

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 2016-10-13



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 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
2.6.1 Utilizando v-bind:key	22
2.6.2 Uso do set	23
2.6.3 Como remover um item	23
2.6.4 Loops em objetos	23
2.7 Eventos e métodos	24
2.7.1 Modificando a propagação do evento	25
2.7.2 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
2.11.1 Databind único	34

CONTEÚDO

2.11.2	Databind com html	35
2.11.3	Databind em Atributos	35
2.11.4	Expressões	35
2.12	Filtros	36
2.12.1	uppercase	36
2.12.2	lowercase	36
2.12.3	currency	36
2.12.4	pluralize	37
2.12.5	json	37
2.13	Diretivas	37
2.13.1	Argumentos	37
2.13.2	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
2.19.1	Checkbox	42
2.19.2	Radio	43
2.19.3	Select	43
2.19.4	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
3.8.1	camelCase vs. kebab-case	57
3.8.2	Validações e valor padrão	57
3.9	Slots e composição de componentes	59

CONTEÚDO

3.10	Eventos e comunicação entre componentes	60
3.10.1	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.map	76
4.4	Configurando o router-view	77
4.5	Criando novos componentes	79
4.6	Criando um menu	81
4.6.1	Repassando parâmetros no link	82
4.7	Classe ativa	83
4.8	Filtrando rotas pelo login	85
5.	Vue Resource	87
5.1	Testando o acesso Ajax	88
5.2	Métodos e opções de envio	92
5.3	Trabalhando com resources	93

Parte 2 - Criando um blog com Vue, Express e MongoDB 96

6.	Express e MongoDB	97
6.1	Criando o servidor RESTful	97
6.2	O banco de dados MongoDB	97
6.3	Criando o projeto	102
6.4	Estrutura do projeto	103
6.5	Configurando os modelos do MongoDB	103
6.6	Configurando o servidor Express	105
6.7	Testando o servidor	115

6.8	Testando a api sem o Vue	116
7.	Implementando o Blog com Vue	122
7.1	Reconfigurando o packages.json	122
7.2	Instalando pacotes do vue e materialize	122
7.3	Configurando o router e resource	123
7.4	Configurando a interface inicial da aplicação	126
7.5	Obtendo posts	130
7.6	Logout	146
7.7	Refatorando a home	147
7.8	Conclusão	150

Parte 3 – Conceitos avançados 151

8.	Vuex e Flux	152
8.1	O que é Flux?	152
8.2	Conhecendo os problemas	152
8.3	Quando usar?	153
8.4	Conceitos iniciais	153
8.5	Exemplo simples	154
8.5.1	Criando o projeto	155
8.5.2	Criando componentes	155
8.5.3	Incluindo Vuex	157
8.5.4	Criando uma variável no state	159
8.5.5	Criando mutations	160
8.5.6	Criando actions	161
8.5.7	Criando getters	162
8.5.8	Alterando o componente Display para exibir o valor do contador	162
8.5.9	Alterando o componenet Increment	163
8.5.10	Testando a aplicação	164
8.6	Revedo o fluxo	164
8.7	Chrome vue-devtools	165
8.8	Repassando dados pelo vuex	169
8.9	Tratando erros	172

CONTEÚDO

8.10	Gerenciando métodos assíncronos	173
8.11	Informando ao usuário sobre o estado da aplicação	174
8.12	Usando o vuex para controlar a mensagem de resposta ao usuário	177
8.13	Vuex modular	181
9.	Mixins	200
9.1	Criando mixins	200
9.2	Conflito	204
10.	Plugins	206
10.1	Criando um plugin	206

Parte 4 - Criando um sistema com Vue e Php 210

11.	Preparando o ambiente	211
11.1	Preparando o servidor web	211
11.2	Apache no Linux	211
11.3	Instalando o PHP no Linux	213
11.4	Instalando o MySql no Linux	214
11.5	MySql Workbench para linux	215
11.6	Instalando o Composer no linux	216
11.7	Testando o Slim no Linux	217
11.8	Instalando o Apache/php/MySql no Windows	219
11.9	Instalando o MySql Workbench no Windows	221
11.10	Instalando o Composer no Windows	221
11.11	Testando o Slim no Windows	223
11.11.1	Criando o domínio virtual (virtual host)	226
12.	Banco de dados	229
12.1	Importando o banco de dados	229
12.2	Conhecendo as tabelas	230
13.	Criando o servidor PHP/Slim	231
13.1	Criando o diretório do servidor	231
13.2	Crie o arquivo .htaccess	232

CONTEÚDO

13.3	Configurando o acesso ao banco de dados	233
13.4	Configurando o CORS	236
13.5	Login (login.php)	237
13.6	Incluindo o token JWT de autenticação (login)	245
13.7	Verificando o login	248
13.8	Salvando categorias	251
13.9	Exibindo categorias (com paginação e busca)	253
13.10	Removendo uma categoria	256
13.11	Cadastro de fornecedores	257
13.12	Cadastro de Produtos	261
14.	Criando o sistema Sales no cliente	268
14.1	Configurando o bootstrap	269
14.2	Configurações iniciais	270
14.2.1	Validação de formulários (Vue Validator)	270
14.2.2	Configuração do Vue Resource	270
14.2.3	Arquivo de configuração	271
14.2.4	Controle <loading>	271
14.2.5	Controle <error>	272
14.3	Criando a estrutura do Vuex	273
14.4	Criando a tela de login	277
14.5	A action setLogin	284
14.6	O componente Admin.vue	285
14.7	Reprogramando o App.vue	286
14.8	Persistindo o login	288
14.9	Tela de Admin	290
14.10	Menu da tela Admin	290
14.11	Criando a tela de Categorias	297
14.11.1	Autenticação	301
14.11.2	Template	302
14.12	Paginação de categorias (e busca)	306
14.13	Removendo uma categoria	313
14.14	Validação	315
14.15	Algumas considerações sobre a tela de categorias	316
14.16	Cadastro de Fornecedores	316
14.17	Alterando o actions.js, adicionando o showLogin e hideLoding	317

CONTEÚDO

14.18	Alterando o getter.js	317
14.19	Incluindo o Suppliers.vue	317
14.20	Incluindo a tela de cadastro de produtos	324
14.20.1	Propriedade data	337
14.20.2	Método created()	338
14.20.3	Método newProduct()	339
14.20.4	Método saveProduct()	339

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 múltimos slots em um componente
- Removido o uso de \$dispatch e \$broadcast

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:

```
sudo apt-get install git node npm
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:

```
sudo curl -sL https://deb.nodesource.com/setup_6.x | sudo -\
E bash -
sudo apt-get install -y nodejs
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/>

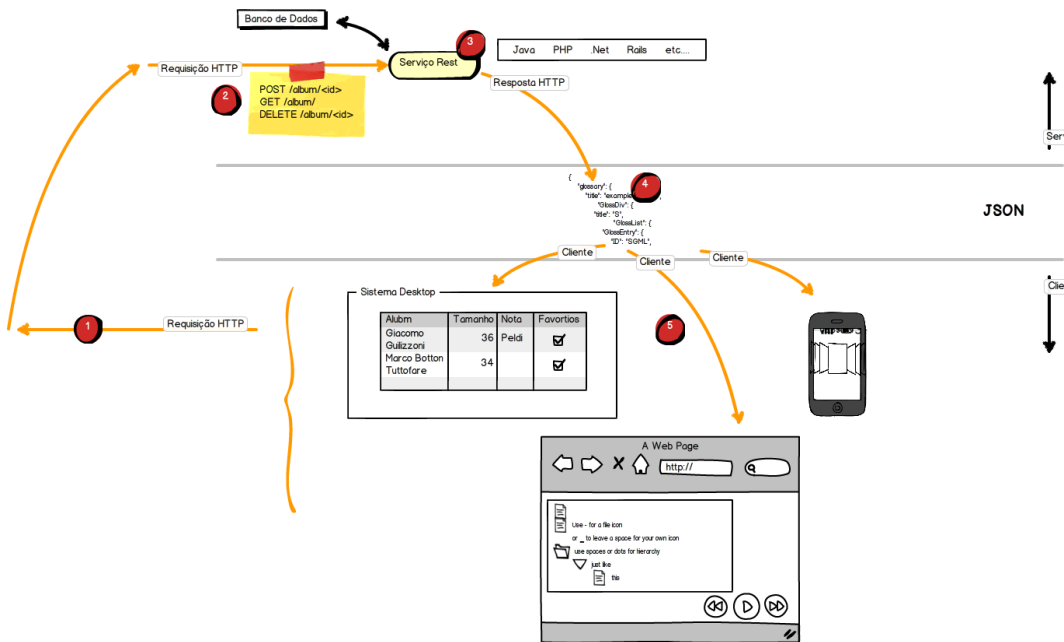
npm i -g

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

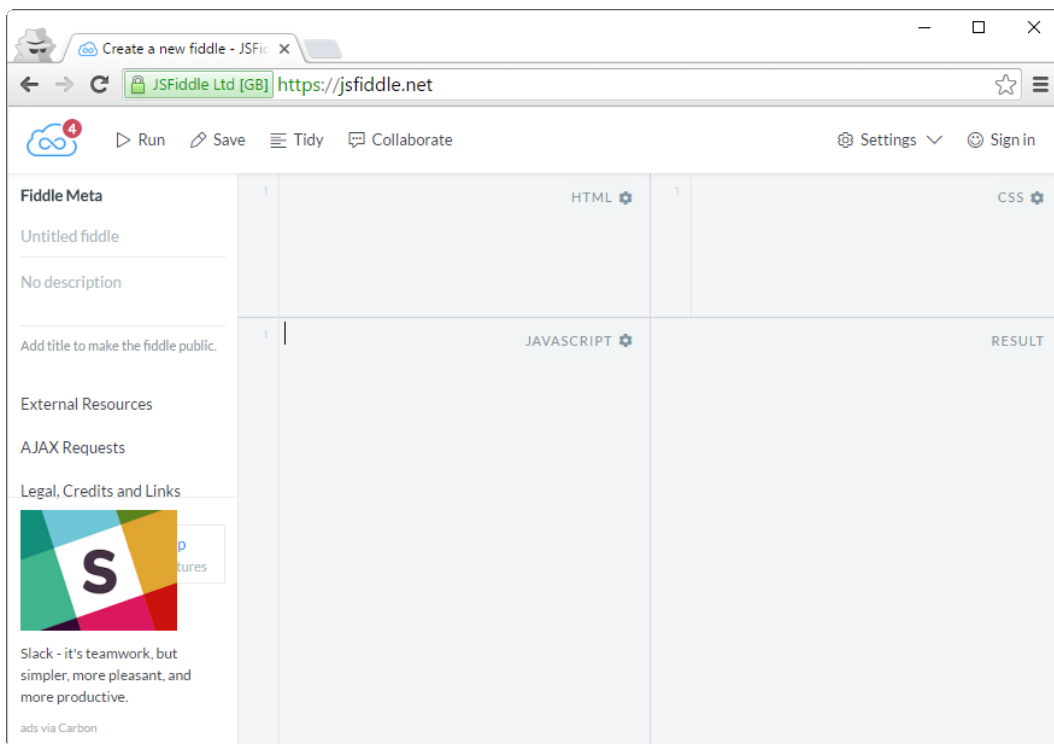
Neste capítulo iremos aprender alguns conceitos básicos sobre o Vue. Não veremos (ainda) a instalação do mesmo, porque como estamos apresentando cada conceito em separado, é melhor usarmos um editor online, neste caso o jsFiddle.

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:

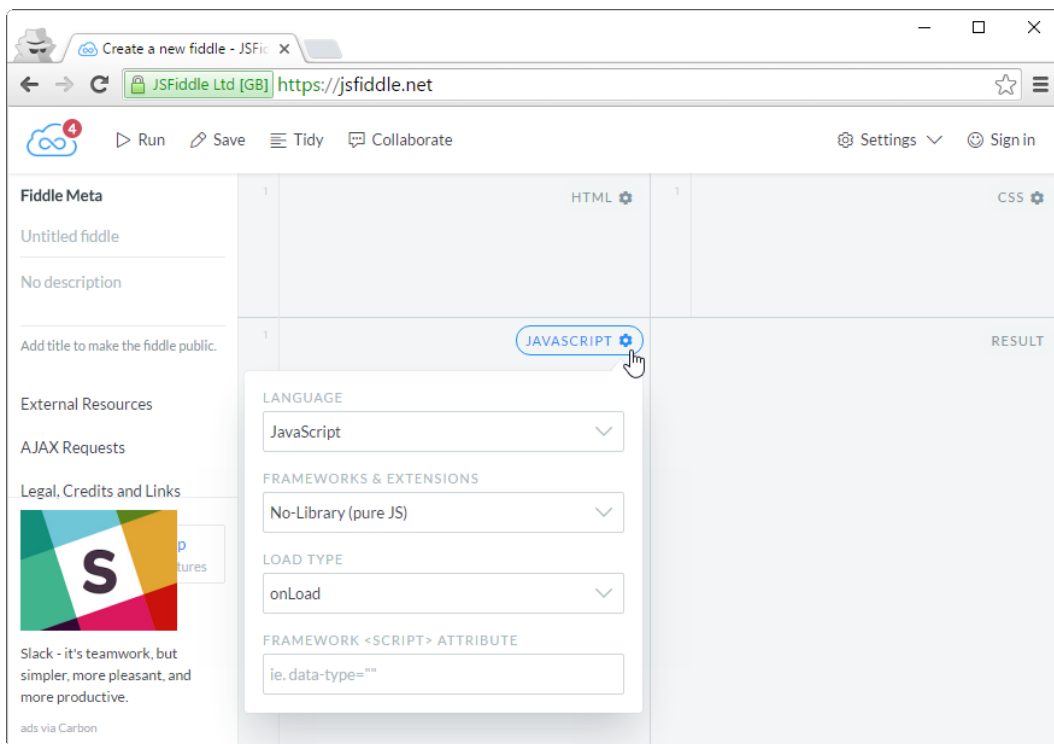
¹jsfiddle.net



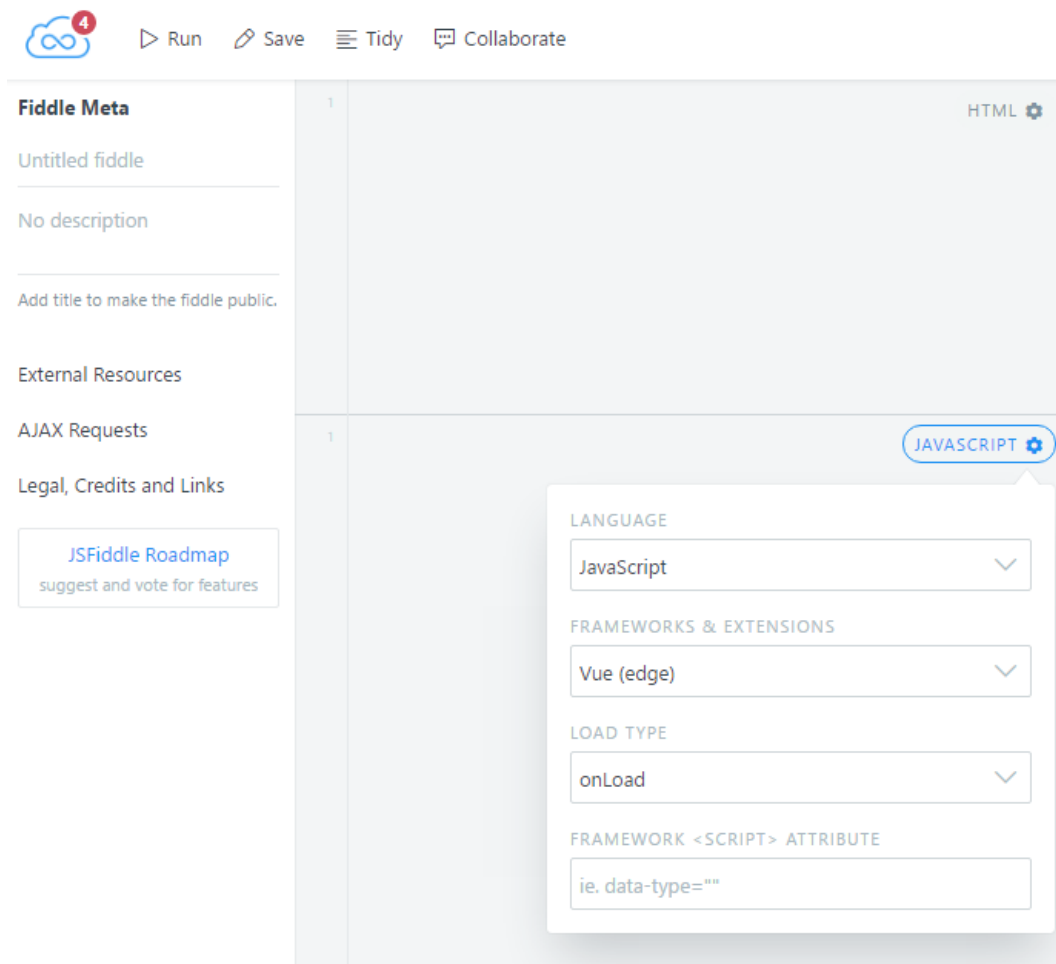
Caso queira, pode criar uma conta e salvar todos os códigos que criar, para poder consultar no futuro. No jsFiddle, temos 4 áreas sendo elas: *html*, *javascript*, *css* e *result*. Quando clicamos no botão Run, o html/javascript/css são combinados e o resultado é apresentado.

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:

```
<div id="app">
  {{msg}}
</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`.

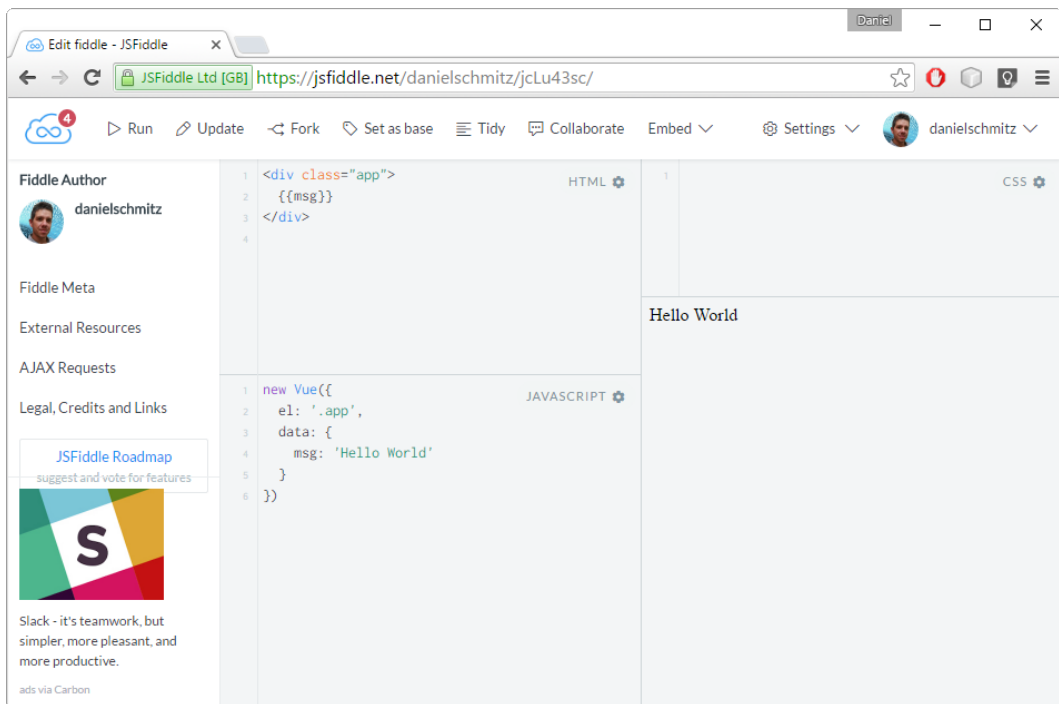
Agora vamos à parte Javascript. Para trabalhar com `vue`, basta criar um objeto `Vue` e repassar alguns parâmetros, veja:

```
new Vue({
  el: '#app',
  data: {
    msg: 'Hello World'
  }
})
```

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.

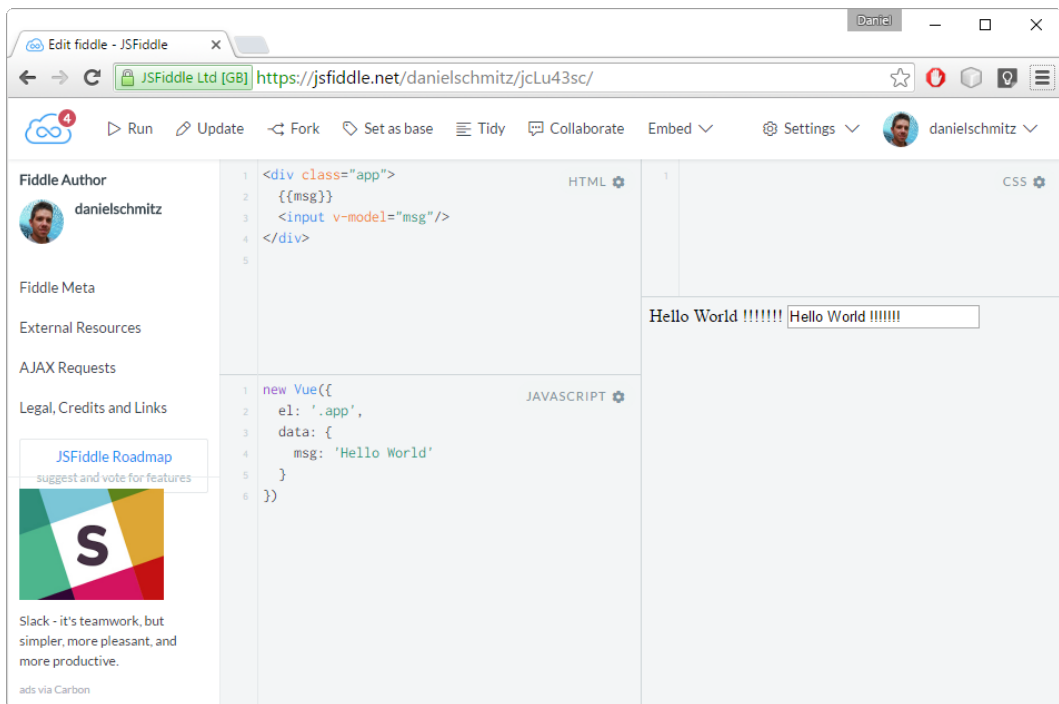


2.4 Two way databind

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`.

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

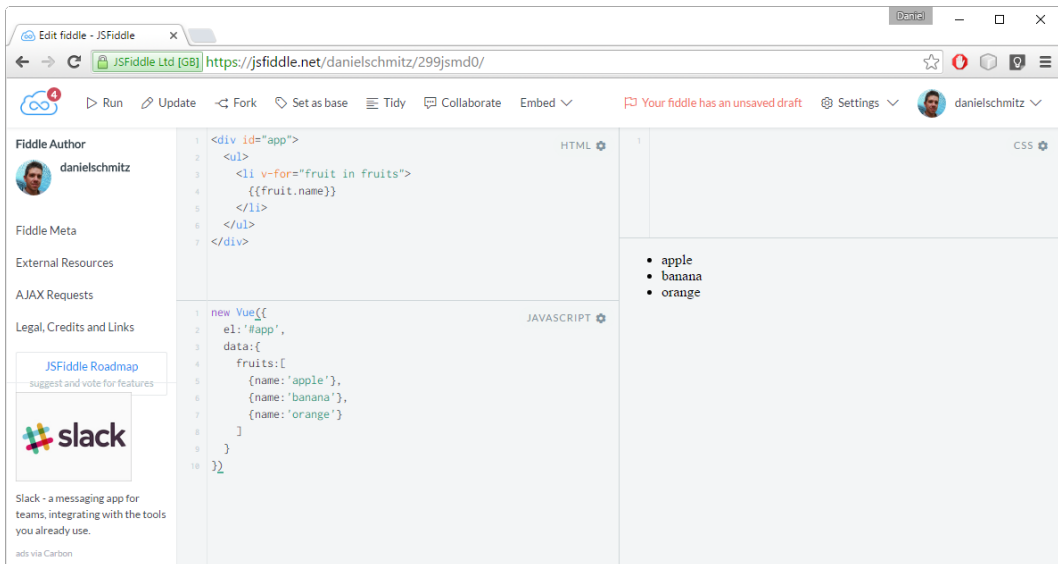
Existem dezenas de comandos do vue que iremos aprender ao longo desta obra. Um deles é o `v-for` que faz um loop no elemento em que foi inserido. Crie um novo jsFiddle com o seguinte código:

```
<div id="app">
  <ul>
    <li v-for="fruit in fruits">
      {{fruit.name}}
    </li>
  </ul>
</div>
```

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

```
new Vue({  
  el: '#app',  
  data: {  
    fruits: [  
      {name: 'apple'},  
      {name: 'banana'},  
      {name: 'orange'}  
    ]  
  }  
})
```

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.

2.6.1 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:

```
{
  items: [
    { _uid: '88f869d', ... },
    { _uid: '7496c10', ... }
  ]
}
```

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

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

</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.

2.6.2 Uso do set

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

```
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:

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

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

2.6.3 Como remover um item

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

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

2.6.4 Loops em objetos

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

```
<ul id="app" >
  <li v-for="(value, key) in object">
    {{ key }} : {{ value }}
  </li>
</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.

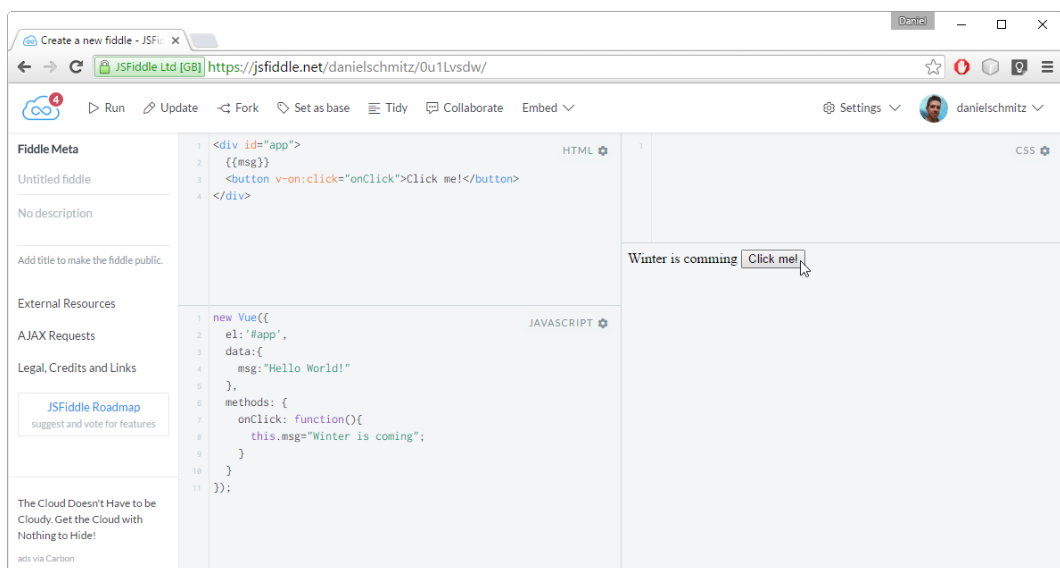
```
<div id="app">
  {{msg}}
  <button v-on:click="onClick">Click me!</button>
</div>
```

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:

```
new Vue({
  el: '#app',
  data: {
    msg: "Hello World!"
  },
  methods: {
    onClick: function(){
      this.msg="Winter is coming";
    }
  }
})
```

```
    }  
  }  
});
```

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:



2.7.1 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.

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

e:

```
methods: {
  say: function (msg, event) {
    event.preventDefault()
    alert(msg)
  }
}
```

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

```
<!-- a propagação do evento Click será cancelada -->
<a v-on:click.stop="doThis"></a>

<!-- o evento de submit não irá mais recarregar a página -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- os modificadores podem ser encadeados -->
<a v-on:click.stop.prevent="doThat"></a>
```

2.7.2 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:

```
<input v-on:keyup.enter="submit">
```

ou

```
<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:

```
<div id="app" class="container">

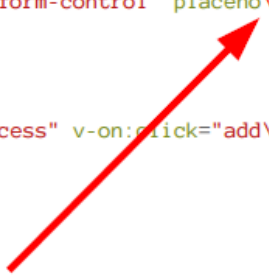
  <div class="row">
    <div class="col-xs-8">
      <input type="text" class="form-control" placeholder="\
Add a task" v-model="inputTask">
    </div>
    <div class="col-xs-4">
      <button class="btn btn-success" v-on:click="addTask">
        Adicionar
      </button>
    </div>
  </div>
  <br/>
  <div class="row">
    <div class="col-xs-10">
      <table class="table">
        <thead>
          <tr>
            <th>Task Name</th>
            <th></th>
          </tr>
        </thead>
        <tbody>
          <tr v-for="task in tasks">
            <td class="col-xs-11">
              {{task.name}}
            </td>
            <td class="col-xs-1">
              <button class="btn btn-danger" v-on:click="re\
moveTask(task.id)">
                x
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

```
        </td>
      </tr>
    </tbody>
  </table>
</div>
</div>
</div>
```

Quebra de linha no código fonte

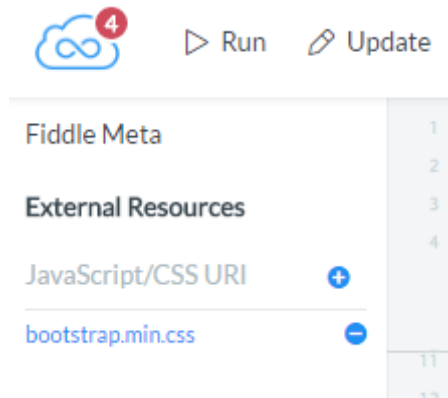
Cuidado com a quebra de página nos códigos desta obra. Como podemos ver na imagem a seguir:

```
<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:


```
new Vue({
  el: '#app',
  data: {
    tasks: [
      {id:1,name:"Learn Vue"},
      {id:2,name:"Learn Npm"},
      {id:3,name:"Learn Sass"}
    ],
    inputTask: ""
  },
  methods: {
    addTask(){
      if (this.inputTask.trim()!=""){
        this.tasks.push(
          {name:this.inputTask,
            id:this.tasks.length+1}
          )
        this.inputTask="";
      }
    },
    removeTask(id){
      for(var i = this.tasks.length; i--;) {
        if(this.tasks[i].id === id) {
          this.tasks.splice(i, 1);
        }
      }
    }
  }
})
```

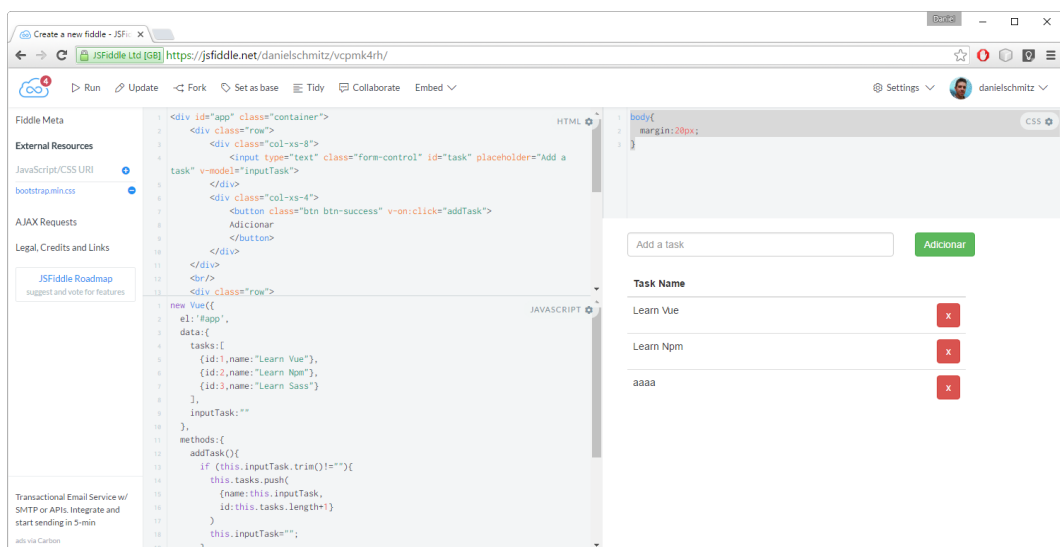
O código Vue, apesar de extenso, é fácil de entender. Primeiro criamos as duas variáveis: `tasks` possui um array de objetos que será a base da tabela que foi criada no html. Já a variável `inputTask` realiza um two way databind com a caixa de texto

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.

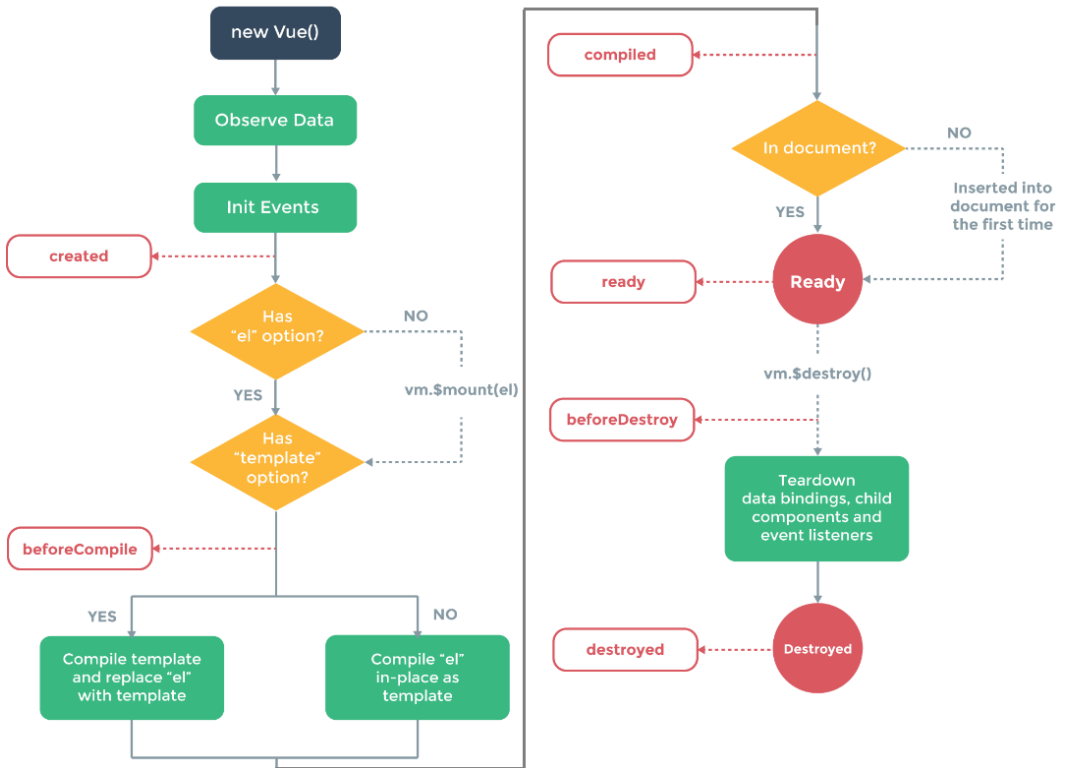
O resultado deste código pode ser visto na figura a seguir. Perceba que o formulário e a lista de tarefas possui o estilo do bootstrap.



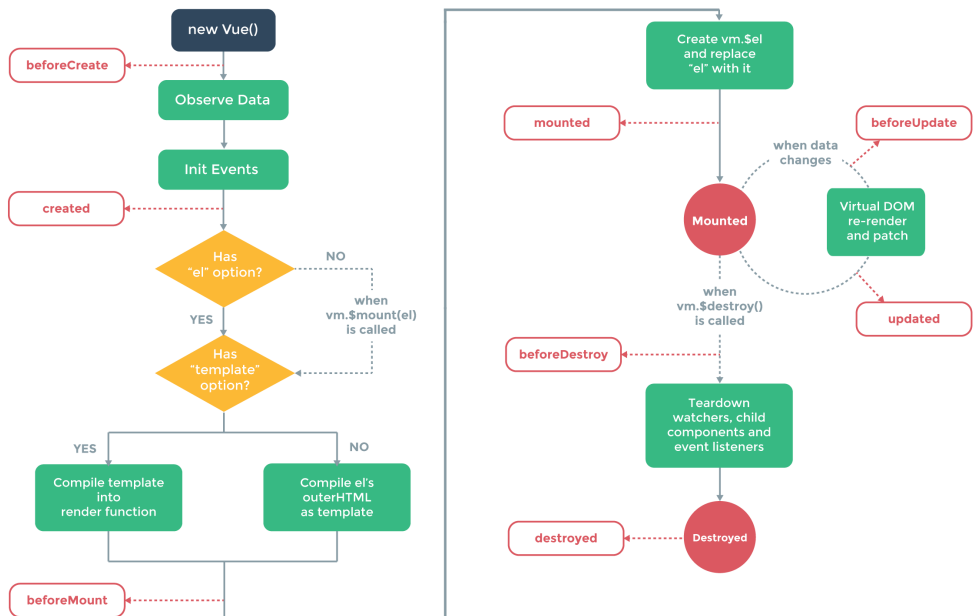
Caso queira ter acesso direto a este código pelo jsFiddle, (clique aqui)[<https://jsfiddle.net/tbg8fo51>]

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.

2.11.1 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:

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

2.11.2 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.

2.11.3 Databind em Atributos

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

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

2.11.4 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.

```
<div>
  {{ msg | capitalize }}
</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:

2.12.1 uppercase

Converte todas as letras para maiúscula

2.12.2 lowercase

Converte todas as letras para minúscula

2.12.3 currency

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

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

2.12.4 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”.

2.12.5 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:

```
<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.

