acessar qualquer propriedade de data através de this
* a diretiva v-show
* detalhe entre v-show e slots
* como realizar transições com auxílio do componente transition
* a criar mais um componente

Hoje temos apenas um componente que equivale a uma de nossas páginas, o App. Precisamos de mais um, aquele que será o responsável pelo cadastro de novas fotos. Esse componente não será um componente compartilhável, pois não queremos reutilizar uma "página" dentro de outra. É por isso que ele não ficará dentro da pasta shared.

Vamos criá-lo em alurapic/src/components/cadastro/Cadastro.vue. Por enquanto será uma página simples, com apenas um título:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

</div>

</template>

<script>

export default {

}

</script>

<style scoped>

.centralizado {

text-align: center;

}

</style>

Excelente, criamos nosso novo componente. No entanto, quando nossa aplicação é visualizada no navegador é o componente App o primeiro a ser renderizado. Tudo bem, mas se quisermos agora carregar o componente Cadastro? Hora queremos um componente e hora queremos outro. Como resolver? Para chegarmos à solução precisamos primeiro entender o conceito de Single Page Application.

A grosso modo, Single Page Applications são páginas que não recarregam durante seu uso com experiência semelhante a um aplicativo. Neste tipo de aplicação carregamos apenas a página principal da aplicação, por exemplo, index.html. Depois que essa página é carregada, o desenvolvedor usurpa o controle do navegador e a busca de novas páginas deixa de ser feita pelo navegador e passam a ser feitas pelo JavaScript.

Se as páginas são carregadas através de JavaScript elas podem ser inseridas no meio de index.html através de manipulação de DOM. Com isso, o endereço no browser continua o mesmo, o que muda é o conteúdo de index.htmlque é carregado dinamicamente. No entanto, se o endereço de index.html não muda, como poderemos por exemplo, abrir direto em nosso navegador a página X ou Y? Veja que há um monte de complexidade envolvida nesse processo.

É por isso que frameworks como Vue vem nos ajudar. Por fim, para entendermos como uma página que nunca recarrega é capaz de exibir outras páginas precisamos entender o conceito de rotas e como implementá-la com Vue. Assunto do próximo vídeo.

Agora que temos uma visão geral do que é uma SPA, precisamos realizar um ajuste em nossa aplicação antes de continuar. Hoje, temos o componente App, o primeiro a ser carregado pela nossa aplicação e que exibe nossos painéis de fotos que podem ser filtrados. No entanto, em uma SPA, o primeiro componente visual funciona como um placeholder, ou melhor, local no qual outras páginas serão inseridas através do framework. Lembre-se que páginas também são componentes, dessa forma, será em App que de acordo com um endereço especial acessado pelo usuário, o Vue saberá qual componente enxertar dentro de App para exibí-lo. Entenda-o como uma grande lacuna que hora exibirá um componente, hora exibirá outro. Sendo assim, precisamos extrair todo o código que lista nossas fotos para um novo componente. Vamos chamá-lo de Home.

Para acelerar o processo, copie o conteúdo do arquivo alurapic/src/App.vue para dentro do arquivo alurapic/src/components/home/Home.vue. Com o conteúdo copiado, faremos poucos ajustes. O primeiro, é que a div do template não terá a classe corpo e o título da página será fixo como <h1>Alurapic</h1>. Como o título é fixo, não precisamos da propriedade titulo em nossa função data, inclusive podemos remover a classe corpo da tag <style>. Nosso arquivo fica assim:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

export default {

components: {

'meu-painel': Painel,

'imagem-responsiva': ImagemResponsiva

},

data () {

return {

fotos: [],

filtro: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

let exp = new RegExp(this.filtro.trim(), 'i');

return this.fotos.filter(foto => exp.test(foto.titulo));

} else {

return this.fotos;

}

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

.centralizado {

text-align: center;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

.filtro {

display: block;

width: 100%;

}

</style>

Veja também que o caminho dos imports precisou mudar para ../shared para descer um nível, já que o componente agora esta em sua pasta específica. Excelente, agora temos Home e Cadastro como componentes que correspondem a páginas em nossa aplicação.

Agora, vamos ajustar App.vue. Não haverá um código sequer dentro de <script> e o único estilo dentro de <style>será a classe corpo. Seu template terá, por enquanto, apenas a tag <div class="corpo"></div>:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<!-- é aqui que encontrará os componentes de página que queremos visualizar -->

</div>

</template>

<script>

</script>

<style>

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

</style>

Você não deve ter ficado surpreso em ver no browser a página em branco, isso porque como App é o primeiro componente a ser carregado, seu template esta vazio. Chegou a hora de realizarmos as configurações necessárias para que seja possível hora exibir o conteúdo de Home e hora o conteúdo de Cadastro.

Agora que já separamos a responsabilidade de App que será o ponto no qual exibiremos os componentes Home e Cadastro já podemos instalar a infraestrutura necessária para que a magia do SPA aconteça. Por padrão o Vue não é capaz de resolver a questão que abordamos que é o carregamento de um ou outro componente de acordo com uma rota específica, para isso, precisaremos instalar o módulo VueRouter.

No terminal, dentro da pasta alurapic e com o CLI parado, vamos baixar o módulo com o comando:

npm install vue-router@2.1.1 --save

Dentro de instantes o módulo será baixado e armazenado dentro de alurapic/node\_modules. No entanto, baixar o módulo não é suficiente, precisamos registrá-lo. Para isso, vamos alterar alurapic/src/main.js e realizar um procedimento que já fizemos com o módulo VueResource:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

// importando o router!

import VueRouter from 'vue-router';

Vue.use(VueResource);

Vue.use(VueRouter);

// registrando o router

new Vue({

el: '#app',

render: h => h(App)

})

Excelente, importamos e registramos o módulo no global view object Vue. Agora, precisamos configurar as rotas da aplicação. Rotas? Você entenderá no próximo vídeo.

Para que o VueRouter saiba qual componente carregar precisamos registrar rotas para esses componentes. Rotas nada mais são que endereços especiais que são interceptados pelo VueRouter e a partir do endereço ele decidirá qual componente deve ser exibido em App que é o primeiro componente a ser exibido em nossa aplicação.

Vou estipular que os seguintes endereços serão válidos:

http://localhost:8080/#/

http://localhost:8080/#/cadastro

O primeiro carregará o componente Home e o segundo o componente Cadastro. Você deve estar achando estranho o #no endereço. Ele é importante, porque eles fazem com que o browser não dispare uma requisição para o servidor, pois não é uma URL válida. No entanto, sendo algo totalmente válido para o VueRouter, ele extrairá a informação que vem logo após o # para saber qual componente carregar. Ele faz um dê para entre o pedaço da url que vem logo após o #com o seu respectivo componente.

Sendo assim, é uma boa prática declarar as rotas da aplicação em um arquivo em separado. Vamos criar o arquivo alurapic/src/routes.js. Nele exportaremos uma constante que um um array:

// alurapic/src/routes.js

export const routes = [

/\* rotas aqui \*/

];

Quando queremos exportar o valor de uma variável é necessário usar o prefixo const. Agora, vamos importar os componentes Home e Cadastro que equivalem a páginas:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

No array routes, precisamos ter um objeto Javascript com as propriedades path e component. O primeiro é a caminho que identifica o componente, o segundo o componente que será carregado para este caminho presente na url do navegador:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

{ path: '', component: Home },

{ path: '/cadastro', component: Cadastro }

];

Veja que para o componente Home usamos o path como uma string em branco. Esse é o padrão quando queremos acessar o componente como /#/. Já path do componente Cadastro é /cadastro que se traduzirá em uma URL como http://localhost:8080/#/cadastro.

Mas inda falta mais duas configurações. A primeira, é passar as rotas que configuramos para o VueRouter. Para isso, vamos importar routes de routes.js:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

// tem que vir entre chaves, porque não é default

import { routes } from './routes';

Vue.use(VueRouter);

Vue.use(VueResource);

new Vue({

el: '#app',

render: h => h(App)

})

Agora que importamos a rota, vamos criar uma instância de VueRouter passando como parâmetro um objeto JavaScript com a propriedade routes que deve receber como parâmetro as rotas que importamos. No caso, tanto a propriedade quando as rotas importadas possuem o mesmo nome:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

import { routes } from './routes';

Vue.use(VueRouter);

const router = new VueRouter({

routes : routes

});

Vue.use(VueResource);

new Vue({

el: '#app',

render: h => h(App)

})

Em ES6, quando o valor e a propriedade possuem o mesmo nome, podemos simplesmente fazer assim:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

import { routes } from './routes';

Vue.use(VueRouter);

const router = new VueRouter({

routes

});

Vue.use(VueResource);

new Vue({

el: '#app',

render: h => h(App)

})

Agora que temos efetivamente nossas rotas, precisamos passará-la como parâmetro para a view instance, aquela que renderiza nosso componente App:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

import { routes } from './routes';

Vue.use(VueRouter);

const router = new VueRouter({

routes : routes

});

Vue.use(VueResource);

new Vue({

el: '#app',

router,

render: h => h(App)

})

Como o nome da propriedade tem o mesmo nome da nossa variável, podemos fazer apenas router ao invés de router: router.

Por fim, precisamos usar uma diretiva especial do VueRouter, uma que indica em que lugar do template de App os componentes serão carregados. Essa diretiva se chama router-view:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<router-view></router-view>

</div>

</template>

<script>

</script>

<style>

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

</style>

Quando o CLI rodando, veja que nossa aplicação será aberta automaticamente na URL http://localhost:8080/#/. Se quisermos acessar a página de cadastro, fazemos http://localhost:8080/#/cadastro.

Por fim, não há nada de errado com o # no endereço, é algo completamente válido e muito usado. No entanto, podemos removê-lo usando o modo history do VueRouter. No entanto, para este modo funcionar, seu backend que compartilha sua aplicação em Vue deve retornar sempre index.html para todos para qualquer endereço que chegar até ele, inclusive deve retornar index.html para páginas de erro. O Vue CLI já faz isso por padrão, mas se você for hospedar sua aplicação seja lá onde for, lembre-se desse detalhe.

Para ativarmos o modo history basta adicionarmos a propriedade mode com o valor history na instância de VueRouter.

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

import { routes } from './routes';

Vue.use(VueRouter);

// adicionando a propriedade mode com o valor history.

const router = new VueRouter({

routes,

mode: 'history'

});

Vue.use(VueResource);

new Vue({

el: '#app',

router,

render: h => h(App)

})

Veja que agora podemos acessar Home através de http://localhost:8080/ e Cadastro através de http://localhost:8080/cadastro. Esses endereços não dispararam uma requisição para o servidor e serão interceptados pelo VueRouter para saber qual componente carregar.

