

2016

Vue.js

na prática

Browserify
Node/npm
Restfull APIs
Material Design
Json Web Token
Express/MongoDB

Daniel Schmitz

Vue.js na prática (PT-BR)

Daniel Schmitz e Daniel Pedrinha Georgii

Esse livro está à venda em <http://leanpub.com/livro-vue>

Essa versão foi publicada em 2017-05-25



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2016 - 2017 Daniel Schmitz e Daniel Pedrinha Georgii

Gostaria de agradecer as fantásticas comunidades brasileiras:

vuejs-brasil.slack.com

laravel-br.slack.com

telegram.me/vuejsbrasil

Aos autores do blog www.vuejs-brasil.com.br por divulgarem o vue em português. Vocês são incríveis!

Conteúdo

Parte 1 – Conhecendo o Vue	6
1. Introdução	7
1.1 Tecnologias empregadas	7
1.2 Instalação do node	9
1.3 Uso do npm	10
1.4 Conhecendo um pouco o RESTfull	11
2. Conhecendo Vue.js	13
2.1 Uso do jsFiddle	13
2.2 Configurando o jsFiddle para o Vue	14
2.3 Hello World, vue	16
2.4 Two way databind	18
2.5 Criando uma lista	19
2.6 Detectando alterações no Array	21
Utilizando v-bind:key	22
Uso do set	23
Como remover um item	23
Loops em objetos	23
2.7 Eventos e métodos	24
Modificando a propagação do evento	25
Modificadores de teclas	26
2.8 Design reativo	27
2.9 Criando uma lista de tarefas	27
2.10 Eventos do ciclo de vida do Vue	32
2.11 Compreendendo melhor o Data Bind	34
Databind único	34

CONTEÚDO

	Databind com html	35
	Databind em Atributos	35
	Expressões	35
2.12	Filtros	36
	uppercase	36
	lowercase	36
	currency	36
	pluralize	37
	json	37
2.13	Diretivas	37
	Argumentos	37
	Modificadores	38
2.14	Atalhos de diretiva (Shorthands)	38
2.15	Alternando estilos	39
2.16	Uso da condicional v-if	40
2.17	Exibindo ou ocultando um bloco de código	41
2.18	v-if vs v-show	41
2.19	Formulários	42
	Checkbox	42
	Radio	43
	Select	43
	Atributos para input	44
2.20	Conclusão	44
3.	Criando componentes	45
3.1	Vue-cli	45
3.2	Criando o primeiro projeto com vue-cli	46
3.3	Executando o projeto	46
3.4	Conhecendo a estrutura do projeto	47
3.5	Conhecendo o packages.json	48
3.6	Componentes e arquivos .vue	49
3.7	Criando um novo componente	51
3.8	Adicionando propriedades	55
	camelCase vs. kebab-case	57
	Validações e valor padrão	57
3.9	Slots e composição de componentes	59

CONTEÚDO

3.10	Eventos e comunicação entre componentes	60
	Repassando parâmetros	63
3.11	Reorganizando o projeto	64
3.12	Adicionando algum estilo	67
3.13	Alterando o cabeçalho	71
3.14	Alterando o rodapé	72
3.15	Conteúdo da aplicação	74
4.	Vue Router	75
4.1	Instalação	75
4.2	Configuração	75
4.3	Configurando o router	76
4.4	Configurando o router-view	77
4.5	Criando novos componentes	79
4.6	Criando um menu	81
5.	Vue Resource	83
5.1	Testando o acesso Ajax	84
5.2	Métodos e opções de envio	88
5.3	Trabalhando com resources	89

Parte 2 - Criando um blog com Vue 1.0, Express e MongoDB 92

6.	Express e MongoDB	93
6.1	Criando o servidor RESTful	93
6.2	O banco de dados MongoDB	93
6.3	Criando o projeto	98
6.4	Estrutura do projeto	99
6.5	Configurando os modelos do MongoDB	99
6.6	Configurando o servidor Express	101
6.7	Testando o servidor	111
6.8	Testando a api sem o Vue	112

Parte 3 – Conceitos avançados 118

7. Vuex e Flux	119
7.1 O que é Flux?	119
7.2 Conhecendo os problemas	119
7.3 Quando usar?	120
7.4 Conceitos iniciais	120
7.5 Exemplo simples	121
Criando o projeto	122
Criando componentes	122
Incluindo Vuex	124
Criando uma variável no state	126
Criando mutations	127
Criando actions	128
Criando getters	129
Alterando o componente Display para exibir o valor do contador	129
Alterando o component Increment	130
Testando a aplicação	131
7.6 Revendo o fluxo	131
7.7 Chrome vue-devtools	132
7.8 Repassando dados pelo vuex	136
7.9 Tratando erros	139
7.10 Gerenciando métodos assíncronos	140
7.11 Informando ao usuário sobre o estado da aplicação	141
7.12 Usando o vuex para controlar a mensagem de resposta ao usuário	144
7.13 Vuex modular	148
8. Mixins	168
8.1 Criando mixins	168
8.2 Conflito	172
9. Plugins	174
9.1 Criando um plugin	174

Uma nota sobre PIRATARIA

Esta obra não é gratuita e não deve ser publicada em sites de domínio público como o *scrib*. Por favor, contribua para que o autor invista cada vez mais em conteúdo de qualidade **na língua portuguesa**, o que é muito escasso. Publicar um ebook sempre foi um risco quanto a pirataria, pois é muito fácil distribuir o arquivo pdf.

Se você obteve esta obra sem comprá-la no site <https://leanpub.com/livro-vue>, pedimos que leia o ebook e, se acreditar que o livro mereça, compre-o e ajude o autor a publicar cada vez mais.

Obrigado!!

Novamente, por favor, não distribua este arquivo. Obrigado!

Novas Versões

Um livro sobre programação deve estar sempre atualizado para a última versão das tecnologias envolvidas na obra, garantindo pelo menos um suporte de 1 ano em relação a data de lançamento. Você receberá um e-mail sempre que houver uma nova atualização, juntamente com o que foi alterado. Esta atualização não possui custo adicional.

Implementação do Vue 2 no livro:

06/10/2016

- Capítulo 2
 - A versão do Vue no jsFiddle foi alterada de 1.0 para “edge”
 - Removida a informação sobre \$index no laço v-for
 - A propriedade track-by foi alterada para key
 - O método \$set foi alterado para simplesmente set
 - O método \$remove foi alterado para Array.splice
 - A propriedade \$key no loop de objetos foi alterada para key
 - Adicionado os dois fluxos de eventos, tanto o Vue 1 quanto o Vue 2
 - O uso de databind unico com * foi alterado para o uso da diretiva v-once
 - O uso de três chaves {{{ para a formatação do html foi alterado para a diretiva v-html
 - O uso de propriedades computadas foi alterada para v-bind
 - Alterações no filtros, agora só funcionam em {{ ... }}
 - Removido o uso de filtros para laços v-for
 - Adicionado um exemplo com o uso do lazy em campos de formulário
 - Remoção do debounce para campos de formulário

13/10/2016

- Capítulo 3
 - Não é necessário usar browserify-simple#1.0 mais, use apenas browserify-simple
 - A tag <app> do index.html foi alterada para <div id=”app”> no projeto my-vue-app
 - Componentes devem ter pelo menos uma tag pai

- Componente Menu foi alterado para MyMenu
- Removido o uso de múltiplos slots em um componente
- Removido o uso de \$dispatch e \$broadcast

14/10/2016

- Capítulo 4
 - router.map não é mais usado, os parâmetros de configuração são repassados na instância do Vue
 - router.start não é mais usado, ele é iniciado diretamente na instância do Vue
 - A diretiva v-link foi substituída pelo componente <router-link>

Suporte

Para suporte, você pode abrir uma *issue* no github no do seguinte endereço:

<https://github.com/danielschmitz/vue-codigos/issues>

Você pode também sugerir novos capítulos para esta obra.

Código Fonte

Todos os exemplos desta obra estão no github, no seguinte endereço:

<https://github.com/danielschmitz/vue-codigos>

Parte 1 - Conhecendo o Vue

1. Introdução

Seja bem vindo ao mundo Vue (pronuncia-se “view”), um framework baseado em componentes reativos, usado especialmente para criar interfaces web, na maioria das vezes chamadas de SPA - Single Page Application ou aplicações de página única, com somente um arquivo html. Vue.js foi concebido para ser simples, reativo, baseado em componentes, compacto e expansível.

Nesta obra nós estaremos focados na aprendizagem baseada em exemplos práticos, no qual você terá a chance de aprender os conceitos iniciais do framework, e partir para o desenvolvimento de uma aplicação um pouco mais complexa.

1.1 Tecnologias empregadas

Nesta obra usaremos as seguintes tecnologias:

Node

Se você é desenvolvedor Javascript com certeza já conhece o node. Para quem está conhecendo agora, o node pode ser caracterizado como uma forma de executar o Javascript no lado do servidor. Com esta possibilidade, milhares de desenvolvedores criam e publicam aplicações que podem ser usadas pelas comunidade. Graças ao node, o Javascript tornou-se uma linguagem amplamente empregada, ou seria mérito do Javascript possibilitar uma tecnologia como o node? Deixamos a resposta para o leitor.

npm

O **node package manager** é o gerenciador de pacotes do Node. Com ele pode-se instalar as bibliotecas javascript/css existentes, sem a necessidade de realizar o download do arquivo zip, descompactar e mover para o seu projeto. Com npm também podemos, em questão de segundos, ter uma aplicação base pronta para uso. Usaremos muito o **npm** nesta obra. Se você ainda não a usa, com os exemplos mostrados ao longo do livro você terá uma boa base nessa tecnologia.

Editor de textos

Você pode usar qualquer editor de textos para escrever o seu código Vue. Recomenda-se utilizar um editor leve e que possua suporte ao vue, dentre estes temos:

- Sublime Text 3
- Visual Studio Code
- Atom

Todos os editores tem o plugin para dar suporte ao Vue. Nesta obra usaremos extensivamente o Visual Studio Code.

Servidor Web

Em nossos exemplos mais complexos, precisamos comunicar com o servidor para realizar algumas operações com o banco de dados. Esta operação não pode ser realizada diretamente pelo Javascript no cliente. Temos que usar alguma linguagem no servidor. Nesta obra estaremos utilizando o próprio Node, juntamente com o servidor Express para que possamos criar um simples blog devidamente estruturado.

Browserify

Este pequeno utilitário é um “module bundler” capaz de agrupar vários arquivos javascript em um, possibilitando que possamos dividir a aplicação em vários pacotes separados, sendo agrupados somente quando for necessário. Existe outro *modules bundler* chamado webpack, que é mais complexo consequentemente mais poderoso - que deixaremos de lado nessa obra, não por ser ruim, mas por que o browserify atende em tudo que precisamos.

Material Design e materialize-css

Material Design é um conceito de layout criado pelo Google, usado em suas aplicações, como o Gmail, Imbox, Plus etc. O conceito engloba um padrão de design que pode ser usado para criar aplicações. Como os sistemas web usam folha de estilos (CSS), usamos a biblioteca materialize-css, que usa o conceito do material design.

Postman

Para que possamos testar as requisições REST, iremos fazer uso constante do Postman, um plugin para o Google Chrome que faz requisições ao servidor. O Postman irá simular a requisição como qualquer cliente faria, de forma que o programador que trabalha no lado do servidor não precisa necessariamente programar no lado do cliente.

1.2 Instalação do node

Node e npm são tecnologias que você precisa conhecer. Se ainda não teve a oportunidade de usá-las no desenvolvimento web, esse é o momento. Nesta obra, não iremos abordar a instalação de frameworks javascript sem utilizar o npm.

Para instalar o node/npm no Linux, digite na linha de comando:

```
1 sudo apt-get install git node npm
2 sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Para instalar o Node no Windows, acesse [o site oficial](https://nodejs.org/en/)¹ e faça o download da versão estável. Certifique-se de selecionar o item “npm package manager” para instalar o npm também.

Verifique a versão do node no seu ambiente Linux através do comando `node -v`, e caso não seja 5 ou superior, proceda com esta instalação:

```
1 sudo curl -sL https://deb.nodesource.com/setup_6.x | sudo -\
2 E bash -
3 sudo apt-get install -y nodejs
4 sudo ln -s /usr/bin/nodejs /usr/bin/node
```

¹<https://nodejs.org/en/>

1.3 Uso do npm

Será através do *npm* que instalaremos quase todas as ferramentas necessárias para o desenvolvimento. Para compreender melhor como o **npm** funciona, vamos exibir alguns comandos básicos que você irá usar na linha de comando (tanto do Linux, quanto do Windows):

npm init

Este comando inicializa o npm no diretório corrente. Inicializar significa que o arquivo `package.json` será criado, com várias informações sobre o projeto em questão, como o seu nome, a versão do projeto, o proprietário entre outras. Além de propriedades do projeto, também são armazenadas os frameworks e bibliotecas adicionados ao projeto.

npm install ou npm i

Instala uma biblioteca que esteja cadastrada na base do npm, que pode ser acessada [neste endereço](#)². O comando `npm i` produz o mesmo efeito. Quando uma biblioteca é adicionada, o diretório `node_modules` é criado e geralmente a biblioteca é adicionada em `node_modules/nomedabiblioteca`. Por exemplo, para instalar o vue, execute o comando `npm i vue` e perceba que o diretório `node_modules/vue` foi adicionado.

npm i --save ou npm i -S

Já sabemos que `npm i` irá instalar uma biblioteca ou framework. Quando usamos `--save` ou `--S` dizemos ao npm para que este framework seja adicionado também ao arquivo de configuração `package.json`. O framework será referenciado no item `dependencies`.

npm i --saveDev ou npm i -D

Possui um comportamento semelhante ao item acima, só que a configuração do pacote instalado é referenciado no item `devDependencies`. Use a opção `-D` para instalar pacotes que geralmente não fazem parte do projeto principal, mas que são necessários para o desenvolvimento tais como testes unitários, automatização de tarefas, um servidor virtual de teste etc.

²<https://www.npmjs.com/>

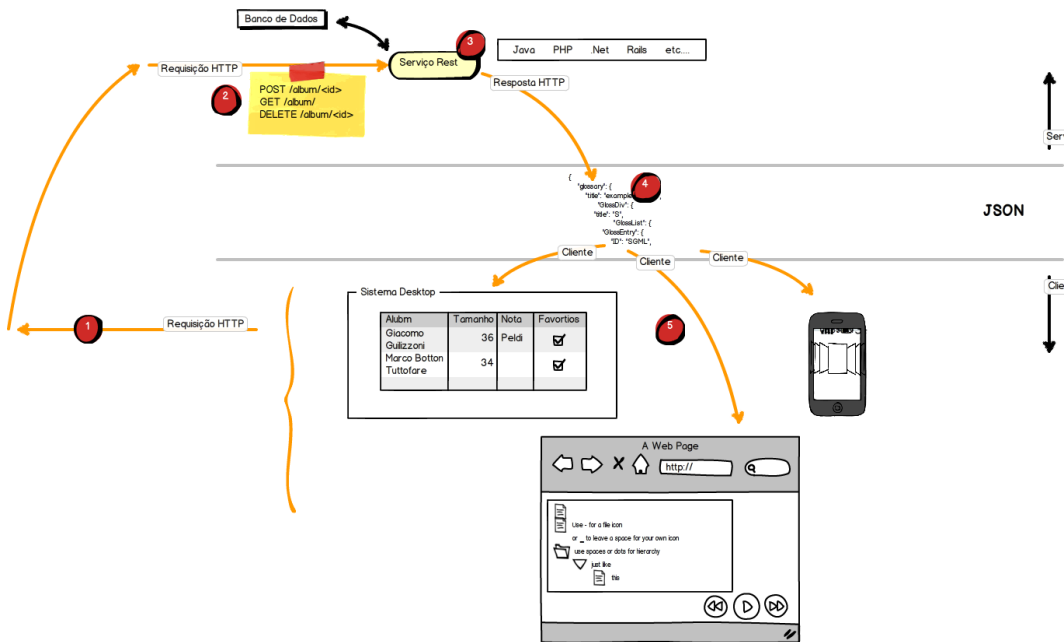
npm i -g

Instala a biblioteca/framework de forma global ao sistema, podendo assim ser utilizado em qualquer projeto. Por exemplo, o `live-server` é um pequeno servidor web que “emula” o diretório atual como um diretório web e cria um endereço para acesso, como `http://localhost:8080`, abrindo o navegador no diretório em questão. Como se usa o `live-server` em quase todos os projetos javascript criados, é comum usar o comando `npm i -g live-sevrer` para que se possa usá-lo em qualquer projeto.

1.4 Conhecendo um pouco o RESTfull

Na evolução do desenvolvimento de sistemas web, os serviços chamados *webservices* estão sendo gradativamente substituídos por outro chamado *RESTful*, que é ‘quase’ a mesma coisa, só que possui um conceito mais simples. Não vamos nos prender em conceitos, mas sim no que importa agora. O que devemos saber é que o Slim Framework vai nos ajudar a criar uma API REST na qual poderemos fazer chamadas através de uma requisição HTTP e obter o resultado em um formato muito mais simples que o XML, que é o JSON.

A figura a seguir ilustra exatamente o porquê do *RESTful* existir. Com ela (e com o slim), provemos um serviço de dados para qualquer tipo de aplicação, seja ela web, desktop ou mobile.



Nesta imagem, temos o ciclo completo de uma aplicação RESTful. Em '1', temos o cliente realizando uma requisição HTTP ao servidor. Todo ciclo começa desta forma, com o cliente requisitando algo. Isso é realizado através de uma requisição http 'normal', da mesma forma que um site requisita informações a um host.

Quando o servidor recebe essa requisição, ele a processa e identifica qual api deve chamar e executar. Nesse ponto, o cliente não mais sabe o que está sendo processado, ele apenas está aguardando a resposta do servidor. Após o processamento, o servidor responde ao cliente em um formato conhecido, como o json. Então, o que temos aqui é o cliente realizando uma consulta ao servidor em um formato conhecido (http) e o servidor respondendo em json.

Desta forma, conseguimos garantir uma importância muito significativa entre servidor e cliente. Ambos não se conhecem, mas sabem se comunicar entre si. Assim, tanto faz o cliente ser um navegador web ou um dispositivo mobile. Ou tanto faz o servidor ser PHP ou Java, pois a forma de conversa entre elas é a mesma.

2. Conhecendo Vue.js

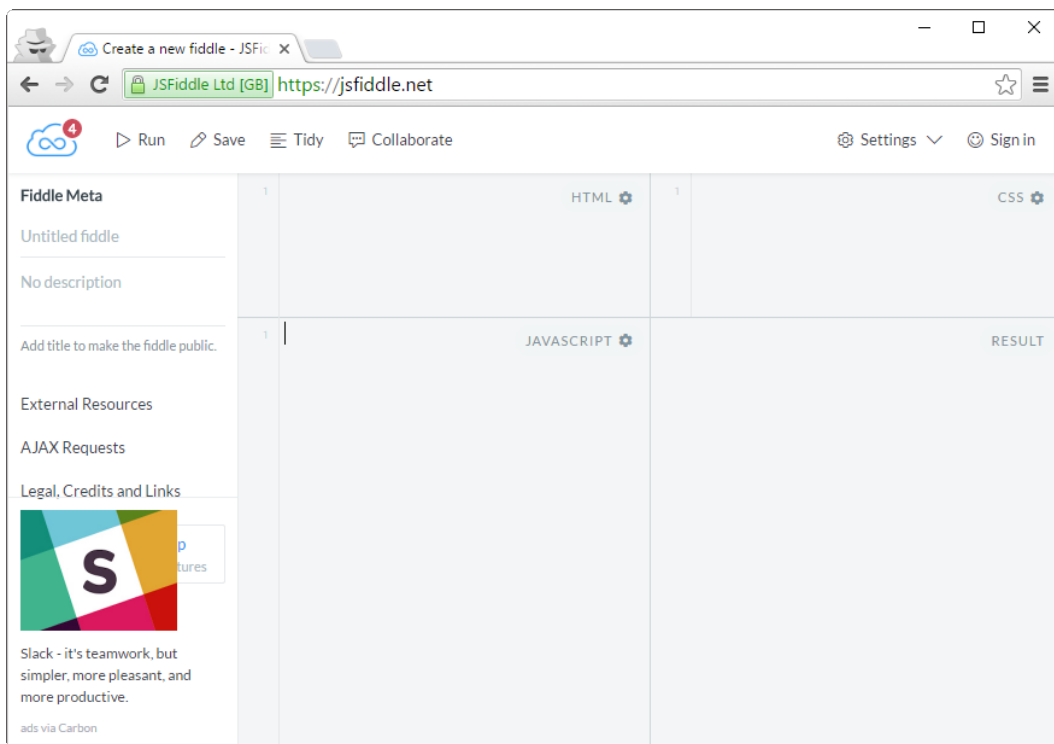
Neste capítulo iremos aprender alguns conceitos básicos sobre o Vue. Não veremos (ainda) a instalação do mesmo, porque como estamos apresentando cada conceito em separado, é melhor usarmos um editor online, neste caso o jsFiddle.

2.1 Uso do jsFiddle

jsFiddle é um editor html/javascript/css online, sendo muito usado para aprendizagem, resolução de problemas rápidos e pequenos testes. É melhor utilizar o jsFiddle para aprendermos alguns conceitos do Vue do que criar um projeto e adicionar vários arquivos.

Acesse no seu navegador o endereço jsfiddle.net¹. Você verá uma tela semelhante a figura a seguir:

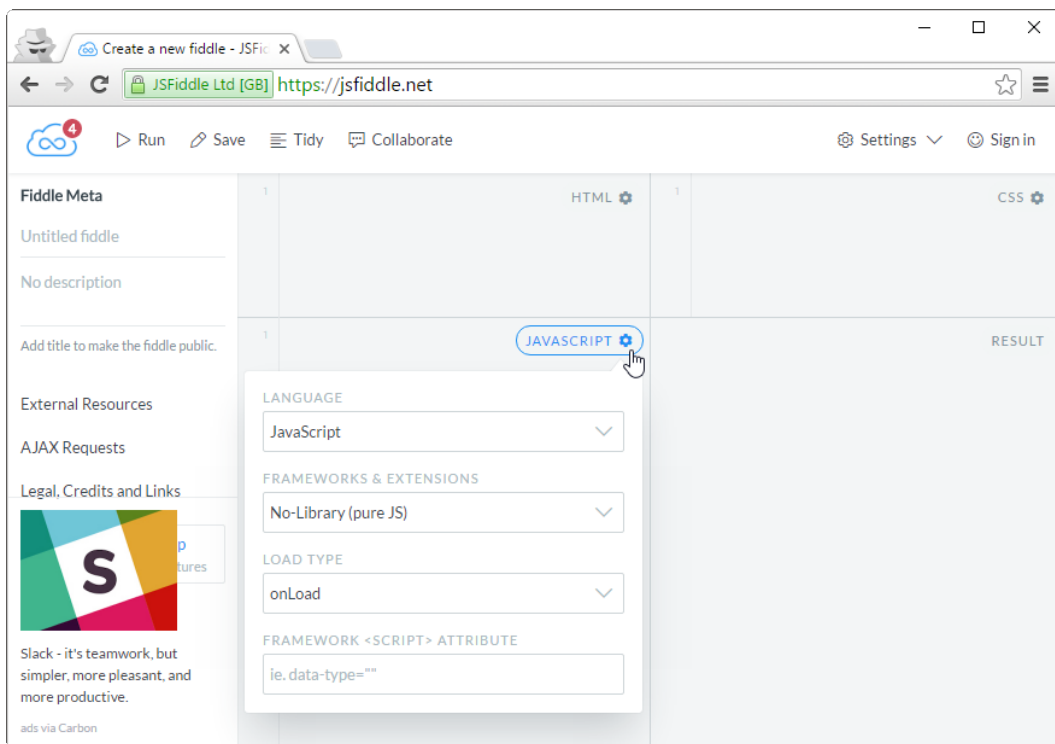
¹jsfiddle.net



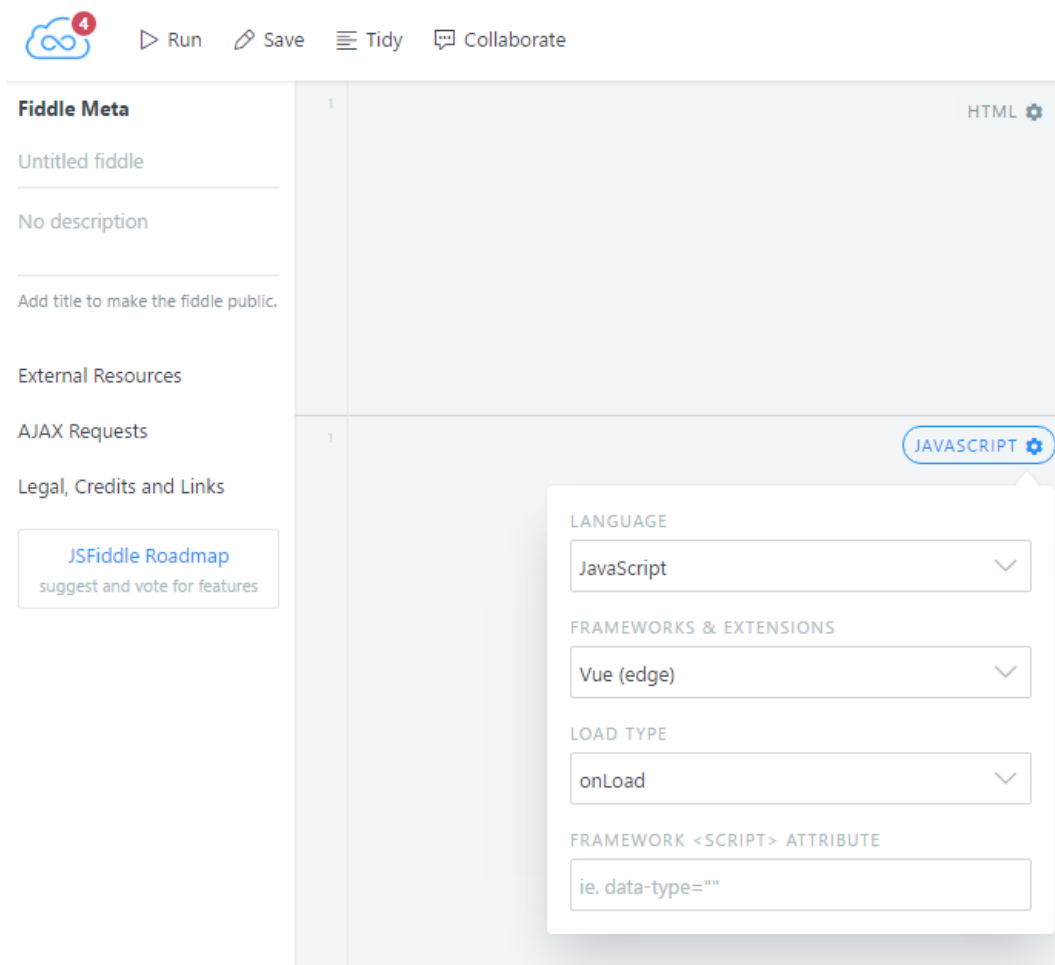
Caso queira, pode criar uma conta e salvar todos os códigos que criar, para poder consultar no futuro. No jsFiddle, temos 4 áreas sendo elas: *html*, *javascript*, *css* e *result*. Quando clicamos no botão Run, o html/javascript/css são combinados e o resultado é apresentado.

2.2 Configurando o jsFiddle para o Vue

Para que possamos usar o jsFiddle em conjunto com o vue, clique no ícone de configuração do javascript, de acordo com a figura a seguir:



Na caixa de seleção Frameworks & Extensions, encontre o item Vue e escolha a versão Vue (edge), deixando a configuração desta forma:



Agora que o jsFiddle está configurado com o Vue, podemos começar nosso estudo inicial com vue.

2.3 Hello World, vue

Para escrever um Hello World com vue apresentamos o conceito de databind. Isso significa que uma variável do vue será ligada diretamente a alguma variável no html.

Comece editando o código html, adicionando o seguinte texto:

```
1 <div id="app">
2   {{msg}}
3 </div>
```

Neste código temos dois detalhes. O primeiro, criamos uma `div` cujo o `id` é `app`. No segundo, usamos `{{ e }}` para adicionar uma variável chamada `msg`. Esta variável será preenchida pelo `vue`.

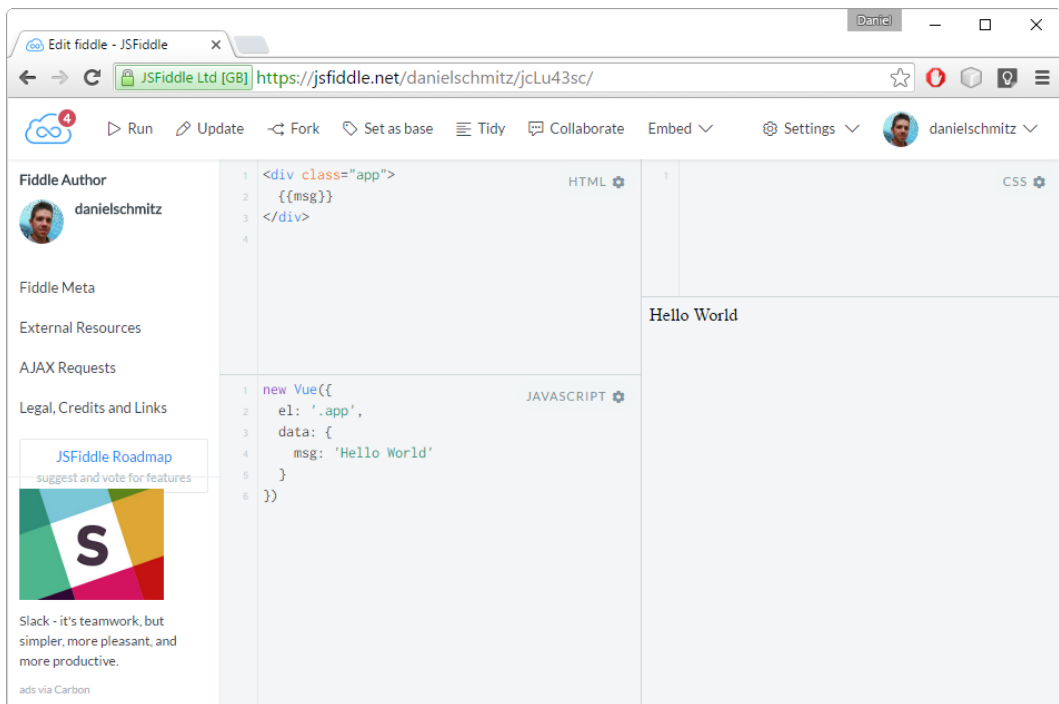
Agora vamos à parte Javascript. Para trabalhar com `vue`, basta criar um objeto `Vue` e repassar alguns parâmetros, veja:

```
1 new Vue({
2   el: '#app',
3   data: {
4     msg: 'Hello World'
5   }
6 })
```

O objeto criado `new Vue()` possui uma configuração no formato JSON, onde informamos a propriedade `el`, que significa o elemento em que esse objeto `vue` será aplicado no documento html. Com o valor `#app`, estamos apontando para a `div` cujo `id` é `app`.

A propriedade `data` é uma propriedade especial do `Vue` no qual são armazenadas todas as variáveis do objeto `vue`. Essas variáveis podem ser usadas tanto pelo próprio objeto `vue` quanto pelo `data-bind`. No nosso exemplo, a variável `data.msg` será ligada ao `{{msg}}` do html.

Após incluir estes dois códigos, clique no botão `Run` do `jsFiddle`. O resultado será semelhante à figura a seguir.

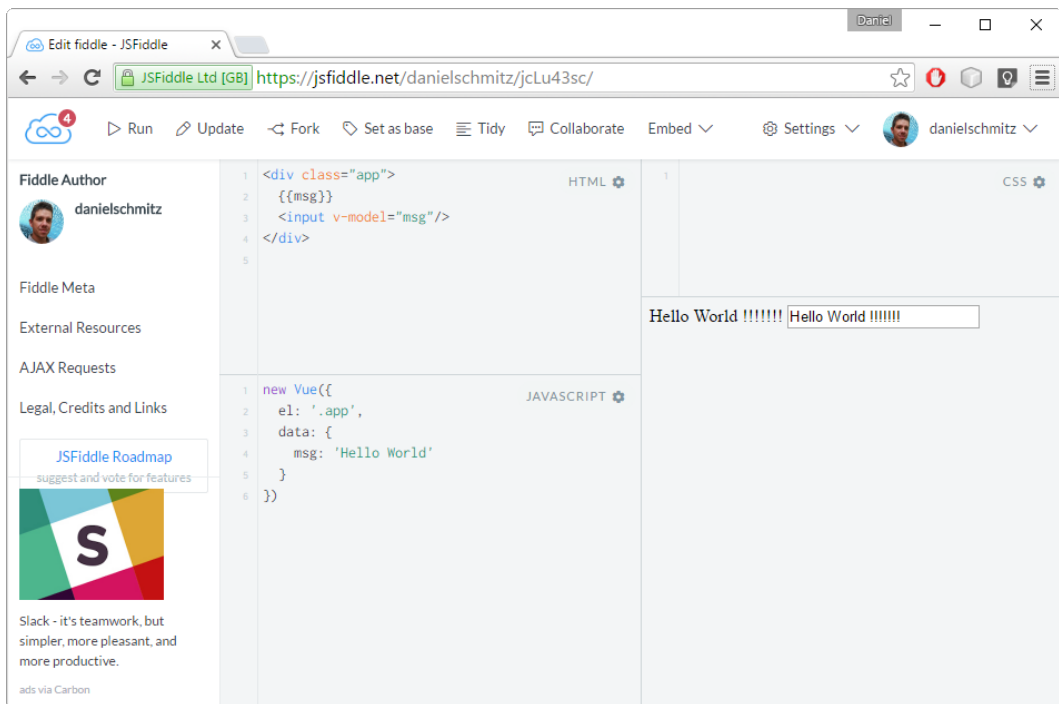


2.4 Two way databind

O conceito de “two-way” permite que o vue possa observar uma variável qualquer e atualizar o seu valor a qualquer momento. No exemplo a seguir, criamos um campo para alterar o valor da variável `msg`.

```
1 <div id="app">
2   {{msg}}
3   <input v-model="msg"/>
4 </div>
```

Veja que o elemento `input` possui a propriedade `v-model`, que é uma propriedade do vue. Ela permite que o vue observe o valor do campo *input* e atualize a variável `msg`. Quando alterarmos o valor do campo, a variável `data.msg` do objeto vue é alterada, e suas referências atualizadas. O resultado é semelhante à figura a seguir:



2.5 Criando uma lista

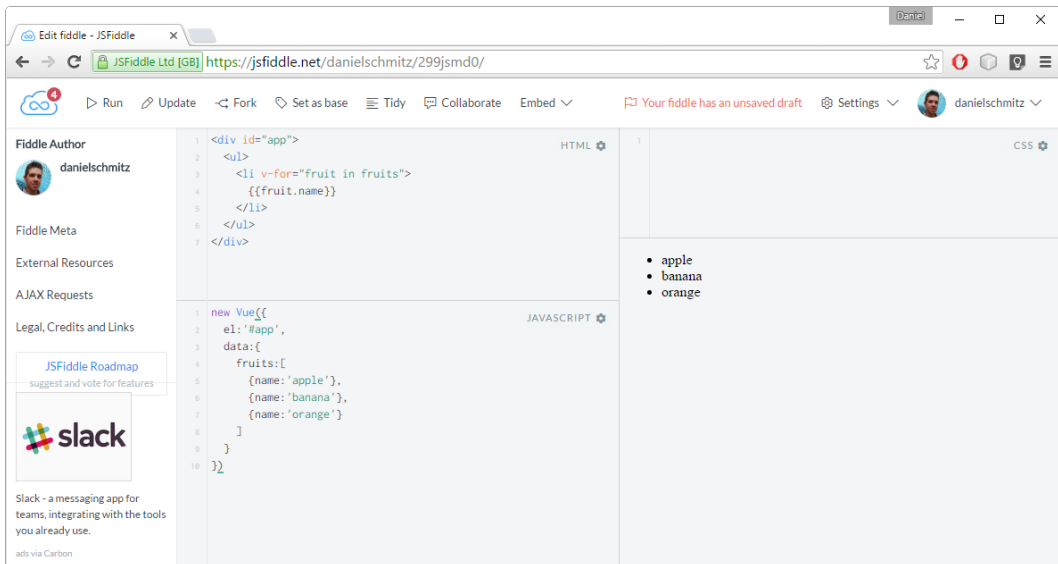
Existem dezenas de comandos do vue que iremos aprender ao longo desta obra. Um deles é o `v-for` que faz um loop no elemento em que foi inserido. Crie um novo jsFiddle com o seguinte código:

```
1 <div id="app">
2   <ul>
3     <li v-for="fruit in fruits">
4       {{fruit.name}}
5     </li>
6   </ul>
7 </div>
```

Veja que usamos `v-for` no elemento ``, que irá se repetir de acordo com a variável `fruits` declarada no objeto `vue`:

```
1 new Vue({
2   el: '#app',
3   data: {
4     fruits: [
5       {name: 'apple'},
6       {name: 'banana'},
7       {name: 'orange'}
8     ]
9   }
10 })
```

No objeto `vue`, cria-se o array `fruits` na propriedade `data`. O resultado é exibido a seguir:



2.6 Detectando alterações no Array

Vue consegue observar as alterações nos Arrays, fazendo com que as alterações no html sejam refletidas. O métodos que o Vue consegue observar são chamados de métodos modificadores, listados a seguir:

push()

Adiciona um item ao Array

pop()

Remove o último elemento do Array

shift()

Remove o primeiro elemento do Array

unshift()

Adiciona novos itens no início de um Array

splice()

Adiciona itens no Array, onde é possível informar o índice dos novos itens.

sort()

Ordena um Array

reverse()

inverte os itens de um Array

Existem métodos que o Vue não consegue observar, chamados de *não modificadores*, tais como `filter()`, `concat()` e `slice()`. Estes métodos não provocam alterações no array original, mas retornam um novo Array que, para o Vue, poderiam ser sobrepostos inteiramente.

Por exemplo, ao usarmos `filter`, um novo array será retornado de acordo com a expressão de filtro que for usada. Quando substituímos esse novo array, pode-se imaginar que o Vue irá remover o array antigo e recriar toda a lista novamente, mas ele não faz isso! Felizmente ele consegue detectar as alterações nos novos elementos do array e apenas atualizar as suas referências. Com isso substituir uma lista inteira de elementos nem sempre ocasiona em uma substituição completa na DOM.

Utilizando v-bind:key

Imagine que temos uma tabela com diversos registros sendo exibidos na página. Esses registros são originados em uma consulta Ajax ao servidor. Suponha que exista um filtro que irá realizar uma nova consulta, retornando novamente novos dados do servidor, que obviamente irão atualizar toda a lista de registros da tabela.

Perceba que, a cada pesquisa, uma nova lista é gerada e atualizada na tabela, o que pode se tornar uma operação mais complexa para a DOM, resultado até mesmo na substituição de todos os itens, caso o `v-for` não consiga compreender que os itens alterados são os mesmos.

Podemos ajudar o Vue nesse caso através da propriedade `key`, na qual iremos informar ao Vue qual propriedade da lista de objetos deverá ser mapeada para que o Vue possa manter uma relação entre os itens antes e após a reconsulta no servidor. Suponha, por exemplo, que a consulta no servidor retorne objetos que tenham uma propriedade chamada `_uid`, com os seguintes dados:

```
1 {  
2   items: [  
3     { _uid: '88f869d', ... },  
4     { _uid: '7496c10', ... }  
5   ]  
6 }
```

Então pode-se dizer ao `v-for` que use essa propriedade como base da lista, da seguinte forma:

```
1 <div v-for="item in items" v-bind:key="_uid">  
2  
3 </div>
```

Desta forma, quando o Vue executar uma consulta ao servidor, e este retornar com novos dados, o `v-for` saberá pelo `uid` que alguns registros não se alteraram, e manterá a DOM intacta.

Uso do set

Devido a uma limitação do Javascript, o Vue não consegue perceber uma alteração no item de um array na seguinte forma:

```
1 this.fruits[0] = {}
```

Quando executamos o código acima, o item que pertence ao índice 0 do array, mesmo que alterado, não será atualizado na camada de visualização da página.

Para resolver este problema, temos que usar um método especial do Vue chamado `set`, da seguinte forma:

```
1 this.fruits.set(0, {name: 'foo'});
```

O uso do `set` irá forçar a atualização do Vue na lista html.

Como remover um item

Para que se possa remover um item, usamos o método `Array.prototype.splice` conforme o exemplo a seguir:

```
1 methods: {  
2   removeTodo: function (todo) {  
3     var index = this.todos.indexOf(todo)  
4     this.todos.splice(index, 1)  
5   }  
6 }
```

Loops em objetos

Pode-se realizar um loop entre as propriedades de um objeto da seguinte forma:

```
1 <ul id="app" >
2   <li v-for="(value, key) in object">
3     {{ key }} : {{ value }}
4   </li>
5 </ul>
```

Onde `key` é o nome da propriedade do objeto, e `value` é o valor desta propriedade. Por exemplo, em um objeto `{ "id": 5 }`, o valor de `key` será `id` e o valor de `value` será `5`.

2.7 Eventos e métodos

Podemos capturar diversos tipos de eventos e realizar operações com cada um deles. No exemplo a seguir, incluímos um botão que irá alterar uma propriedade do `vue`.

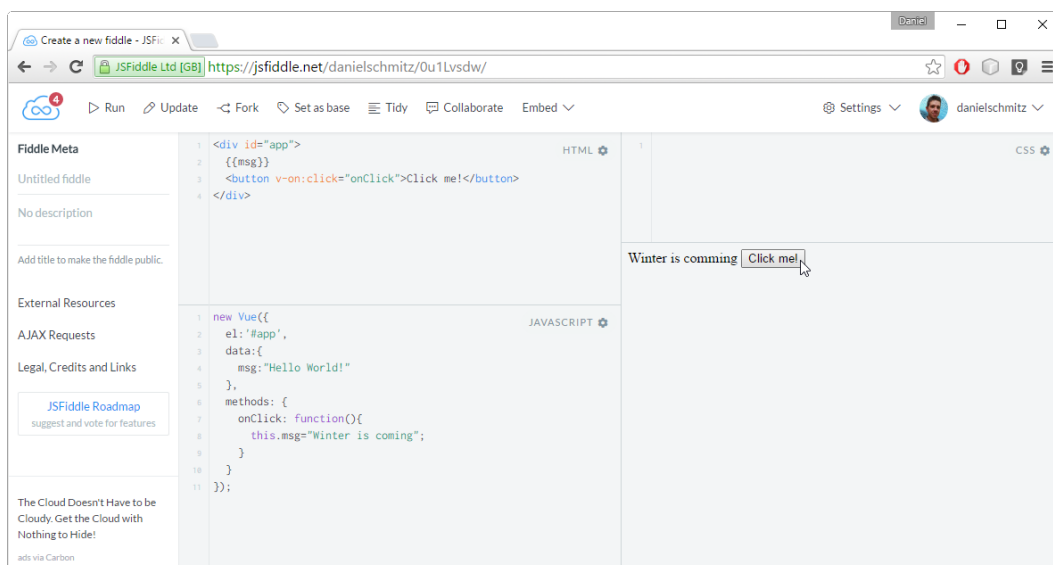
```
1 <div id="app">
2   {{ msg }}
3   <button v-on:click="onClick">Click me!</button>
4 </div>
```

No elemento `button` temos a captura do evento `click` pelo `vue`, representado por `v-on:click`. Esta captura irá executar o método `onClick`, que estará declarado no objeto `Vue`. O objeto é exibido a seguir:

```
1 new Vue({
2   el: '#app',
3   data: {
4     msg: "Hello World!"
5   },
6   methods: {
7     onClick: function(){
8       this.msg="Winter is coming";
```

```
9      }  
10    }  
11  });
```

O objeto `vue` possui uma nova propriedade chamada `methods`, que reúne todos os métodos que podem ser referenciados no código html. O método `onClick` possui em seu código a alteração da variável `msg`. O resultado deste código após clicar no botão é exibida a seguir:



Modificando a propagação do evento

Quando clicamos em um botão ou link, o evento “click” irá executar o método indicado pelo `v-on:click` e, além disso, o evento se propagará até o navegador conseguir capturá-lo. Essa é a forma natural que o evento se comporta em um navegador.

Só que, às vezes, é necessário capturar o evento `click` mas não é desejável que ele se propague. Quando temos esta situação, é natural chamar o método `preventDefault` ou `stopPropagation`. O exemplo a seguir mostra como um evento pode ter a sua propagação cancelada.


```
1 <button v-on:click="say('hello!', $event)">
2   Submit
3 </button>
```

e:

```
1 methods: {
2   say: function (msg, event) {
3     event.preventDefault()
4     alert(msg)
5   }
6 }
```

O vue permite que possamos cancelar o evento ainda no html, sem a necessidade de alteração no código javascript, da seguinte forma:

```
1 <!-- a propagação do evento Click será cancelada -->
2 <a v-on:click.stop="doThis"></a>
3
4 <!-- o evento de submit não irá mais recarregar a página -->
5 <form v-on:submit.prevent="onSubmit"></form>
6
7 <!-- os modificadores podem ser encadeados -->
8 <a v-on:click.stop.prevent="doThat"></a>
```

Modificadores de teclas

Pode-se usar o seguinte modificador `v-on:keyup.13` para capturar o evento “keyup 13” do teclado que corresponde a tecla enter. Também pode-se utilizar:

```
1 <input v-on:keyup.enter="submit">  
2 ou  
3 <input @keyup.enter="submit">
```

Algumas teclas que podem ser associadas:

- enter
- tab
- delete
- esc
- space
- up
- down
- left
- right

2.8 Design reativo

Um dos conceitos principais do Vue é o que chamamos de design reativo, onde elementos na página são alterados de acordo com o estado dos objetos gerenciados pelo Vue. Ou seja, não há a necessidade de navegar pela DOM (Document Object Model) dos elementos HTML para alterar informações.

2.9 Criando uma lista de tarefas

Com o pouco que vimos até o momento já podemos criar uma pequena lista de tarefas usando os três conceitos aprendidos até o momento:

- Pode-se armazenar variáveis no objeto Vue e usá-las no html
- Pode-se alterar o valor das variáveis através do two way databind
- Pode-se capturar eventos e chamar funções no objeto Vue

No código html, vamos criar um campo *input* para a entrada de uma tarefa, um botão para adicionar a tarefa e, finalmente, uma lista de tarefas criadas:

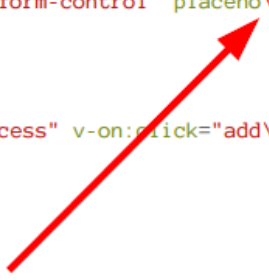
```
1 <div id="app" class="container">
2
3   <div class="row">
4     <div class="col-xs-8">
5       <input type="text" class="form-control" placeholder="\
6 Add a task" v-model="inputTask">
7     </div>
8     <div class="col-xs-4">
9       <button class="btn btn-success" v-on:click="addTask">
10         Adicionar
11       </button>
12     </div>
13   </div>
14   <br/>
15   <div class="row">
16     <div class="col-xs-10">
17       <table class="table">
18         <thead>
19           <tr>
20             <th>Task Name</th>
21             <th></th>
22           </tr>
23         </thead>
24         <tbody>
25           <tr v-for="task in tasks">
26             <td class="col-xs-11">
27               {{task.name}}
28             </td>
29             <td class="col-xs-1">
30               <button class="btn btn-danger" v-on:click="re\
31 moveTask(task.id)">
32                 x
33               </button>
```

```
34         </td>
35     </tr>
36 </tbody>
37 </table>
38 </div>
39 </div>
40 </div>
```

Quebra de linha no código fonte

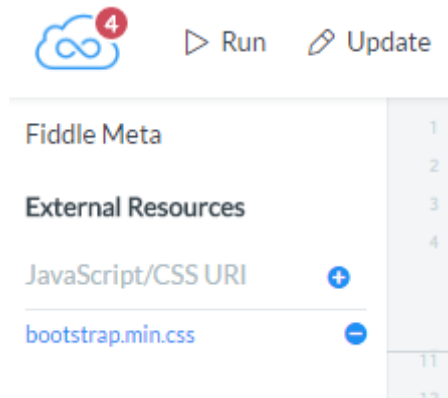
Cuidado com a quebra de página nos códigos desta obra. Como podemos ver na imagem a seguir:

```
<div class="row">
  <div class="col-xs-8">
    <input type="text" class="form-control" placeholder="Add a task" v-model="inputTask">
  </div>
  <div class="col-xs-4">
    <button class="btn btn-success" v-on:click="addTask">
      Adicionar
    </button>
  </div>
</div>
```



Quando uma linha quebra por não caber na página, é adicionado uma contra barra, que deve ser omitida caso você esteja copiando o código diretamente do arquivo PDF/EPUB desta obra.

Neste código HTML podemos observar o uso de classes nos elementos da página. Por exemplo, a primeira <div> contém a classe container. O elemento input contém a classe form-control. Essas classes são responsáveis em estilizar a página, e precisam de algum framework css funcionando em conjunto. Neste exemplo, usamos o Bootstrap, que pode ser adicionado no jsFiddle de acordo com o detalhe a seguir:



O endereço do arquivo adicionado é:

<https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css>

Com o bootstrap adicionado, estilizar a aplicação torna-se uma tarefa muito fácil. Voltando ao html, criamos uma caixa de texto que possui como `v-model` o valor `inputTask`. Depois, adicionamos um botão que irá chamar o método `addTask`. A lista de tarefas é formada pelo elemento `table` e usamos `v-for` para criar um loop nos itens do array `tasks`.

O loop preenche as linhas da tabela, onde repassamos a propriedade `name` e usamos a propriedade `id` para criar um botão para remover a tarefa. Perceba que o botão que remove a tarefa tem o evento `click` associado ao método `removeTask(id)` onde é repassado o `id` da tarefa.

Com o html pronto, já podemos estabelecer que o objeto Vue terá duas variáveis, `inputTask` e `tasks`, além de dois métodos `addTask` e `removeTask`. Vamos ao código javascript:

```
1  new Vue({
2    el: '#app',
3    data: {
4      tasks: [
5        {id:1,name:"Learn Vue"},
6        {id:2,name:"Learn Npm"},
7        {id:3,name:"Learn Sass"}
8      ],
9      inputTask: ""
10   },
11   methods: {
12     addTask(){
13       if (this.inputTask.trim()!=""){
14         this.tasks.push(
15           {name:this.inputTask,
16             id:this.tasks.length+1}
17         )
18         this.inputTask="";
19       }
20     },
21     removeTask(id){
22       for(var i = this.tasks.length; i--;) {
23         if(this.tasks[i].id === id) {
24           this.tasks.splice(i, 1);
25         }
26       }
27     }
28   }
29 })
```

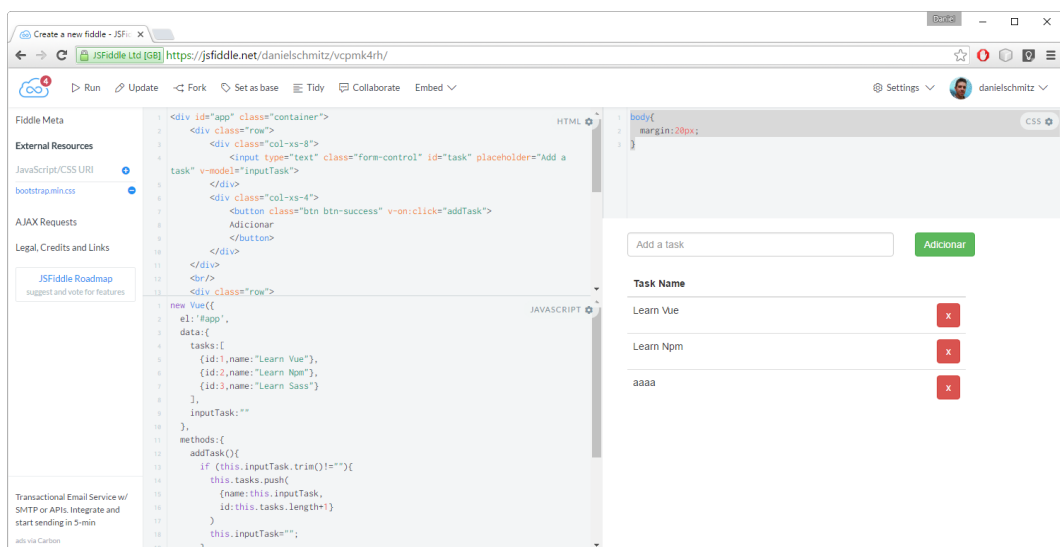
O código Vue, apesar de extenso, é fácil de entender. Primeiro criamos as duas variáveis: `tasks` possui um array de objetos que será a base da tabela que foi criada no html. Já a variável `inputTask` realiza um two way databind com a caixa de texto

do formulário, onde será inserida a nova tarefa.

O método `addTask()` irá adicionar uma nova tarefa a lista de tarefas `this.tasks`. Para adicionar esse novo item, usamos na propriedade `id` a quantidade de itens existentes da lista de tarefas.

O método `removeTask(id)` possui um parâmetro que foi repassado pelo botão html, e é através deste parâmetro que removemos a tarefa da lista de tarefas.

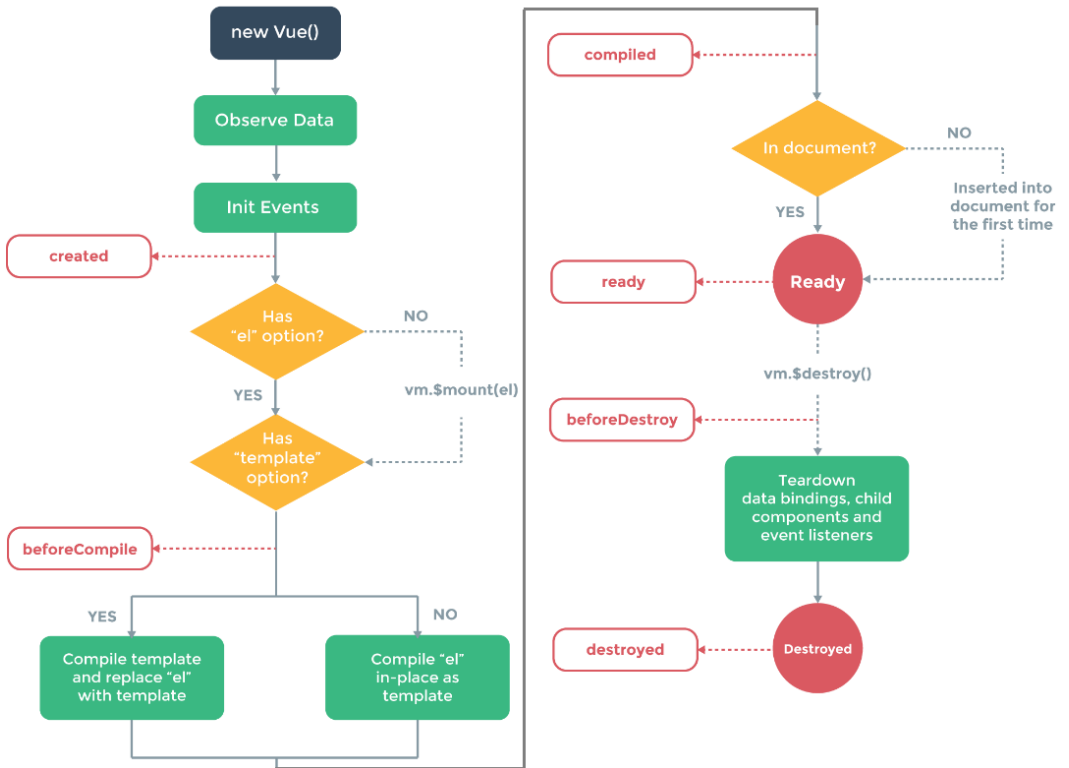
O resultado deste código pode ser visto na figura a seguir. Perceba que o formulário e a lista de tarefas possui o estilo do bootstrap.



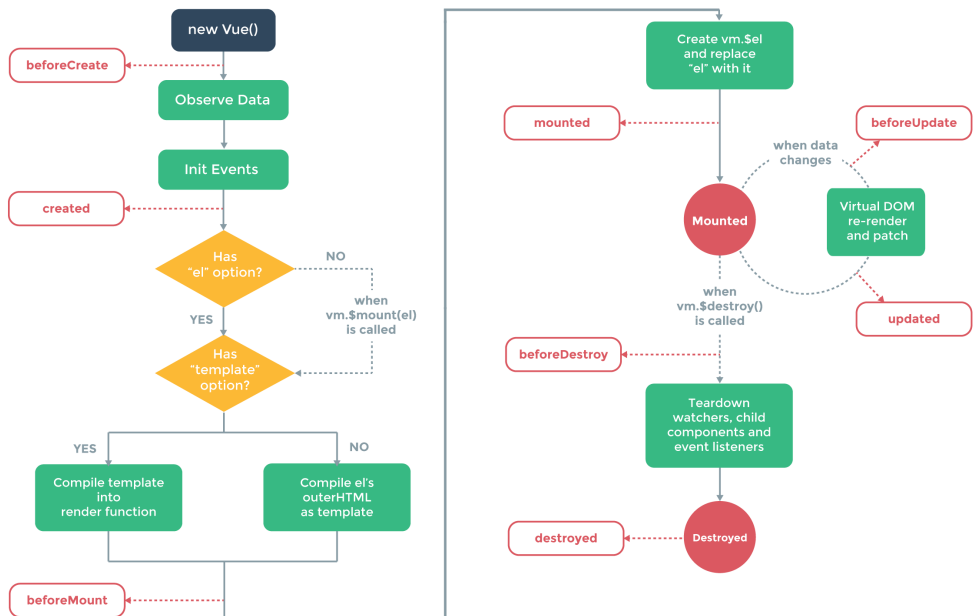
Caso queira ter acesso direto a este código pelo jsFiddle, (clique aqui)[<https://jsfiddle.net/tbg8fo51>]

2.10 Eventos do ciclo de vida do Vue

Quando um objeto Vue é instanciado, ele possui um ciclo de vida completo, desde a sua criação até a finalização. Este ciclo é descrito pela imagem a seguir:



Vue 1:



Vue 2:

Em vermelho, temos os eventos que são disparados durante este ciclo, e que podem ser usados em determinadas ocasiões no desenvolvimento de sistemas. Os eventos que podem ser capturados são: `beforeCreated`, `created`, `beforeMount`, `mounted`, `beforeUpdate`, `updated`, `beforeDestroy`, `destroyed`.

Para realizar requisições ajax, costuma-se utilizar o evento `updated`. Veremos com mais detalhes este tópico quando abordarmos `vue-resource`.

2.11 Compreendendo melhor o Data Bind

Existem alguns pequenos detalhes que precisamos abordar sobre o databind. Já sabemos que, ao usar `{{ }}`, conseguimos referenciar uma variável do Vue diretamente no html.

Databind único

Caso haja a necessidade de aplicar o data-bind somente na primeira vez que o Vue for iniciado, deve-se usar `v-once`, conforme o código a seguir:

```
1 <span> Message: {{ msg }} </span>
```

Databind com html

O uso de `{{ e }}` não formata o html por uma simples questão de segurança. Isso significa que, se você tiver na variável `msg` o valor `Hello World`, ao usar `{{msg}}` a resposta ao navegador será `Hello World`, de forma que as tags do html serão exibidas, mas não formatadas.

Para que você possa formatar código html no databind, é necessário usar três a diretiva `v-html`, como por exemplo `{{msg}}>`. Desta forma, o valor `Hello World` será exibido no navegador em negrito.

Databind em Atributos

Para usar `data-bind` em atributos, use a diretiva `v-bind`, como no exemplo a seguir:

```
1 <button v-bind:class="'btn btn-' + size"></button>
```

Expressões

Pode-se usar expressões dentro do databind, como nos exemplos a seguir:

```
{{ number + 1 }}
```

Se `number` for um número e estiver declarado no objeto `Vue`, será adicionado o valor 1 à ele.