2.13.1 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>

```
<a v-bind:href="url"></a>
```

é o mesmo que:

```
<a :href="url"></a>
```

ou então:

```
<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`.

2.13.2 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:

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

Podemos usar o atalho:


```
<input :type="form.type"
:placeholder="form.placeholder"
: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:

```
<!-- full syntax -->
<a v-on:click="doSomething"></a>

<!-- shorthand -->
<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:

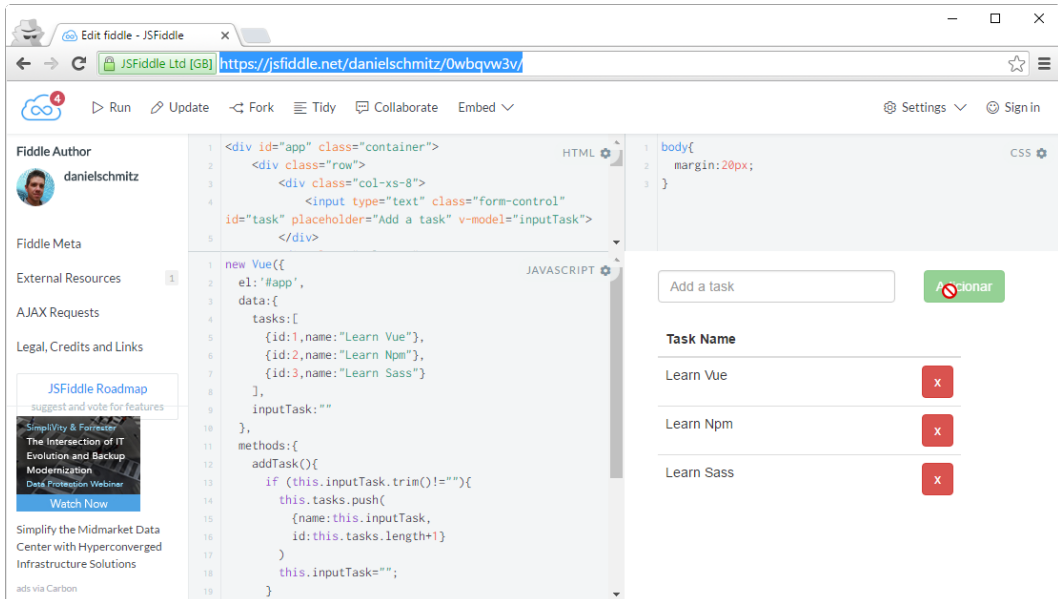
```
addTask(){
  if (this.inputTask.trim()!=""){
    //.....
  }
}
```

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

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

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/>

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

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

```
<h1 v-if="name===' '>Hello World</h1>
<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:

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

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

2.19.1 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:

```
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">  
<label for="jack">Jack</label>  
<input type="checkbox" id="john" value="John" v-model="checkedNames">  
<label for="john">John</label>  
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">  
<label for="mike">Mike</label>  
<br>  
<span>Checked names: {{ checkedNames | json }}</span>
```

```
new Vue({  
  el: '...',  
  data: {  
    checkedNames: []  
  }  
})
```

Resultado:



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

2.19.2 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.

2.19.3 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:

```
<select v-model="selected">
  <option v-for="option in options" v-bind:value="option.value">
    {{ option.text }}
  </option>
</select>
```

2.19.4 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:

```
$ 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:

```
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:

```
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:


```
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:

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

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 presenteia 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:

```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
  </div>
</template>

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

```
      msg: 'Hello Vue!'
    }
  }
}
</script>

<style>
  body {
    font-family: Helvetica, sans-serif;
  }
</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:

```
data: {
  msg: "Hello World"
}
```

Temos:

```
export default {
  data(){
    return {
      msg: 'Hello Vue!'
    }
  }
}
```

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`

```
<template>
  <div>My Menu</div>
</template>
<script>
  export default{

  }
</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

```
<template>
  <div id="app">
    <my-menu></my-menu>
    <h1>{{ msg }}</h1>
  </div>
</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

```
<template>
  <div id="app">
    <my-menu></my-menu>
    <h1>{{ msg }}</h1>
  </div>
</template>

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

  export default {
    data () {
      return {
```

```
      msg: 'Hello Vue!'
    }
  }
}
</script>

<style>
  body {
    font-family: Helvetica, sans-serif;
  }
</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`

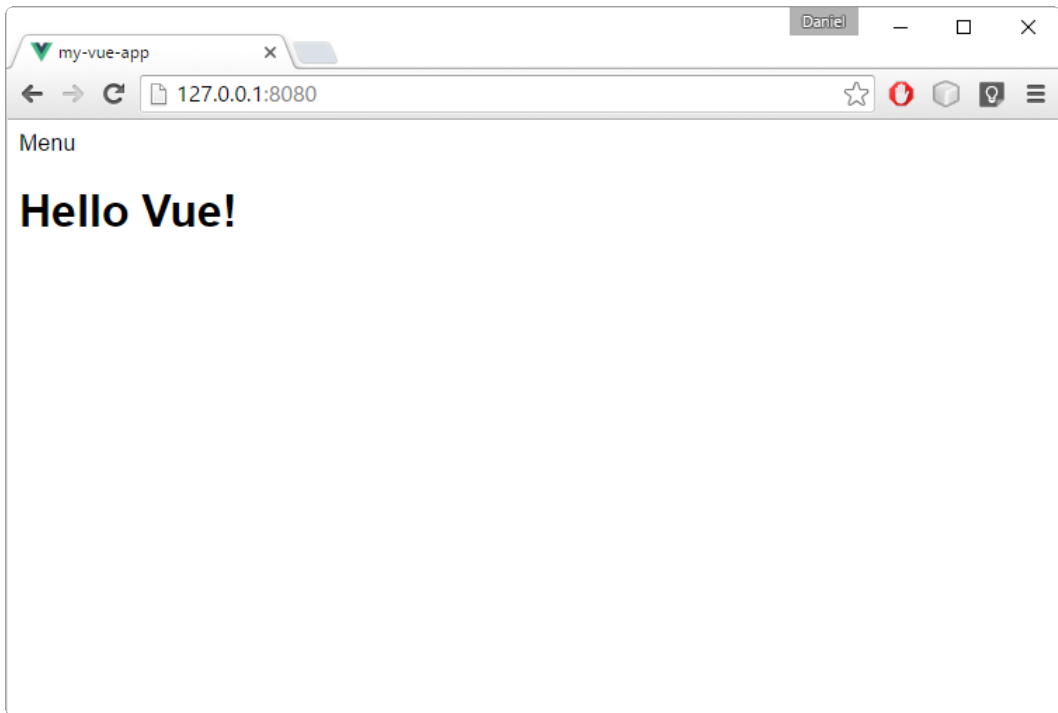
```
<template>
  <div id="app">
    <my-menu></my-menu>
    <h1>{{ msg }}</h1>
  </div>
</template>

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

  export default {
    components:{
      MyMenu
```

```
    },  
    data () {  
      return {  
        msg: 'Hello Vue!'  
      }  
    }  
  }  
}  
</script>  
  
<style>  
  body {  
    font-family: Helvetica, sans-serif;  
  }  
</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:

srcMyMenu.vue

```
<template>
  <h4>{{title}}</h4>
</template>
<script>
  export default{
    props: ['title']
  }
</script>
```

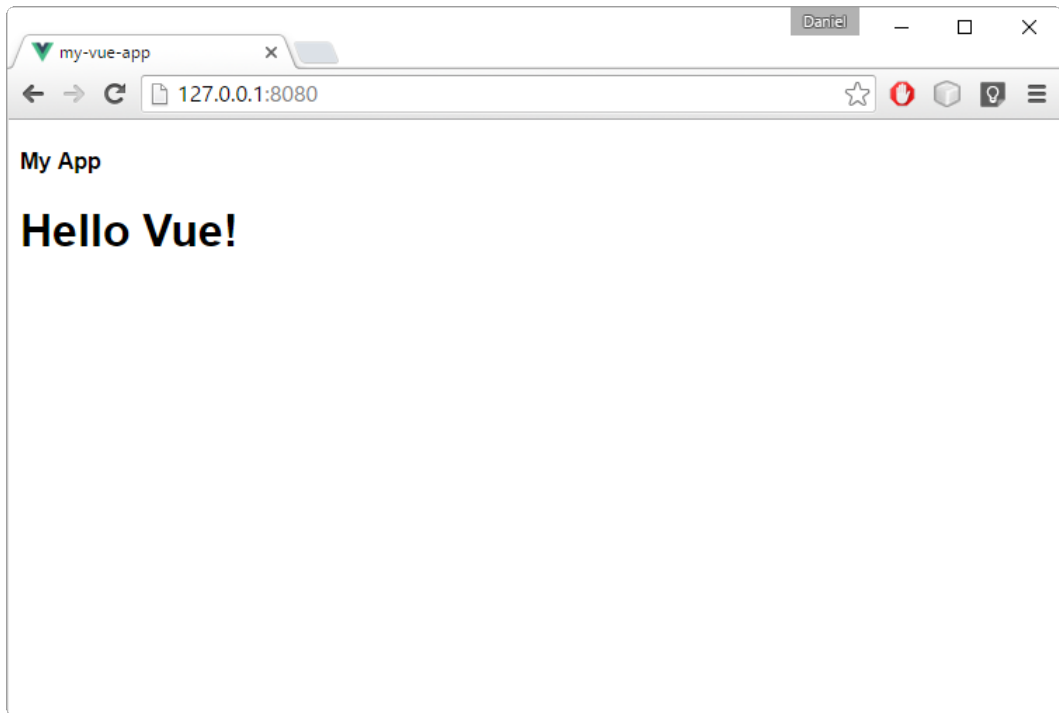
Desta forma, a propriedade `title` pode ser repassada no componente `App`, da seguinte forma:

`src/App.vue`

```
<template>
  <div id="app">
    <my-menu title="My App"></my-menu>
    <h1>{{ msg }}</h1>
  </div>
</template>
<script>

</script>
```

O resultado é semelhante a figura a seguir:



3.8.1 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`.

3.8.2 Validações e valor padrão

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

```
props: {  
  // tipo número  
  propA: Number,  
  
  // String ou número (1.0.21+)  
  propM: [String, Number],  
  
  // Uma propriedade obrigatória  
  propB: {  
    type: String,  
    required: true  
  },  
  
  // um número com valor padrão  
  propC: {  
    type: Number,  
    default: 100  
  },  
  
  // Uma validação customizada  
  propF: {  
    validator: function (value) {  
      return value > 10  
    }  
  },  
  
  //Retorna o JSON da propriedade  
  propH: {  
    coerce: function (val) {  
      return JSON.parse(val) // cast the value to Object  
    }  
  }  
}
```

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:

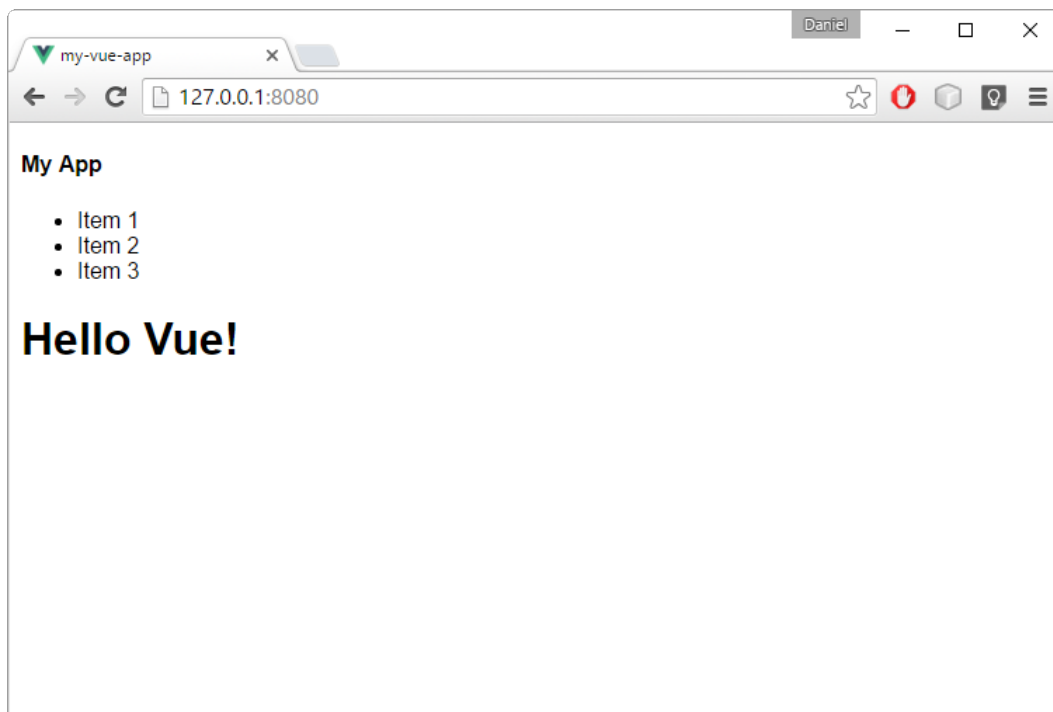
```
<my-menu title="My App">
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</my-menu>
```

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

```
<template>
  <div>
    <h4>{{title}}</h4>
    <div>
      <slot></slot>
    </div>
  </div>
</template>
<script>
  export default{
    props: ['title']
  }
</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

```
<template>
  <div id="app">
    <my-menu title="My App" v-on:menu-click="onMenuClick">
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
      </ul>
    </my-menu>
    <h1>{{ msg }}</h1>
  </div>
</template>

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

  export default {
    components:{
      MyMenu
    },
    data () {
      return {
```

```
      msg: 'Hello Vue!'
    },
    methods: {
      onMenuClick: function(e) {
        alert("menu click");
      }
    }
  }
</script>

<style>
  body {
    font-family: Helvetica, sans-serif;
  }
</style>
```

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

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

Que irá chamar o método onMenuClick, definido pelo método:

```
methods: {
  onMenuClick: function() {
    alert("menu click");
  }
}
```

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

src/Menu.vue

```
<template>
  <div @click="onMenuClick">
    <h4>{{title}}</h4>
    <div>
      <slot></slot>
    </div>
    <hr/>
  </template>
<script>
  export default{
    props: ['title'],
    methods:{
      onMenuClick : function(){
        this.$emit('menu-click');
      }
    }
  }
</script>
```

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

3.10.1 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

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

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

src/App.vue

```
methods:{  
  onClick: function(message){  
    alert(message);  
  }  
}
```

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

```
<template>
  <div id="app">
    <app-header></app-header>
    <app-content></app-content>
    <app-footer></app-footer>
  </div>
</template>

<script>
  import AppHeader from './layout/AppHeader.vue'
  import AppContent from './layout/AppContent.vue'
  import AppFooter from './layout/AppFooter.vue'

  export default {
    components: {
      AppHeader, AppContent, AppFooter
    }
  }
</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

```
<template>
  <h4>Header</h4>
</template>
<script>
  export default{

  }
</script>
```

src/layout/AppContent.vue

```
<template>
  <h4>Content</h4>
</template>
<script>
  export default{

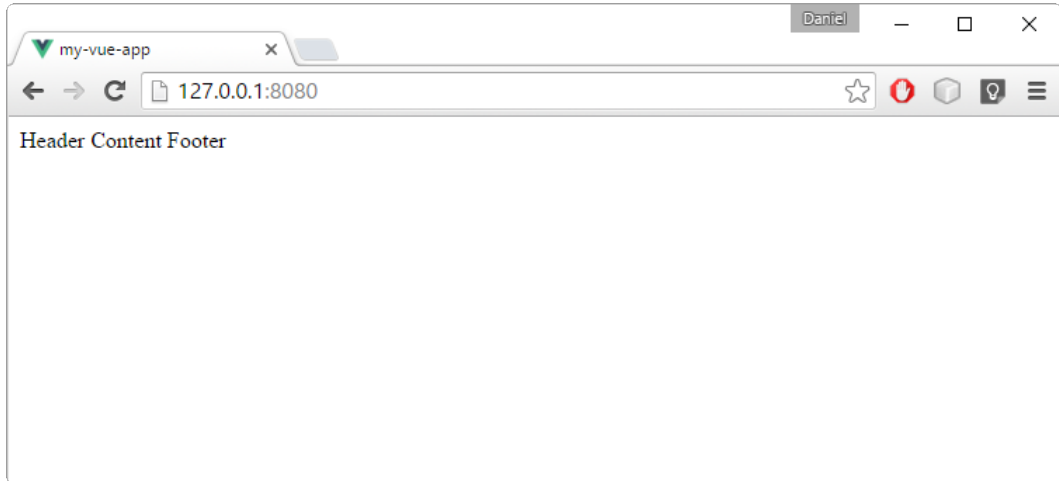
  }
</script>
```

src/layout/AppFooter.vue

```
<template>
  <h5>Footer</h5>
</template>
<script>
  export default{

  }
</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:

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

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

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>my-vue-app</title>

  <!--Materialize Styles-->
  <link href="http://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
  <link type="text/css" rel="stylesheet" href="node_modules/materialize-css/dist/css/materialize.min.css" media="screen,projection"/>

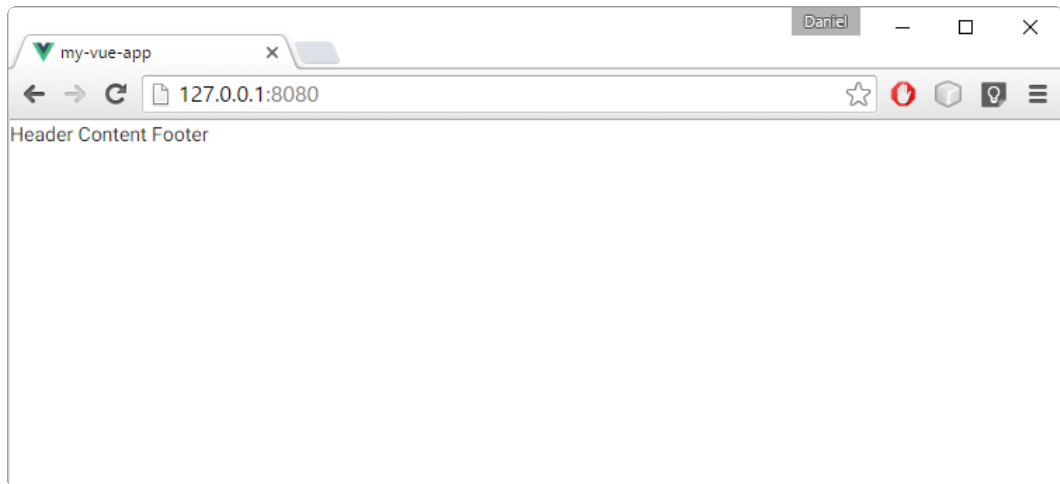
</head>
<body>
  <app></app>
  <!--Materialize Javascript-->
  <script src="node_modules/jquery/dist/jquery.min.js"></script>
  <script src="node_modules/materialize-css/dist/js/materialize.min.js"></script>
  <script src="dist/build.js"></script>
</body>
</html>
```



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:

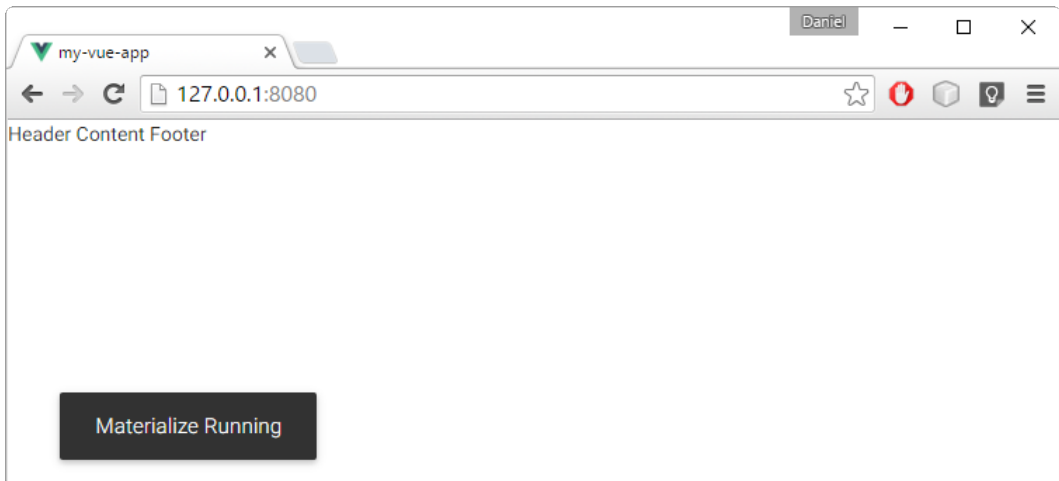
`src/App.vue`

```
<template>
  <div id="app">
    <app-header></app-header>
    <app-content></app-content>
    <app-footer></app-footer>
  </div>
</template>
```

```
<script>
  import AppHeader from './layout/AppHeader.vue'
  import AppContent from './layout/AppContent.vue'
  import AppFooter from './layout/AppFooter.vue'

  export default {
    components:{
      AppHeader,AppContent,AppFooter
    },
    created:function(){
      Materialize.toast('Materialize Running', 1000)
    }
  }
</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:

```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo">Logo</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li><a href="sass.html">Sass</a></li>
      <li><a href="badges.html">Components</a></li>
      <li><a href="collapsible.html">JavaScript</a></li>
    </ul>
  </div>
</nav>
```

E adicionar no template do AppHeader:

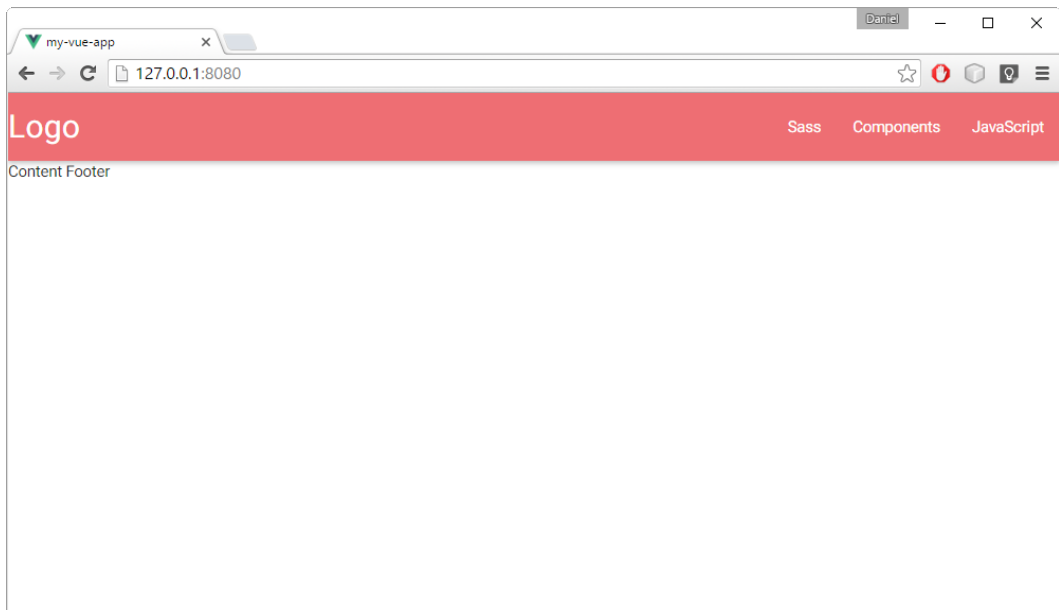
src/layout/AppHeader.vue

```
<template>
  <nav>
    <div class="nav-wrapper">
      <a href="#" class="brand-logo">Logo</a>
      <ul id="nav-mobile" class="right hide-on-med-and-down\
">
        <li><a href="sass.html">Sass</a></li>
        <li><a href="badges.html">Components</a></li>
        <li><a href="collapsible.html">JavaScript</a></li>
      </ul>
    </div>
  </nav>
</template>
```

```
<script>
  export default{

  }
</script>
```

O que resulta em:



3.14 Alterando o rodapé

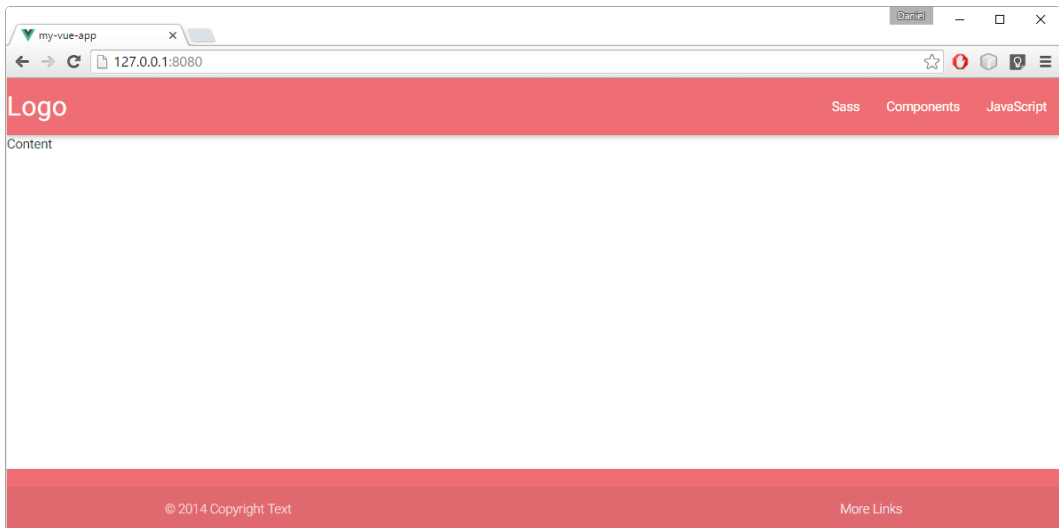
O mesmo pode ser aplicado ao rodapé da página, copiando um estilo direto do materialize e aplicando no AppFooter:

src/layout/AppFooter.vue

```
<template>
  <footer class="page-footer">
    <div class="footer-copyright">
      <div class="container">
        © 2014 Copyright Text
        <a class="grey-text text-lighten-4 right" href="#" \
>More Links</a>
      </div>
    </div>
  </footer>
</template>
<script>
  export default{

  }
</script>
<style>
  footer {
    position: fixed;
    bottom: 0;
    width: 100%;
  }
</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 AppContent 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:

```
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

```
import Vue from 'vue'
import App from './App.vue'
import VueRouter from 'vue-router'
```

```
Vue.use(VueRouter)
const router = new VueRouter()
router.map({
  '/': {
    component: ??????????????
  }
})
```

```
});  
router.start(App, 'App')
```

Ao adotarmos o `vue-router`, passamos a importá-lo através do `import VueRouter from 'vue-router'` e configuramos o `Vue` para usá-lo através do `Vue.use(VueRouter)`. A configuração do router é feita pelo `router.map`, que por enquanto ainda não possui um componente principal. Após a configuração, usamos `router.start` para iniciar a aplicação carregando o componente `App`.

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:

```
<template>  
  Hello World Router  
</template>  
<script>  
  export default{  
  
  }  
</script>
```

4.3 Configurando o `router.map`

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

main.js

```
import Vue from 'vue'
import App from './App.vue'
import VueRouter from 'vue-router'

import HelloWorldRouter from './components/HelloWorldRouter\
.vue'

Vue.use(VueRouter)
const router = new VueRouter()
router.map({
  '/': {
    component: HelloWorldRouter
  }
});
router.start(App, 'App')
```

4.4 Configurando o router-view

Após a configuração do primeiro componente gerenciado pelo Router, nós precisamos configurar onde o componente será carregado. Veja que, uma coisa é iniciar o router no App, outra coisa é configurar o local onde os componentes serão carregados.

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

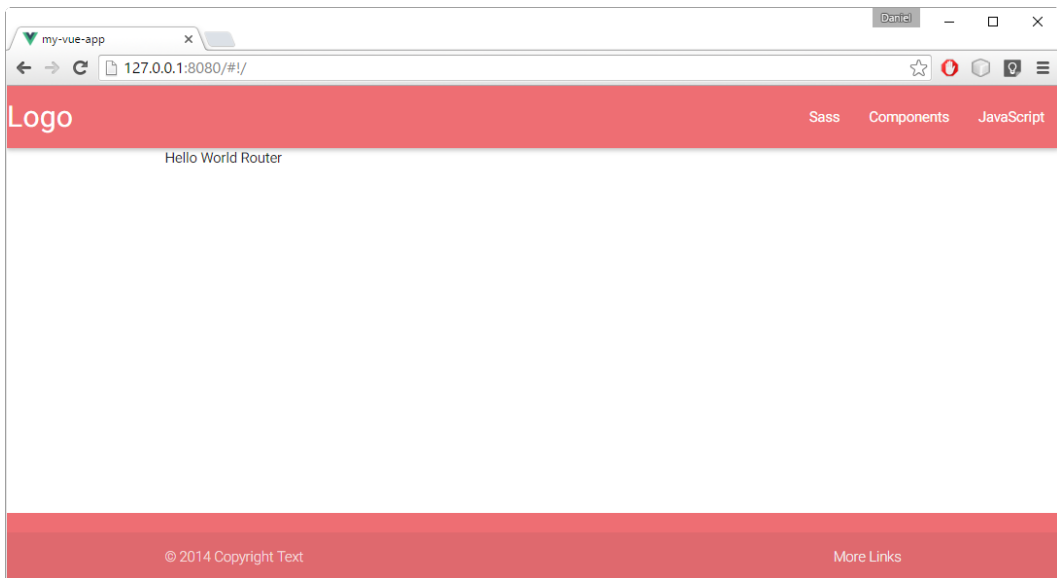
src/layout/AppContent.vue

```
<template>
  <div class="container">
    <router-view></router-view>
  </div>
</template>
<script>
  export default{

  }
</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

```
<template>
  <div class="row">
    <div class="col s12 m6">
      <div class="card blue-grey darken-1">
        <div class="card-content white-text">
          <span class="card-title">Card Title</span>
          <p>I am a very simple card. I am good at containing\
small bits of information.
          I am convenient because I require little markup t\
o use effectively.</p>
        </div>
        <div class="card-action">
          <a href="#">This is a link</a>
          <a href="#">This is a link</a>
        </div>
      </div>
    </div>
  </div>
</template>
<script>
  export default{

  }
</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

```
<template>
  <a class="waves-effect waves-light btn">button</a>
  <a class="waves-effect waves-light btn"><i class="material-
l-icons left">cloud</i>button</a>
  <a class="waves-effect waves-light btn"><i class="material-
l-icons right">cloud</i>button</a>
</template>
<script>
  export default{
  }
</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

```
import Vue from 'vue'
import App from './App.vue'
import VueRouter from 'vue-router'

import HelloWorldRouter from './components/HelloWorldRouter\
.vue'
import Card from './components/Card.vue'
import Buttons from './components/Buttons.vue'

Vue.use(VueRouter)
const router = new VueRouter()
router.map({
  '/': {
```

```
    component: HelloWorldRouter
  },
  '/card': {
    component: Card
  },
  '/buttons': {
    component: Buttons
  }
});
router.start(App, 'App')
```

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

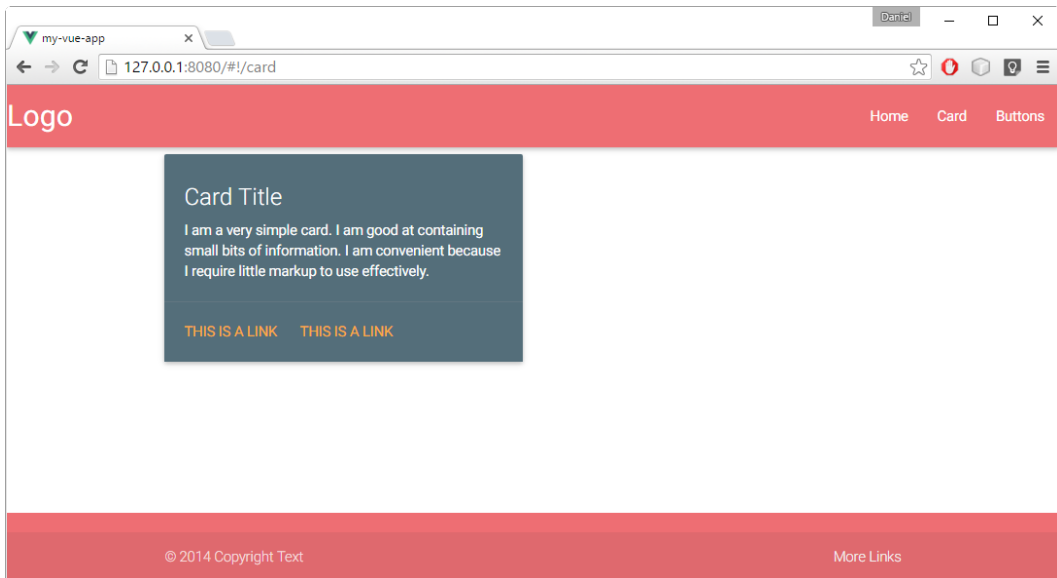
```
<template>
  <nav>
    <div class="nav-wrapper">
      <a href="#" class="brand-logo">Logo</a>
      <ul id="nav-mobile" class="right hide-on-med-and-down\
">

        <li><a v-link="{ path: '/' }">Home</a></li>
        <li><a v-link="{ path: '/card' }">Card</a></li>
        <li><a v-link="{ path: '/buttons' }">Buttons</a></li>
```

```
i>
    </ul>
  </div>
</nav>
</template>
<script>
  export default{

  }
</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.



4.6.1 Repassando parâmetros no link

Caso seja necessário repassar parâmetros para o router, pode-se fazer o seguinte:

```
<a v-link="{ path: '/user/edit', params: { userId: 123 } }">\nEdit User</a>
```

Neste caso, o router atua na geração do link `/#/user/edit?userId=123`, que pode ser usado através da propriedade `$route.params`, como por exemplo `$route.params.userId`.

4.7 Classe ativa

Quando selecionamos um item do menu, podemos alterar a classe daquele item para indicar ao usuário que o item está selecionado. Isso é feito através de três passos distintos. Primeiro, precisa-se descobrir qual o nome da classe que deixa o item de menu com a aparência de selecionada. Depois, temos que configurar o router para indicar que aquela classe é a que indica que o item está ativo. Por último, adicionamos a marcação `v-link-active` ao elemento html correspondente ao item de menu.

O projeto `my-vue-app` usa o `materialize`, então a classe que determina o item ativo é `active`. Para configurá-la, alteramos a criação do router, no arquivo `main.js`, da seguinte forma:

`src/main.js`

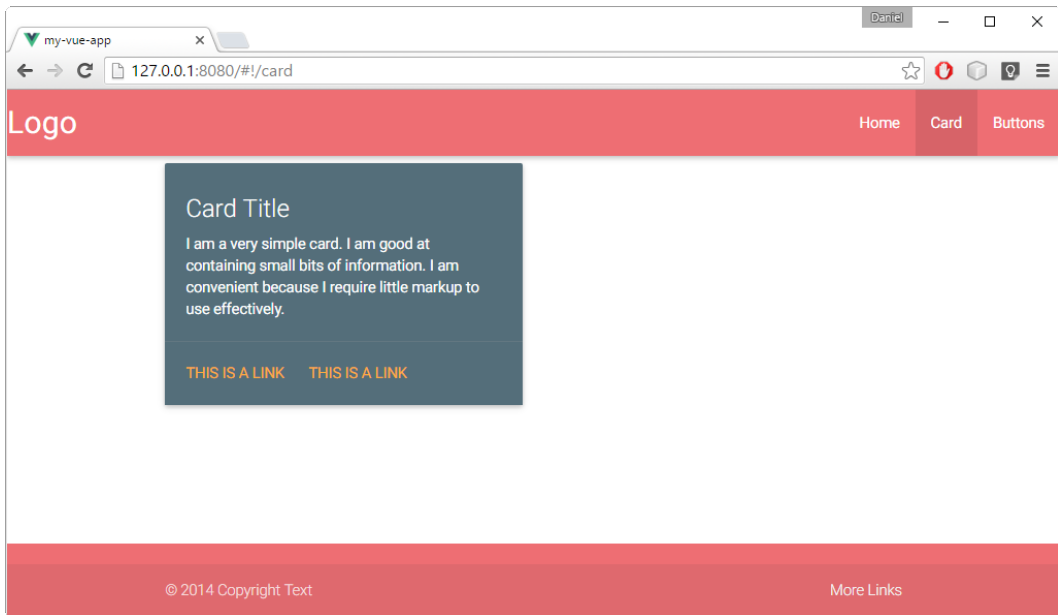
```
...  
const router = new VueRouter({  
  linkActiveClass: 'active'  
})  
...
```

Os itens de menu são configurados no `AppHeader`, da seguinte forma:

```
<template>
  <nav>
    <div class="nav-wrapper">
      <a href="#" class="brand-logo">Logo</a>
      <ul id="nav-mobile" class="right hide-on-med-and-down\
">
        <li v-link-active><a v-link="{ path: '/' }">Home</a>
      </li>
        <li v-link-active><a v-link="{ path: '/card' }">Car\
      </a></li>
        <li v-link-active><a v-link="{ path: '/buttons' }">\
      Buttons</a></li>
      </ul>
    </div>
  </nav>
</template>
<script>
  export default{

  }
</script>
```

Perceba que o item “v-link-active” do Vue foi adicionado no elemento da lista de itens de menu. O resultado desta configuração é exibido a seguir:



4.8 Filtrando rotas pelo login

Uma das funcionalidades do router é prover uma forma de redirecionar o roteamento dado algum parâmetro. Vamos supor que a rota `/card` necessite que o usuário esteja logado. Para isso, podemos adicionar uma variável no mapeamento da rota, conforme o exemplo a seguir:

```
'/foo': {  
  component: HelloWorldRouter  
},  
'/card': {  
  component: Card,  
  auth: true  
},  
'/buttons': {  
  component: Buttons
```