Vamos alterar alurapic/src/App.vue e adicionar um menu. Não vamos nos preocupar com estilização do menu, apenas um menu funcional para não perdemos o foco nesta parte tão importante:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<nav>

<ul>

<li><a href="/">Home</a></li>

<li><a href="/cadastro">Cadastro</a></li>

</ul>

</nav>

<router-view></router-view>

</div>

</template>

<script>

</script>

<style>

/\* código omitido \*/

</style>

Nosso menu é exibido e quando clicamos no link Cadastro somos direcionados para o componente Cadastro. Mas nem tudo esta perfeito. Veja que nossas navegações estão disparando o carregamento da página. Se estamos em uma Single Page Application, isso não deveria acontecer, pois já temos tudo o que precisamos carregado. O problema esta no uso da tag a para realizar a navegação. Para resolver esse problema, precisamos usar o componente router-link:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<nav>

<ul>

<li><router-link to="/">Home</router-link></li>

<li><router-link to="/cadastro">Cadastro</router-link></li>

</ul>

</nav>

<router-view></router-view>

</div>

</template>

<script>

</script>

<style>

/\* código omitido \*/

</style>

Agora sim! Veja que a página não recarregando enquanto clicamos nos itens do menu. Muito melhor! No entanto, se olharmos o arquivo alurapic/src/routes.js já temos a lista com todas as rotas definidas nesse arquivo. Lá temos o path de cada rota, mas não temos o título. Não tem problema, vamos adicionar a propriedade titulo no array de routes. Isso não causará nenhum erro devido a natureza dinâmica do JavaScript e nos permitirá importar esse array para criar nosso menu dinamicamente. Toda vez que uma nova rota for adicionada em routes, automaticamente ela aparecerá como item do menu:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

// adicionando a propriedade título

export const routes = [

{ path: '', component: Home, titulo: 'Home' },

{ path: '/cadastro', component: Cadastro, titulo: 'Cadastro' }

];

Agora, importando routes em App e disponibilizando a lista de rotas através da função data. Ah, desta vez vamos usar **in** no lugar de **for** na diretiva v-for. Eu prefiro **of**, mas se você vem do Angular 1 pode preferir o **in**:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<nav>

<ul>

<li v-for="route in routes">

<router-link :to="route.path ? route.path : '/'">{{route.titulo}}</router-link>

</li>

</ul>

</nav>

<router-view></router-view>

</div>

</template>

<script>

import { routes } from './routes';

export default {

data() {

return {

routes

}

}

}

</script>

<style>

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

</style>

Vocês devem estar estranhando a linha <router-link :to="route.path ? route.path : '/'">{{route.titulo}}</router-link>. Precisamos testar essa condição, porque o path de Home é um string vazia, mas quando usamos no componente router-link precisamos usar um /. Aliás, nosso menu é um forte candidato para se tornar um componente, mas não faremos isso agora.

Um das coisas mais bonitas do mundo da programação é podermos aplicar com conhecimento já adquirido e aplicá-lo em outros contextos. No capítulo anterior aprendemos a trabalhar com transições. Se refletirmos por um instante, a saída de uma página para a outra é uma mudança de estado, logo, passível de animação.

Vamos aplicar a mesma transição que fizemos com o componente Painel com a única diferença que será executada mais rapidamente. Aprendemos que precisamos envolver o componente desejado como alvo da transição através da tag transition. Sendo assim, qual componente você acha que devemos envolver para conseguir realizar transições entre páginas? O componente router-view em App!.

Alterando App:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<nav>

<ul>

<li v-for="route in routes">

<router-link :to="route.path ? route.path : '/'">

{{route.titulo}}

</router-link>

</li>

</ul>

</nav>

<transition name="pagina">

<router-view></router-view>

</transition>

</div>

</template>

<script>

import { routes } from './routes';

export default {

data() {

return {

routes

}

}

}

</script>

<style>

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.pagina-enter-active, .pagina-leave-active {

transition: opacity .3s

}

.pagina-enter, .pagina-leave-active {

opacity: 0

}

</style>

Veja que usamos os mesmos estilos, com a diferença que usamos como name do componente transitiono valor pagina e a transição foi reduzida para 200 milissegundos. Experimente clicar nos itens do menu da aplicação e veja a transição entre páginas sendo realizada. Se você domina CSS pode aplicar transições mais elaboradas, o mais importante é saber como integrá-las ao Vue.

Não precisamos meditar muito para ver que há algumas regrinhas para a criação do nosso menu, por isso ele é um forte candidado a ser tornar um componente em nossa aplicação. É exatamente isso que faremos.

Vamos criar o arquivo alurapic/src/components/shared/menu/Menu.vue. Ele receberá de um componente pai uma lista de rotas apenas. Vamos aproveitar e colocar uma validação dessa propriedade deixando explicito que aceitamos apenas o tipo Array:

<!-- alurapic/src/components/shared/menu/Menu.vue -->

<template>

<nav>

<ul>

<li v-for="rota in rotas">

<router-link :to="rota.path ? rota.path : '/'">{{rota.titulo}}</router-link>

</li>

</ul>

</nav>

</template>

<script>

export default {

props: {

rotas: {

type: Array,

required: true

}

}

}

</script>

<style scoped>

</style>

Eu preferi chamar internamente de rota ao invés de routes para não confundir com o array de rotas que se chama routes. Mas nada impede que vocês utilizem o mesmo nome.

Agora que temos nosso componente criado, vamos importá-lo em App, inclusive preciamos adicioná-lo na propriedade components para que seja acessível no template de App. Usaremos como nome meu-menu:

<!-- alurapic/src/App.vue -->

<template>

<div class="corpo">

<meu-menu :rotas="routes"/>

<transition name="pagina">

<router-view></router-view>

</transition>

</div>

</template>

<script>

import { routes } from './routes';

import Menu from './components/shared/menu/Menu.vue';

export default {

components: {

'meu-menu' : Menu

},

data() {

return {

routes

}

}

}

</script>

<style>

.corpo {

font-family: Helvetica, sans-serif;

margin: 0 auto;

width: 96%;

}

.pagina-enter-active, .pagina-leave-active {

transition: opacity .3s

}

.pagina-enter, .pagina-leave-active {

opacity: 0

}

</style>

Excelente, quando a página é recarregada tudo continua funcionando como antes. Agora que temos o menu isolado em um componente, fica mais fácil aplicar aquele estilo profissional ao menu. Mas isso eu deixarei para os meus alunos mestres em CSS.

Neste capítulo aprendemos:

* a criar um novo componente/página
* o conceito de Single Page Application
* extrair de App.vue um componente de página, separando responsabilidades
* a baixar e registrar o módulo VueRouter
* criar rotas da aplicação
* o componente router-view
* o modo history vs hash
* a criar um menu baseado em nossas configurações de rota
* a tornar o menu um componente
* executar efeito de transição na troca de páginas
* a validar propriedades em um componente

Como já configuramos as rotas da aplicação, nada nos impede de partirmos para nossa página de cadastro. No entanto, podemos expandir ainda mais nosso conhecimento sobre criação de componentes.

Em breve, precisaremos implementar a funcionalidade de exclusão de fotos. Vamos deixar preparado o terreno adicionando um botão com o título remover, mas espere um pouco. Quais são as características desse botão?

O botão deve ter a cor vermelha e letra branca, porque remoção é algo sério e o vermelho chama atenção. Ele deve ter um rótulo, ou seja, um nome. Além disso, precisamos indicar seu tipo, se vai ser button ou submit por exemplo. Por fim, queremos que esse botão exiba uma janela de confirmação da operação. Agora eu te pergunto, e se quisermos o mesmo botão na cor azul e sem confirmação? Teremos que criar outro botão.

Você já deve ter percebido que o botão é um forte candidato para tornamos um componente. Ele será bastante configurável e será usado não apenas em Home, mas em Cadastro também em breve.

Vamos criar o componente alurapic/src/shared/botao/Botao.vue:

<!-- alurapic/src/components/shared/botao/Botao.vue -->

<template>

<button class="botao botao-perigo" :type="tipo">{{rotulo}}</button>

</template>

<script>

export default {

props: ['tipo', 'rotulo'],

}

</script>

<style scoped>

.botao {

display: inline-block;

padding: 10px;

border-radius: 3px;

margin: 10px;

font-size: 1.2em;

}

.botao-perigo {

background: firebrick;

color: white;

}

.botao-padrao {

background: darkcyan;

color: white;

}

</style>

Vamos por partes. Primeiro, vamos fazer com que o botão receba apenas como parâmetro do componente pai o seu tipo e rótulo. Por enquanto, vamos deixar fixo a classe botao-perigo.

Agora que já temos o componente criado, vamos importá-lo em Home e utilizá-lo dentro do painel, logo abaixo do nosso componente de imagem responsiva:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="titulo">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

<meu-botao rotulo="remover" tipo="button"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

// Fazendo o import do botão. Não esqueça de adicioná-lo em components

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'meu-painel': Painel,

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

// código omitido

}

</script>

<style>

/\* código omitido \*/

</style>

Isso já é suficiente para que nosso botão seja exibido.

Agora, se quisermos executar a lógica de exclusão, precisamos criar um método no componente pai Home que será chamada assim que o botão for clicado:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

<meu-botao rotulo="remover" tipo="button" @click="remove()"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

// Fazendo o import do botão. Não esqueça de adicioná-lo em components

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'meu-painel': Painel,

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

methods: {

remove() {

alert('Precisa saber qual foto remover!');

}

}

// código omitido

}

</script>

<style>

/\* código omitido \*/

</style>

Veja que associamos a chamada do método remove declarada em methods através do evento click usando o atalho @click para binding deste tipo de evento. Mas quando clicamos no botão, nenhum evento é disparado!

## Disparando eventos nativos

Isso acontece, porque o componente que criamos é uma caixa preta e só podemos lidar com o que ele oferece. No entanto, podemos usar o modificador .native no evento clique para que o evento click, nativo de toda tag do mundo HTML seja disparado. Isso parece que vai resolver nosso problema:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

<meu-botao rotulo="remover" tipo="button" @click.native="remove()"/>

<!-- código posterior omitido -->

Agora, quando clicamos no botão, nosso alert é exibido. Excelente, mas precisamos saber qual foto clicamos para futuramente realizarmos um pedido à nossa API para que a delete.

Como o nosso botão esta sendo construindo dentro de um elemento com a diretiva v-for, podemos passar o elemento que esta sendo iterado na lista diretamente para o método. Lembre-se que usamos o nome foto para referenciar cada elemento da lista. É este nome que passaremos para o método remove:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

<!-- passando foto como parâmetro do método remove do componente Home -->

<meu-botao rotulo="remover" tipo="button" @click.native="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

// Fazendo o import do botão. Não esqueça de adicioná-lo em components

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'meu-painel': Painel,

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

methods: {

remove(foto) {

// exibindo o título da foto selecionado

alert(foto.titulo);

}

}

// código omitido

}

</script>

<style>

/\* código omitido \*/

</style>

Excelente, quando clicamos no botão remove de cada foto vemos o título da foto correspondente sendo exibida. Mas vocês devem lembrar que precisamos confirmar a exclusão. Nesse sentido, podemos usar a função confirm para conseguir rapidamente essa funcionalidades:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

methods: {

remove(foto) {

if(confirm('Confirma?')) {

alert(foto.titulo);

}

}

}

<!-- código posterior omitido -->

Perfeito, só exibimos o alerta se confirmarmos. Mas pensem comigo. Todo lugar que precisarmos de uma confirmação teremos que ter esse código que chama o confirm. E se no lugar do confirm fosse exibido um model bonitão? Teríamos que repetir esse código em vários lugares. A ideia é colocar o código de confirmação no próprio botão, além disso, podemos ativar ou não a confirmação configurando nosso componente para tornar o botão ainda mais reutilizável. Como faremos isso? É o que veremos no próximo vídeo!

Antes de continuarmos precisamos entender o seguinte. O evento clique do nosso botão chama um método no componente pai Home. Agora, precisamos que o código de confirmação seja executado pelo nosso botão, mas a lógica de execução deve ser feita no elemento pai. Em suma, nosso elemento filho precisa de alguma maneira chamar um método do seu elemento pai.

Para isso, precisamos trabalhar com eventos customizados. Por exemplo, não implementamos ainda esse evento, mas quando clicamos no botão e o evento botaoAtivado for disparado, o método remove recebendo a foto deve ser chamado. Não preciso dizer que botaoAtivado será o evento que criaremos. Como todo evento, podemos associá-lo a um elemento usando o atalho @:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

<meu-botao rotulo="remover" tipo="button" @botaoAtivado="remove(foto)"/>

<!-- código anterior omitido -->

A ideia é o seguinte. Quando nosso componente Botao for clicado, ele disparará um o evento customizado botaoAtivado, mas apenas se o usuário confirmar. Quando esse evento for disparado, o método associado ao nome do nosso evento será executado.