```
{{ ok ? 'YES' : 'NO' }}
```

Se `ok` for uma variável booleana, e se for verdadeiro, o texto `YES` será retornado. Caso contrário, o texto `NO` será retornado.

```
{{ message.split('').reverse().join('') }}
```

Nesta expressão o conteúdo da variável `message` será quebrada em um array, onde cada letra será um item deste array. O método `reverse()` irá reverter os índices do array e o método `join` irá concatenar os itens do array em uma string. O resultado final é uma string com as suas letras invertidas.

É preciso ter alguns cuidados em termos que são sentenças e não expressões, como nos exemplos `{{ var a = 1 }}` ou então `{{ if (ok) {return message} }}`. neste caso não irão funcionar

2.12 Filtros

Filtros são usados, na maioria das vezes, para formatar valores de databind. O exemplo a seguir irá pegar todo o valor de `msg` e transformar a primeira letra para maiúscula.

```
1 <div>
2   {{ msg | capitalize }}
3 </div>
```

Na versão 2.0 do Vue, os filtros somente funcionam em expressões `{{ ... }}`. Filtros em laços `v-for` não são mais usados.

O nome do filtro a ser aplicado deve estar após o caractere *pipe* |. *Capitalize* é somente um dos filtros disponíveis. Existem alguns pré configurados, veja:

uppercase

Converte todas as letras para maiúscula

lowercase

Converte todas as letras para minúscula

currency

Converte um valor para moeda, incluindo o símbolo '\$' que pode ser alterado de acordo com o seguinte exemplo:

```
1 {{ total | currency 'R$' }}
```

pluralize

Converte um item para o plural de acordo com o valor do filtro. Suponha que tenhamos uma variável chamada `amount` com o valor 1. Ao realizarmos o filtro `{{ amount | pluralize 'item' }}` teremos a resposta “item”. Se o valor `amount` for dois ou mais, teremos a resposta “items”.

json

Converte um objeto para o formato JSON, retornando a sua representação em uma string.

2.13 Diretivas

As diretivas do Vue são propriedades especiais dos elementos do html, geralmente se iniciam pela expressão `v-` e possui as mais diversas funcionalidades. No exemplo a seguir, temos a diretiva `v-if` em ação:

```
1 <p v-if="greeting">Hello!</p>
```

O elemento `<p>` estará visível ao usuário somente se a propriedade `greeting` do objeto Vue estiver com o valor `true`.

Ao invés de exibir uma lista completa de diretivas (pode-se consultar a [api](http://vuejs.org/api)², sempre), vamos apresentar as diretivas mais usadas ao longo de toda esta obra.

Na versão do Vue 2, a criação de diretivas personalizadas não é encorajada, pois o Vue foca principalmente na criação de componentes.

Argumentos

Algumas diretivas possuem argumentos que são estabelecidos após o uso de dois pontos. Por exemplo, a diretiva `v-bind` é usada para atualizar um atributo de qualquer elemento html, veja:

²<http://vuejs.org/api>

```
1 <a v-bind:href="url"></a>
2
3 é o mesmo que:
4
5 <a :href="url"></a>
6
7 ou então:
8
9 <a href="{{url}}"></a>
```

Outro exemplo comum é na diretiva `v-on`, usada para registrar eventos. O evento `click` pode ser registrado através do `v-on:click`, assim como a tecla `enter` pode ser associada por `v-on:keyup.enter`.

Modificadores

Um modificador é um elemento que ou configura o argumento da diretiva. Vimos o uso do modificador no exemplo anterior, onde `v-on:keyup.enter` possui o modificar “`enter`”. Todo modificador é configurado pelo ponto, após o uso do argumento.

2.14 Atalhos de diretiva (Shorthands)

Existem alguns atalhos muito usados no Vue que simplificam o código html. Por exemplo, ao invés de termos:

```
1 <input v-bind:type="form.type"
2 v-bind:placeholder="form.placeholder"
3 v-bind:size="form.size">
```

Podemos usar o atalho:

```
1 <input :type="form.type"  
2 :placeholder="form.placeholder"  
3 :size="form.size">
```

Ou seja, removemos o “v-bind:” e deixamos apenas p “:”. Sempre quando ver um “:” nos elementos que o Vue gerencia, lembre-se do v-bind.

Outro atalho muito comum é na manipulação de eventos, onde trocamos o v-on: por @. Veja o exemplo:

```
1 <!-- full syntax -->  
2 <a v-on:click="doSomething"></a>  
3  
4 <!-- shorthand -->  
5 <a @click="doSomething"></a>
```

2.15 Alternando estilos

Uma das funcionalidades do design reativo é possibilitar a alteração de estilos no código html dado algum estado de variável ou situação específica.

No exemplo da lista de tarefas, no método addTask, tínhamos a seguinte verificação no código javascript:

```
1 addTask(){  
2   if (this.inputTask.trim()!=""){  
3     //.....  
4   }  
5 }
```

Com design reativo podemos alterar, por exemplo, o botão que inclui a tarefa para:

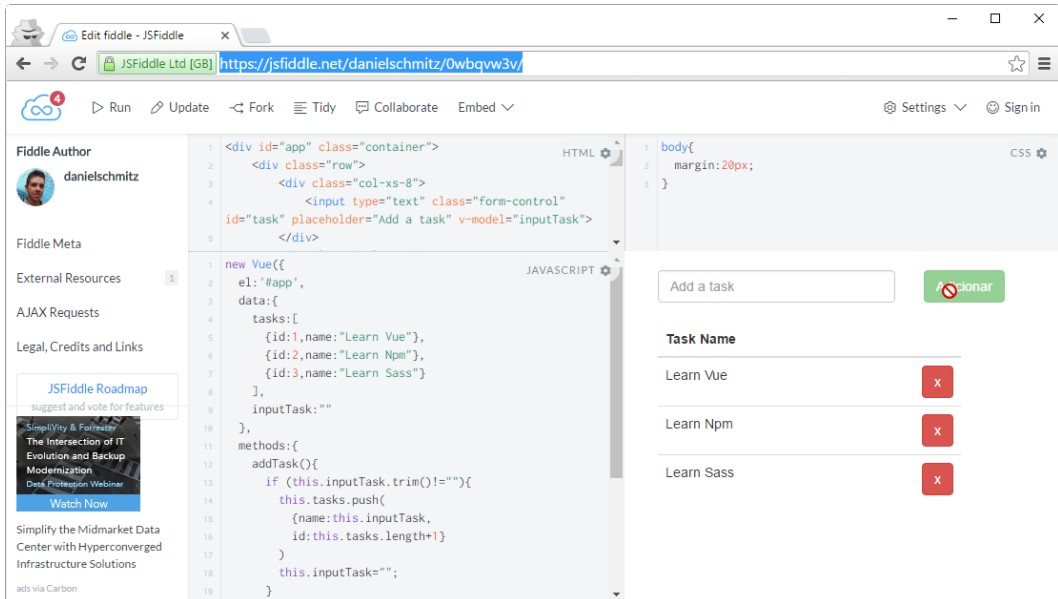
```

1 <button class="btn btn-success"
2 :class="{ 'disabled': inputTask.trim() === '' }"
3 @click="addTask"
4 >

```

Teste esta variação neste [link](https://jsfiddle.net/danielschmitz/0wbqvw3v/)³

Veja que adicionamos o *shorthand* :class incluindo a classe disabled do bootstrap, fazendo a mesma verificação para que, quando não houver texto digitado na caixa de texto, o botão Adicionar ficará desabilitado, conforme a figura a seguir.



2.16 Uso da condicional v-if

O uso do v-if irá exibir o elemento html de acordo com alguma condição. Por exemplo:

³<https://jsfiddle.net/0wbqvw3v/1/>

```
1 <h1 v-if="showHelloWorld">Hello World</h1>
```

É possível adicionar `v-else` logo após o `v-if`, conforme o exemplo a seguir:

```
1 <h1 v-if="name===' '>Hello World</h1>
2 <h1 v-else>Hello {{name}}</h1>
```

O `v-else` tem que estar imediatamente após o `v-if` para que possa funcionar.

A diretiva `v-if` irá incluir ou excluir o item da DOM do html. Caso haja necessidade de apenas omitir o elemento (usando `display:none`), usa-se `v-show`.

2.17 Exibindo ou ocultando um bloco de código

Caso haja a necessidade de exibir um bloco de código, pode-se inserir `v-if` em algum elemento html que contém esse bloco. Se não houver nenhum elemento html englobando o html que se deseja tratar, pode-se usar o componente `<template>`, por exemplo:

```
1 <template v-if="ok">
2   <h1>Title</h1>
3   <p>Paragraph 1</p>
4   <p>Paragraph 2</p>
5 </template>
```

2.18 v-if vs v-show

A diferença principal entre `v-if` e `v-show` está na manipulação do DOM do html. Quando usa-se `v-if`, elementos são removidos ou inseridos na DOM, além de todos os data-binds e eventos serem removidos ou adicionados também. Isso gera um custo de processamento que pode ser necessário em determinadas situações.

Já o `v-show` usa estilos apenas para esconder o elemento da página, mas não da DOM, o que possui um custo baixo, mas pode não ser recomendado em algumas situações.

2.19 Formulários

O uso de formulários é amplamente requisitado em sistemas ambientes web. Quanto melhor o domínio sobre eles mais rápido e eficiente um formulário poderá ser criado.

Para ligar um campo de formulário a uma variável do Vue usamos `v-model`, da seguinte forma:

```
1 <span>Message is: {{ message }}</span>
2 <br>
3 <input type="text" v-model="message">
```

Checkbox

Um input do tipo checkbox deve estar ligado a uma propriedade do `v-model` que seja booleana. Se um mesmo `v-model` estiver ligado a vários checkboxes, um array com os itens selecionados será criado.

Exemplo:

```
1 <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
2 Jack</label>
3 <input type="checkbox" id="john" value="John" v-model="checkedNames">
4 John</label>
5 <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
6 Mike</label>
7 <br>
8 <span>Checked names: {{ checkedNames | json }}</span>
```

```
1 new Vue({  
2   el: '...',  
3   data: {  
4     checkedNames: []  
5   }  
6 })
```

Resultado:



☒ Jack ☒ John ☐ Mike
Checked names: ["Jack", "John"]

Radio

Campos do tipo Radio só aceitam um valor. Para criá-los, basta definir o mesmo v-model e o Vue irá realizar o databind.

Select

Campos do tipo select podem ser ligados a um v-model, onde o valor selecionado irá atualizar o valor da variável. Caso use a opção `multiple`, vários valores podem ser selecionados.

Para criar opções dinamicamente, basta usar `v-for` no elemento `<option>` do select, conforme o exemplo a seguir:

```
1 <select v-model="selected">
2   <option v-for="option in options" v-bind:value="option.va\
3 lue">
4     {{ option.text }}
5   </option>
6 </select>
```

Atributos para input

Existem três atributos que podem ser usados no elemento input para adicionar algumas funcionalidades extras. São eles:

lazy Atualiza o model após o evento change do campo input, que ocorre geralmente quando o campo perde o foco. Exemplo: `<input v-model.lazy="name">`

number

Formata o campo input para aceitar somente números.

2.20 Conclusão

Abordamos quase todas as funcionalidades do vue e deixamos uma das principais, *Components*, para o próximo capítulo, que necessita de uma atenção em especial.

3. Criando componentes

Uma das principais funcionalidades do Vue é a componentização. Nesta metodologia, começamos a pensar no sistema web como um conjunto de dezenas de componentes que se interagem entre si.

Quando criamos uma aplicação web mais complexa, ou seja, aquela aplicação que vai além de uma simples tela, usamos a componentização para separar cada parte e deixar a aplicação com uma cara mais dinâmica.

Esta separação envolve diversas tecnologias que serão empregadas neste capítulo.

3.1 Vue-cli

O “vue-cli” é um client do node que cria o esqueleto de uma aplicação completa em vue. Ele é fundamental para que possamos criar a aplicação inicial, e compreender como os componentes funcionam no Vue.

Para instalar o `vue-cli`, digite na linha de comando do seu sistema operacional:

```
1 $ npm install -g vue-cli
```



No linux, não esqueça do `sudo`

Após a instalação global do `vue-cli`, digite “vue” na linha comando e certifique-se da seguinte saída:

```
c:\Users\daniel>vue

Usage: vue <command> [options]

Commands:

  init      generate a new project from a template
  list      list available official templates
  help [cmd] display help for [cmd]

Options:

  -h, --help      output usage information
  -V, --version    output the version number
```

3.2 Criando o primeiro projeto com vue-cli

Para criar o primeiro projeto com `vue cli`, digite o seguinte comando:

```
1 vue init browserify-simple my-vue-app
```

O instalador lhe pergunta algumas configurações. Deixe tudo como no padrão até a criação da estrutura do projeto.

Após o término, acesse o diretório criado “my-vue-app” e digite:

```
1 npm install
```

3.3 Executando o projeto

Após executar este comando todas as bibliotecas necessárias para que a aplicação Vue funcione corretamente são instaladas. Vamos dar uma olhada na aplicação, para isso basta executar o seguinte comando:

1 npm run dev

Este comando irá executar duas tarefas distintas. Primeiro, ele irá compilar a aplicação Vue utilizando o **Browserify**, que é um gerenciador de dependências. Depois, ele inicia um pequeno servidor web, geralmente com o endereço `http://localhost:8080`. Verifique a saída do comando para obter a porta corretamente.

Com o endereço correto, acesse-o no navegador e verifique se a saída “Hello Vue!” é exibida.

Deixe o comando `npm run dev` sendo executado, pois quando uma alteração no código é realizada, o npm irá recompilar tudo e atualizar o navegador.

3.4 Conhecendo a estrutura do projeto

Após a criação do projeto, vamos comentar cada arquivo que foi criado.

.babelrc

Contém configurações da compilação do Javascript para es2015

.gitignore

Arquivo de configuração do Git informando quais os diretórios deverão ser ignorados. Este arquivo é útil caso esteja usando o controle de versão git.

index.html

Contém um esqueleto html básico da aplicação. Nele podemos ver a tag `<div id="app"></div>` que é onde toda a aplicação Vue será renderizada. Também temos a inclusão do arquivo `dist/build.js` que é um arquivo javascript compactado e minificado, contendo toda a aplicação. Esse arquivo é gerado constantemente, a cada alteração do projeto.

node_modules

O diretório `node_modules` contém todas as bibliotecas instaladas pelo npm.

O comando `npm install` se encarrega de ler o arquivo `package.json`, baixar tudo o que é necessário e salvar na pasta `node_modules`.

src/main.js

É o arquivo javascript que inicia a aplicação. Ele contém o comando `import` que será interpretado pelo `browserify` e a instância `Vue`, que aprendemos a criar no capítulo anterior.

src/App.vue

Aqui temos a grande novidade do `Vue`, que é a sua forma de componentização baseada em `template`, `script` e `style`. Perceba que a sua estrutura foi criada para se comportar como um componente, e não mais uma instância `Vue`. O foco desse capítulo será a criação desses componentes, então iremos abordá-lo com mais ênfase em breve.

package.json

Contém toda a configuração do projeto, indicando seu nome, autor, scripts que podem ser executados e bibliotecas dependentes.

3.5 Conhecendo o `package.json`

Vamos abrir o arquivo `package.json` e conferir o item “`scripts`”, que deve ser semelhante ao texto a seguir:

```
1 "scripts": {  
2   "watchify": "watchify -vd -p browserify-hmr -e src/main\  
3   .js -o dist/build.js",  
4   "serve": "http-server -o -s -c 1 -a localhost",  
5   "dev": "npm-run-all --parallel watchify serve",  
6   "build": "cross-env NODE_ENV=production browserify -g e\  
7   nvify src/main.js | uglifyjs -c warnings=false -m > dist/bu\  
8   ld.js"  
9 }
```

Veja que temos 4 scripts prontos para serem executados. Quando executamos `npm run dev` estamos executando o seguinte comando: `npm-run-all --parallel watchify serve`. Ou seja, estamos executando o script **watchify** e em paralelo, o script **serve**.

O script **watchify** nos apresenta com o uso do gerenciador de dependências **browserify**, que não é o foco desta obra, mas é válido pelo menos sabermos que ele existe. O que ele faz? Ele vai pegar todas as dependências de bibliotecas que começam no `src/main.js`, juntar tudo em um arquivo único em `dist/build.js`.

3.6 Componentes e arquivos .vue

Uma das melhores funcionalidades do Vue é usar componentes para que a sua aplicação seja criada. Começamos a ver componentes a partir deste projeto, onde todo componente é um arquivo com a extensão `.vue` e possui basicamente três blocos:

template

É o template HTML do componente, que possui as mesmas funcionalidades que foram abordadas no capítulo anterior.

script

Contém o código javascript que adiciona as funcionalidades ao template e ao componente.

style

Adiciona os estilos css ao componente

O arquivo `src\App.vue` é exibido a seguir:


```
1 <template>
2   <div id="app">
3     <h1>{{ msg }}</h1>
4   </div>
5 </template>
6
7 <script>
8   export default {
9     data () {
10       return {
11         msg: 'Hello Vue!'
12       }
13     }
14   }
15 </script>
16
17 <style>
18   body {
19     font-family: Helvetica, sans-serif;
20   }
21 </style>
```

Inicialmente usamos o `<template>` para criar o Html do template App, onde possuímos uma div e um elemento h1. No elemento `<script>` temos o código javascript do template, inicialmente apenas com a propriedade data. A forma como declaramos o data é um pouco diferente da abordada até o momento, devido ao forma como o browserify trata o componente `.vue`.

Ao invés de termos:

```
1 data: {
2   msg: "Hello World"
3 }
```

Temos:

```
1 export default {  
2   data(){  
3     return {  
4       msg: 'Hello Vue!'  
5     }  
6   }  
7 }
```

Na parte <style>, podemos adicionar estilos css que correspondem ao template.

Caso o seu editor de textos/IDE não esteja formatando a sintaxe dos arquivos .vue, instale o plugin apropriado a sua IDE. Geralmente o plugin tem o nome *Vue* ou *Vue Syntax*

3.7 Criando um novo componente

Para criar um novo componente, basta criar um novo arquivo .vue com a definição de template, script e style. Crie o arquivo MyMenu.vue e adicione o seguinte código:

src/MyMenu.vue

```
1 <template>  
2   <div>My Menu</div>  
3 </template>  
4 <script>  
5   export default{  
6  
7   }  
8 </script>
```

Neste momento ainda temos um componente simples, que possui unicamente o texto “My Menu”. Para incluir esse componente na aplicação, retorne ao App.vue e adicione-o no template, veja:

src/App.vue

```
1 <template>
2   <div id="app">
3     <my-menu></my-menu>
4     <h1>{{ msg }}</h1>
5   </div>
6 </template>
```

Perceba que o componente `MyMenu.vue` tem a tag `<my-menu>`. Esta é uma convenção do Vue chamada de *kebab-case*. Após adicionar o componente, e com o `npm run dev` em execução, recarregue a aplicação e perceba que o texto “Menu” ainda não apareceu na página.

Isso acontece porque é necessário executar dois passos extras no carregamento de um componente. Primeiro, é preciso dizer ao Browserify para carregar o componente e adicioná-lo ao arquivo `build.js`. Isso é feito através do `import`, assim como no arquivo `main.js` importou o componente `App.vue`, o componente `App.vue` deve importar o componente `Menu.vue`, veja:

src/App.vue

```
1 <template>
2   <div id="app">
3     <my-menu></my-menu>
4     <h1>{{ msg }}</h1>
5   </div>
6 </template>
7
8 <script>
9   import MyMenu from './MyMenu.vue'
10
11   export default {
12     data () {
13       return {
```

```
14      msg: 'Hello Vue!'
15    }
16  }
17 }
18 </script>
19
20 <style>
21   body {
22     font-family: Helvetica, sans-serif;
23   }
24 </style>
```

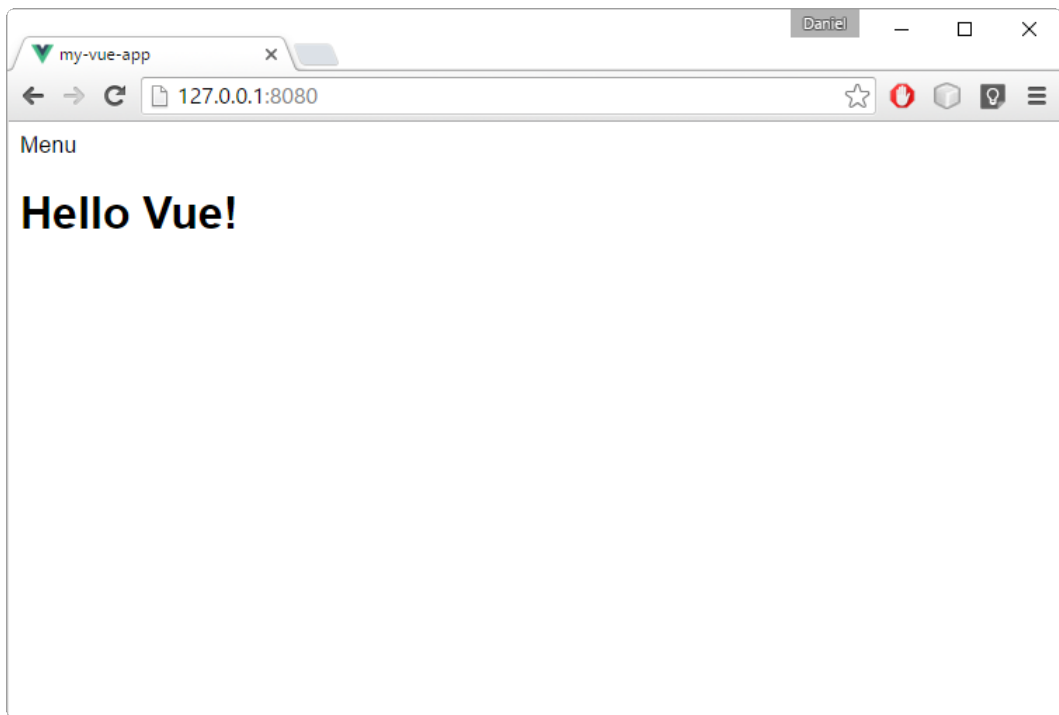
Ao incluirmos `import MyMenu from './MyMenu.vue'` estaremos referenciando o componente `MyMenu` e permitindo que o `Browserify` adicione-o no arquivo `build.js`. Além desta alteração, é necessário também configurar o componente `App` dizendo que ele usa o componente `MyMenu`. No `Vue`, sempre que usarmos um componente precisamos fornecer esta informação através da propriedade `components`. Veja o código a seguir:

`src/App.vue`

```
1 <template>
2   <div id="app">
3     <my-menu></my-menu>
4     <h1>{{ msg }}</h1>
5   </div>
6 </template>
7
8 <script>
9   import MyMenu from './MyMenu.vue'
10
11   export default {
12     components:{
13       MyMenu
```

```
14     },
15     data () {
16         return {
17             msg: 'Hello Vue!'
18         }
19     }
20 }
21 </script>
22
23 <style>
24     body {
25         font-family: Helvetica, sans-serif;
26     }
27 </style>
```

Agora que adicionamos o *import* e referenciamos o componente, o resultado do `<my-menu>` pode ser observado na página, que a princípio é semelhante a figura a seguir.



3.8 Adicionando propriedades

Vamos supor que o componente `<my-menu>` possui uma propriedade chamada `title`. Para configurar propriedades no componente, usamos a propriedade `props` no script do componente, conforme o exemplo a seguir:

src/MyMenu.vue

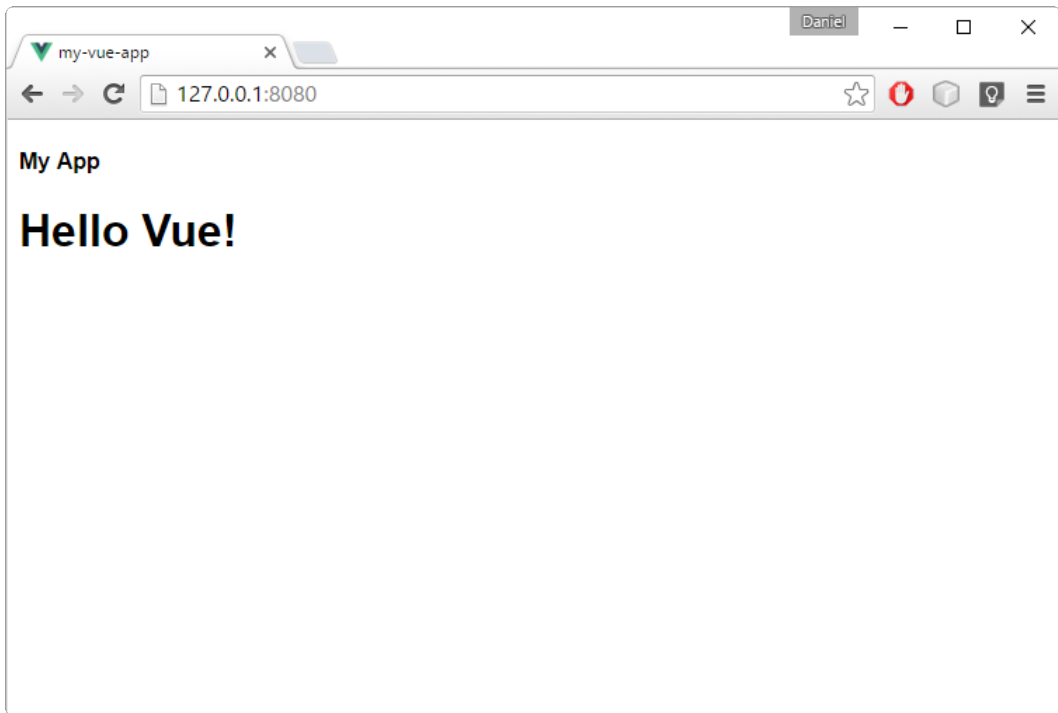
```
1 <template>
2   <h4>{{title}}</h4>
3 </template>
4 <script>
5   export default{
6     props: ['title']
7   }
8 </script>
```

Desta forma, a propriedade `title` pode ser repassada no componente `App`, da seguinte forma:

src/App.vue

```
1 <template>
2   <div id="app">
3     <my-menu title="My App"></my-menu>
4     <h1>{{ msg }}</h1>
5   </div>
6 </template>
7 <script>
8
9 </script>
```

O resultado é semelhante a figura a seguir:



camelCase vs. kebab-case

A forma como as propriedades são organizadas observam o modo kebab-case. Isso significa que uma propriedade chamada `myMessage` deve ser usada no template da seguinte forma: `my-message`.

Validações e valor padrão

Pode-se adicionar validações nas propriedades de um componente, conforme os exemplos a seguir:


```
1  props: {
2    // tipo número
3    propA: Number,
4
5    // String ou número (1.0.21+)
6    propM: [String, Number],
7
8    // Uma propriedade obrigatória
9    propB: {
10   type: String,
11   required: true
12 },
13
14 // um número com valor padrão
15 propC: {
16   type: Number,
17   default: 100
18 },
19
20
21 // Uma validação customizada
22 propF: {
23   validator: function (value) {
24     return value > 10
25   }
26 },
27
28 //Retorna o JSON da propriedade
29 propH: {
30   coerce: function (val) {
31     return JSON.parse(val) // cast the value to Object
32   }
33 }
```

Os tipos de propriedades podem ser: String, Number, Boolean, Function, Object e Array.

3.9 Slots e composição de componentes

Componentes do Vue podem ser compostos de outros componentes, textos, códigos html etc. Usamos a tag `<slot></slot>` para criar a composição de componentes. Suponha que queremos um componente menu que tenha a seguinte configuração:

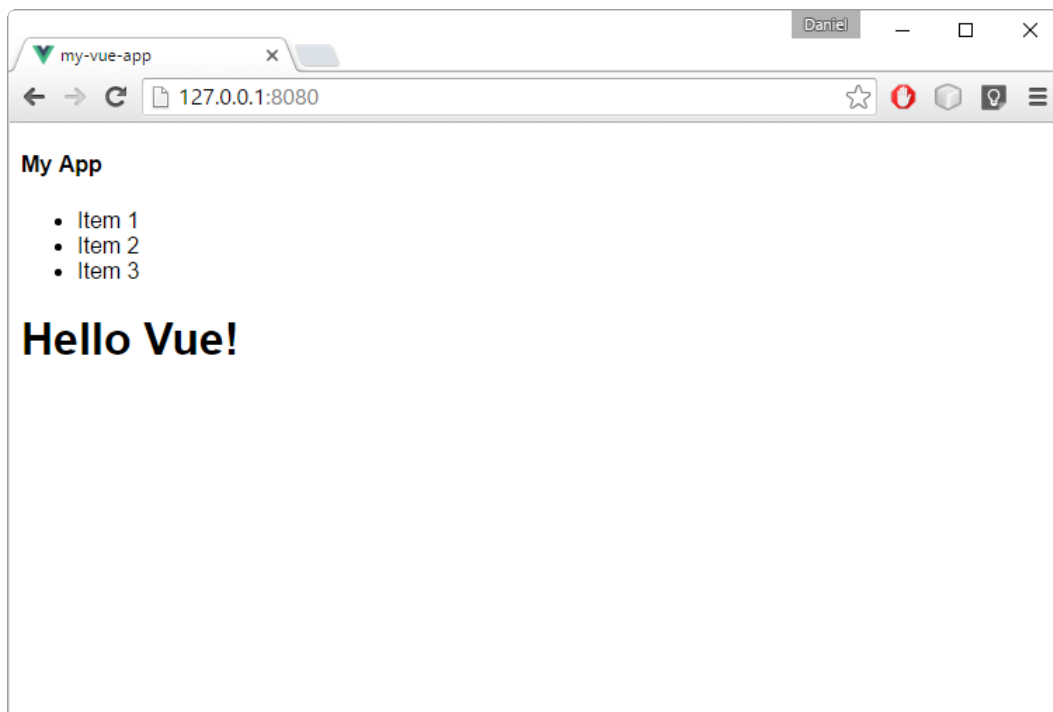
```
1 <my-menu title="My App">
2   <ul>
3     <li>Item 1</li>
4     <li>Item 2</li>
5     <li>Item 3</li>
6   </ul>
7 </my-menu>
```

O conteúdo interno ao menu é a sua composição, e pode ser configurado da seguinte forma:

```
1 <template>
2   <div>
3     <h4>{{title}}</h4>
4     <div>
5       <slot></slot>
6     </div>
7   </div>
8 </template>
9 <script>
10   export default{
11     props: ['title']
12   }
13 </script>
```

Veja que, quando usamos a tag “<slot>”, o conteúdo do novo componente MyMenu será inserido dentro desta tag. Neste caso, o conteúdo composto pelo “...” será inserido dentro do slot, realizando assim a composição do componente.

Com isso, pode-se criar componentes que contém outros componentes com facilidade. O resultado do uso de slots no menu é semelhante a figura a seguir:



3.10 Eventos e comunicação entre componentes

Já vimos duas formas de um componente se comunicar com outro. Temos inicialmente a criação de propriedades através do atributo props e a criação de slots que permitem que componentes possam ser compostos por outros componentes.

Agora veremos como um componente pode enviar mensagens para outro componente, de forma independente entre eles, através de eventos. Cada componente Vue funciona de forma independente, sendo que eles podem trabalhar com 4 formas de eventos personalizados, que são:

- Escutar eventos através do método `$on()`
- Disparar eventos através do método `$emit()`

Assim como na DOM, os eventos do Vue param assim que encontram o seu callback, exceto se ele retornar “true”.

Vamos supor que, quando clicamos em um botão, devemos notificar a App que um item do menu foi clicado. Primeiro, precisamos configurar na App para que ele escute os eventos do menu, veja:

src/App.vue

```
1 <template>
2   <div id="app">
3     <my-menu title="My App" v-on:menu-click="onMenuClick">
4       <ul>
5         <li>Item 1</li>
6         <li>Item 2</li>
7         <li>Item 3</li>
8       </ul>
9     </my-menu>
10    <h1>{{ msg }}</h1>
11  </div>
12 </template>
13
14 <script>
15   import MyMenu from './MyMenu.vue'
16
17   export default {
18     components:{
19       MyMenu
20     },
21     data () {
22       return {
```

```
23     msg: 'Hello Vue!'
24   }
25 },
26   methods:{
27     onMenuClick:function(e){
28       alert("menu click");
29     }
30   }
31 }
32 </script>
33
34 <style>
35   body {
36     font-family: Helvetica, sans-serif;
37   }
38 </style>
```

Na tag `<my-menu>` temos a configuração do evento `menu-click`:

```
1 v-on:click="onMenuClick"
```

Que irá chamar o método `onMenuClick`, definido pelo método:

```
1 methods:{
2   onMenuClick:function(){
3     alert("menu click");
4   }
5 }
```

Com o evento configurado, podemos dispará-lo dentro do componente `MyMenu`:

src/Menu.vue

```
1 <template>
2   <div @click="onMenuClick">
3     <h4>{{title}}</h4>
4     <div>
5       <slot></slot>
6     </div>
7     <hr/>
8 </template>
9 <script>
10   export default{
11     props: ['title'],
12     methods:{
13       onMenuClick : function(){
14         this.$emit('menu-click');
15       }
16     }
17   }
18 </script>
```

Criamos um botão que chama o método `onButtonClick`, que por sua vez usa o `$emit` para disparar o evento `button-click`.

Repassando parâmetros

A passagem de parâmetros pode ser realizada adicionando um segundo parâmetro no disparo do evento, como no exemplo a seguir:

src/Menu.vue

```
1 methods:{  
2   onClick : function(){  
3     this.$emit('button-click',"emit event from menu");  
4   }  
5 }
```

e no componente App, podemos capturar o parâmetro da seguinte forma:

src/App.vue

```
1 methods:{  
2   onClick:function(message){  
3     alert(message);  
4   }  
5 }
```

3.11 Reorganizando o projeto

Agora que vimos alguns conceitos da criação de componentes, podemos reorganizar o projeto e criar algumas funcionalidades básicas, de forma a deixar a aplicação com um design mais elegante.

Primeiro, vamos dividir a aplicação em Header, Content e Footer. Queremos que, o componente `App.vue` tenha a seguinte estrutura inicial:

src/App.vue

```
1 <template>
2   <div id="app">
3     <app-header></app-header>
4     <app-content></app-content>
5     <app-footer></app-footer>
6   </div>
7 </template>
8
9 <script>
10   import AppHeader from './layout/AppHeader.vue'
11   import AppContent from './layout/AppContent.vue'
12   import AppFooter from './layout/AppFooter.vue'
13
14   export default {
15     components: {
16       AppHeader, AppContent, AppFooter
17     }
18   }
19 </script>
```

Perceba que criamos três componentes: AppHeader, AppContent, AppFooter. Estes componentes foram criados no diretório layout, para que possamos começar a separar melhor cada funcionalidade de cada componente. Então, componentes que façam parte do layout da aplicação ficam no diretório layout. Componentes que compõem formulários podem ficar em um diretório chamado form, enquanto que componentes ligados a relatórios podem estar em report. Também pode-se separar os componentes de acordo com as regras de negócio da aplicação. Por exemplo, os componentes ligados a listagem, formulário e relatório de uma tabela “Products” pode estar localizada no diretório src/app/product.

Os componentes AppHeader, AppContent, AppFooter a princípio não tem nenhuma informação, possuindo apenas um pequeno texto, veja:

src/layout/AppHeader.vue

```
1 <template>
2   <h4>Header</h4>
3 </template>
4 <script>
5   export default{
6
7   }
8 </script>
```

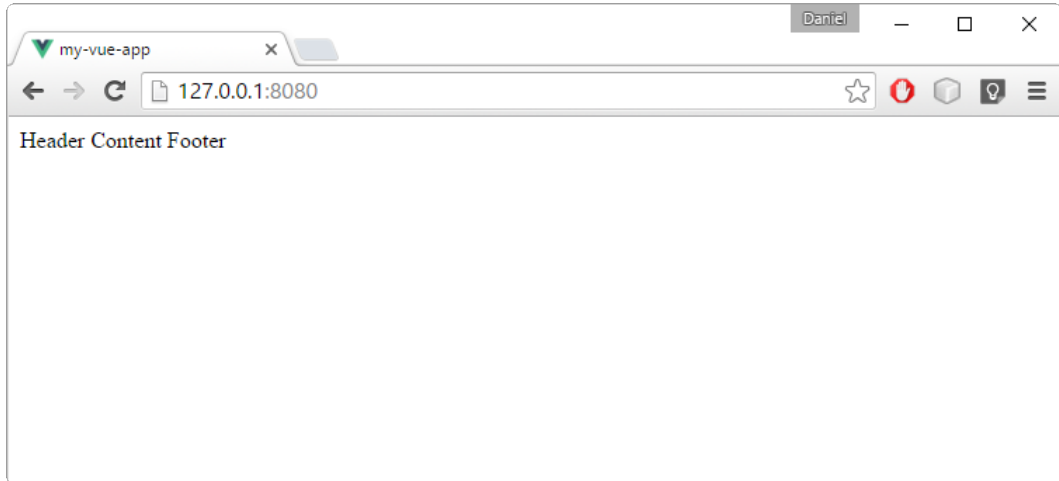
src/layout/AppContent.vue

```
1 <template>
2   <h4>Content</h4>
3 </template>
4 <script>
5   export default{
6
7   }
8 </script>
```

src/layout/AppFooter.vue

```
1 <template>
2   <h5>Footer</h5>
3 </template>
4 <script>
5   export default{
6
7   }
8 </script>
```

Após refatorar o App.vue e criar os componentes AppHeader,AppContent,AppFooter, temos uma simples página web semelhante a figura a seguir:



3.12 Adicionando algum estilo

Agora vamos adicionar algum estilo a nossa aplicação. Existem dezenas de frameworks CSS disponíveis no mercado, gratuitos e pagos. O mais conhecidos são bootstrap, semantic-ui, Foundation, UI-Kit, Pure, Materialize. Não existe melhor ou pior, você pode usar o framework que mais gosta. Nesta obra usaremos o Materialize CSS que segue as convenções do Material Design criado pelo Google.

Para adicionar o Materialize no projeto, instale-o através do seguinte comando:

```
1 npm i -S materialize-css
```

Após a instalação, precisamos referenciá-lo no arquivo index.html, da seguinte forma:

index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <meta name="viewport" content="width=device-width, initia\
6  l-scale=1.0"/>
7    <title>my-vue-app</title>
8
9    <!--Materialize Styles-->
10   <link href="http://fonts.googleapis.com/icon?family=Mater\
11  ial+Icons" rel="stylesheet">
12   <link type="text/css" rel="stylesheet" href="node_modules\
13  /materialize-css/dist/css/materialize.min.css" media="scre\
14  en,projection"/>
15
16 </head>
17 <body>
18   <app></app>
19   <!--Materialize Javascript-->
20   <script src="node_modules/jquery/dist/jquery.min.js"></sc\
21  ript>
22   <script src="node_modules/materialize-css/dist/js/materia\
23  lize.min.js"></script>
24   <script src="dist/build.js"></script>
25 </body>
26 </html>
```



Atenção quanto a quebra de linha no código html acima, representado pelo \

Para instalar o materialize, é preciso adicionar dois estilos CSS. O primeiro são os ícones do google e o segundo o CSS do materialize que está no `node_modules`. No final do `<body>`, temos que adicionar os arquivos javascript do materialize, que incluindo o jQuery. Eles devem ser referenciados **após** o app e **antes** do `build.js`.

Assim que o materialize é instalado, percebe-se que a fonte dos textos que compreendem o `AppHeader`, `AppContent`, `AppFooter` mudam, de acordo com a imagem a seguir:



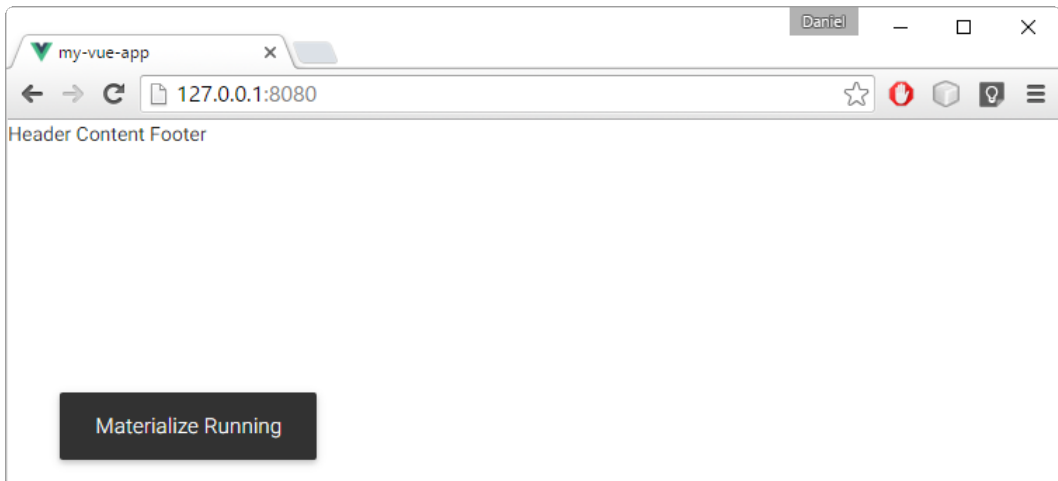
Para testar se está tudo correto, podemos usar a função **toast** do Materialize que exiba uma notificação na tela. Vamos incluir essa notificação no evento *created* do `App.vue`, veja:

src/App.vue

```
1 <template>
2   <div id="app">
3     <app-header></app-header>
4     <app-content></app-content>
5     <app-footer></app-footer>
6   </div>
7 </template>
8
```

```
9 <script>
10   import AppHeader from './layout/AppHeader.vue'
11   import AppContent from './layout/AppContent.vue'
12   import AppFooter from './layout/AppFooter.vue'
13
14   export default {
15     components:{
16       AppHeader,AppContent,AppFooter
17     },
18     created:function(){
19       Materialize.toast('Materialize Running', 1000)
20     }
21   }
22 </script>
```

Após recarregar a página, surge uma mensagem de notificação conforme a imagem a seguir:



3.13 Alterando o cabeçalho

Vamos alterar o AppHeader para exibir um cabeçalho no estilo materialize. Pelos exemplos do site oficial, podemos copiar o seguinte código:

```
1 <nav>
2   <div class="nav-wrapper">
3     <a href="#" class="brand-logo">Logo</a>
4     <ul id="nav-mobile" class="right hide-on-med-and-down">
5       <li><a href="sass.html">Sass</a></li>
6       <li><a href="badges.html">Components</a></li>
7       <li><a href="collapsible.html">JavaScript</a></li>
8     </ul>
9   </div>
10 </nav>
```

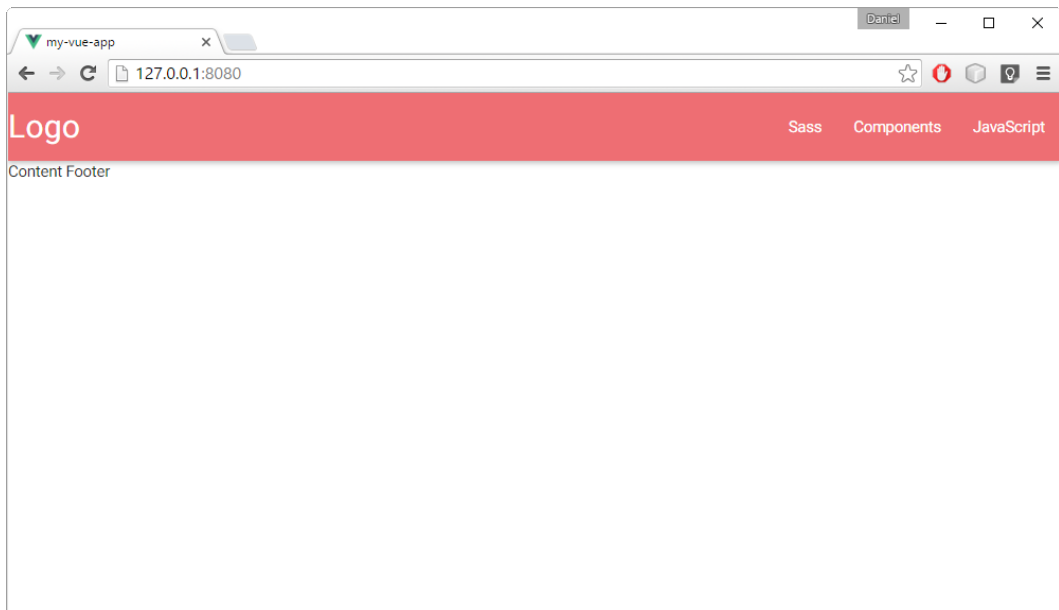
E adicionar no template do AppHeader:

src/layout/AppHeader.vue

```
1 <template>
2   <nav>
3     <div class="nav-wrapper">
4       <a href="#" class="brand-logo">Logo</a>
5       <ul id="nav-mobile" class="right hide-on-med-and-down\
6   ">
7         <li><a href="sass.html">Sass</a></li>
8         <li><a href="badges.html">Components</a></li>
9         <li><a href="collapsible.html">JavaScript</a></li>
10      </ul>
11    </div>
12  </nav>
13 </template>
```

```
14 <script>
15   export default{
16
17   }
18 </script>
```

O que resulta em:



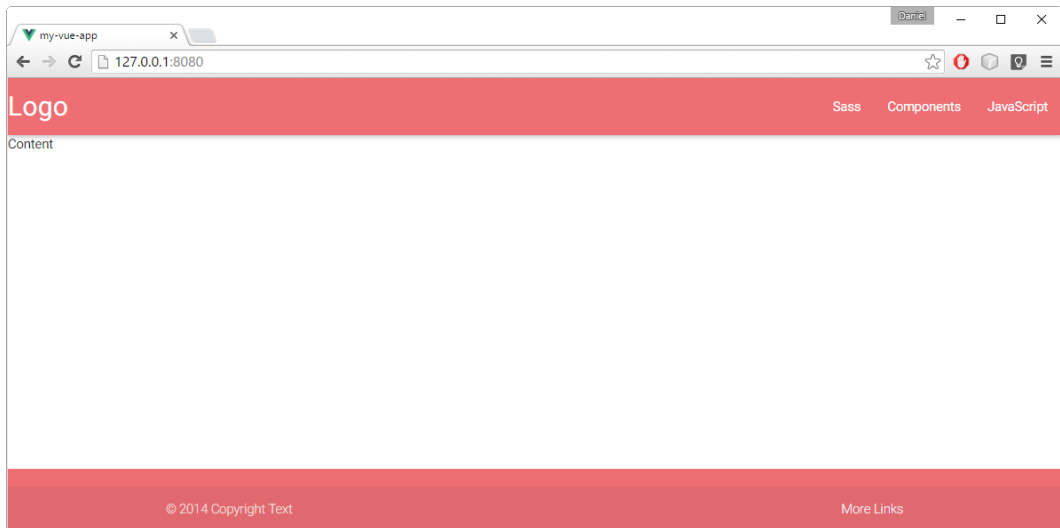
3.14 Alterando o rodapé

O mesmo pode ser aplicado ao rodapé da pagina, copiando um estilo direto do materialize e aplicando no AppFooter:

src/layout/AppFooter.vue

```
1 <template>
2   <footer class="page-footer">
3     <div class="footer-copyright">
4       <div class="container">
5         © 2014 Copyright Text
6         <a class="grey-text text-lighten-4 right" href="#" \
7 >More Links</a>
8       </div>
9     </div>
10  </footer>
11 </template>
12 <script>
13   export default{
14
15   }
16 </script>
17 <style>
18   footer {
19     position: fixed;
20     bottom: 0;
21     width: 100%;
22   }
23 </style>
```

Aqui usamos o `style` para fixar o rodapé na parte inferior da página, que produz o seguinte resultado:



3.15 Conteúdo da aplicação

A última parte que precisamos preencher é relacionada ao conteúdo da aplicação, onde a maioria das telas serão criadas. Isso significa que precisamos gerenciar as telas que são carregadas na parte central da aplicação, sendo que quando clicamos em um link ou em algum botão, o conteúdo pode ser alterado.

Para realizar este gerenciamento temos uma biblioteca chamada `vue-router`, na qual iremos abordar no próximo capítulo.

4. Vue Router

Dando prosseguimento ao projeto ‘my-vue-app’ que criamos no capítulo anterior, precisamos fornecer no AppComponent uma forma de gerenciar várias telas que compõem o sistema.

4.1 Instalação

Para isso, usamos a biblioteca chamada vue-router. Para instalá-la, usamos novamente o npm:

```
1 npm i -S vue-router
```

4.2 Configuração

Com o vue-router instalado, precisamos configurá-lo. Para isso, precisamos alterar o arquivo main.js que irá conter uma inicialização diferente, veja:

src/main.js

```
1 import Vue from 'vue'
2 import App from './App.vue'
3 import VueRouter from 'vue-router'
4
5 var router = new VueRouter()
6
7 Vue.use(VueRouter)
8
9 new Vue({
```

```
10   el: '#app',  
11   router: router,  
12   render: h => h(App)  
13 })
```

Ao adotarmos o vue-router, passamos a importá-lo através do `import VueRouter from 'vue-router'` e criamos a variável `router` que a princípio não possui nenhum parâmetro.

Todo plugin ou biblioteca do Vue é importado pelo comando `Vue.use`

Ainda existem dois detalhes para que o código funcione. Primeiro, precisamos criar um componente que será o componente a ser carregado quando a aplicação navegar até a raiz da aplicação “/”. Segundo, precisamos configurar em qual layout os componentes gerenciados pelo router serão carregados.

Para simplificar, criamos o componente “HelloWorldRouter”, no diretório `src/components`, apenas com um simples texto:

```
1 <template>  
2   <div>Hello World Router</div>  
3 </template>  
4 <script>  
5   export default{  
6  
7   }  
8 </script>
```

4.3 Configurando o router

Então podemos voltar ao `main.js` e relacionar esse componente ao router, veja:

main.js

```
1  import Vue from 'vue'
2  import App from './App.vue'
3  import VueRouter from 'vue-router'
4
5  import HelloWorldRouter from './components/HelloWorldRouter\
6  .vue'
7
8  var router = new VueRouter({
9    routes: [
10     { path: '/', component: HelloWorldRouter }
11   ]
12 })
13
14 Vue.use(VueRouter)
15
16 new Vue({
17   el: '#app',
18   router: router,
19   render: h => h(App)
20 })
```

As rotas são configuradas através do parâmetro *routes*, que é um array contendo objetos do tipo: {path: '...', component: '...'}.

4.4 Configurando o router-view

Após a configuração do primeiro componente gerenciado pelo Router, nós precisamos configurar **onde** o componente será carregado.

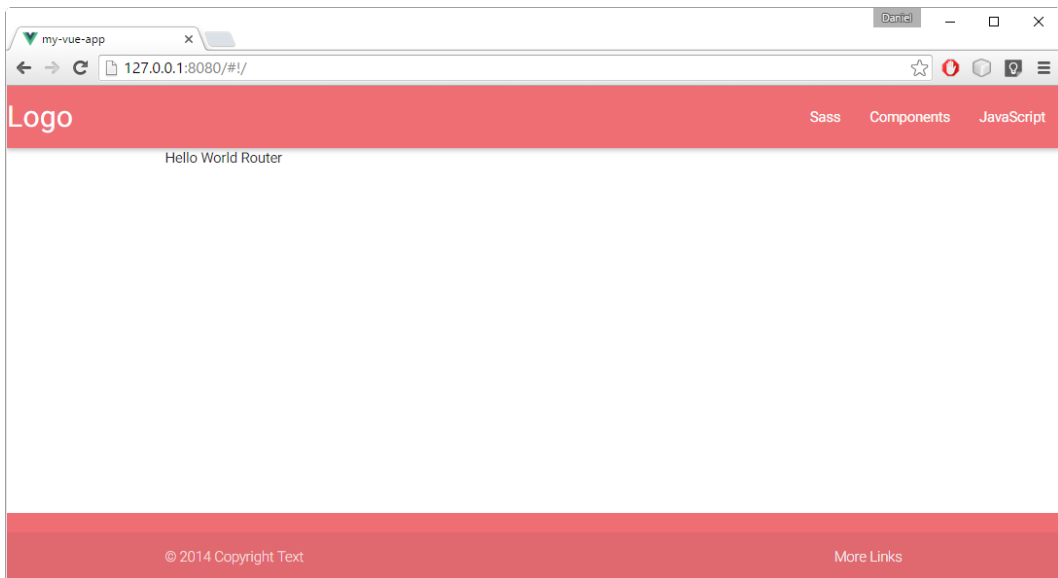
Voltando ao AppContent, vamos refatorá-lo para permitir que os componentes sejam carregados nele:

src/layout/AppContent.vue

```
1 <template>
2   <div class="container">
3     <router-view></router-view>
4   </div>
5 </template>
6 <script>
7   export default{
8
9   }
10 </script>
```

Agora o AppContent possui uma div com a classe container, e nesta div temos o elemento `<router-view></router-view>`. Será neste elemento que os componentes do Vue Router serão carregados.

Ao verificarmos a aplicação, temos a seguinte resposta:



4.5 Criando novos componentes

Para exemplificar como o router funciona, vamos criar mais dois componentes, veja:

src/components/Card.vue

```
1  <template>
2    <div class="row">
3      <div class="col s12 m6">
4        <div class="card blue-grey darken-1">
5          <div class="card-content white-text">
6            <span class="card-title">Card Title</span>
7            <p>I am a very simple card. I am good at containing\
8  small bits of information.
9            I am convenient because I require little markup t\
10 o use effectively.</p>
11          </div>
12          <div class="card-action">
13            <a href="#">This is a link</a>
14            <a href="#">This is a link</a>
15          </div>
16        </div>
17      </div>
18    </div>
19  </template>
20  <script>
21    export default{
22
23    }
24  </script>
```

Este componente foi retirado de um exemplo do Materialize, e usa CSS para desenhar um objeto que se assemelha a um card.

Outro componente irá possuir alguns botões, veja:

src/components/Buttons.vue

```
1 <template>
2   <a class="waves-effect waves-light btn">button</a>
3   <a class="waves-effect waves-light btn"><i class="material-
4 l-icons left">cloud</i>button</a>
5   <a class="waves-effect waves-light btn"><i class="material-
6 l-icons right">cloud</i>button</a>
7 </template>
8 <script>
9   export default{
10   }
11 </script>
```

Novamente estamos apenas copiando alguns botões do Materialize para exibir como exemplo no Vue Router.

Com os dois componentes criados, podemos configurar o mapeamento do router no arquivo `main.js`, veja:

src/main.js

```
1 import Vue from 'vue'
2 import App from './App.vue'
3 import VueRouter from 'vue-router'
4
5 import HelloWorldRouter from './components/HelloWorldRouter\
6 .vue'
7 import Card from './components/Card.vue'
8 import Buttons from './components/Buttons.vue'
9
10 Vue.use(VueRouter)
11 const router = new VueRouter({
12   routes: [
13     { path: '/', component: HelloWorldRouter },
```

```
14     { path: '/card', component: Card },
15     { path: '/buttons', component: Buttons }
16   ]
17 })
```

Configuramos o router para, quando o endereço “/card” for chamado, o componente Card seja carregado. O mesmo para /buttons. Se você digitar esse endereço na barra de endereços do navegador, como por exemplo `http://127.0.0.1:8080/#!/buttons` poderá ver os botões carregados, mas vamos criar um menu com esses itens.