```
    }  
  });
```

Perceba que a entrada `/card` possui o parâmetro `auth: true`. Após inserir esta informação, temos que usar o método `router.beforeEach` para tratar as rotas cujo o parâmetro `auth` for `true`. Como ainda não estamos tratando o login do usuário, vamos apenas fazer uma simulação:

```
router.beforeEach(function(transition){  
  let authenticated = false  
  if (transition.to.auth && !authenticated)  
  {  
    transition.redirect('/login')  
  } else {  
    transition.next()  
  }  
})
```

Neste código, se `auth` for `true` e `authenticated` for `false`, o fluxo da rota que irá mudar para `/login`, que ainda não foi implementado.

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:

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

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

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

import VueRouter from 'vue-router'
import VueResource from 'vue-resource'

import HelloWorldRouter from './components/HelloWorldRouter\
.vue'
import Card from './components/Card.vue'
import Buttons from './components/Buttons.vue'

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

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:

```
Vue.http.options.root = '/root';  
Vue.http.headers.common['Authorization'] = 'Basic YXBpOnBhc\  
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:

```
[  
  {  
    "name": "User1",  
    "email": "user1@gmail.com",  
    "country": "USA"  
  },  
  {  
    "name": "User2",  
    "email": "user2@gmail.com",  
    "country": "Mexico"  
  },  
  {  
    "name": "User3",  
    "email": "user3@gmail.com",  
    "country": "France"  
  },  
  {  
    "name": "User4",  
    "email": "user4@gmail.com",  
    "country": "Brazil"  
  }  
]
```

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 é altere-o para:

`src/components/Buttons.vue`

```
<template>

  <a @click="callUsers"
  class="waves-effect waves-light btn">Call Users</a>

  <a @click="countUsers"
  class="waves-effect waves-light btn">
    <i class="material-icons left">cloud</i>Count Users
  </a>

  <a class="waves-effect waves-light btn">
    <i class="material-icons right">cloud</i>button
  </a>

</hr>
<pre>
  {{ users | json }}
</pre>
</template>
<script>
  export default{
    data() {
      return{
        users: null
      }
    },
    methods: {
```

```
callUsers: function(){
  this.$http({url: '/users.json', method: 'GET'})
    .then(function (response) {
      this.users = response.data
    }, function (response) {
      Materialize.toast('Erro!', 1000)
    });
},
countUsers: function(){
  Materialize.toast(this.users.length, 1000)
}
}
}
</script>
```

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

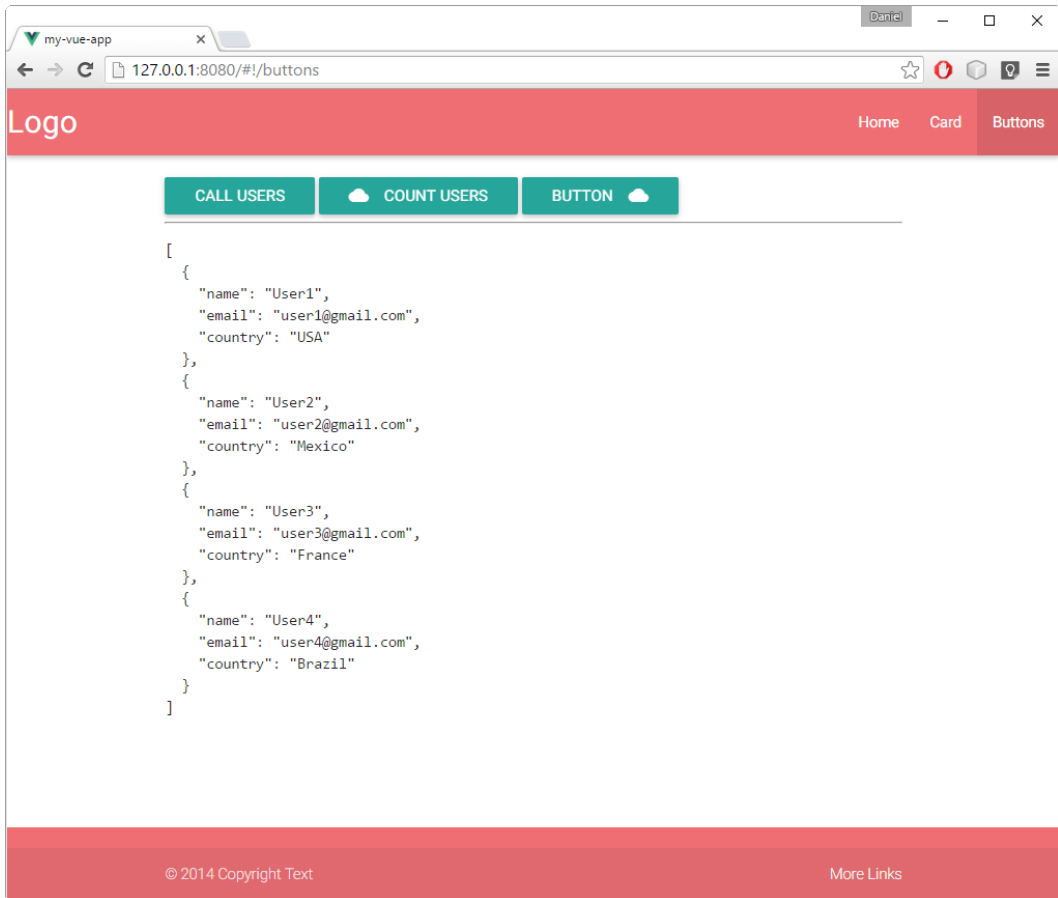
```
this.$http({url: '/users.json', method: 'GET'})
  .then(function (response) {
    this.users = response.data
  }, function (response) {
    Materialize.toast('Error: ' + response.statusText, 3000)
  });
```

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:

```
{{ 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`:

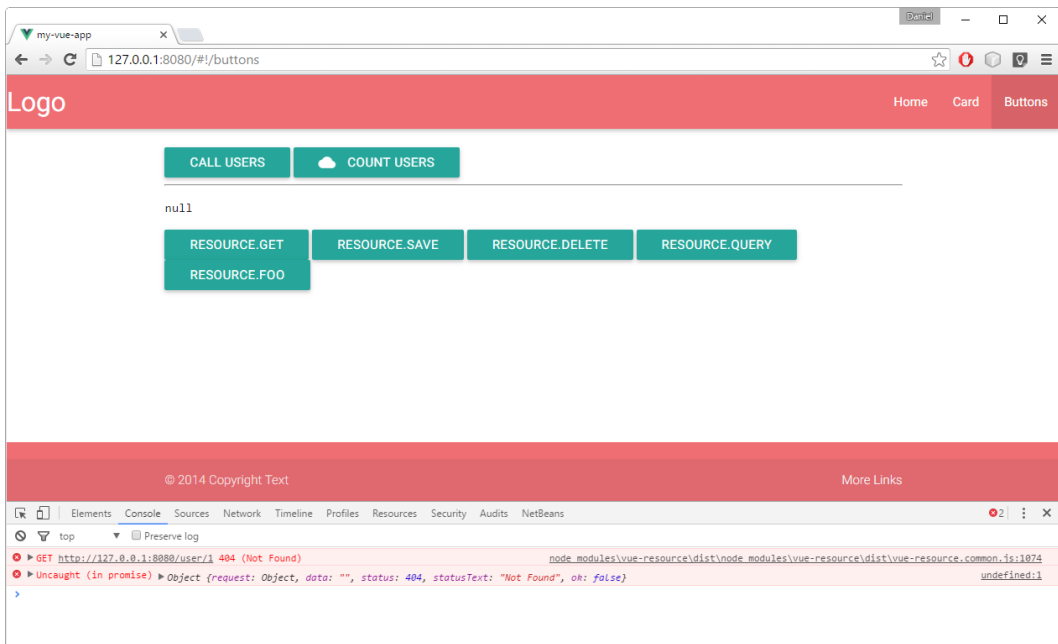
```
export default{
  data() {
    return{
      users: null,
      resourceUser : this.$resource('user{/id}')
```

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

```
<template>
  ...
  <a class="btn" @click="resourceGet">resource.get</a>
  ...
</template>
<script>
  methods: {
    ...
    resourceGet: function(){
      this.resourceUser.get({id:1}).then(function (response\
    ) {
      console.log(response)
```

```
    });  
  }  
}  
...  
</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`:


```
resourceUser : this.$resource('user{/id}', null, {'foo': {method: 'GET', url: "/user/foo"}}}
```

Neste exemplo, pode-se realizar a chamada:

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

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

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 um 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.

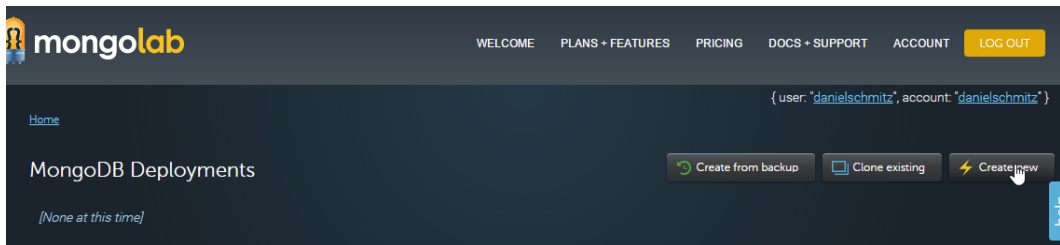
Todas estas tecnologias podem ser instaladas via **npm**, conforme será visto a seguir.

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

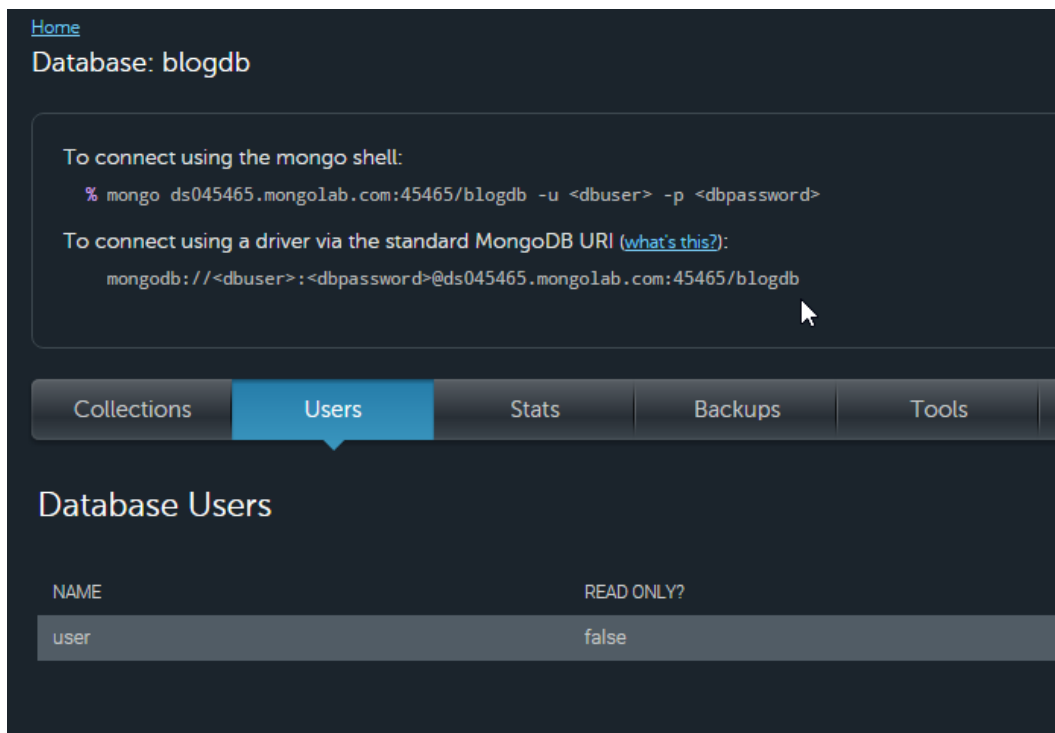
The most economical plans for production applications running on AWS. Plans come standard with 2 data nodes plus an arbiter node.

<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 provides connection instructions:

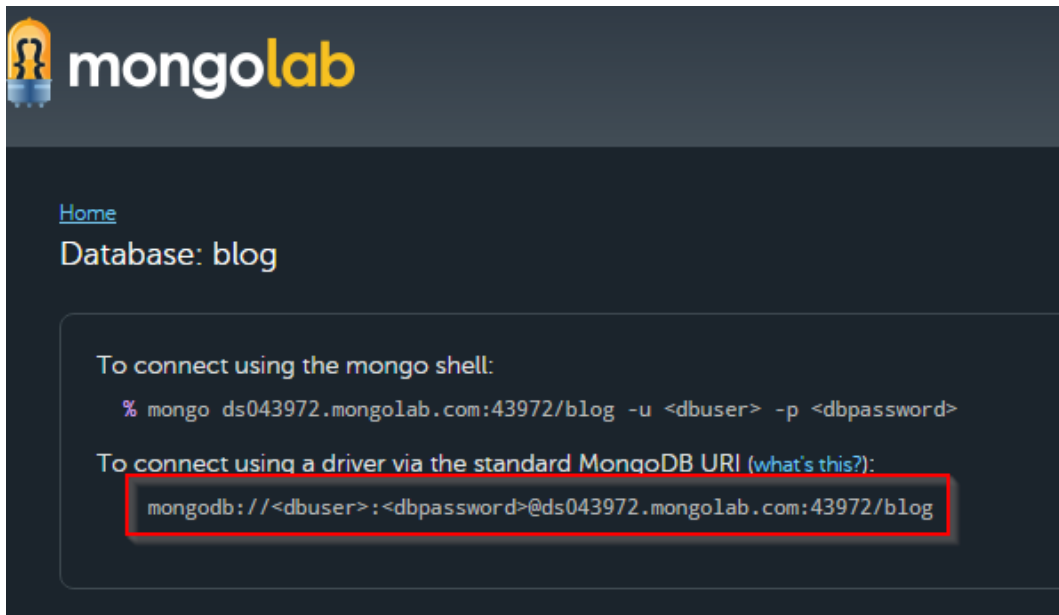
To connect using the mongo shell:
`% mongo ds045465.mongolab.com:45465/blogdb -u <dbuser> -p <dbpassword>`

To connect using a driver via the standard MongoDB URI ([what's this?](#)):
`mongodb://<dbuser>:<dbpassword>@ds045465.mongolab.com:45465/blogdb`

Below the instructions is a navigation bar with tabs: 'Collections', 'Users' (selected), 'Stats', 'Backups', and 'Tools'. Under the 'Users' tab, the section 'Database Users' is visible. It contains a table with two columns: 'NAME' and 'READ ONLY?'. The table has one row with the values 'user' and 'false'.

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:

```
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:

```
npm i
```

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


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

6.4 Estrutura do projeto

A estrutura do projeto está focada na seguinte arquitetura:

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

```
var mongoose    = require('mongoose');
var Schema      = mongoose.Schema;

var userSchema = new Schema({
  name: String,
  login: String,
  password: String
});

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

```
var mongoose    = require('mongoose');
var Schema      = mongoose.Schema;

var postSchema = new Schema({
  title: String,
  author: String,
  body: String,
  user: {type: mongoose.Schema.Types.ObjectId, ref: 'User'},
  date: { type: Date, default: Date.now }
});

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:

```
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:

```
$ 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.

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

Após a instalação, execute:

```
$ nodemon server.js
```

```
[nodemon] 1.8.1  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching: *.*  
[nodemon] starting `node server.js`  
Listen: 8080
```

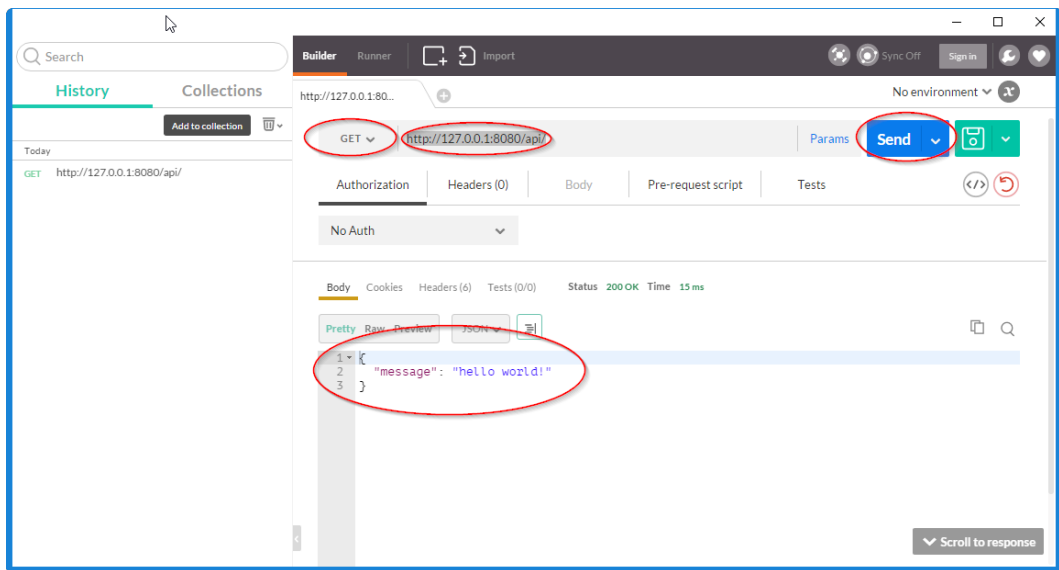
A resposta já indica que, sempre quando o arquivo `server.js` for atualizado, o comando `node server.js` também será.

6.8 Testando a api sem o Vue

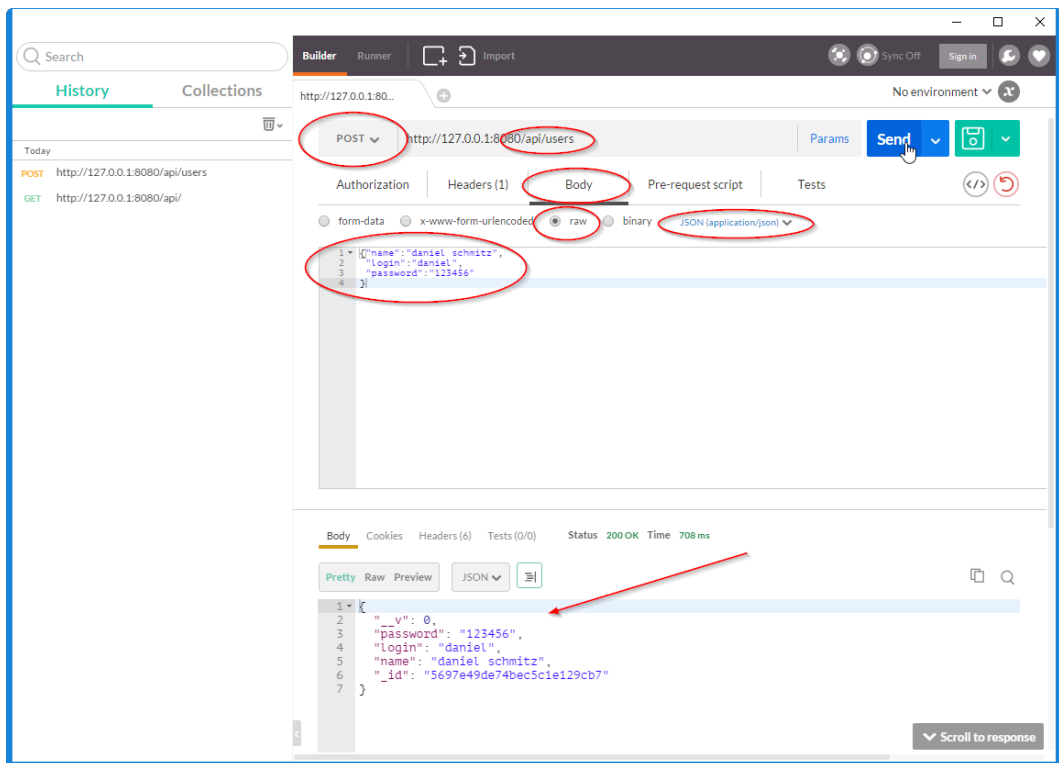
Pode-se testar a api que acabamos de criar, enviando e recebendo dados, através de um programa capaz de realizar chamadas Get/Post ao servidor. Um destes programas se chama **Postman**³, que pode ser instalado com um plugin para o Google Chrome.

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

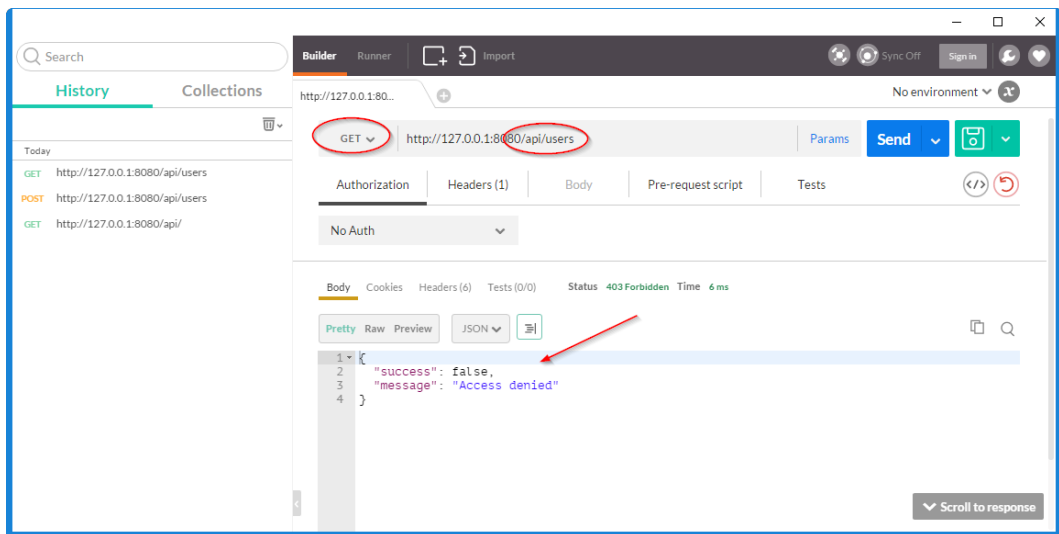
³<https://www.getpostman.com/>



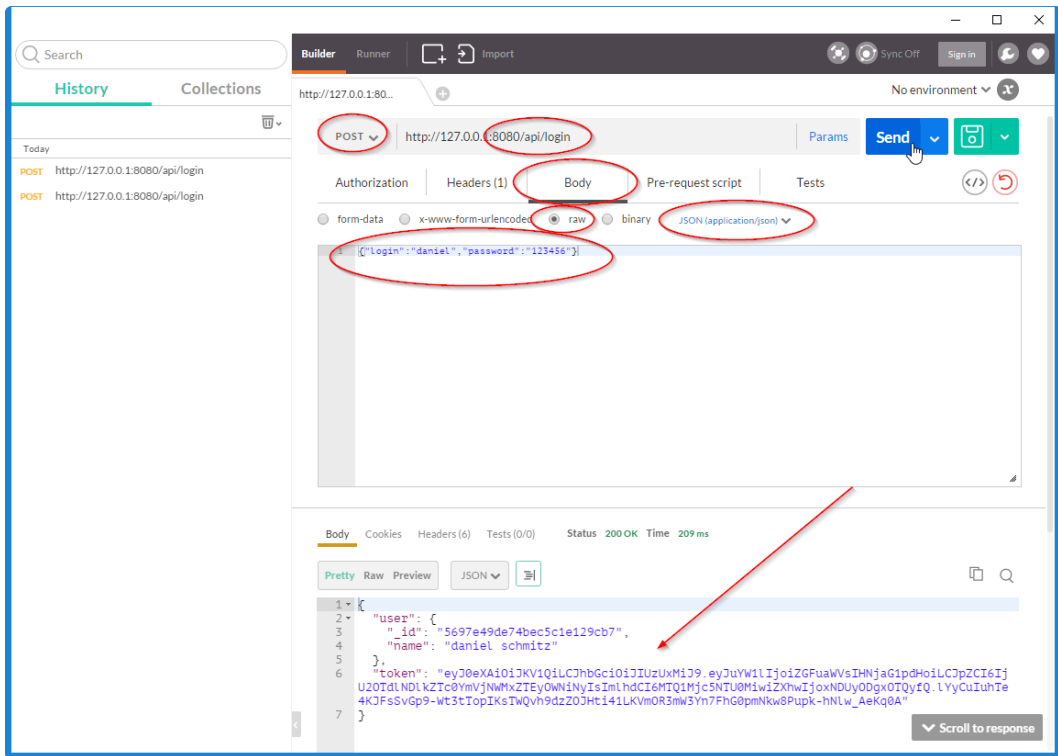
Perceba que obtemos a resposta “hello world”, conforme configurado no servidor. Para criar um usuário, podemos realizar um POST à url `/users` repassando os seguintes dados:



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:

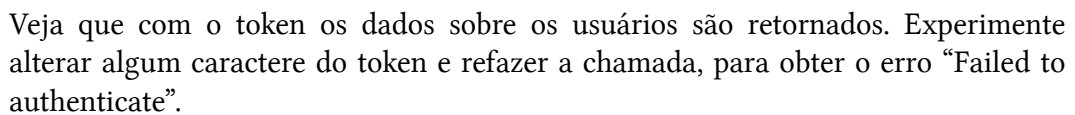


Para realizar o login, acessamos a URL `/api/login`, da seguinte forma:



Veja que ao repassarmos login e password, o token é gerado e retornado para o *postman*. Copie e cole este token para que possamos utilizá-lo nas próximas chamadas ao servidor. No Vue, iremos armazenar este token em uma variável.

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



7. Implementando o Blog com Vue

Após criar o servidor express, e testá-lo no Postman, vamos retornar ao projeto vue criado pelo vue-cli e acertar alguns detalhes.

7.1 Reconfigurando o `packages.json`

O vue-cli usa o servidor `http-server` para exibir como web server em sua aplicação. Como criamos um servidor mais robusto, usando *express*, precisamos alterar a chamada ao servidor na entrada `scripts.serve`.

Localize a seguinte linha do arquivo `package.json`:

```
"serve": "http-server -c 1 -a localhost",
```

e troque por:

```
"serve": "nodemon server.js",
```

Agora quando executarmos `npm run dev`, o servidor express estará pronto para ser executado. A aplicação Vue ficará disponível na url `http://localhost:8080` e a api do express ficará disponível no endereço `http://localhost:8080/api`.

7.2 Instalando pacotes do vue e materialize

Até o momento instalamos apenas as bibliotecas do servidor express e mongodb. Agora vamos instalar as bibliotecas do vue:

```
npm i -S vue-router vue-resource materialize-css
```

7.3 Configurando o router e resource

Altere o arquivo main.js para possibilitar o uso do vue-router e do vue-resource:

src/main.js

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

import VueRouter from 'vue-router'
import VueResource from 'vue-resource'

Vue.use(VueResource)
Vue.use(VueRouter)

const router = new VueRouter({
  linkActiveClass: 'active'
})

router.start(App, 'App')
```

Ainda não configuramos as rotas do v-router, vamos fazer isso agora. Ao invés de adicionar o mapeamento de rotas no arquivo main.js, vamos criar um arquivo chamado routes.js e adicionar lá, veja:

src/routes.js

```
const Routes = {
  '/': {
    component: {
      template: "<b>root</b>"
    }
  },
  '/login': {
    component: {
      template: "<b>login</b>"
    }
  },
  '/logout': {
    component: {
      template: "<b>logout</b>"
    }
  },
  '/addPost': {
    component: {
      template: "<b>addPost</b>"
    }
  },
  '/removePost': {
    component: {
      template: "<b>removePost</b>"
    }
  },
}

export default Routes;
```

O arquivo de rotas possui, por enquanto, um componente com um simples template,

indicando um html a ser carregado quando a rota for usada. Depois iremos criar cada componente individualmente.

Para adicionar as rotas no arquivo main.js, usamos o import, da seguinte forma:

src/main.js

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

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

```
import Routes from './routes.js'
```

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

```
const router = new VueRouter({
  linkActiveClass: 'active'
})
```

```
router.map(Routes)
router.start(App, 'App')
```

Para finalizar a configuração das rotas, vamos adicionar o '`<router-view></router-view>`' no componente App, veja:

src/App.vue

```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
    <router-view></router-view>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        msg: 'Hello Vue!'
      }
    }
  }
</script>

<style>
  body {
    font-family: Helvetica, sans-serif;
  }
</style>
```

7.4 Configurando a interface inicial da aplicação

Após instalar o materialize pelo npm, precisamos referenciá-lo no arquivo index.html. Adicione o arquivo css e js de acordo com o código a seguir (partes em amarelo):

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>blog</title>

  <!--Materialize Styles-->
  <link href="http://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
  <link type="text/css" rel="stylesheet" href="node_modules/materialize-css/dist/css/materialize.min.css" media="screen,projection"/>
</head>
<body>
  <app></app>

  <!--Materialize Javascript-->
  <script src="node_modules/jquery/dist/jquery.min.js"></script>
  <script src="node_modules/materialize-css/dist/js/materialize.min.js"></script>

  <script src="dist/build.js"></script>
</body>
</html>
```

Para adicionar um cabeçalho, editamos o arquivo App.vue incluindo o seguinte html:

src/App.vue

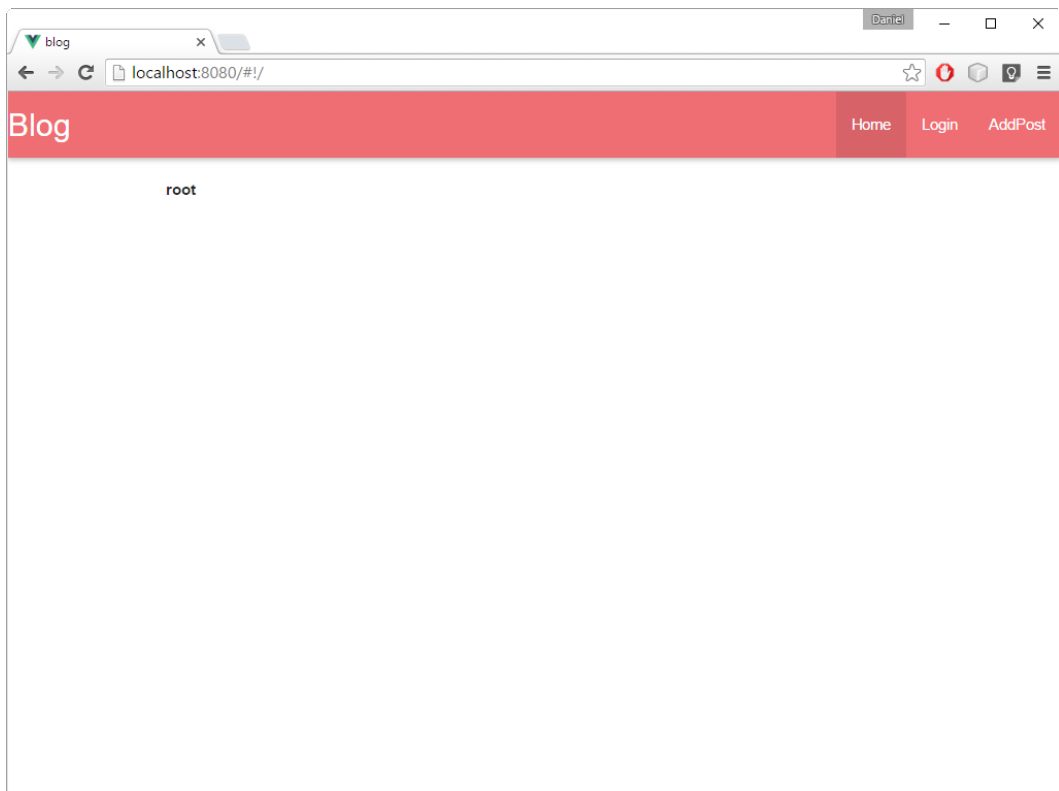
```
<template>
  <div id="app">
    <nav>
      <div class="nav-wrapper">
        <a href="#" class="brand-logo">Blog</a>
        <ul id="nav-mobile" class="right hide-on-med-and-down">
          <li v-link-active><a v-link="{ path: '/' }">Home</a></li>
          <li v-link-active><a v-link="{ path: '/login' }">Login</a></li>
          <li v-link-active><a v-link="{ path: '/addPost' }">AddPost</a></li>
        </ul>
      </div>
    </nav>
    <br/>
    <div class="container">
      <router-view></router-view>
    </div>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        msg: 'Hello Vue!'
      }
    }
  }
}
```

```
</script>

<style>
  body {
    font-family: Helvetica, sans-serif;
  }
</style>
```

Até o momento, o design do site é semelhante a figura a seguir:



7.5 Obtendo posts

Os posts serão carregados na página principal do site, podemos chamá-la de “Home”. Primeio, criamos o componente, inicialmente vazio:

```
<template>
  Home
</template>
<script>
  export default {

  }
</script>
```

E configuramos o arquivo routes.js para usar o componente, ao invés de um simples template:

```
import Home from './Home.js'

const Routes = {
  '/home': {
    component: Home
  },
  '/login': {
    component: {
      template: "<b>login</b>"
    }
  },
  ....
}
```

Perceba que não existe a rota “/” (raiz), pois a rota padrão será a “/home”. Para configurar este comportamento, devemos adicionar um “redirect” no Vue Router, no arquivo main.js, veja:

src/main.js

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

import VueRouter from 'vue-router'
import VueResource from 'vue-resource'

import Routes from './routes.js'

Vue.use(VueResource)
Vue.use(VueRouter)

const router = new VueRouter({
  linkActiveClass: 'active'
})

router.redirect({
  '/': '/home'
})

router.map(Routes)

router.start(App, 'App')
```

Voltando ao `Home.vue`, precisamos acessar a url “/api/posts” para obter os posts já cadastrados e exibi-los na página:

src/Home.vue

```
<template>
  <div v-show="showProgress" class="progress">
    <div class="indeterminate"></div>
  </div>
</template>
<script>
  export default {
    data () {
      return {
        posts: null,
        showProgress: true
      }
    },
    created: function () {
      this.showProgress = true;
      this.$http.get('/api/posts').then(function (response) {
        this.showProgress = false
        this.posts = response.data
        console.log(this.posts);
      }, function (error) {
        this.showProgress = false
        Materialize.toast('Error: ' + error.statusText, 3\
000)
      })
    }
  }
</script>
```

O código até o momento exibe no template uma div com uma barra de progresso, indicando algumas atividade ajax para o usuário. Esta div é controlada pela variável `showProgress`, que é declarada com o valor `false` na seção `data` do componente `vue`.

No método `created`, executado quando o componente está devidamente criado, iniciamos o acesso Ajax a api, através do método `this.$http.get('/api/posts')`, onde o resultado deste acesso é atribuído a variável `posts` e exibido no console do navegador. Perceba o uso da variável `showProgress` para controlar a sua visualização.