Agora, precisamos alterar Botao para que dispare nosso evento toda vez que for clicado. Para isso, no seu template, vamos adicionar o evento click que chamará um método do componente. É nele que realizamos a lógica de confirmação disparando o evento ou não:

<!-- alurapic/src/components/shared/botao/Botao.vue -->

<template>

<button class="botao botao-perigo" :type="tipo" @click="disparaAcao()">{{rotulo}}</button>

</template>

<script>

export default {

props: ['tipo', 'rotulo'],

methods: {

disparaAcao() {

if(confirm('Confirma operacao?')) {

this.$emit('botaoAtivado');

}

}

}

}

</script>

<style scoped>

.botao {

display: inline-block;

padding: 10px;

border-radius: 3px;

margin: 10px;

font-size: 1.2em;

}

.botao-perigo {

background: firebrick;

color: white;

}

.botao-padrao {

background: darkcyan;

color: white;

}

</style>

É através de this.$emit que disparamos um evento customizado passando como nome do parâmetro o evento.

Com o CLI rodando, nosso projeto com certeza foi recarregado pelo seu navegador, inclusive já podemos realizar nosso teste. Quando clicamos em remover, o diálogo de confirmação é exibido. Se não confirmamos, nada é exibido na tela, quando confirmamos, o alerta com o título da foto é exibido.

Apesar de funcionar nosso botão ainda esta incompleto. Precisamos fazer com que o componente possa confirmar ou não uma ação, inclusive aplicar a classe correta de acordo com seu estilo. é isso que veremos no próximo vídeo.

Vamos fazer com que nosso componente que representa nosso botão receba como parâmetro confirmacao. Se receber true, a confirmação será efetuada, se receber false, nenhum confirmação será feita:

<!-- alurapic/src/components/shared/botao/Botao.vue -->

<template>

<button class="botao botao-perigo" :type="tipo" @click="disparaAcao()">{{rotulo}}</button>

</template>

<script>

export default {

props: ['tipo', 'rotulo', 'confirmacao'],

methods: {

disparaAcao() {

if(this.confirmacao) {

if(confirm('Confirma operacao?')) {

this.$emit('botaoAtivado');

}

return;

}

this.$emit('botaoAtivado');

}

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Agora, em Home, podemos adicionar confirmacao="false" em noss componente para que nenhuma confirmação seja realizada:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="titulo">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

<meu-botao

rotulo="remover"

tipo="button"

confirmacao="false"

@botaoAtivado="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

export default {

// código omitido

}

</script>

<style>

/\* código omitido \*/

</style>

Não funciona! Veja que mesmo com confirmacao="false" a confirmação é exibida. Qual a razão disso? O problema é que não fizemos um data binding entre com a propriedade confirmacao, para tal, precisamos adicionar v-bind: ou simplesmente : antes do nome da propriedade. Sem o binding, o valor é passado uma única vez para dentro do componente como texto e não como referência. Em JavaScript qualquer texto é considerado true e por isso o alerta é sempre exibido.

Podemos verificar o tipo dentro do componente Botao da seguinte maneira:

<!-- alurapic/src/components/shared/botao/Botao.vue -->

<template>

<button class="botao botao-perigo" :type="tipo" @click="disparaAcao()">{{rotulo}}</button>

</template>

<script>

export default {

props: ['tipo', 'rotulo', 'confirmacao'],

methods: {

disparaAcao() {

/ exibindo o tipo da propriedade. Sem o : será string, com : será boolean

console.log(typeof(this.confirmacao));

if(this.confirmacao) {

if(confirm('Confirma operacao?')) {

this.$emit('botaoAtivado');

}

return;

}

this.$emit('botaoAtivado');

}

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Agora, quando usamos : o valor passado deixa de ser uma string e passa a ser a expressão, no caso false, um booleano!

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="titulo">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

<meu-botao

rotulo="remover"

tipo="button"

:confirmacao="false"

@botaoAtivado="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

// código posterior omitido

Faça um teste e veja que agora a confirmação não é exibida. Só não esqueça de colocar confirmacao="true" novamente, pois nesse cenário faz sentido pedirmos a confirmação do usuário.

Agora, precisamos lidar com o estilo do botão. Vamos adicionar a propriedade estilo. Se o seu valor for padrão, usaremos a classe botao botao-padrao, no entanto, se for perigo, usaremos a classe botao botao-perigo. Primeiro, vamos adicionar a propriedade em Botao para que ele aceite recebê-la:

<!-- alurapic/src/components/shared/botao/Botao.vue -->

<template>

<button :class="estiloDoBotao" :type="tipo" @click="disparaAcao()">{{rotulo}}</button>

</template>

<script>

export default {

props: {

tipo: {

required: true,

type: String

},

rotulo: {

required: true,

type: String

},

confirmacao: {

required: false,

default: false,

type: Boolean

},

estilo: {

required: false,

default: 'padrao',

type: String

}

},

// código posterior omitido

Agora, em Home, vamos passar a propridade estilo="padrao" para que nosso botão seja apresentado com o estilo padrão, ou seja, numa cor azul clara:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="titulo">Alurapic</h1>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto in fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

<meu-botao

rotulo="remover"

tipo="button"

:confirmacao="true"

@botaoAtivado="remove(foto)"

estilo="padrao"/>

</meu-painel>

</li>

</ul>

</div>

</template>

Agora, em Botao, a classe da tag button deve receber um ou outro estilo de acordo com o parâmetro passado para estilo. Vamos usar uma computed property para isso. Nosso Botao final fica assim:

<!-- alurapic/src/components/shared/botao/Bota.vue -->

<template>

<button :class="estiloDoBotao" :type="tipo" @click="disparaAcao()">{{rotulo}}</button>

</template>

<script>

export default {

props: {

tipo: {

required: true,

type: String

},

rotulo: {

required: true,

type: String

},

confirmacao: {

required: false,

default: false,

type: Boolean

},

estilo: {

required: false,

default: 'padrao',

type: String

}

},

methods: {

disparaAcao() {

console.log(typeof(this.confirmacao));

if(this.confirmacao) {

if(confirm('Confirma operacao?')) {

this.$emit('botaoAtivado');

}

return;

}

this.$emit('botaoAtivado');

}

},

computed: {

estiloDoBotao() {

// se o valor é padrão ou não passou nada para estilo

if(this.estilo == 'padrao') return 'botao botao-padrao';

if(this.estilo == 'perigo') return 'botao botao-perigo';

}

}

}

</script>

<style scoped>

.botao {

display: inline-block;

padding: 10px;

border-radius: 3px;

margin: 10px;

font-size: 1.2em;

}

.botao-perigo {

background: firebrick;

color: white;

}

.botao-padrao {

background: darkcyan;

color: white;

}

</style>

Recarregando a página, vamos que nosso botão esta na cor azul. Sabemos que isso foi apenas um teste, pois seu estilo deve ser perigo. Alterando:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

<meu-botao

rotulo="remover"

tipo="button"

:confirmacao="true"

@botaoAtivado="remove(foto)"

estilo="perigo"/>

<!-- código posterior omitido -->

Excelente, temos mais um componente reutilizável que nos será útil ao longo do treinamento.

Terminamos mais um componente. Ponto para nós. No entanto, vimos que se o programador esquecer de usar o dois pontos na propriedade confirmacao do nosso Botao lá em Home o valor será passado como string fazendo com que a confirmação não funcione direito. A boa notícia é que podemos indicar o tipo aceito por uma propdentro de um componente, inclusive se o seu valor é obrigatório. Nossa aplicação continua dando erro, mas a mensagem no console deixa claro que determinada regra foi ferida.

Vamos alterar Botao. Nela, todas as propriedades são tipo String, exceto confirmacaoque é boolean. Além disso, confirmacao e estilos não são propriedades obrigatórias:

<!-- alurapic/src/components/shared/botao/Bota.vue -->

<template>

<button :class="estiloDoBotao" :type="tipo" estilo="perigo" @click="disparaAcao()">{{rotulo}}</button>

</template>

<script>

export default {

props: {

tipo: {

type: String,

required: true

},

rotulo: {

type: String,

required: true

},

confirmacao: Boolean,

estilo: String

},

methods: {

/\* código omitido \*/

},

computed: {

/\* código omitido \*/

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Agora, se voltarmos para Home e removermos : da propriedade confirmacao será exibida a seguinte mensagem no console do Chrome:

vue.common.js?e881:519[Vue warn]: Invalid prop: type check failed for prop "confirmacao". Expected Boolean, got String.

(found in component <meu-botao> at Botao.vue)

Muito mais fácil de detectar o problema, não? Só não se esqueça de voltar com os dois pontos lá na propriedade que modificamos por último.

Neste capítulo aprendemos:

* a criar um novo componente reutilizável
* a criar nossos próprios eventos
* a aplicar classes condicionalmente
* a diferença do valor passado para a propriedade de um componente sem bind e com bind
* a validar propriedades de componentes

Curso 2

Até o momento, aprendemos a criar componentes reutilizáveis extremamente úteis. No entanto, o meu designer, que vocês já devem ter percebido que é meu amigo imaginário, me pediu para implementar a seguinte funcionalidade em nossa aplicação.

Toda vez que dermos um duplo clique na imagem do painel ela será rotacionada 90 graus. Mas ele esta na dúvida se gira imagem mesmo ou o próprio painel como um todo. Sendo assim, precisaremos fazer a rotação com os dois componentes e verificar em qual deles o efeito fica mais interessante.

Podemos quebrar a cabeça um pouco e criar métodos ou computed properties em nossos componentes, inclusive trabalhar com eventos para implementar essa solução. Se vamos conseguir não tenho certeza, mas mesmo que consigamos se quisermos usar essa mesma funcionalidade em outros componentes precisaremos repetir seu código o que torna a manutenção da aplicação mais complicada. Par solucionar problemas como esse o Vue possui as diretivas.

## Diretivas vs Componentes

Diretivas são blocos de código isolados que operam sobre componentes e que manipulam o DOM apenas, diferente de componentes que são unidades confinadas de código que possuem sua apresentação e lógica. Inclusive, já usamos algumas diretivas do Vue como v-for, v-show entre outras.

Talvez você tenha torcido o nariz quando escutou a frame "manipulação de DOM", pois é exatamente a manipulação de DOM que o Vue tenta evitar com seus mecanismos de data binding. No entanto, esse tipo de manipulação é totalmente permitida dentro das diretivas.

Vamos iniciar os trabalhos criando o arquivo alurapic/src/directives/Transform.js. Veja que o arquivo não é um single file template com a extensão .vue, mas um arquivo .js como outro script qualquer.

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Veja que a primeira instrução da nossa diretiva é a importação do global view object Vue. É através dele que registraremos nossa diretiva para que seja acessível pelas views da nossa aplicação.

O próximo passo é invocar o método Vue.directive que recebe como parâmetro o nome da diretiva e como segundo um objeto JavaScript que conterá as configurações da diretiva:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

/\* ainda sem qualquer configuração \*/

});

Nossa diretiva ainda não esta completa, mas quando formos utilizá-la em nossos template somos obrigados a utilizar o prefixo v-. Veja que diretivas como v-for e v-show possui o mesmo prefixo. Sendo assim, para utilizarmos nossa diretiva precisaremos fazer v-meu-transform.

## A função bind e acesso ao elemento do DOM

Agora que você já entendeu como usaremos a diretiva que estamos criando em nossos templates podemos partir para sua configuração. O objeto passado como segundo parâmetro precisa ter o método bind, um hook chamado toda vez que a diretiva é associada ao elemento do DOM. O método recebe três parâmetros. O nome desses três parâmetros podem ser qualquer um, no entanto vamos utilizar aqueles que deixaram claro o papel de cada um:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

}

});

Por enquanto, vamos no ater apenas ao primeiro parâmetro da diretiva, que nada mais é do que o elemento do DOM no qual a diretiva esta sendo aplicada. Vamos imprimir no console esse parâmetro:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

console.log('Diretiva iniciada');

console.log(el);

}

});

Por enquanto vamos deixar essa implementação e experimentar utilizá-la em nossa aplicação.

