

2016

Vue.js

na prática

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

Daniel Schmitz

Vue.js na prática

Daniel Schmitz e Daniel Pedrinha Georgii

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

Essa versão foi publicada em 2016-06-20



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 a fantástica comunidade brasileira laravel-br que se encontra
no <https://laravel-br.slack.com>*

Conteúdo

Parte 1 – Conhecendo o Vue	4
1. Introdução	5
1.1 Tecnologias empregadas	5
1.2 Instalação do node	7
1.3 Uso do npm	7
1.4 Conhecendo um pouco o RESTfull	8
2. Conhecendo Vue.js	11
2.1 Uso do jsFiddle	11
2.2 Configurando o jsFiddle para o Vue	12
2.3 Hello World, vue	14
2.4 Two way databind	16
2.5 Criando uma lista	17
2.6 Detectando alterações no Array	19
2.6.1 Utilizando track-by	20
2.6.2 Uso do \$set	21
2.6.3 Uso do \$remove	21
2.6.4 Loops em objetos	21
2.7 Eventos e métodos	22
2.7.1 Modificando a propagação do evento	23
2.8 Design reativo	25
2.9 Criando uma lista de tarefas	25
2.10 Eventos do ciclo de vida do Vue	30
2.11 Compreendendo melhor o Data Bind	32
2.11.1 Databind único	32
2.11.2 Databind com html	32

CONTEÚDO

2.11.3	Databind em Atributos	32
2.11.4	Expressões	33
2.12	Filtros	33
2.12.1	uppercase	34
2.12.2	lowercase	34
2.12.3	currency	34
2.12.4	pluralize	34
2.12.5	json	34
2.12.6	debounce	34
2.12.7	limitBy	34
2.12.8	filterBy	35
2.12.9	orderBy	37
2.13	Filtros personalizados	38
2.13.1	Filtros “two way”	38
2.14	Diretivas	39
2.14.1	Argumentos	39
2.14.2	Modificadores	40
2.15	Atalhos de diretiva (Shorthands)	40
2.16	Alternando estilos	41
2.17	Uso da condicional v-if	42
2.18	Exibindo ou ocultando um bloco de código	43
2.19	v-if vs v-show	43
2.20	Formulários	43
2.20.1	Checkbox	44
2.20.2	Radio	45
2.20.3	Select	45
2.20.4	Atributos para input	45
2.21	Conclusão	46
3.	Criando componentes	47
3.1	Vue-cli	47
3.2	Criando o primeiro projeto com vue-cli	48
3.3	Executando o projeto	48
3.4	Conhecendo a estrutura do projeto	49
3.5	Conhecendo o packages.json	50
3.6	Componentes e arquivos .vue	51

CONTEÚDO

3.7	Criando um novo componente	53
3.8	Adicionando propriedades	57
3.8.1	camelCase vs. kebab-case	59
3.8.2	Validações e valor padrão	59
3.9	Slots e composição de componentes	61
3.9.1	Usando vários slots	62
3.10	Eventos e comunicação entre componentes	64
3.10.1	Repassando parâmetros	68
3.11	Reorganizando o projeto	68
3.12	Adicionando algum estilo	71
3.13	Alterando o cabeçalho	75
3.14	Alterando o rodapé	76
3.15	Conteúdo da aplicação	78
4.	Vue Router	79
4.1	Instalação	79
4.2	Configuração	79
4.3	Configurando o router.map	80
4.4	Configurando o router-view	81
4.5	Criando novos componentes	83
4.6	Criando um menu	85
4.6.1	Repassando parâmetros no link	86
4.7	Classe ativa	87
4.8	Filtrando rotas pelo login	89
5.	Vue Resource	91
5.1	Testando o acesso Ajax	92
5.2	Métodos e opções de envio	96
5.3	Trabalhando com resources	97

Parte 2 - Criando um blog com Vue,Express e MongoDB 100

6.	Express e MongoDB	101
6.1	Criando o servidor RESTful	105

CONTEÚDO

6.2	O banco de dados MongoDB	106
6.3	Criando o projeto	110
6.4	Estrutura do projeto	111
6.5	Configurando os modelos do MondoDB	111
6.6	Configurando o servidor Express	113
6.7	Testando o servidor	123
6.8	Testando a api sem o Vue	124
7.	Implementando o Blog com Vue	130
7.1	Reconfigurando o packages.json	130
7.2	Instalando pacotes do vue e materialize	130
7.3	Configurando o router e resource	131
7.4	Configurando a interface inicial da aplicação	134
7.5	Obtendo posts	138
7.6	Configurando o Vue Validator	143
7.7	Realizando o login	144
7.8	Token de autenticação	148
7.9	Criando um Post	151
7.10	Logout	154
7.11	Refatorando a home	155
7.12	Conclusão	158

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.

Suporte

Para suporte, você deve 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. Vue.js foi concebido para ser simples, reativo, baseado em componentes e compacto.

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. Outro exemplo será abordado com o o trio Apache, Php e MySQL, no qual iremos criar uma pequena rede social de demonstração.

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.

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.

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.

¹<https://nodejs.org/en/>

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.

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.

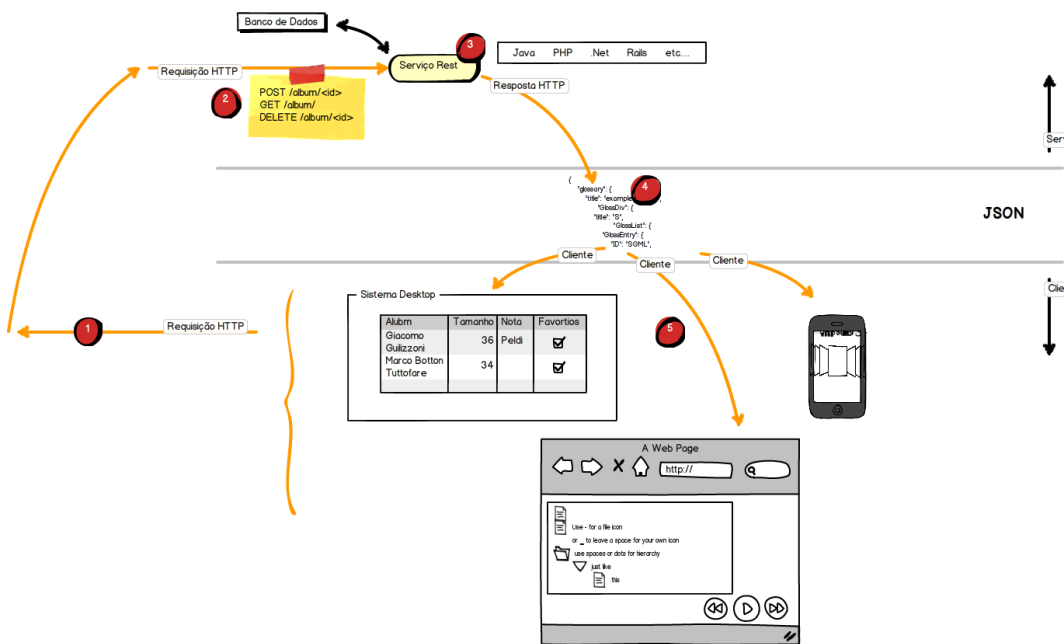
1.4 Conhecendo um pouco o RESTfull

Na evolução do desenvolvimento de sistemas web, os serviços chamados *webservices* estão sendo gradativamente substituídos por outro chamado *RESTful*, que é ‘quase’

²<https://www.npmjs.com/>

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 esta 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

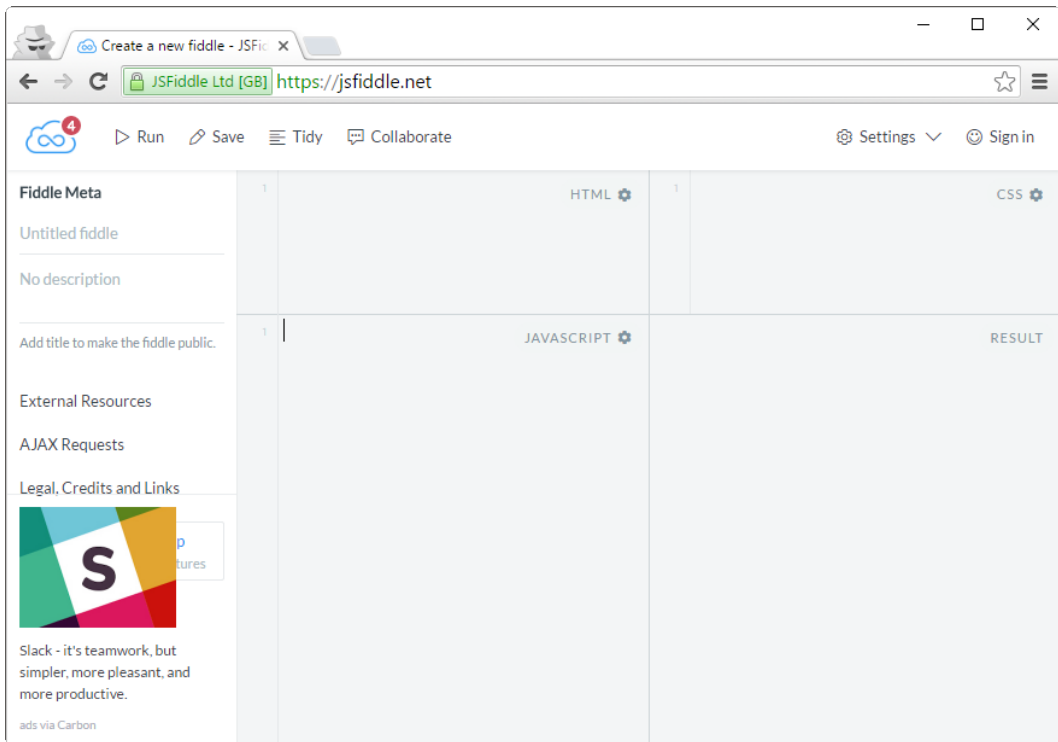
Neste capítulo iremos aprender alguns conceitos básicos sobre o Vue. Neste capítulo não veremos (ainda) a instalação do mesmo, porque como estamos apresentando cada conceito em separado, é melhor usarmos um editor online, neste caso o jsFiddle.

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:

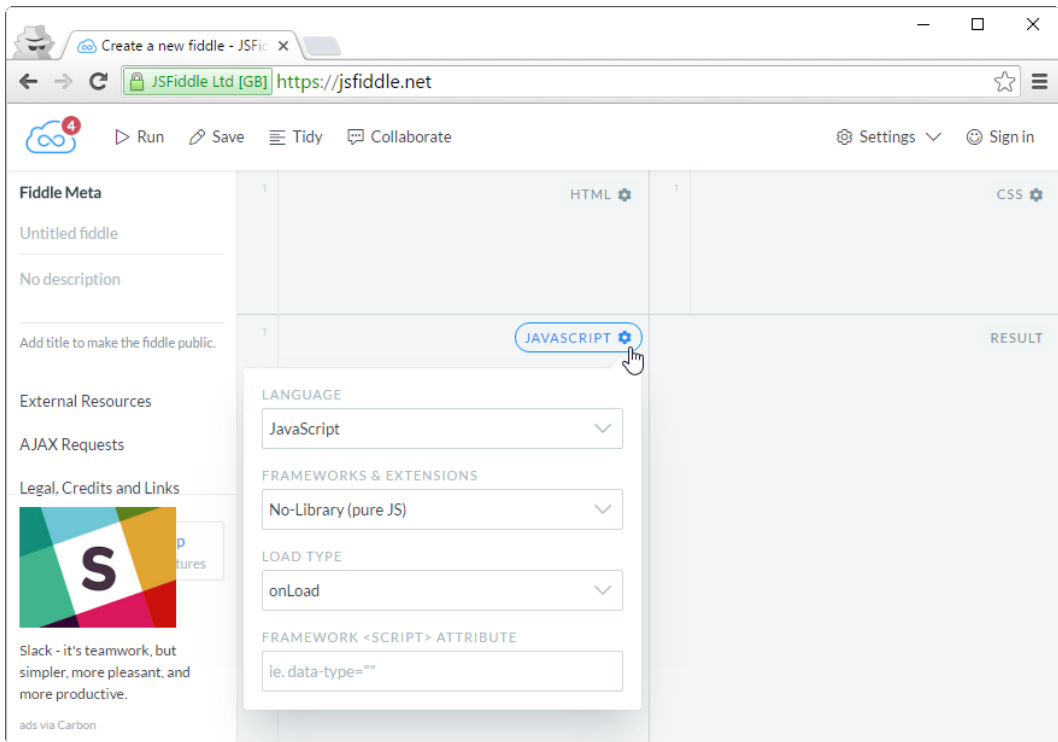
¹jsfiddle.net



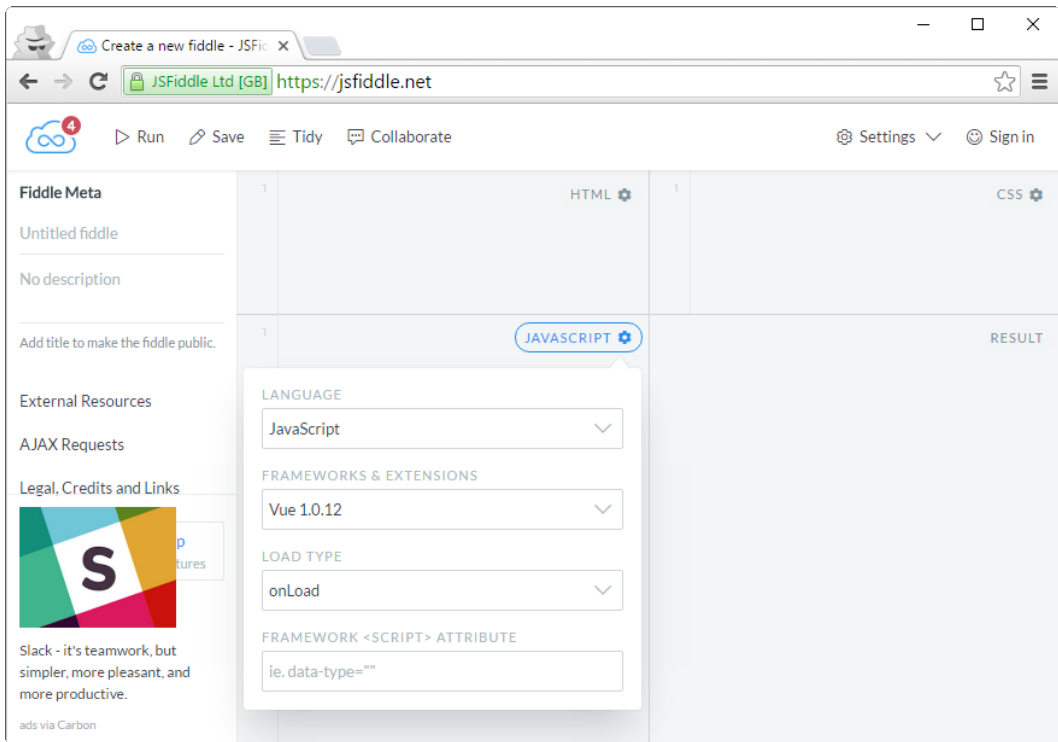
Caso queira, pode criar uma conta e salvar todos os códigos que criar, para poder consultar no futuro. No jsFiddle, temos 4 áreas sendo elas: *html*, *javascript*, *css* e *result*. Quando clicamos no botão Run, o html/javascript/css são combinados e o resultado é apresentado.