Após a variável `posts` estar preenchida, podemos criar o template que irá exibir os posts na página. Usando o framework `materialize`, podemos criar um card para cada post, conforme o código a seguir:

src/Home.vue

```
<template>
  <div v-show="showProgress" class="progress">
    <div class="indeterminate"></div>
  </div>

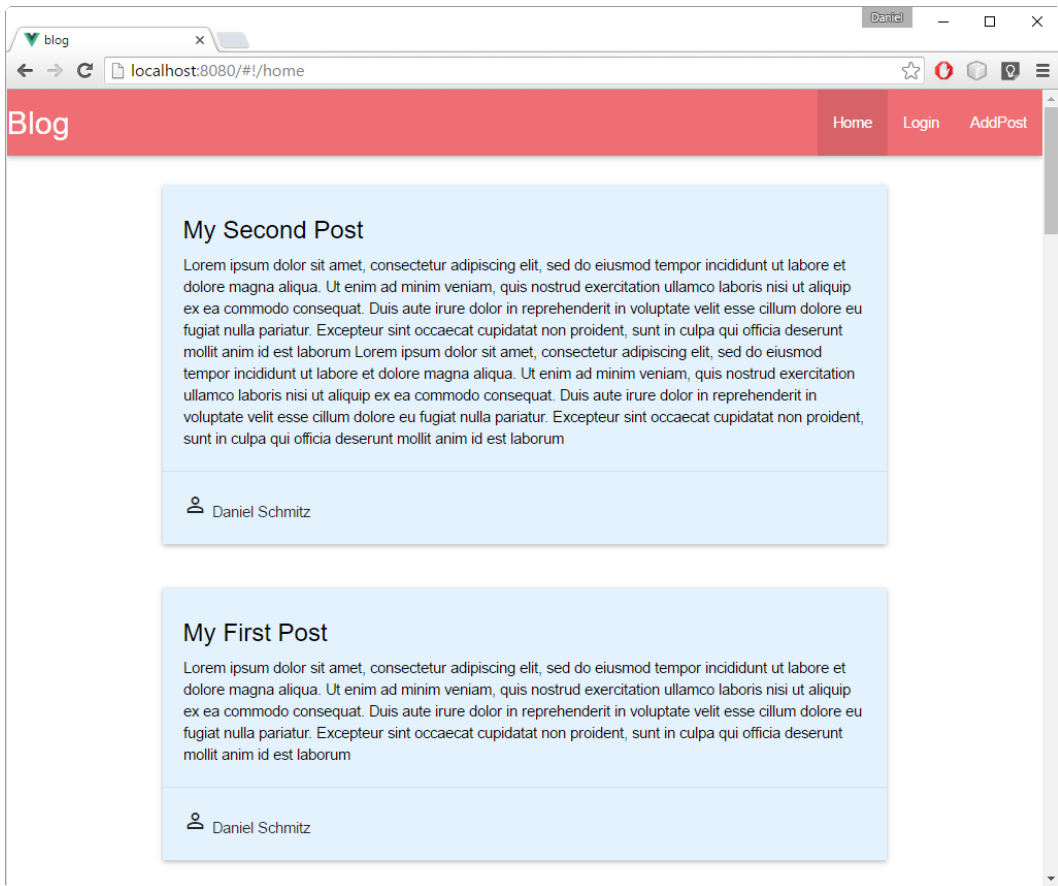
  <div class="row" v-for="post in posts">
    <div class="col s12">
      <div class="card blue lighten-5">
        <div class="card-content black-text">
          <span class="card-title">{{post.title}}</span>
          <p>{{post.text}}</p>
        </div>
        <div class="card-action">
          <span><i class="material-icons">perm_identity</i>
i> {{post.user.name}}</span>
          <!-- <a href="#">This is a link</a> -->
        </div>
      </div>
    </div>
  </div>

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

```
data () {
  return {
    posts: null,
    showProgress: true
  }
},
created: function () {
  this.showProgress = true;
  this.$http.get('/api/posts').then(function (response) {
    this.showProgress = false
    this.posts = response.data
    console.log(this.posts);
  }, function (error) {
    this.showProgress = false
    Materialize.toast('Error: ' + error.statusText, 3\
000)
  })
}
</script>
```

Na seção <template> do Home.vue incluímos o um formato básico de card retirado [deste link](http://materializecss.com/cards.html)¹, incluindo o v-for para navegar entre os posts, exibindo o título do post, o texto e o autor, obtendo uma interface semelhante a exibida a seguir:

¹<http://materializecss.com/cards.html>



Configurando o Vue Validator

O plugin Vue Validator será usado nos formulários. Adicione o plugin através do seguinte comando:

```
npm i -S vue-validator
```

Altere o arquivo main.js incluindo o Vue Validator:

src/main.js

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

import VueRouter from 'vue-router'
import VueResource from 'vue-resource'
import VueValidator from 'vue-validator'

import Routes from './routes.js'

Vue.use(VueResource)
Vue.use(VueRouter)
Vue.use(VueValidator)

const router = new VueRouter({
  linkActiveClass: 'active'
})

router.redirect({
  '/': '/home'
})

router.map(Routes)

router.start(App, 'App')
```

Realizando o login

Para adicionar um post, o usuário necessita estar logado. Vamos agora implementar esta funcionalidade. Primeiro, criamos o componente `Login.vue`, com o seguinte código:

src/Login.vue

```
<template>

  <validator name="validateForm">
    <form class="col s12">

      <div class="row">

        <div class="input-field col s12">
          <i class="material-icons prefix">account_circle\
</i>
          <input id="login" type="text" v-model="user.log\
in" v-validate:login="{ required: false, minlength: 4 }" />
          <label for="login">Login</label>
          <div>
            <span class="chip red lighten-5 right" v-if="\
$validateForm.login.required">Campo requerido</span>
            <span class="chip red lighten-5 right" v-if="\
$validateForm.login.minlength">Mínimo 4 caracteres</span>
          </div>
        </div>

        <div class="input-field col s12">
          <i class="material-icons prefix">vpn_key</i>
          <input id="password" type="password" v-model="u\
ser.password" v-validate:password="{ required: false, minle\
ngth: 4 }" />
          <label for="password">Password</label>
          <div>
            <span class="chip red lighten-5 right" v-if="\
$validateForm.password.required">Campo requerido</span>
            <span class="chip red lighten-5 right" v-if="\
```

```

$validateForm.password.minlength">Mínimo 4 caracteres</span\
>

    </div>
  </div>

  <div class="input-field col s12 m3">
    <input type="checkbox" id="createaccount" v-mod\
el="user.isNew" />
    <label for="createaccount">Create Account?</lab\
el>
  </div>

  <div class="input-field col s12 m9" v-show="user.\
isNew">
    <input id="name" type="text" v-model="user.name\
" />
    <label for="name">Your Name</label>
  </div>

</div>

<div class="input-field col s12">
  <button class="waves-effect waves-light btn right\
" @click="doLogin" v-if="$validateForm.valid">Enviar</butto\
n>
</div>

</form>
</validator>

<div v-show="showProgress" class="progress">
  <div class="indeterminate"></div>
</div>

```

```
</template>
<script>
  export default{
    data () {
      return {
        user: {
          name: "",
          password: "",
          login: "",
          isNew: false
        }
      }
    },
    methods: {
      doLogin: function(){

      }
    }
  }
</script>
```

O login possui três campos de formulário, sendo os dois primeiros representando o usuário e a senha e o último o nome do usuário caso ele deseje se cadastrar. Usamos o Vue Validator para que os campos sejam obrigatoriamente preenchidos, e o botão de enviar surge apenas se a validação estiver correta.

Também incluímos uma caixa de verificação (Create Account) para que os usuários possam se cadastrar (ao invés de logarem no sistema). Quando o usuário acionar o botão Enviar, o método `doLogin` será executado. O código deste método é apresentado a seguir:

src/Login.vue

```
....
methods: {
  doLogin: function() {
    this.showProgress = true;
    this.$http.post('/api/login', this.user).then(function\
(response) {
      this.showProgress = false;
      this.$router.go("/home")
    }, function(error) {
      this.showProgress = false;
      console.log(error);
      Materialize.toast('Error: ' + error.data, 3000)
    });
  }
}
....
```

O método `doLogin` irá realizar uma chamada POST a url `/api/login`. Esta, por sua vez, retorna o token de autenticação do usuário, ou uma mensagem de erro. Com o token de autenticação podemos configurar que o usuário está logado, mas ainda não sabemos onde guardar esta informação no sistema.

Token de autenticação

Quando o usuário realiza o login, é preciso guardar algumas informações tais como o nome de usuário, o seu id e o token que será usado nas próximas requisições. Para fazer isso, vamos criar uma variável chamada “Auth” que irá guardar as informações do usuário autenticado.

Primeiro, crie o arquivo `auth.js` com o seguinte código:

src/auth.js

```
export default {
  setLogin: function(data){
    localStorage.setItem("username",data.user.name);
    localStorage.setItem("userid",data.user._id);
    localStorage.setItem("token",data.token);
  } ,
  getLogin: function(){
    return {
      name:localStorage.getItem("username"),
      id:localStorage.getItem("userid"),
      token:localStorage.getItem("token")
    }
  },
  logout:function(){
    localStorage.removeItem("username");
    localStorage.removeItem("userid");
    localStorage.removeItem("token");
  }
}
```

Para usar esta classe, basta importá-la e preencher as informações de login. No caso do Login, isso é feito antes de redirecionar o fluxo do router para /home, veja:

src/Login.vue

`<script>``import Auth from "../auth.js"`

```
export default{
  data () {
    return {
      showProgress:false,
      user: {
        name:"",
        password:"",
        login:"",
        isNew:false
      }
    }
  },
  created: function(){
    console.log(Auth);
  },
  methods:{
    doLogin:function(){
      this.showProgress=true
      this.$http.post('/api/login',this.user).then(function(response){

        this.showProgress=false
        Auth.setLogin(response.data)
        this.$router.go("/home")

      },function(error){
        this.showProgress=false
      })
    }
  }
}
```



```
        console.log(error)
        Materialize.toast('Error: ' + error.data, 3000)
    });
  }
}
```

Desta forma, após o login, sempre poderemos importar auth.js e usá-lo. As informações de login ficam localizadas no localStorage do navegador.

Criando um Post

Vamos criar o seguinte formulário para adicionar Posts:

src/AddPost.vue

```
<template>
  <h4>Add Post</h4>
  <validator name="validateForm">
    <form class="col s12">

      <div class="row">

        <div class="input-field col s12">
          <input id="login" type="text" v-model="post.title" v-validate:login="{ required: false, minlength: 3 }" />
          <label for="login">Title</label>
          <div>
            <span class="chip red lighten-5 right" v-if="$validateForm.login.required">Campo requerido</span>
            <span class="chip red lighten-5 right" v-if="$validateForm.login.minlength">Mínimo 4 caracteres</span>
          </div>
        </div>
        <div class="input-field col s12">
```

```

        <textarea id="textarea1" class="materialize-text\
area" v-model="post.text"></textarea>
        <label for="textarea1">Texto</label>
      </div>
    </div>
    <div class="input-field col s12">
      <button class="waves-effect waves-light btn right" \
@click="add" v-if="$validateForm.valid">Enviar</button>
    </div>
  </form>
</validator>

<div v-show="showProgress" class="progress">
  <div class="indeterminate"></div>
</div>

</template>
<script>
import Auth from './Auth.js'

export default{
  data (){
    return{
      post:{
        title:"",
        token:"",
        text:"",
        user:{
          _id:""
        }
      }
    }
  },
},

```

```
created: function(){
  let login = Auth.getLogin();
  if (login.token==null){
    this.$router.go("/login")
  }else{
    this.post.user._id=login.id;
    this.post.token=login.token;
  }
},
methods:{
  add: function(){
    this.$http.post('/api/posts',this.post).then(function\
(response){
    this.$router.go("/home");
  },function(error){
    //console.log(error)
    Materialize.toast('Error: ' + error.data.message, 3\
000)
  });
}
}
```

</script>

Este componente verifica se o usuário está logado no método `created`. Em caso negativo, o fluxo de execução do router é redirecionado para a página de Login. O formulário que adiciona o Post é semelhante ao formulário de login, e após criar no botão Enviar, o método `add` é acionado onde executamos uma chamada Ajax enviando os dados do Post.

Para finalizar a inclusão do Post, temos que adicionar no arquivo `routes.js` o componente `AddPost`, conforme o código a seguir:

```
import Home from './Home.vue'
import Login from './Login.vue'
import AddPost from './AddPost.vue'
import Logout from './Logout.vue'

const Routes = {
  '/home': {
    component: Home
  },
  '/login':{
    component: Login
  },
  '/logout':{
    component: {template:'Logout'}
  },
  '/addPost':{
    component: AddPost
  }
}

export default Routes;
```

7.6 Logout

O logout da aplicação é realizado pela chamada ao componente Logout.vue, criado a seguir:

src/Logout.vue

```
<template>
  Logout
</template>
<script>
  import Auth from './auth.js'
  export default {
    data () {
      return {

      }
    },
    created: function () {
      Auth.logout();
      console.log(Auth.getLogin());
      this.$router.go('/home');
    }
  }
</script>
```

No carregamento do componente (método `created`), efetuamos o `logout` e redirecionar o router para a `/home`.

7.7 Refatorando a home

Para finalizar este pequeno sistema, vamos refatorar a home para permitir que os usuários possam excluir os seus posts. Isso é feito através do seguinte código:

```
<template>
  <div v-show="showProgress" class="progress">
    <div class="indeterminate"></div>
  </div>
  <div class="row" v-for="post in posts">
    <div class="col s12">
      <div class="card blue lighten-5">
        <div class="card-content black-text">
          <span class="card-title">{{post.title}}</span>
          <p>{{post.text}}</p>
        </div>
        <div class="card-action">
          <span><i class="material-icons">perm_identity</i> \
            {{post.user.name}}</span>
          <a href="#" @click="remove(post)" class="right blue-text" v-if="login.token!=null && login.id==post.user._id\">Remove</a>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
  import Auth from './auth.js'
  export default {
    data () {
      return {
        posts: null,
        showProgress: true,
        login: Auth.getLogin()
      }
    },
  },
```

```

    created: function(){
      //console.log(Auth.getLogin());
      this.showProgress=true;
      this.loadPosts();
    },
    methods: {
      remove: function(post){
        post.token = Auth.getLogin().token;
        this.$http.delete('/api/posts/'+post._id,post).then\
(function(response){
          this.loadPosts();
        },function(error){
          //console.log(error)
          Materialize.toast('Error: ' + error.data.message, 3\
000)
        });
      },loadPosts:function(){
        this.$http.get('/api/posts').then(function(response){
          this.showProgress=false
          this.posts = response.data
          console.log(this.posts);
        },function(error){
          this.showProgress=false
          Materialize.toast('Error: ' + error.statusText, 300\
0)
        })
      }
    }
  }
</script>

```

O método remove irá remover o post do usuário logado, se o mesmo for dele. O método que recarrega os posts foi refatorado para um novo método chamado

`loadPosts`, no qual pode ser chamado sempre que um Post for removido.

7.8 Conclusão

Criamos um pequeno projeto chamado “blog” no qual testamos algumas funcionalidades como o uso de ajax, login, validação de formulários, uso do router e do resource, entre outros.

Parte 3 - Conceitos avançados

8. Vuex e Flux

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

8.1 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, conseqüentemente, eventos e alterações de dados a todo momento.

8.2 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/>

8.3 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!

8.4 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.

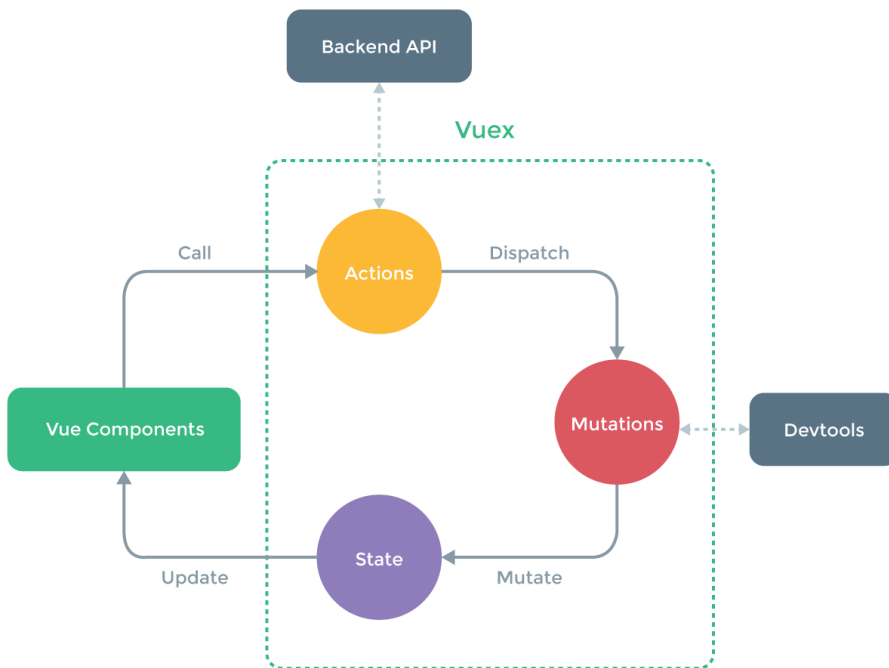
Mutation

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:



8.5 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.

8.5.1 Criando o projeto

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

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

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

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

Para instalar todas as bibliotecas restantes, faça:

```
npm i
```

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

```
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.

8.5.2 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:

```
<template>
  <div>
    <h3>Count is 0</h3>
  </div>
</template>

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

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

```
<template>
  <div>
    <button>+1</button>
    <button>-1</button>
  </div>
</template>

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

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

```
<template>
  <div id="app">
    <Display></Display>
    <Increment></Increment>
  </div>
</template>

<script>
  import Display from './Display.vue'
  import Increment from './Increment.vue'

  export default {
    components: {
      Display, Increment
    },
    data () {
      return {
        msg: 'Hello Vue!'
      }
    }
  }
</script>
```

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

8.5.3 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:

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

Vue.use(Vuex)

const state = {

}

const mutations = {

}

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

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:


```
<template>
  <div id="app">
    <Display></Display>
    <Increment></Increment>
  </div>
</template>

<script>
  import Display from './Display.vue'
  import Increment from './Increment.vue'
  import store from './store.js'

  export default {
    components: {
      Display, Increment
    },
    data () {
      return {
        msg: 'Hello Vue!'
      }
    },
    store
  }
</script>
```

8.5.4 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:

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

Vue.use(Vuex)

const state = {
  count: 0
}

const mutations = {

}

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

8.5.5 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:

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

Vue.use(Vuex)

const state = {
  count: 0
}
```

```
const mutations = {  
  INCREMENT(state){  
    state.count++;  
  },  
  DECREMENT(state){  
    state.count--;  
  }  
}  
  
export default new Vuex.Store({  
  state,  
  mutations  
})
```

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

8.5.6 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:

```
export const incrementCounter = function ({ dispatch, state\  
  }) {  
  dispatch('INCREMENT')  
}  
  
export const decrementCounter = function ({ dispatch, state\  
  }) {  
  dispatch('DECREMENT')  
}
```

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

8.5.7 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:

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

8.5.8 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:

```
<template>  
  <div>  
    <h3>Count is {{getCount}}</h3>  
  </div>  
</template>  
  
<script>  
  
  import { getCount } from './getters'  
  
  export default {  
    vuex: {  
      getters: {  
        getCount  
      }  
    }  
  }  
}
```

```
    }  
  }  
</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.

8.5.9 Alterando o componentet 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.

```
<template>  
  <div>  
    <button @click="incrementCounter">+1</button>  
    <button @click="decrementCounter">-1</button>  
  </div>  
</template>  
  
<script>  
  import { incrementCounter, decrementCounter } from './act\  
ions'  
  
  export default {  
    vuex: {  
      actions: {
```

```
      incrementCounter, decrementCounter
    }
  }
}
</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.

8.5.10 Testando a aplicação

Certifique-se de que a aplicação esteja 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.

8.6 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.

8.7 Chrome vue-devtools

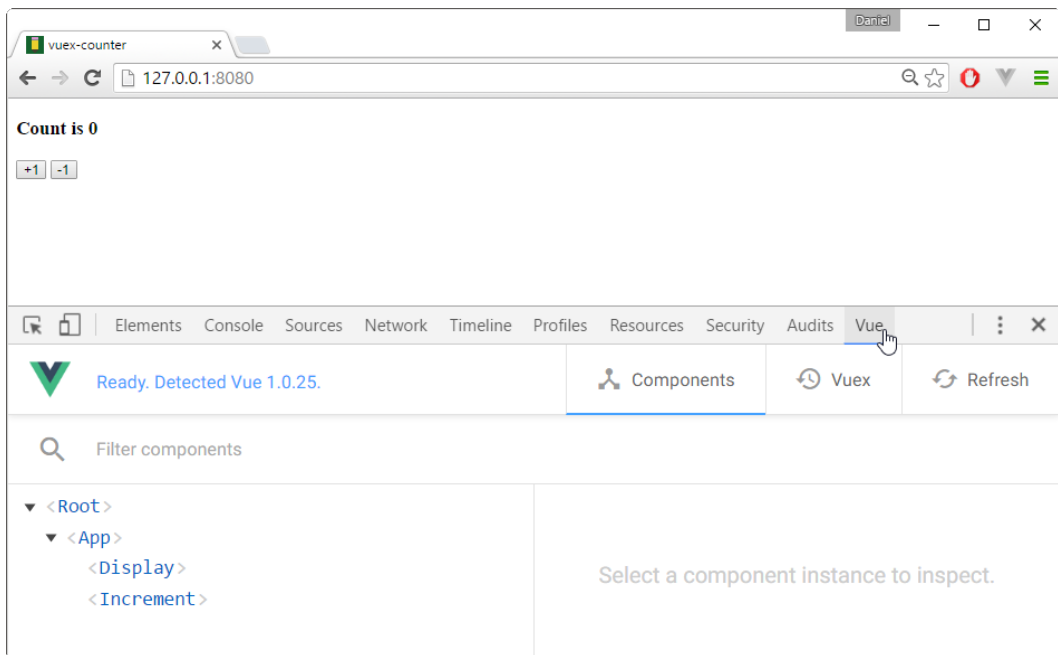
O Chrome [vue-devtools](https://github.com/vuejs/vue-devtools)² é um plugin para o navegador Google Chrome que irá lhe ajudar melhor no debug das aplicações em Vue. Você pode obter o Vue DevTools no [Chrome Web Store](https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd)³.

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/>).

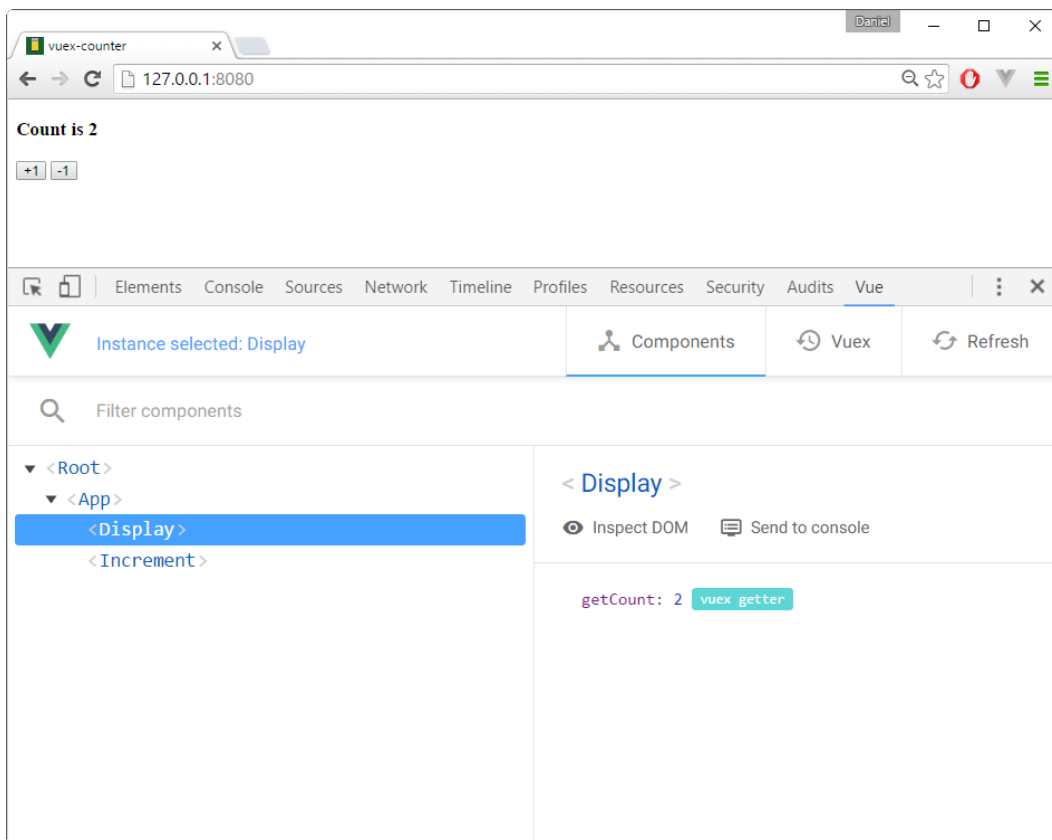
Tecla F12 para abrir o Chrome Dev Tools, e clique na aba Vue, conforme a imagem a seguir:

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

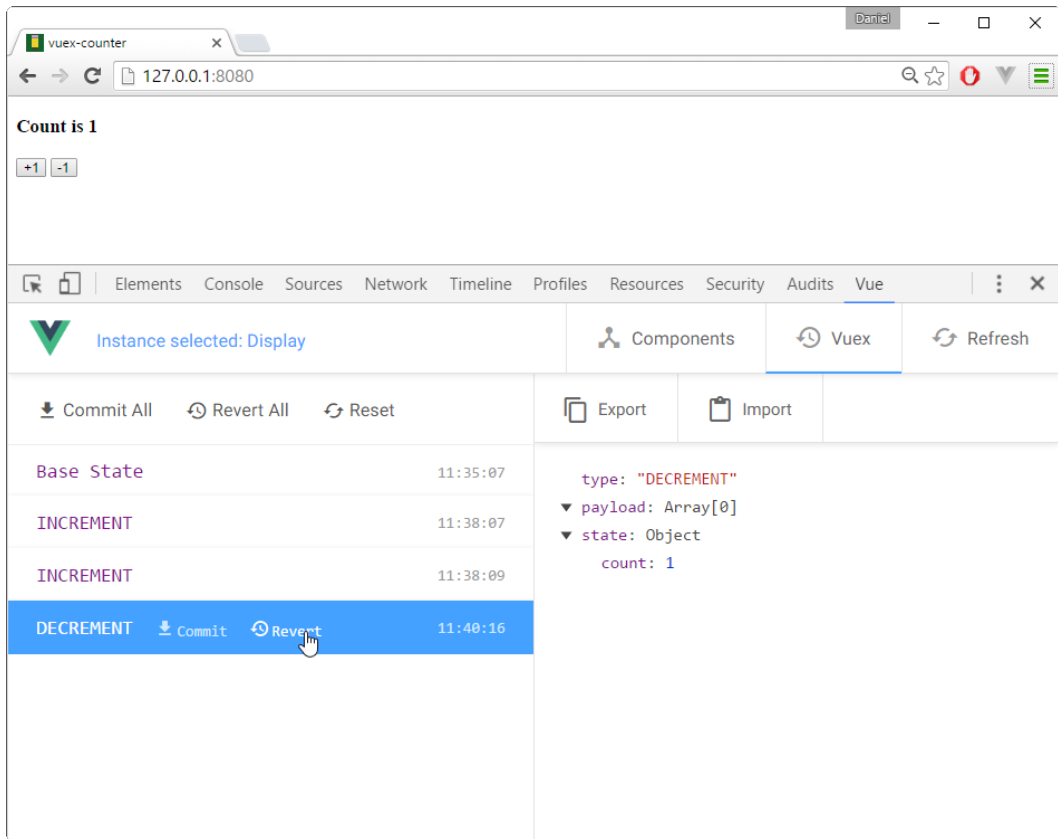
³<https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd>



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.

8.8 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.

```
<template>
  <div>
    <button @click="incrementCounter">+1</button>
    <button @click="decrementCounter">-1</button>
  </div>
  <div>
    <input type="text" v-model="incrementValue">
    <button @click="tryIncrementCounterWithValue">increment\
  </button>
  </div>
</template>
...
```

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:

```
<script>
  import { incrementCounter, decrementCounter, incrementCounterWithValue } from './actions'

  export default {
    vuex: {
      actions: {
        incrementCounter, decrementCounter, incrementCounterWithValue
      }
    }
  }
</script>
```

```
ithValue
    }
  },
  data () {
    return {
      incrementValue:0
    }
  },
  methods: {
    tryIncrementCounterWithValue(){
      this.incrementCounterWithValue(this.incrementValue)
    }
  }
}
</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:

```
export const incrementCounter = function ({ dispatch, state\
}) {
  dispatch('INCREMENT')
}

export const decrementCounter = function ({ dispatch, state\
}) {
  dispatch('DECREMENT')
}
```

```
export const incrementCounterWithValue = function ({ dispatch, state }, value) {  
  dispatch('INCREMENTVALUE', parseInt(value))  
}
```

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:

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

```
Vue.use(Vuex)
```

```
const state = {  
  count: 0  
}
```

```
const mutations = {  
  INCREMENT(state){  
    state.count++;  
  },  
  DECREMENT(state){  
    state.count--;  
  },  
  INCREMENTVALUE(state, value){  
    state.count=state.count+value  
  },  
}
```

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

```
    state,  
    mutations  
  })
```

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`.

8.9 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:

```
...  
export const incrementCounterWithValue = function ({ dispatch,  
ch, state }, value) {  
  
  let intValue = parseInt(value);  
  if (isNaN(intValue)) {  
    throw "Impossível converter para número inteiro"  
  } else {  
    dispatch('INCREMENTVALUE', intValue)  
  }  
}
```

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

```
...
tryIncrementCounterWithValue(){
  try{
    this.incrementCounterWithValue(this.incrementValue)
  } catch (error) {
    alert(error)
  }
}
...

```

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.

8.10 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:

```
export const incrementCounter = function ({ dispatch, state } ) {
  setTimeout(function(){
    dispatch('INCREMENT')
  }, 2000)
}
```

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.

8.11 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:

```
<template>
  <div>
    <button @click="incrementCounter">+1</button>
    <button @click="decrementCounter">-1</button>
  </div>
  <div>
    <input type="text" v-model="incrementValue">
    <button @click="tryIncrementCounterWithValue">increment\
  </button>
  </div>
  <div v-show="waitMessage">
    Aguarde...
  </div>
</template>
```

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

```
...
data () {
  return{
    incrementValue:0,
    waitMessage: false
  }
},
...
```


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:

```
<template>
  ...
  <button @click="tryIncrementCounter">+1</button>
  ...
</template>
<script>
  ...
  methods: {
    tryIncrementCounter(){
      this.waitMessage=true;
      this.incrementCounter();
    },
    tryIncrementCounterWithValue(){
      ...
    }
  }
  ...
</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:

```

export const incrementCounter = function ({ dispatch, state\
}, callback) {
  setTimeout(function(){
    dispatch('INCREMENT')
    callback();
  }, 2000)
}
...

```

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:

```

...
tryIncrementCounter(){
  let t = this;
  this.waitMessage=true;
  this.incrementCounter(function(){
    t.waitMessage=false;
  });
},
...

```

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?

8.12 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:

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

```
Vue.use(Vuex)
```

```
const state = {
  count: 0,
  showWaitMessage: false
}
```

```
const mutations = {
```

```
    INCREMENT(state){
      state.count++;
    },
    DECREMENT(state){
      state.count--;
    },
    INCREMENTVALUE(state,value){
      state.count=state.count+value
    },
    SHOW_WAIT_MESSAGE(state){
      state.showWaitMessage = true;
    },
    HIDE_WAIT_MESSAGE(state){
      state.showWaitMessage = false;
    }
  }
}

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

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

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

export function getShowWaitMessage(state){
  return state.showWaitMessage;
}
```

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

```
export const incrementCounter = function ({ dispatch, state\
}) {
  dispatch('SHOW_WAIT_MESSAGE')
  setTimeout(function(){
    dispatch('HIDE_WAIT_MESSAGE')
    dispatch('INCREMENT')
  },2000)
}
```

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:

```
<template>
  <div>
    <button @click="incrementCounter">+1</button>
    <button @click="decrementCounter">-1</button>
  </div>
  <div>
    <input type="text" v-model="incrementValue">
    <button @click="tryIncrementCounterWithValue">increment\
  </button>
</div>
  <div v-show="getShowWaitMessage">
    Aguarde...
  </div>
</template>

<script>
  import { incrementCounter, decrementCounter, incrementCou\
nterWithValue } from './actions'
```

```
import {getShowWaitMessage} from './getters'

export default {
  vuex: {
    actions: {
      incrementCounter,
      decrementCounter,
      incrementCounterWithValue
    },
    getters: {
      getShowWaitMessage
    }
  },
  data () {
    return{
      incrementValue:0
    }
  },
  methods: {
    tryIncrementCounterWithValue(){
      try{
        this.incrementCounterWithValue(this.incrementValu\
e)
      } catch (error) {
        alert(error)
      }
    }
  }
}
```

</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/danielschmitz/vue-codigos/commit/e9b57115e9075962859cd6965d25c980d1e61cd5)⁴.

8.13 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:

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

Para instalar as bibliotecas vue extras, usamos o npm:

```
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:

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

```
src
|- modules
|- login
|- user
|- product
|- supplier
|- App.vue
|- 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:

```
src
|- modules
|- login
|- index.js
|- user
|- index.js
|- product
|- index.js
|- supplier
|- index.js
|- App.vue
|- main.js
```

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

src/modules/login/index.js

```
const state = {  
  username : "",  
  email : "",  
  token : ""  
}  
  
const mutations = {  
  
}  
  
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

```
const state = {  
  list : [],  
  selected : {}  
}  
  
const mutations = {  
  
}  
  
export default { state, mutations }
```

src/modules/product/index.js

```
const state = {  
  list : [],  
  selected : {}  
}  
  
const mutations = {  
  
}  
  
export default { state, mutations }
```

src/modules/supplier/index.js

```
const state = {  
  list : [],  
  selected : {}  
}  
  
const mutations = {  
  
}  
  
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:

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

import login from './modules/login'
import user from './modules/user'
import product from './modules/product'
import supplier from './modules/supplier'
```

```
Vue.use(Vuex)
```

```
export default new Vuex.Store({
  modules: {
    login,
    user,
    product,
    supplier
  }
})
```

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

src/App.vue

```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
  </div>
</template>

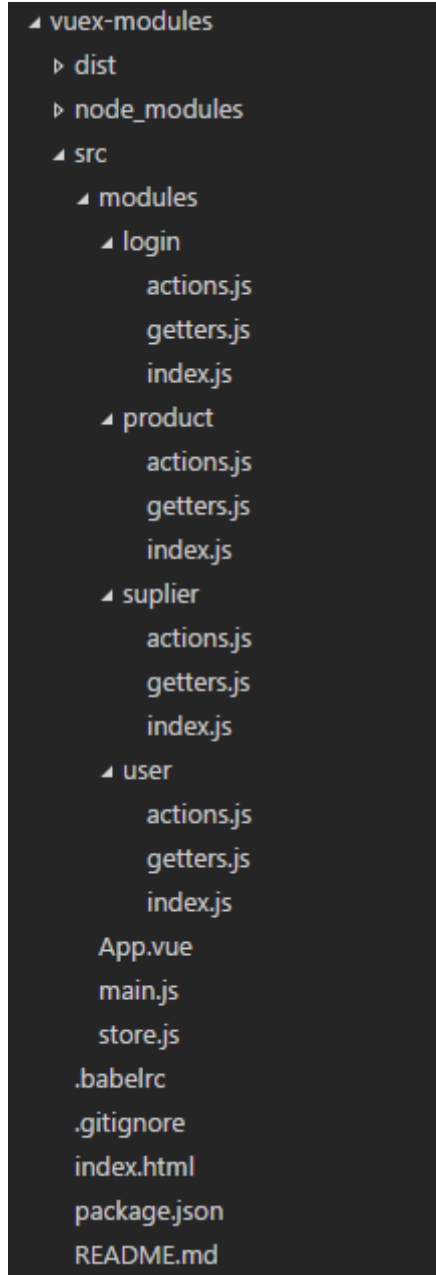
<script>
  import store from './store'

  export default {
    data () {
```

```
      return {
        msg: 'Hello Vue!'
      }
    },
    store
  }
</script>

<style>
  body {
    font-family: Helvetica, sans-serif;
  }
</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

```
{
  "username": "foo",
  "email": "foo@gmail.com",
  "token": "abigtext"
}
```

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

```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
    <button @click="doLogin">Login</button>
  </div>
</template>

<script>
  import store from './store'

  import {doLogin} from './modules/login/actions'

  export default {
    data () {
      return {
        msg: 'Hello Vue!'
      }
    }
  }
}
```

```
    },  
    vuex : {  
      actions : {  
        doLogin  
      }  
    },  
    store  
  }  
</script>
```

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

```
export function doLogin({dispatch}){  
  this.$http.get("/login.json").then(  
    (response)=>{  
      dispatch("SETLOGIN", JSON.parse(response.data))  
    },  
    (error)=>{  
      console.error(error.statusText)  
    }  
  )  
}
```

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

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

import VueResource from 'vue-resource'
```

```
Vue.use(VueResource)
```

```
new Vue({
  el: 'body',
  components: { App }
})
```

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:

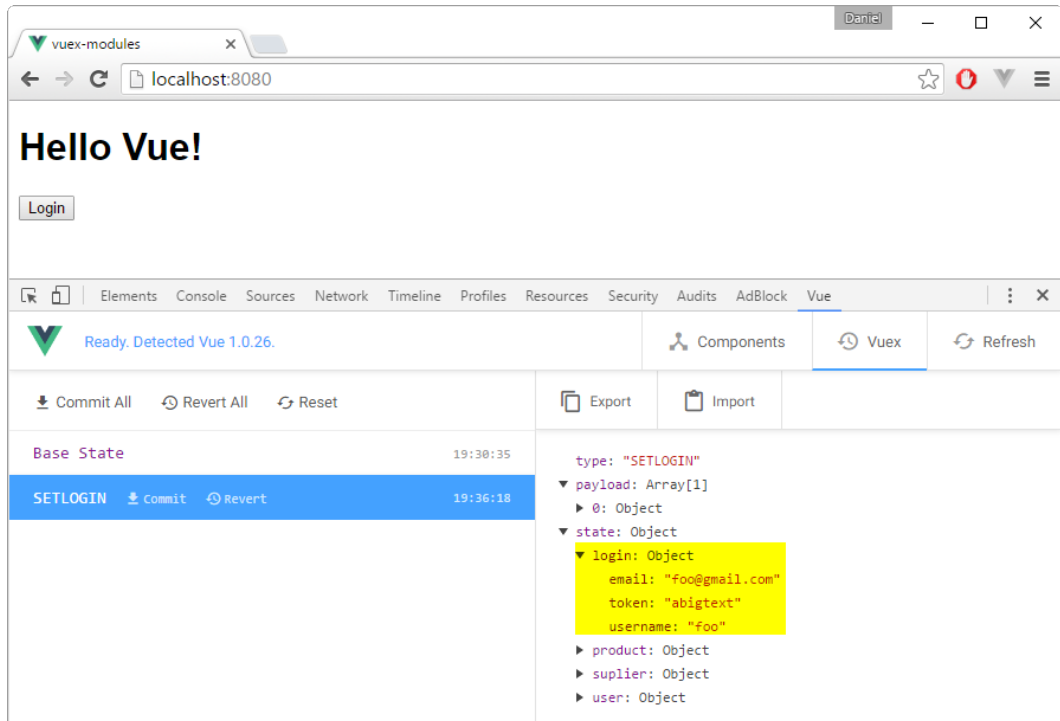
`src/modules/login/index.js`

```
const state = {
  username : "",
  email : "",
  token : ""
}

const mutations = {
  SETLOGIN (state, data) {
    console.log(data)
    state.username = data.username;
    state.email = data.email;
    state.token = data.token;
  }
}

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

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

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

```
export function doLogin({dispatch}){
  this.$http.get("/login.json").then(
    (response)=>{
      dispatch("SETLOGIN", JSON.parse(response.data))
    },
    (error)=>{
      console.error(error.statusText)
    }
  )
}

export function doLogout({dispatch}){
  let login = {
    username : "",
    email : "",
    token : ""
  }
  dispatch("SETLOGIN", login)
}
```