## Importando nossa diretiva

Em nenhum momento carregamos o módulo Transform.js. Todo arquivo .js e .vuecriados são módulos que precisam ser importados. Faremos isso em nosso arquivo alurapic/src/main.js:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

import { routes } from './routes';

// importando o arquivo `Transform.js`.

import './directives/Transform';

Vue.use(VueRouter);

// código posterior omitido

Veja que a cláusula import precisa receber apenas o caminho do arquivo. É desta forma porque o módulo Transform.js não exporta código algum, precisamos apenas carregá-lo para que nossa diretiva seja registrada através do global view object. Excelente! Isso já é suficiente para que nossa diretiva esteja ativa.

Agora, vamos abrir alurapic/src/components/home/Home.vue e adicionar a diretiva no elemento <imagem-responsiva/>:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform/>

<!-- código posterior omitido

Não esqueça que o Vue CLI precisa estar de pé para que possamos visualizar o resultado em nosso navegador. Assim que a página for atualizada, abra o console do navegador e veja o resultado. Veja que para cada componente ele exibirá o elemento do DOM correspondente àquele componente.

Agora que já vimos que nossa diretiva esta atuando para cada componente ImagemResponsiva criado dinamicamente através da diretiva v-for podemos dar início a implementação da lógica de rotação.

Um ponto importante é que cada diretiva é aplicada isoladamente para cada elemento, sendo assim, cada diretiva pode ter suas variáveis de controle específicas para cada elemento. No caso, vamos declarar a variável current que será iniciada assim que nossa diretiva for aplicada no elemento. Ela guardará o eixo atual utilizado para rotacionarmos o elemento. Todos começarão de zero:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

}

});

## Implementando nossa lógica

A cada duplo clique que realizamos, vamos incrementar a variável current em 90graus. Mas como trabalharemos com evento dentro da nossa diretiva? Se fosse no componente, poderíamos usar @dblclick no template, mas veja que aqui não temos template. A boa notícia é o fato de el ser o elemento do DOM no qual a diretiva esta associada e, sendo um elemento do DOM, podemos adicionar evento da maneira tradicional no mundo JavaScript através de addEventListener:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

current+=90;

});

}

});

A cada clique o valor de current da imagem que estamos clicando será incrementado. Agora, basta aplicamos uma transformação do tipo rotate do CSS para conseguimos o evento desejado. Via DOM, aplicamos um estilo diretamente no elemento através de elemento.style.nomeDoPropriedade:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

current+=90;

this.style.transform = `rotate(${current}deg)`;

});

}

});

Aqui usamos template string do ES2015 para nos ajudar a construir a string dinamicamente com base no valor atual da variável current. Muito mais interessante do que se tivéssemos feito uma concatenação. Templates strings são criadas usando crase (backstick) e no lugar da concatenação, usamos ${nomeDaVariavel} para interpolar dentro da string valores de variáveis. Se você quer saber mais sobre ES2015 e template strings, sugiro cursar meu curso de JavaScript avançado aqui da Alura.

Vamos testar o resultado? Com a página recarregada, quando clicamos duas vezes nas imagens elas são rotacionadas de 90 em 90 graus. No diretiva é tão genérica que podemos adicioná-la ao título da página, permitindo sua rotação no duplo clique:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado" v-meu-transform>Alurapic</h1>

<!-- código posterior omitido -->

Faça um teste e veja o resultado!

Excelente. Mas se quisermos incrementar de 50 em 50? Para isso nossa diretiva precisará receber a quantidade em graus do incremento. É isso que veremos no próximo vídeo.

Podemos passar parâmetros para nossa diretiva. No caso, vamos fazer com que a rotação de cada imagem seja feita de 15 em 15 graus:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform="15"/>

<!-- código posterior omitido -->

Veja que atribuímos o valor 15 à nossa diretiva no template de Home.vue. Agora, no código de nossa diretiva, precisamos ter acesso a esse valor. Isso é possível através de binding.value:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 90;

current+=incremento;

this.style.transform = `rotate(${current}deg)`;

});

}

});

Se não for passado parâmetro algum, usaremos 90 por padrão. Caso seja, utilizaremos o valor passado como parâmetro para rotacionar nosso elemento. Faça um teste. Veja que a imagem é incrementa de 15 em 15 graus e nosso título de 90 em 90, porque não passamos nenhum parâmetro.

Para ficar ainda melhor, podemos executar uma transição na rotação:

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 90;

current+=incremento;

this.style.transition = "transform 0.5s";

this.style.transform = `rotate(${current}deg)`;

});

}

});

Tudo funciona perfeitamente. Mas se quisermos alterar o tempo de transição? Veja que usamos o valor fixo de meio segundo. Nossa diretiva já recebe um parâmetro que é a quantidade de graus que desejamos rotacionar, e agora?

## Passando mais de um parâmetro

Podemos passar um objeto JavaScript como parâmetro para a diretiva no qual cada propriedade é um parâmetro da diretiva. Alterando Home.vue:

<!-- alurapic/src/components/home/Home.vue -->

<!-- código anterior omitido -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform="{incremento: 15, animate: true}"/>

<!-- código posterior omitido -->

Agora, alterando a lógica do nosso hook binding:

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = 90;

let animate = false;

if(binding.value) {

incremento = binding.value.incremento

animate = binding.value.animate

}

current+=incremento;

this.style.transform = `rotate(${current}deg)`;

if (animate) this.style.transition = "transform 0.5s";

});

}

});

Apesar de funcional, podemos fazer um pouquinho diferente. É isso que veremos no próximo vídeo.

Podemos deixar nossa diretiva mais bem organizada. Por exemplo, hoje passamos um objeto com dois parâmetros para nossa diretiva. Mas veja que um parâmetro é usado para controlarmos se realizamos animações ou não. Nesse caso, podemos usar um modificador com esta finalidade.

Por exemplo, nossa diretiva em uso ficará assim:

<!-- alurapic/src/components/home/Home.vue -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform.animate="15"/>

Veja que usamos o nome da diretiva seguido de ponto e o nome do modificador que estamos usando.

Agora, na declaração da nossa diretiva, podemos acessar binding.modifiers que é um objeto JavaScript cuja as propriedades são nossos modificadores.

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 90;

current+=incremento;

this.style.transform = `rotate(${current}deg)`;

if (binding.modifiers.animate) this.style.transition = "transform 0.5s";

});

}

});

O valor de binding.modifiers.animate é sempre true quando o modificador estiver presente. Podemos ter mais de um modificador se assim desejarmos. Por exemplo, vamos adicionar o modificador reverse para que a imagem gire em sentido anti horário:

<!-- alurapic/src/components/home/Home.vue -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform.animate.reverse="15"/>

Alterando o código da nossa diretiva para levar em consideração o novo modificador:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 90;

if(binding.modifiers.reverse) {

current-=incremento;

} else {

current+=incremento;

}

this.style.transform = `rotate(${current}deg)`;

if (binding.modifiers.animate) this.style.transition = "transform 0.5s";

});

}

});

Excelente, nossa diretiva de transformação funciona rotacionando qualquer elemento na tela. Aliás, você a adicionou no meu-painel ao invés de utiliza-la em imagem-responsiva? Bem, eu prefiro seu uso em imagem-responsiva e o design, amigo imaginário, também.

Mas vejamos o seguinte. E se quisermos, no lugar de rotacionar a imagem quisermos ampliá-la? Veja que o modificador animacao faz sentido, já o reverse, não muito. Para isso, precisamos trabalhar com argumentos.

Vamos alterar a chamada da nossa diretiva no template de Home:

<!-- alurapic/src/components/home/Home.vue -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:rotate.animate.reverse="15"/>

Veja que logo após o nome da diretiva foi adicionando dois ponto seguido do nome do parâmetro. No caso, estamos passando rotate. Aliás, esse será o efeito padrão quando nenhum argument for passado. Agora, vamos alterar nossa diretiva para levar em consideração esse argumento:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 0;

let efeito;

if(!binding.arg || binding.arg == 'rotate') {

if(binding.modifiers.reverse) {

current-=incremento;

} else {

current+=incremento;

}

efeito = `rotate(${current}deg)`;

}

this.style.transform = efeito;

if (binding.modifiers.animate) this.style.transition = "transform 0.5s";

});

}

});

É através de binding.arg que verificamos a existência ou não de um argumento. Um teste indica que tudo continua funcionando. Por fim, vamos adicionar a condição que aumenta o tamanho da imagem:

// alurapic/src/directives/Transform.js

import Vue from 'vue';

Vue.directive('meu-transform', {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 0;

let efeito;

if(!binding.arg || binding.arg == 'rotate') {

if(binding.modifiers.reverse) {

current-=incremento;

} else {

current+=incremento;

}

efeito = `rotate(${current}deg)`;

} else if(binding.arg == 'scale') {

efeito = `scale(${incremento})`;

}

this.style.transform = efeito;

if (binding.modifiers.animate) this.style.transition = "transform 0.5s";

});

}

});

Veja que o modificador reverse será ignorado quando o argumento for scale. Isso não é feio não, podemos ter vários modificadores que se aplicam a um tipo de argumento. Outro ponto que mudamos é o valor padrão de 90 para 0. Não faz sentido usar um scale com 90!

Hoje, nosso componente esta assim:

<!-- alurapic/src/directives/Transform.js -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:rotate.animate.reverse="30"/>

Agora, vamos mudar para scale. Podemos remover o modificador reverse já que ele não terá afeito algum quando usarmos scale:

<!-- alurapic/src/directives/Transform.js -->

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:scale.animate="1.2"/>

Podemos também não usar animação omitindo o modificador animate e por ai vai. Por fim, para que a imagem rotacionada e ampliada não fazer de dentro do nosso painel podemos alterar nosso componente Painel e adicionar o estilo overflow: hidden:

<!-- alurapic/src/components/painel/Painel.vue -->

<template>

<!-- código omitido -->

</template>

<script>

export default {

// código omitido

}

</script>

<style>

/\* estilos anteriores omitidos \*/

.painel-conteudo {

overflow: hidden;

}

</style>

O mais bacana é que essa alteração no estilo do nosso componente se refletirá em todos os lugares que ele for utilizado.

Vimos que a função bind de uma diretiva recebe três parâmetros. No entanto, o terceiro, vnode, não foi usado. Este parâmetro é uma referência para o Virtual DOM utilizado pelo Vue. Ele permite acessar propriedades do Componente no qual ela foi associada.

No entanto, essa dependência pode acabar com o reuso da diretiva, por isso deve ser usado com parcimônia. Além disso, não é comum querermos interagir com esse parâmetro.

Vejamos um exemplo:

import Vue from 'vue';

Vue.directive('calopsita', ()=> {

bind(el, binding, vnode) {

console.log(vnode.context.propriedadeDoComponenteNoQualADiretivaFoiAssociada);

}

});

Hoje, em nosso arquivo Transform.js registramos diretamente nossa diretiva através do *global view object*. No entanto, podemos ter maior controle do que utilizarmos em nossa aplicação explicitando a importação da diretiva no próprio componente.

Para isso, vamos alterar o arquivo alurapic/src/directives/transform.js para:

export default {

bind(el, binding, vnode) {

let current = 0;

el.addEventListener('dblclick', function() {

let incremento = binding.value || 90;

let efeito;

if(!binding.arg || binding.arg == 'rotate') {

if(binding.modifiers.reverse) {

current-=incremento;

} else {

current+=incremento;

}

efeito = `rotate(${current}deg)`;

} else if(binding.arg == "scale") {

efeito = `scale(${incremento})`;

}

el.style.transform = efeito;

if(binding.modifiers.animate) el.style.transition = 'transform 0.5s';

});

}

};

O módulo Transform agora exporta um objeto que contém as configurações da nossa diretiva. Inclusive veja que nenhum nome foi definido para ela.

Por fim, precisamos alterar o componente alurapic/src/components/home/Home.vue para importar nossa diretiva e adicioná-la ao componente:

<template>

<!-- código omitido -->

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue';

import Botao from '../shared/botao/Botao.vue';

// importou diretiva. Tem que adicionar na propriedade directives logo abaixo!

import transform from '../../directives/Transform';

export default {

components: {

'meu-painel' : Painel,

'imagem-responsiva': ImagemResponsiva,

'meu-botao' : Botao

},

directives: {

'meu-transform': transform

},

// código posterior omitido

Essa forma permite darmos o apelido que desejarmos para a diretiva em nossos componentes.