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 1.0.12, 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`.

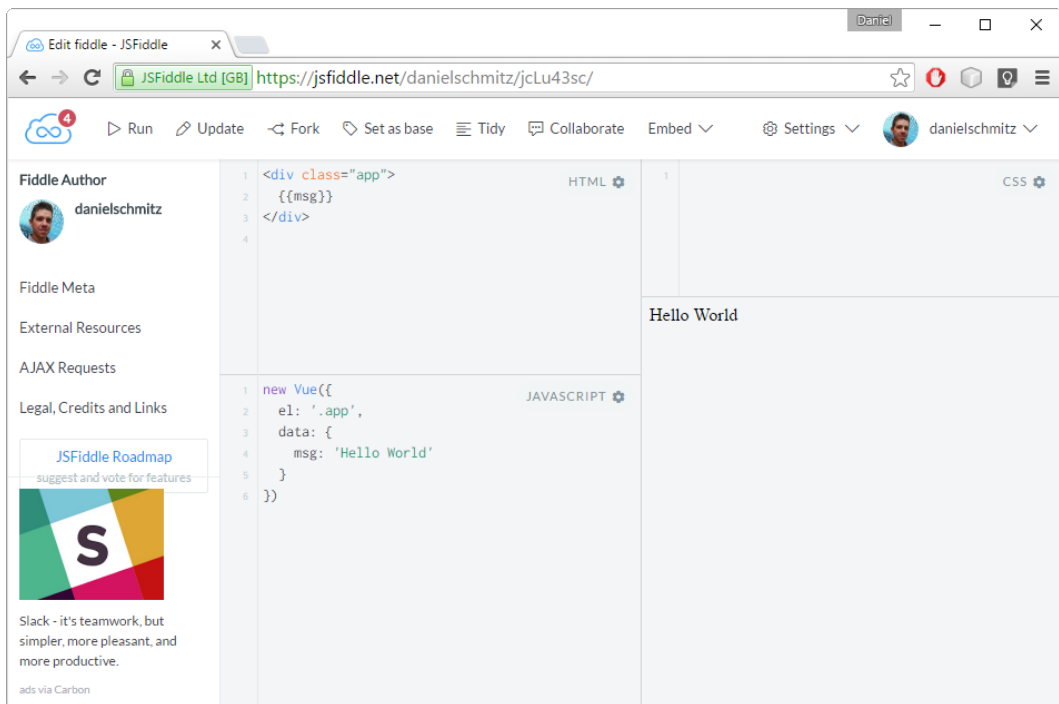
Agora vamos a parte Javascript. Para trabalhar com `vue`, basta criar um objeto `Vue` e repassar alguns parâmetros, veja:

```
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 a figura a seguir.

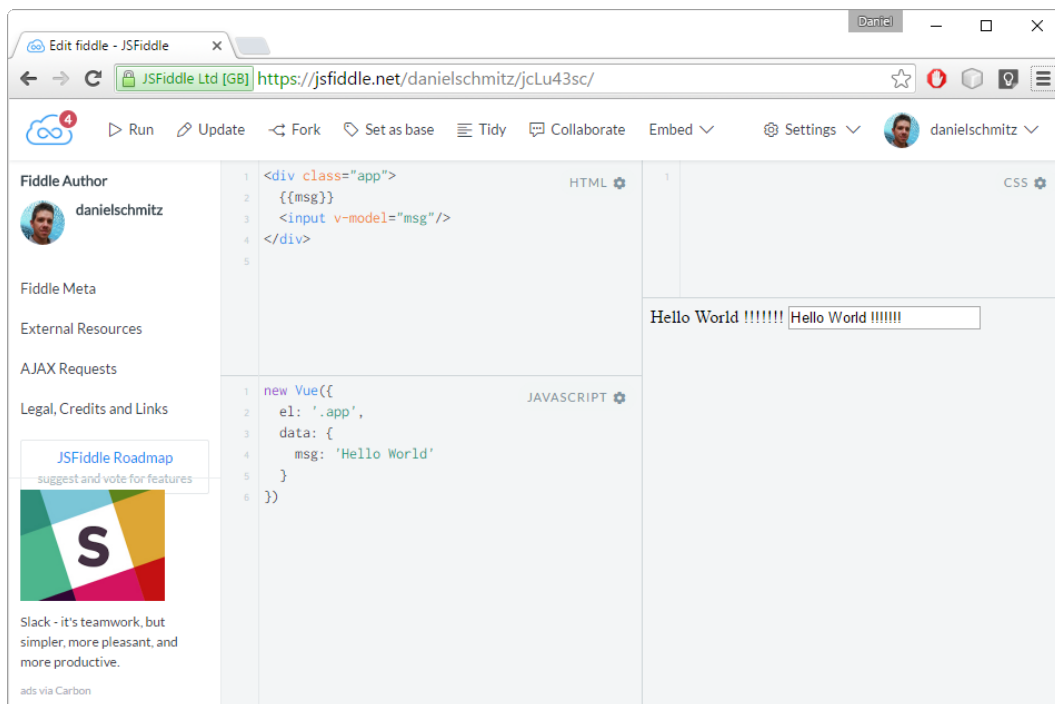


2.4 Two way databind

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 class="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

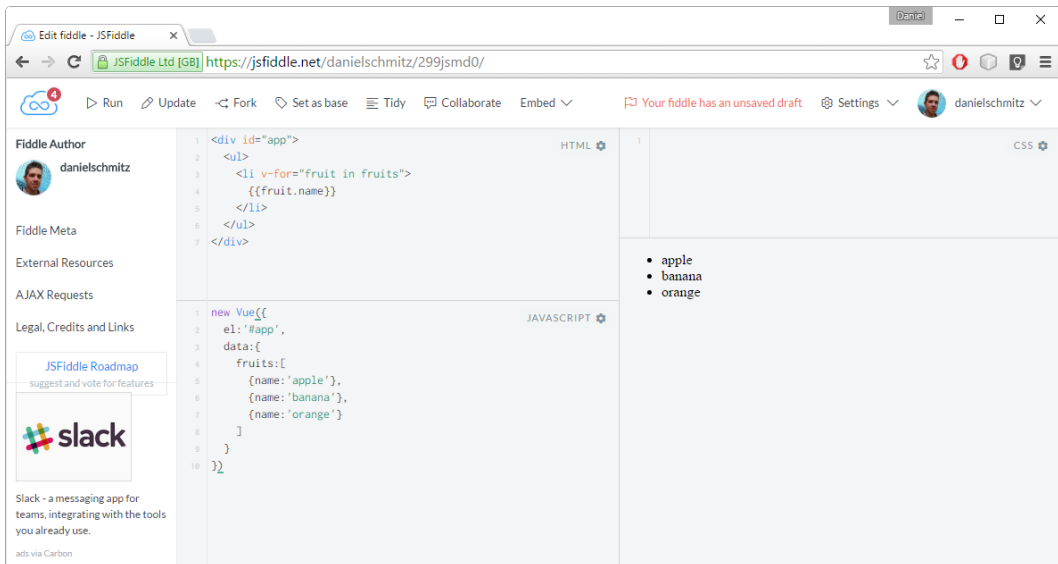
Existem dezenas de comandos do vue que iremos aprender ao longo desta obra. Um deles é o `v-for` que faz um loop no elemento em que foi inserido. Crie um novo jsFiddle com o seguinte código:

```
<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:



O laço `v-for` possui uma variável especial chamada `$index` que retorna o índice do item atual.

A partir da versão 1.0.17 do Vue, pode-se usar `item of items` ao invés de `item in items`, para se adequar semanticamente as iterações do javascript.

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 track-by

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 `track-id`, 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 reconconsulta 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" track-by="_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. Pode-se verificar esse comportamento [neste link](#)².

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. Clique [neste link](#)³ para verificar o resultado.

2.6.3 Uso do `$remove`

Além do `$set` também temos o método `$remove` que remove um item do array e atualiza o html.

2.6.4 Loops em objetos

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

²<https://jsfiddle.net/danielschmitz/fpzc8x2b/>

³<https://jsfiddle.net/danielschmitz/fpzc8x2b/3/>

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

2.7 Eventos e métodos

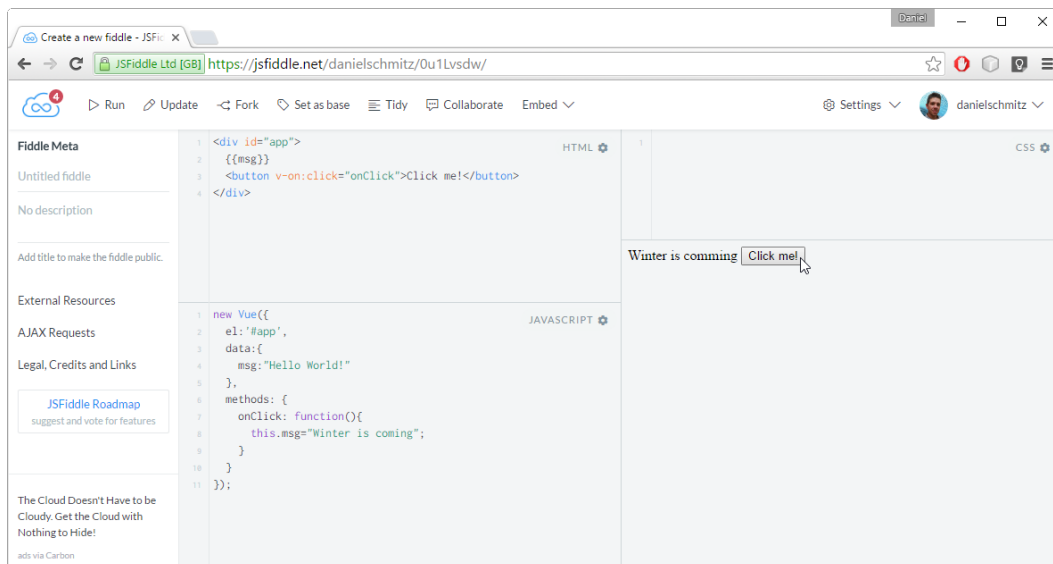
Podemos capturar diversos tipos de eventos e realizar operações com cada um deles. No exemplo a seguir, incluímos um botão que irá alterar uma propriedade do vue.

```
<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">
```

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" \

```

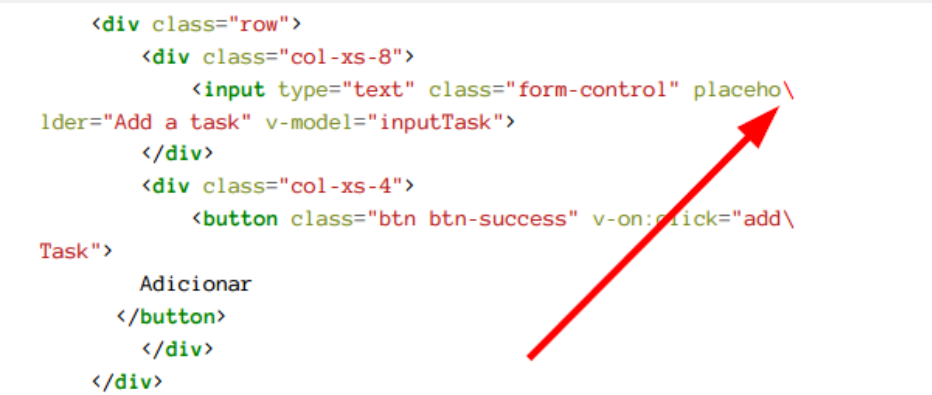
```

v-on:click="removeTask(task.id)">
                                x</button>
                                </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:



```

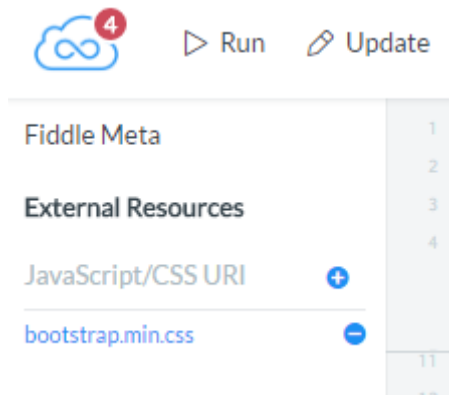
<div class="row">
  <div class="col-xs-8">
    <input type="text" class="form-control" placeholder="Add a task" v-model="inputTask">
  </div>
  <div class="col-xs-4">
    <button class="btn btn-success" v-on:click="addTask">
      Adicionar
    </button>
  </div>
</div>