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

```
<template>
  <div id="app">
    <button v-show="!isLoggedIn" @click="doLogin">Login</button>
    <div v-show="isLoggedIn">
      <button @click="doLogout"> Logout</button>
      <button > Users</button>
      <button > Products</button>
      <button > Suppliers</button>
    </div>
  </div>
</template>
```

```
<script>
  import store from './store'

  import {doLogin,doLogout} from './modules/login/actions'
  import {isLoggedIn} from './modules/login/getters'

  export default {
    data () {
      return {
        msg: 'Hello Vue!'
      }
    },
    vuex :{
      actions :{
        doLogin,doLogout
      },
      getters : {
        isLoggedIn
      }
    }
  }
}
```

```
    },  
    store  
  }  
</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

```
[  
{  
  "id": 1,  
  "name": "product1",  
  "quantity": 200  
},  
{  
  "id": 2,  
  "name": "product2",  
  "quantity": 100  
},  
]
```

```
{
  "id": 3,
  "name": "product3",
  "quantity": 50
},
{
  "id": 4,
  "name": "product4",
  "quantity": 10
},
{
  "id": 5,
  "name": "product5",
  "quantity": 20
},
{
  "id": 6,
  "name": "product6",
  "quantity": 300
}
]
```

src/modules/product/actions.js

```
export function loadProducts({dispatch}){
  this.$http.get("/products.json").then(
    (response)=>{
      dispatch("SETPRODUCTS", JSON.parse(response.data))
    },
    (error)=>{
      console.error(error.statusText)
    }
  )
}
```

```
)  
}
```

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`

```
const state = {  
  list : [],  
  selected : {}  
}  
  
const mutations = {  
  SETPRODUCTS(state, data) {  
    state.list = data;  
  },  
}  
  
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`

```
export function getProducts(state){  
  return state.product.list;  
}  
  
export function hasProducts(state){  
  return state.product.list.length>0  
}
```

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

```
<template>
  <div id="app">
    <button v-show="!isLoggedIn" @click="doLogin">Login</butt\
on>
    <div v-show="isLoggedIn">
      <button @click="doLogout"> Logout</button>
      <button > Users</button>
      <button @click="loadProducts">Products</button>
      <button > Suppliers</button>

      <br>

      <div v-show="hasProducts">
        <h4>Products</h4>
        <table border="1">
          <thead>
            <tr>
              <td>id</td>
              <td>name</td>
              <td>quantity</td>
            </tr>
          </thead>
          <tbody>
            <tr v-for="p in getProducts">
              <td>{{p.id}}</td>
              <td>{{p.name}}</td>
              <td>{{p.quantity}}</td>
            </tr>
          </tbody>
        </table>
      </div>
    </div>
  </div>
</template>
```

```
        </tbody>
      </table>
    </div>

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

<script>
  import store from './store'
  import {doLogin,doLogout} from './modules/login/actions'
  import {loadProducts} from './modules/product/actions'
  import {isLoggedIn} from './modules/login/getters'
  import {hasProducts,getProducts} from './modules/product/\
getters'

  export default {
    data () {
      return {
        msg: 'Hello Vue!'
      }
    },
    vuex :{
      actions :{
        doLogin,doLogout,
        loadProducts
      },
      getters : {
        isLoggedIn,
        hasProducts,getProducts
      }
    },
    store
```



```
    }  
</script>  
  
<style>  
  body {  
    font-family: Helvetica, sans-serif;  
  }  
</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

9. 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.

9.1 Criando mixins

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

```
vue init browserify-simple#1 vue-mixin
cd vue-mixin
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

```
export const eventMixin = {  
  ``js  
    created: function(){  
      console.log(`[${this.$options.name}] created`);  
    }  
}
```

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

src/Comp1.vue

```
<template>Comp 1</template>  
<script>  
import {eventMixin} from './eventMixin'  
  
export default{  
  mixins: [eventMixin]  
}  
</script>
```

src/Comp2.vue

```
<template>Comp 2</template>  
<script>  
import {eventMixin} from './eventMixin'  
  
export default{  
  mixins: [eventMixin]  
}  
</script>
```

src/Comp3.vue

```
<template>Comp 3</template>
<script>
import {eventMixin} from './eventMixin'

export default{
  mixins: [eventMixin]
}
</script>
```

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

src/App.vue

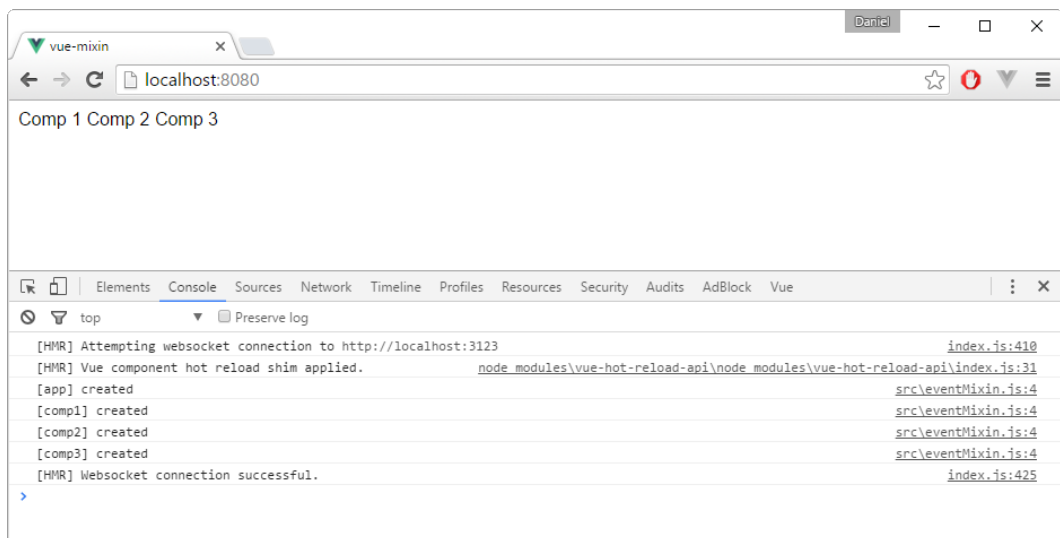
```
<template>
  <div id="app">
    <comp1></comp1>
    <comp2></comp2>
    <comp3></comp3>
  </div>
</template>

<script>
import Comp1 from './Comp1.vue'
import Comp2 from './Comp2.vue'
import Comp3 from './Comp3.vue'
import {eventMixin} from './eventMixin'

export default {
  components :{
    Comp1,Comp2,Comp3
  },
  data () {
```

```
    return {  
      msg: 'Hello Vue!'  
    },  
    mixins: [eventMixin]  
  }  
</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.

9.2 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:

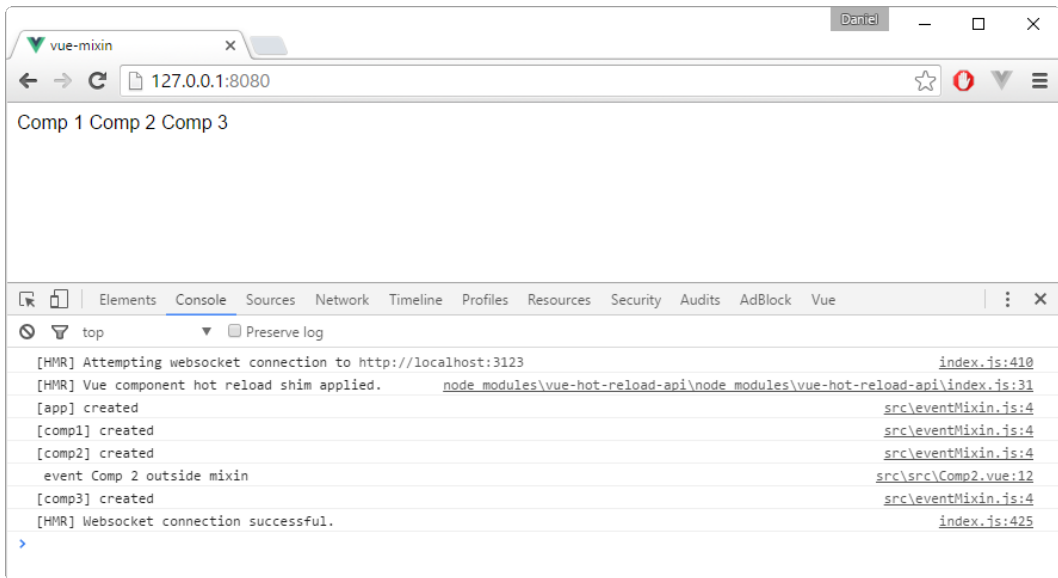
src/Comp2.vue

```
<template>Comp 2</template>
<script>
import {eventMixin} from './eventMixin'

export default{
  mixins: [eventMixin],
  created : function() {
    console.log(" event Comp 2 outside mixin ")
  }
}
</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.

10. 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

10.1 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:

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

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.


```
vue init browserify-simple#1 plugin-real
cd plugin-real
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:

```
module.exports = {
  install: function (Vue, options) {

    Vue.filter('real', function() {
      var tmp = arguments[0]+'';
      tmp = tmp.replace(/([0-9]{2})$/g, ",$1");
      if( tmp.length > 6 )
        tmp = tmp.replace(/([0-9]{3}),([0-9]{2})$/g, ".$1,$2");
      return 'R$ ' + tmp;
    })
  }
}
```

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](http://wbruno.com.br/expressao-regular/formatar-em-moeda-reais-expressao-regular-em-javascript/)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 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

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

import real from '../real'
Vue.use(real)

new Vue({
  el: 'body',
  components: { App }
})
```

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

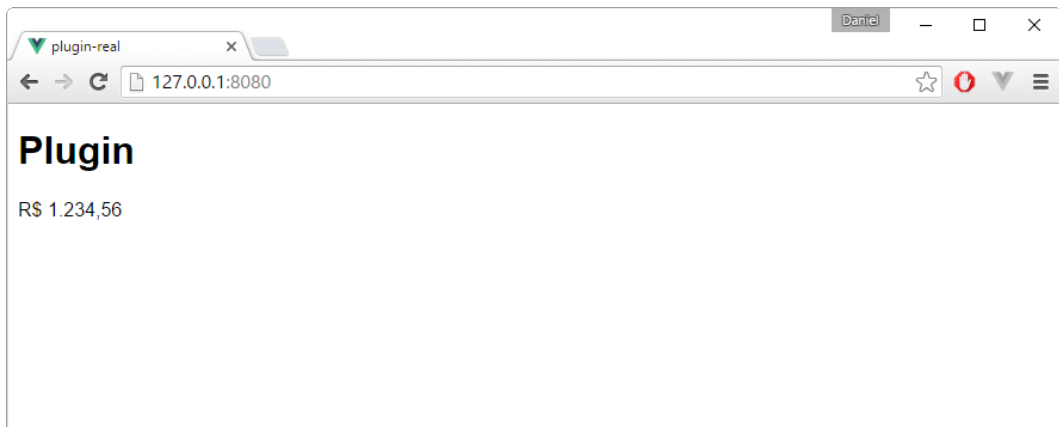
src/main.js

```
<template>
  <div id="app">
    <h1>Plugin</h1>
    <p>{{ valor | real }}
  </div>

</template>
```

```
<script>
export default {
  data () {
    return {
      valor: 123456
    }
  }
}
</script>
```

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



Parte 4 - Criando um sistema com Vue e Php

11. Preparando o ambiente

Neste capítulo abordaremos como instalar o servidor Apache/Php/MySQL tanto para Linux quanto para Windows. Caso já tenha o servidor pronto, com Apache+php+MySQL+Composer, pode ir direto ao capítulo “Criando o domínio virtual (virtual host)”.

11.1 Preparando o servidor web

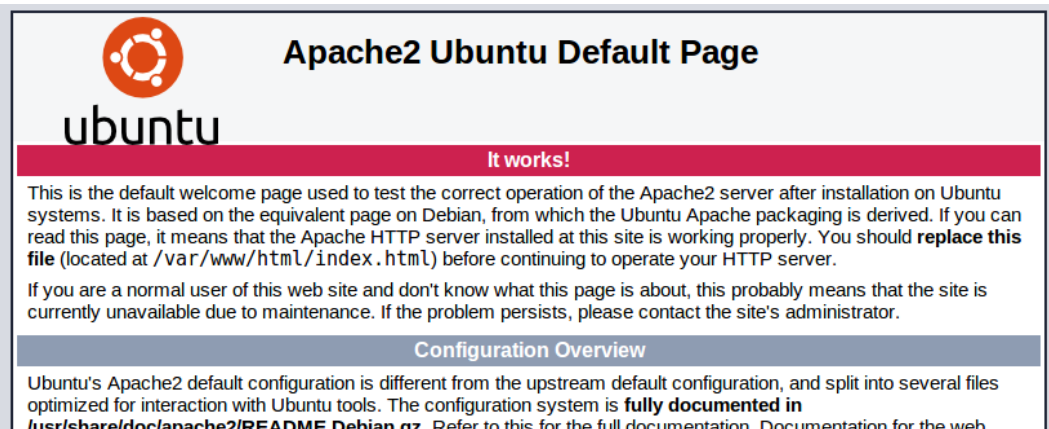
Dividiremos esta instalação em duas partes, uma para o sistema operacional Linux, rodando a distribuição Ubuntu 14.04, e outra para a versão Windows 10.

11.2 Apache no Linux

Utilizando a distribuição Ubuntu, na versão 14.04, iremos instalar o Apache através do gerenciador de pacotes apt-get, muito conhecido nesta distribuição. Para que possamos instalar o apache, abra um console (ou terminal, chame como quiser) e digite o seguinte comando:

```
$ sudo apt-get install apache2
```

Este comando irá instalar o Apache no seu sistema, e é fundamental que você conheça um pouco de Linux para seguir em frente. Após a instalação, o servidor estará funcionando e você poderá testar o apache ao apontar o seu navegador para o endereço `http://localhost`, tendo a seguinte resposta:



Esta página está localizada em `/var/www/html/index.html`, ou seja, o documento raiz do apache está localizado neste diretório, que é um diretório do sistema operacional, fora da área de acesso da sua conta. Neste caso, ou você cria as páginas HTML neste diretório como *root*, ou configura mais um pouco o Apache para utilizar a sua própria área.

Contas no Linux

Você que usa o Linux no dia a dia sabe que existe uma conta de usuário no sistema, e você trabalha em seus projetos nessa conta. Nesta obra, estaremos repetindo esse processo, ou seja, utilizaremos a conta linux que está localizada no diretório `/home` para fornecer informações sobre configuração e acesso.

No nosso caso em específico, estaremos utilizando o caminho `/home/user`, mas lembre-se de trocar esse caminho para o nome da sua conta.

No Apache temos uma configuração chamada `HomeDir`. Ela funciona da seguinte forma: ao acessar `http://localhost/~user` o apache irá apontar para `/home/user/public_html`. Desta forma, estaremos criando código HTML/PHP dentro do diretório pessoal da conta e não no diretório do sistema `/var/www`. Para habilitar esta configuração, deve-se realizar os seguintes procedimentos:

```
$ cd /etc/apache2/mods-enabled
$ sudo ln -s ../mods-available/userdir.conf userdir.conf
$ sudo ln -s ../mods-available/userdir.load userdir.load
$ sudo /etc/init.d/apache2 restart
```

Após executar estes comandos, e reiniciar o Apache, crie um diretório chamado `public_html` na sua conta (em `/home/user/`) e crie uma página `index.html` com um simples `Hello World` nela. Aponte o seu navegador para `http://localhost/~user` e veja a página entrar em ação.

Para terminar de configurar o Apache, precisamos habilitar um recurso chamado `Rewrite` que será muito importante no Slim Framework. Basicamente, o `Rewrite` permite que urls como `www.site.com/user/1` sejam convertidas em algo como `www.site.com/index.php?item1=user&item2=1`. Ou seja, ele faz uma reescrita da URL (daí o nome `Rewrite`). Configurar este recurso é muito fácil, bastando apenas executar os seguintes comandos:

```
$ sudo a2enmod rewrite
$ sudo service apache2 restart
```

11.3 Instalando o PHP no Linux

Para instalar o PHP no Linux, basta executar o seguinte comando:

```
$ sudo apt-get install php5 php-pear libapache2-mod-php5
$ sudo a2enmod php5
```

Isso fará com que o módulo do `php5` seja instalado no `apache`, além de instalar o `php5` no sistema. É necessário realizar uma configuração extra no Apache para que o `php` funcione no diretório `public_html`, que consiste em editar o arquivo `/etc/apache2/mods-available/php5.conf` através do seguinte comando:

```
$ sudo gedit /etc/apache2/mods-available/php5.conf
```

Neste caso, gedit é um editor de textos simples para o ubuntu. No arquivo que abre, deve-se encontrar a linha `php_admin_flag engine Off` e comentá-la, deixando o código da seguinte forma:

```
</FilesMatch>

# Running PHP scripts in user directories is disabled by default
#
# To re-enable PHP in user directories comment the following lines
# (from <IfModule ...> to </IfModule>.) Do NOT set it to On as it
# prevents .htaccess files from disabling it.
#<IfModule mod_userdir.c>
#     <Directory /home/*/public_html>
#         php_admin_value engine Off
#     </Directory>
#</IfModule>
```

Esta linha não permite que scripts php sejam executadas em diretórios do usuário, o que melhora a segurança do server. Mas como queremos juntamente o contrário, pois estaremos em um ambiente de testes, devemos comentá-la para permitir executar scripts php no nosso `public_html`.

11.4 Instalando o MySQL no Linux

O MySQL é instalado da mesma forma que os outros programas que instalamos através do `apt-get`. O comando para a instalar o MySQL é

```
$ sudo apt-get install mysql-server
```

Durante a instalação, surgirá uma tela requisitando uma senha para o administrador do banco, o usuário `root`. Deixe-a em branco, pois estamos criando um banco de testes somente. Após a instalação do MySQL, podemos testá-lo através do seguinte comando:


```
$ mysql -u root
```

Após digitar este comando, você irá logar no console do mysql e digitar comandos em sql para acessar tabelas e dados. Por exemplo, o seguinte comando:

```
mysql> show databases;
```

Produzirá o seguinte resultado:

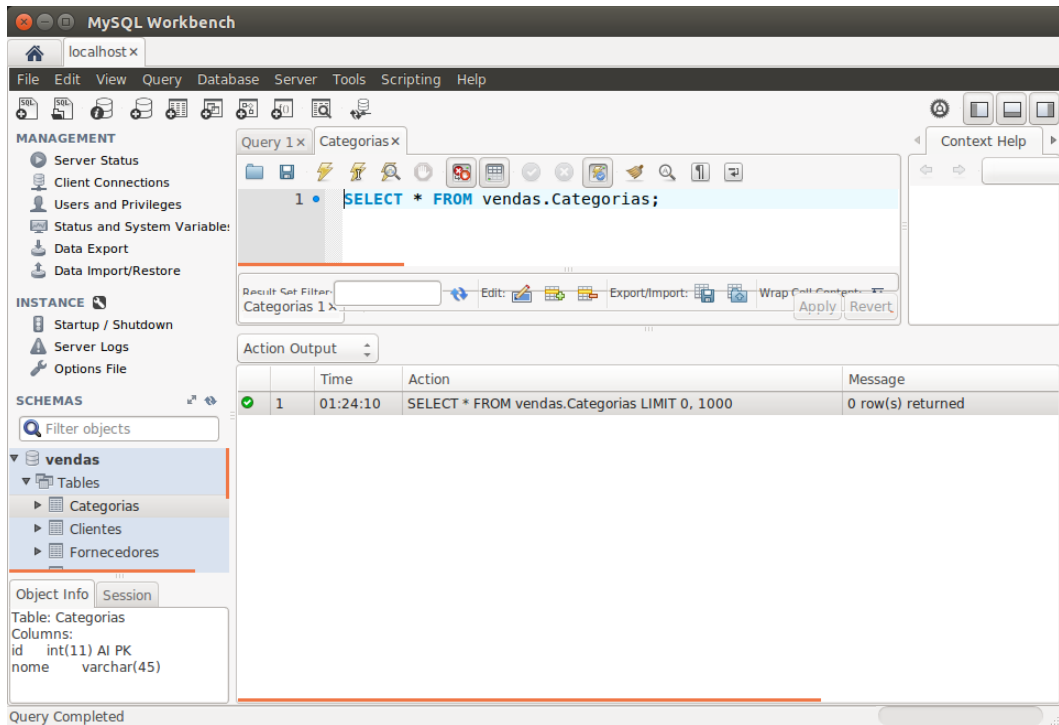
```
+-----+  
| Database          |  
+-----+  
| information_schema |  
| mysql             |  
| performance_schema |  
+-----+  
4 rows in set (0.02 sec)
```

11.5 MySql Workbench para linux

É claro que não queremos utilizar o console do MySql para trabalhar com tabelas e dados do sistema. Para isso temos um cliente gráfico poderoso e gratuito, chamado MySql Workbench. Para instalá-lo, use novamente o apt-get conforme o comando a seguir:

```
$ sudo apt-get install mysql-workbench mysql-workbench-data
```

O Workbench pode ser aberto na interface gráfica do Ubuntu, conforme a figura a seguir:



11.6 Instalando o Composer no linux

O Composer é um gerenciador de pacotes a nível de projeto, e não de sistema. Usamos o composer para instalar o Slim Framework (e qualquer outro framework php que precisar), de forma semelhante ao Linux. Isso significa que, ao invés de baixarmos o pacote com o Slim Framework, iremos utilizar o Composer para que ele mesmo faça o download e a instalação do mesmo.

No linux, o composer será instalado a nível de projeto, significando que temos que repetir este processo em todos os projetos que criamos. Até podemos instalar o composer a nível do sistema, mas estaríamos fugindo um pouco do nosso escopo. Como estamos no linux, uma linha digitada a mais no console não será problema algum.

O Composer necessita do *curl* para ser instalado. Se o seu sistema linux ainda não tem essa ferramenta, é hora de instalá-la:

```
$ sudo apt-get install curl
```

Após instalar o curl, podemos instalar o composer no diretório onde criamos o nosso primeiro projeto Slim, que será em `/home/user/public_html/helloworld`. Lembre-se que **user** é o seu usuário.

```
$ mkdir /home/user/public_html/helloworld
$ cd /home/user/public_html/helloworld
$ curl -s https://getcomposer.org/installer | php
```

Neste momento será criado um arquivo `composer.phar` no diretório `helloworld`. Este arquivo é uma espécie de *executável* do composer. Para instalar o Slim, execute o seguinte comando:

```
php composer.phar require slim/slim "^3.0"
```

Através deste comando, o Slim Framework será instalado na pasta `vendor`, e o arquivo `composer.json` que também foi criado possuirá a seguinte configuração:

```
{
  "require": {
    "slim/slim": "^3.0"
  }
}
```

11.7 Testando o Slim no Linux

Iremos aproveitar o tópico anterior e continuar com o `helloworld` criando uma aplicação bem simples com o Slim Framework. Também faremos o mesmo na instalação para Windows, de forma a mostrar algumas pequenas variações. Primeiramente, deve-se criar o arquivo `.htaccess` na raiz do projeto, da seguinte forma:

```
$ touch .htaccess
```

E adicionar o seguinte conteúdo:

```
RewriteEngine On
RewriteBase /~user/helloworld

RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [QSA,L]
```

Lembre-se que, ao invés de *user*, coloque o seu usuário do Linux. Após a configuração do rewrite, podemos certificar que, quando um endereço url do tipo `http://localhost/~user/helloworld/user` será convertido para `http://localhost/~user/h`

Para testarmos se está tudo ok até este momento, basta apontar o seu navegador para o endereço `http://localhost/~user/helloworld/` e encontrar a mensagem de erro *Not Found - The requested URL /~user/helloworld/index.php was not found on this server.*

Agora vamos criar o arquivo `index.php` que é o arquivo principal do nosso sistema de teste. Ele conterá o Slim e será sempre o arquivo inicial acessado, independente da URL. Chamamos isso de *entry point* da aplicação.

`index.php`

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Resp\
onse $response) {
    $name = $request->getAttribute('name');
    $response->getBody()->write("Hello, $name");
```

```
    return $response;  
});  
$app->run();
```

Primeiro importamos as classes usadas no arquivo, que são `ServerRequestInterface` e `ResponseInterface`, que possuem o apelido de `Request` e `Response`. O uso do `use` atende aos requisitos do PSR-7 do PHP, e o Slim Framework possui suporte a ele.

Depois, incluímos o `autoload.php` gerado pelo Composer. Ele irá se encarregar de adicionar o necessário para que o Slim funcione, assim como qualquer outra biblioteca.

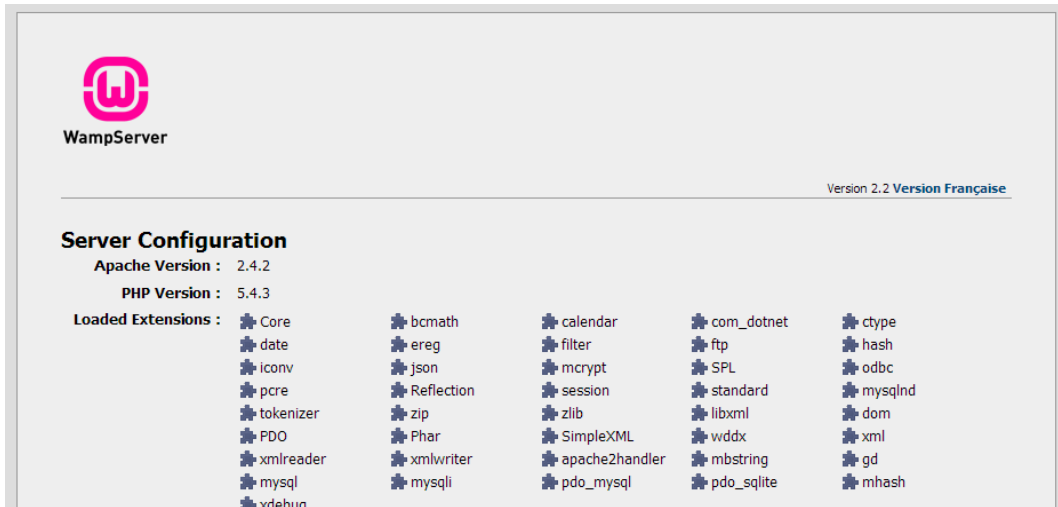
A variável `$app` representa a classe `Slim`, onde criamos uma entrada HTTP com o método GET. Esta entrada possui dois parâmetros. O primeiro é o caminho que o Slim deve comparar com o caminho digitado no navegador. Neste caso, usamos `/hello/{name}`, ou seja, o método está sendo configurado para esperar algo como `localhost/helloworld/hello/joe`, onde *joe* será atribuído à variável `name`. Através do `$response->getBody()->write` uma mensagem de retorno será enviada ao cliente, repassando inclusive o nome atribuído a variável `name`.

11.8 Instalando o Apache/php/MySQL no Windows

Se você utilizando Windows, existe um programa chamado Wamp Server que instala automaticamente todos os programas necessários para executar páginas em PHP e acessar o banco de dados.

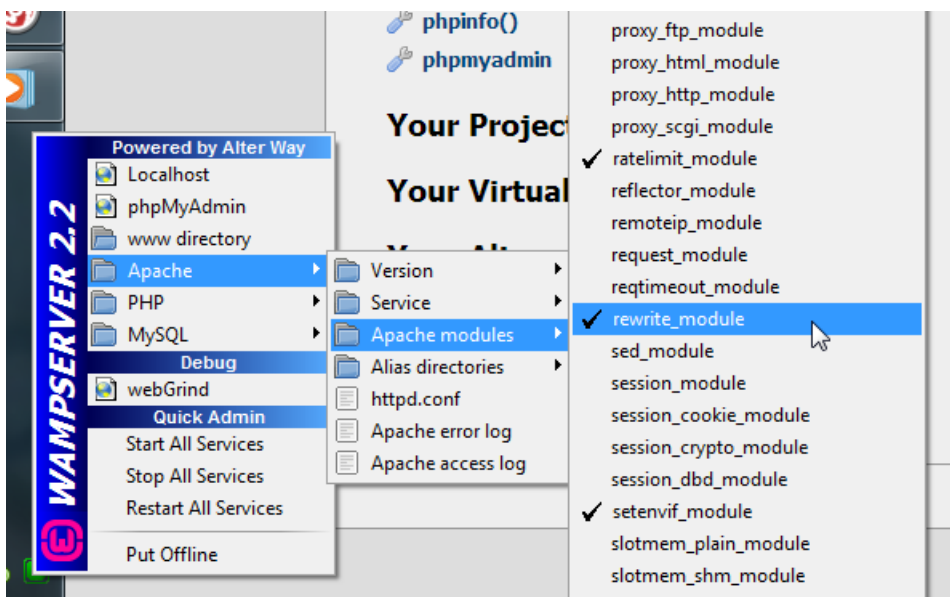
Para instalá-lo, acesse <http://www.wampserver.com/en/>¹ e clique em Download, e faça o download de acordo com a versão do seu sistema operacional (32 ou 64 bits). Após realizar o download e instalar o Wamp Server, o diretório raiz do servidor será `c:\wamp\www` que pode ser acessado através da url `http://localhost/`, de acordo com a imagem a seguir.

¹<http://www.wampserver.com/en/>



Caso a url não esteja acessível, não esqueça de iniciar o serviço do Wamp Server, executando o arquivo `c:\wamp\wampmanager.exe`.

Para finalizar a instalação do Wamp, é necessário habilitar a extensão `rewrite_module`, e isso é realizado acessando o menu do wamp pelo seu ícone na bandeja do Windows, e navegando até ativar o item `rewrite_module` de acordo com a figura a seguir.



Essa configuração é necessária para que o Slim possa funcionar. Basicamente, o Rewrite permite que urls como `www.site.com/user/1` sejam convertidas em algo como `www.site.com/index.php?item1=user&item2=1`. Ou seja, ele faz uma reescrita da URL (daí o nome Rewrite).

11.9 Instalando o MySQL Workbench no Windows

O MySQL Workbench é um editor visual para o banco de dados MySQL. Para instalá-lo, acesse <http://www.mysql.com/products/workbench/>² e clique em Download Now. Faça o download e instale. Para acesso ao MySQL, o Wamp usa como padrão o usuário root e a senha deixe em branco.

11.10 Instalando o Composer no Windows

O composer é um gerenciador de pacotes a nível de projeto, e não de sistema. Usamos o composer para instalar o Slim Framework (e qualquer outro que precisar) de forma semelhante ao Linux.

²<http://www.mysql.com/products/workbench/>

Antes de instalar o Composer, é necessário realizar uma mudança no arquivo `php.ini` do Wamp Server, para habilitar o `openssl`. Você deve fazer isso em 2 arquivos:

```
c:\wamp\bin\php\php5.X.X\php.ini
```

```
c:\wamp\bin\apache\Apache2.X.X\bin\php.ini
```

Nestes dois arquivos, localize o código:

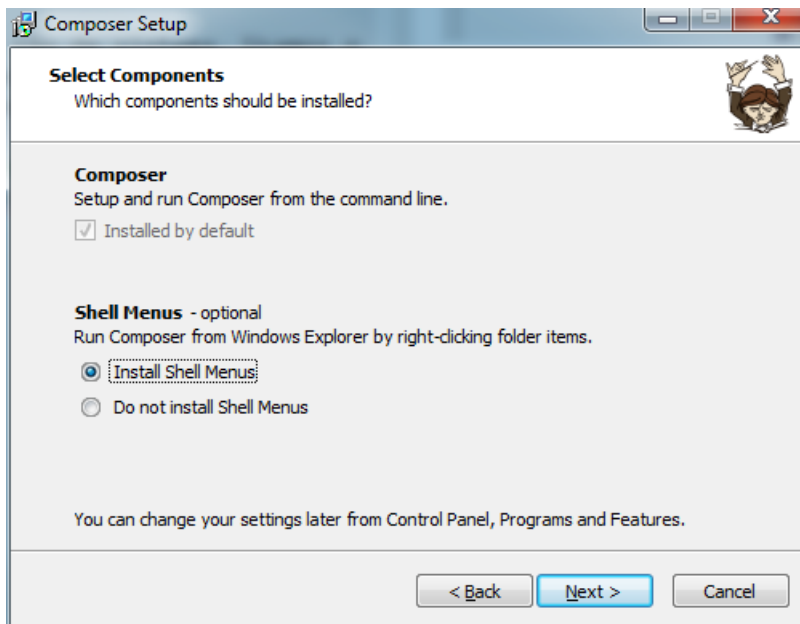
```
;extension=php_openssl.dll
```

e remova o comentário “;” deixando esta linha da seguinte forma:

```
extension=php_openssl.dll
```

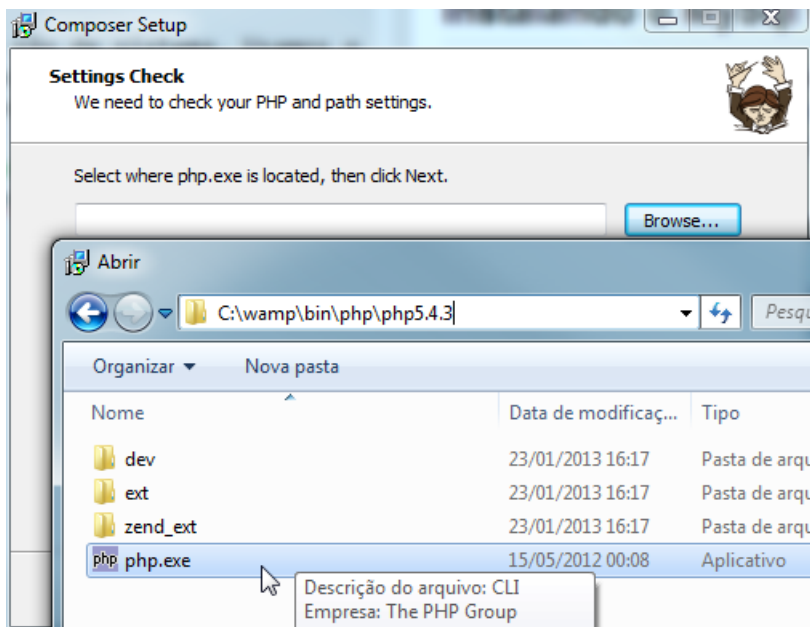
Após esta alteração, reinicie o Wamp Server, pelo seu ícone na bandeja do windows, acessando o item `Restart All Services`.

Para instalar o composer no Windows, acesse <https://getcomposer.org/download/>³ e faça o download do `Composer-Setup.exe` na seção `Windows Installer`. Durante a instalação, escolha o item ‘`Install Shell Menus`’ de acordo com a figura a seguir.



³<https://getcomposer.org/download/>

Na próxima tela, o instalador do Composer lhe pede o caminho do arquivo `php.exe`, que está em `c:\wamp\bin\php\php5.4.3`



Após finalizar a instalação, o composer deve estar pronto para uso. Como selecionamos a opção “Install Shell Menus”, ela deve estar disponível no menu de contexto do Explorer.

11.11 Testando o Slim no Windows

Agora que temos tudo que precisamos para utilizar o Slim Framework, vamos criar um pequeno “Hello World” no ambiente windows. Primeiro, crie a pasta `c:\wamp\www\helloworld` e nela, iremos preparar a instalação do Slim Framework.

Para instalar o slim, execute o seguinte comando:

```
composer require slim/slim "^3.0"
```

Através deste comando, o Slim Framework será instalado na pasta `vendor`, e o arquivo `composer.json` que também foi criado possuirá a seguinte configuração:

```
{  
  "require": {  
    "slim/slim": "^3.0"  
  }  
}
```

Iremos aproveitar o tópico anterior e continuar com o helloworld criando uma aplicação bem simples com o Slim Framework. Primeiramente, deve-se criar o arquivo `.htaccess` na raiz do projeto com o seguinte conteúdo:

```
RewriteEngine On  
RewriteBase /  
  
RewriteCond %{REQUEST_FILENAME} !-f  
RewriteRule ^ index.php [QSA,L]
```

Após a configuração do rewrite, podemos certificar que, quando um endereço url do tipo `http://localhost/helloworld/user` será convertido para `http://localhost/helloworld/`

Para testarmos se está tudo ok até este momento, basta apontar o seu navegador para o endereço `http://localhost/helloworld/` e encontrar a mensagem de erro *Not Found - The requested URL /~user/helloworld/index.php was not found on this server.*

Agora vamos criar o arquivo `index.php` que é o arquivo principal do nosso sistema de teste. Ele conterá o Slim e será sempre o arquivo inicial acessado, independente da URL. Chamamos isso de *entry point* da aplicação.

index.php

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Resp\
onse $response) {
    $name = $request->getAttribute('name');
    $response->getBody()->write("Hello, $name");

    return $response;
});
$app->run();
```

Primeiro importamos as classes usadas no arquivo, que são `ServerRequestInterface` e `ResponseInterface`, que possuem o apelido de `Request` e `Response`. O uso do `use` atende aos requisitos do PSR-7 do PHP, e o Slim Framework possui suporte a ele.

Depois, incluímos o `autoload.php` gerado pelo Composer. Ele irá se encarregar de adicionar o necessário para que o Slim funcione, assim como qualquer outra biblioteca.

A variável `$app` representa a classe `Slim`, onde criamos uma entrada HTTP com o método GET. Esta entrada possui dois parâmetros. O primeiro é o caminho que o Slim deve comparar com o caminho digitado no navegador. Neste caso, usamos `/hello/{name}`, ou seja, o método está sendo configurado para esperar algo como `localhost/helloworld/hello/joe`, onde *joe* será atribuído à variável `name`. Através do `$response->getBody()->write` uma mensagem de retorno será enviada ao cliente, repassando inclusive o nome atribuído a variável `name`.

11.11.1 Criando o domínio virtual (virtual host)

É muito útil criar um domínio virtual para cada sistema que desenvolvemos. Através de um domínio virtual podemos facilmente “simular” um endereço de web no próprio computador, sem a necessidade de configurar servidores externos. Como exemplo, iremos configurar o domínio `mysite.dev`, fazendo com que ao digitarmos `www.mysite.dev`, o Apache irá apontar para a pasta `c:\wamp\www\mysite` ou para `/home/user/mysite`.