Chegou a hora de continuarmos a implementar nosso componente Cadastro, aquele pelo qual o usuário poderá cadastrar novas fotos. Sugiro que você vá até a explicação do capítulo e copie o código do nosso componente Cadastro atualizado. Ele possui um formulário já prontinho e estilizado para que possamos começar os trabalhos rapidamente.

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado"></h2>

<form>

<div class="controle">

<label for="titulo">TÍTULO</label>

<input id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input id="url" autocomplete="off">

<imagem-responsiva/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea id="descricao" autocomplete="off"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

}

}

</script>

<style scoped>

.centralizado {

text-align: center;

}

.controle {

font-size: 1.2em;

margin-bottom: 20px;

}

.controle label {

display: block;

font-weight: bold;

}

.controle label + input, .controle textarea {

width: 100%;

font-size: inherit;

border-radius: 5px

}

.centralizado {

text-align: center;

}

</style>

Visualize o componente em seu navegador clicando no menu Cadastro. Não há muita novidade, a não ser o envolvimento do componente Botao com o título "VOLTAR" pelo componente router-link para que cliques no botão gerem uma navegação para Home. O primeiro passo é aprendermos a capturar os dados inseridos pelo usuário para depois enviá-los para nossa API.

Agora que já temos nosso formulário, precisamos capturar a entrada do usuário para que possamos enviá-la para a nossa API responsável pelo cadastro. O primeiro passo é adicionar no objeto retornado pela função data do nosso componente a propriedade foto. Seu valor será um objeto com as propriedades titulo, url e descricao:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código omitido -->

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data() {

return {

foto: {

titulo: '',

url: '',

descricao: ''

}

}

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Agora, precisamos de alguma maneira fazer com que cada entrada do usuário em nosso formulário seja gravada na sua respectiva propriedade do objeto foto que acabamos de criar. Já aprendemos a realizar um data binding da view para a fonte de dados. Vamos usar o evento @input em cada campo do nosos formulário:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado"></h2>

<form @submit.prevent="grava()">

<div class="controle">

<label for="titulo">TÍTULO</label>

<input @input="foto.titulo = $event.target.value" id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input @input="foto.url = $event.target.value" id="url" autocomplete="off">

<imagem-responsiva/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea @input="foto.descricao= $event.target.value" id="descricao" autocomplete="off"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<!-- código posterior omitido -->

Agora que realizamos uma associação de cada campo do formulário com cada propriedade do nosso objeto foto podemos fazer o seguinte teste. Quando clicarmos no botão "GRAVAR" vamos exibir no console nosso objeto foto. Como ele já terá recebido os dados do formulário poderemos visualizar esses dados. Dessa forma, teremos a garantia que tudo esta funcionando para depois enviarmos esses dados para nossa API.

No mundo JavaScript, o evento submit é disparado toda vez que um formulário é submetido. Sendo assim, vamos criar o método grava em nosso componente Cadastroque será chamado para este evento. É nele que exibiremos os dados do nosso objeto no console:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado"></h2>

<!-- associando o evento com método do componente -->

<form @submit="grava()">

<div class="controle">

<label for="titulo">TÍTULO</label>

<input @input="foto.titulo = $event.target.value" id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input @input="foto.url = $event.target.value" id="url" autocomplete="off">

<imagem-responsiva/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea id="descricao" autocomplete="off" @input="foto.titulo = $event.target.value"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data() {

return {

foto: {

titulo: '',

url: '',

descricao: ''

}

}

},

methods: {

grava() {

console.log(this.foto);

this.foto = {};

}

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Parece que ainda falta algo. Quando clicamos no botão "GRAVAR", o formulário é submetido e nossa aplicação recarregada. Isso não faz sentido em uma SPA, além disso, pelo fato da aplicação recarregar não conseguimos ver o resultado no console. A boa notícia é que podemos usar um event modifier. O Vue nos disponibiiza o modificador prevent que equivale ao event.preventDefault() padrão do JavaScript que cancela o comportamento padrão do evento, em nosso caso, o comportamento padrão do evento submit é submeter o formulário:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado"></h2>

<form @submit.prevent="grava()">

<!-- código posterior omitido -->

Agora sim! Quando clicamos no botão e o formulário é submetido nossa aplicação não recarrega e o método grava é chamado exibindo no console os dados do objeto fotoque foram alimentados através do formulário.

No entanto, se vocês estão bem atentos, devem ter percebido que o formulário continua preenchido. Isso pode confundir o usuário fazendo-o pensar que a operação não funcionou.

Podemos resolver isso facilmente no método grava atribuindo para a propriedade foto um novo objeto com as propriedade titulo, url e descricao em branco:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado"></h2>

<form @submit.prevent="grava()">

<div class="controle">

<label for="titulo">TÍTULO</label>

<input @input="foto.titulo = $event.target.value" id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input @input="foto.url = $event.target.value" id="url" autocomplete="off">

<imagem-responsiva/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea id="descricao" autocomplete="off" @input="foto.titulo = $event.target.value"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data() {

return {

foto: {

titulo: '',

url: '',

descricao: ''

}

}

},

methods: {

grava() {

console.log(this.foto);

this.foto = {

titulo: '',

url: '',

descricao: ''

};

}

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Nenhum erro aconteceu, mas também o formulário não foi limpo. O que houve? A resposta esta no tipo de data binding que foi realizado. Usamos um data binding de eventos que vai da view para a fonte de dados, mas não fizemos o data binding inverso que vai da fonte de dados para view. Por isso nossa view não foi atualizada quando alteramos a propriedade foto. Aprendemos a realizar este último tipo de data binding através de v-bind ou seu atalho :. Vamos fazer uma asssociação que vai da fonte de dados para a a view através de :value:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado"></h2>

<form @submit.prevent="grava()">

<div class="controle">

<label for="titulo">TÍTULO</label>

<input :value="foto.titulo" @input="foto.titulo = $event.target.value" id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input :value="foto.url" @input="foto.url = $event.target.value" id="url" autocomplete="off">

<imagem-responsiva/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea :value="foto.descricao" id="descricao" autocomplete="off" @input="foto.descricao = $event.target.value"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<!-- código posterior omitido -->

Agora sim! Nosso formulário é limpo! Temos aqui dois tipos de data binding ativos ao mesmo tempo, cada um com uma direção diferente o que caracteriza um **two way-data binding**. Podemos até usar como subtítulo do nosso componente o título da foto que estamos digitando. A media que formos digitando no campo, a interpolação exibirá o valor mais atual:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado">{{ foto.titulo }}</h2>

<!-- código posterior omitido -->

Tudo funciona, mas o Vue possui um atalho que pode nos poupar muito código na hora de usarmos esses dois tipos de associação de dados ao mesmo tempo. É o que veremos no próximo vídeo.

O Vue disponibiliza a diretiva v-model que faz exatamente o que acabamos de fazer sem termos que explicitar os dois tipos de associações de dados que fizemos. Alterando nosso componente:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado">{{ foto.titulo }}</h2>

<form @submit.prevent="grava()">

<div class="controle">

<label for="titulo">TÍTULO</label>

<input v-model="foto.titulo" id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input v-model="foto.url" id="url" autocomplete="off">

<imagem-responsiva/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea v-model="foto.descricao" id="descricao" autocomplete="off"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<!-- código posterior omitido -->

Veja que escrevemos muito menos com a diretiva v-model e temos o mesmo resultado que antes. Tanto no código anterior quanto na diretiva v-model os dados são atualizados por padrão a cada entrada de dados que realizarmos nos elementos de entrada do formulário. Tanto isso é verdade que o subtítulo da página é atualizado a cada dígito no campo do título do formulário.

Agora que já temos uma maneira elegante de capturar a entrada do usuário, que tal passarmos as informações de que nosso componente ImagemResponsiva necessita para ser renderizado?

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código anterior omitido -->

<div class="controle">

<label for="url">URL</label>

<input v-model="foto.url" id="url" autocomplete="off">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

</div>

<!-- código posterior omitido -->

</template>

<!-- código posterior omitido -->

Excelente, mas essa solução deixa a desejar. A cada dígito no campo da URL atualizaremos foto.url com o valor digitado e, como realizamos um bind da propriedade de foto.url com a propriedade url de ImagemResponsiva, toda entrada fará com que o componente seja atualizado, ou seja, por debaixo dos panos o src de imgserá alterado, fazendo com que uma requisição seja disparada.

Em um mundo ideal, a atualização da URL entrada pelo usuário só deveria ser associada à foto.url quando ele deixásse o campo, ou seja, queremos postergar a aplicação da diretiva v-model apenas para o campo com o endereço da foto.

Isso é possível através do modificador .lazy suportado pela diretiva v-model. Alterando nosso template:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código anterior omitido -->

<div class="controle">

<label for="url">URL</label>

<input v-model.lazy="foto.url" id="url" autocomplete="off">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo"/>

</div>

<!-- código posterior omitido -->

</template>

<!-- código posterior omitido -->

Podemos adicionar o modificador também no campo do título, fazendo com que o valor do título digitado só seja exibido na como subtítulo da página assim que o usuário sair do campo:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código anterior omitido -->

<div class="controle">

<label for="titulo">TÍTULO</label>

<input v-model.lazy="foto.titulo" id="titulo" autocomplete="off">

</div>

<!-- código posterior omitido -->

Por fim, vamos exibir ImagemResponsiva apenas quando o valor de foto.url for diferente de branco:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código anterior omitido -->

<div class="controle">

<label for="url">URL</label>

<input v-model.lazy="foto.url" id="url" autocomplete="off">

<imagem-responsiva v-show="foto.url" :url="foto.url" :titulo="foto.titulo"/>

</div>

<!-- código posterior omitido -->

</template>

<!-- código posterior omitido -->

Nosso formulário é funcional, no entanto você deve ter reparado que tanto na inicialização da propriedade foto e na limpeza do formulário temos um código repetido que cria um objeto com as propriedades titulo, url e descricao. E se nosso formulário tiver mais um campo? Precisaremos de adicionar mais uma propriedade nesses dois lugares.

Nesse contexto, faria mais sentido se tivéssemos uma forma, um model para criar instâncias de Foto. No mundo da orientação a objetos usamos classes para isso. Como o Vue utiliza o ES2105 por padrão, podemos criar uma classe que representa uma foto em nosso sistema.

Vamos criar o arquivo alurapic/src/domain/foto/Foto.js:

// alurapic/src/domain/foto/Foto.js

export default class Foto {

constructor(titulo='', url='', descricao='') {

this.titulo = titulo;

this.url = url;

this.descricao = descricao;

}

}

Se você já domina ES2015 não há nenhuma novidade. Temos um construtor que recebe três parâmetros e caso eles não sejam passados será utilizado uma string vazia como valor padrão.

Vale destacar a sintaxe export default Foto indicando que o módulo Foto.js exporta por padrão a classe Foto. Isso evita que tenhamos que indicar na instrução import o que importar do módulo.