```

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 algumas notações nas classes dos elementos html. 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);
        }
      }
    }
  }
})
```

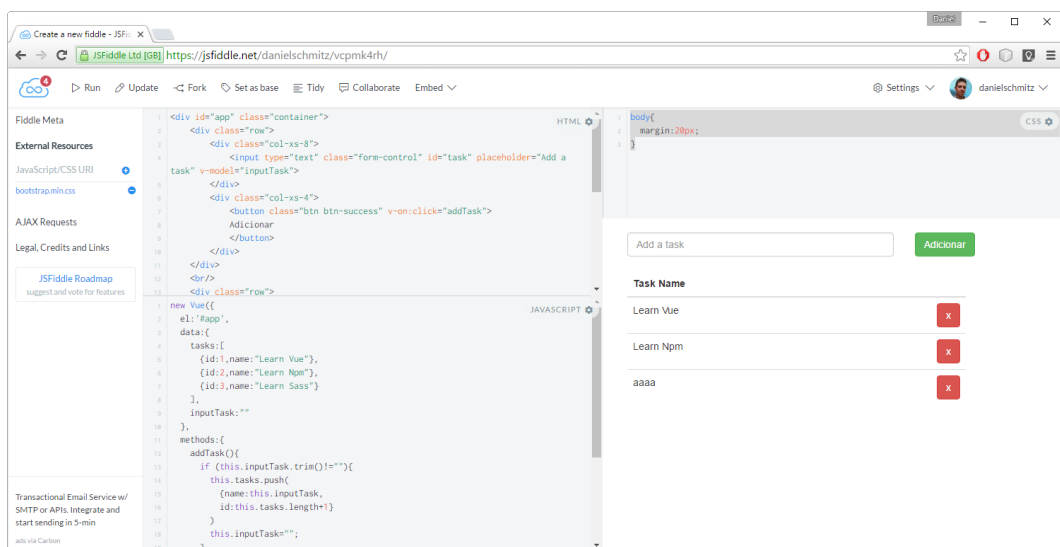
O código Vue, apesar de extenso, é fácil de entender. Primeiro criamos as duas variáveis: `tasks` possui um array de objetos que será a base da tabela que foi criada no html. Já a variável `inputTask` realiza um two way databind com a caixa de texto

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.

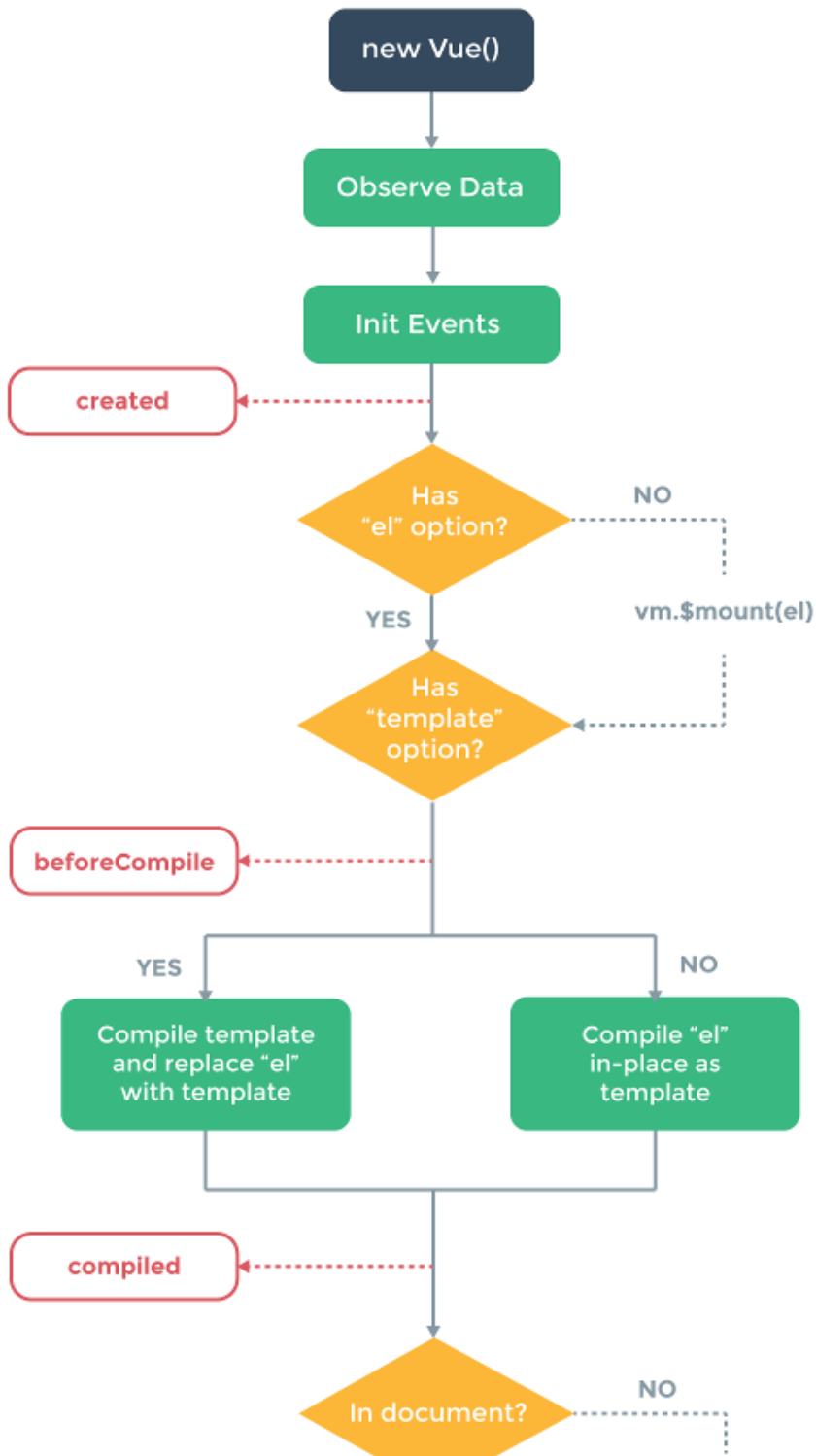
O resultado deste código pode ser visto na figura a seguir. Perceba que o formulário e a lista de tarefas possui o estilo do bootstrap.



Caso queira ter acesso direto a este código pelo jsFiddle, (clique aqui)[<https://jsfiddle.net/danielschmitz/vcpmk4rh/>]

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:



Em vermelho, temos os eventos que são disparados durante este ciclo, e que podem ser usados em determinados ocasiões no desenvolvimento de sistemas. O eventos que podem ser capturados são: `created`, `beforeCompile`, `compiled`, `ready`, `beforeDestroy`, `destroy`.

Para realizar requisições ajax, costuma-se utilizar o evento `ready`. 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 `"*"` antes da variável, conforme o código a seguir:

```
{{*msg}}
```

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 chaves, como por exemplo `{{{msg}}}`. Desta forma, o valor `Hello World` será exibido no navegador em negrito.

2.11.3 Databind em Atributos

Pode-se usar o `databind` dentro de atribos, como por exemplo: `<div id="item-{{ id }}"></div>`

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} }}`.

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>
```

O nome do filtro a ser aplicado deve estar após o caractere *pipe* |, sendo que pode-se concatenar diversos filtros em uma única expressão. *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.

2.12.6 debounce

Limitado para diretivas, adiciona um atraso em milissegundos no evento em questão, como por exemplo `<input @keyup="onKeyUp | debounce 500">`

2.12.7 limitBy

Limitado por diretivas, é usado para limitar os valores de um `v-for`, usado principalmente para prover paginações. O exemplo a seguir irá mostrar somente os 10 primeiros itens:

```
<div v-for="item in items | limitBy 10"></div>
```

E este exemplo exibe os itens de 5 a 15:

```
<div v-for="item in items | limitBy 10 5"></div>
```

2.12.8 filterBy

Limitado a diretivas, realiza um filtro em loops, principalmente no v-for. Vamos a alguns exemplos para exemplificar melhor este processo.

```
<div v-for="item in items | filterBy 'hello'">
```

Neste exemplo, somente valores que contém hello serão exibidos. Geralmente, o array items é um array de objetos (e não de strings), então para que possamos fazer um filtro em uma propriedade de um item do objeto, podemos fazer da seguinte forma:

```
<div v-for="user in users | filterBy 'Jack' in 'name'">
```

No caso acima, users é um array de objetos em que a propriedade name é filtrada pela string Jack.

No próximo exemplo, criamos uma lista de frutas e usamos o recurso de filtro para: 1) capitalizar os itens e 2) filtrar de acordo com o que for digitado em uma caixa de texto, veja:

```
<div id="app">
  <input v-model="filterValue"/>
  <ul>
    <li v-for="fruit in fruits | filterBy filterValue in 'name'">
      {{fruit.name | capitalize}}
    </li>
  </ul>
</div>
```

e

```
new Vue({
  el: '#app',
  data: {
    fruits: [
      {name: 'apple'},
      {name: 'banana'},
      {name: 'orange'},
      {name: 'lemon'},
      {name: 'grape'},
      {name: 'raspberry'},
      {name: 'coconut'},
      {name: 'pear'}
    ],
    filterValue: ''
  }
})
```

Este exemplo pode ser acessado pelo jsFiddle clicando [aqui](https://jsfiddle.net/danielschmitz/dy765x76/)⁴. nele, temos a criação da lista usando o filtro `filterBy filterValue in 'name'` onde `filterName` é a caixa de texto ligada pelo `v-model`.

⁴<https://jsfiddle.net/danielschmitz/dy765x76/>

Se houver necessidade de filtrar por mais campos, pode-se utilizar a seguinte expressão:

```
<li v-for="user in users | filterBy searchText in 'name' '\
phone'"></li>
```

2.12.9 orderBy

Limitada a diretivas, principalmente a v-for, é usada para ordenar os itens de um array. Os exemplos a seguir ilustram o processo:

```
<ul>
  <li v-for="user in users | orderBy 'name'">
    {{ user.name }}
  </li>
</ul>
```

Para ordenar de forma decrescente, use:

```
<ul>
  <li v-for="user in users | orderBy 'name' -1">
    {{ user.name }}
  </li>
</ul>
```

Pode-se ordenar com dois ou mais valores:

```
<ul>
  <li v-for="user in users | orderBy 'lastName'
    'firstName'">
    {{ user.lastName }}, {{ user.firstName }}
  </li>
</ul>
```

2.13 Filtros personalizados

É possível criar filtros personalizados através do comando `Vue.filter`. O exemplo a seguir cria um filtro que irá inverter os caracteres de posição.

```
Vue.filter('reverse', function (value) {
  return value.split('').reverse().join('')
})
```

Após a criação do filtro, pode-se usá-lo da seguinte forma:

```
<span v-text="message | reverse"></span>
```

2.13.1 Filtros “two way”

Pode-se criar filtros personalizados que são aplicados quando se está exibindo valores na tela e quando se está atualizando o `v-model`. Um exemplo seria a formatação de moeda, que quando exibida na view pode apresentar o símbolo da moeda corrente, e quando atualizado no model deve ser um valor float. O exemplo a seguir ilustra este filtro:

```
Vue.filter('currencyDisplay', {
  read: function(val) {
    return '$'+val.toFixed(2)
  },
  write: function(val, oldVal) {
    var number = +val.replace(/^[^\d.]/g, '')
    return isNaN(number) ? 0 : parseFloat(number.toFixed(2))
  }
})
```

2.14 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)⁵, sempre), vamos apresentar as diretivas mais usadas ao longo de toda esta obra.

2.14.1 Argumentos

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

⁵<http://vuejs.org/api>

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

é o mesmo que:

```
<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`.

2.14.2 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.

2.15 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>
```

2.16 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()!=""){  
        //.....  
    }  
},
```

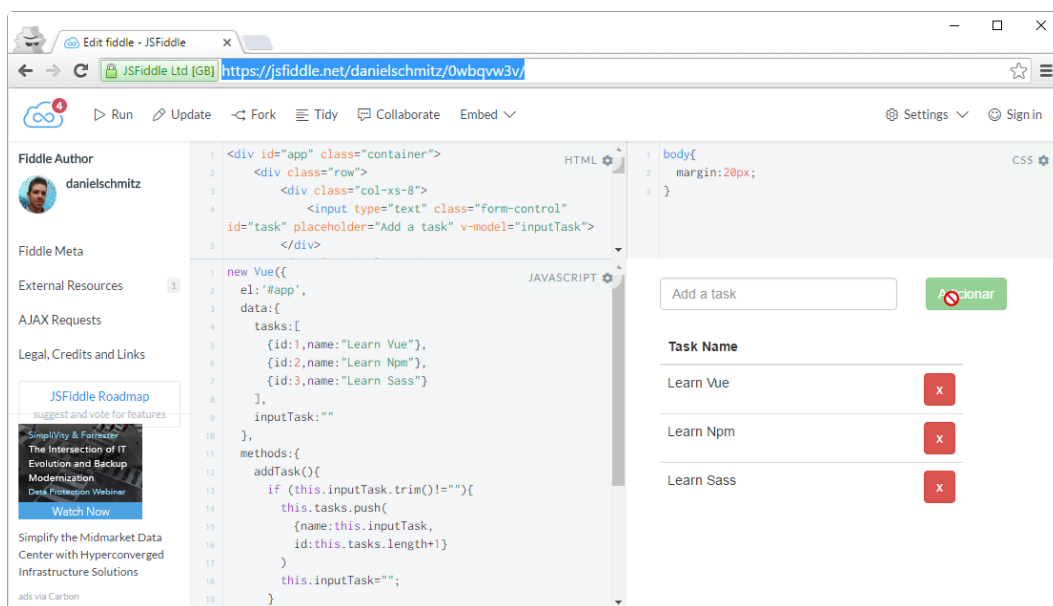
Com design reativo podemos alterar, por exemplo, o botão que inclui a tarefa para:

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

Teste esta variação neste [link](https://jsfiddle.net/danielschmitz/0wbqvw3v/)⁶

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.

⁶<https://jsfiddle.net/danielschmitz/0wbqvw3v/>



2.17 Uso da condicional v-if

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

```
<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`.