4.6 Criando um menu

Agora que temos três componentes gerenciados pelo router, podemos criar um menu com o link para estes componentes. Voltando ao LayoutHeader, vamos alterar aquele menu horizontal com o seguinte html:

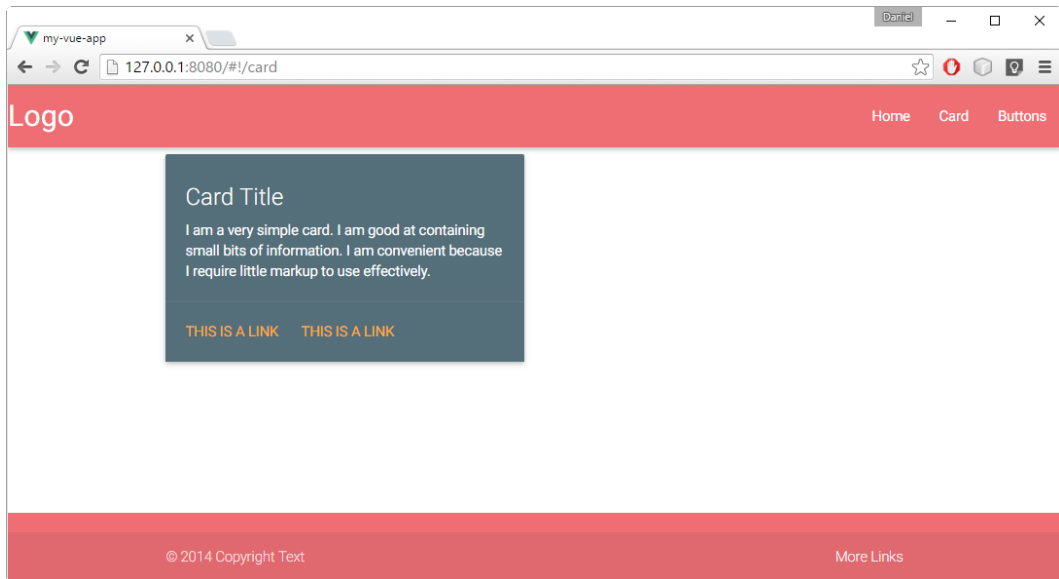
src/layout/AppHeader.vue

```
1 <template>
2   <nav>
3     <div class="nav-wrapper">
4       <a href="#" class="brand-logo">Logo</a>
5       <ul id="nav-mobile" class="right hide-on-med-and-down\
6     ">
7         <li><router-link to="/">Home</router-link></li>
8         <li><router-link to="/cards">Cards</router-link></li>
9     i>
10        <li><router-link to="/buttons">Buttons</router-link\
11    ></li>
12    </ul>
13  </div>
14 </nav>
15 </template>
```



```
16 <script>
17   export default{
18
19   }
20 </script>
```

Após atualizar a página, podemos clicar nos itens de menu no topo da aplicação e verificar que os conteúdos estão sendo carregados de acordo com a url. Desta forma podemos usar o router para navegar entre os diversos componentes da aplicação.



5. Vue Resource

O plugin Vue Resource irá lhe ajudar a prover acesso ao servidor web através de requisições Ajax usando XMLHttpRequest ou JSONP.

Para adicionar o Vue Resource no seu projeto, execute o seguinte comando:

```
1 npm i -S vue-resource
```

Após a instalação do Vue Resource pelo npm, podemos configurá-lo no arquivo main.js:

```
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 import VueRouter from 'vue-router'
5 import VueResource from 'vue-resource'
6
7 import HelloWorldRouter from './components/HelloWorldRouter\
8 .vue'
9 import Card from './components/Card.vue'
10 import Buttons from './components/Buttons.vue'
11
12 Vue.use(VueResource)
13 Vue.use(VueRouter)
14 ...
15 ...
```

Após iniciar o Vue Resource, pode-se configurar o diretório raiz que o servidor usar(caso necessário) e a chave de autenticação, conforme o exemplo a seguir:

```
1 Vue.http.options.root = '/root';
2 Vue.http.headers.common['Authorization'] = 'Basic YXBpOnBhc\
3 3N3b3Jk';
```

No projeto `my-vue-app` estas configurações não serão necessárias.

5.1 Testando o acesso Ajax

Para que possamos simular uma requisição Ajax, crie o arquivo `users.json` na raiz do projeto com o seguinte código:

```
1 [
2 {
3   "name": "User1",
4   "email": "user1@gmail.com",
5   "country": "USA"
6 },
7 {
8   "name": "User2",
9   "email": "user2@gmail.com",
10  "country": "Mexico"
11 },
12 {
13   "name": "User3",
14   "email": "user3@gmail.com",
15   "country": "France"
16 },
17 {
18   "name": "User4",
19   "email": "user4@gmail.com",
20   "country": "Brazil"
21 }
22 ]
```

Com o arquivo criado na raiz do projeto, pode-se realizar uma chamada ajax na url “/users.json”. Vamos fazer esta chamada no componente Buttons que criamos no capítulo anterior.

Abra o arquivo `src/components/Buttons.vue` e adicione e altere-o para:

`src/components/Buttons.vue`

```
1  <template>
2  <div>
3    <a @click="callUsers"
4      class="waves-effect waves-light btn">Call Users</a>
5
6    <a @click="countUsers"
7      class="waves-effect waves-light btn">
8      <i class="material-icons left">cloud</i>Count Users
9    </a>
10
11   <a class="waves-effect waves-light btn">
12     <i class="material-icons right">cloud</i>button
13   </a>
14
15   <hr/>
16   <pre>
17     {{ users | json }}
18   </pre>
19
20 </div>
21 </template>
22 <script>
23   export default{
24     data() {
25       return{
26         users: null
27       }
28     }
29   }
```

```
28   },
29   methods: {
30     callUsers: function(){
31       this.$http({url: '/users.json', method: 'GET'})
32       .then(function (response) {
33         this.users = response.data
34       }, function (response) {
35         Materialize.toast('Erro!', 1000)
36       });
37     },
38     countUsers: function(){
39       Materialize.toast(this.users.length, 1000)
40     }
41   }
42 }
43 </script>
```

No primeiro botão do componente, associamos o método `callUsers` que irá usar o Vue Resource para realizar uma chamada Ajax:

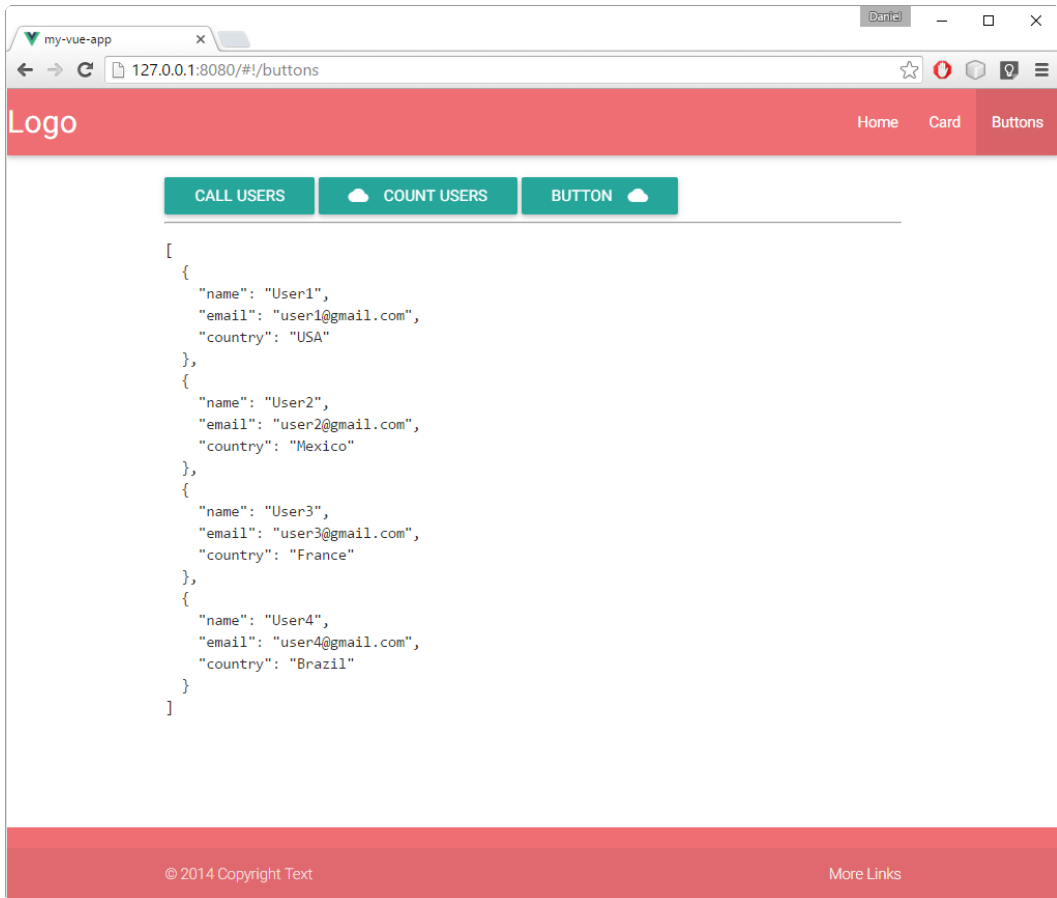
```
1  this.$http({url: '/users.json', method: 'GET'})
2  .then(function (response) {
3    this.users = response.data
4  }, function (response) {
5    Materialize.toast('Error: ' + response.statusText, 3000)
6  });
```

Esta chamada possui a Url de destino e o método GET. Na resposta `.then` existem dois parâmetros, sendo o primeiro deles executado se a requisição ajax for realizada com sucesso, e o segundo se houver algum erro. Quando a requisição é executada com sucesso, associamos a variável `users`, que foi criada no data do componente Vue ao `response.data`, que contém um array de usuários representado pelo json criado em `users.json`.

No template, também criamos a seguinte saída:

1 `{{ users | json }}`

Isso irá imprimir os dados da variável `this.users`, que a princípio é `null`, e após clicar no botão para realizar a chamada Ajax, passa a se tornar um array de objetos, semelhante a imagem a seguir:



O segundo botão irá imprimir na tela através do `Materialize.toast` a quantidade de registros que existe na variável `this.users`, neste caso 4.

5.2 Métodos e opções de envio

Vimos no exemplo anterior como realizar uma chamada GET pelo Vue Response. Os métodos disponíveis para realizar chamadas Ajax ao servidor são:

- `get(url, [data], [options])`
- `post(url, [data], [options])`
- `put(url, [data], [options])`
- `patch(url, [data], [options])`
- `delete(url, [data], [options])`
- `jsonp(url, [data], [options])`

As opções que podem ser repassadas ao servidor são:

Parâmetro	Tipo	Descrição
<code>url</code>	<code>string</code>	URL na qual a requisição será realizada

`| method | string | Método HTTP (GET, POST, ...)`

`| data | Object, string | Dados que podem ser enviados ao servidor` `| params | Object | Um objeto com parâmetros que podem ser enviados em uma requisição GET` `| headers | Object | Cabeçalho HTTP da requisição` `| xhr | Object | Um objeto com parâmetros XHR1` `| upload | Object | Um objeto que contém parâmetros XHR.upload2` `| jsonp | string | Função callback para uma requisição JSONP` `| timeout | number | Timeout da requisição (0 significa sem timeout)` `| beforeSend | function | Função de callback que pode alterar o cabeçalho HTTP antes do envio da requisição` `| emulateHTTP | boolean | Envia as requisições PUT, PATCH e DELETE como requisições POST e adiciona o cabeçalho HTTP X-HTTP-Method-Override` `| emulateJSON | boolean | Envia os dados da requisição (data) com o tipo application/x-www-form-urlencoded`

¹<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>

²<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/upload>

5.3 Trabalhando com resources

Pode-se criar um objeto do tipo `this.$resource` que irá fornecer uma forma diferente de acesso ao servidor, geralmente baseado em uma API pré especificada. Para testarmos o resource, vamos criar mais botões e analisar as chamadas http que o Vue realiza no servidor. Nesse primeiro momento as chamadas HTTP resultarão em erro, mas precisamos apenas observar como o Vue trabalha com resources.

Primeiro, criamos um resource no componente `Buttons.vue`:

```
1 export default{
2   data() {
3     return{
4       users: null,
5       resourceUser : this.$resource('user{/id}')
```

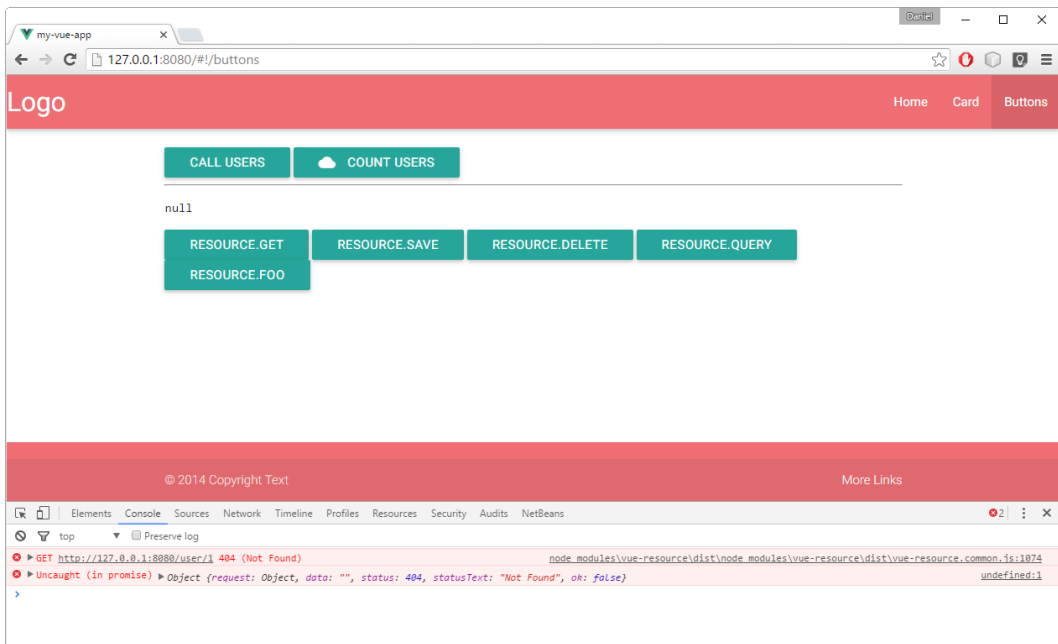
Então, criamos um botão que irá chamar cada tipo de resource. O primeiro deles é o `resource.get`, veja:

```
1 <template>
2   ...
3   <a class="btn" @click="resourceGet">resource.get</a>
4   ...
5 </template>
6 <script>
7   methods:{
8     ...
9     resourceGet:function(){
10       this.resourceUser.get({id:1}).then(function (response\
11 ) {
12       console.log(response)
```



```
13     });  
14   }  
15 }  
16 ...  
17 </script>
```

Como configuramos o resource com o caminho `user{/id}`, o `resource.get` irá realizar a seguinte chamada http:



Outras chamadas de resource podem ser:

- save (POST)
- query (GET)
- update (PUT)
- remove (DELETE)
- delete (DELETE)

Também é possível criar resources customizados, como no exemplo a seguir onde criamos a chamada `/foo`:

```
1 resourceUser : this.$resource('user{/id}', null, {'foo': {meth\
2 od: 'GET', url: "/user/foo"}}}
```

Neste exemplo, pode-se realizar a chamada:

```
1 this.resourceUser.foo({id:1}).then(function (response) {
2   console.log(response)
3 });
```

Parte 2 - Criando um blog com Vue 1.0, Express e MongoDB



Atenção

Este capítulo usa o Vue 1.0

6. Express e MongoDB

Após revermos todos os conceitos relevantes do Vue, vamos criar um exemplo funcional de como integrá-lo a uma api. O objetivo desta aplicação é criar um simples blog com *Posts* e *Users*, onde é necessário realizar um login para que o usuário possa criar um Post.

6.1 Criando o servidor RESTful

Usaremos uma solução 100% Node.js, utilizando as seguintes tecnologias:

- **express:** É o servidor web que ficará responsável em receber as requisições web vindas do navegador e respondê-las corretamente. Não usaremos o *live-server*, mas o **express** tem a funcionalidade de autoloading através da biblioteca **nodemon**.
- **body-parser:** É uma biblioteca que obtém os dados JSON de uma requisição POST.
- **mongoose:** É uma adaptador para o banco de dados MongoDB, que é um banco NoSql que possui funcionalidades quase tão boas quanto a um banco de dados relacional.
- **jsonwebtoken:** É uma biblioteca node usada para autenticação via web token. Usaremos esta biblioteca para o login do usuário.

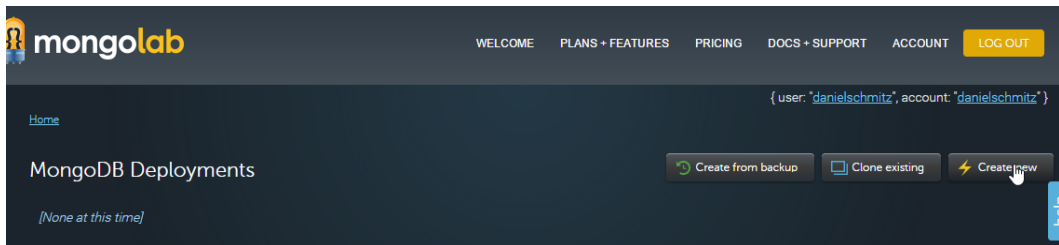
Todas estas tecnologias podem ser instaladas via **npm**, conforme será visto a seguir.

6.2 O banco de dados MongoDB

O banco de dados MongoDB possui uma premissa bem diferente dos bancos de dados relacionais (aqueles em que usamos SQL), sendo orientados a documentos auto

contidos (NoSql). Resumindo, os dados são armazenados no formato JSON. Você pode instalar o MongoDB em seu computador e usá-lo, mas nesta obra estaremos utilizando o serviço <https://mongolab.com/>¹ que possui uma conta gratuita para bancos públicos (para testes).

Acesse o link <https://mongolab.com/welcome/>² e clique no botão **Sign Up**. Faça o cadastro no site e logue (será necessário confirmar o seu email). Após o login, na tela de administração, clique no botão **Create New**, conforme a imagem a seguir:



Na próxima tela, escolha a aba **Single-node** e o plano **Sandbox**, que é gratuito, conforme a figura a seguir:

¹<https://mongolab.com/>

²<https://mongolab.com/welcome/>

Plan ([view pricing page](#)) :

Single-node Replica set cluster

These plan(s) are perfect for development/testing/staging environments as well as for utility instances that do not require high-availability.

Standard Line

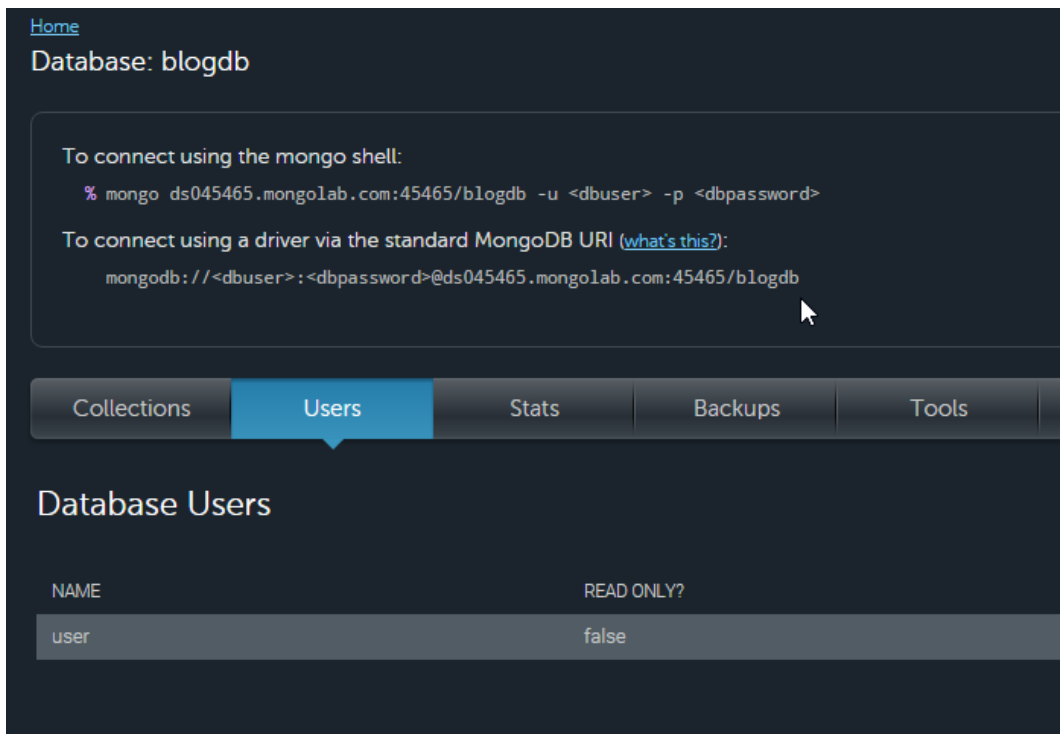
The most economical plans for production applications running on AWS. Plans come standard with 2 data nodes plus an arbiter node.

<input checked="" type="radio"/> Sandbox (shared, 0.5 GB)	FREE
<input type="radio"/> M3 Single-node (7,5 GB, 120 GB SSD block storage)	\$420
<input type="radio"/> M4 Single-node (15 GB, 240 GB SSD block storage)	\$835
<input type="radio"/> M5 Single-node (34,2 GB, 480 GB SSD block storage)	\$1310
<input type="radio"/> M6 Single-node (68,4 GB, 700 GB SSD block storage)	\$2045

Ainda nesta tela, forneça o nome do banco de dados. Pode ser `blog` e clique no botão `Create new MongoDB deployment`. Na próxima tela, com o banco de dados criado, acesse-o e verifique se a mensagem “A database user is required...” surge, conforme a imagem a seguir:



Clique no link e adicione um usuário qualquer (login e senha) que irá acessar este banco, conforme a imagem a seguir:



The screenshot shows the MongoDB Atlas web interface. At the top, there's a 'Home' link and the text 'Database: blogdb'. Below this, a box contains instructions for connecting to the database. The first instruction is 'To connect using the mongo shell:' followed by the command: `% mongo ds045465.mongolab.com:45465/blogdb -u <dbuser> -p <dbpassword>`. The second instruction is 'To connect using a driver via the standard MongoDB URI (what's this?):' followed by the URI: `mongodb://<dbuser>:<dbpassword>@ds045465.mongolab.com:45465/blogdb`. Below the instructions, there's a navigation bar with tabs: 'Collections', 'Users' (which is highlighted), 'Stats', 'Backups', and 'Tools'. Under the 'Users' tab, the title 'Database Users' is displayed. Below the title, there's a table with two columns: 'NAME' and 'READ ONLY?'. The table contains one row with the name 'user' and the value 'false'.

Home
Database: blogdb

To connect using the mongo shell:

```
% mongo ds045465.mongolab.com:45465/blogdb -u <dbuser> -p <dbpassword>
```

To connect using a driver via the standard MongoDB URI ([what's this?](#)):

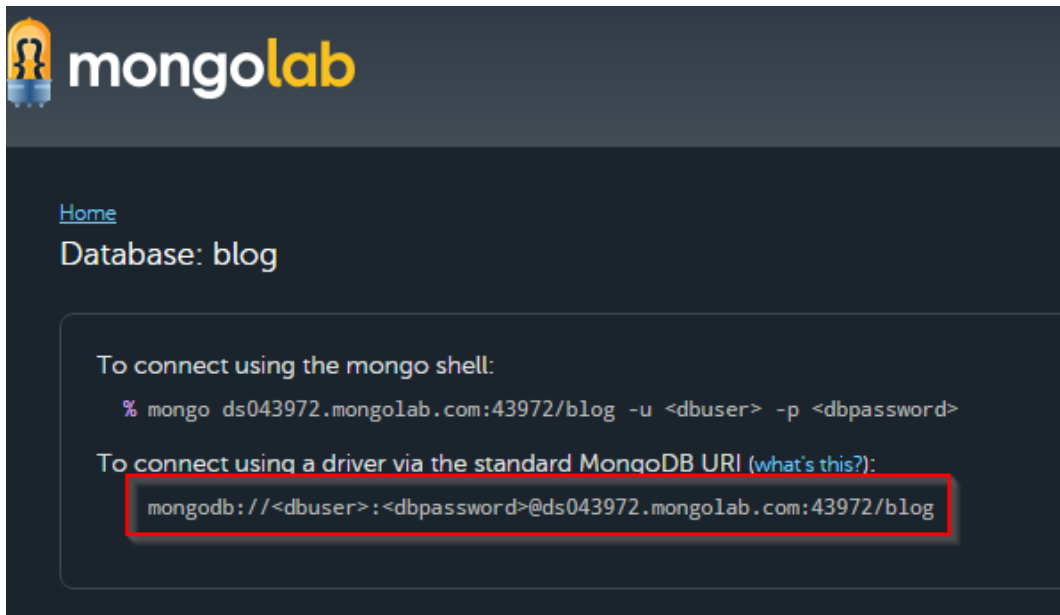
```
mongodb://<dbuser>:<dbpassword>@ds045465.mongolab.com:45465/blogdb
```

Collections Users Stats Backups Tools

Database Users

NAME	READ ONLY?
user	false

Após criar o usuário, iremos usar a URI de conexão conforme indicado na sua tela de administração:



6.3 Criando o projeto

Vamos usar o `vue-cli` para criar o projeto inicial, executando o seguinte comando:

```
1 vue init browserify-simple#1 blog
```



Se o comando `vue` não estiver disponível no sistema, execute `npm i -g vue-cli`, conforme foi explicado nos capítulos anteriores.

O diretório `blog` é criado, com a estrutura básica do Vue. Acesse o diretório e execute:

```
1 npm i
```

Isso irá instalar todos os pacotes npm iniciais do Vue. Para instalar os pacotes iniciais do servidor `express`, execute o seguinte comando:

```
1 npm i -D express body-parser jsonwebtoken mongoose
```

6.4 Estrutura do projeto

A estrutura do projeto está focada na seguinte arquitetura:

```
1 blog/
2 |- node_modules - Módulos do node que dão suporte a toda a\
3   aplicação
4 |- src - Código Javascript do cliente Vue da aplicação
5 |- model - Arquivos de modelo do banco de dados MongoDB
6 |- server.js - Arquivo que representa o servidor web em Exp\
7   ress
8 |- dist - Contém o arquivo compilado Javascript do Vue
9 |- index.html - Arquivo que contém todo o entry-point da ap\
10  licação
```

A pasta `src` contém toda a aplicação Vue, sendo que quando executarmos o Browserify via comando `npm`, o arquivo `build.js` será criado e copiado para a pasta `dist`.

6.5 Configurando os modelos do MongoDB

O Arquivo `server.js` contém tudo que é necessário para que a aplicação funcione como uma aplicação web. Iremos explicar passo a passo o que cada comando significa. Antes, vamos abordar os arquivos que representam o modelo MongoDB:

/model/user.js

```
1 var mongoose    = require('mongoose');
2 var Schema      = mongoose.Schema;
3
4 var userSchema = new Schema({
5   name: String,
6   login: String,
7   password: String
8 });
9
10 module.exports = mongoose.model('User', userSchema);
```

O modelo User é criado com o auxílio da biblioteca mongoose. Através do Schema criamos um modelo (como se fosse uma tabela) chamada User, que possui os campos name, login e password.

A criação do modelo Post é exibida a seguir:

/model/post.js

```
1 var mongoose    = require('mongoose');
2 var Schema      = mongoose.Schema;
3
4 var postSchema = new Schema({
5   title: String,
6   author: String,
7   body: String,
8   user: {type: mongoose.Schema.Types.ObjectId, ref: 'User'},
9   date: { type: Date, default: Date.now }
10 });
11
12 module.exports = mongoose.model('Post', postSchema);
```

Na criação do modelo `Post`, usamos quase todos os mesmos conceitos do `User`, exceto pelo relacionamento entre `Post` e `User`, onde configuramos que o `Post` possui uma referência ao modelo `User`.

6.6 Configurando o servidor Express

Crie o arquivo `server.js` para que possamos inserir todo o código necessário para criar a *api*. Vamos, passo a passo, explicar como este servidor é configurado.

`server.js`

```
1 var express = require('express');
2 var app = express();
3 var bodyParser = require('body-parser');
4 var jwt = require('jsonwebtoken');
```

Inicialmente referenciamos as bibliotecas que usaremos no decorrer do código. Também é criada a variável `app` que contém a instância do servidor *express*.

`server.js`

```
5 //secret key (use any big text)
6 var secretKey = "MySuperSecretKey";
```

Na linha 6 criamos uma variável chamada `secretKey` que será usada em conjunto com o módulo `jsonwebtoken`, para que possamos gerar um token de acesso ao usuário, quando ele logar. Em um servidor de produção, você deverá alterar o `MySuperSecretKey` por qualquer outro texto.

server.js

```
7 //Database in the cloud
8 var mongoose = require('mongoose');
9 mongoose.connect('mongodb://USER:PASSWORD@__URL__/blog', f\
10 unction (err) {
11     if (err) { console.error("error! " + err) }
12 });
```

Importamos a biblioteca mongoose na linha 8 e usamos o comando connect para conectar no banco de dados do serviço mongolab. Lembre de alterar o endereço de conexão com o que foi criado por você.

server.js

```
12 //bodyparser to read json post data
13 app.use(bodyParser.urlencoded({ extended: true }));
14 app.use(bodyParser.json());
```

Nas linhas 13 e 14 configuramos o bodyParser, através do método use do express, que está representado pela variável app. O bodyParser irá obter os dados de uma requisição em JSON e formatá-los para que possamos usar na aplicação.

server.js

```
15 //Load mongodb model schema
16 var Post = require('./model/post');
17 var User = require('./model/user');
```

Nas linhas 16 e 17 importamos os models que foram criados e que referenciam Post e User. Estes esquemas (“Schema”) serão referenciados pelas variáveis Post e User.

server.js

```
17 var router = express.Router();
```

A linha 18 cria o router, que é a preparação para o express se comportar como uma API. Um router é responsável em obter as requisições e executar um determinado código dependendo do formato da requisição. Geralmente temos quatro tipos de requisição:

- GET: Usada para obter dados. Pode ser acessada pelo navegador através de uma URL.
- POST: Usada para inserir dados, geralmente vindos de um formulário.
- DELETE: Usado para excluir dados
- PUT: Pode ser usado para editar dados. Não iremos usar PUT neste projeto, mas isso não lhe impede de usá-lo.