Agora que já temos nossa classe criada, vamos importá-la em alurapic/src/components/cadastro/Cadastro.vue utilizar instâncias dessa classe evitando que tenhamos que definir em Cadastro a estrutura de um objeto foto:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Cadastro</h1>

<h2 class="centralizado">{{ foto.titulo }}</h2>

<form @submit.prevent="grava()">

<div class="controle">

<label for="titulo">TÍTULO</label>

<input v-model.lazy="foto.titulo" id="titulo" autocomplete="off">

</div>

<div class="controle">

<label for="url">URL</label>

<input v-model.lazy="foto.url" id="url" autocomplete="off">

<imagem-responsiva v-show="foto.url" :url="foto.url" :titulo="foto.titulo"/>

</div>

<div class="controle">

<label for="descricao">DESCRIÇÃO</label>

<textarea v-model="foto.descricao" id="descricao" autocomplete="off"></textarea>

</div>

<div class="centralizado">

<meu-botao rotulo="GRAVAR" tipo="submit"/>

<router-link to="/"><meu-botao rotulo="VOLTAR" tipo="button"/></router-link>

</div>

</form>

</div>

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

// importando a classe Foto

import Foto from '../../domain/foto/Foto.js';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data() {

return {

foto: new Foto()

}

},

methods: {

grava() {

console.log(this.foto);

this.foto = new Foto();

}

}

}

</script>

<style scoped>

.centralizado {

text-align: center;

}

.controle {

font-size: 1.2em;

margin-bottom: 20px;

}

.controle label {

display: block;

font-weight: bold;

}

.controle label + input, .controle textarea {

width: 100%;

font-size: inherit;

border-radius: 5px

}

.centralizado {

text-align: center;

}

</style>

Agora que já temos tudo no lugar já podemos enviar os dados do formulário para nossa API. Sabemos que o módulo VueResource adiciona no global view object o objeto $httpque possui um método para cada método suportado pelo HTTP. No caso, nossa API espera receber uma requisição com o método POST para http://localhost:3000/v1/fotos.

Alterando nosso código:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

/\* código omitido \*/

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

import Foto from '../../domain/foto/Foto.js';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data() {

return {

foto: new Foto()

}

},

methods: {

grava() {

this.$http

.post('http://localhost:3000/v1/fotos', this.foto)

.then(() => this.foto = new Foto(), err => console.log(err));

}

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Assim que cadastramos uma foto, podemos voltar para Home e ver que a nova foto aparece na lista de fotos trazidas do servidor. Mas se nos arrependermos da foto que cadastramos e quisermos apagá-la? É isso que veremos no próximo capítulo.

Neste capítulo aprendemos:

* a capturar dados inseridos em um formulário
* o papel do modificador prevent na submissão de um formulário
* a utilizar a diretiva v-model para two-way data binding
* a criar e a utilizar modelos
* enviar dados para uma API REST através do método post de $http

No capítulo anterior aprendemos a enviar dados para nossa API, mas tão importante quando o cadastro de novas fotos é sua exclusão. Já temos o método remove que já recebe a foto que desejamos excluir. Daí, basta usarmos this.$http para realizar uma requisição com o método delete que receberá uma URL que aponta para o endereço do recurso que desejamos apagar. Como as fotos vieram do servidor, todas elas possuem a informação \_id que foi gerado automaticamente pela API.

Além dessa alteração, vamos também colocar um parágrafo centralizado que exibirá a informação de mensagem, propriedade que criaremos em nossa função data. No momento da remoção, vamos alterar a propriedade e através do mecanismo de data binding a mensagem será exibida automaticamente para o usuário.

Resumindo:

* criar a propriedade mensagem na função data iniciada com uma string vazia
* adicionar um parágrafo que exibe a mensagem usando a diretiva v-show para oculta-lo quando a mensagem estiver vazia
  + usar $http no método remove para solicitar a exclusão da foto pela nossa API

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<!-- novo elemento para exibir mensagens para o usuário -->

<p v-show="mensagem" class="centralizado">{{ mensagem }}</p>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:scale.animate="1.2"/>

<meu-botao

rotulo="remover"

tipo="button"

estilo="perigo"

:confirmacao="true"

@botaoAtivado="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

export default {

components: {

'meu-painel': Painel,

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data () {

return {

fotos: [],

filtro: '',

mensagem: ''

}

},

computed: {

fotosComFiltro() {

if (this.filtro) {

let exp = new RegExp(this.filtro.trim(), 'i');

return this.fotos.filter(foto => exp.test(foto.titulo));

} else {

return this.fotos;

}

}

},

methods: {

remove(foto) {

this.$http

.delete(`http://localhost:3000/v1/fotos/${foto.\_id}`)

.then(() => {

// assim que apagar, exibe a mensagem para o usuário

this.mensagem = 'Foto removida com sucesso'

},

err => {

this.mensagem = 'Não foi possível remover a foto';

console.log(err);

}

)

}

},

created() {

this.$http

.get('http://localhost:3000/v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

.centralizado {

text-align: center;

}

.lista-fotos {

list-style: none;

}

.lista-fotos .lista-fotos-item {

display: inline-block;

}

.filtro {

display: block;

width: 100%;

}

</style>

Excelente. Quando clicamos em remover, vemos a mensagem 'Foto removida com sucesso', mas a foto ainda continua! Mais estranho ainda é que se recarregarmos nossa aplicação a foto que apagarmos não é exibida. O que esta acontecendo? É o que veremos no próximo vídeo.

O problema é que apagamos a informação lá no banco de dados acessado pela nossa API, mas no lado do cliente, isto é, no lado do navegador a foto que apagamos ainda faz parte da lista definida na propriedade fotos da nossa função data. É por isso que ao recarregarmos a página a foto desaparece, porque uma nova requisição é feito para a API e uma nova lista, sem a foto que acabamos de excluir é recebida. Através do mecanismo de data binding a atribuição da nova lista na propriedade fotos faz com que nossa view seja atualizada refletindo o estado atual da lista.

Uma solução é realizar uma nova requisição para o servidor para buscarmos a lista atualizada, no entanto, é sempre bom evitar requisições desnecessárias, e essa é uma delas. Podemos simplesmente usar a velha conhecida função splice do JavaScript para remover um item do array:

// alurapic/src/components/home/Home.vue

// código anterior omitido

methods: {

remove(foto) {

this.$http

.delete(`http://localhost:3000/v1/fotos/${foto.\_id}`)

.then(

() => {

let indice = this.fotos.indexOf(foto); // acha a posição da foto na lista

this.fotos.splice(indice, 1); // a instrução altera o array

this.mensagem = 'Foto removida com sucesso'

},

err => {

this.mensagem = 'Não foi possível remover a foto';

console.log(err);

}

)

}

},

// código posterior omitido

Excelente, agora temos uma sincronia entre o estado das fotos no servidor com o estado das fotos no cliente.

Hoje, usamos o endereço localhost:3000 para consumir nossa API localmente. Sabemos que esse é um endereço de desenvolvimento e que futuramente precisamos alterá-lo para apontar para o endereço da API hospedada em algum servidor da web.

Talvez, você tenha que trocar ainda mesmo em ambiente de desenvolvimento, por exemplo, caso esteja acessando uma API na porta 8080. São todos motivos genuínos de mudança, mas do jeito que nossa aplicação esta teremos problemas de manutenção.

Acessamos a API nos componentes Home Cadastro. Se o endereço mudar, teremos que lembrar de alterar nos dois componentes. Em uma aplicação com escopo reduzido como a nossa não é tão difícil procurar por todos os locais que acessam a API, mas em uma aplicação maior isso pode ser tornar um pesadelo. Basta esquecermos de alterar em um local para que a aplicação deixe de funcionar corretamente.

A boa notícia é que o VueRouter nos permite centralizar o domínio da nossa API em um único local evitando assim que tenhamos de informá-lo toda vez que formos acessar nossa API.

Vamos abrir o arquivo alurapic/src/main.js, local onde registramos o VueRouter.

Com o arquivo aberto, vamos atribuir o endereço http://localhost:3000 para a propriedade Vue.http.options.root:

// alurapic/src/main.js

import Vue from 'vue'

import App from './App.vue'

import VueResource from 'vue-resource';

import VueRouter from 'vue-router';

import { routes } from './routes';

import './directives/Transform';

Vue.use(VueRouter);

const router = new VueRouter({

routes,

mode: 'history'

});

Vue.use(VueResource);

// http usará sempre o endereço abaixo

Vue.http.options.root = 'http://localhost:3000';

new Vue({

el: '#app',

router,

render: h => h(App)

})

O valor passado para a propriedade root será a raiz de toda requisição que fizermos através de VueRouter e seus artefatos como $http.

Agora, em todo local que usamos $http podemos remover a parte do endereço http://localhost:3000.

Alterando nosso componente Cadastro.

// alurapic/src/components/cadastro/Cadastro.vue

// código anterior omitido

methods: {

grava() {

console.log(this.foto);

// removido parte do endereço!

this.$http

.post('v1/fotos', this.foto)

.then(() => this.foto = new Foto(), err => console.log(err));

}

}

}

// código posterior omitido

Agora em nosso componente Home:

// alurapic/src/components/home/Home.vue

// código anterior omitido

methods: {

remove(foto) {

// removido parte do endereço aqui

this.$http

.delete(`v1/fotos/${foto.\_id}`)

.then(

() => {

let indice = this.fotos.indexOf(foto);

this.fotos.splice(indice, 1);

this.mensagem = 'Foto removida com sucesso'

},

err => {

this.mensagem = 'Não foi possível remover a foto';

console.log(err);

}

)

}

},

created() {

// removido parte do endereço aqui

this.$http

.get('v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

// código posterior omitido

Nossa aplicação deve continuar funcionando da mesma forma como antes, apenas centralizamos o domínio da nossa API em um único lugar.

Que tal se pudermos melhorar ainda mais nosso código? É isso que veremos no próximo vídeo!

Aprendemos a utilizar $http para consumir API's. No entanto, $http é um objeto especializado apenas na realização de requisições assíncronas ou se você preferir, requisições Ajax. Contudo, há um outro objeto mais especializado focado no consumo de API's que seguem o Padrão REST. É o **$resource**.

Quando trabalhamos com Single Page Applications, é extremanente comum acessarmos API's criadas com o padrão REST. REST é um estilo arquitetural para construir aplicações em rede que utiliza nada mais nada menos que o próprio protocolo http para sua comunicação. Resumidamente, cada recurso que desejamos acessar possui um identificador único e usamos os métodos (GET, PUT, POST, entre outros) do http para executarmos operações.

Geralmente é a equipe do backend que constroi as API's disponibilizando os endereços para a equipe front-end poder consumir. Como este é um curso focado em front, a construção de uma API REST esta fora do escopo. Na Alura você encontra cursos de REST, inclusive com linguagens diferentes, caso queria se aprofundar no assunto.

Então, como a API disponibilizada pelo curso segue o padrão REST, podemos usar $resouce ao invés de $http.

Primeiro, precisamos ter um recurso configurado. Vamos alterar o componente Home e no método created vamos adicionar dinamicamente a propriedade resource em nosso componente. Ela armazenará o retorno de this.$resource que nada mais é do que um recurso configurado para um endereço:

// alurapic/src/components/home/Home.vue

// código anterior omitido

created() {

// agora conseguimos acessar o recurso configurado em outros métodos do nosso componente

this.resource = this.$resource('v1/fotos');

this.$http

.get('v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

// código posterior omitido

Veja que imediatamente abaixo da nova propriedade, há o código que usa $http para buscar as fotos da nossa API. Vamos usar this.resource em seu lugar. Por convenção, se chamarmos o método query do nosso recurso, estaremos fazendo um requisição do tipo GET para v1/fotos:

// alurapic/src/components/home/Home.vue

// código anterior omitido

created() {

this.resource = this.$resource('v1/fotos');

this.resource

.query()

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

/\*

this.$http

.get('v1/fotos')

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

\*/

}

}

// código posterior omitido

Não parece ter mudado muita coisa, mas para quem chama o método query não precisa saber o detalhe do endereço, apenas no momento da criação do recurso. Isso se tornará mais evidente agora, quando formos alterar o código que apaga um foto da nossa API.

Sabemos que endereços como v1/fotos/3 quando usado o método DELETE apaga um recurso da noss API. Claro, ela esta preparada para entender endereços como esse inclusive o método empregado.

Eu acabei de dizer para vocês que o endereço da nosso recurso será definido em um lugar apenas, em this.resource. Mas e agora? Precisamos de um endereço que aceite receber o código da foto para que seja apagada.

Para isso, precisamos alterar o endereço passado para $resource adicionando, um curinga, ou melhor, um parâmetro:

fotos`:

// alurapic/src/components/home/Home.vue

// código anterior omitido

created() {

// veja que foi adicionado {/id}

this.resource = this.$resource('v1/fotos{/id}');

this.resource

.query()

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

// código posterior omitido

Esse parâmetro do endereço da nossa API, quando não for passado, é ignorado. Perfeito para o método query que só leva em consideração v1/fotos, excluindo o parâmetro.

Agora, para apagarmos um recurso com o this.resource que criamos, fazemos da seguinte forma:

// alurapic/src/components/home/Home.vue

// código anterior omitido

methods: {

remove(foto) {

// a chave do objeto é o parâmetro usando no endereço do recurso

this.resource

.delete({id: foto.\_id})

.then(

() => {

let indice = this.fotos.indexOf(foto);

this.fotos.splice(indice, 1);

this.mensagem = 'Foto removida com sucesso'

},

err => {

this.mensagem = 'Não foi possível remover a foto';

console.log(err);

}

)

}

},

created() {

// parametrizando o endereço

this.resource = this.$resource('v1/fotos{/id}');

this.resource

.query()

.then(res => res.json())

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

// código posterior omitido

Veja que a função this.resource.delete recebe como primeiro parâmetro um objeto JavaScript. A chave do objeto tem que ser exatamente igual ao parâmetro que adicionamos em nossa rota, no caso id. Para esta chave, passamos o valor. O resource sabe adicionar o parâmetro passado para completar o endereço.

Por fim, vamos alterar o componente Cadastro e substiuir $http pelo $resource. Precisamos adicionar o método created para configurarmos nosso resource.

Por fim, usamos o método save para realizarmos um post para nossa API:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código omitido -->

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

import Foto from '../../domain/foto/Foto.js';

export default {

components: {

'imagem-responsiva': ImagemResponsiva,

'meu-botao': Botao

},

data() {

return {

foto: new Foto(),

resource: {}

}

},

methods: {

grava() {

console.log(this.foto);

// o método save realiza um POST por debaixo dos panos enviado os dados passado como parâmetro

this.resource

.save(this.foto)

.then(() => this.foto = new Foto(), err => console.log(err));

}

},

created() {

this.resource = this.$resource('v1/fotos{/id}');

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Veja que $resource foi feito para consumir API's no padrão REST. Se a API das suas aplicações não usa esse padrão, você pode usar o $http sem problema algum.

Apesar do nosso código estar um pouquinho mais sofisticado ele ainda deixa a desejar. Veja que tanto em Home quanto em Cadastro definimos o recurso no método created. E se o endereço do recurso mudar? Precisaremos mudar em vários lugares.

O ideal é centralizar todo acesso para API's em uma classe com essa responsabilidade que possui diversas vantagens. É isso que veremos no próximo vídeo.

Vamos criar uma classe especializada em consumir nossa API. Como o Vue usa ES2015, não temos restrições no que podemos usar essa fantástica linguagem. Até porque, o CLI converte nosso código para ES5 garantindo assim maior compatibilidade entre os navegadores.

Nossa classe se chamará FotoService e ficará no mesmo diretório do modelo Foto:

// alurapic/src/domain/foto/FotoService.js

export default class FotoService {

constructor(resource) {

this.\_resource = resource('v1/fotos{/id}');

}

cadastra(foto) {

return this.\_resource.save(foto);

}

lista() {

return this.\_resource

.query()

.then(res => res.json());

}

apaga(id) {

return this.\_resource.delete({ id });

}

}

Veja que ela recebe em seu construtor um resource, no caso, será o this.$resource do componente que utilizará o serviço. Veja que internamente na propriedade this.\_resource guardamos um instância configurada.

Por fim, temos métodos de alto nível que encapsulam os métodos do resource. Como os métodos de um recurso do Vue Resource devolvem uma promise, podemos retornar a promise resultante nesses métodos que criamos. Quem chamar os métodos terá acesso ao resultado encadeando uma chamada a função then.

Vamos alterar agora nosso componente Home que precisa importar a classe para em seguida, no método created, criarmos uma instância dessa classe passando $resourcecomo parâmetro. Por fim, não teremos mais a propriedade resource adicionada dinamicamente no componente, mas service:

<!-- alurapic/src/componens/home/Home.js -->

<template>

<!-- código omitido -->

</template>

<script>

import Painel from '../shared/painel/Painel.vue';

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

// importando FotoService

import FotoService from '../../domain/foto/FotoService';

export default {

/\* código anterior omitido \*/

methods: {

remove(foto) {

this.service

.apaga(foto.\_id)

.then(

() => {

let indice = this.fotos.indexOf(foto);

this.fotos.splice(indice, 1);

this.mensagem = 'Foto removida com sucesso'

},

err => {

this.mensagem = 'Não foi possível remover a foto';

console.log(err);

}

)

}

},

created() {

// criando uma instância do nosso serviço que depende de $resource

this.service = new FotoService(this.$resource);

this.service

.lista()

.then(fotos => this.fotos = fotos, err => console.log(err));

}

}

</script>

<style>

/\* código omitido \*/

</style>

Agora, em Cadastro:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<!-- código omitido -->

</template>

<script>

import ImagemResponsiva from '../shared/imagem-responsiva/ImagemResponsiva.vue'

import Botao from '../shared/botao/Botao.vue';

import Foto from '../../domain/foto/Foto';

import FotoService from '../../domain/foto/FotoService';

export default {

// código anterior omitido

methods: {

grava() {

this.service

.cadastra(this.foto)

.then(() => this.foto = new Foto(), err => console.log(err));

}

},

created() {

this.service = new FotoService(this.$resource);

}

}

</script>

<style scoped>

/\* código omitido \*/

</style>

Agora nossos componentes ficam desacoplados do endereço da API, inclusive dos detalhes de conversão de dados.

Que tal refletirmos um pouco sobre as rotas da nossa Single Page Application?

Em nosso componente Cadastro, temos o botão VOLTAR que ao ser clicado dispara uma rota para o endereço /. Aprendemos que ele equivale ao path: '' de uma de nossas rotas registradas em alurapic/src/routes.js.

Mas se a rota para o componente Home mudar? Teremos que lembrar de alterá-la o botão VOLTAR e em todos os outros lugares da aplicação que estão amarradas ao endereço.

A boa notícia é que podemos dar um nome para as rotas e referenciarmos este nome em todos os lugares que precisamos gerar uma navegação. A vantagem disso é que se o path da rota mudar, o restante da nossa aplicação não será afetado.

Vamos alterar o arquivo routes.js e adicionar a propriedade name para das duas rotas que temos:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

{ path: '', name: 'home', component: Home, titulo: 'Home' },

{ path: '/cadastro', name:'cadastro', component: Cadastro, titulo: 'Cadastro' }

];

Agora, no componente Home, vamos alterar nosso botão VOLTAR. O primeiro passo é que a propriedade to da diretiva router-link precisa fazer uma data binding com o router global, local de onde ele procurará as rotas. Sendo assim, precisamos adicionar os dois pontos antes da propriedade. Por fim, seu valor é um objeto JavaScript nome temos a propriedade name e como valor o nome de uma rota válida:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<!-- código anterior omitido -->

<router-link :to="{name: 'home'}"><meu-botao rotulo="VOLTAR" tipo="button"/></a>

<!-- código posterior omitido -->

Excelente, tudo continua funcionando. Ah, não precisamos alterar o código do nosso menu que é gerado automaticamente, porque se alterarmos algo em nossas rotas, ele refletirá sempre a última versão.

Por fim, o que acontece se acessarmos um endereço que não existe? Apensar o menu da aplicação é exibida. Que tal gerarmos uma navegação para Home? Para isso, vamos colocar uma nova rota em routes.js:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

{ path: '', name: 'home', component: Home, titulo: 'Home' },

{ path: '/cadastro', name:'cadastro', component: Cadastro, titulo: 'Cadastro' },

{ path: '\*', component: Home }

];

O curinga \* atribuído à propriedade path da nova rota a ativará caso a o endereço acessado não veja o valor de algumas das nossas rotas.

Façamos um teste acessando um endereço que não existe. Somos direcionados para Home. No entanto, se você esta tão atento quanto eu, verá que apareceu mais uma opção em nosso menu, mas sem nome. O problema é que a nova rota esta sendo considerada item de menu, o que não deveria acontecer.

Podemos resolver isso adicionando mais uma propriedade na configuração de nossas rotas para indicar se ela deve entrar no menu ou não:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

{ path: '', name: 'home', component: Home, titulo: 'Home', menu: true },

{ path: '/cadastro', name:'cadastro', component: Cadastro, titulo: 'Cadastro', menu: true },

{ path: '\*', component: Home, menu: false }

];

Apenas as rotas que tivem menu: true serão exibidas em nosso menu. Por fim, precisamos filtrar a lista de rotas para que apenas as que representem menu sejam consideradas:

<!-- alurapic/src/App.vue -->

<template>

<!-- código omitido -->

</template>

<script>

import { routes } from './routes';

import Menu from './components/shared/menu/Menu.vue';

export default {

components: {

'meu-menu' : Menu

},

data() {

return {

routes: routes.filter(route => route.menu)

}

}

}

</script>

<style>

/\* código omitido \*/

</style>

Perfeito!

Neste capítulo aprendemos:

* a usar o método delete para excluir recursos através da nossa API
* a sincronizar o estado da exclusão da API no cliente
* a aproveitar Vue.http.options.root e facilitar a manutenção
* a utilizar $resource
* a isolar $resource em serviços
* rotas nomeadas e suas vantagens

Precisamos implementar a funcionalidade de alteração de fotos. Para isso, em Home. vamos adicionar um novo botão ao lado do botão Remover:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<p v-show="mensagem" class="centralizado">{{ mensagem }}</p>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:scale.animate="1.2"/>

<!-- novo botão! -->

<router-link :to="{ name: 'cadastro'}">

<meu-botao

rotulo="Alterar"

tipo="button"/>

</router-link>

<meu-botao

rotulo="remover"

tipo="button"

estilo="perigo"

:confirmacao="true"

@botaoAtivado="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<!-- código posterior omitido -->

Veja que ele usa a rota nomeada cadastro. Quando clicamos no botão, somos direcionados para Cadastro. No entanto, precisamos enviar o ID da foto que estamos alterado para Cadastro para que seja possível buscamos a foto da nossa API exibindo-a no formulário.

Para que isso seja possível, vamos alterar nosso arquivo routes.js e adicionar um curinga na rota:

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

{ path: '', name: 'home', component: Home, titulo: 'Home', menu: true },

{ path: '/cadastro/:id', name:'cadastro', component: Cadastro, titulo: 'Cadastro', menu: true },

{ path: '\*', component: Home, menu: false }

];

Agora, toda vez que endereços como cadastro/1 ou cadastro/22 forem acessados, o Vue Router saberá que é essa a rota que deve ser seguida.

Agora, no template de Home, podemos realizar uma navegação para cadastro passando como parâmetro o ID da foto. Isso acarreta uma mudança em nosso router-link:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<p v-show="mensagem" class="centralizado">{{ mensagem }}</p>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:scale.animate="1.2"/>

<router-link :to="{ name: 'cadastro', params: { id : foto.\_id }}">

<meu-botao

rotulo="Alterar"

tipo="button"/>

</router-link>

<meu-botao

rotulo="remover"

tipo="button"

estilo="perigo"

:confirmacao="true"

@botaoAtivado="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<!-- código posterior omitido -->

Reparem que além do name, o objeto passado para a propriedade to agora possui a propriedade params. Essa propriedade é outro objeto JavaScript cuja a propriedade tem que ser exatamente o nome do curinga que usamos na rota. O valor da propriedade é o ID da foto.

Quando clicamos no botão, somos direcionados para /cadastro/IdDaFotoAqui. Excelente, mas temos um problema. Se clicamos no menu Home e depois no menu Cadastro continuamos em Home. Isso acontece porque /cadastro deixou de ser uma rota válida, rotas válidas agora são aquelas que possuem /cadastro/ seguido de qualquer valor. E agora?