2.18 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>
```

2.19 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.

2.20 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">
```

2.20.1 Checkbox

Um input do tipo checkbox deve estar ligado a uma propriedade do vmodel 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:



2.20.2 Radio

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

2.20.3 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>
```

2.20.4 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.

number

Formata o campo input para aceitar somente números.

debounce

O atributo debounce adiciona um atraso (delay) depois que o usuário digita algo, antes do valor ser sincronizado com o modelo. Essa técnica é útil quando estamos criando um campo de busca e não há a necessidade de prover esta busca de imediato. Ela será executada somente após o debounce ocorrer. O debounce é calculado em milissegundos, então `debounce="1000"` corresponde a 1 segundo.

2.21 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 **Browsersify**, 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 `<app></app>` 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. Uma nova propriedade, chamada `components`, é usada para dizer ao `Vue` para iniciar o componente chamado `App`.

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 foco 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 `packages.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 -c 1 -a localhost",  
  "dev": "npm-run-all --parallel watchify serve",  
  "build": "cross-env NODE_ENV=production browserify src/  
main.js | uglifyjs -c warnings=false -m > dist/build.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`. O 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:

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

Algumas pequenas mudanças serão notadas nos templates `.vue`, mas nada que não possa ser compreendido.

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 Menu.vue e adicione o seguinte código:

src/Menu.vue

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

  }
</script>
<style>

</style>
```

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

src/App.vue

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

Perceba que o componente Menu.vue tem a tag <menu>. Esta é uma convenção do Vue. 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">
    <menu></menu>
    <h1>{{ msg }}</h1>
  </div>
</template>

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

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

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

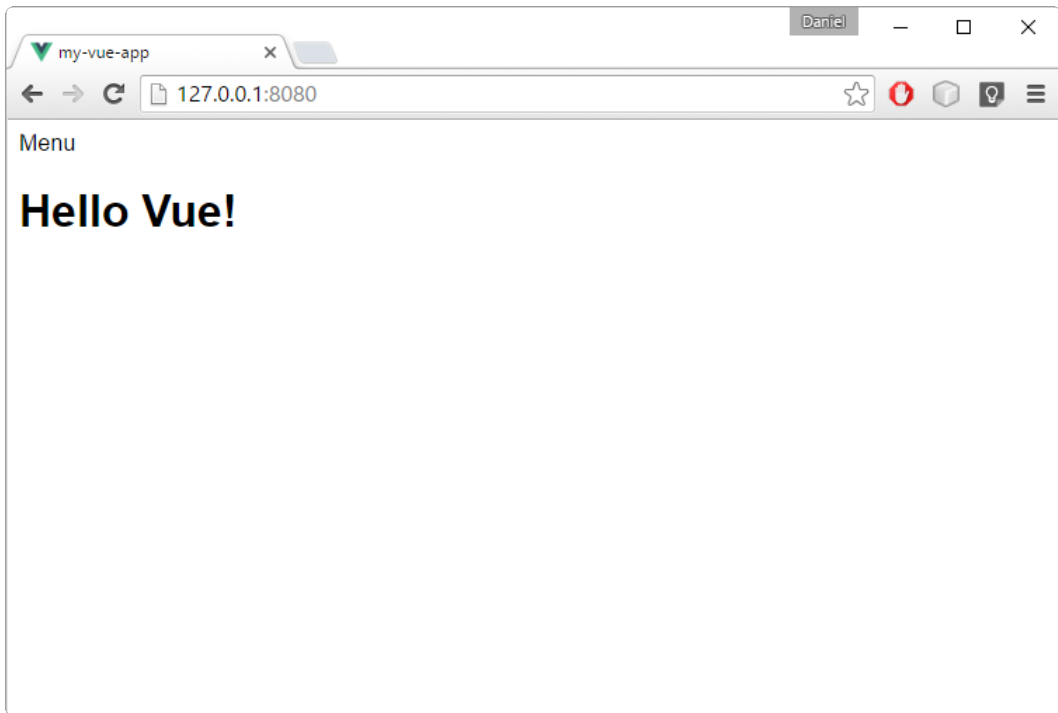
Ao incluirmos `import Menu from './Menu.vue'` estaremos referenciando o componente `Menu` e permitindo que o `Brwoserify` adicione-o no arquivo `build.js`. Além desta alteração, é necessário também configurar o componente `App` dizendo que ele usa o componente `Menu`. No `Vue`, sempre que usarmos um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

`src/App.vue`

```
<template>  
  <div id="app">  
    <menu></menu>  
    <h1>{{ msg }}</h1>  
  </div>  
</template>  
  
<script>  
import Menu from './Menu.vue'  
  
export default {  
  components:{  
    Menu  
  },  
  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 `<menu>` pode ser observado na página, que a princípio é semelhante a figura a seguir.



3.8 Adicionando propriedades

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

src/Menu.vue

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



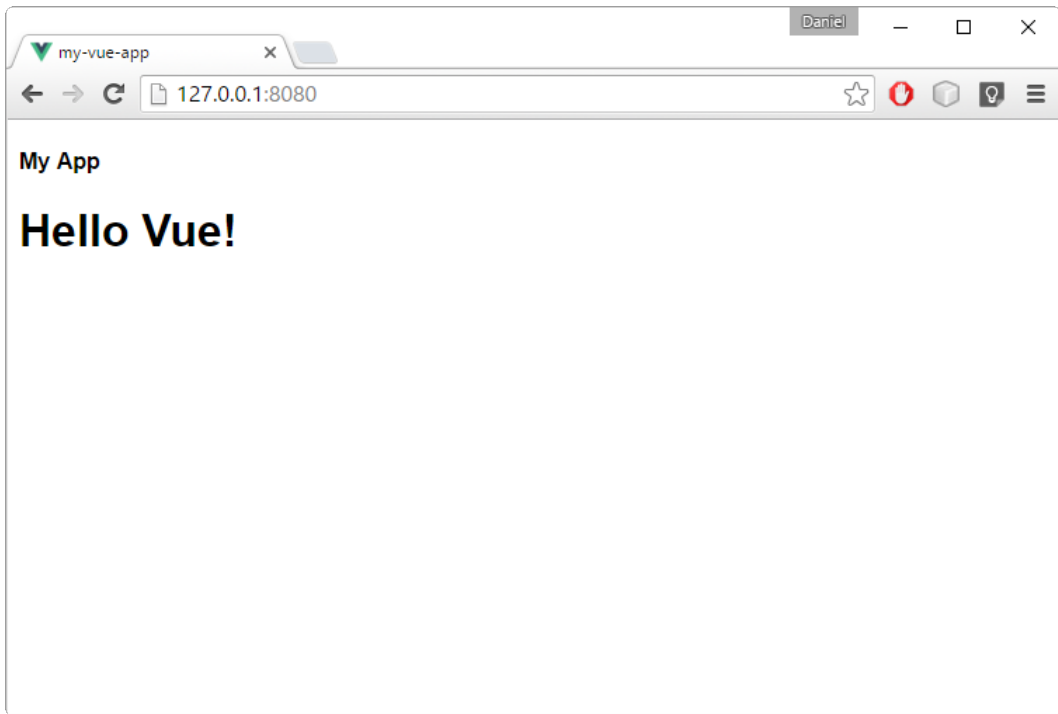
```
    }  
</script>
```

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

`src/App.vue`

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

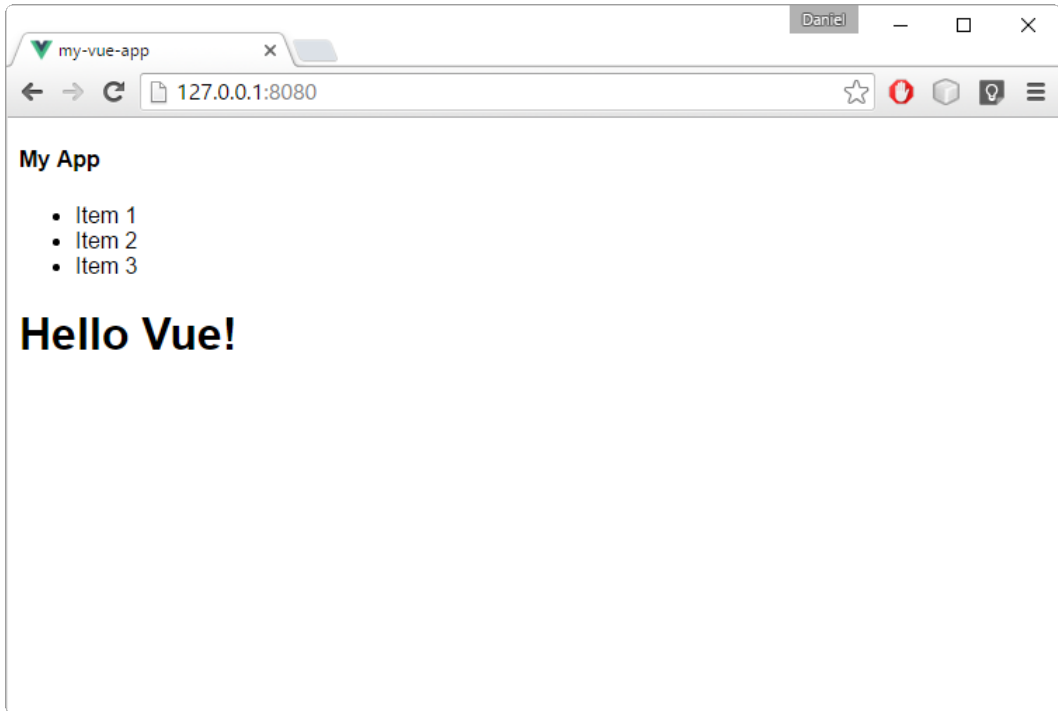
```
<menu title="My App">
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</menu>
```

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

src/Menu.vue

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

Perceba que, o conteúdo interno ao `<menu>` é adicionado na tag `<slot>` na definição 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:



3.9.1 Usando vários slots

Pode-se criar vários slots no componente para adicionar conteúdo em determinadas partes do template. Para isso, usa-se a propriedade `name` no slot e a propriedade `slot` no componente. No exemplo a seguir, adicionamos mais um slot ao componente Menu:

src/Menu.vue

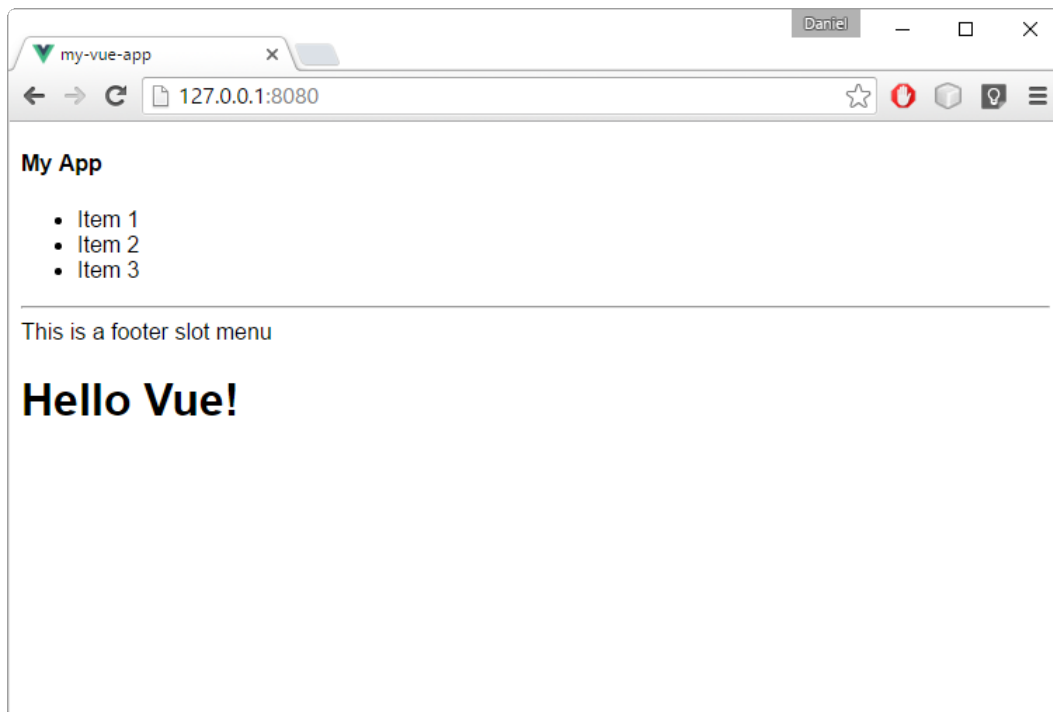
```
<template>
  <h4>{{title}}</h4>
  <div>
    <slot></slot>
  </div>
  <hr/>
  <slot name="footer"></slot>
</template>
<script>
  export default{
    props: ['title']
  }
</script>
```

Após configurar o componente Menu, vamos usar o novo slot no componente App.

src/App.vue

```
<template>
  <div id="app">
    <menu title="My App">
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
      </ul>
      <div slot="footer">This is a footer slot menu</div>
    </menu>
    <h1>{{ msg }}</h1>
  </div>
</template>
```

Perceba que adicionamos uma div dentro do componente `<menu>` e referenciamos essa div com o nome do slot, neste caso `<div slot="footer">`. O resultado 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 `$emit()`
- Disparar um evento que irá se propagar “para cima” através do método `$dispatch()`
- Disparar um evento que irá se propagar “para baixo” através do método `$broadcast`

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 item de menu 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">
    <menu title="My App" v-on:button-click="onButtonClick">
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
      </ul>
      <div slot="footer">This is a footer slot menu</div>
    </menu>
    <h1>{{ msg }}</h1>
  </div>
</template>
```

```
<script>
import Menu from './Menu.vue'
```

```
export default {
  components:{
```



```
      Menu
    },
    data () {
      return {
        msg: 'Hello Vue!'
      }
    },
    methods: {
      onButtonClick: function(e) {
        alert("button clicado");
      }
    }
  }
</script>

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

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

```
v-on:button-click="onButtonClick"
```

Que irá chamar o método onButtonClick, definido pelo método:

```
methods:{
  onButtonClick:function(){
    alert("button  clicado");
  }
}
```

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

src/Menu.vue

```
<template>
  <h4>{{title}}</h4>
  <button @click="onButtonClick">click me</button>
  <div>
    <slot></slot>
  </div>
  <hr/>
  <slot name="footer"></slot>
</template>
<script>
  export default{
    props: ['title'],
    methods:{
      onButtonClick : function(){
        this.$dispatch('button-click');
      }
    }
  }
</script>
```

Criamos um botão que chama o método `onButtonClick`, que por sua vez usa o `$dispatch` 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.$dispatch('button-click', "dispatch event f\rom 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>
  Header
</template>
<script>
  export default{

  }
</script>
```

src/layout/AppContent.vue

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

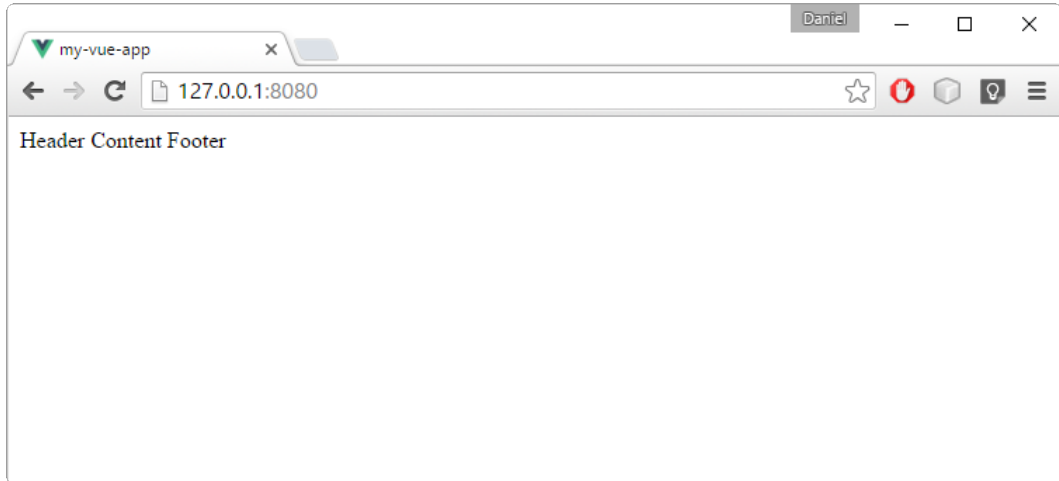
  }
</script>
```

src/layout/AppFooter.vue

```
<template>
  Footer
</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. Os 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 ao 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 página 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`