Além do tipo de requisição também temos a `url` e a passagem parâmetros, que veremos mais adiante.

server.js

```
18 //Static files
19 app.use('/', express.static(__dirname+'/'));
```

Na linha 19 configuramos o diretório / como estático, ou seja, todo o conteúdo neste diretório será tratado como um arquivo que, quando requisitado, deverá ser entregue ao requisitante.

Este conceito é semelhante ao diretório “webroot” de outros servidores web.

Também usamos a variável `__dirname` que retorna o caminho completo até o arquivo `server.js`. Isso é necessário para um futuro deploy da aplicação em servidores “reais”.

server.js

```
23 //middleware: run in all requests
24 router.use(function (req, res, next) {
25   console.warn(req.method + " " + req.url +
26     " with " + JSON.stringify(req.body));
27   next();
28 });
```

Na linha 24 criamos uma funcionalidade chamada “middleware”, que é um pedaço de código que vai ser executado em toda a requisição que o express receber. Na linha 25 usamos o método `console.warn` para enviar uma notificação ao console, exibindo o tipo de método, a url e os parâmetros Json. Esta informação é usada apenas em ambiente de desenvolvimento, pode-se comentá-la em ambiente de produção. O resultado produzido na linha 24 é algo semelhante ao texto a seguir:

```
1 POST /login with {"login":"foo","password":"bar"}
```

O método `JSON.stringify` obtém um objeto JSON e retorna a sua representação no formato texto.

Na linha 27 usamos o método `next()` para que a requisição continue o seu fluxo.

server.js

```
29 //middleware: auth
30 var auth = function (req, res, next) {
31   var token = req.body.token || req.query.token
32   || req.headers['x-access-token'];
33   if (token) {
34     jwt.verify(token, secretKey, function (err, decoded) {
35       if (err) {
36         return res.status(403).send({
37           success: false,
```

```
38         message: 'Access denied'
39     });
40 } else {
41     req.decoded = decoded;
42     next();
43 }
44 });
45 }
46 else {
47     return res.status(403).send({
48         success: false,
49         message: 'Access denied'
50     });
51 }
52 }
```

Na linha 30 temos outro middleware, chamado de `auth`, que é um pouco mais complexo e tem como objetivo verificar se o token fornecido pela requisição é válido. Quando o usuário logar no site, o cliente receberá um token que será usado em toda a requisição. Esta forma de processamento é diferente em relação ao gerenciamento de sessões no servidor, muito comum em autenticação com outras linguagens como PHP e Java.

Na linha 31 criamos a variável `token` que recebe o conteúdo do token vindo do cliente. No caso do `blog`, sempre que precisamos repassar o token ao servidor, iremos utilizar o cabeçalho `http` repassando a variável `x-access-token`.

Na linha 34 usa-se o método `jwt.verify` para analisar o token, repassado pelo cliente. Veja que a variável `secretKey` é usada neste contexto, e que no terceiro parâmetro do método `verify` é repassado um *callback*.

Na linha 35 verificamos se o *callback* possui algum erro. Em caso positivo, o token repassado não é válido e na linha 36 retornamos o erro através do método `res.status(403).send()` onde o *status* 403 é uma informação de acesso não autorizado (Erro `http`, assim como 404 é o *not found*).

Na linha 40 o token é válido, pois nenhum erro foi encontrado. O objeto decodificado é armazenado na variável `req.decoded` para que possa ser utilizada posteriormente e o método `next` irá continuar o fluxo de execução da requisição.

A linha 46 é executada se não houver um token sendo repassado pelo cliente, retornando também um erro do tipo 403.

server.js

```
53 //simple GET / test
54 router.get('/', function (req, res) {
55   res.json({ message: 'hello world!' });
56 });
```

Na linha 54 temos um exemplo de como o router do express funciona. Através do método `router.get` configuramos a url “/”, que quando chamada irá executar o *callback* que repassamos no segundo parâmetro. Este *callback* configura a resposta do router, através do método `res.json`, retornando o objeto `json { message: 'hello world!' }`.

server.js

```
56 router.route('/users')
57 .get(auth, function (req, res) {
58   User.find(function (err, users) {
59     if (err)
60       res.send(err);
61     res.json(users);
62   });
63 })
64 .post(function (req, res) {
65   var user = new User();
66   user.name = req.body.name;
67   user.login = req.body.login;
68   user.password = req.body.password;
69
```

```
70     user.save(function (err) {  
71         if (err)  
72             res.send(err);  
73         res.json(user);  
74     })  
75 });
```

Na linha 56 começamos a configurar o roteamento dos usuários, que será acessado inicialmente pela url “/users”. Na linha 57 configuramos uma requisição GET à url /users, adicionando como *middleware* o método `auth`, que foi criado na linha 29. Isso significa que, antes de executar o *callback* do método “GET /users” iremos verificar se o token repassado pelo cliente é válido. Se for válido, o *callback* é executado e na linha 58 usamos o schema `User` para encontrar todos os usuários do banco. Na linha 61 retornamos este array de usuário para o cliente.

Na linha 64 configuramos o método POST /users que tem como finalidade cadastrar o usuário. Perceba que neste método não usamos o *middleware* `auth`, ou seja, para executá-lo não é preciso estar autenticado. Na linha 65 criamos uma variável que usa as propriedades do “Schema” `User` para salvar o registro. Os dados que o cliente repassou ao express são acessados através da variável `req.body`, que está devidamente preenchida graças ao `body-parser`.

O método `user.save` salva o registro no banco, e é usado o `res.json` para retornar o objeto `user` ao cliente.

server.js

```
76 router.route('/login').post(function (req, res) {  
77     if (req.body.isNew) {  
78         User.findOne({ login: req.body.login }, 'name')  
79         .exec(function (err, user) {  
80             if (err) res.send(err);  
81             if (user != null) {  
82                 res.status(400).send('Login Existente');  
83             }  
84             else {
```

```
85     var newUser = new User();
86     newUser.name = req.body.name;
87     newUser.login = req.body.login;
88     newUser.password = req.body.password;
89     newUser.save(function (err) {
90         if (err) res.send(err);
91         var token = jwt.sign(newUser, secretKey, {
92             expiresIn: "1 day"
93         });
94         res.json({ user: newUser, token: token });
95     });
96 }
97 });
98 } else {
99     User.findOne({ login: req.body.login,
100     password: req.body.password }, 'name')
101     .exec(function (err, user) {
102         if (err) res.send(err);
103         if (user != null) {
104             var token = jwt.sign(user, secretKey, {
105                 expiresIn: "1 day"
106             });
107             res.json({ user: user, token: token });
108         } else {
109             res.status(400).send('Login/Senha incorretos');
110         }
111     });
112 }
113 });
```

Na linha 76 temos a funcionalidade para o Login, acessado através da url /login. Quando o cliente faz a requisição “POST /login” verificamos na linha 77 se a

propriedade `isNew` é verdadeira, pois é através dela que estamos controlando se o usuário está tentando logar ou está criando um novo cadastro.

Na linha 78 usamos o método `findOne` repassando o filtro `{login:req.body.login}` para verificar se o login que o usuário preencher existe. O segundo parâmetro deste método são os campos que deverão ser retornados, caso um usuário seja encontrado. O método `.exec` irá executar o `findOne` e o *callback* será chamado, onde podemos retornar um erro, já que não é possível cadastrar o mesmo login.

Se `req.body.isNew` for falso, o código na linha 99 é executado e fazemos a pesquisa ao banco pelo login e senha. Se houver um usuário com estas informações, usamos o método `jwt.sign` na linha 103 para criar o token de autenticação do usuário, e o retornamos na linha 106. Se não houver um usuário no banco com o mesmo login e senha, retornamos o erro na linha 108.

server.js

```
114   router.route('/posts/:post_id?')
115     .get(function (req, res) {
116       Post
117         .find()
118         .sort([[ 'date', 'descending' ]])
119         .populate('user', 'name')
120         .exec(function (err, posts) {
121           if (err)
122             res.send(err);
123           res.json(posts);
124         });
125     })
126     .post(auth, function (req, res) {
127       var post = new Post();
128       post.title = req.body.title;
129       post.text = req.body.text;
130       post.user = req.body.user._id;
131       if (post.title==null)
132         res.status(400).send('Título não pode ser nulo');
```

```
133     post.save(function (err) {
134         if (err)
135             res.send(err);
136         res.json(post);
137     });
138 })
139 .delete(auth, function (req, res) {
140     Post.remove({
141         _id: req.params.post_id
142     }, function(err, post) {
143         if (err)
144             res.send(err);
145         res.json({ message: 'Successfully deleted' });
146     });
147 });
```

Na linha 114 usamos `/posts/:post_id?` para determinar a url para obtenção de posts. O uso do `:post_id` adiciona uma variável a url, por exemplo `/posts/5`. Já que usamos `/posts/:post_id?`, o uso do `?` torna a variável opcional.

Na linha 115 estamos tratando o método “GET `/posts`” que irá obter todos os posts do banco de dados. Na linha 118 usamos o método `sort` para ordenar os posts, e na linha 119 usamos o método `populate` para adicionar uma referência ao modelo `user`, que é o autor do `Post`. Esta referência é possível porque adicionamos na definição do *schema* de `Post`.

Na linha 126 criamos o método “POST `/posts`” que irá adicionar um `Post`. Criamos uma validação na linha 131 e usamos o método `Post.save()` para salvar o post no banco de dados.

Na linha 139 adicionamos o método “DELETE `/posts/`”, onde o post é apagado do banco de dados. Para apagá-lo, usamos o método `Post.remove` na linha 140, repassando o id (chave) do `Post` e usando o parâmetro `req.params.post_id`, que veio da url `/posts/:post_id?`.

server.js

```
148 //register router
149 app.use('/api', router);
150 //start server
151 var port = process.env.PORT || 8080;
152 app.listen(port);
153 console.log('Listen: ' + port);
```

Finalizando o script do servidor, apontamos a variável `router` para o endereço `/api`, na linha 149. Com isso, toda a api será exposta na url “/api”. Por exemplo, para obter todos os posts do banco de dados, deve-se fazer uma chamada GET ao endereço “/api/posts”. Nas linha 151 e 152 definimos a porta em que o servidor express estará “escutando” e na linha 153 informamos via `console.log` qual a porta foi escolhida.

6.7 Testando o servidor

Para testar o servidor Web, podemos simplesmente executar o seguinte comando:

```
1 $ node server.js
```

Onde temos uma simples resposta: “Listen: 8080”. Se houver alguma alteração no arquivo `server.js`, esta alteração não será refletida na execução corrente do servidor, será necessário terminar a execução e reiniciar. Para evitar este retrabalho, vamos instalar a biblioteca `nodemon` que irá recarregar o servidor sempre que o o arquivo `server.js` for editado.

```
1 $ npm install nodemon -g
```

Após a instalação, execute:

```
1 $ nodemon server.js
2
3 [nodemon] 1.8.1
4 [nodemon] to restart at any time, enter `rs`
5 [nodemon] watching: *.*
6 [nodemon] starting `node server.js`
7 Listen: 8080
```

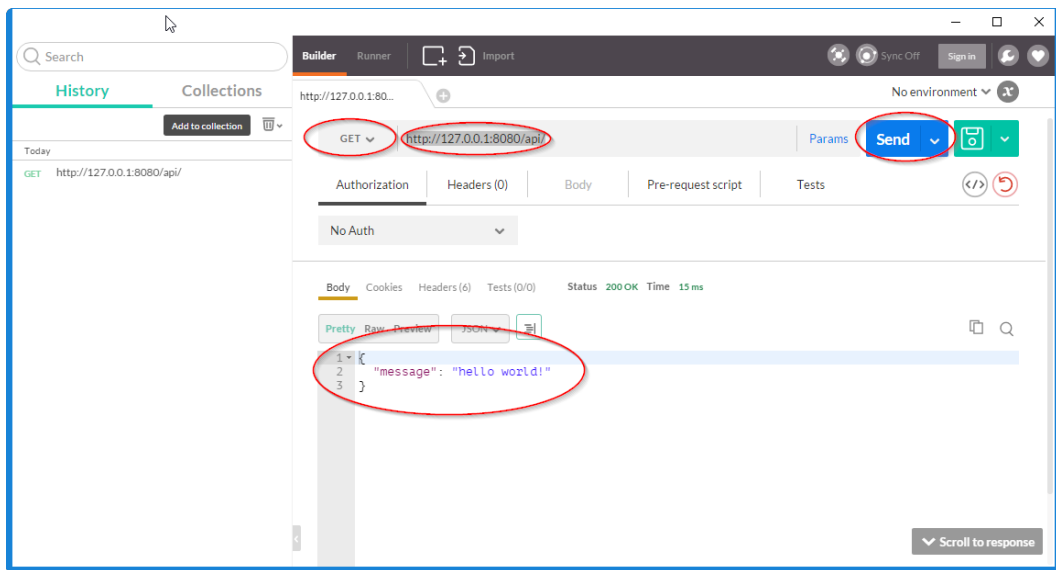
A resposta já indica que, sempre quando o arquivo `server.js` for atualizado, o comando `node server.js` também será.

6.8 Testando a api sem o Vue

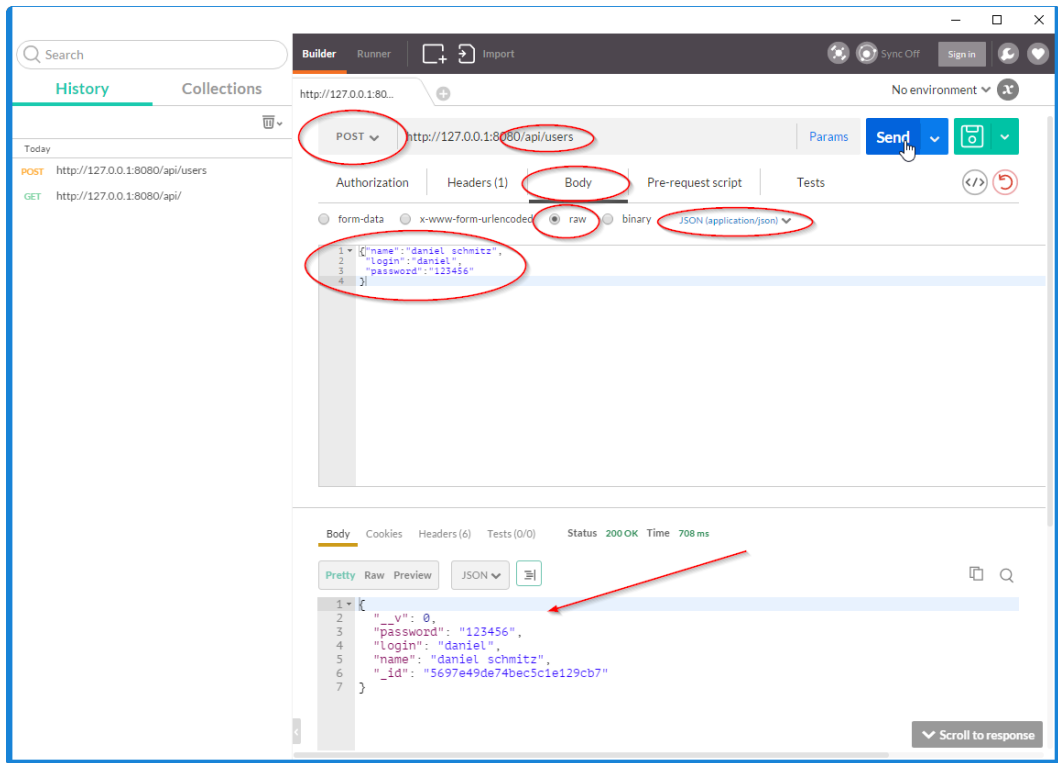
Pode-se testar a api que acabamos de criar, enviando e recebendo dados, através de um programa capaz de realizar chamadas Get/Post ao servidor. Um destes programas se chama **Postman**³, que pode ser instalado com um plugin para o Google Chrome.

Por exemplo, para testar o endereço `http://127.0.0.1:8080/api/`, configuramos o Postman da seguinte forma:

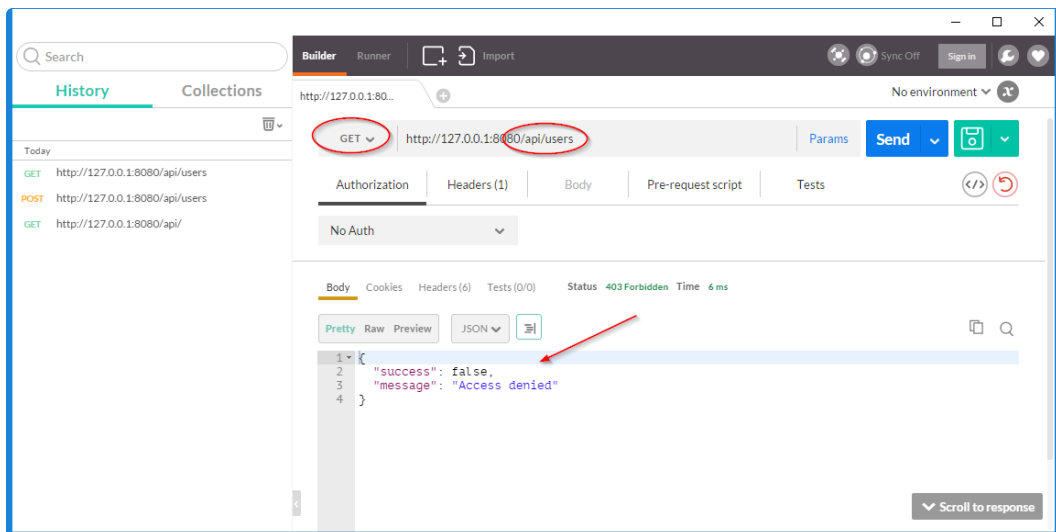
³<https://www.getpostman.com/>



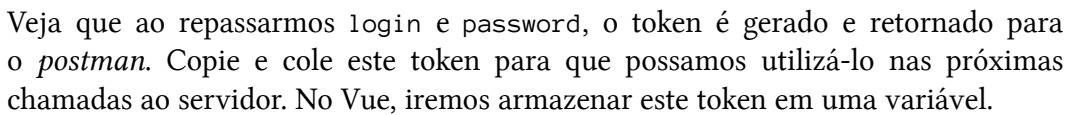
Perceba que obtemos a resposta “hello world”, conforme configurado no servidor. Para criar um usuário, podemos realizar um POST à url `/users` repassando os seguintes dados:



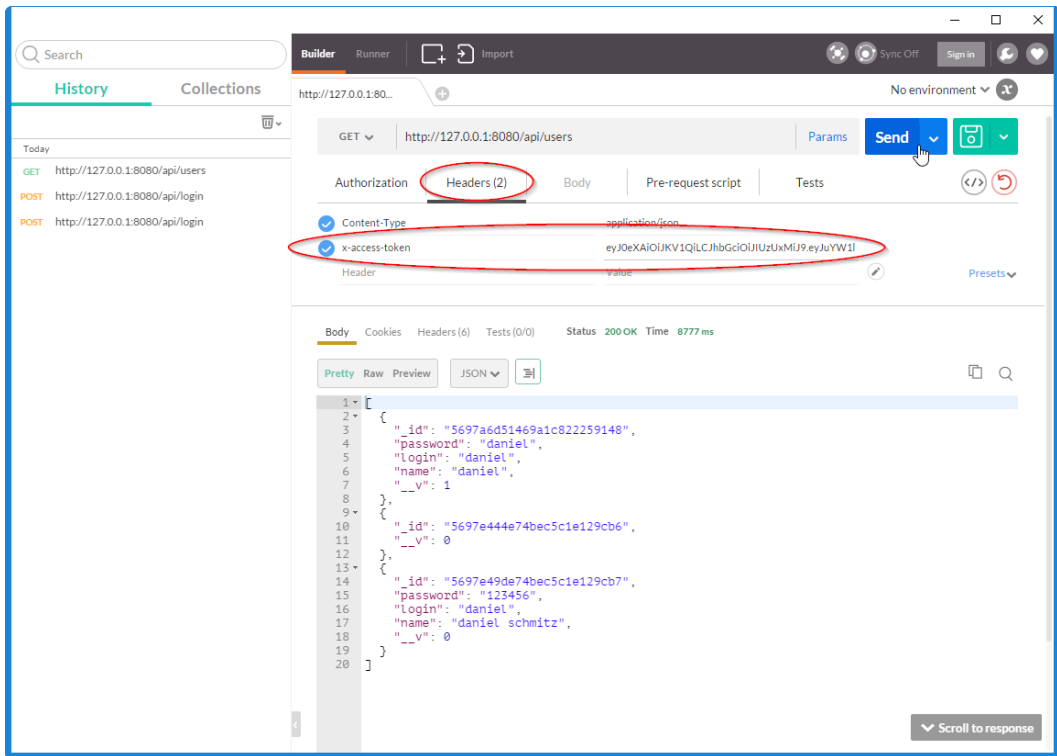
Para testar o login, vamos tentar acessar a url `/api/users`. Como configuramos que esta url deve passar pelo *middleware*, o token não será encontrado e um erro será gerado:



Para realizar o login, acessamos a URL `/api/login`, da seguinte forma:



Com o token, é possível retornar a chamada GET /users e repassá-lo no cabeçalho da requisição HTTP, conforme a imagem a seguir:



Veja que com o token os dados sobre os usuários são retornados. Experimente alterar algum caractere do token e refazer a chamada, para obter o erro “Failed to authenticate”.

Parte 3 - Conceitos avançados

7. Vuex e Flux

Neste capítulo tratamos do Vuex, a implementação do conceito [Flux](https://facebook.github.io/flux/)¹ no Vue. fl

7.1 O que é Flux?

Flux não é uma arquitetura, framework ou linguagem. Ele é um conceito, um paradigma. Quando aprendemos a usar Flux no Vue (ou em outro framework/linguagem), criamos uma forma única de atualização de dados que assegura que tudo esteja no seu devido lugar. Essa forma de atualizar os dados também garante que os componentes de visualização mantenham o seu correto estado, agindo reativamente. Isso é muito importante para aplicações SPAs complexas, que possuem muitas tabelas, muitas telas e, consequentemente, eventos e alterações de dados a todo momento.

7.2 Conhecendo os problemas

Quando criamos um SPA complexo, não podemos lidar com os dados da aplicação apenas criando variáveis globais, pois serão tantas que chega um momento no qual não é possível mais controlá-las. Além da eventual “bagunça”, em aplicações maiores, manter o correto estado de cada ponto do sistema torna-se um problema para o desenvolvedor.

Por exemplo, em um carrinho de compras, deseja-se testar as mais variadas situações na interface após o usuário incluir um item no carrinho. Com Vuex é possível controlar melhor este estado. Pode-se, por exemplo, carregar um estado onde o usuário possui 10 itens no carrinho, sem você ter que adicioná-los manualmente, o que lhe economiza tempo. Também pode-se trabalhar com dados “desligados” da API Rest, enquanto o desenvolvedor está trabalhando na implementação da mesma.

Testes unitários e funcionais também se beneficiam com Vuex, pois para chegar a um estado da sua aplicação, pode-se facilmente alterar os dados da mesma.

¹<https://facebook.github.io/flux/>

7.3 Quando usar?

Use Vuex em aplicações reais, que você irá colocar em produção. Para aprender o básico do Vue não é necessário Vuex. Para qualquer aplicação que irá algum dia entrar em produção (ou produto de cliente ou seu), use-o!

7.4 Conceitos iniciais

Para que possamos compreender bem o Vuex, é preciso inicialmente dominar estes seguintes conceitos:

Store

Store é o principal conceito Flux/Vuex, pois é nele onde o estado da aplicação está armazenado. Quando dizemos **estado**, você pode pensar em um conjunto de dezenas de variáveis que armazenam dados. Como sabemos, o Vue possui uma camada de visualização reativa, que observa variáveis e altera informações na tela de acordo com essas variáveis, ou seja, de acordo com o estado delas. O Store dentro do Vuex é o “lugar” onde as variáveis são armazenadas, onde o estado é armazenado e onde a View o observa para determinar o seu comportamento.

Action

Definido como a ação que pode alterar o estado. Uma action é caracterizada como o **Único** lugar que pode alterar uma informação do estado. Isso significa que, na sua aplicação, ao invés de alterar o Store diretamente, você irá chamar uma Action que, por sua vez, altera o estado de alguma variável do Store.

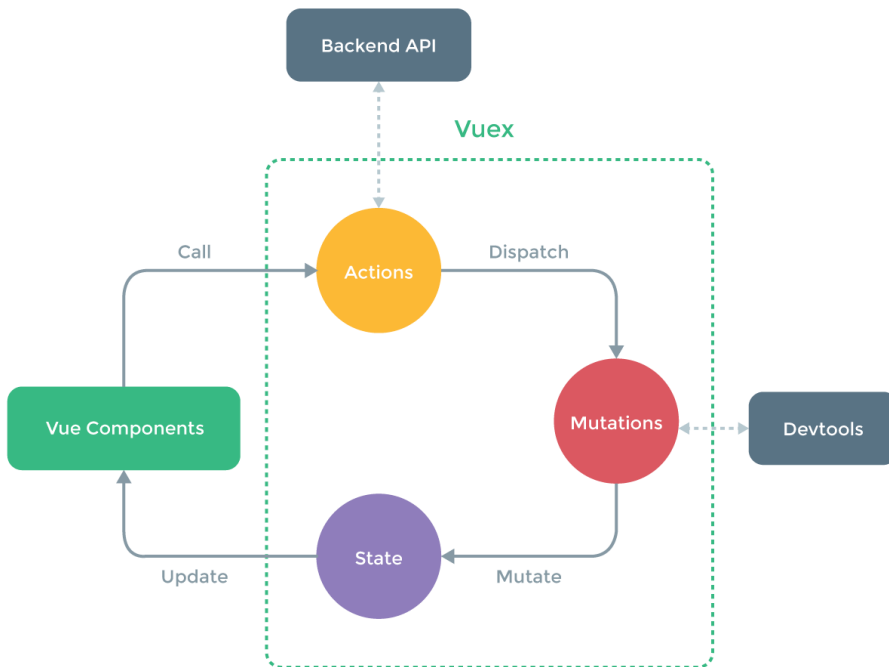
Mutation

Um “mutation” pode ser conceituado como “o evento que muda o store”. Mas não era o Action que fazia isso? Sim, é o action que faz isso, mas ele faz isso usando “mutations”, e são os mutations que acessam diretamente a variável no store. Pode parecer um passo a mais sem necessidade, mas os mutations existem justamente para controlar diretamente o State. Neste fluxo, a sua aplicação chama uma action, que chama o mutation, que altera a variável no Store.

Getters

Getters são métodos responsáveis em observar as variáveis que estão no Store e fornecer essa informação a sua aplicação. Com isso, garantimos que as alterações no estado (ou store) do Vuex irá refletir corretamente na aplicação.

Como pode-se ver, a aplicação nunca acessa ou altera a a variável (Store) diretamente. Existe um fluxo que deve ser seguido para deixar tudo no lugar. A imagem a seguir ilustra como esse fluxo é realizado:



7.5 Exemplo simples

Nesse primeiro exemplo vamos mostrar um simples contador, exibindo o seu valor e um botão para adicionar e remover uma unidade. Tudo será feito em apenas um

arquivo, para que possamos entender o processo.

Criando o projeto

Vamos criar um novo projeto, chamado de vuex-counter. Usando o vue-cli, execute o seguinte comando:

```
1 vue init browserify-simple vuex-counter
```

Entre no diretório e adicione o vuex, da seguinte forma:

```
1 cd vuex-counter
2 npm i -S vuex
```

Para instalar todas as bibliotecas restantes, faça:

```
1 npm i
```

Execute o seguinte comando para compilar o código vue e iniciar o servidor web:

```
1 npm run dev
```

Ao acessar <http://localhost:8080/>, você verá uma mensagem “Hello Vue”. O projeto vuex-counter está pronto para que possamos implementar o vuex nele.

Criando componentes

Vamos criar dois componentes chamados de `Increment` e `Display`. A ideia aqui é reforçar o conceito que o vuex trabalha no SPA, em toda aplicação, sendo que os componentes podem usá-lo quando precisarem.

O componente `Display.vue`, criado no diretório `src`, possui inicialmente o seguinte código:

```
1 <template>
2   <div>
3     <h3>Count is 0</h3>
4   </div>
5 </template>
6
7 <script>
8   export default {
9   }
10 </script>
```

Já o componente `Increment.vue`, criado no diretório `src`, possui dois botões:

```
1 <template>
2   <div>
3     <button>+1</button>
4     <button>-1</button>
5   </div>
6 </template>
7
8 <script>
9   export default {
10   }
11 </script>
```

Para adicionarmos esses dois componentes no componente principal da aplicação (`App.vue`), basta usar a propriedade `components`, da seguinte forma:

```
1 <template>
2   <div id="app">
3     <Display></Display>
4     <Increment></Increment>
5   </div>
6 </template>
7
8
9 <script>
10   import Display from './Display.vue'
11   import Increment from './Increment.vue'
12
13   export default {
14     components: {
15       Display, Increment
16     },
17     data () {
18       return {
19         msg: 'Hello Vue!'
20       }
21     }
22   }
23 </script>
```

Após alterar o código do App.vue, recarregue a página e verifique se surge uma mensagem “Count is 0” e dois botões.