Inicialmente, crie uma pasta chamada `mysite` em `c:\wamp\www` (ou `/home/user/`), e adicione o arquivo `index.html`, com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>MySite</title>
</head>

<body>
  This is my site !
</body>
</html>
```

Agora devemos adicionar uma configuração no arquivo `httpd.conf` do Apache. Este arquivo está localizado em `C:\wamp\bin\apache\apache2.4.9\conf`. Abra-o e adicione a seguinte configuração no final do arquivo:

```
<Directory "c:/wamp/www/">
  Options Indexes FollowSymLinks
  AllowOverride all
  Require local
</Directory>

<<<<<<< HEAD
<VirtualHost mysite.dev:80>
  ServerAdmin your@email.dev
  ServerName mysite.dev
  ServerAlias mysite.dev
  DocumentRoot c:/wamp/www/mysite
=====
<VirtualHost sales.dev:80>
  ServerAdmin your@email.dev
  ServerName mysite.dev
  ServerAlias mysite.dev
  DocumentRoot c:/wamp/www/mysite
>>>>>> ce093dc9d3955d9788fda6be649f552edaf27a4f
</VirtualHost>
```

Esta configuração cria o domínio virtual no apache, apontando diretamente para `c:/wamp/www/mysite` (ou `/home/user/mysite` - não esqueça de alterar os caminhos do virtual host também).

Após criar o virtual host, é preciso configurar o diretório que irá receber o virtual host, adicionando a seguinte entrada logo após a criação do Virtual Host:

```
<Directory "c:/wamp/www/">
  Options Indexes FollowSymLinks
  AllowOverride all
  Require local
</Directory>
```

É necessário reiniciar o apache para que a configuração seja recarregada. Para reiniciar, clique com o botão esquerdo do mouse no ícone do WampServer, na bandeja do Windows, e escolha a opção Restart All Services. O ícone ficará vermelho por um tempo e depois voltará a ficar verde.

Para finalizar a configuração, é necessário alterar o arquivo `hosts` do Windows. Este arquivo fica localizado em `C:\Windows\System32\drivers\etc` (ou `/etc/hosts` no linux). Para que você possa editá-lo, é necessário abrí-lo como administrador do sistema (ou use `sudo` no linux). A melhor forma de fazer isso é abrir o Notepad como administrador (botão direito do mouse no ícone do Notepad e selecione Executar como administrador). Após abrir o Notepad, abra o arquivo `C:\Windows\System32\drivers\etc\hosts`, que deve ser semelhante a imagem a seguir.

```
127.0.0.1      localhost
```

Adicione no final do arquivo a seguinte instrução:

```
127.0.0.1     mysite.dev
```

Após salvar o arquivo `hosts`, abra o navegador e acesse o seguinte endereço: `mysite.dev` para ver o arquivo `html` que criamos com o texto `This is my site !`.

12. Banco de dados

12.1 Importando o banco de dados

O banco de dados do sistema de vendas está disponível no diretório [sales-server do github](https://github.com/danielschmitz/vue-codigos/blob/master/sales-server/database.sql)¹

Após realizar o download do arquivo [database.sql](https://raw.githubusercontent.com/danielschmitz/vue-codigos/master/sales-server/database.sql)², você pode importá-lo no mysql através das seguintes formas:

Pela linha de comando Use o comando `mysql -u root < database.sql` para importar o banco de dados. Se o comando `mysql` não estiver disponível na linha de comando, é preciso adicionar o `mysql` nas variáveis de ambiente `PATH` do sistema.

Pelo phpmyadmin Se estiver utilizando o Wamp Server, pode-se acessar “<http://localhost>” e clicar no link “Phpmyadmin”, em “Tools”. Após acessar o PhpMyAdmin, clique no item “Importar” e escolha o arquivo “`database.sql`” e clique em importar.

Pelo Workbench Se você abrir o arquivo [database.mwb](https://github.com/danielschmitz/vue-codigos/raw/master/sales-server/mysql-database.mwb)³ diretamente pelo Workbench, pode usar a funcionalidade “Database > Forward engineer” para atualizar o banco de dados.

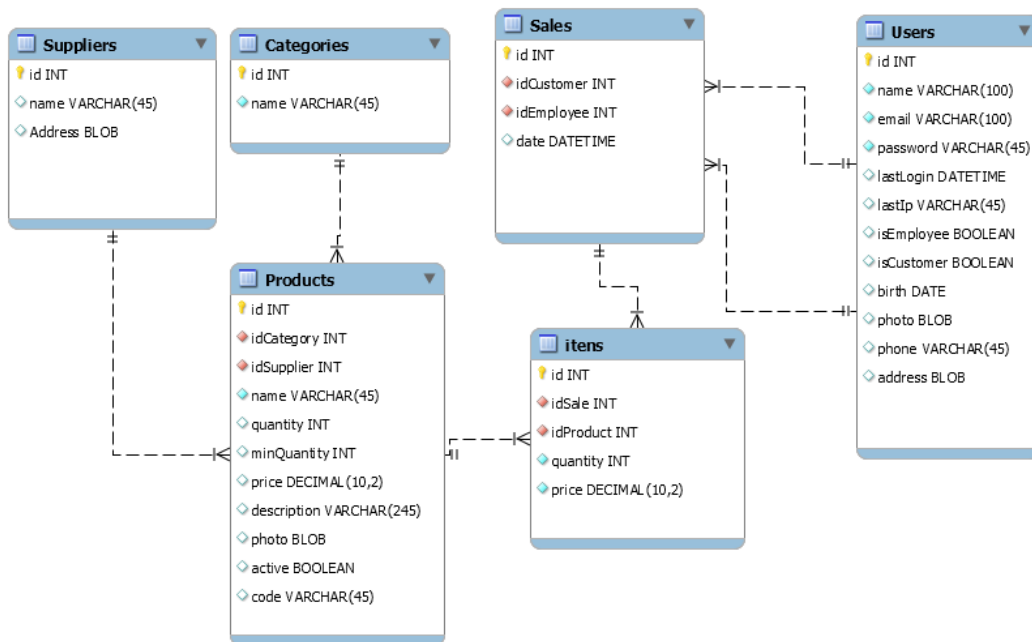
Em todas estas situações, o banco de dados será recriado inicialmente vazio.

¹<https://github.com/danielschmitz/vue-codigos/blob/master/sales-server/database.sql>

²<https://raw.githubusercontent.com/danielschmitz/vue-codigos/master/sales-server/database.sql>

³<https://github.com/danielschmitz/vue-codigos/raw/master/sales-server/mysql-database.mwb>

12.2 Conhecendo as tabelas



13. Criando o servidor PHP/Slim

O servidor PHP/Slim será criado com o objeto de fornecer uma API ao cliente que irá consumi-la. Vamos separar os projetos em *sales-server* e *sales-client*.

13.1 Criando o diretório do servidor

Crie o diretório *sales-server* no seu sistema. No terminal, acesse o diretório e instale o Slim Framework:

```
$ composer require slim/slim "^3.0"
```

Após a instalação pelo composer, o Slim estará pronto para uso. Para testar, vamos criar o arquivo *index.php* (crie a pasta *public* também), com o seguinte código:

public/index.php

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/', function (Request $request, Response $response) {
    $response->getBody()->write("Hello World");

    return $response;
});
```

```
$app->run();
```

Crie o virtual host:

httpd.conf

```
<Directory "c:/path/to/sales/">
    Options Indexes FollowSymLinks
    AllowOverride all
    Require local
</Directory>

<VirtualHost sales.dev:80>
    ServerAdmin your@email.dev
    ServerName sales.dev
    ServerAlias sales.dev
    DocumentRoot C:\path\to\sales-server\public
    ErrorLog C:\path\to\sales-server\sales.error.log
    CustomLog C:\path\to\sales-server\sales.access.log combined
</VirtualHost>
```

Edite o arquivo hosts do seu sistema, adicionando a seguinte entrada:

```
127.0.0.1      sales.dev
```

Reinicie o apache e acesse a url “sales.dev”. Verifique se a mensagem “Hello World” surge. Em caso negativo, verifique se o módulo `rewrite_rule` do apache está carregado. Se não funcionar, verifique se o `<Directory>` do `http.conf` está referenciando para um diretório imediatamente acima do seu diretório *sales-server*.

13.2 Crie o arquivo .htaccess

Crie o arquivo `.htaccess` no diretório `public`:

```
public/.htaccess
```

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [QSA,L]
```

13.3 Configurando o acesso ao banco de dados

Para configurar o banco de dados, use a classe DB.php ([baixe ela aqui¹](https://raw.githubusercontent.com/danielschmitz/vue-codigos/sales/sales-server/DB.php)). Salve este arquivo no diretório sales-server.

Para usar esta classe, basta definir no arquivo public/index.php os parâmetros de conexão e realizar o include, veja:

```
public/index.php
```

```
<?php
require '../vendor/autoload.php';

require '../config.php';
require '../DB.php';

$app = new \Slim\App();

.... continua ....
```

Veja que incluímos o arquivo ../config.php que está fora do diretório público, e contém as configurações de acesso ao banco de dados:

¹<https://raw.githubusercontent.com/danielschmitz/vue-codigos/sales/sales-server/DB.php>

config.php

```
<?php
define("DB_HOST", "localhost");
define("DB_NAME", "sales");
define("DB_USER", "root");
define("DB_PASSWORD", "");
```

A classe DB pode ser usada em qualquer rota do Slim Framework. Em um exemplo simples, vamos criar o arquivo tests.php que conterá alguns testes que possamos fazer no decorrer do desenvolvimento do sistema, veja:

public/tests.php

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

$app->get('/', function (Request $request, Response $response) {
    $response->getBody()->write("Hello World");
    return $response;
});

$app->get('/databases', function (Request $request, Response $response) {

    $dbs = DB::query( 'SHOW DATABASES' );
    while( ( $db = $dbs->fetchColumn( 0 ) ) !== false )
    {
        $response->getBody()->write($db . ", ");
    }
}
```

```
    return $response;
});
```

Movemos para este arquivo o método ‘Hello World’ que estava no arquivo `public/index.php`. Teste a sua aplicação e acesse `sales.dev/databases`. Deverá aparecer as *databases* cadastradas no mysql.

Como o objetivo do sistema em PHP é ser o **mais simples possível**, vamos adicionar as rotas do servidor através de includes, onde cada arquivo representam as rotas de uma determinada área. O novo arquivo ‘`public/index.php`’ fica com o seguinte código:

`public/index.php`

```
<?php
require '../vendor/autoload.php';

require '../config.php';
require '../DB.php';

$app = new \Slim\App;

require 'tests.php';
//require 'login.php'
//require 'employee.php'
//require 'customer.php'
//require 'sales.php'
//require 'category.php'
//require 'supplier.php'
//require 'product.php'

$app->run();
```

Perceba que adicionamos vários requires, cada um representando uma área do

sistema. Primeiro temos `tests.php`, que usaremos ocasionalmente. Depois teremos `login.php`, responsável em trabalhar com os dados de login de todos os usuários.

13.4 Configurando o CORS

Nesta aplicação, teremos o cliente em um domínio diferente do servidor. Geralmente, o cliente está localizado em `http://localhost:8080` e o servidor já configuramos para `http://sales.dev/`. Para que o cliente possa se comunicar com o servidor, é preciso configurar o CORS(HTTP access control). Esta configuração pode ser realizada com o CorsSlim:

```
composer require palanik/corsslim:dev-slim3
```

Após a instalação do CorsSlim, podemos usá-lo logo após a criação da variável APP, veja:

public/index.php

```
<?php
require '../vendor/autoload.php';

require '../config.php';
require '../DB.php';

$app = new \Slim\App;

//CORS - inicio
$corsOptions = array(
    "origin" => "*",
    "exposeHeaders" => array("Content-Type", "X-Requested-With", "X-authentication", "X-client"),
    "allowMethods" => array('GET', 'POST', 'PUT', 'DELETE', '\
OPTIONS')
);
```

```
$cors = new \CorsSlim\CorsSlim($corsOptions);  
$app->add($cors);  
//CORS - fim  
  
require 'tests.php';  
//require 'login.php';  
//require 'employee.php'  
//require 'customer.php'  
//require 'sales.php'  
//require 'category.php'  
//require 'supplier.php'  
//require 'product.php'  
  
$app->run();
```



Em uma aplicação real, no servidor de produção, se houver a necessidade de usar URLs diferentes, deve-se alterar o parâmetro "origin" => "*" do CORS para o domínio de no qual será permitido o acesso. Desta forma, somente o domínio de origem poderá realizar chamadas ajax ao seu servidor, o que é das formas de melhorar a segurança do mesmo.

13.5 Login (login.php)

O arquivo login.php contém os métodos REST para que o cliente possa realizar o login e cadastro no sistema. Estas duas ações são realizadas no mesmo método, exibido a seguir:

public/login.php

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

$app->post('/login', function (Request $request, Response $
response) {

    try{

        $user = (object)$request->getParsedBody();

        if (empty($user->email)){
            throw new \Exception("Email empty...");
        }
        if (empty($user->password)){
            throw new \Exception("Password empty...");
        }

        if ($user->create_account){
            //Criar a conta do usuário
            if (empty($user->name)){
                throw new \Exception("Name empty...");
            }
            //Verifica se o email já está cadastrado
            $sql = "SELECT id FROM Users WHERE email=:email";
            $stmt = DB::prepare($sql);
            $stmt->bindParam("email",$user->email);
            $stmt->execute();

            if ($stmt->rowCount()==0){
```



```
$isEmployee = $user->role=="Employee";
$isCustomer = $user->role=="Customer";
$password = md5($user->password);

$sql = "INSERT INTO users (name,email,password,isEmployee\
oyee,isCustomer) VALUES (:name,:email,:password,:isEmployee\
,:isCustomer)";
$stmt = DB::prepare($sql);
$stmt->bindParam(':name', $user->name);
$stmt->bindParam(':email', $user->email);
$stmt->bindParam(':password', $password);
$stmt->bindParam(':isEmployee', $isEmployee,PDO::PARAM\
M_INT);
$stmt->bindParam(':isCustomer', $isCustomer,PDO::PARAM\
M_INT);
$stmt->execute();

$user->id = DB::lastInsertId();
$user->password = "";

//Token Fake
$user->token = "12345566788990865";

$response->withJson($user);

}else{
    throw new \Exception("Email exists in database");
}

}
else{
    //Tenta logar
```

```
$password = md5($user->password);
$sql = "SELECT * FROM Users WHERE email=:email and pass\
word=:password";
$stmt = DB::prepare($sql);
$stmt->bindParam("email",$user->email);
$stmt->bindParam("password",$password);
$stmt->execute();

if ($stmt->rowCount()==0){
    throw new \Exception("Email and password not found.");
}else{
    $db_user = $stmt->fetch();
    $db_user->password = "";
    //Token Fake
    $db_user->token = "12345566788990865";
    return $response->withJson($db_user);
}
}
}
catch(\Exception $e){
    return $response->withStatus(500)->write($e->getMessage\
());
}

return $response;
}
);
```

Quando usamos `$app->post('/login....')` estamos criando um filtro para que o método POST a url `/login` execute este bloco de código. Perceba que o código desta funcionalidade é extremamente grande, então vamos separar cada parte e explicar passo a passo o que foi feito.

Começamos então com a chamada ao método POST `/login`:

```
<?php
$app->post('/login', function (Request $request, Response $
response))
```

Veja que existem dois parâmetros nesta função anônima: `$request` e `$response`. O `$request` contém todas as informações que são pertinentes a requisição do cliente ao servidor, como a URL, os dados que foram enviados, o *Content Type*, informações sobre o autenticação, etc.

O `$response` contém todas as informações sobre a resposta do servidor ao cliente. Será no `response` que iremos responder ao cliente com um objeto com dados ou uma mensagem de erro.

Todo o login é contido pelo bloco de código try:

```
try{

    ...

}
catch(\Exception $e){
    return $response->withStatus(500)->write($e->getMessage());
}

return $response;
```

Com esta técnica, podemos disparar exceções no código com o propósito de retornar uma mensagem de erro devidamente formatada ao cliente. Veja que o erro é formado pelo `$response->withStatus(500)`, que é o código de retorno *Internal Server Error*.

No bloco `try..catch`, obtemos inicialmente o objeto que o cliente repassou para o servidor, através do método `getParsedBody`, veja:

```
$user = (object)$request->getParsedBody();

if (empty($user->email)){
    throw new \Exception("Email empty...");
}
if (empty($user->password)){
    throw new \Exception("Password empty...");
}
```

Quando o cliente envia os dados para o servidor, podemos usar `$request->getParsedBody()` para transformar estes dados em um objeto devidamente formatado, inclusive com o typecast `(object)`, porque o método `getParsedBody` retorna um array como padrão.

Com o objeto devidamente pronto, pode-se realizar algumas verificações tais como o `$user->email` estar vazio. Se estiver, dispara-se uma exceção (lembre-se que o código está no bloco `try..catch`).

Para criar a conta do usuário, isso se ele selecionou o checkbox `create account` no formulário, usamos a variável `$user->create_account`, conforme o código a seguir:

```
if ($user->create_account){
    //Criar a conta do usuário
    if (empty($user->name)){
        throw new \Exception("Name empty...");
    }
    //Verifica se o email já está cadastrado
    $sql = "SELECT id FROM Users WHERE email=:email";
    $stmt = DB::prepare($sql);
    $stmt->bindParam("email",$user->email);
    $stmt->execute();

    if ($stmt->rowCount()==0){
        ... create account ...
    }else{
        throw new \Exception("Email exists in database");
    }
}
```

```
    }  
  }  
}
```

Se o usuário for criar uma conta, verificamos se o campo `$user->name` está preenchido, e também verificamos se o email do usuário está presente no banco de dados.

Como a classe DB foi incluída no arquivo `index.php`, podemos usá-la para realizar operações PDO (PHP Data Objects) no banco de dados. Isso é feito para verificarmos se o email que está sendo cadastrado já não pertence a tabela `users`:

```
$sql = "SELECT id FROM Users WHERE email=:email";  
$stmt = DB::prepare($sql);  
$stmt->bindParam("email", $user->email);  
$stmt->execute();  
  
if ($stmt->rowCount()==0){  
    ... create account ...  
}else{  
    throw new \Exception("Email exists in database");  
}
```

Se o email existir (`$stmt->rowCount()!=0`), disparamos uma exceção e a conta não será criada. Se o email não for encontrado na tabela `users`, poderemos iniciar o processo de inclusão do usuário, veja:

```
$isEmployee = $user->role=="Employee";
$isCustomer = $user->role=="Customer";
$password = md5($user->password);

$sql = "INSERT INTO users (name,email,password,isEmployee,i\
sCustomer) VALUES (:name,:email,:password,:isEmployee,:isCu\
stomer)";

$stmt = DB::prepare($sql);
$stmt->bindParam(':name', $user->name);
$stmt->bindParam(':email', $user->email);
$stmt->bindParam(':password', $password);
$stmt->bindParam(':isEmployee', $isEmployee,PDO::PARAM_INT);
$stmt->bindParam(':isCustomer', $isCustomer,PDO::PARAM_INT);
$stmt->execute();

$user->id = DB::lastInsertId();
$user->password = "";

//Token Fake
$user->token = "12345567";

$response->withJson($user);
```

Para cadastrar o usuário, inicialmente é feito algumas atribuições nos campos *isEmployee*, *isCustomer* e *password*. Após criar a SQL com o INSERT, usamos o `bindParam` para atribuir cada valor, observando que os campos *isEmployee* e *isCustomer* são do tipo *TinyInt* no MySQL, então é necessário informar que eles são do tipo `PDO::PARAM_INT`.

O comando `$stmt->execute()` irá executar a sql, e o comando `DB::lastInsertId()` irá retornar o ID do sql realizado. Esse id iremos retornar ao cliente que fez a requisição ajax, através do `$response->withJson($user)`.

Perceba que usamos um token “fake”, de mentira, apenas para simular que o usuário foi logado. Quando o servidor retorna o objeto user com o token preenchido, o cliente Vue irá entender que ele está logado com o token e irá tomar as devidas providências.

Se o usuário não estiver cadastrando a conta, e já possui uma, o seguinte código será executado:

```
else{
    //Tenta logar
    $password = md5($user->password);
    $sql = "SELECT * FROM Users WHERE email=:email and passwo\
rd=:password";
    $stmt = DB::prepare($sql);
    $stmt->bindParam("email",$user->email);
    $stmt->bindParam("password",$password);
    $stmt->execute();

    if ($stmt->rowCount()==0){
        throw new \Exception("Email and password not found.");
    }else{
        $db_user = $stmt->fetch();
        $db_user->password = "";
        //Token Fake
        $db_user->token = "1234556";
        return $response->withJson($db_user);
    }
}
```

Neste código, realizamos um SELECT na tabela users e caso o encontre obtemos os dados do usuário, incluindo o token “de mentira” e retornando estes dados ao cliente.

13.6 Incluindo o token JWT de autenticação (login)

O token de autenticação é a peça chave para definir que o usuário está logado. Talvez você já tenha mantido um usuário logado pela sessão do PHP, mas este artifício está

em desuso por questões de segurança e também pelo desenvolvimento mobile, que costume ser “stateless”.

De qualquer forma, manter o usuário pela sessão é muito fácil, bastando apenas usar a variável global `$_SESSION` do php, mas nesta obra não a usaremos. Vamos sempre dar preferência ao JWT. Existem diversas implementações do Token para PHP. Vamos pegar uma delas, que é o *php-jwt*, que será instalado da seguinte forma:

```
composer require firebase/php-jwt:dev-master
```

Após a instalação do *php-jwt*, vamos configurá-lo. Todo token é criado baseado em uma chave que deve estar completamente invisível ao acesso externo (aqui entram conceitos de chave pública/privada). O que precisamos, a princípio, é adicionar uma chave “secreta” no arquivo `config.php`, da seguinte forma:

```
<?php
define("DB_HOST", "localhost");
define("DB_NAME", "sales");
define("DB_USER", "root");
define("DB_PASSWORD", "");

define("SECRECT_KEY", "my_super_secret_key_change_this_in_pr\
oduction");
```

Após criar a chave privada, podemos usar o JWT para criar a chave pública. Assim que o usuário logar, podemos fazer o seguinte:

public/login.php

```
//Token Fake
//$db_user->token = "12345566788990865";

$token_data = array(
    "user_id" => $db_user->id
);

$db_user->token = \Firebase\JWT\JWT::encode(
    $token_data,
    base64_decode(SECRET_KEY),
    'HS512'
);

return $response->withJson($db_user);
}
```

Agora, ao invés de usarmos o token *fake*, usamos a classe `\Firebase\JWT\JWT` para gerar uma token, no qual é necessário repassar o objeto `$token_data`, usar a chave privada que está no `config.php` e informar a forma de encriptação usada, nesse caso a HS512.

A variável `$token_data` pode conter qualquer informações que se julgue necessário para realizar uma validação posterior, caso haja necessidade.

Fizemos o token para o login. Mas quando o usuário cria a conta, queremos também que o token seja retornado. Para isso, retiramos o Token *fake* após criar o usuário no banco de dados, e adicionamos a mesma geração do token pelo JWT, veja:

```
...

$user->id = DB::lastInsertId();
$user->password = "";

//Token Fake
//$user->token = "12345566788990865";

$token_data = array(
    "user_id" => $user->id
);

$user->token = \Firebase\JWT\JWT::encode(
    $token_data,
    base64_decode(SECRET_KEY),
    'HS512'
);

return $response->withJson($user);
}
}
...
```

13.7 Verificando o login

O login do usuário é verificado sempre quando queremos realizar uma filtragem na rota do Slim. Isso é feito através de middlewares. Primeiro, criamos uma variável que corresponde ao middleware “auth”, veja:

public/index.php

```
<?php
require '../vendor/autoload.php';

require '../config.php';
require '../DB.php';

$app = new \Slim\App();

//Auth Middleware
$auth = function ($request, $response, $next) {

    //Get token from header
    $token = $request->getHeader('x-access-token');
    if (empty($token)){
        return $response->withStatus(401)->write("Unauthorized\
d access - You should login");
    }
    try{
        $decoded = \Firebase\JWT\JWT::decode($token[0], base64\
_decode(SECRET_KEY), array('HS256'));
    } catch(\Exception $e){
        return $response->withStatus(401)->write("Unauthorized\
access - Wrong Token - " . $e->getMessage() . " - " . $tok\
en[0]);
    }

    return $next($request, $response);
};
}
```

A variável `$auth` é atribuída a uma função que possui como parâmetros a requisição e resposta da chamada `http` do cliente ao servidor. Nesta função, a primeira tarefa é obter o token que deve estar presente no cabeçalho `HTTP` da requisição.

Com `$request->getHeader('x-access-token')` nós obtemos o token. Isso significa que, no cliente ou no Postman, quando for necessário realizar alguma requisição que necessite de login, é indispensável repassar o token pelo cabeçalho com a chave `x-access-token`.

Se o token estiver vazio, retornamos a resposta com o erro “401”. Se o token estiver preenchido, usamos então `\Firebase\JWT\JWT::decode` para validá-lo. O `decode` retorna um objeto que contém o `user_id` que foi informado na sua codificação, no ato de login. Não iremos usá-lo aqui, mas pode ser usado para eventuais regras de segurança. Por exemplo, pode-se adicionar o IP de origem ao logar, e na verificação do token, ver se o ip é o mesmo.

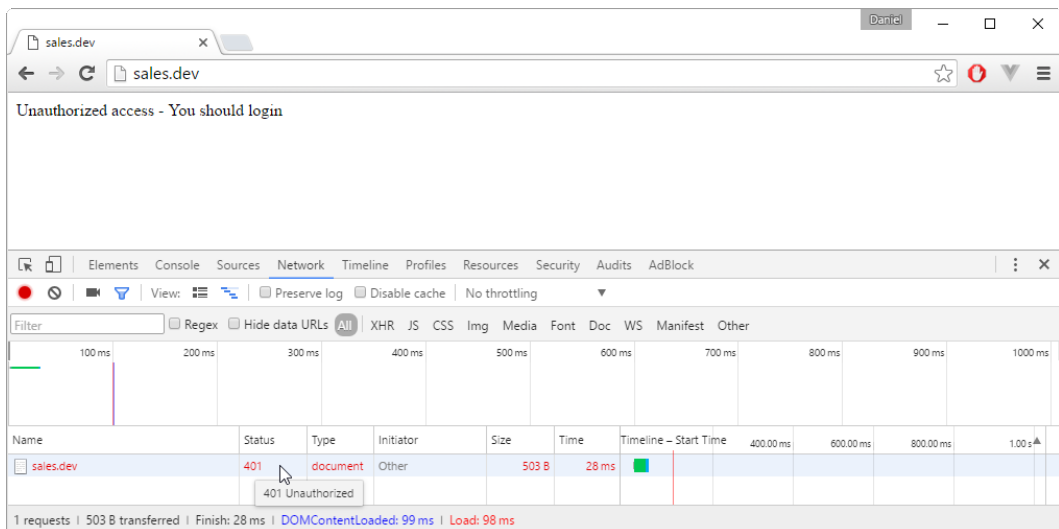
O método `\Firebase\JWT\JWT::decode` irá disparar exceções se o token for inválido. Por isso usamos `try...catch` para essa validação. Qualquer problema que ocorrer, a exceção é lançada e o retornamos uma resposta com o status 401.

Se houver o token e se ele for válido, não há nada a fazer, então no final do método nós usamos o `next` para que a requisição continue.

Com o middleware pronto, pode-se usá-lo adicionando-o da seguinte forma:

```
$app->get('/', function (Request $request, Response $response) {
    $response->getBody()->write("Hello World");
    return $response;
})->add($auth);
```

Perceba o `->add($auth)` no final, adicionando o middleware no “`->get('/')`”. Isso significa que o middleware será executado e retornará um erro, pois não repassamos o token (será feito no cliente). Aquele “hello world” que recebíamos como resposta, será substituído por:



13.8 Salvando categorias

Vamos criar o método que irá salvar categorias. Salvar pode ser atribuído a inserir e editar, podemos verificar se é um ou outro se o objeto possui o id preenchido.

No arquivo `index.php`, descomente o `require 'category.php'` para que ele possa ser incluído, e crie o arquivo com o seguinte código:

`public/category.php`

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

$app->post('/category', function (Request $request, Response
e $response) {
    try{
        $category = (object)$request->getParsedBody();
        if (!empty($category->id)){
            //update
```

```

        $sql = "UPDATE Categories SET name=:name WHERE id=\
:id";

        $stmt = DB::prepare($sql);
        $stmt->bindParam(':name', $category->name);
        $stmt->bindParam(':id', $category->id, PDO::PARAM_INT\
T);

        $stmt->execute();
        return $response->withJson($category);
    }else{
        //insert
        $sql = "INSERT INTO Categories (name) VALUES (:name\
)";

        $stmt = DB::prepare($sql);
        $stmt->bindParam(':name', $category->name);
        $stmt->execute();
        $category->id = DB::lastInsertId();
        return $response->withJson($category);
    }
}

catch(\Exception $e){
    return $response->withStatus(500)->write($e->getMessa\
ge());
}

})->add($auth);
}

```

Usa-se o `$app->post('/category')` que irá responder a um POST `category`, que será enviado pelo Vue. Neste método, um objeto `$category` será obtido pelo `$request->getParsedBody()` do Slim. Verificamos então se existe o `$category->id`, para avaliarmos se estamos inserindo ou editando um registro.

Se for para editar, usamos o PDO para realizar o Update. Atenção ao tratamento de campos que são do tipo `int`, onde o `PDO::PARAM_INT` é necessário. Após realizar o update, retornamos o objeto através do `$response->withJson`.

Se for inserir, realizamos o Insert. O `$category->id` é obtido pelo `lastInsertId` e o objeto `$category` é retornado ao cliente, agora com o id preenchido.

No final do código temos o `->add($auth)`, isso faz com que o método necessite de autenticação, que será repassada pelo cliente.

13.9 Exibindo categorias (com paginação e busca)

Para exibir categorias, precisamos definir alguns parâmetros extras para realizar a paginação. No MySQL, a paginação é feita com o parâmetro `LIMIT`, que aceita o valor inicial e a quantidade de itens a serem retornados. Além da paginação, também precisamos prever uma busca prévia pelo campo “nome”.

O código a seguir ilustra todo o processo:

public/category.php

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

$app->get('/categories', function (Request $request, Response $response) {

    $sql = "";
    $parameters = $request->getQueryParams();
    $start =(int)$parameters['start'];
    $limit =(int)$parameters['limit'];

    $keyword=null;
    if (array_key_exists("q", $parameters)){
        $keyword = $parameters['q'];
    }

    if (!empty($start)&&!empty($limit)){
```

```
$start--;

$stmt = null;
if (empty($keyword)){
    $sql = "SELECT id,name FROM Categories LIMIT :start,:\n
limit";
    $stmt = DB::prepare($sql);
    $stmt->bindParam(':start', $start,PDO::PARAM_INT);
    $stmt->bindParam(':limit', $limit,PDO::PARAM_INT);

}else{
    $keywordLike = "%".$keyword."%";
    $sql = "SELECT id,name FROM Categories WHERE name LIK\
E :keyword LIMIT :start,:limit";
    $stmt = DB::prepare($sql);
    $stmt->bindParam(':start', $start,PDO::PARAM_INT);
    $stmt->bindParam(':limit', $limit,PDO::PARAM_INT);
    $stmt->bindParam(':keyword', $keywordLike);
}
$stmt->execute();

$sqlCount = null;
$total = 0;
if (empty($keyword)){
    $sqlCount = "SELECT count(id) FROM Categories";
    $stmtCount = DB::prepare($sqlCount);
    $stmtCount->execute();
    $total = $stmtCount->fetchColumn();
}else{
    $keywordLike = "%".$keyword."%";
    $sqlCount = "SELECT count(id) FROM Categories WHERE \
name LIKE :keyword";
    $stmtCount = DB::prepare($sqlCount);
```



```
        $stmtCount->bindParam(':keyword', $keywordLike);
        $stmtCount->execute();
        $total = $stmtCount->fetchColumn();
    }

    return $response->withJson($stmt->fetchAll())->withHeader('Access-Control-Expose-Headers', 'x-total-count')->withHeader('x-total-count', $total);

} else {
    $sql = "SELECT id,name FROM Categories";
    $stmt = DB::prepare($sql);
    $stmt->execute();

    return $response->withJson($stmt->fetchAll());
}
})->add($auth);
```

O método para obter categorias é o GET `Categories`. Usamos o `$request->getQueryParams()` para obter os valores `start`, `limit` e `q` que deverão ser passados pela url, como por exemplo `/categories?start=1&limit=10&q=busca`.

Se estes parâmetros estiverem preenchidos, então realizamos a sql com o LIMIT. Perceba que, tanto `start` quanto `limit` são campos inteiros, então é necessário usar `PDO::PARAM_INT` para a formatação correta da SQL.

Também executamos outra SQL, que é a `$sqlCount` usada para contar quanto registros existem na tabela de categorias. Isso é necessário para que possamos montar corretamente a paginação.

O retorno dos dados pelo objeto `request` é bastante peculiar:

```
return $response->withJson($stmt->fetchAll())
->withHeader('Access-Control-Expose-Headers', 'x-total-count\
')
->withHeader('x-total-count', $total);
```

Veja que, além de retornar os dados com o Json, também retornamos dois cabeçalhos adicionais na requisição, que é o Access-Control-Expose-Headers, necessário para consultas com CORs (domínios diferentes) e o x-total-count, criado para retornar o total de registros. Essa informação será usada no cliente, para montar a paginação.

Se start e limit não forem repassados pela query string, um select normal com todas as categorias será retornado.

13.10 Removendo uma categoria

Para remover uma categoria, usamos o método DELETE, repassando o id da categoria em questão, veja:

public/category.php

```
$app->delete('/category/{id}', function (Request $request, \
Response $response) {
    try{
        $id = $request->getAttribute('id');
        if (!empty($id)){

            $sql = "DELETE FROM Categories WHERE id=:id";
            $stmt = DB::prepare($sql);
            $stmt->bindParam(':id', $id, PDO::PARAM_INT);
            $stmt->execute();
            return $response;
        }
    }
    catch(\Exception $e){
```

```

        return $response->withStatus(500)->write($e->getMessage\
    ());
    }

    })->add($auth);

```

13.11 Cadastro de fornecedores

A princípio, o código para cadastrar fornecedores(Suppliers) é semelhante ao cadastro de categorias, apenas pela adição do campo “Endereço”, que é um campo do tipo BLOB, no qual deve ser usado o parâmetro PDO: :PARAM_LOB. O código fonte desta funcionalidade é exibida a seguir:

public/supplier.php

```

<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

$app->get('/suppliers', function (Request $request, Response
e $response){

    $sql = "";
    $parameters = $request->getQueryParams();
    $start =(int)$parameters['start'];
    $limit =(int)$parameters['limit'];

    $keyword=null;
    if (array_key_exists("q", $parameters)){
        $keyword = $parameters['q'];
    }

    if (!empty($start)&&!empty($limit)){

```

```
$start--;

$stmt = null;
if (empty($keyword)){
    $sql = "SELECT id,name,address FROM Suppliers LIMIT :start,:limit";
    $stmt = DB::prepare($sql);
    $stmt->bindParam(':start', $start,PDO::PARAM_INT);
    $stmt->bindParam(':limit', $limit,PDO::PARAM_INT);

}else{
    $keywordLike = "%".$keyword."%";
    $sql = "SELECT id,name,address FROM Suppliers WHERE name LIKE :keyword LIMIT :start,:limit";
    $stmt = DB::prepare($sql);
    $stmt->bindParam(':start', $start,PDO::PARAM_INT);
    $stmt->bindParam(':limit', $limit,PDO::PARAM_INT);
    $stmt->bindParam(':keyword', $keywordLike);
}
$stmt->execute();

$sqlCount = null;
$total = 0;
if (empty($keyword)){
    $sqlCount = "SELECT count(id) FROM Suppliers";
    $stmtCount = DB::prepare($sqlCount);
    $stmtCount->execute();
    $total = $stmtCount->fetchColumn();
}else{
    $keywordLike = "%".$keyword."%";
    $sqlCount = "SELECT count(id) FROM Suppliers WHERE name LIKE :keyword";
    $stmtCount = DB::prepare($sqlCount);
```

```

        $stmtCount->bindParam(':keyword', $keywordLike);
        $stmtCount->execute();
        $total = $stmtCount->fetchColumn();
    }

    return $response->withJson($stmt->fetchAll())->withHeader(
        'Access-Control-Expose-Headers', 'x-total-count')->withHeader(
        'x-total-count', $total);

} else {
    $sql = "SELECT id,name,address FROM Suppliers";
    $stmt = DB::prepare($sql);
    $stmt->execute();

    return $response->withJson($stmt->fetchAll());
}
})->add($auth);

$app->post('/supplier', function (Request $request, Response $response) {
    try {
        $supplier = (object)$request->getParsedBody();
        if (!empty($supplier->id)) {
            //update
            $sql = "UPDATE Suppliers SET name=:name,address=:address WHERE id=:id";
            $stmt = DB::prepare($sql);
            $stmt->bindParam(':name', $supplier->name);
            $stmt->bindParam(':id', $supplier->id, PDO::PARAM_INT);
            $stmt->bindParam(':address', $supplier->address, PDO::PARAM_LOB);
            $stmt->execute();
            return $response->withJson($supplier);
        }
    } catch (Exception $e) {
        return $response->withStatus(500);
    }
});

```

```
    }else{
        //insert
        $sql = "INSERT INTO Suppliers (name,address) VALUES (\
:name,:address)";
        $stmt = DB::prepare($sql);
        $stmt->bindParam(':name', $supplier->name);
        $stmt->bindParam(':address', $supplier->address,PDO::\
PARAM_LOB);
        $stmt->execute();
        $supplier->id = DB::lastInsertId();
        return $response->withJson($supplier);
    }

}

catch(\Exception $e){
    return $response->withStatus(500)->write($e->getMessage\
());
}

})->add($auth);

$app->delete('/supplier/{id}', function (Request $request, \
Response $response) {
    try{
        $id = $request->getAttribute('id');
        if (!empty($id)){

            $sql = "DELETE FROM Suppliers WHERE id=:id";
            $stmt = DB::prepare($sql);
            $stmt->bindParam(':id', $id,PDO::PARAM_INT);
            $stmt->execute();
            return $response;
        }
    }
```

```
    }  
    catch(\Exception $e){  
        return $response->withStatus(500)->write($e->getMessage\  
( ));  
    }  
  
    }->add($auth);
```

Após criar o arquivo `supplier.php` e adicionar este código, descomente no arquivo `index.php` a linha que faz o `require`.

13.12 Cadastro de Produtos

O cadastro de produtos em PHP é realizado no arquivo `product.php`, e inicialmente teremos o seguinte código:

```
<?php  
use \Psr\Http\Message\ServerRequestInterface as Request;  
use \Psr\Http\Message\ResponseInterface as Response;  
  
$app->get('/products', function (Request $request, Response\  
    $response) {  
  
    $sql = "";  
    $parameters = $request->getQueryParams();  
    if (array_key_exists("start", $parameters)){  
        $start =(int)$parameters['start'];  
    }  
    if (array_key_exists("limit", $parameters)){  
        $limit =(int)$parameters['limit'];  
    }  
}
```

```
$keyword=null;
if (array_key_exists("q", $parameters)){
    $keyword = $parameters['q'];
}

if (!empty($start)&&!empty($limit)){
    $start--;

    $stmt = null;
    if (empty($keyword)){
        $sql = "SELECT id,name,idCategory,idSupplier,quantity\
,minQuantity,price,description,active,code FROM Products LI\
MIT :start,:limit";
        $stmt = DB::prepare($sql);
        $stmt->bindParam(':start', $start,PDO::PARAM_INT);
        $stmt->bindParam(':limit', $limit,PDO::PARAM_INT);

    }else{
        $keywordLike = "%".$keyword."%";
        $sql = "SELECT id,name,idCategory,idSupplier,quantity\
,minQuantity,price,description,active,code FROM Products WH\
ERE name LIKE :keyword LIMIT :start,:limit";
        $stmt = DB::prepare($sql);
        $stmt->bindParam(':start', $start,PDO::PARAM_INT);
        $stmt->bindParam(':limit', $limit,PDO::PARAM_INT);
        $stmt->bindParam(':keyword', $keywordLike);
    }
    $stmt->execute();

    $sqlCount = null;
    $total = 0;
    if (empty($keyword)){
        $sqlCount = "SELECT count(id) FROM Products";
```



```

$stmtCount = DB::prepare($sqlCount);
$stmtCount->execute();
$total = $stmtCount->fetchColumn();
}else{
    $keywordLike = "%".$keyword."%";
    $sqlCount = "SELECT count(id) FROM Products WHERE name LIKE :keyword";
    $stmtCount = DB::prepare($sqlCount);
    $stmtCount->bindParam(':keyword', $keywordLike);
    $stmtCount->execute();
    $total = $stmtCount->fetchColumn();
}

return $response->withJson($stmt->fetchAll())->withHeader('Access-Control-Expose-Headers', 'x-total-count')->withHeader('x-total-count', $total);

}else{
    $sql = "SELECT id,name,idCategory,idSupplier,quantity,maxInQuantity,price,description,active,code FROM Products";
    $stmt = DB::prepare($sql);
    $stmt->execute();

    return $response->withJson($stmt->fetchAll());
}

})->add($auth);

$app->post('/product', function (Request $request, Response $response) {

    try{

```

```
$product = (object)$request->getParsedBody();

$product->price = DB::decimalToMySQL($product->price);

if (!empty($product->id)){
    //update
    $sql = "UPDATE Products SET name=:name,idCategory=:id\
Category,idSupplier=:idSupplier,quantity=:quantity,minQuant\
ity=:minQuantity,price=:price,description=:description,acti\
ve=:active,code=:code WHERE id=:id";
    $stmt = DB::prepare($sql);
    $stmt->bindParam(':name', $product->name);
    $stmt->bindParam(':idCategory', $product->idCategory, \
PDO::PARAM_INT);
    $stmt->bindParam(':idSupplier', $product->idSupplier, \
PDO::PARAM_INT);
    $stmt->bindParam(':quantity', $product->quantity,PDO:\
:PARAM_INT);
    $stmt->bindParam(':minQuantity', $product->minQuantit\
y,PDO::PARAM_INT);
    $stmt->bindParam(':price', $product->price);
    $stmt->bindParam(':description', $product->descriptio\
n,PDO::PARAM_LOB);
    $stmt->bindParam(':active', $product->active,PDO::PAR\
AM_INT);
    $stmt->bindParam(':code', $product->code);
    $stmt->bindParam(':id', $product->id,PDO::PARAM_INT);
    $stmt->execute();
    return $response->withJson($product);
}else{
    //insert
    $sql = "INSERT INTO Products (name,idCategory,idSuppl\
ier,quantity,minQuantity,price,description,active,code) VAL\
```

```

UES (:name,:idCategory,:idSupplier,:quantity,:minQuantity,:price,:description,:active,:code)";
$stmt = DB::prepare($sql);
$stmt->bindParam(':name', $product->name);
$stmt->bindParam(':idCategory', $product->idCategory,\
PDO::PARAM_INT);
$stmt->bindParam(':idSupplier', $product->idSupplier,\
PDO::PARAM_INT);
$stmt->bindParam(':quantity', $product->quantity,PDO:\
:PARAM_INT);
$stmt->bindParam(':minQuantity', $product->minQuantit\
y,PDO::PARAM_INT);
$stmt->bindParam(':price', $product->price);
$stmt->bindParam(':description', $product->descriptio\
n,PDO::PARAM_LOB);
$stmt->bindParam(':active', $product->active,PDO::PAR\
AM_INT);
$stmt->bindParam(':code', $product->code);

$stmt->execute();
$product->id = DB::lastInsertId();
return $response->withJson($product);
}

}

catch(\Exception $e){
    return $response->withStatus(500)->write($e->getMessage\
());
}

})->add($auth);

$app->delete('/product/{id}', function (Request $request, R\

```

```

response $response) {
    try{
        $id = $request->getAttribute('id');
        if (!empty($id)){

            $sql = "DELETE FROM Products WHERE id=:id";
            $stmt = DB::prepare($sql);
            $stmt->bindParam(':id', $id,PDO::PARAM_INT);
            $stmt->execute();
            return $response;
        }
    }
    catch(\Exception $e){
        return $response->withStatus(500)->write($e->getMessage\
    ());
    }

})->add($auth);

```

Como o produto possui mais campos, incluindo chaves estrangeiras, o código é um pouco mais complexo. A parte mais importante do código é na referência aos seus campos, como por exemplo:

```

$sql = "UPDATE Products SET name=:name,idCategory=:idCateg\
ory,idSupplier=:idSupplier,quantity=:quantity,minQuantity=: \
minQuantity,price=:price,description=:description,active=:a\
ctive,code=:code WHERE id=:id";
$stmt = DB::prepare($sql);
$stmt->bindParam(':name', $product->name);
$stmt->bindParam(':idCategory', $product->idCategory, \
PDO::PARAM_INT);
$stmt->bindParam(':idSupplier', $product->idSupplier, \
PDO::PARAM_INT);

```

```
$stmt->bindParam(':quantity', $product->quantity, PDO::\
:PARAM_INT);
$stmt->bindParam(':minQuantity', $product->minQuantit\
y, PDO::PARAM_INT);
$stmt->bindParam(':price', $product->price);
$stmt->bindParam(':description', $product->descriptio\
n, PDO::PARAM_LOB);
$stmt->bindParam(':active', $product->active, PDO::PAR\
AM_INT);
$stmt->bindParam(':code', $product->code);
$stmt->bindParam(':id', $product->id, PDO::PARAM_INT);
$stmt->execute();
```

No caso da referência ao `idCategory`, é necessário o uso do `PDO::PARAM_INT`, e no caso do `$product->description` é usado `PDO::PARAM_LOB` pois o campo é do tipo BLOB.

Outro detalhe importante é que usamos o método `decimalToMySQL` para converter o valor de moeda para o formato americano, que possui o ponto como separador de centavos:

```
$product->price = DB::decimalToMySQL($product->price);
```

Não esqueça de remover o `product.php` do arquivo `index.php`:

```
require 'tests.php';
require 'login.php';
//require 'employee.php'
//require 'customer.php'
//require 'sales.php'
require 'category.php';
require 'supplier.php';
require 'product.php';
```

14. Criando o sistema Sales no cliente

Com o servidor pronto, podemos iniciar a criação do sistema de vendas no cliente. Como foi estabelecido, o sistema funcionará de forma independente do servidor, em outro diretório.



Uma nota importante Este sistema tem como objetivo mostrar as mais diversas técnicas de uso do framework Vue, e não é uma “receita de bolo”.

Para uma aprendizagem melhor, deve-se escrever todo o código do sistema, não apenas copiar e colar. Deve-se compreender cada passo e, em caso de dúvida, pergunte ao autor.

No gerenciamento de acesso ao servidor, o crud exemplifica diversas técnicas e você deve escolher a que melhor lhe agrada. Por exemplo, na tela de cadastro de Categorias usamos vuex, na tela de cadastro de Fornecedores usamos o Vue Resource diretamente no componente, e finalmente na tela de cadastro de Produtos, usamos o conceito de *services*. Perceba que são várias formas de fazer a mesma coisa (acessar o servidor), então vc pode escolher a que mais lhe agrada.

Vamos usar o vue-cli para criar o projeto:

```
$ vue init browserify-simple#1 sales-client  
$ cd sales-client  
$ npm install
```

Após a criação e instalação das bibliotecas iniciar, vamos aproveitar para instalar tudo que é necessário para o projeto, veja:

```
$ npm i -S vuex vue-router vue-resource vue-validator boots\
trap moment jquery
```

14.1 Configurando o bootstrap

O bootstrap será a camada css da nossa aplicação, que conterà, a princípio, uma tela de login. Antes de configurar esta tela, é preciso configurar a inclusão dos arquivos css no projeto, mais precisamente no arquivo index.html:

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>sales-client</title>
  <link rel="stylesheet" href="node_modules/bootstrap/dist/\
css/bootstrap.min.css">
</head>

<body>
  <app></app>
  <script src="node_modules/jquery/dist/jquery.min.js"></sc\
ript>
  <script src="node_modules/bootstrap/dist/js/bootstrap.min\
.js"></script>
  <script src="dist/build.js"></script>
</body>

</html>
```

Alteramos o index.html adicionando o jQuery e o Bootstrap, ambos serão utilizados durante o desenvolvimento da aplicação.

14.2 Configurações iniciais

É necessário realizar algumas configurações iniciais no projeto, que compreende a instalação de bibliotecas e criação de componentes que serão utilizados na aplicação.

14.2.1 Validação de formulários (Vue Validator)

É necessário adicionar o `vue-validator` no arquivo `src/main.js`, para que a validação passe a funcionar em todos os formulários:

`src/main.js`

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

import VueValidator from 'vue-validator'

Vue.use(VueValidator)

new Vue({
  el: 'body',
  components: { App }
})
```

14.2.2 Configuração do Vue Resource

Após configurar o validator, voltamos ao `main.js` para adicionar o `Vue Resource`, necessário para realizarmos as chamadas Ajax no servidor.

src/main.js

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

import VueValidator from 'vue-validator'
import VueResource from 'vue-resource'

Vue.use(VueValidator)
Vue.use(VueResource)

new Vue({
  el: 'body',
  components: { App }
})
```

14.2.3 Arquivo de configuração

Criamos no diretório src o arquivo `config.js` que terá algumas informações sobre a aplicação. A primeira informação é a URL de acesso a api do servidor, neste caso configurada no capítulo anterior, onde configuramos o domínio `sales.dev`:

src/config.js

```
export const URL = "http://sales.dev/"
```

14.2.4 Controle <loading>

Quando realizarmos uma ação ajax do cliente para o servidor, é preciso indicar ao usuário que essa requisição está sendo feita. Uma das formas de se indicar este processo é adicionando uma imagem de *loading*, que pode ser obtida [neste link](http://www.ajaxload.info/)¹. Salve a imagem na raiz do projeto como *loading.gif*.

¹<http://www.ajaxload.info/>

Crie o diretório `src/controls`. No diretório `controls`, ficam os principais controles da aplicação, o que é diferente dos componentes, que são as telas da aplicação.

O controle `<loading>` possui o seguinte código:

`src/controls/Loading.vue`

```
<template>
  
</template>
<script>
  import {isLoading} from '../vuex/getters'
  export default {
    vuex: {
      getters: {
        isLoading
      }
    }
  }
</script>
```

A princípio, este controle não funcionará, porque já implementamos o Vuex nele. Perceba que a imagem somente aparece na página se o *getter* `isLoading` for `true`.

14.2.5 Controle `<error>`

O controle `<error>` exibirá a mensagem de erro que estiver definida o store (vuex) da aplicação. O código para este controle é exibido a seguir:

src/controls/Error.vue

```
<template>
  <div class="alert alert-danger"
    v-show="hasError">
    {{getError}}</div>
</template>
<script>
  import {getError,hasError} from '../vuex/getters'

  export default{
    vuex: {
      getters: {
        getError,hasError
      }
    }
  }
</script>
```

Este controle também está intimamente ligado ao Vuex, que será configurado no próximo logo a seguir. Usamos o *getter* `hasError` para verificar se a caixa de erro deve aparecer na página, e o *getter* `getError` para exibir a mensagem.

14.3 Criando a estrutura do Vuex

O Vuex estará presente em toda a manipulação de dados da aplicação. Vamos criar uma estrutura simplificada (sem módulos) do Vuex, da seguinte forma:

- O diretório `src/vuex` conterá o store, as actions e os getters
- Todas as actions estarão no arquivo `src/vuex/actions.js`
- Todos os getters estarão no arquivo `src/vuex/getters.js`

src/vuex/store.js

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    error: {
      message: ""
    },
    loading: false,
    login: {
      name: null,
      email: null,
      token: null
    }
  },
  mutations: {
    SET_LOGIN (state, login) {
      state.login = login
    },
    SHOW_LOADING (state) {
      state.loading=true;
    },
    HIDE_LOADING (state) {
      state.loading=false;
    },
    SHOW_ERROR (state,msg) {
      state.error.message=msg;
      setTimeout(function(){
        state.error.message=""
      }, 3000)
    }
  }
})
```

```
    }, 5000)
  },
  HIDE_ERROR (state) {
    state.error.message="";
  }
}
})
```

A princípio o state possui o objeto login, loading e error. Com o desenvolvimento da aplicação iremos adicionando novas funcionalidades.

O arquivo `src/vuex/getters.js` possui os getters iniciais da aplicação, inicialmente retornando dados do login, loading e error.

`src/vuex/getters.js`

```
export function getLogin(state){
  return state.login;
}

export function isLoggedIn(state){
  return state.login.token!=null
}

export function isLoading(state){
  return state.loading
}

export function hasError(state){
  return state.error.message!="";
}

export function getError(state){
  return state.error.message;
}
```

Os actions são responsáveis em realizar alguma lógica na aplicação e disparar os mutations do store. A princípio, temos:

src/vuex/actions.js

```
export function setError({dispatch},msg){
  dispatch("SHOW_ERROR",msg);
}

export function clearError({dispatch}){
  dispatch("HIDE_ERROR");
}
```

No arquivo App.vue configuramos o vuex, da seguinte forma:

src/App.vue

```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
  </div>
</template>
<script>
  import store from './vuex/store'
  export default {
    data () {
      return {
        msg: 'Hello Vue!'
      }
    },
    store
  }
```

```
</script>
<style>
  body {
    background-color: #eee;
  }
</style>
```

No arquivo `App.vue`, importamos o `store` e o adicionamos a instância do componente, após o objeto `data`.

14.4 Criando a tela de login

Com o `vuex` pronto, podemos partir para a criação da tela de login. A estrutura que iremos montar é aplicada no `App.vue`, onde deveremos ter dois componentes: `Login` e `Admin`.

Vamos inicialmente criar o componente `Login.vue`, que ficará no diretório `src/components`:

`src/components/Login.vue`

```
<template>
  ...Template...
</template>
<script>
  import {setLogin} from '../vuex/actions'
  import Loading from '../controls/Loading.vue'
  import Error from '../controls/Error.vue'
  import {isLoading} from '../vuex/getters'

  export default{
    components: {
      Loading, Error
    },
  },
}
```

```
data() {
  return{
    user: {
      email: "",
      password: "",
      create_account: false,
      name: "",
      role: ""
    }
  },
  methods: {
    tryLogin(){
      this.setLogin(this.user);
    }
  },
  vuex: {
    actions: {
      setLogin
    },
    getters: {
      isLoading
    }
  }
}
</script>
<style scoped>

</style>
```

Inicialmente damos um destaque apenas no *script* do componente *Login.vue*. Veja que, na propriedade `Components`, temos a inclusão do `Loading` e o `Error`, que já foram criados previamente e serão usados no formulário de login.

Na propriedade `data` temos o objeto `user` que reflete os dados do formulário de login. No objeto `user` temos várias propriedades que serão ligadas aos campos do formulário através do *v-model*.

Em `methods` temos o `tryLogin`, que irá chamar a *action* `setLogin`, que ainda não programamos, mas será abordada a seguir. Na configuração do `vuex` temos a *action* `setLogin` e o *getter* `isLoading`. Este *getter* retorna `true` se o *mutation* `SHOW_LOADING` for executado (que irá alterar o `store.loading`). Aqui temos um dos muitos exemplos de design reativo que será empregado no formulário.

Vamos agora ver como ficou o template do componente `Login.vue`:

`src/components/Login.vue`

```
<template>
  <div class="container">
    <div class="row">
      <validator name="validation">
        <form class="form-signin" novalidate >
          <h2 class="center-block">Sales System Login</h2>
          <label for="email" class="sr-only">Email</label>
          <input type="email" id="email"
            class="form-control"
            placeholder="Email"
            autofocus
            v-model="user.email"
            v-validate:email="{ minlength: 4 }"
          >
          <label for="inputPassword" class="sr-only">Password\
rd</label>
          <input type="password" id="inputPassword"
            class="form-control"
            placeholder="Password"
            required v-model="user.password"
            v-validate:password="{ minlength: 6 }">
          <div class="checkbox">
```

```

    <label>
      <input type="checkbox"
        v-model="user.create_account">
      Create Account
    </label>
  </div>
  <div v-if="user.create_account">
    <label for="inputName" class="sr-only">Name</la\
bel>

    <input type="text" id="inputName"
      class="form-control"
      placeholder="Your Name"
      v-model="user.name"
      v-validate:name="{ minlength: 4 }"
    >

    <label for="inputName" class="sr-only">Role</la\
bel>

    <div class="radio">
      <label>
        <input type="radio" name="optionsRadios"
          id="optionsRadios1"
          value="Employee"
          checked v-model="user.role">
        Employee
      </label>
    </div>
    <div class="radio">
      <label>
        <input type="radio" name="optionsRadios"
          id="optionsRadios2"
          value="Customer"
          v-model="user.role">
        Customer

```

```

        </label>
    </div>
</div>
<br/>
<button class="btn btn-lg btn-primary btn-block"
@click.prevent="tryLogin"
:disabled="!$validation.valid||isLoading">
Ir</button>
<br/>
<loading></loading>
<error></error>
<div class="alert alert-danger"
v-show="$validation.email.touched&&$validation.em\
ail.minLength">
    Email too short</div>
<div class="alert alert-danger"
v-show="$validation.password.touched&&$validation\
.password.minLength">
    Password too short</div>
<div class="alert alert-danger"
v-show="$validation.name&&$validation.name.touche\
d&&$validation.name.minLength">
    Name too short</div>
</form>
</validator>
</div>
</div>
</template>
<script>
    ... script ...
</script>
<style scoped>
    .form-signin {

```

```
    max-width: 330px;
    padding: 15px;
    margin: 0 auto;
}

.form-signin .form-signin-heading,
.form-signin .checkbox {
    margin-bottom: 10px;
}

.form-signin .checkbox {
    font-weight: normal;
}

.form-signin .form-control {
    position: relative;
    height: auto;
    -webkit-box-sizing: border-box;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
    padding: 10px;
    font-size: 16px;
}

.form-signin .form-control:focus {
    z-index: 2;
}

.form-signin input[type="email"] {
    margin-bottom: -1px;
    border-bottom-right-radius: 0;
    border-bottom-left-radius: 0;
}
```

```
.form-signin input[type="password"] {  
  margin-bottom: 10px;  
  border-top-left-radius: 0;  
  border-top-right-radius: 0;  
}  
</style>
```

O template é extenso, e copia a ideia do html [deste link](#)².

Alguns pontos importantes deste html são descritos a seguir:

- O html “<validator>” é o VueValidator instalado previamente. Com ele podemos validar os campos do formulário. A validação é aplicada na diretiva `v-validate`
- O `v-model` usa o objeto `user` que será enviado por ajax para o login.
- O `user.create_account` é um booleano integrado ao campo `checkbox` que, quando selecionado, exibe mais algumas informações para que o usuário possa se cadastrar.
- O botão para efetuar o login possui o “@click.prevent”. O `prevent` irá executar o `preventDefault` do javascript. para que a tela não seja recarregada, o que é um comportamento comum para um botão que está inserido no `<form>`.
- O botão possui o `bind :disabled` que desabilita o botão se: 1) A validação não for válida 2) A variável `isLoading` do `getter` for `true`, geralmente quando uma requisição ajax está sendo carregada.
- Os controles já criados `<loading>` e `<error>` são inseridos após o botão. Nos formulários que iremos criar ao longo do sistema, podemos escolher onde colocar tais controles. O `<loading>` irá exibir um *spinner* indicando uma atividade Ajax no formulário, e o `<error>` irá exibir alguma mensagem de erro do servidor. Na verdade, tanto *loading* quanto *error* são controles reativos ao `vuex`. Eles observam as propriedades `store.error.message`.
- Exibimos algumas mensagens de erro usando o bootstrap, com as classes *alert* *alert-danger*. Essas mensagens são exibidas somente se houverem erros na validação.

²<http://getbootstrap.com/examples/signin/>

Além do <template>, temos o <style scoped> que irá embutir os estilos css no formulário. Todos os estilos foram copiados do [formulário original](#)³ e não são o foco desta obra.

14.5 A action setLogin

O botão para logar chama a action setLogin, exibida a seguir:

src/vuex/actions.js

```
import {URL} from '../config.js'

export function setLogin({dispatch},user){
  dispatch("SHOW_LOADING")
  this.$http.post(`${URL}/login`,user).then(response=>{
    dispatch("HIDE_LOADING")
    console.log("setLogin",response.data)
    dispatch("SET_LOGIN",response.data)
  },
  (error)=>{
    dispatch("HIDE_LOADING")
    console.log(error)
    dispatch("SHOW_ERROR",error.body||"Network error")
  }
  )
}

export function setError({dispatch},msg){}

export function clearError({dispatch}){}
```

Inicialmente, o setLogin dispara o mutation SHOW_LOADING que irá exibir a mensagem de “carregando” em algum lugar. Perceba que estamos alterado o nosso store

³<http://getbootstrap.com/examples/signin/>

e isso reflete nos getters que a camada de visualização usa. No caso do login, o controle `<loading>` que foi adicionado irá exibir um spinner (a imagem *loading.gif* que criamos em um capítulo anterior).

Usamos o `this.$http.post` para realizar um post (usando o Vue Resource) para o endereço “URL+/login”, onde a URL está configurada no arquivo `config.js`. O post repassa a variável `user` que será enviada no formato JSON para o servidor. Temos o uso de Promise para capturar os dois eventos de resposta do servidor, que pode ser o `(response)` se o servidor retornar sem erros (status 200). Neste caso, disparamos o mutation `HIDE_LOADING` que irá ocultar o spinner do controle `<loading>` e disparamos outro mutation, `SET_LOGIN`, repassando o `response.data` que será atribuído ao `state.login`. Quando isso acontecer, diversas informações sobre o login serão atribuídas, dentre elas o `state.login.token`, que é o token de autenticação criado pelo servidor, e que indicará que o usuário encontra-se logado.

Se por acaso houver um erro ao acessar o login no servidor, o evento `(error)` será chamado (ao invés do `response`). Neste caso, além do `HIDE_LOADING` também chamamos o mutation `SHOW_ERROR`, repassando o erro pelo `error.body` (que são justamente as mensagens de Exception do php).

14.6 O componente Admin.vue

A princípio, o componente `Admin.vue` possui somente este código:

`src/components/Admin.vue`

```
<template>
  Admin
</template>
<script>
  export default{

  }
</script>
<style></style>
```

14.7 Reprogramando o App.vue

Agora que conseguimos efetuar o login, podemos voltar ao componente App.vue e escrever o código que irá alterar entre a tela de login e a tela de administração do sistema.

src/App.vue

```
<template>
  <div id="app">
    <Login v-if="!isLoggedIn"></Login>
    <Admin v-else></admin>
  </div>
</template>

<script>

  import store from './vuex/store'
  import {isLoggedIn} from './vuex/getters'

  import Login from './components/Login.vue'
  import Admin from './components/Admin.vue'

  export default {
    components: {
      Login, Admin
    },
    data () {
      return {
        msg: 'Hello Vue!'
      }
    },
    store,
    vuex: {
```



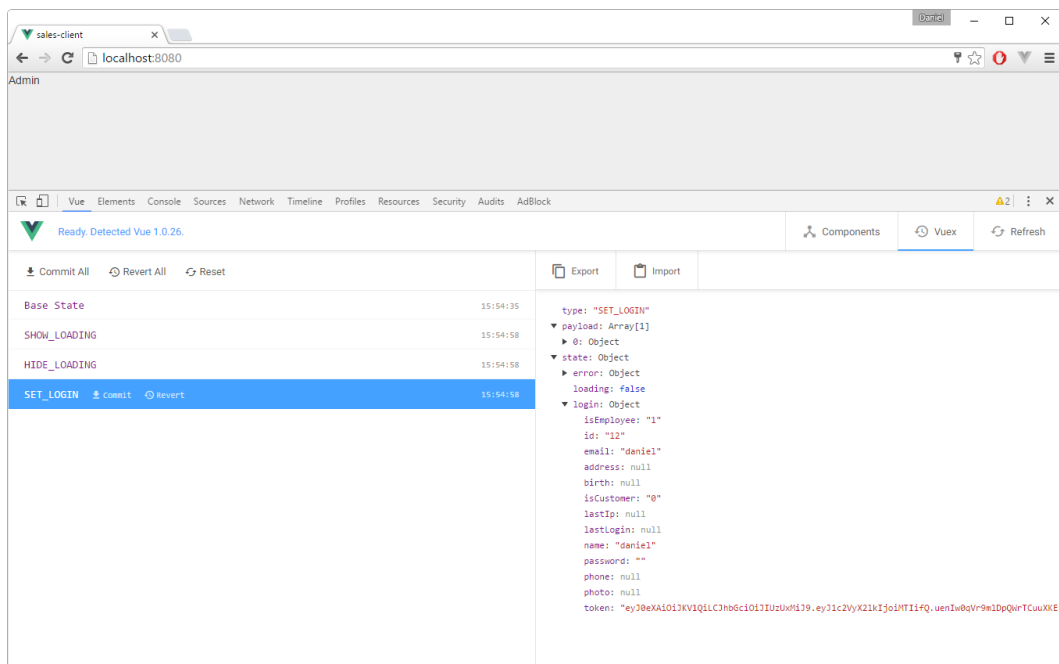
```
      getters: {
        isLoggedIn
      }
    }
  }
</script>
<style>
  body {
    background-color: #eee;
  }
</style>
```

Veja que o template inclui os dois componentes que criamos: <Login> e <Admin>. O login é exibido somente se o getter isLoggedIn for verdadeiro, isto é, somente se a variável state.login.token for nula. Caso contrário, o Admin será exibido.

Como o vue usa design reativo, assim que o login executar a mutation SET_LOGIN repassando o user.token, este será preenchido e o getter isLoggedIn será true, alterando assim do <Login> para o <Admin>.

A partir deste momento, com o user.token preenchido, podemos simular que o usuário está logado, e começar a programar a tela de Admin.

Inicialmente temos então a seguinte tela:



14.8 Persistindo o login

Até o momento, quando efetuamos o login, armazenamos esta informação no vuex (state.login). Quando recarregamos a página, esta informação se perde, pois ele está presente apenas na memória. É necessário, de alguma forma, persistir este login em um local, como o Local Storage.

O `LocalStorage` permite guardar pequenas informações de texto e iremos usá-lo exatamente para isso, para guardar o login. Para isso, edite o mutation `SET_LOGIN` para:

src/vuex/store.js

```
...
SET_LOGIN (state, login) {
  state.login = login
  localStorage.setItem("login", JSON.stringify(state.login));
},
...
```

Agora estamos, além de armazenar o valor em memória pelo `state.login`, usando o `localStorage` para armazenar o login, usando o `JSON.stringify` para converter o objeto em texto puro no formato JSON. Com isso, após efetuar o login, o mesmo será armazenado no `localStorage`.

Agora precisamos carregar este login quando a aplicação é recarregada. Para isso, criamos uma nova action, exibida a seguir:

src/vuex/actions.js

```
...
export function setLoginFromLocalStorage({dispatch}) {
  let login = JSON.parse(localStorage.getItem("login"))
  if ((login !== null) && (login.token !== null)) {
    dispatch("SET_LOGIN", login)
  }
}
...
```

Este método tenta obter o conteúdo que está no Local Storage, e se existir, ele irá chamar o mutarion `SET_LOGIN`, como se o usuário tivesse terminado de logar. Desta forma, mesmo fechando o navegador e reabrindo, retornado ao sistema, o usuário será redirecionado para a tela de Admin, não repassando para o login, conforme as aplicações fazem normalmente.

14.9 Tela de Admin

A tela de administração do sistema será formada por um menu superior, contendo os itens:

- Usuários
- Fornecedores
- Categorias
- Produtos
- Vendas
- PDV
- Logout

Os 5 primeiros itens serão cruds comuns, contendo um grid que irá exibir os dados de cada tabela, um campo de busca pelo nome, botões para editar o item e excluir o item.

O PDV irá exibir uma tela diferenciada, onde o usuário logado irá realizar uma venda completa, cadastrando o cliente, adicionando itens e fechando a compra.

14.10 Menu da tela Admin

O menu é formado pela navbar do bootstrap. Inicialmente, teremos o seguinte código:

src/components/Admin.vue

```
<template>
  <nav class="navbar navbar-inverse">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapse\
d" data-toggle="collapse" data-target="#bs-example-navbar-c\
ollapse-1"
          aria-expanded="false">
```

```

    <span class="sr-only">Toggle navigation</span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
  </button>
  <a class="navbar-brand" href="#">Sistema de vendas</a>
</div>

<div class="collapse navbar-collapse" id="bs-example-na\
vbar-collapse-1">

  <ul class="nav navbar-nav navbar-right">
    <li><a href="#">Usuários</a></li>
    <li><a href="#">Fornecedores</a></li>
    <li><a href="#">Categorias</a></li>
    <li><a href="#">Produtos</a></li>
    <li><a href="#">Vendas</a></li>
    <li><a href="#">PDV</a></li>
    <li><a href="#">Logout</a></li>
  </ul>
</div>
<!-- /.navbar-collapse -->
</div>
<!-- /.container-fluid -->
</nav>

</template>
<script>
  export default{

  }
</script>
<style></style>

```

Apenas o html é complexo (retirado [deste link](#)⁴), existem dois pontos nele que fizemos algumas alterações. Primeiro, adicionamos o nome do sistema no navbar-brand e, depois, criamos um simples menu no posterior que, a princípio, ainda não está totalmente configurado. Para que nós possamos configurá-lo, deve-se implementar o Vue Router.

Para incluir o VueRouter na aplicação, o arquivo main.js é alterado de acordo com o código a seguir:

src/main.js

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

import VueValidator from 'vue-validator'
import VueResource from 'vue-resource'
```

```
import VueRouter from 'vue-router'
import Routes from './routes.js'
```

```
Vue.use(VueValidator)
Vue.use(VueResource)
```

```
Vue.use(VueRouter)
const router = new VueRouter({
  linkActiveClass: 'active',
})
router.map(Routes)
router.start(App, 'App')
```

```
new Vue({
  el: 'body',
```

⁴<http://getbootstrap.com/examples/starter-template/>

```
—components: { App }  
}}
```

Na configuração do router, o item “linkActiveClass” irá ajudar no menu da aplicação a marcar o item na qual a rota está ativa. Também adicionamos o arquivo `routes.js` pelo método `router.map`, que conterà as rotas do Router:

`src/routes.js`

```
import Users from './components/Users.vue'  
  
const Routes = {  
  '/users': {  
    component: Users  
  }  
}  
  
export default Routes;
```

Inicialmente temos apenas a rota `/users` apontando para o componente `Users.vue`, que deve ser criado a seguir:

`src/components/Users.vue`

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

Voltando ao menu do `Admin.vue`, já podemos configurar a rota para o link “Usuários”:

src/components/Admin.vue

```

...
<ul class="nav navbar-nav navbar-right">
  <li v-link-active><a v-link="{ path: '/users' }">Usuários\
</a></li>
  <li><a href="#">Fornecedores</a></li>
  <li><a href="#">Categorias</a></li>
  <li><a href="#">Produtos</a></li>
  <li><a href="#">Vendas</a></li>
  <li><a href="#">PDV</a></li>
  <li><a href="#">Logout</a></li>
</ul>

```

Veja que usamos o `v-link-active` que será responsabilidade do VueRouter adicionar a classe “active” quando a rota coincidir com o link. Também usamos o `v-link` para relacionar o item a de menu a rota “/users”.

Para finalizar, precisamos configurar onde o conteúdo do componente carregado pela rota será carregado. Isso é feito logo abaixo do menu, no próprio componente Admin.vue, onde adicionamos o `<router-view>`, veja:

```

<template>

  <nav class="navbar navbar-inverse">
    ... navbar com o menu ...
  </nav>
  <div class="container">
    <router-view></router-view>
  </div>

</template>
<script>
  export default{

```



```
    }  
</script>
```

Após adicionar o <router-view>, recarregue a aplicação e verifique no navegador se, ao clicar no menu “Usuários”, o conteúdo de `Users.vue` é carregado.

Para terminar a configuração do router, crie as outras telas. No final, teremos a seguinte configuração:

```
import Users from './components/Users.vue'  
import Suppliers from './components/Suppliers.vue'  
import Categories from './components/Categories.vue'  
import Products from './components/Products.vue'  
import Sales from './components/Sales.vue'  
import PDV from './components/PDV.vue'  
import Logout from './components/Logout.vue'  
  
const Routes = {  
  '/users': {  
    component: Users  
  },  
  '/suppliers': {  
    component: Suppliers  
  },  
  '/categories': {  
    component: Categories  
  },  
  '/products': {  
    component: Products  
  },  
  '/sales': {  
    component: Sales  
  },  
  '/pdv': {
```

```

      component: PDV
    },
    '/logout': {
      component: Logout
    },
  },
}

```

```
export default Routes;
```

Lembre-se que você tem q criar os componentes Categories.vue, Users.vue etc.

src/components/Admin.vue

```

...
<ul class="nav navbar-nav navbar-right">
  <li v-link-active><a v-link="{ path: '/users' }">Usuários\
</a></li>
  <li v-link-active><a v-link="{ path: '/suppliers' }">Forn\
ecedores</a></li>
  <li v-link-active><a v-link="{ path: '/categories' }">Cat\
egorias</a></li>
  <li v-link-active><a v-link="{ path: '/products' }">Produ\
tos</a></li>
  <li v-link-active><a v-link="{ path: '/sales' }">Vendas</\
a></li>
  <li v-link-active><a v-link="{ path: '/pdv' }">PDV</a></l\
i>
  <li v-link-active><a v-link="{ path: '/logout' }">Logout<\
/a></li>
</ul>

```

14.11 Criando a tela de Categorias

Começaremos pela tela de categorias, que possui apenas dois campos, id e name. Inicialmente, vamos dar uma olhada em cada parte do Vuex relativo as categorias, veja:

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    itens_per_page: 10,
    error: {
      message: ""
    },
    loading: false,
    login: {
      name: null,
      email: null,
      token: null
    },
    category: {
      list: [],
      selected: {},
      total: 1,
      page: 1
    }
  },
  mutations: {
    SET_LOGIN(state, login) {
      state.login = login
    }
  }
})
```

```
    localStorage.setItem("login", JSON.stringify(state.login));
  },
  SHOW_LOADING(state) {
    state.loading = true;
  },
  HIDE_LOADING(state) {
    state.loading = false;
  },
  SHOW_ERROR(state, msg) {
    state.error.message = msg || "Network error";
    setTimeout(function () {
      state.error.message = ""
    }, 5000)
  },
  HIDE_ERROR(state) {
    state.error.message = "";
  },
  SET_CATEGORIES(state, categories) {
    state.category.list = categories;
  },
  SET_CATEGORY(state, category) {
    state.category.selected = category;
  },
  SET_TOTAL_CATEGORIES(state, total) {
    state.category.total = total;
  },
  SET_PAGE_CATEGORY(state, page) {
    state.category.page = page;
  }
}
})
```

O state category possui diversas propriedades:

- **list** O array de dados que irá compor a tabela
- **selected** O objeto selecionado, que irá compor o formulário
- **total** A quantidade total de registros, que será usada para formar a paginação
- **page** A página atual em que a tabela está

Também temos vários mutations que gerenciam estas propriedades, como o SET_CATEGORIES, o SET_TOTAL_CATEGORIES e o SET_PAGE_CATEGORY.

src/vuex/actions.js

```
export function setCategory({dispatch}, category)
{
  dispatch("SET_CATEGORY", category);
}

export function saveCategory({dispatch, state})
{
  dispatch("SHOW_LOADING")

  this.$http.post(`${URL}/category`, state.category.selected)
    .then(response=>{
      dispatch("SET_CATEGORY", response.json())
    },
    error=>{
      dispatch("SHOW_ERROR", error.body)
    }
  ).finally(function(){
    dispatch("HIDE_LOADING")
    this.loadCategories();
  })
}
```

```
export function loadCategories({dispatch, state})
{
  dispatch('SHOW_LOADING')

  let start = (state.category.page * state.itens_per_page) \
- (state.itens_per_page - 1);

  this.$http.get(`${URL}/categories?start=${start}&limit=${\
state.itens_per_page}`).then(response=>{
    dispatch("SET_CATEGORIES",response.json())
    dispatch('SET_TOTAL_CATEGORIES',response.headers['x-tot\
al-count']);
  },error=>{
    dispatch("SHOW_ERROR", error.body)
  })
  .finally(function(){
    dispatch("HIDE_LOADING")
  })
}
```

As ações (actions.js) são responsáveis em acessar o servidor e disparar os mutations com os dados obtido. Na action `saveCategory`, realizamos um POST para a url `"/category"`, repassando o `state.category.selected` que é o objeto que está presente no formulário.

Na action `loadCategories`, criamos a variável `start` que irá obter o registro inicial a ser buscado no banco dado o número da página que será manipulado pela paginação.

Depois, usamos o `this.$http.get` chamando a url `"/categories"`, e repassando uma query string com os valores `start` e `limit`. A variável `limit` é justamente a quantidade de itens por página, configurada no `state` do `store.js`. Com promises (do javascript), temos duas situações: *response* ou *error*. No *response*, a requisição foi realizada com sucesso e o servidor retornou com os dados. Com isso, podemos chamar o mutation que irá alterar a lista de categorias (`SET_CATEGORIES`), e também chamamos o mutation que irá alterar o valor total de registros (`SET_TOTAL_CATEGORIES`). Perceba

que neste total, nós estamos obtendo este valor através do cabeçalho de resposta da requisição.

Em caso de error, disparamos o mutation responsável em exibir este erro.

Também temos o `.finally` que é um *callback* do *Promise* que é disparado sempre, independente do *response* ou *error* ter sido executado. Neste caso, o usamos para esconder a mensagem de carregando do componente.

14.11.1 Autenticação

Em ambas as requisições, é necessário enviar o token de autenticação do cliente ao servidor. Repare que no código esse token não foi adicionado, já que existe um meio mais simples de se fazer isso.

Vamos alterar o arquivo `Admin.vue` e usar a configuração global `Vue.http.headers` para adicionar o token, a qualquer requisição do cliente ao servidor, veja:

`src/components/Admin.vue`

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

<script>
  import Vue from 'vue'
  import {getLogin} from '../vuex/getters'
  export default{
    vuex:{
      getters: {
        getLogin
      }
    },
    ready(){
      Vue.http.headers.common['x-access-token'] = this.getLogin\
      ogin.token;
```

```

    }
  }
</script>

```

Neste código, usamos o `Vue.http.headers.common['x-access-token']` para fornecer o token de autenticação a qualquer requisição do cliente para o servidor. O valor do token é obtido pelo getter.

14.11.2 Template

Agora que temos o Vuex configurado, podemos criar a tela no componente `Categories.vue`, veja:

`src/components/Categories.vue`

```

<template>

  <div class="panel panel-default">
    <div class="panel-heading">
      <div class="row">
        <div class="col-xs-6">
          <h4>Categorias <small>({{getTotalCategories}})</small></h4>
        </div>
        <div class="col-xs-6">
          <button @click.prevent="newCategory" class="btn btn-default pull-right">Novo</button>
        </div>
      </div>
    <div class="panel-body">
      <div class="row">
        <div class="col-md-6">
          <table class="table table-bordered">

```



```

    <thead>
      <tr>
        <th>Id</th>
        <th width="100%">Nome</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="category in getCategories">
        <td>{{category.id}}</td>
        <td>{{category.name}}</td>
      </tr>
    </tbody>
  </table>
</div>
<div class="col-md-6">
  <Error></Error>
  <form>
    <div class="form-group">
      <label for="name">Id</label>
      <input type="input" class="form-control" id="id\
" placeholder="id" v-model="getCategory.id" readonly >
    </div>
    <div class="form-group">
      <label for="name">Nome</label>
      <input type="input" class="form-control" id="na\
me" placeholder="Nome" v-model="getCategory.name">
    </div>
    <button @click.prevent="trySaveCategory" class="b\
tn btn-default" :disabled="isLoading">Salvar</button>
    <Loading></Loading>
  </form>
</div>
</div>

```

```

    </div>
  </div>
</div>
</template>
<script>
  import {setCategory,saveCategory,loadCategories} from '../\
/vuex/actions.js'
  import {getCategory,getCategories,getTotalCategories,isLo\
ading} from '../vuex/getters.js'
  import Loading from '../controls/Loading.vue'
  import Error from '../controls/Error.vue'

  export default{
    components: {
      Loading, Error
    },
    vuex:{
      actions:{
        setCategory,saveCategory,loadCategories
      },
      getters:{
        getCategory,isLoading,getCategories,getTotalCategor\
ies
      }
    },
    created(){
      this.loadCategories();
    },
    methods:{
      newCategory(){
        this.setCategory({});
      },
      trySaveCategory(){

```

```
        this.saveCategory();
    }
}
</script>
```

Este template irá mostrar a seguinte tela:

Id	Nome
1	aaaaaa
2	bbbbbb
3	sssss
4	teste
5	aaaa
6	sdsdsd
7	asasas
8	asasas
9	asasas
10	sqsqsqsq

Id

Nome

Salvar

À esquerda temos uma tabela com os 10 primeiros registros, que são carregados graças a action `loadCategories` executada no evento `created` do componente. O botão “novo” irá chamar o método `newCategory` que a princípio chama a action `setCategory`, repassando um objeto vazio.

Perceba que todo o componente não possui código de manipulação de dados, graças do desenvolvimento reativo que está observando o state do vuex. Por exemplo, a lista de dados que são carregados na tabela observa o getter `getCategories` que por sua vez observa o `state.category.list`. Sempre quando este objeto mudar, a tabela mudará automaticamente.

Veja os componentes `<Loading>` e `<Error>` sendo adicionados no formulário. Ambos também são reativos, então quando alguma action dispara o mutation `SHOW_LOADING`, o state `loading` torna-se `true` e o componente passa a mostrar o ícone de “carregando”.

É importante compreender esta sinergia entre o html, os getters e as actions, de forma a aproveitar o máximo os conceitos envolvidos. ### Editar uma categoria

Como exercício, pense em uma forma de editar o item. Você pode adicionar um link na lista de categorias formada pela tabela:

```
<td><a @click.prevent="tryEdit(category)">{{category.name}}\n</a></td>
```

Onde o método tryEdit irá usar o action setCategory repassando o objeto selecionado:

```
tryEdit(category){\n  this.setCategory(category);\n}
```

14.12 Paginação de categorias (e busca)

Para criar a paginação de categorias, podemos criar um componente chamado *Pagination*, que ficará no diretório *controls*. Esta paginação será genérica a qualquer tabela, sendo necessária duas informações principais:

- A quantidade total de itens: Isso será necessário para calcular a quantidade de páginas
- A quantidade de itens por página: Isso será necessário para calcular a quantidade de páginas
- A página atual da lista: Quando avançamos a paginação, precisamos saber qual a nova página a ser exibida

O componente de paginação usa os elementos do bootstrap. O seu código é descrito a seguir:

src/controls/Pagination.vue

```
<template>
  <nav>
    <ul class="pagination ">
      <li>
        <a href="#" aria-label="Previous" v-show="showPreviousButton" @click.prevent="goPreviousPage()">
          <span aria-hidden="true" >&laquo;</span>
        </a>
      </li>
      <li><a href="#" @click.prevent="goPreviousPage" v-show="showPreviousButton">{{page-1}}</a></li>
      <li class="active"><a>{{page}}</a></li>
      <li><a href="#" @click.prevent="goNextPage" v-show="showNextButton">{{page+1}}</a></li>
    </ul>
  </nav>
</template>
<script>
  export default{
    props: ["total", "page", "itensPerPage"],
    computed: {
      totalPages: function(){
        return Math.ceil(this.total / this.itensPerPage) || 1
      },
      showNextButton: function(){
        return this.page !== this.totalPages
      },
      showPreviousButton: function(){
        return this.page !== 1
      }
    },
  },
}
```

```
methods: {
  goNextPage: function(){
    this.$emit('change-page',this.page+1)
  },
  goPreviousPage: function(){
    this.$emit('change-page',this.page-1)
  },
  goFirstPage: function(){
    this.$emit('change-page',1)
  },
  goLastPage: function(){
    this.$emit('change-page',this.totalPages)
  },
  goPage: function(page){
    this.$emit('change-page',page)
  }
}
</script>
```

Na parte HTML, usamos (e abusamos!) da reatividade do Vue. Por exemplo, o botão para retornar uma página:

```
<a href="#" aria-label="Previous" v-show="showPreviousButto\
n" @click.prevent="goPreviousPage()">
  <span aria-hidden="true" >&laquo;</span>
</a>
```

Usa o v-show para que somente seja exibido quando a página é diferente da primeira:

```
showPreviousButton: function(){
  return this.page!=1
}
```

Para compor os números da página, usa-se o template `{{page-1}}` por exemplo, que vai exibir a página atual menos 1. Então se estivermos na página 3, o número 2 será impresso.

Usamos props para determinar os parâmetros da paginação, isso significa que estes parâmetros deverão repassados pelo componente pai. Também usamos o evento `change-page` que é disparado quando o usuário clica em algum link da paginação. Quem vai lidar com essa mudança de página é o componente pai, que irá atualizar a sua referida page, e o `Pagination` usa a reatividade para cuidar do resto.

Voltando ao componente `Categorias.vue`, nós inserimos a paginação abaixo da tabela, veja:

src/components/Categories.vue

```
<template>

  ...
  <table class="table table-bordered table-hover">
    ...
  </table>
  <div class="text-center">
    <Pagination :total="getTotalCategories" :items-per-page="\
getItemsPerPage" :page="getCategoryPage" @change-page="onCh\
angePage"></Pagination>
  </div>
  ...
</template>
<script>
  import {setCategory,saveCategory,loadCategories,changeCat\
egoriesPage} from '../vuex/actions.js'
  import {getCategory,getCategories,getTotalCategories,isLo\
```

```
ading, getCategoryPage, getItensPerPage} from '../vuex/getter\
s.js'

import Loading from '../controls/Loading.vue'
import Error from '../controls/Error.vue'
import Pagination from '../controls/Pagination.vue'

export default{
  components: {
    Loading, Error, Pagination
  },
  vuex: {
    actions: {
      setCategory, saveCategory, loadCategories, changeCateg\
oriesPage
    },
    getters: {
      getCategory, isLoading, getCategories, getTotalCategor\
ies, getCategoryPage, getItensPerPage
    }
  },
  created() {
    this.loadCategories();
  },
  data() {
    return {
      keyword: ""
    }
  },
  methods: {
    newCategory() {
      this.setCategory({});
    },
    trySaveCategory() {
```



```
        this.saveCategory(this.keyword);
    },
    tryEdit(category){
        this.setCategory(category);
    },
    onChangePage(page){
        this.changeCategoriesPage(page, this.keyword)
    },
    trySearch(){
        this.changeCategoriesPage(1, this.keyword)
        this.loadCategories(this.keyword)
    }
}
}
</script>
```

Incluimos o <Pagination>, juntamente com mais alguns métodos como o `getItemsPerPage`. Também criamos o método que responde ao evento `changepage`, chamado de `onChangePage`, que irá disparar a action `changeCategoriesPage`, que por sua vez altera o `state.category.page`.

Perceba também a variável `keyword` que será usada para buscas. Sempre que o usuário adicionar um novo registro ou mudar a página, essa busca deve ser preservada. Por isso ela é repassada pelos métodos `loadCategories` e `saveCategory`.

Voltando ao `actions.js`, temos algumas pequenas mudanças para adicionar o termo `keyword` no gerenciamento de categorias, veja:

src/vuex/actions.js

```
export function saveCategory({dispatch, state},keyword){
  dispatch("SHOW_LOADING")

  this.$http.post(`${URL}/category`, state.category.selecte\
d).then(response => {
    dispatch("SET_CATEGORY", response.json())
  },
  error => {
    dispatch("SHOW_ERROR", error.body)
  }
  ).finally(function () {
    dispatch("HIDE_LOADING")
    this.loadCategories(keyword);
  })
}

export function loadCategories({dispatch, state},keyword){
  dispatch('SHOW_LOADING')

  let start = (state.category.page * state.itens_per_page) \
- (state.itens_per_page - 1);

  let keywordString=""
  if (keyword!=null){
    keywordString=`&q=${keyword}`
  }

  this.$http.get(`${URL}/categories?start=${start}&limit=${\
state.itens_per_page}${keywordString}`).then(response => {
    dispatch("SET_CATEGORIES", response.json())
    dispatch('SET_TOTAL_CATEGORIES', response.headers['x-to\
```

```
tal-count']]);
  },
  error => {
    dispatch("SHOW_ERROR", error.body)
  }
).finally(function () {
  dispatch("HIDE_LOADING")
})
}

export function changeCategoriesPage({dispatch, state}, page\
e, keyword) {
  dispatch('SET_CATEGORY_PAGE', page)
  this.loadCategories(keyword)
}
```

Após realizar estas alterações, temos uma tela de cadastro de categorias quase 100% funcional, necessitando apenas da funcionalidade de apagar categorias, que será feito na próxima atualização.

Para verificar como ficou a tela de cadastro de categorias, acesse esta gif:

<http://i.imgur.com/LWCREX5.gifv>

14.13 Removendo uma categoria

Vamos criar um botão que possui a finalidade de excluir a categoria. Este botão pode ficar ao lado do botão salvar e pode-se usar a classe `pull-right` para mover o botão totalmente para a direita.

O código para incluir este botão é:

```

<button @click.prevent="trySaveCategory" class="btn btn-default" :disabled="isLoading">Salvar</button>
<Loading></Loading>
<button @click.prevent="tryDeleteCategory" class="btn btn-default pull-right" :disabled="isLoading||getCategory.id==null">
  Apagar
</button>

```

O botão é ativado somente se houver um registro no formulário, ou seja, se houver um id de categoria preenchido. O método trySaveCategory é exibido a seguir:

```

tryDeleteCategory(){
  if (confirm(`Deseja apagar "${this.getCategory.name}"s ?`)) {
    this.deleteCategory(this.keyword)
  }
}

```

Antes de chamar a action, usamos o método confirm para perguntar ao usuário se ele deseja realmente excluir o registro. Em caso positivo, o método deleteCategory criado na action será executado.

```

export function deleteCategory({dispatch,state},keyword)
{
  dispatch('SHOW_LOADING')
  this.$http.delete(`${URL}/category/${state.category.selected.id}`).then(reponse => {
    dispatch("SET_CATEGORY", {})
  },
  error => {
    dispatch("SHOW_ERROR", error.body)
  }
}

```

```

    ).finally(function(){
      dispatch("HIDE_LOADING")
      this.loadCategories(keyword);
    })
  }
}

```

Nesta action, usamos `this.$http.delete` para fazer uma requisição DELETE ao servidor, repassando o id da categoria. O parâmetro `keyword` é repassado ao `loadcategories` para que, se houver uma busca ativa, ela será preservada.

14.14 Validação

Uma última funcionalidade necessária ao formulário de cadastro e edição de categorias é a validação, realizada através do `vue-validator`.

A validação é adicionada diretamente no formulário, da seguinte forma:

```

<validator name="validateForm">
  <form>
    <div class="form-group">
      <label for="name">Id</label>
      <input type="input" class="form-control" id="id" placeholder="id" v-model="getCategory.id" readonly>
    </div>
    <div class="form-group">
      <label for="name">Nome</label>
      <input type="input" class="form-control" id="name" placeholder="Nome" v-model="getCategory.name" v-validate: name="{ required: true, minlength: 2 }">
      <div v-show="$validateForm.invalid">
        <span class="label label-info" v-if="$validateForm.name.required">Campo requerido</span>
      </div>
    </div>
  </form>

```

```

    <span class="label label-info" v-if="$validateForm.\
name.minlength">Mínimo 4 caracteres</span>
  </div>
</div>
  <button @click.prevent="trySaveCategory" class="btn btn\
-default" :disabled="isLoading||$validateForm.invalid" >Sal\
var</button>
  <Loading></Loading>
  <button @click.prevent="tryDeleteCategory" class="btn b\
tn-default pull-right" :disabled="isLoading||getCategory.id\
==null">Apagar</button>
</form>
</validator>

```

14.15 Algumas considerações sobre a tela de categorias

A tela de categorias está pronta, e com ela podemos ver diversas técnicas empregadas para o seu desenvolvimento. Tais técnicas, como o uso do Vuex, não são a única forma de se programar com vue. Você pode, por exemplo, optar em não usar o Vuex e ter todas as variáveis que estão no state implementadas no data do componente.

A variável keyword, por exemplo, que não está no state, teve um efeito colateral no qual ela deveria ser repassada em alguns métodos, como o método saveCategory.

O ponto em que queremos chegar é que, a forma como a tela de categorias foi programada não é a única forma e nem mesmo a única correta.

Para que possamos experimentar uma outra forma de se criar uma tela CRUD, vamos iniciar a criação da tela de cadastro de fornecedores (Suppliers).

14.16 Cadastro de Fornecedores

Nesta tela vamos criar um CRUD de fornecedores (suppliers), só que ao invés de usar Vuex como foi feito na tela de categorias, iremos deixar toda a implementação no controle Suppliers.vue.

Fazemos isso para que você possa experimentar estas duas formas de programar em vue, deixando a sua escolha a forma que melhor lhe agrada.

14.17 Alterando o actions.js, adicionando o showLogin e hideLoding

Como não vamos nesta tela usar o dispatch, temos que implementá-lo no actions, da seguinte forma:

src/vuex/actions.js

```
export function showLoading({dispatch}) {  
  dispatch("SHOW_LOADING");  
}
```

```
export function hideLoading({dispatch}) {  
  dispatch("HIDE_LOADING");  
}
```

14.18 Alterando o getter.js

O getter também recebe um novo método, que é usado para obter o número de itens de uma página:

```
export function itensPerPage(state){  
  return state.itens_per_page  
}
```

14.19 Incluindo o Suppliers.vue

Agora podemos alterar o Suppliers.vue para mostrar os fornecedores. No template, é muito semelhante ao crud de categorias, exceto pela mudança no nome das variáveis (que agora não estão mais no vuex), e na inclusão de um text area que representa o campo endereço.

src/components/Suppliers.vue

```

<template>
  <div class="panel panel-default">
    <div class="panel-heading">
      <div class="row">
        <div class="col-xs-6">
          <h4>Fornecedores <small>({{total}})</small></h4>
        </div>
        <div class="col-xs-6">
          <button @click.prevent="newSupplier" class="btn btn-\
tn-default pull-right">Novo</button>
        </div>
      </div>
    </div>
    <div class="panel-body">
      <div class="row">
        <div class="col-md-6">
          <div class="form-group form-inline">
            <input type="input" class="form-control" id="k\
eyword" name="keyword" placeholder="Buscar por nome" v-mode\
l="keyword">
            <button @click.prevent="search" class="btn btn-\
default">Buscar</button>
          </div>
          <table class="table table-bordered table-hover">
            <thead>
              <tr>
                <th>Id</th>
                <th width="100%">Nome</th>
              </tr>
            </thead>
            <tbody>

```



```

        <tr v-for="supplier in suppliers">
            <td>{{supplier.id}}</td>
            <td role="button" @click.prevent="edit(supplier)"><a>{{supplier.name}}</a></td>
        </tr>
    </tbody>
</table>
<div class="text-center">
    <Pagination :total="total" :items-per-page="itemsPerPage" :page="page" @change-page="onChangePage"></Pagination>
</div>
</div>
<div class="col-md-6">
    <Error></Error>
    <validator name="validateForm">
        <form>
            <div class="form-group">
                <label for="name">Id</label>
                <input type="input" class="form-control" id="id" placeholder="id" v-model="supplier.id" readonly>
            </div>
            <div class="form-group">
                <label for="name">Nome</label>
                <input type="input" class="form-control" id="name" placeholder="Nome" v-model="supplier.name" v-validate:{"required: true, minlength: 2"}>
                <div v-show="$validateForm.invalid">
                    <span class="label label-info" v-if="$validateForm.name.required">Campo requerido</span>
                    &nbsp;
                    <span class="label label-info" v-if="$validateForm.name.minlength">Mínimo 2 caracteres</span>
                </div>
            </div>
        </form>
    </validator>

```

```

        </div>
        <label for="name">Endereço</label>
        <textarea class="form-control" rows="5" id=\
"address" v-model="supplier.address"></textarea>

    </div>
    <button @click.prevent="saveSupplier" class="\
btn btn-default" :disabled="isLoading||$validateForm.invali\
d" >Salvar</button>
    <Loading></Loading>
    <button @click.prevent="deleteSupplier" class\
="btn btn-default pull-right" :disabled="isLoading||supplie\
r.id==null">Apagar</button>
    </form>
  </validator>
</div>
</div>
</div>
</div>
</div>
</template>

```

Na parte do script nota-se as maiores diferenças, veja:

```

import {isLoading,itemsPerPage} from '../vuex/getters.js'
import {showLoading,hideLoading,setError} from '../vuex/act\
ions.js'
import Loading from '../controls/Loading.vue'
import Error from '../controls/Error.vue'
import Pagination from '../controls/Pagination.vue'
import {URL} from '../config.js'

```

```
export default{
  components: {
    Loading, Error, Pagination
  },
  vuex:{
    getters:{
      isLoading, itensPerPage
    },
    actions:{
      showLoading, hideLoading, setError
    }
  },
  created(){
    this.loadSuppliers(this.keyword);
  },
  data(){
    return{
      keyword: "",
      suppliers: [],
      supplier: {},
      total: 1,
      page: 1
    }
  },
  methods:{
    newSupplier(){
      this.supplier = {}
    },
    saveSupplier(){
      this.showLoading();
      let t = this;
      this.$http.post(`${URL}/supplier`, this.supplier).then(\
response =>
```

```

    {
      t.supplier = response.json()
    },
    error => {
      t.setError(error.body)
    }
  ).finally(function () {
    t.hideLoading();
    t.loadSuppliers();
  })
},
edit(supplier){
  this.supplier=supplier
},
onChangePage(page){
  this.page = page
  this.loadSuppliers();
},
search(){
  this.page = 1
  this.loadSuppliers()
},
deleteSupplier(){
  if (confirm(`Deseja apagar "${this.supplier.name}"s ?\`
`)){
    this.showLoading()
    let t = this;
    this.$http.delete(`${URL}/supplier/${this.supplier.\
id}`).then(response => {
      t.supplier={}
    },
    error => {
      t.setError(error.body)
    }
  )
}

```

```

    }
    ).finally(function () {
      t.hideLoading()
      t.loadSuppliers();
    })
  }
},
loadSuppliers(){
  this.showLoading();
  let start = (this.page * this.itensPerPage) - (this.itensPerPage - 1);
  let keywordString=""
  if (this.keyword!=""){
    keywordString=`&q=${this.keyword}`
  }
  let t = this;
  this.$http.get(`${URL}/suppliers?start=${start}&limit=${this.itensPerPage}${keywordString}`).then(response => {
    t.suppliers = response.json()
    t.total= response.headers['x-total-count']
  },
  error => {
    t.setError(error.body)
  })
  ).finally(function () {
    t.hideLoading();
  })
}
}
}

```

Perceba que o objeto data possui várias informações sobre o crud, informações estas que estavam no store na tela de categorias. O método loadSuppliers usa o Vue

Resource para obter os dados de fornecedores do servidor. Ao invés de `dispatch`, usamos o método `this.showLoading()` que irá acessar a `action` e relizar o comando, para que o controle `<Loading>` possa responder a chamada `ajax`. O mesmo acontece com `hideLoading`.

Ainda no `loadSuppliers`, usamos uma referência ao `this` através da variável `t`, para que ele possa ser acessado no `callback` do `$http.get` com mais facilidade.

Deixamos a cargo do leitor o método de acesso a dados que deseja usar, lembrando que não é aconselhável sobrecarregar o `store` do `vuex` com todos os dados da aplicação. Lembre-se de usar o `vuex` somente quando há a necessidade de compartilhar a mesma informação entre os componentes.

14.20 Incluindo a tela de cadastro de produtos

O cadastro de produtos é um pouco mais complexo que as outras telas, e implementa o conceito de *services* que podem ser usados caso você goste. Lembre que nesse exemplo de `Vue+Slim Framework` estamos testando várias alternativas, e esta é uma frequentemente usada (*business services*).

Crie o diretório `src/services`, com estes três arquivos:

`src/services/Category.js`

```
import {URL} from '../config.js'
import Vue from 'vue'

export default class CategoryService{

  static getAll(){
    return Vue.http.get(`${URL}/categories`);
  }

}
```

Perceba que o `service CategoryService` é uma classe (definido pelo “`export default class`”) e que possui um método estático “`static`” chamado `getAll`, que retorna um

Promise gerado pelo `Vue.http.get`. A maioria dos métodos de um service são estáticos e retornam um Promise. Esta não é uma regra fixa, mas um caminho válido. Neste service, retornamos todas as categorias (que irá futuramente preencher uma caixa *select*).

src/services/Supplier.js

```
import {URL} from '../config.js'
import Vue from 'vue'

export default class SupplierService{

  static getAll(){
    return Vue.http.get(`${URL}/suppliers`);
  }

}
```

O service de Fornecedores é muito semelhante ao service de categorias.

src/services/Product.js

```
import {URL} from '../config.js'
import Vue from 'vue'

export default class ProductService{

  static save(product){
    return Vue.http.post(`${URL}/product`,product)
  }

  static delete(id){
    return Vue.http.delete(`${URL}/product/${id}`)
  }

}
```

```

static getAll(page, itensPerPage, keyword=null){

    let start = (page * itensPerPage) - (itensPerPage - 1);
    let keywordString=""
    if (keyword!=null){
        keywordString=`&q=${keyword}`
    }
    return Vue.http.get(`${URL}/products?start=${start}&lim\
it=${itensPerPage}${keywordString}`)
}

}

```

O service de Produtos é um pouco mais complexo, possui os métodos: save, delete e getAll. Isso acontece porque os métodos save e delete dos outros services (Supplier e Category) estão usando outras metodologias. Neste caso, o “save” do *Category* está no *vuex* e o save do *Product* está diretamente no componente *Product.vue*. Perceba que todos os métodos do *ProductService* são estáticos e retornam Promises.

No arquivo *src/components/Product.vue* todo o código para exibir e alterar produtos é exibido a seguir:

src/components/Products.vue

```

<template>
  <div class="panel panel-default">
    <div class="panel-heading">
      <div class="row">
        <div class="col-xs-6">
          <h4>Produtos <small>({{total}})</small></h4>
        </div>
        <div class="col-xs-6">
          <button @click.prevent="newProduct" class="btn btn\
n-default pull-right">Novo</button>
        </div>
      </div>
    </div>
  </div>

```



```

    </div>
  </div>
  <div class="panel-body">
    <div class="row">
      <div class="col-md-12">

        <div class="form-group form-inline">
          <input type="input" class="form-control" id="k\
eyword" name="keyword" placeholder="Buscar por nome" v-mode\
l="keyword">
          <button @click.prevent="search" class="btn btn-\
default">Buscar</button><Loading></Loading>
        </div>

        <table class="table table-bordered table-hover">
          <thead>
            <tr>
              <th>Id</th>
              <th width="100%">Nome</th>
            </tr>
          </thead>
          <tbody>
            <tr v-for="product in products">
              <td>{{product.id}}</td>
              <td role="button" @click.prevent="edit(prod\
uct)"><a>{{product.name}}</a></td>
            </tr>
          </tbody>
        </table>

        <div class="text-center">

```

```

        <Pagination :total="total" :items-per-page="ite\
nsPerPage" :page="page" @change-page="onChangePage"></Pagin\
ation>

        </div>

    </div>
</div>
</div>
</div>
</div>

<!-- Modal new/edit -->

<div id="productModal" class="modal fade" tabindex="-1" rol\
e="dialog">
    <div class="modal-dialog">

        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="mod\
al" aria-label="Close"><span aria-hidden="true">&times;</sp\
an></button>
                <h4 class="modal-title">{{product.id==null?'Novo':'Ed\
itar'}} Produto</h4>
            </div>
            <div class="modal-body">

                <Error></Error>

                <validator name="validateForm">
                    <form>

```

```
<div class="row">

  <div class="form-group col-xs-12 col-sm-4">
    <label for="id">Id</label>
    <input type="input" class="form-control" id="\
id" placeholder="id" v-model="product.id" readonly>
  </div>

  <div class="form-group col-xs-12 col-sm-8">
    <label for="code">Código</label>
    <input type="input" class="form-control" id="\
code" placeholder="id" v-model="product.code">
  </div>

</div>

<div class="row">

  <div class="form-group col-xs-12 col-sm-6">
    <label for="idCategory">Categoria</label>
    <select id="idCategory" class="form-control" \
v-model="product.idCategory">
      <option v-for="category in categories" value\
="{{category.id}}">
        {{category.name}}
      </option>
    </select>
  </div>

  <div class="form-group col-xs-12 col-sm-6">
    <label for="idSupplier">Fornecedor</label>
    <select id="idSupplier" class="form-control" v-m\
odel="product.idSupplier">
```

```

        <option v-for="supplier in suppliers" value="{\
{supplier.id}}">
            {{supplier.name}}
        </option>
    </select>
</div>

</div>

<div class="form-group">
    <label for="name">Nome</label>
    <input type="input" class="form-control" id="name" \
placeholder="Nome" v-model="product.name" v-validate:name="\
{ required: true, minlength: 2 }">
    <div v-show="$validateForm.invalid">
        <span class="label label-info" v-if="$validateFor\
m.name.required">Campo requerido</span>
        &nbsp;
        <span class="label label-info" v-if="$validateFor\
m.name.minlength">Mínimo 2 caracteres</span>
    </div>
</div>

<div class="row">

    <div class="form-group col-xs-12 col-sm-6">
        <label for="quantity">Quantidade em estoque</labe\
l>
        <input type="number" class="form-control" id="qua\
ntity" placeholder="Quantidade" v-model="product.quantity" \
min="0">
    </div>

```

```
<div class="form-group col-xs-12 col-sm-6">
  <label for="minQuantity">Quantidade Mínima</label>
  <input type="number" class="form-control" id="min\
Quantity" placeholder="Quantidade Mínima" v-model="product.\
minQuantity" min="1" >
</div>

</div>

<div class="row">

  <div class="form-group col-xs-12 col-sm-6">
    <label for="price">Preço</label>
    <input type="input" class="form-control" id="pric\
e" placeholder="Preço" v-model="product.price">
  </div>

  <div class="col-xs-12 col-sm-6">
    <div class="checkbox">
      <label>
        <input type="checkbox" v-model="product.active">
        Ativo?
      </label>
    </div>
  </div>

</div>

<div class="row">
  <div class="form-group col-xs-12">
    <label for="description">Descrição</label>
```

```

        <textarea class="form-control" rows="5" id="description" v-model="product.description"></textarea>
      </div>
    </div>

  </form>
</validator>
</div>
<div class="modal-footer">
  <Loading></Loading>
  <button @click.prevent="saveProduct" class="btn btn-default" :disabled="isLoading||$validateForm.invalid" >Salvar</button>

</div>
</div><!-- /.modal-content -->
</div><!-- /.modal-dialog -->
</div><!-- /.modal -->

</template>
<script>
  import {isLoading,itensPerPage} from '../vuex/getters.js'
  import {showLoading,hideLoading,setError} from '../vuex/actions.js'

  import Loading from '../controls/Loading.vue'
  import Pagination from '../controls/Pagination.vue'
  import Error from '../controls/Error.vue'
  import {URL} from '../config.js'

  //services
  import CategoryService from '../services/Category.js'
  import SupplierService from '../services/Supplier.js'

```

```
import ProductService from '../services/Product.js'

export default{
  components: {
    Loading, Error, Pagination
  },
  vuex:{
    getters:{
      isLoading, itensPerPage
    },
    actions:{
      showLoading, hideLoading, setError
    }
  },
  created(){
    this.loadProducts(this.keyword)

    CategoryService.getAll().then(
      result=>{
        this.$set('categories', result.json())
      },
      error=>{
        console.log(error)
      })

    SupplierService.getAll().then(
      result=>{
        this.$set('suppliers', result.json())
      },
      error=>{
        console.log(error)
      })
  }
}
```

```
    },
    ready(){
        //temp
        //$('#productModal').modal('show')
    },
    data(){
        return{
            keyword:"",
            products: [],
            categories: [],
            suppliers: [],
            errorMessage: "",
            product: {
                id:null,
                idCategory: null,
                idSupplier: null,
                code:null,
                name:null,
                quantity: 0,
                minQuantity: 1,
                price: null,
                description: null,
                active: null
            },
            total: 1,
            page: 1
        }
    },
    methods:{
        newProduct(){
            this.product = {}
            $('#productModal').modal('show')
```



```
    },
    saveProduct(){

        const onResponse = r => {
            this.$set('product', {})
            $('#productModal').modal('hide')
        }
        const onError = e => this.setError(e.body)
        const onFinally = () => {
            this.hideLoading()
            this.loadProducts()
        }

        this.showLoading()
        ProductService.save(this.product)
        .then(onResponse, onError)
        .finally(onFinally)

    },
    edit(product){
        this.product = product
        $('#productModal').modal('show')
    },
    onChangePage(page){
        this.page = page
        this.loadProducts()
    },
    search(){
        this.page = 1
        this.loadProducts()
    },
    deleteProduct(){
```

```

    const onResponse = r => this.$set('product', {})
    const onError = e => this.setError(e.body)
    const onFinally = () => {
      this.hideLoading()
      this.loadProducts()
    }

    if (confirm(`Deseja apagar "${this.product.name}"s ?` \
  )){
      this.showLoading()
      ConfigService.delete(this.product.id)
        .then(onResponse, onError)
        .finally(onFinally)
    }
  },
  loadProducts(){
    const onResponse = r => {
      this.$set('products', r.json())
      this.$set('total', r.headers['x-total-count'])
    }
    const onError = e => this.setError(e.body)
    const onFinally = () => {
      this.hideLoading()
    }

    this.showLoading()
    ProductService.getAll(this.page, this.itensPerPage, this.\
keyword)
      .then(onResponse, onError)
      .finally(onFinally)
  }
}

```

`</script>`

Este código é muito extenso, mas basicamente ele mostra uma tabela com os produtos cadastrados. Se o usuário clicar em um produto, usamos o modal do bootstrap (em conjunto com o jQuery) para exibir o formulário que o usuário poderá editá-lo.

Vamos comentar a seguir alguns detalhes importantes:

14.20.1 Propriedade data

O atributo data do componente contém:

```
data(){  
  return{  
    keyword: "",  
    products: [],  
    categories: [],  
    suppliers: [],  
    errorMessage: "",  
    product: {  
      id: null,  
      idCategory: null,  
      idSupplier: null,  
      code: null,  
      name: null,  
      quantity: 0,  
      minQuantity: 1,  
      price: null,  
      description: null,  
      active: null  
    },  
    total: 1,  
    page: 1  
  }  
}
```

```

    }
  }

```

Que é extenso, mas compreensível. As propriedades `categories` e `suppliers` serão preenchidas no carregamento do componente, e servem de suporte no formulário de edição do Produto, exibindo o select:

```

<select id="idSupplier" class="form-control" v-model="produ\
ct.idSupplier">
  <option v-for="supplier in suppliers" value="{{supplier.id\
}}">
    {{supplier.name}}
  </option>
</select>

```

A propriedade `product` representa UM produto e suas respectivas propriedades, como `idSupplier`, `description...` etc.

14.20.2 Método `created()`

No método `created()` temos:

```

created(){
  this.loadProducts(this.keyword)
  CategoryService.getAll().then(result=>{
    this.$set('categories',result.json())
  },
  error=>{
    console.log(error)
  })
  SupplierService.getAll().then(result=>{
    this.$set('suppliers',result.json())
  },

```

```
error=>{  
  console.log(error)  
})  
}
```

Ou seja, carregamos as categorias e os fornecedores utilizando os respectivos services que foram criados.

14.20.3 Método newProduct()

O método newProduct é executado quando o usuário clica no botão “Novo”, usando jQuery para abrir a janela modal com o formulário:

```
newProduct(){  
  this.product = {}  
  $('#productModal').modal('show')  
}
```

14.20.4 Método saveProduct()

O método saveProduct possui uma implementação bem comum no JavaScript. Criamos inicialmente métodos que serão chamados pelo retorno do Promise do service. Veja:

```
saveProduct(){  
  
  const onResponse = r => {  
    this.$set('product', {})  
    $('#productModal').modal('hide')  
  }  
  const onError = e => this.setError(e.body)  
  const onFinally = () => {  
    this.hideLoading()  
  }  
}
```

```
        this.loadProducts()  
    }  
    // continua....  
}
```

E depois usamos estes métodos na chamada do método save:

```
saveProduct(){  
  
    const onResponse = r => {  
        this.$set('product', {})  
        $('#productModal').modal('hide')  
    }  
    const onError = e => this.setError(e.body)  
    const onFinally = () => {  
        this.hideLoading()  
        this.loadProducts()  
    }  
  
    this.showLoading()  
  
    ProductService.save(this.product)  
    .then(onResponse, onError)  
    .finally(onFinally)  
  
}
```

Com estas observações você tem a compreensão necessária para compreender como a tela de cadastro de produtos funciona. Após você compreender como esta tela funciona, e deixá-la funcional no seu sistema, podemos partir para a tela de PDV, em uma versão bem básica (ainda), que será abordada na próxima atualização.