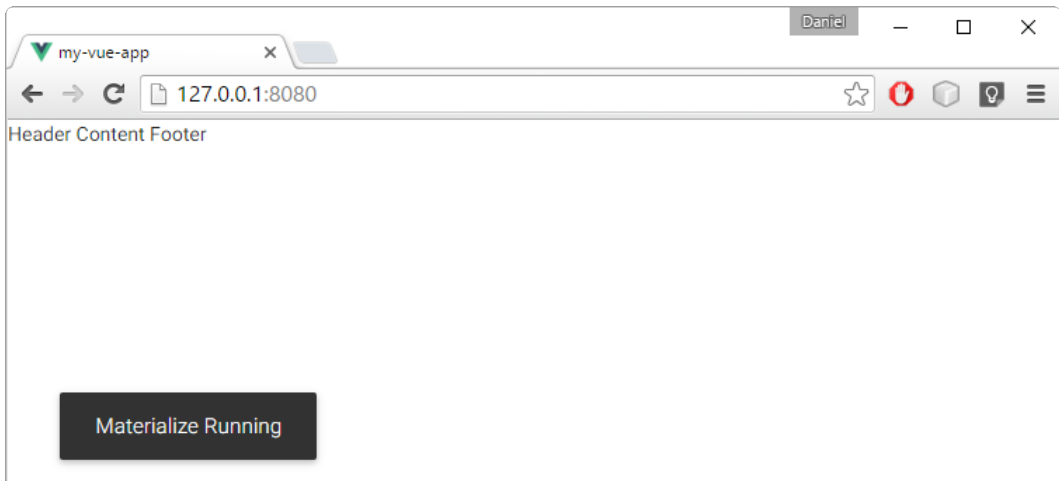
```
<template>
  <div id="app">
    <app-header></app-header>
    <app-content></app-content>
    <app-footer></app-footer>
  </div>
</template>
```



```
<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>
```

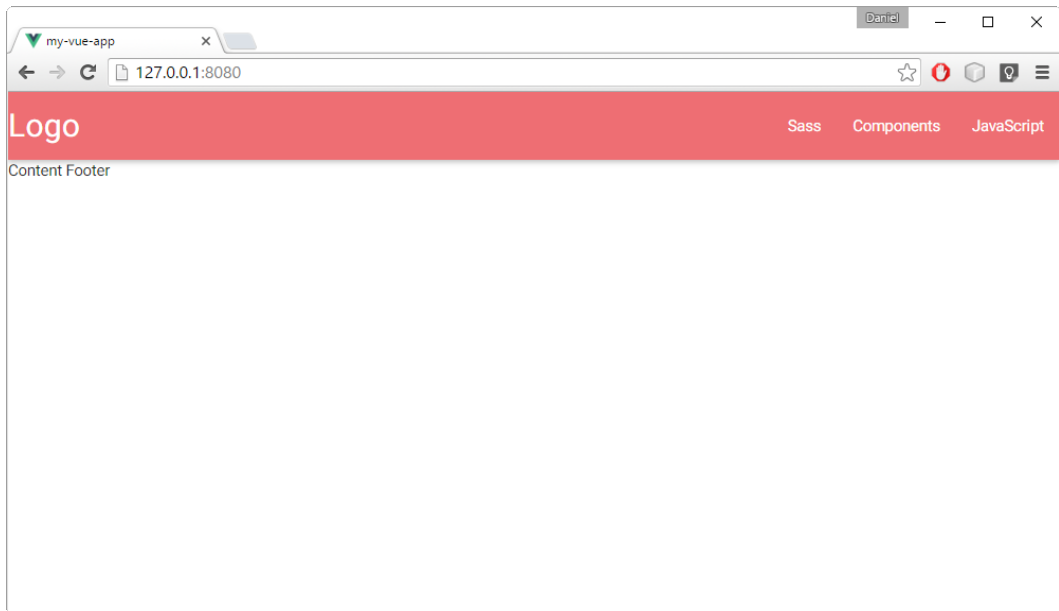
E adicionar no template do AppHeader:

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

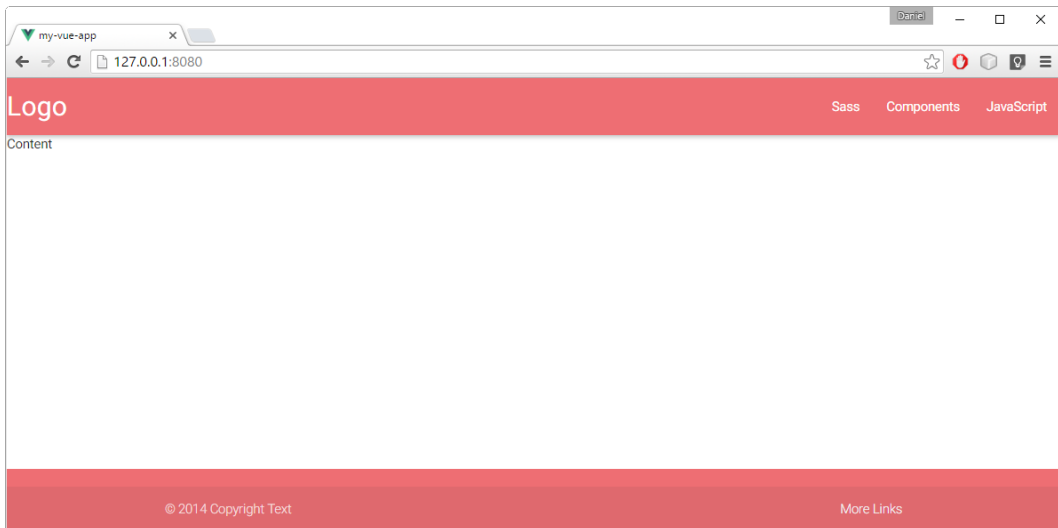
O mesmo pode ser aplicado ao rodapé da pagina, copiando um estilo direto do materialize e aplicando no AppFooter:

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:

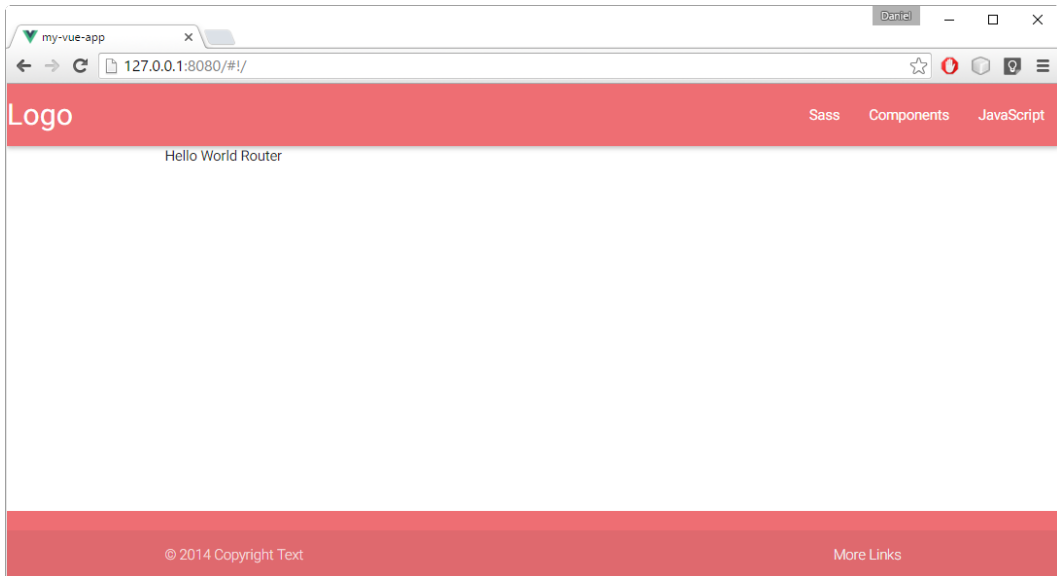
src/layout/AppContent.vue

```
<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 to 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:

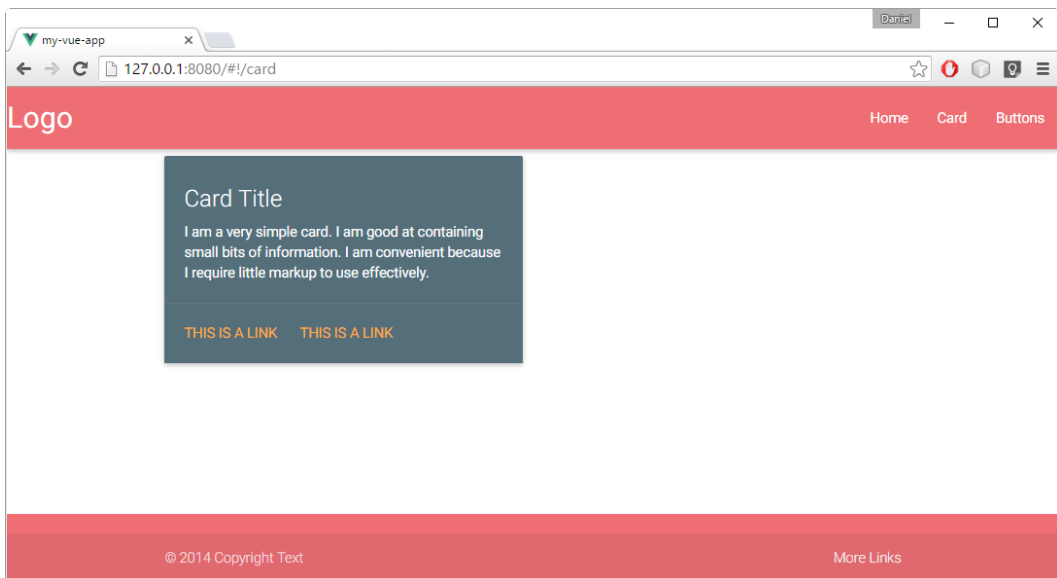
src/layout/AppHeader.vue

```
<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>
```

```
    </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>
```

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`

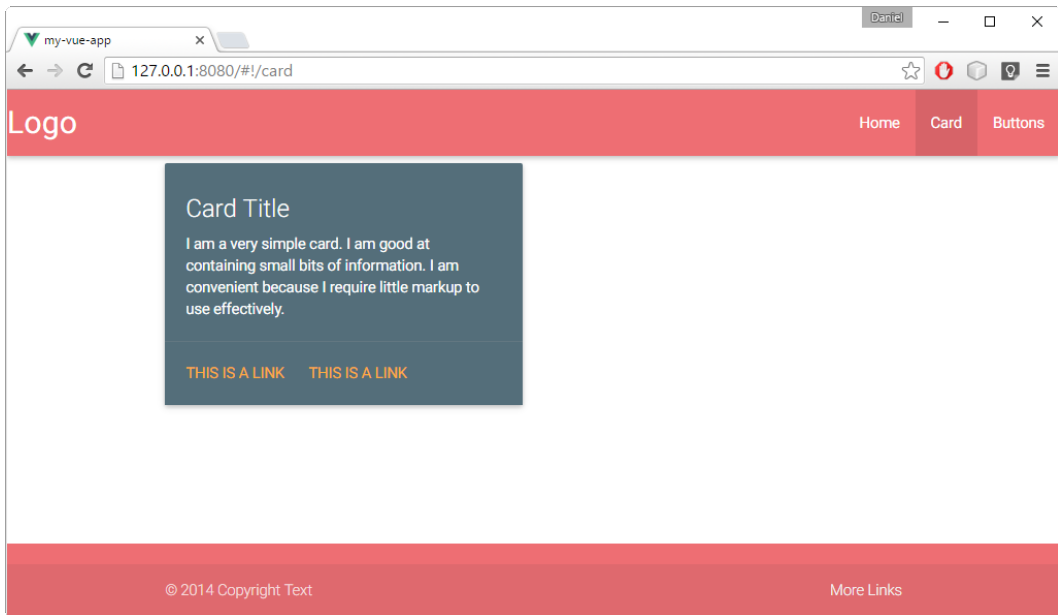
```
...  
const router = new VueRouter({  
  linkActiveClass: 'active'  
})  
...
```

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' }">Card<\
/a></li>
      <li v-link-active><a v-link="{ path: '/buttons' }">Bu\
ttons</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 rotamento 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) {  
  //SIMULAÇÃO:  
  let authenticated = false;  
  console.log(transition);  
  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:

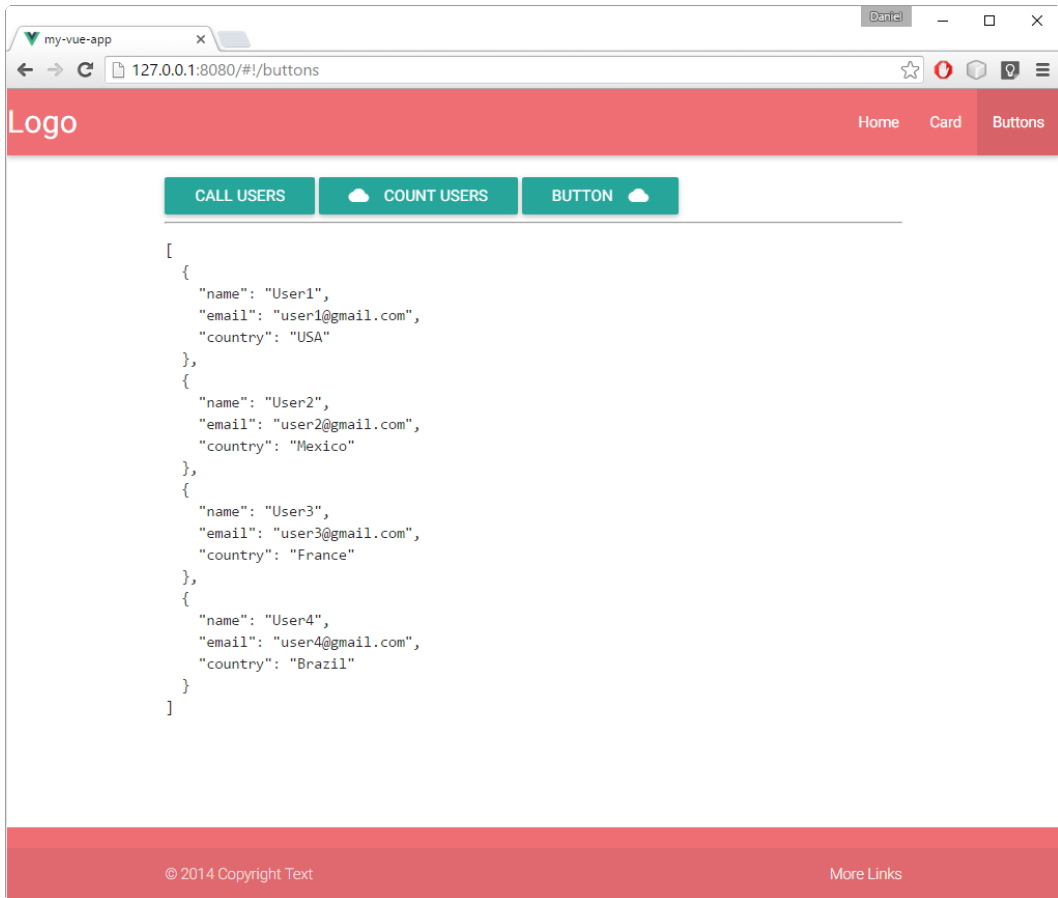
```
this.$http({url: '/users.json', method: 'GET'})
    .then(function (response) {
        this.users = response.data
    }, function (response) {
        Materialize.toast('Error: ' + response.stat\
usText, 3000)
    });
```

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
url	string	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 [XHR](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)¹ object| `|upload|` | Object | Um objeto que contém parâmetros [XHR.upload](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/upload)² | `|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}')
    }
  },
}
```

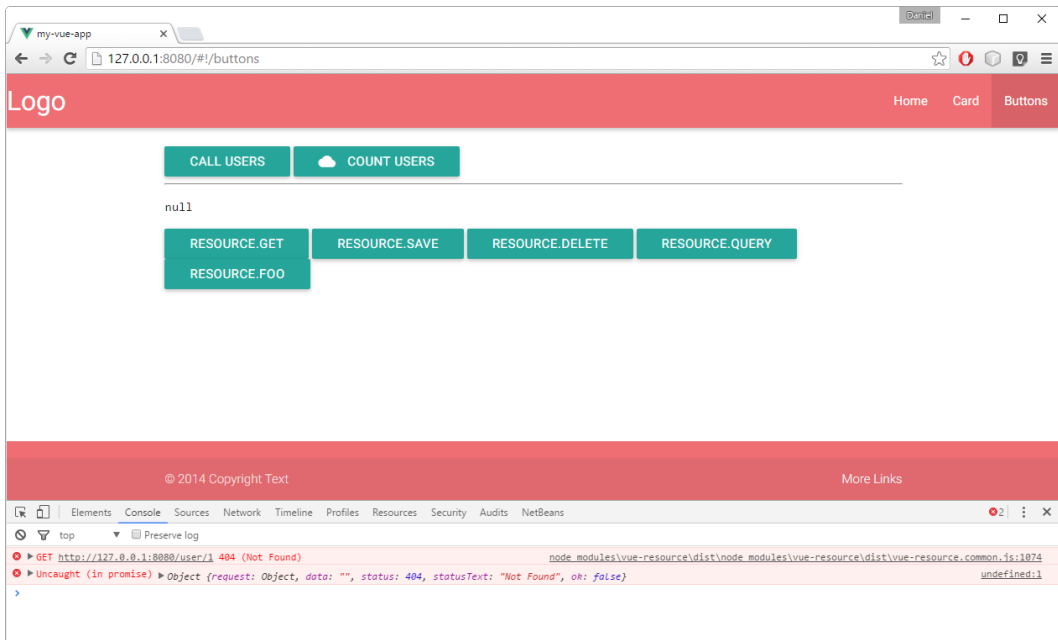
Então, criamos um botão que irá chamar cada tipo de resource. O primeiro deles é o `resource.get`, veja:

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

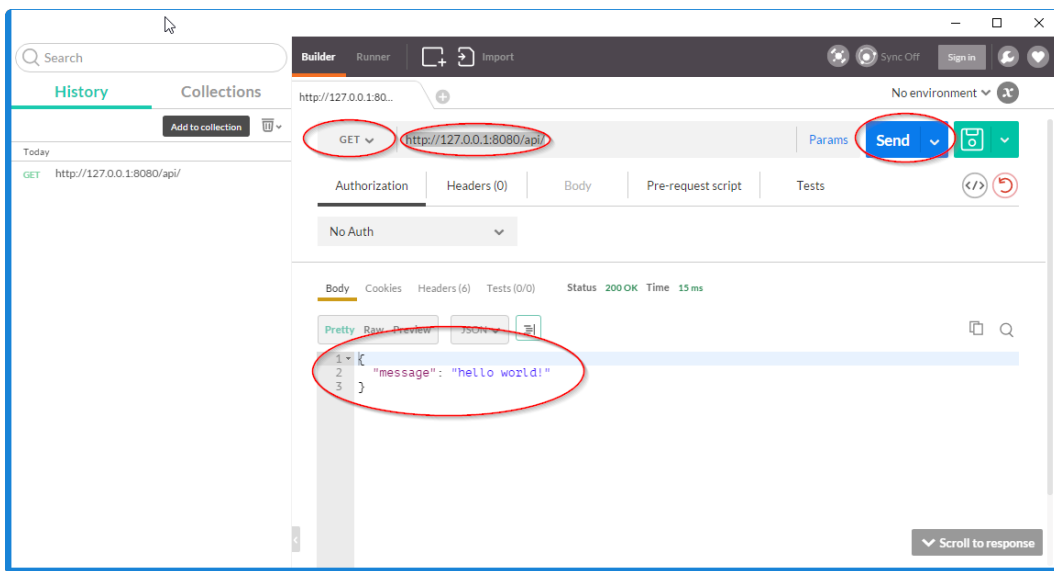
```
this.resourceUser.foo({id:1}).then(function (response) {  
    console.log(response)  
});
```

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

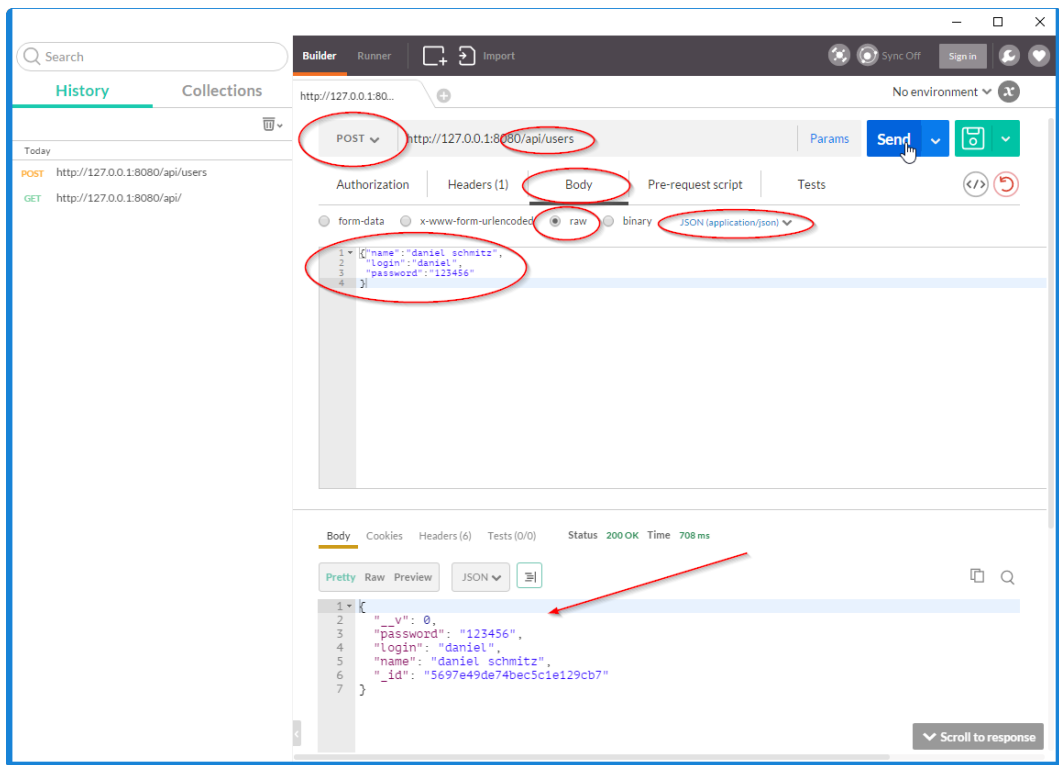
6. Express e MongoDB

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.

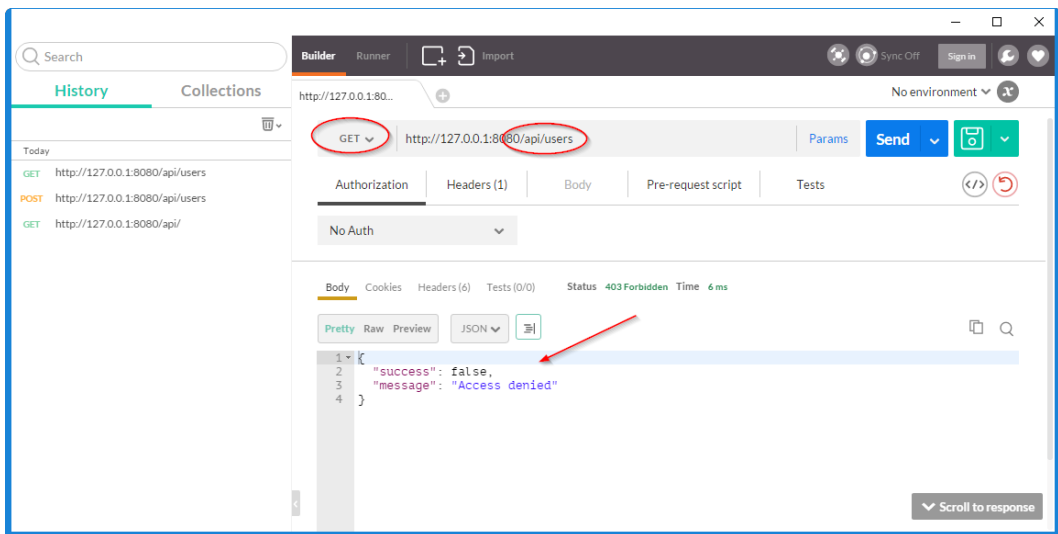
Por exemplo, para testar o endereço `http://127.0.0.1:8080/api/`, configuramos o Postman da seguinte forma:



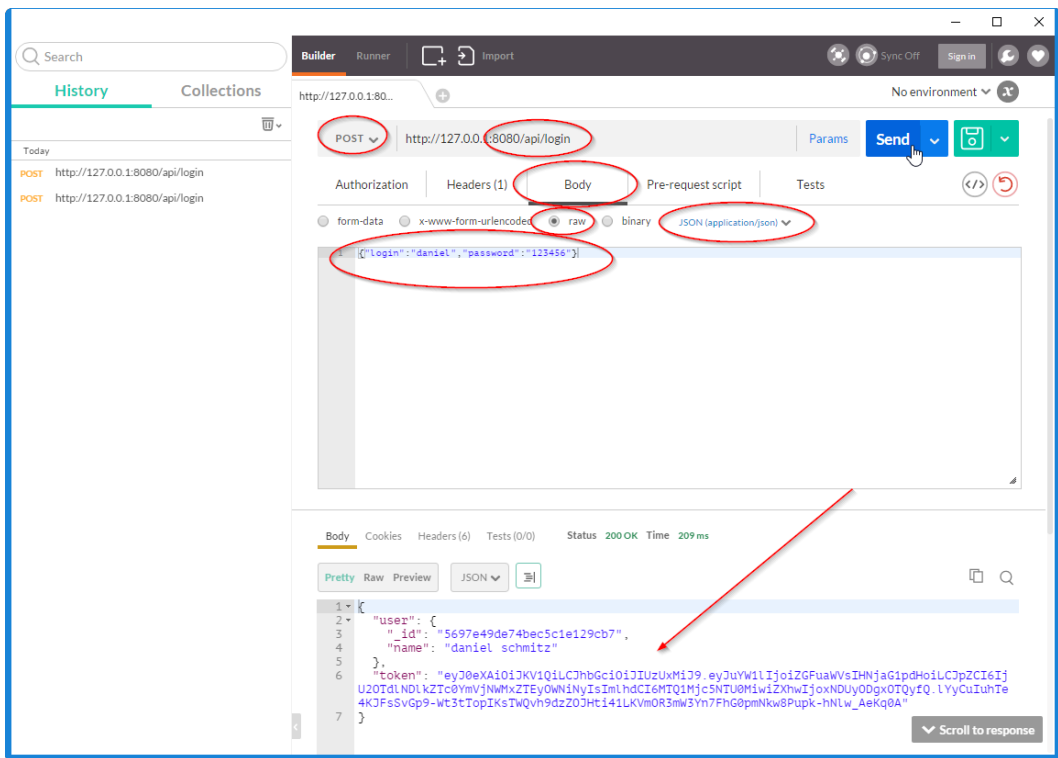
Perceba que obtemos a resposta “hello world”, conforme configurado no servidor. Para criar um usuário, podemos realizar um POST à url `/users` repassando os seguintes dados:



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:

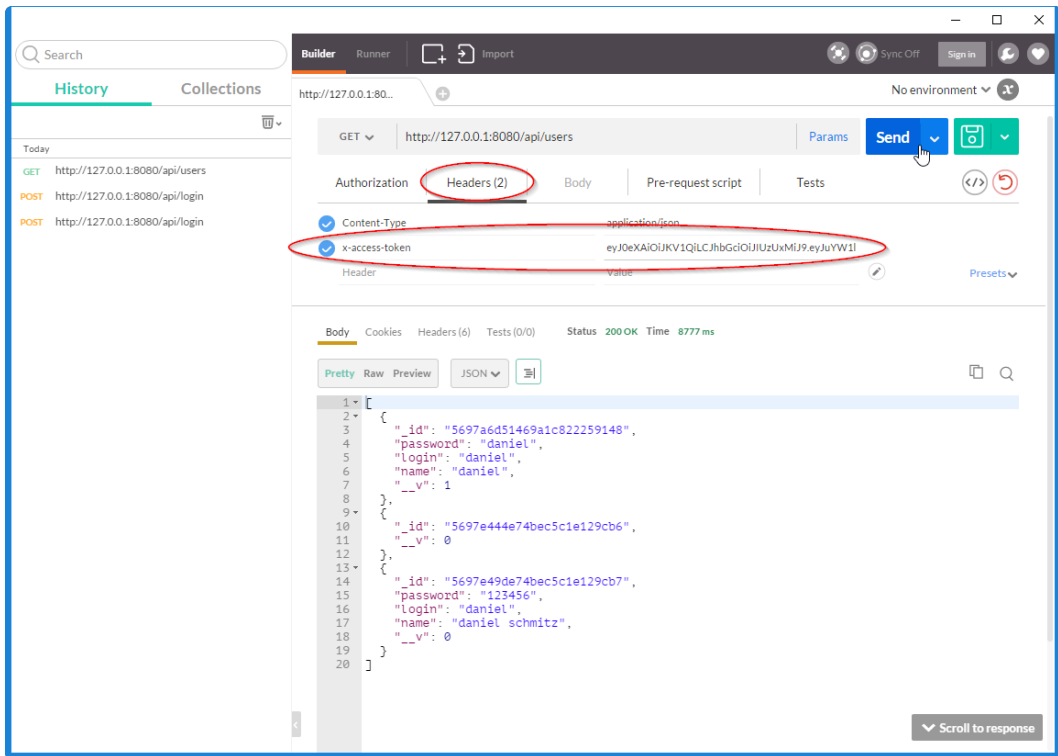


Para realizar o login, acessamos a URL `/api/login`, da seguinte forma:



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:



Veja que com o token os dados sobre os usuários são retornados. Experimente alterar algum caractere do token e refazer a chamada, para obter o erro “Failed to authenticate”, onde é necessário realizar um login para que o usuário possa criar um Post.

6.1 Criando o servidor RESTful

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

- **express:** É o servidor web que ficará responsável em receber as requisições web vindas do navegador e respondê-las corretamente. Não usaremos o *live-server*, mas o **express** tem a funcionalidade de autoloading através da biblioteca **nodemon**.
- **body-parser:** É uma biblioteca que obtém os dados JSON de uma requisição POST.

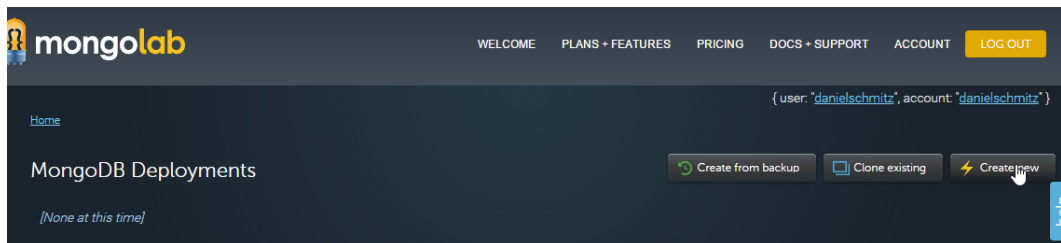
- **mongoose:** É uma adaptador para o banco de dados MongoDB, que é um banco NoSql que possui funcionalidades quase tão boas quanto a um banco de dados relacional.
- **jsonwebtoken:** É uma biblioteca node usada para autenticação via web token. Usaremos esta biblioteca para o login do usuário.

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

6.2 O banco de dados MongoDB

O banco de dados MongoDB possui uma premissa bem diferente dos bancos de dados relacionais (aqueles em que usamos SQL), sendo orientados a documentos auto contidos (NoSql). Resumindo, os dados são armazenados no formato JSON. Você pode instalar o MongoDB em seu computador e usá-lo, mas nesta obra estaremos utilizando o serviço <https://mongolab.com/>¹ que possui uma conta gratuita para bancos públicos (para testes).

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



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

¹<https://mongolab.com/>

²<https://mongolab.com/welcome/>

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

Single-node Replica set cluster

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

Standard Line

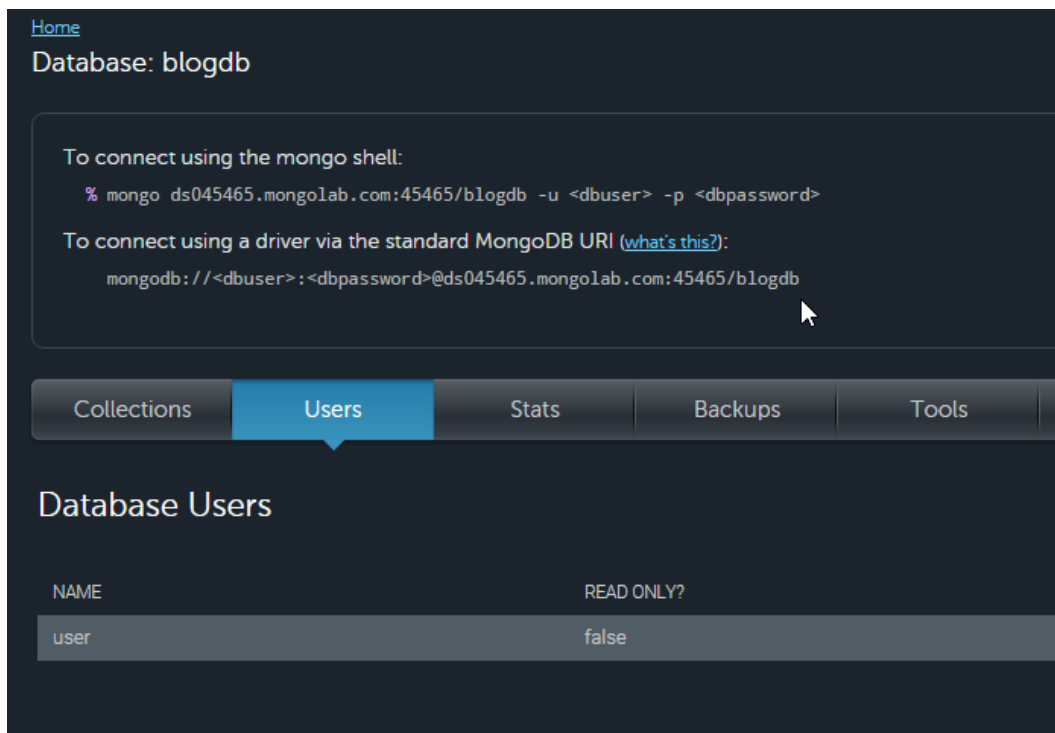
The most economical plans for production applications running on AWS. Plans come standard with 2 data nodes plus an arbiter node.

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

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



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



[Home](#)
Database: blogdb

To connect using the mongo shell:

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

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

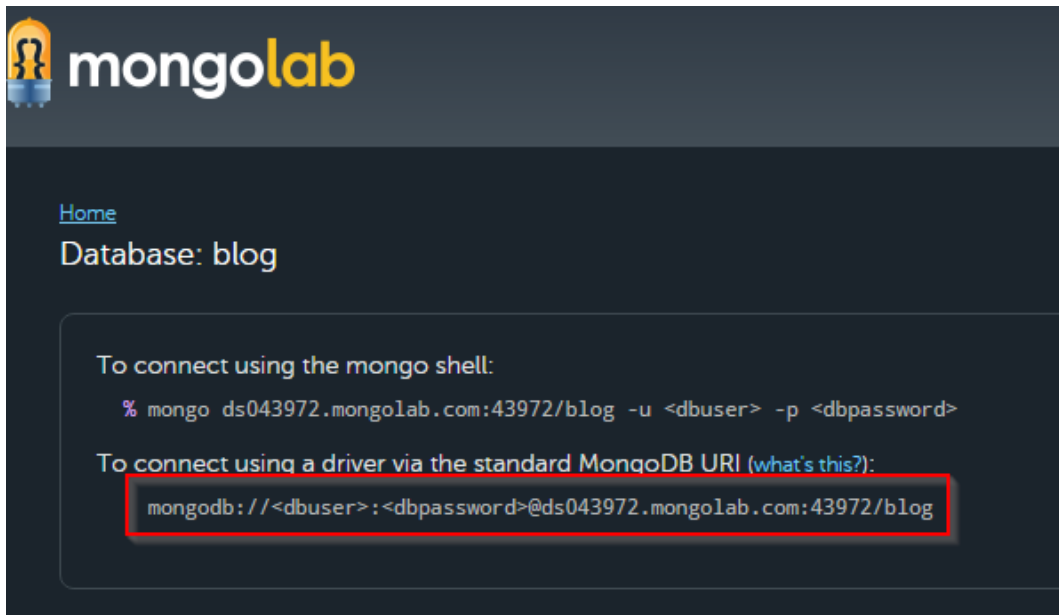
```
mongodb://<dbuser>:<dbpassword>@ds045465.mongolab.com:45465/blogdb
```

Collections **Users** Stats Backups Tools

Database Users

NAME	READ ONLY?
user	false

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



6.3 Criando o projeto

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

```
vue init browserify-simple 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 Ex\
press
|- dist - Contém o arquivo compilado Javascript do Vue
|- index.html - Arquivo que contém todo o entry-point da a\
plicação
```

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 MondoDB

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 dos Posts 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.

A linha 16 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

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

Na linha 18 configuramos o diretório `public` 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 tenta 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     } else {
97         User.findOne({ login: req.body.login,
98             password: req.body.password }, 'name')
99             .exec(function (err, user) {
100                 if (err) res.send(err);
101                 if (user != null) {
102                     var token = jwt.sign(user, secretKey, {
103                         expiresIn: "1 day"
104                     });
105                     res.json({ user: user, token: token });
106                 } else {
107                     res.status(400).send('Login/Senha incorretos');
108                 }
109             });
110     }
111 });
```

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 não 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 );
134     post.save(function (err) {
```



```
135         if (err)
136             res.send(err);
137         res.json(post);
138     });
139 })
140 .delete(auth, function (req, res) {
141     Post.remove({
142         _id: req.params.post_id
143     }, function(err, post) {
144         if (err)
145             res.send(err);
146         res.json({ message: 'Successfully deleted' });
147     });
148 });
```

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
```

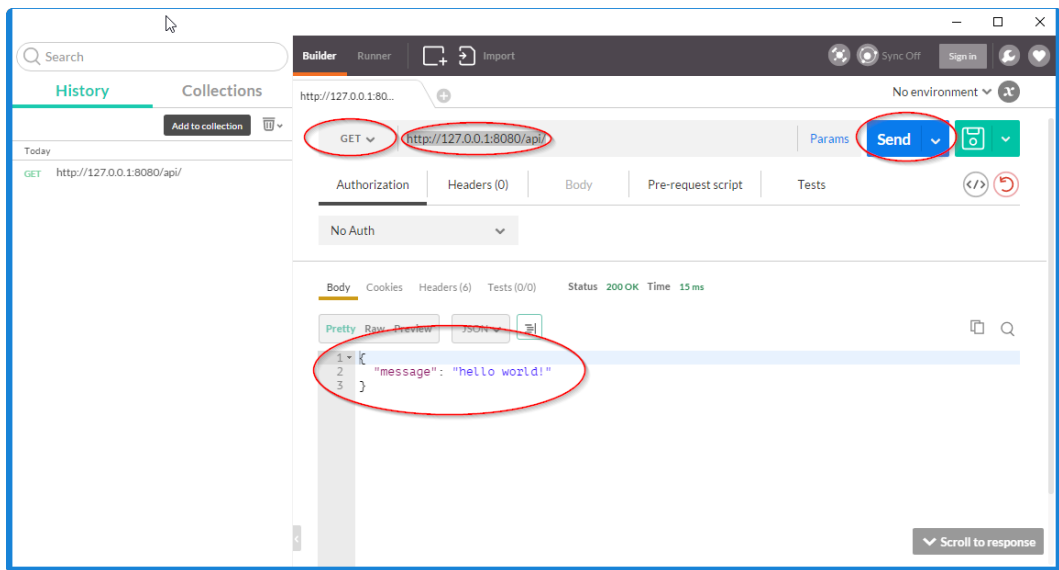
A resposta já indica que, sempre quando o arquivo `server.js` for atualizado, o comando `node server.js` também será.

6.8 Testando a api sem o Vue

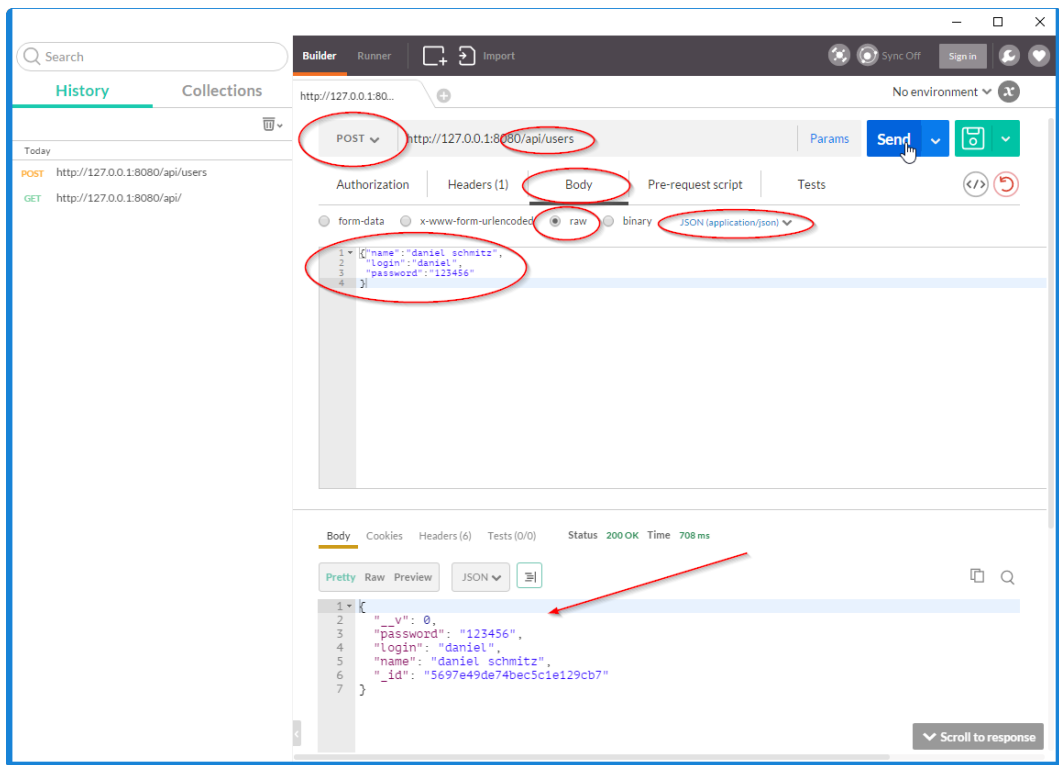
Pode-se testar a api que acabamos de criar, enviando e recebendo dados, através de um programa capaz de realizar chamadas Get/Post ao servidor. Um destes programas se chama **Postman**³, que pode ser instalado com um plugin para o Google Chrome.

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

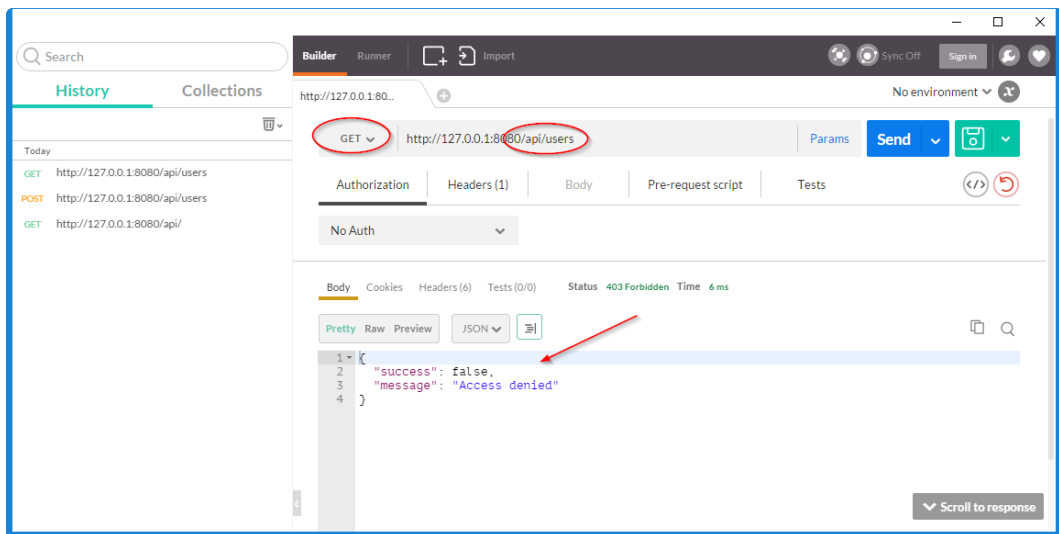
³<https://www.getpostman.com/>



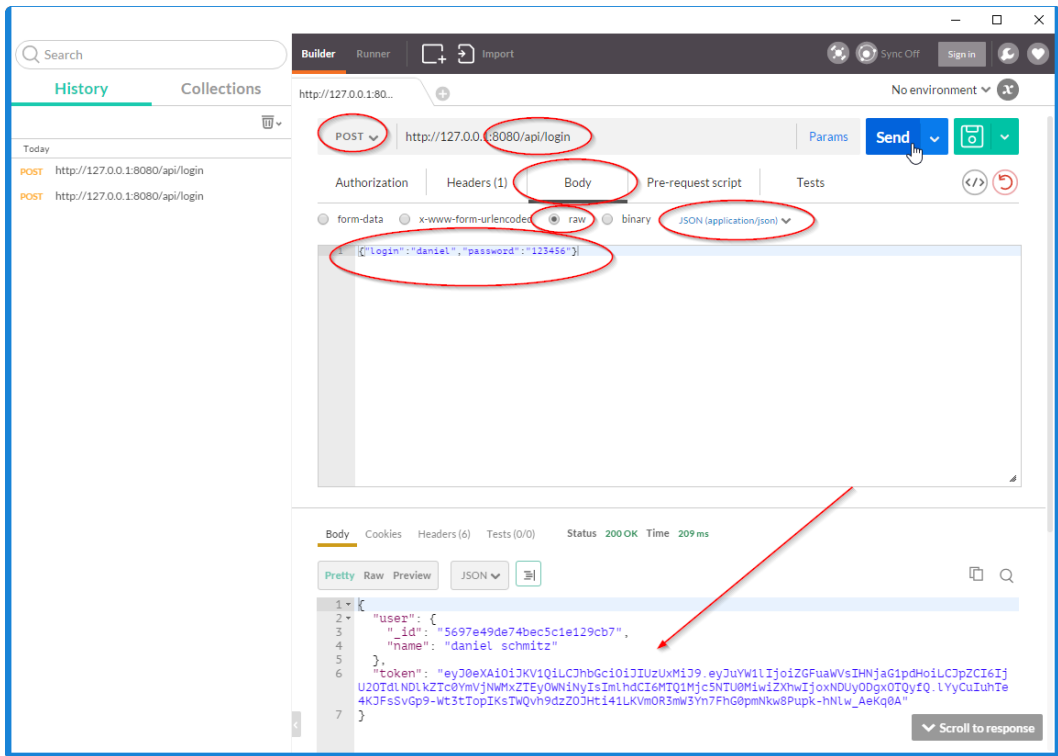
Perceba que obtemos a resposta “hello world”, conforme configurado no servidor. Para criar um usuário, podemos realizar um POST à url /users repassando os seguintes dados:



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:

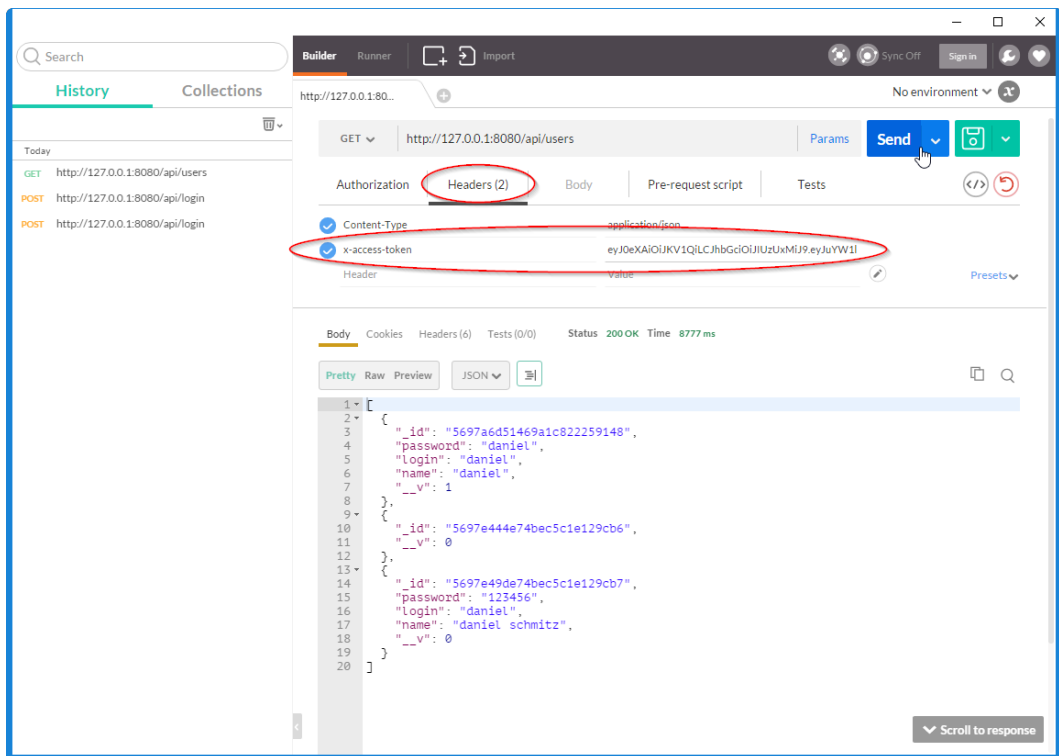


Para realizar o login, acessamos a URL `/api/login`, da seguinte forma:



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:



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

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 aceitar 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

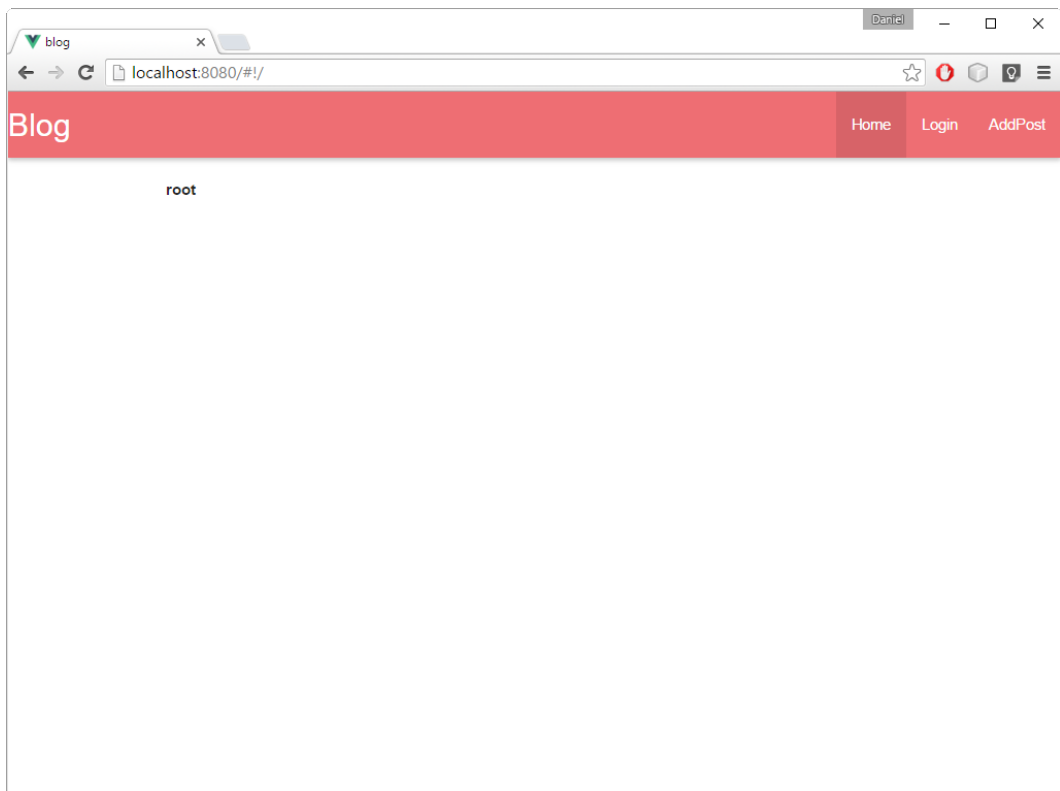
```
<template>
  <div id="app">
    <nav>
      <div class="nav-wrapper">
        <a href="#" class="brand-logo">Blog</a>
        <ul id="nav-mobile" class="right hide-on-med-and-down">
          <li v-link-active><a v-link="{ path: '/' }">Home</a></li>
          <li v-link-active><a v-link="{ path: '/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> {{post.user.name}}</span>
          <!-- <a href="#">This is a link</a> -->
        </div>
      </div>
    </div>
  </div>
</template>
<script>
  export default {
```

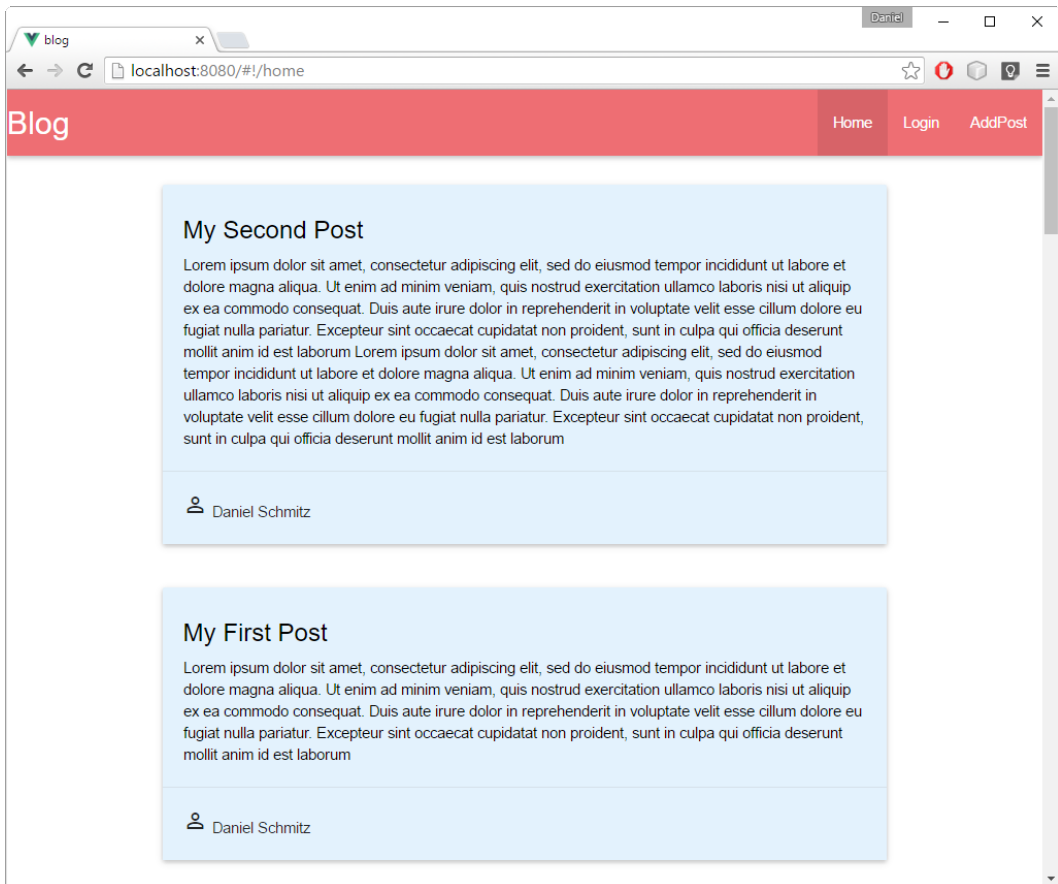
```

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>



7.6 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')
```

7.7 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>
i>
          <input id="login" type="text" v-model="user.login\"
" v-validate:login="{ required: false, minlength: 4 }" />
          <label for="login">Login</label>
          <div>
            <span class="chip red lighten-5 right" v-if="$v\
alidateForm.login.required">Campo requerido</span>
            <span class="chip red lighten-5 right" v-if="$v\
alidateForm.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="use\
r.password" v-validate:password="{ required: false, minleng\
th: 4 }" />
          <label for="password">Password</label>
          <div>
            <span class="chip red lighten-5 right" v-if="$v\
alidateForm.password.required">Campo requerido</span>
            <span class="chip red lighten-5 right" v-if="$v\
```



```
alidateForm.password.minlength">Mínimo 4 caracteres</span>
    </div>
  </div>

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

  <div class="input-field col s12 m9" v-show="user.is\
New">
    <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</button>
</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 quatro campos de formulário, sendo os dois primeiros representando o usuário e a senha. 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 para os usuários que estão se cadastrando. Além do login, iremos cadastrar o usuário. 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.$http.post('/api/login', this.user).then(function\
(response) {
        this.showProgress = false;
        console.log(response);
        this.$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.

7.8 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.

7.9 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="$v\
alidateForm.login.required">Campo requerido</span>
            <span class="chip red lighten-5 right" v-if="$v\
alidateForm.login.minlength">Mínimo 4 caracteres</span>
          </div>
        </div>
      </div>
    </form>
  </validator>
</template>
```

```

    </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;
```

7.10 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*.

7.11 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, 3\
000)
        })
      }
    }
  }
</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.

7.12 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.

Nas próximas semanas estaremos implementando um novo sistema, com apache/php/mysql, e para isso você pode contribuir fornecendo ideias e sugestões na nossa [página de issues].(<https://github.com/danielschmitz/vue-codigos/issues>). Contribua você também!