Incluindo Vuex

Com o vuex devidamente instalado, podemos criar o arquivo que será o Store da aplicação, ou seja, é nesse arquivo que iremos armazenar as variáveis na qual a aplicação usa.

Crie o arquivo store.js no diretório src, com o seguinte código:

```
1  import Vue from 'vue'
2  import Vuex from 'vuex'
3
4  Vue.use(Vuex)
5
6  const state = {
7
8  }
9
10 const mutations = {
11
12 }
13
14 export default new Vuex.Store({
15   state,
16   mutations
17 })
```

No `store.js` estamos importando o Vuex de `vuex`, que foi previamente instalado pelo npm. Então dizemos ao Vue que o plugin Vuex será usado. Cria-se duas constantes chamadas de `state`, que é onde as variáveis da aplicação serão armazenadas, e `mutations` que são os eventos que alteram as variáveis do `state`.

No final cria-se o Store através do `new Vuex.Store`. Perceba que exportamos a classe, então podemos importá-la na aplicação principal, da seguinte forma:

```
1 <template>
2   <div id="app">
3     <Display></Display>
4     <Increment></Increment>
5   </div>
6 </template>
7
8
9 <script>
10   import Display from './Display.vue'
11   import Increment from './Increment.vue'
12   import store from './store.js'
13
14   export default {
15     components: {
16       Display, Increment
17     },
18     data () {
19       return {
20         msg: 'Hello Vue!'
21       }
22     },
23     store
24   }
25 </script>
```

Criando uma variável no state

No arquivo `store.js` (no qual chamaremos de `Store`) temos a constante `state` que é onde informamos o “estado” da aplicação, ou seja, as variáveis que definem o comportamento do sistema. No nosso contador, precisamos de uma variável que irá armazenar o valor do contador, que poderá ser incrementado. Vamos chamar esse contador de `count`, com o valor inicial 0, veja:

```
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 Vue.use(Vuex)
5
6 const state = {
7   count: 0
8 }
9
10 const mutations = {
11
12 }
13
14 export default new Vuex.Store({
15   state,
16   mutations
17 })
```

Criando mutations

Com a variável pronta, podemos definir as mutations que irão acessá-la. Lembre-se, somente os mutations podem acessar o state. O código para criar o mutation INCREMENT e o mutation DECREMENT é definido a seguir:

```
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 Vue.use(Vuex)
5
6 const state = {
7   count: 0
8 }
9
```

```
10 const mutations = {
11   INCREMENT(state){
12     state.count++;
13   },
14   DECREMENT(state){
15     state.count--;
16   }
17 }
18
19 export default new Vuex.Store({
20   state,
21   mutations
22 })
```

Os mutations, por convenção, estão em letras maiúsculas. Eles possuem como primeiro parâmetro o state, e a partir dele podem acessar a variável count.

Criando actions

As ações, que chamaremos de actions, são as funções que chamam os mutations. Por convenção, as actions são armazenadas em um arquivo separado, chamado de actions.js, no diretório src, inicialmente com o seguinte código:

```
1 export const incrementCounter = function ({ dispatch, state\
2   }) {
3     dispatch('INCREMENT')
4   }
5
6 export const decrementCounter = function ({ dispatch, state\
7   }) {
8     dispatch('DECREMENT')
9   }
```

Neste momento, as *actions* apenas disparam o mutation. Pode parecer um passo extra desnecessário, mas será útil quando trabalharmos com ajax.

Criando getters

Os getters são responsáveis em retornar o valor de uma variável de um state. As views (componentes) consomem os getters para observar as alterações que ocorrem no State.

Crie o arquivo `getters.js` no diretório `src` com o seguinte código:

```
1 export function getCount (state) {  
2   return state.count  
3 }
```

Alterando o componente Display para exibir o valor do contador

Com a estrutura pronta (state,mutation,action,getter) podemos finalmente voltar aos componentes e usar o Vuex. Primeiro, vamos ao componente `Display` e usar o getter para atualizar o valor da mensagem `Count is 0`, veja:

```
1 <template>  
2   <div>  
3     <h3>Count is {{getCount}}</h3>  
4   </div>  
5 </template>  
6  
7 <script>  
8  
9   import { getCount } from './getters'  
10  
11  export default {  
12    vuex: {  
13      getters: {  
14        getCount  
15      }  
16    }  
17  }
```



```
17   }  
18 </script>
```



Não esqueça de utilizar { e } no import, para que a função `getCount` possa ser importada com êxito. Isso é necessário devido ao novo *destructuring array* do ECMAScript 6.

O componente `Display` importa o método `getCount` do arquivo `getters.js`, então é referenciado na configuração do objeto `Vuex`, e finalmente é usado no `template` como um `bind` comum. Esta configuração irá garantir que, quando o `state.count` for alterado pelo `mutation`, o valor no `template` também será alterado.

Alterando o component Increment

Voltando ao componente `Increment.vue`, vamos referenciar as ações que estão no `actions.js` e usá-las nos botões para inserir e remover uma unidade do contador.

```
1 <template>  
2   <div>  
3     <button @click="incrementCounter">+1</button>  
4     <button @click="decrementCounter">-1</button>  
5   </div>  
6 </template>  
7  
8 <script>  
9   import { incrementCounter, decrementCounter } from './act\  
10 ions'  
11  
12   export default {  
13     vuex: {  
14       actions: {  
15         incrementCounter, decrementCounter
```

```
16      }  
17    }  
18  }  
19 </script>
```

Perceba que importamos `incrementCounter` e `decrementCounter`, e então referenciamos ele na propriedade `vuex` do componente. Com isso, o `vuex` cuida de realizar todas as configurações necessárias para fazer o `bind`. Então o método `incrementCounter` é referenciado no botão.

Quando o usuário clicar no botão `+1`, o action `incrementCounter` será chamado, que por sua vez chama o mutation `INCREMENT`, que altera o `state.count`. O `getter` é notificado e consequentemente o componente `Display` altera o valor.

Testando a aplicação

Certifique-se da aplicação estar devidamente compilada e clique nos botões `+1` e `-1` para ver o contador aumentando ou diminuindo. Caso isso não aconteça, analise no console da aplicação (F12 no Chrome) se houve algum erro.

Com a aplicação funcionando, podemos estudar mais alguns detalhes antes de prosseguir para um exemplo real.

7.6 Revendo o fluxo

Com a aplicação pronta, podemos ver como o Flux age no projeto, revendo desde o usuário clicar no botão até a atualização do contador do `Display`.

- Usuário clica no botão do componente `Increment`
- O função `incrementCounter` é chamada. Esta função está no `actions.js`
- A ação `incrementCounter` chama o mutation através do `dispatch(' INCREMENT')`, onde o mutation está no `store.js`
- O mutation `INCREMENT` irá alterar o valor da variável `state.count`
- O `Vuex` encarrega-se de notificar esta mudança

- Como temos um getter que observa esta variável, os componentes que o usam também serão notificados.
- Com o componente devidamente observando o getter, a variável que possui o bind a ele também altera.

7.7 Chrome vue-devtools

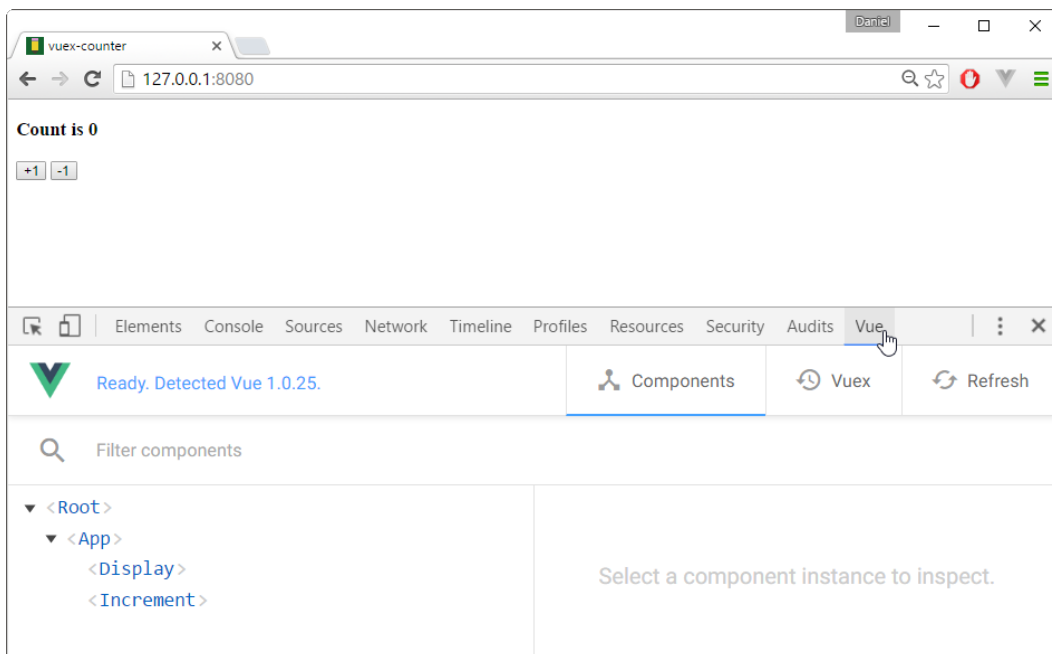
O Chrome [vue-devtools](https://github.com/vuejs/vue-devtools)² é um plugin para o navegador Google Chrome que irá lhe ajudar melhor no debug das aplicações em Vue. Você pode obter o Vue DevTools no [Chrome Web Store](https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd)³.

Após a instalação, reinicie o seu navegador e acesse novamente a aplicação vuex-counter (que deve estar disponível no endereço <http://127.0.0.1:8080/>).

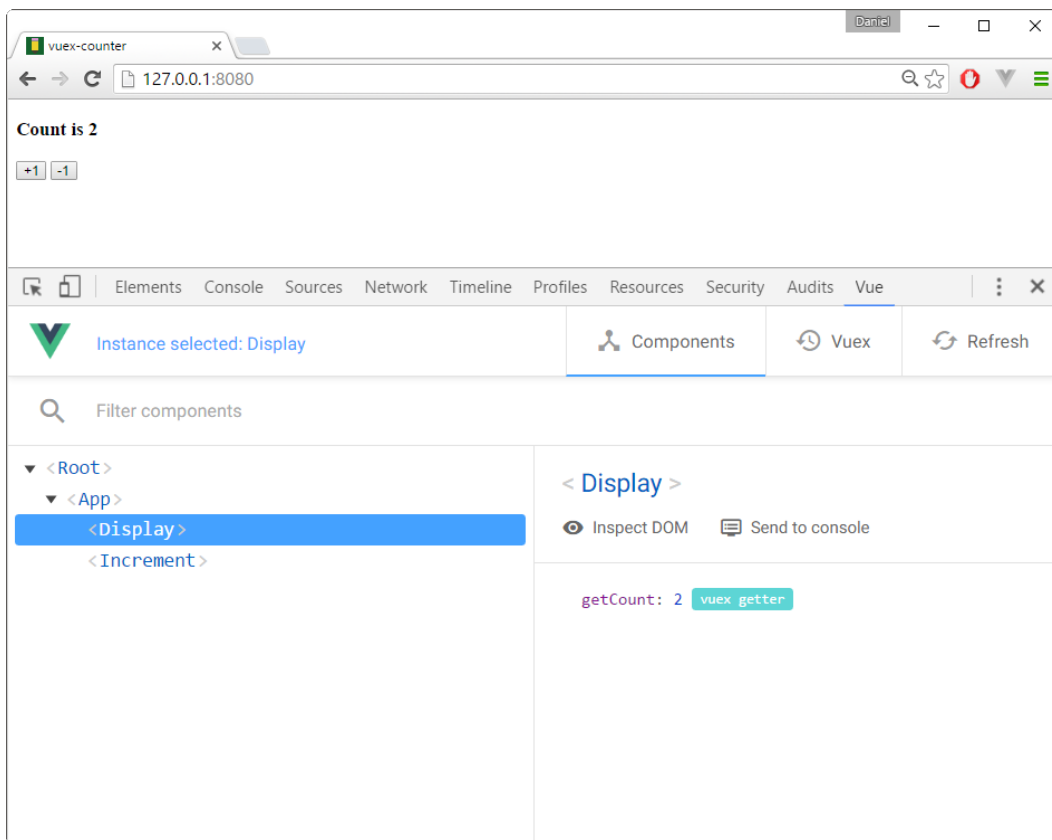
Tecla F12 para abrir o Chrome Dev Tools, e clique na aba Vue, conforme a imagem a seguir:

²<https://github.com/vuejs/vue-devtools>

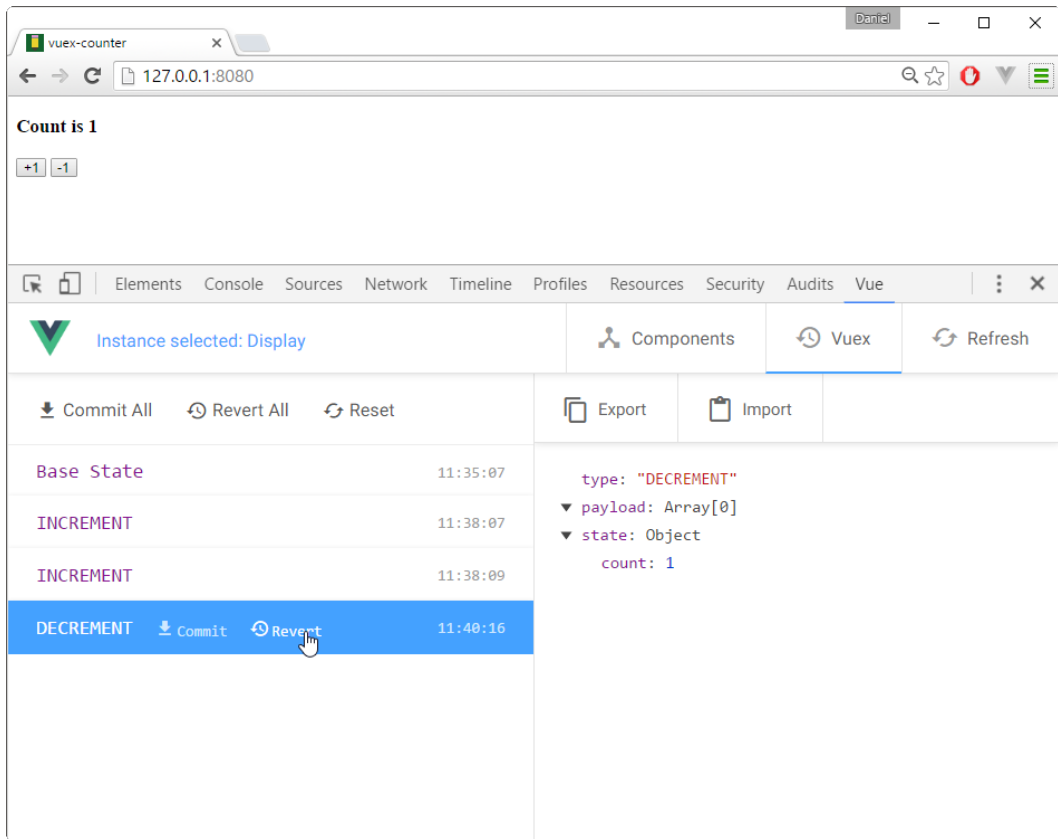
³<https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd>



Na aba Vue, você pode verificar como os componentes estão sendo inseridos na página, e para cada componente pode-se avaliar cada estado das variáveis, como no exemplo a seguir:



Na aba Vuex, pode-se ver os mutations sendo executados, e suas respectivas alterações no estado do state. Pode-se, inclusive, fazer um “rollback” de um mutation, conforme a imagem a seguir:



Esse rollback nos mutations podem ser úteis quando estamos criando uma estrutura mais complexa e testando a aplicação, pois podemos “voltar no tempo” e executar novamente o mesmo processo, alterando inclusive o código javascript.

Por exemplo, suponha que uma ação de adicionar um item em um carrinho de compras dispare alguns mutations. Eles estarão mapeados no Vuex. Caso altere algum código javascript e recompile a aplicação, fazendo um reload da página logo após o processo, pode-se voltar na aba Vuex que a pilha de ações dos mutations estarão mantidas, bastando clicar no botão “Commit All” para aplicar todas as mutations novamente.

7.8 Repassando dados pelo vuex

Na maioria das vezes precisamos passar dados do componente (view) para o Store. Isso pode ser feito respeitando a passagem de parâmetros entre cada passo do fluxo.

Voltando ao projeto vuex-counter, vamos adicionar uma caixa de texto ao componente Increment, e um botão para incrementar a quantidade digitada pelo usuário.

```
1 <template>
2   <div>
3     <button @click="incrementCounter">+1</button>
4     <button @click="decrementCounter">-1</button>
5   </div>
6   <div>
7     <input type="text" v-model="incrementValue">
8     <button @click="tryIncrementCounterWithValue">increment\
9 </button>
10  </div>
11 </template>
12 ...
```

Veja que a caixa de texto possui o v-model ligando a variável incrementValue. Além da caixa de texto, o botão irá chamar o método tryIncrementCounterWithValue. Vamos analisar o código script do componente increment:

```
1 <script>
2   import { incrementCounter, decrementCounter, incrementCou\
3 nterWithValue } from './actions'
4
5   export default {
6     vuex: {
7       actions: {
8         incrementCounter, decrementCounter, incrementCounterW\
```

```
9   ithValue
10   }
11   },
12   data () {
13     return{
14       incrementValue:0
15     }
16   },
17   methods: {
18     tryIncrementCounterWithValue(){
19       this.incrementCounterWithValue(this.incrementValue)
20     }
21   }
22 }
23 </script>
```

Veja que importamos uma terceira Action chamada `incrementCounterWithValue`, que iremos criar logo a seguir. Esta action é chamada pelo método `tryIncrementCounterWithValue`, no qual usamos apenas para demonstrar que as Actions podem ser chamadas pelos métodos do componente ou diretamente no template. O método `tryIncrementCounterWithValue` repassa o valor `this.incrementValue` que foi definido no data do componente.

A nova action no arquivo `actions.js` é criada da seguinte forma:

```
1  export const incrementCounter = function ({ dispatch, state\
2  }) {
3    dispatch('INCREMENT')
4  }
5
6  export const decrementCounter = function ({ dispatch, state\
7  }) {
8    dispatch('DECREMENT')
9  }
```



```
10
11 export const incrementCounterWithValue = function ({ dispatch\
12 ch, state },value) {
13   dispatch('INCREMENTVALUE',parseInt(value))
14 }
```

Perceba que o método `incrementCounterWithValue` possui o parâmetro `value`, após a definição do `dispatch` e do `state`. Este parâmetro `value` será usado para chamar o mutation `INCREMENTVALUE`, repassando o `value` devidamente convertido para `int`.

O mutation é exibido a seguir:

```
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 Vue.use(Vuex)
5
6 const state = {
7   count: 0
8 }
9
10 const mutations = {
11   INCREMENT(state){
12     state.count++;
13   },
14   DECREMENT(state){
15     state.count--;
16   },
17   INCREMENTVALUE(state,value){
18     state.count=state.count+value
19   },
20 }
21
22 export default new Vuex.Store({
```

```
23   state,  
24   mutations  
25 })
```

Veja que o mutation `INCREMENTVALUE` possui um segundo parâmetro, chamado de `value`, que é justamente o parâmetro `value` repassado pela `action`. Este parâmetro é usado para incrementar o valor na variável `state.count`.

7.9 Tratando erros

As `actions` do Vuex ficam responsáveis em lidar com possíveis erros que possam invalidar a chamada ao mutation. Uma das formas de tratar este erro é usar o bloco de código `try...catch...exception`. Por exemplo, na `action` poderíamos ter:

```
1  ...  
2  export const incrementCounterWithValue = function ({ dispatch\  
3  ch, state }, value) {  
4  
5    let intValue = parseInt(value);  
6    if (isNaN(intValue)) {  
7      throw "Impossível converter para número inteiro"  
8    } else {  
9      dispatch('INCREMENTVALUE', intValue)  
10    }  
11  }
```

Se o valor de `value` não puder ser convertido, é disparado uma exceção, que deve ser capturada no componente, conforme o exemplo a seguir:

```
1 ...
2 tryIncrementCounterWithValue(){
3   try{
4     this.incrementCounterWithValue(this.incrementValue)
5   } catch (error) {
6     alert(error)
7   }
8 }
9 ...
```

É preciso estar atento ao bloco `try...catch`, porque somente métodos síncronos funcionarão. Métodos assíncronos como leitura de arquivos, ajax, local storage, entre outros, funcionam apenas com callbacks, que serão vistos a seguir.

7.10 Gerenciando métodos assíncronos

Um dos métodos assíncronos mais comuns que existe é a requisição ajax ao servidor, na qual o fluxo de código do javascript não aguarda a resposta do mesmo. Ou seja, o fluxo continua e é necessário lidar com esse tipo de comportamento dentro do action.

Para isso, precisa-se trabalhar com callbacks, e é isso que iremos fazer como exemplo na action `incrementCounter`.

Vamos adicionar 2 segundos a chamada do mutation, da seguinte forma:

```
1 export const incrementCounter = function ({ dispatch, state\
2   }) {
3   setTimeout(function(){
4     dispatch('INCREMENT')
5   },2000)
6 }
```

Agora, quando clica-se no botão +1, o valor somente é atualizado após dois segundos. Nesse meio tempo, precisamos informar ao usuário que algo está acontecendo, ou seja, precisamos adicionar uma mensagem “Carregando...” no componente, e precisamos também remover a mensagem quando o mutation é disparado.

7.11 Informando ao usuário sobre o estado da aplicação

Para que possamos exibir ao usuário que um método está sendo executado, adicione mais uma div no componente Increment, com o seguinte código:

```
1  <template>
2    <div>
3      <button @click="incrementCounter">+1</button>
4      <button @click="decrementCounter">-1</button>
5    </div>
6    <div>
7      <input type="text" v-model="incrementValue">
8      <button @click="tryIncrementCounterWithValue">increment\
9    </button>
10  </div>
11  <div v-show="waitMessage">
12    Aguarde...
13  </div>
14 </template>
```

A mensagem “Aguarde...” é exibida se `waitMessage` for `true`. Inicialmente, o valor desta variável é `false`, conforme o código a seguir:

```
1  ...
2  data () {
3    return{
4      incrementValue:0,
5      waitMessage:false
6    }
7  },
8  ...
```

Quando clicamos no botão para adicionar mais uma unidade, precisamos alterar o valor da variável `waitMessage` para `true`. Para fazer isso, o botão `+1` passa a chamar o método `tryIncrementCounter`, ao invés de chamar `incrementCounter` diretamente, veja:

```
1  <template>
2    ...
3    <button @click="tryIncrementCounter">+1</button>
4    ...
5  </template>
6  <script>
7    ...
8    methods: {
9      tryIncrementCounter(){
10        this.waitMessage=true;
11        this.incrementCounter();
12      },
13      tryIncrementCounterWithValue(){
14        ...
15      }
16    }
17    ...
18  </script>
```

Neste momento, quando o usuário clicar no botão +1, alteramos o valor da variável `waitMessage`, e com isso a palavra “Aguarde...” surge na página.

Após o mutation alterar o valor do `state.count`, precisamos esconder novamente a mensagem “Aguarde...”, alterando o valor da variável `waitMessage` para `false`. Isso é feito através de um callback, veja:

```
1 export const incrementCounter = function ({ dispatch, state\
2   },callback) {
3   setTimeout(function(){
4     dispatch('INCREMENT')
5     callback();
6   },2000)
7 }
8 ...
```

Agora a action `incrementCounter` tem um parâmetro chamado `callback`, que é chamada logo após o `setTimeout`, que está simulando um acesso ajax.

No template, basta usar o callback que será chamado pela action para novamente esconder a mensagem “Aguarde...”, da seguinte forma:

```
1 ...
2 tryIncrementCounter(){
3   let t = this;
4   this.waitMessage=true;
5   this.incrementCounter(function(){
6     t.waitMessage=false;
7   });
8 },
9 ...
```

No método `tryIncrementCounter` criamos a variável “t” que é uma referência ao “this”, pois ela será perdida no callback devido ao escopo. Se você já trabalhou com javascript e callbacks, conhece este processo. Quando chamamos o método ‘`incrementCounter`’, temos no primeiro parâmetro uma função, que usa a variável `t` para

alterar a variável `waitMessage` novamente para `false`, escondendo assim a mensagem.

Neste ponto, quando o usuário clicar no botão “+1”, a mensagem “Aguarde” surge e, quando o contador é alterado, a mensagem desaparece.

Até então está tudo certo, só que não... A forma como criamos esta funcionalidade está atrelada ao fato de termos que criar um callback, e manipularmos o escopo do fluxo de código para conseguir alterar uma simples variável. Esta é a forma até comum no Javascript, mas devemos estar atentos a esse tipo de “vício”. Para que possamos exibir uma mensagem ao usuário de uma forma mais correta, por que não usar o Flux novamente?

7.12 Usando o vuex para controlar a mensagem de resposta ao usuário

Para que possamos controlar a mensagem “Aguarde...” pelo vuex, precisamos executar os seguintes passos:

- Criar a variável no state que armazena o estado da visualização da mensagem
- Criar um mutation para exibir a mensagem, outro para esconder.
- Criar um getter que irá retornar o valor do estado da visualização da mensagem
- Alterar o componente para que a div que contém a mensagem “Aguarde...” possa observar o seu state.

Primeiro, alteramos o `store.js` incluindo a variável no state e as mutations:

```
1  import Vue from 'vue'
2  import Vuex from 'vuex'
3
4  Vue.use(Vuex)
5
6  const state = {
7    count: 0,
8    showWaitMessage: false
9  }
10
11 const mutations = {
12   INCREMENT(state){
13     state.count++;
14   },
15   DECREMENT(state){
16     state.count--;
17   },
18   INCREMENTVALUE(state,value){
19     state.count=state.count+value
20   },
21   SHOW_WAIT_MESSAGE(state){
22     state.showWaitMessage = true;
23   },
24   HIDE_WAIT_MESSAGE(state){
25     state.showWaitMessage = false;
26   }
27 }
28
29 export default new Vuex.Store({
30   state,
31   mutations
32 })
```


Criamos o getter que irá retornar o a variável `state.showWaitMessage`:

```
1 export function getCount (state) {  
2   return state.count  
3 }  
4  
5 export function getShowWaitMessage(state){  
6   return state.showWaitMessage;  
7 }
```

Altere a action para que possa disparar os mutations quando necessário, conforme o código a seguir:

```
1 export const incrementCounter = function ({ dispatch, state\  
2   }) {  
3   dispatch('SHOW_WAIT_MESSAGE')  
4   setTimeout(function(){  
5     dispatch('HIDE_WAIT_MESSAGE')  
6     dispatch('INCREMENT')  
7   },2000)  
8 }
```

A action `incrementCounter` dispara o mutation `SHOW_WAIT_MESSAGE` antes de simular a requisição ajax, que neste caso é feito pelo “`setTimeout`”. Após os 2 segundos, disparamos o mutation ‘`HIDE_WAIT_MESSAGE`’ e também o ‘`INCREMENT`’.

Para finalizar, voltamos ao componente `Increment` para referenciar `div` com o “Aguarde...”, da seguinte forma:

```
1 <template>
2   <div>
3     <button @click="incrementCounter">+1</button>
4     <button @click="decrementCounter">-1</button>
5   </div>
6   <div>
7     <input type="text" v-model="incrementValue">
8     <button @click="tryIncrementCounterWithValue">increment\
9 </button>
10  </div>
11  <div v-show="getShowWaitMessage">
12    Aguarde...
13  </div>
14 </template>
15
16 <script>
17   import { incrementCounter, decrementCounter, incrementCou\
18 nterWithValue } from './actions'
19   import { getShowWaitMessage } from './getters'
20
21   export default {
22     vuex: {
23       actions: {
24         incrementCounter,
25         decrementCounter,
26         incrementCounterWithValue
27       },
28       getters: {
29         getShowWaitMessage
30       }
31     },
32     data () {
33       return{
```

```
34         incrementValue:0
35     }
36 },
37     methods: {
38         tryIncrementCounterWithValue(){
39             try{
40                 this.incrementCounterWithValue(this.incrementValu\
41 e)
42             } catch (error) {
43                 alert(error)
44             }
45         }
46     }
47 }
48 </script>
```

Perceba que a alteração deixa o código mais limpo, com menos métodos no componente para manipular. Ainda existe uma outra melhoria que pode ser feita, e deixaremos como exercício para o usuário, que é trocar a div que possui o “Aguarde...” por um componente próprio. Deixaremos a resposta no [github](https://github.com)⁴.