Para resolvermos, vamos voltar a rota para o jeito anterior e adicionar mais uma rota, essa sim parametrizada e que não faz parte do menu:

// alurapic/src/routes.js

import Home from './components/home/Home.vue';

import Cadastro from './components/cadastro/Cadastro.vue';

export const routes = [

{ path: '', name: 'home', component: Home, titulo: 'Home', menu: true },

{ path: '/cadastro', name:'cadastro', component: Cadastro, titulo: 'Cadastro', menu: true },

{ path: '/cadastro/:id', name:'altera', component: Cadastro, titulo: 'Cadastro', menu: false },

{ path: '\*', component: Home, menu: false }

];

A nova rota tem o nome altera, não podemos nos esquecer de alterar esse nome no router-link de Home:

<!-- alurapic/src/components/home/Home.vue -->

<template>

<div>

<h1 class="centralizado">Alurapic</h1>

<p v-show="mensagem" class="centralizado">{{ mensagem }}</p>

<input type="search" class="filtro" @input="filtro = $event.target.value" placeholder="filtre pelo título da foto">

<ul class="lista-fotos">

<li class="lista-fotos-item" v-for="foto of fotosComFiltro">

<meu-painel :titulo="foto.titulo">

<imagem-responsiva :url="foto.url" :titulo="foto.titulo" v-meu-transform:scale.animate="1.2"/>

<router-link :to="{ name: 'altera', params: { id : foto.\_id }}">

<meu-botao

rotulo="Alterar"

tipo="button"/>

</router-link>

<meu-botao

rotulo="remover"

tipo="button"

estilo="perigo"

:confirmacao="true"

@botaoAtivado="remove(foto)"/>

</meu-painel>

</li>

</ul>

</div>

</template>

<!-- código posterior omitido -->

Agora, conseguimos acessar Cadastro diretamente ou passando parâmetro para ela. Precisamos agora é descobrir como conseguimos extrair o ID da foto passado para Cadastro para que possamos então buscar a foto em nossa API.

Como o VueRouter é registrado no global view object temos acesso ao $route em nosso componentes. É através dele que conseguimos extrair o parâmetro passado na rota para que possamos utilizá-lo em nosso componente.

O primeiro passo que faremos em Cadastro é adicionar a propriedade id como propriedade do objeto retornado pela função data. Seu valor será this.$route.params.id:

// alurapic/src/components/cadastro/Cadastro.vue

// código anterior omitido

data() {

return {

foto: new Foto(),

id: this.$route.params.id

}

},

// código posterior omitido

É através de $route.params que temos acesso ao parâmetro passado para o componente via rota. No caso, como o nome do nosso parâmetro é id, usamos $route.params.id.

Agora que já sabemos o ID da foto, podemos buscá-la em nossa API. Mas primeiro, precisamos criar o método busca em FotoService:

export default class FotoService {

/\* código anterior omitido \*/

busca(id) {

return this.\_resource

.get({ id })

.then(res => res.json());

}

}

O método busca recebe o ID da foto e nos devolve uma promise que os fornecerá o objeto foto trazido de nossa API.

Agora, podemos usar este novo método em Cadastro, mas dentro do método created. Inclusive, precisamos testar se this.id possui valor, porque quando acessamos Cadastro diretamente sem passar parâmetro, o id será undefined.

// alurapic/src/components/cadastro/Cadastro.vue

// código anterior omitido

created() {

this.service = new FotoService(this.$resource);

if(this.id) {

this.service

.busca(this.id)

.then(foto => this.foto = foto);

}

}

// código posterior omitido

Perfeito! Um teste demonstra que ao clicarmos em qualquer foto em Home, somos direcionados para Cadastro com o formulário exibido com os dados da foto que desejamos alterar.

Agora que em Cadastro já sabemos como obter o id da foto passado na URL já podemos implementar a lógica de alteração. No caso, vamos alterar o método cadastrade FotoService. A solução para saber se estamos incluindo ou alterado é simples. Se o objeto foto passado como parâmetro tiver a propriedade \_id é porque estamos alterando, se não tiver, é porque ele nunca foi gravado no banco e precisamos incluí-lo através da nossa API.

Aprendemos que o método save do nosso resource é aquele que realiza um POST por debaixo dos panos, para alteração, usaremos o método update que recebe dois parâmetros. O primeiro é o id da foto que desejamos alterar em nossa API, porque noss API depende dessa informação para encontrar o recurso. O segundo, é o objeto foto em si.

// alurapic/src/service/FotoService.js

export default class FotoService {

constructor(resource) {

this.\_resource = resource('v1/fotos{/id}');

}

cadastra(foto) {

if(foto.\_id) {

return this.\_resource.update({ id: foto.\_id }, foto);

} else {

return this.\_resource.save(foto);

}

}

// código posterior omitido

}

Será que funciona? Vamos primeiro cadastrar uma novo foto para ver se nada quebrou. Em seguida, vamos escolher uma foto da lista e alterar seu nome. Quando clicamos em GRAVAR o formulário é limpo. Mas será que realmente gravou? Para isso, vamos voltar para Home e constatar a mudança, excelente! Mas pode ficar ainda melhor.

Quando cadastramos uma novo foto, faz sentido limparmos o formulário e continuarmos em Cadastro para que o usuário possa cadastrar mais fotos. Já a alteração, que tal assim que alterarmos com sucesso a foto voltarmos para Home? Precisamos aprender a gerar uma navegação através do nosso componente.

O VueRouter disponibiliza no global view instance o objeto $router que possui o método push. Podemos passar para esse método um objeto JavasScript que representa nossa rota nomeada, assim como fizemos em router-link:

// alurapic/src/components/cadastro/Cadastro.vue

methods: {

grava() {

this.service

.cadastra(this.foto)

.then(() => {

if(this.id) this.$router.push({ name: 'home'});

this.foto = new Foto()

},

err => console.log(err));

}

},

Reparem que a navegação só será realizada caso o ID da foto esteja preenchido, ou seja, em uma alteração.

## A diretiva v-if e v-else

Para ficar ainda melhor, podemos exibir o título "Inclusão" ou o título "Alteração" no formulário de acordo com seu estado. Para isso, vamos alterar o título h1 já existente para Alteração e adicionar outro título h1 com o texto Inclusão em nosso formulário:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 class="centralizado">Alteração</h1>

<h1 class="centralizado">Inclusão</h1>

<!-- código posterior omitido -->

Não precisamos refletir muito sobre essa alteração para saber que ela exibirá os dois títulos, independente se estamos alterando ou incluindo. É ai que entra a diretiva v-ife v-else. Alterando:

<!-- alurapic/src/components/cadastro/Cadastro.vue -->

<template>

<div>

<h1 v-if="foto.\_id" class="centralizado">Alteração</h1>

<h1 v-else class="centralizado">Inclusão</h1>

<!-- código posterior omitido -->

Se o ID da foto for definido, apenas o h1 com o texto "Alteração" será exibido, caso contrário será o com o texto "Inclusão".

A diretiva v-if e v-else são parecidas com a diretiva v-show, mas enquanto o v-show apenas esconde o elemento do DOM através da propriedade "display: none" por debaixo dos panos, o uso do v-if e v-else literalmente removem e adicionam o elemento no DOM.

Hoje, quando acessamos nossa aplicação sem que nossa API esteja rodando o que acontece? Vemos uma página com apenas o menu com conteúdo em branco. Se abrirmos o console do Chrome vemos informações mais detalhadas.

Deixar a página em branco pode suscitar muitas dúvidas no usuário a respeito do que esta acontecendo. É por isso que precisamos exibir uma mensagem para que ele saiba que não foi possível buscar as imagens e que ele pode tentar mais tarde.

Aprendemos que toda promise possui no método then dois callbacks, funções que são chamadas em um tempo futuro. O primeiro é o callback de sucesso e o segundo de fracasso. Sendo assim, podemos alterar o componente Home desta forma:

// alurapic/src/components/home/Home.vue

// código anterior omitido

created() {

this.service = new FotoService(this.$resource);

this.service.lista()

.then(

fotos => this.fotos = fotos,

err => {

console.log(err); // logando o erro que veio do server para o desenvolvedor

this.mensagem = 'Não foi possível obter as fotos. Tenta mais tarde.';

});

}

// código posterior omitido

Se pararmos nosso servidor com a nossa API e recarregarmos nossa aplicação, vemos a mensagem de erro sendo exibida. Funciona, mas a podemos aplicar uma solução ainda melhor.

Veja que em todos os lugares que o método lista de FotoService for chamado, teremos que logar a mensagem de erro que veio do servidor, o que é uma boa prática para em seguida definir a mensagem que desejamos exibir para o usuário. Essa mensagem é importante, porque não faz sentido mostrar o erro que veio do servidor para a tela dele, é uma informação muita técnica que só geraria mais confusão.

Então, podemos isolar o processo de logar a mensagem de erro do servidor e a mensagem de erro de alto nível, isto é, aquela feita para o usuário no próprio serviço. A vantagem dessa abordagem é que todos os lugares que chamarem o método lista não precisaram logar o erro e nem definir sua própria mensagem de erro.

Alterando FotoService:

// alurapic/src/domain/foto/FotoService.js

export default class FotoService {

// código anterior omitido

lista() {

return this.\_resource

.query()

.then(

res => res.json(),

err => {

console.log(err);

throw new Error('Não foi possível obter as fotos. Tenta mais tarde');

}

)

}

// código posterior omitido

}

Veja que na função then que realizamos a conversão dos dados para json, no callback de erro, logamos o erro e lançamos um exceção com nossa mensagem de erro. Quem chamar o método lista, além de passar uma função para obter o resultado da operação, passar um callback de erro, terá acesso a mensagem de erro definida pelo serviço. Veja que em nenhum momento quem usa o método lista precisará logar o erro e definir a mensagem.

Alterando o componente Home:

// alurapic/src/components/home/Home.vue

// código anterior omitido

created() {

this.service = new FotoService(this.$resource);

this.service.lista()

.then(fotos => this.fotos = fotos, err => this.mensagem = err.message);

}

// código posterior omitido

Muito mais enxuto e o valor de this.mensagem será err.message, o texto do objeto de Errorlançado pelo serviço.

Podemos usar essa estratégia em vários lugares da nossa aplicação, mas por brevidade vamos aplicar em mais um local apenas, na exclusão. Alterando FotoService:

export default class FotoService {

// código anterior omitido

apaga(id) {

return this.\_resource

.delete({ id })

.then(null, err => {

console.log(err);

throw new Error('Não foi possível remover a foto. Tente mais tarde');

});

}

// código posterior omitido

}

Como a operação de deleção não nos devolve nenhum dado, passamos null para thene em seguida passamos o callback de erro. Nele usamos a mesma estratégia que utilizamos em lista.

Por fim, vamos alterar em Home a parte do código que solicita a nossa API a exclusão de uma foto:

// código anterior omitido

methods: {

remove(foto) {

this.service

.apaga(foto.\_id)

.then(

() => {

let indice = this.fotos.indexOf(foto);

this.fotos.splice(indice, 1);

this.mensagem = 'Foto removida com sucesso'

},

err => this.mensagem = err.message

)

}

},

// código posterior omitido

Por fim, em nosso FotoService é possível escrutinarmos a resposta de erro vinda do servidor, por exemplo, para sabermos o status da requisição entre outras coisas que podem ser levadas em consideração para lançarmos uma mensagem de erro mais apropriada com o erro vindo do servidor varia. Para saber as possível propriedades que o objeto de resposta de erro contém basta verificar sua saída no console.

Neste capítulo aprendemos:

* a criar rotas parametrizadas
* a passar parâmetros para router-link
* a extrair parâmetros de rota em um componente com $route.params
* a realizarmos navegação programaticamente
* a implementar uma lógica de alteração
* boas práticas com mensagens e tratamento de erros
* as diretivas v-if e v-else
* diferença ente v-if/v-else e v-show