7.13 Vuex modular

É possível trabalhar de forma modular no vue de forma que cada conceito seja fisicamente separado em arquivos. Supondo, por exemplo, que no seu sistema você tenha informações sobre o login do usuário, um conjunto de telas que trabalham com informações sobre usuários (tabela users), outro conjunto de telas que lidam com informações sobre produtos (tabela products) e mais um sobre fornecedores (suppliers).

Pode-se optar por um único store, um único action e getters, ou pode-se optar por separar estes conceitos em módulos.

Vamos criar um outro projeto para ilustrar esta situação:

⁴<https://github.com/danielschmitz/vue-codigos/commit/e9b57115e9075962859cd6965d25c980d1e61cd5>

```
1 vue init browserify-simple vuex-modules
2 cd vuex-modules
3 npm install
```

Para instalar as bibliotecas vue extras, usamos o npm:

```
1 npm i -S vuex vue-resource vue-route
```

Com tudo instalado, podemos dar início a estrutura da aplicação. Lembre-se que, o que será apresentado aqui não é a única forma que você deve usar o vuex, é apenas uma sugestão. Como estamos dividindo a aplicação em módulos, porque não criar um diretório chamado *modules* e inserir os três módulos iniciais nele:

Estrutura:

```
1 src
2 |- modules
3 |- login
4 |- user
5 |- product
6 |- supplier
7 |- App.vue
8 |- main.js
```

Após criar estes diretórios (login,user,product,supplier), crie em separado os seus respectivos “stores”. Só que, neste caso, cada store de cada módulo terá o nome “index.js”, veja:

Estrutura:

```
1  src
2  |- modules
3  |- login
4  |- index.js
5  |- user
6  |- index.js
7  |- product
8  |- index.js
9  |- supplier
10 |- index.js
11 |- App.vue
12 |- main.js
```

Agora, cada store de cada módulo conterá o seu state e o seu mutation, veja:

src/modules/login/index.js

```
1  const state = {
2    username : "",
3    email : "",
4    token : ""
5  }
6
7  const mutations = {
8
9  }
10
11 export default { state, mutations }
```

O módulo de login possui três propriedades no seu state. Os mutations serão criados na medida que for necessário.

src/modules/user/index.js

```
1  const state = {  
2    list : [],  
3    selected : {}  
4  }  
5  
6  const mutations = {  
7  
8  }  
9  
10 export default { state, mutations }
```

src/modules/product/index.js

```
1  const state = {  
2    list : [],  
3    selected : {}  
4  }  
5  
6  const mutations = {  
7  
8  }  
9  
10 export default { state, mutations }
```

src/modules/supplier/index.js

```
1  const state = {
2    list : [],
3    selected : {}
4  }
5
6  const mutations = {
7
8  }
9
10 export default { state, mutations }
```

Criamos os outros 3 stores, que a princípio são bem parecidos.

Com a estrutura modular pronta, podemos configurar o store principal que irá configurar todos estes módulos:

```
1  import Vue from 'vue'
2  import Vuex from 'vuex'
3
4  import login from './modules/login'
5  import user from './modules/user'
6  import product from './modules/product'
7  import supplier from './modules/supplier'
8
9  Vue.use(Vuex)
10
11 export default new Vuex.Store({
12   modules: {
13     login,
14     user,
15     product,
```

```
16     supplier
17   }
18 })
```

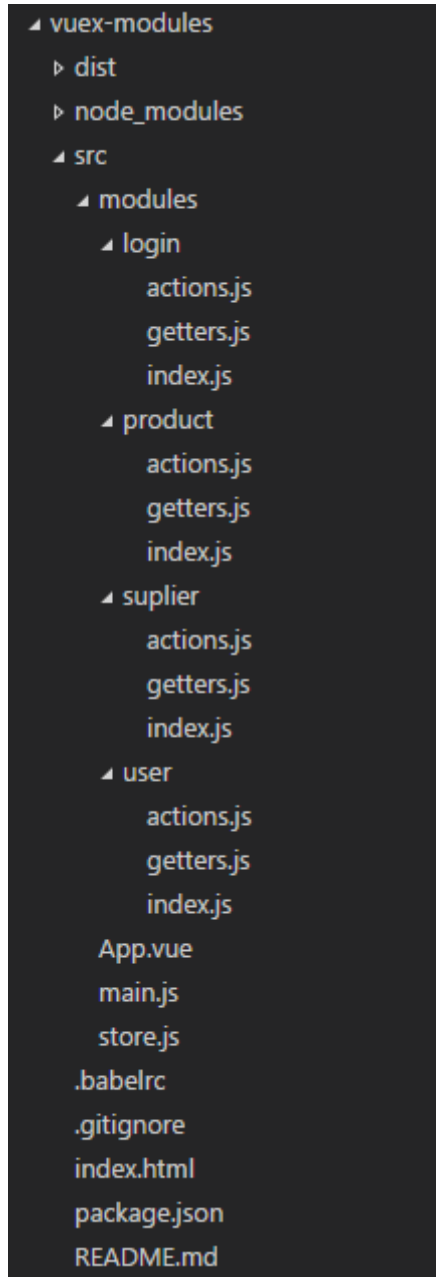
Com o store pronto, vamos referenciá-lo no componente principal, App.vue, veja:

src/App.vue

```
1 <template>
2   <div id="app">
3     <h1>{{ msg }}</h1>
4   </div>
5 </template>
6
7 <script>
8   import store from './store'
9
10  export default {
11    data () {
12      return {
13        msg: 'Hello Vue!'
14      }
15    },
16    store
17  }
18 </script>
19
20 <style>
21   body {
22     font-family: Helvetica, sans-serif;
23   }
24 </style>
```

Agora temos o componente App.vue com o seu store, e o store.js referenciando os quatro módulos criados. Para finalizar esta arquitetura, precisamos criar as actions e

os `getters` de cada módulo. Isso é feito criando os arquivos `actions.js` e `getters.js` em cada módulo, a princípio vazios, resultando em uma estrutura semelhante a figura a seguir:



Com a estrutura básica pronta, vamos iniciar o uso do vuex. Primeiro, vamos criar

um botão que irá simular o login/logout. O botão login irá realizar uma chamada ajax ao arquivo “login.json”, que tem o seguinte conteúdo:

login.json

```
1 {  
2   "username": "foo",  
3   "email": "foo@gmail.com",  
4   "token": "abigtext"  
5 }
```

Perceba que estamos apenas testando o uso dos módulos, então não há a necessidade de implementar qualquer funcionalidade no backend.

O botão “login” é adicionado no “App.vue”:

src/App.vue

```
1 <template>  
2   <div id="app">  
3     <h1>{{ msg }}</h1>  
4     <button @click="doLogin">Login</button>  
5   </div>  
6 </template>  
7  
8 <script>  
9   import store from './store'  
10  
11   import {doLogin} from './modules/login/actions'  
12  
13   export default {  
14     data () {  
15       return {  
16         msg: 'Hello Vue!'  
17       }  
18     }  
19   }  
20 }
```

```
18     },
19     vuex :{
20       actions :{
21         doLogin
22       }
23     },
24     store
25   }
26 </script>
```

O botão chama a action `doLogin`, que é uma action do modulo de login, veja:

```
1 export function doLogin({dispatch}){
2   this.$http.get("/login.json").then(
3     (response)=>{
4       dispatch("SETLOGIN",JSON.parse(response.data))
5     },
6     (error)=>{
7       console.error(error.statusText)
8     }
9   )
10 }
```

Como o action usa o `VueResource` para conexão com ajax, precisamos primeiro carregar esse plugin no arquivo `main.js`:

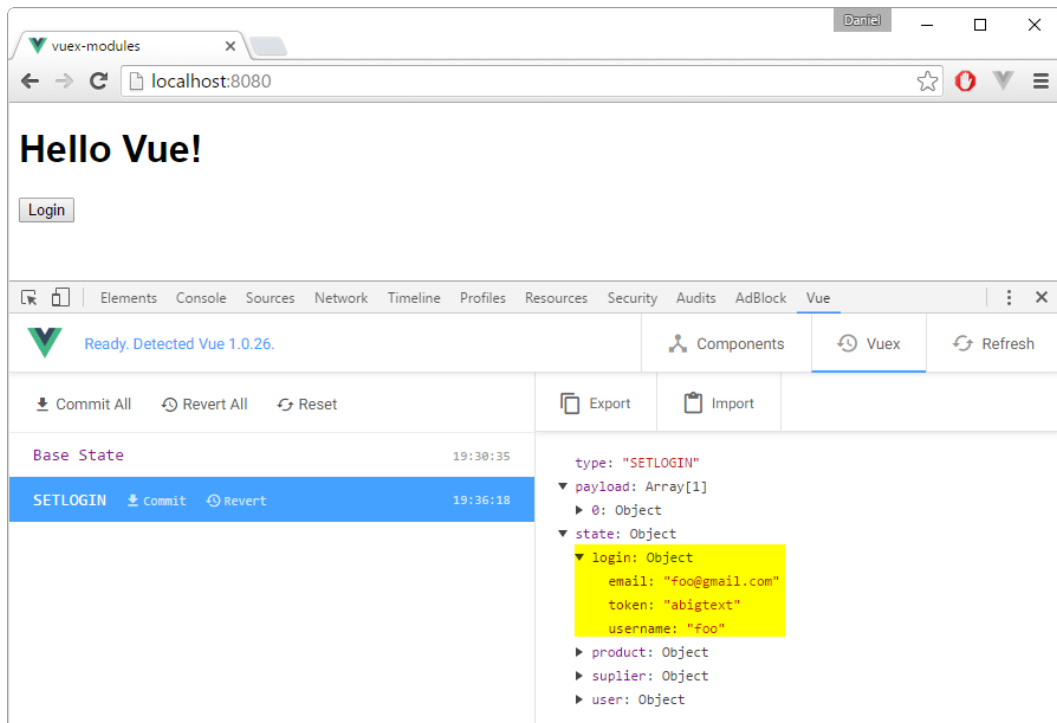
```
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 import VueResource from 'vue-resource'
5
6 Vue.use(VueResource)
7
8 new Vue({
9   el: 'body',
10   components: { App }
11 })
```

Perceba que, quando a requisição ajax é feita com sucesso, é disparado o mutation “SETLOGIN”, repassando como segundo parâmetro o `response.data`, que é justamente o texto que está no arquivo `login.json`. O mutation “SETLOGIN” é exibido a seguir:

`src/modules/login/index.js`

```
1 const state = {
2   username : "",
3   email : "",
4   token : ""
5 }
6
7 const mutations = {
8   SETLOGIN (state, data) {
9     console.log(data)
10    state.username = data.username;
11    state.email = data.email;
12    state.token = data.token;
13  }
14 }
15
16 export default { state, mutations }
```

Todos os passos para realizar este login foram realizados. Pode-se agora testar a aplicação e certificar que os dados do login foram atualizados corretamente no vuex, veja:



Com os dados sendo armazenados no state, podemos criar o getter “isLogged”, que irá controlar se o usuário está logado ou não. Usaremos esse getter para controlar os outros botões que iremos usar.

src/modules/login/getters.js

```
1 export function isLogged(state){
2   return state.login.token != ""
3 }
```

A forma como calculamos se o usuário está logado ou não envolve verificar se o token de autenticação é nulo ou não. Para que possamos efetuar um logout na aplicação, cria-se outra action, chamada de doLogout:

src/modules/login/actions.js

```
1 export function doLogin({dispatch}){
2   this.$http.get("/login.json").then(
3     (response)=>{
4       dispatch("SETLOGIN", JSON.parse(response.data))
5     },
6     (error)=>{
7       console.error(error.statusText)
8     }
9   )
10 }
11 export function doLogout({dispatch}){
12   let login = {
13     username : "",
14     email : "",
15     token : ""
16   }
17   dispatch("SETLOGIN", login)
18 }
```

Ao fazer o logout, chamamos o mutation “SETLOGIN” informando um login vazio.

Agora que temos o login/logout prontos, juntamente com o getter isLoggedIn, podemos usá-lo da seguinte forma:

src/App.vue

```
1 <template>
2   <div id="app">
3     <button v-show="!isLoggedIn" @click="doLogin">Login</butt\
4 on>
5     <div v-show="isLoggedIn">
6       <button @click="doLogout"> Logout</button>
7       <button > Users</button>
8       <button > Products</button>
9       <button > Suppliers</button>
10    </div>
11  </div>
12 </template>
13
14 <script>
15   import store from './store'
16   import {doLogin,doLogout} from './modules/login/actions'
17   import {isLoggedIn} from './modules/login/getters'
18
19   export default {
20     data () {
21       return {
22         msg: 'Hello Vue!'
23       }
24     },
25     vuex :{
26       actions :{
27         doLogin,doLogout
28       },
29       getters : {
30         isLoggedIn
31       }
```



```
32     },
33     store
34   }
35 </script>
```

O resultado da alteração no componente App.vue é que os botões Logout, Users, Products e Suppliers só aparecem quando o usuário está “logado”, ou seja, quando ele clica no botão login.

Continuando na demonstração do Vuex modular, vamos usar o botão “Products” para exibir na tela dados que podem ser obtidos do servidor. Neste caso, estamos trabalhando com o módulo products, então devemos criar:

- a action loadProducts
- a mutation SETPRODUCTS
- a mutation SETPRODUCT
- o getter getProducts
- o getter getProduct

A action loadProcuts irá ler o seguinte arquivo json:

products.json

```
1  [
2  {
3    "id": 1,
4    "name": "product1",
5    "quantity": 200
6  },
7  {
8    "id": 2,
9    "name": "product2",
10   "quantity": 100
11  },
```

```
12 {
13   "id": 3,
14   "name": "product3",
15   "quantity": 50
16 },
17 {
18   "id": 4,
19   "name": "product4",
20   "quantity": 10
21 },
22 {
23   "id": 5,
24   "name": "product5",
25   "quantity": 20
26 },
27 {
28   "id": 6,
29   "name": "product6",
30   "quantity": 300
31 }
32 ]
```

src/modules/product/actions.js

```
1 export function loadProducts({dispatch}){
2   this.$http.get("/products.json").then(
3     (response)=>{
4       dispatch("SETPRODUCTS", JSON.parse(response.data))
5     },
6     (error)=>{
7       console.error(error.statusText)
8     }
9   )
10 }
```

```
9    )  
10 }
```

Após ler o arquivo json, o mutation “SETPRODUCTS” será chamado, repassando o `response.data`. Este mutation é criado a seguir:

src/modules/product/index.js

```
1  const state = {  
2    list : [],  
3    selected : {}  
4  }  
5  
6  const mutations = {  
7    SETPRODUCTS(state, data) {  
8      state.list = data;  
9    },  
10 }  
11  
12 export default { state, mutations }
```

O mutation “SETPRODUCTS” irá preencher a variável `list`, que poderá ser acessada através de um getter, veja:

src/modules/product/getters.js

```
1  export function getProducts(state){  
2    return state.product.list;  
3  }  
4  
5  export function hasProducts(state){  
6    return state.product.list.length>0  
7  }
```

Neste getter, criamos dois métodos. O primeiro, retorna a lista de produtos. O segundo irá retornar se a lista de produtos possui itens. Iremos usá-la no componente App.vue, da seguinte forma:

src/App.vue

```
1 <template>
2   <div id="app">
3     <button v-show="!isLoggedIn" @click="doLogin">Login</butt\
4 on>
5     <div v-show="isLoggedIn">
6       <button @click="doLogout"> Logout</button>
7       <button > Users</button>
8       <button @click="loadProducts">Products</button>
9       <button > Suppliers</button>
10
11     <br>
12
13     <div v-show="hasProducts">
14       <h4>Products</h4>
15       <table border="1">
16         <thead>
17           <tr>
18             <td>id</td>
19             <td>name</td>
20             <td>quantity</td>
21           </tr>
22         </thead>
23         <tbody>
24           <tr v-for="p in getProducts">
25             <td>{{p.id}}</td>
26             <td>{{p.name}}</td>
27             <td>{{p.quantity}}</td>
28           </tr>
```

```
29         </tbody>
30     </table>
31 </div>
32
33 </div>
34 </div>
35 </template>
36
37 <script>
38     import store from './store'
39     import {doLogin,doLogout} from './modules/login/actions'
40     import {loadProducts} from './modules/product/actions'
41     import {isLoggedIn} from './modules/login/getters'
42     import {hasProducts,getProducts} from './modules/product/\
43 getters'
44
45     export default {
46         data () {
47             return {
48                 msg: 'Hello Vue!'
49             }
50         },
51         vuex :{
52             actions :{
53                 doLogin,doLogout,
54                 loadProducts
55             },
56             getters : {
57                 isLoggedIn,
58                 hasProducts,getProducts
59             }
60         },
61         store
```

```
62    }
63  </script>
64
65  <style>
66    body {
67      font-family: Helvetica, sans-serif;
68    }
69  </style>
```

Perceba que o botão para carregar produtos chama a action `loadProducts`, que faz o ajax e chama o mutation `SETPRODUCTS`, que preenche a lista do seu state. Os getters são atualizados e tanto `getProducts` quanto `hasProducts` mudam. Com o design reativo, a div que possui a diretiva `v-show="hasProducts"` irá aparecer, juntamente com a tabela criada através da diretiva `v-for`, que usa o `getProducts` para iterar entre os elementos e exibir os produtos na tela.

Com isso exibimos um ciclo completo entre as fases do vuex modular. É válido lembrar que a estrutura de diretórios é apenas uma sugestão, e você pode mudá-la da forma que achar válido. O uso do vuex modular também é uma sugestão de design de projeto, nada impede que você use somente um store e que armazene todos os states, actions e getters em somente um lugar.

Deixamos como tarefa para o leitor:

- Criar um botão para remover todos os produtos do state
- Carregar todos os usuários
- Carregar todos os suppliers

8. Mixins

No Vue, os mixins são objetos que contém propriedade e métodos que podem ser “anexadas” a qualquer componente vue da sua aplicação. Quando isso acontece, o mixin passa a se tornar parte do componente. Pode-se usar mixins para reaproveitar funcionalidades na sua aplicação, dependendo do que deseja-se fazer.

Vamos supor que você deseja acompanhar o fluxo de eventos criados por cada Componente do Vue. Suponho que você tenha 3 componentes, você teria que usar os eventos *create*, *beforeComplete*, *compiled*, *ready*, *beforeDestroy* e *destroyed* em cada componente. Mas com o mixin você pode implementar uma vez e “anexá-los” no componente que deseja.

8.1 Criando mixins

Vamos criar o projeto “vue-mixin” com o seguinte comando:

```
1 vue init browserify-simple#1 vue-mixin
2 cd vue-mixin
3 npm install
```

O mixin que queremos usar é exibido a seguir, onde o evento *created* é capturado e usamos o *console.log* para recuperar o nome do componente e exibir uma mensagem.

src/eventMixin.js

```
1 export const eventMixin = {  
2   ``js  
3   created: function(){  
4     console.log(`[${this.$options.name}] created`);  
5   }  
6 }
```

Crie 3 componentes chamados de Comp1, Comp2 e Comp3, todos eles com o mixin já configurado:

src/Comp1.vue

```
1 <template>Comp 1</template>  
2 <script>  
3 import {eventMixin} from './eventMixin'  
4  
5 export default{  
6   mixins: [eventMixin]  
7 }  
8 </script>
```

src/Comp2.vue

```
1 <template>Comp 2</template>  
2 <script>  
3 import {eventMixin} from './eventMixin'  
4  
5 export default{  
6   mixins: [eventMixin]  
7 }  
8 </script>
```

src/Comp3.vue

```
1 <template>Comp 3</template>
2 <script>
3 import {eventMixin} from './eventMixin'
4
5 export default{
6   mixins: [eventMixin]
7 }
8 </script>
```

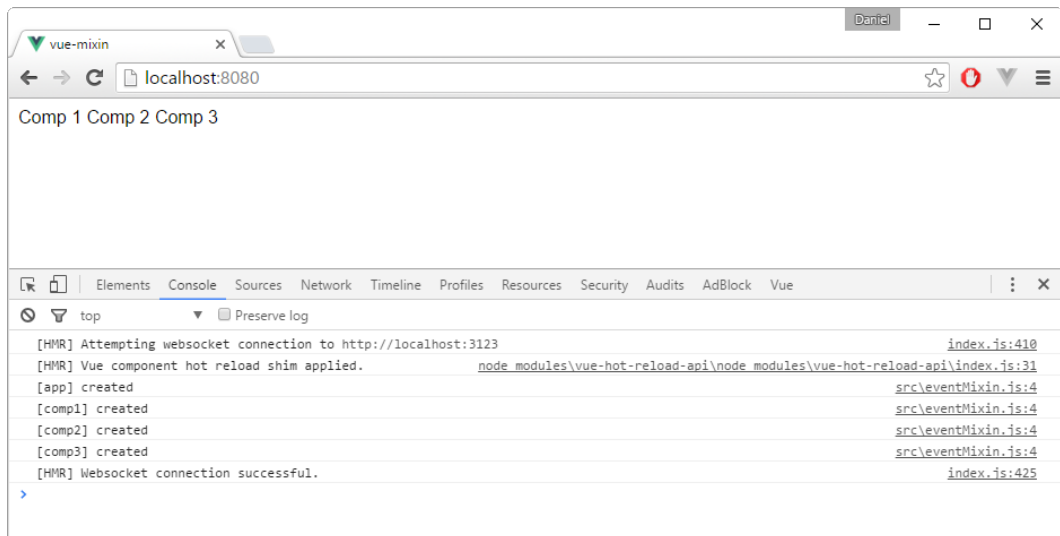
Para finalizar, adicione os três componentes no App.vue, da seguinte forma:

src/App.vue

```
1 <template>
2   <div id="app">
3     <comp1></comp1>
4     <comp2></comp2>
5     <comp3></comp3>
6   </div>
7 </template>
8
9 <script>
10 import Comp1 from './Comp1.vue'
11 import Comp2 from './Comp2.vue'
12 import Comp3 from './Comp3.vue'
13 import {eventMixin} from './eventMixin'
14
15 export default {
16   components :{
17     Comp1,Comp2,Comp3
18   },
19   data () {
```

```
20     return {
21       msg: 'Hello Vue!'
22     }
23   },
24   mixins: [eventMixin]
25 }
26 </script>
```

Perceba que adicionamos os três componentes na aplicação, e também adicionamos o mixin no App.vue. Ao executar a aplicação através do comando `npm run dev`, temos a resposta semelhante a figura a seguir:



Com isso podemos usar os mixins para adicionar funcionalidades extras aos componentes, sem a necessidade de implementar herança, ou então chamar métodos globais. Veja que os mixins podem se anexar a qualquer propriedade do Vue, seja ela `data`, `methods`, `filters`, entre outras.

8.2 Conflito

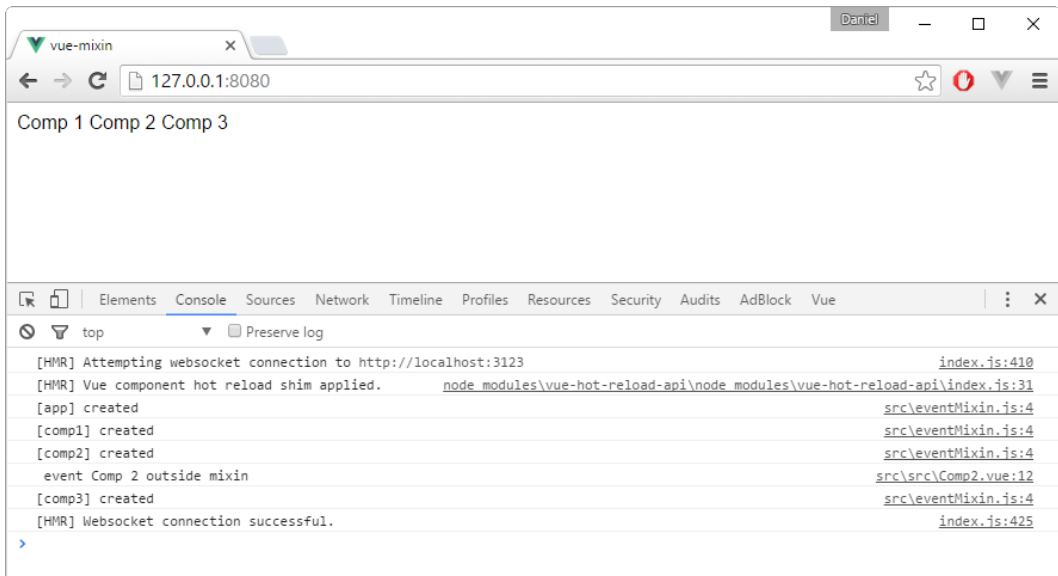
Sabemos que os mixins são anexados ao eventos, métodos e propriedades do componente Vue. Quando dois eventos ou propriedades entram em conflito, o vue realiza um merge destas propriedades. Por exemplo, suponha que o seu Comp2 possua agora o seguinte código:

src/Comp2.vue

```
1 <template>Comp 2</template>
2 <script>
3 import {eventMixin} from './eventMixin'
4
5 export default{
6   mixins: [eventMixin],
7   created : function() {
8     console.log(" event Comp 2 outside mixin ")
9   }
10 }
11 </script>
```

Perceba que tanto o mixin, quanto o componente, possuem o evento *created*. Neste caso, o evento do mixin será executado em primeiro lugar, e o evento do componente em segundo.

A resposta para o código acima é exibido a seguir:



Para métodos, componentes e diretivas, se houver alguma propriedade em conflito, a propriedade do componente será usada e a propriedade do mixin será descartada.

9. Plugins

Um plugin tem como principal finalidade adicionar funcionalidades globais ao seu Vue. Pense em um plugin como algo que várias pessoas possam usar, não somente no seu projeto. Nós já vimos diversos plugin ao longo dessa obra, como o *vue-router*, *vuex*, *vue-resource* etc.

Em tese, um plugin pode adicionar as seguintes funcionalidades:

- Adicionar propriedades e métodos globais
- Adicionar diretivas, filtros e transições
- Adicionar propriedades à instância Vue
- Adicionar uma API com vários métodos e propriedades que usam os três itens anteriores

9.1 Criando um plugin

Um plugin Vue deve implementar o seu método *install* e, nele, configurar suas funcionalidades. O exemplo básico de um plugin é exibido a seguir:

```
1 MyPlugin.install = function (Vue, options) {  
2   // 1. add global method or property  
3   Vue.myGlobalMethod = ...  
4   // 2. add a global asset  
5   Vue.directive('my-directive', {})  
6   // 3. add an instance method  
7   Vue.prototype.$myMethod = ...  
8 }
```

Para usar um plugin, é necessário importá-lo e usar o comando `Vue.use`.

Para exemplificar este processo, vamos criar um simples plugin chamado “real”, que converte um inteiro para reais.

```
1 vue init browserify-simple#1 plugin-real
2 cd plugin-real
3 npm init
```



Não é preciso criar um projeto completo com vue-cli para desenvolver um plugin. O ideal é que o plugin seja “puro” com o seu arquivo javascript e o arquivo package.json para que possa ser adicionado ao gerenciador de pacotes no futuro. Um exemplo simples de plugin pode ser encontrado no [vue-moment](https://github.com/brockpetrie/vue-moment)¹

Após criar o projeto, vamos criar o plugin adicionando o arquivo ‘real.js’ na raiz do mesmo:

```
1 module.exports = {
2   install: function (Vue, options) {
3
4     Vue.filter('real', function() {
5       var tmp = arguments[0]+'';
6       tmp = tmp.replace(/([0-9]{2})$/g, ",$1");
7       if( tmp.length > 6 )
8         tmp = tmp.replace(/([0-9]{3}),([0-9]{2})$/g, ".$1,$2");
9       return 'R$ ' + tmp;
10    })
11  }
12 }
```

Neste código, usamos o `module.exports` para que o plugin possa ser importado pelos componentes do projeto, no caso com `require`. O método `install` é o único método obrigatório que deve ser implementado para o Vue o tratar como um plugin.

No método `install`, criamos um filtro chamado “real” e usamos um pouco de lógica ([retirado deste link](#)²) para formatar um número inteiro para o formato de moeda em reais.

¹<https://github.com/brockpetrie/vue-moment>

²<http://wbruno.com.br/expressao-regular/formatar-em-moeda-reais-expressao-regular-em-javascript/>

Neste método, usamos `arguments[0]` para obter o primeiro argumento do filter, que é o o seu valor. Por exemplo, se usarmos `123 | filter, arguments[0]` assume o valor “123”.

No final do filtro, retornamos o valor de `tmp` que é o valor convertido com pontos e vírgulas, devidamente corrigido.

Com o plugin pronto, podemos instalar o plugin no arquivo `main.js`, onde a instância `Vue` é criada. Para isso, faça o `require` do plugin e o comando `Vue.use`, veja:

src/main.js

```
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 import real from '../real'
5 Vue.use(real)
6
7 new Vue({
8   el: 'body',
9   components: { App }
10 })
```

Com o plugin carregado, podemos usá-lo normalmente em qualquer parte da aplicação, como foi feito no `App.vue`:

src/main.js

```
1 <template>
2   <div id="app">
3     <h1>Plugin</h1>
4     <p>{{ valor | real }}
5   </div>
6
7 </template>
8
```

```
9  <script>
10  export default {
11    data () {
12      return {
13        valor: 123456
14      }
15    }
16  }
17  </script>
```

Perceba que usamos o filtro “real” juntamente com a variável `valor`, o que produz o seguinte resultado:

