Cours Laravel 5.5 – vue.js

Dans ce chapitre je vous propose de voir un exemple simple d'utilisation d'un framework Javascript, en l'occurrence Vue.js, comme gestionnaire côté client associé à Laravel.

Je vous propose de démarrer le projet à partir de celui que j'ai élaboré dans cet article avec Materialize (lien direct de téléchargement). Il y a un lien dans l'article pour vous permettre de télécharger le projet de départ.

Comme exemple je vous propose une simple page qui permet de laisser une pensée pour les utilisateurs enregistrés, donc un simple texte qu'on a envie de partager.

*Ce n'est pas une initiation à **Vue.js**, pour ça vous pouvez vous référer à mon cours ici. C'est juste pour montrer son utilisation avec Laravel.*

Mettez à jour le fichier **.env** selon votre configuration, vous pouvez changer le nom de l'application :

```
APP_NAME=Pensées
```

Vous pouvez télécharger le code final de ce chapitre ici.

D'autre part j'ai mis en ligne l'application en démonstration ici. Vous pouvez y laisser vos pensées

Le serveur

Données

On va donc créer une table pour mémoriser les pensées :

```
php artisan make:migration create_pensees_table
```

Changez ainsi le code de la méthode **up** :

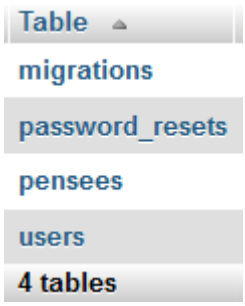
```
public function up()
{
```

```
    Schema::create('pensees', function (Blueprint $table) {
        $table->increments('id');
        $table->timestamps();
        $table->text('text');
        $table->integer('user_id')->unsigned();
        $table->foreign('user_id')->references('id')->on('users');
    });
}
```

Puis lancez la migration :

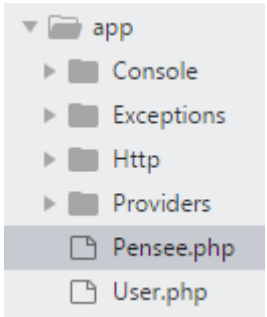
```
php artisan migrate
```

Si tout s’est bien passé vous devez avoir ces 4 tables :



On va aussi créer le modèle :

```
php artisan make:model Pensee
```



On va aussi établir les relations. Dans le modèle **User** on va avoir :

```
public function pensees()
{
    return $this->hasMany(Pensee::class);
}
```

Et la réciproque dans le modèle **Pensee** :

```
public function user()
```

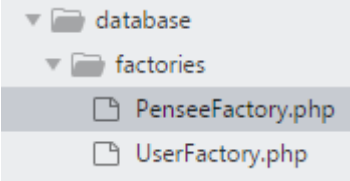
```
{
    return $this->belongsTo(User::class);
}
```

On va aussi penser à l'assignation de masse pour la création d'un pensée dans le modèle **Pensee** :

```
protected $fillable = [
    'text', 'user_id',
];
```

On va un peu remplir nos tables. On commence par créer un factory pour les pensées :

```
php artisan make:factory PenseeFactory --model=Pensee
```



En complétant le code pour la génération du texte :

```
<?php

use Faker\Generator as Faker;

$factory->define(App\Pensee::class, function (Faker $faker) {
    return [
        'text' => $faker->text,
    ];
});
```

Lancez **tinker** :

```
php artisan tinker
```

On va maintenant créer 6 utilisateurs avec chacun une pensée :

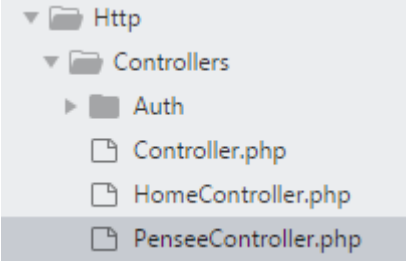
```
factory(App\User::class,  6)->create()->each(function  ($u)
{$u->pensees()->save(factory(App\Pensee::class)->make());});
```

Voilà, pour les données on est parés !

Contrôleur

On va créer un contrôleur de ressource pour nos pensées :

```
php artisan make:controller PenseeController --resource
```



On va garder que les méthodes : **index**, **create** et **destroy** :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PenseeController extends Controller
{
    public function index()
    {
        //
    }

    public function store(Request $request)
    {
        //
    }

    public function destroy($id)
    {
        //
    }
}
```

On va un peu coder tout ça en ajoutant une méthode initiale pour charge l’application :

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Pensee;

class PenseeController extends Controller
{

    public function __construct()
    {
        $this->middleware('auth')->except('app', 'index');
    }

    public function app()
    {
        return view('pensees.index');
    }

    public function index(Request $request)
    {
        $pensees = Pensee::with('user')->latest()->get();

        $user = auth()->check() ? auth()->id() : 0;

        return response()->json([$pensees, $user]);
    }

    public function store(Request $request)
    {
        $request->validate([
            'text' => 'required|max:1000',
        ]);

        $request->merge(['user_id' => $request->user()->id]);

        $pensee = Pensee::create($request->all());

        return Pensee::with('user')->find($pensee->id);
    }

    public function destroy(Pensee $pensee)
    {
        $this->authorize('delete', $pensee);
```

```
        $pensee->delete();

        return response()->json();
    }
}
```

En gros que des choses qu'on a déjà vues dans ce cours…

*La méthode **merge** permet d'ajouter des éléments à la requête.*

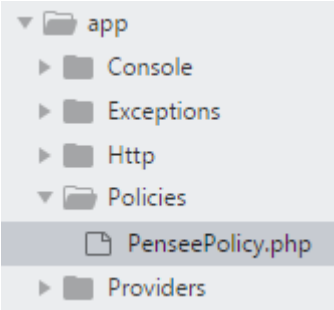
Protections

On doit protéger les urls qui sont réservées aux utilisateurs authentifiée, on va donc ajouter un constructeur dans le contrôleur pour ajouter le middleware **auth** pour les méthodes concernées :

```
public function __construct()
{
    $this->middleware('auth')->except('app', 'index');
}
```

D'autre part il faut faire en sorte que seul le rédacteur d'une pensée puisse supprimer cette pensée :

```
php artisan make:policy PenseePolicy --model=Pensee
```



On va garder seulement la méthode **delete** avec ce code :

```
<?php

namespace App\Policies;

use App\User;
use App\Pensee;
use Illuminate\Auth\Access\HandlesAuthorization;
```

```
class PenseePolicy
{
    use HandlesAuthorization;

    public function delete(User $user, Pensee $pensee)
    {
        return $user->id == $pensee->user_id;
    }
}
```

Il faut l’enregistrer dans **AuthServiceProvider** :

```
use App\ { Pensee, Policies\PenseePolicy };

class AuthServiceProvider extends ServiceProvider
{
    protected $policies = [
        Pensee::class => PenseePolicy::class,
    ];

   ...
```

Et pour terminer on l’utilise dans le contrôleur :

```
public function destroy(Pensee $pensee)
{
    $this->authorize('delete', $pensee);

    $pensee->delete();

    return response()->json();
}
```

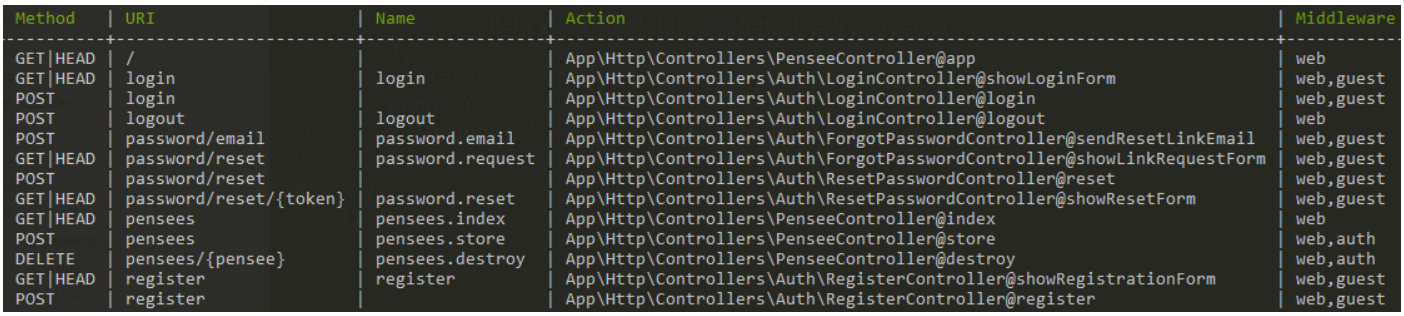
Routes

On a aussi besoin des routes pour accéder aux méthodes du contrôleur qu’on vient de créer et pour un peu réorganiser. Remplacez toutes les routes existantes par celles-ci :

```
Auth::routes();

Route::get('/', 'PenseeController@app');

Route::resource('pensees', 'PenseeController', ['only' => [
```

```
    'index', 'store', 'destroy',
]]);
```

Vous devez donc avoir toutes ces routes :



Method	URI	Name	Action	Middleware
GET|HEAD	/		App\Http\Controllers\PenseeController@app	web
GET|HEAD	login	login	App\Http\Controllers\Auth\LoginController@showLoginForm	web,guest
POST	login		App\Http\Controllers\Auth\LoginController@login	web,guest
POST	logout	logout	App\Http\Controllers\Auth\LoginController@logout	web
POST	password/email	password.email	App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail	web,guest
GET|HEAD	password/reset	password.request	App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm	web,guest
POST	password/reset		App\Http\Controllers\Auth\ResetPasswordController@reset	web,guest
GET|HEAD	password/reset/{token}	password.reset	App\Http\Controllers\Auth\ResetPasswordController@showResetForm	web,guest
GET|HEAD	pensees	pensees.index	App\Http\Controllers\PenseeController@index	web
POST	pensees	pensees.store	App\Http\Controllers\PenseeController@store	web,auth
DELETE	pensees/{pensee}	pensees.destroy	App\Http\Controllers\PenseeController@destroy	web,auth
GET|HEAD	register	register	App\Http\Controllers\Auth\RegisterController@showRegistrationForm	web,guest
POST	register		App\Http\Controllers\Auth\RegisterController@register	web,guest

Notre serveur est maintenant prêt !

Le client

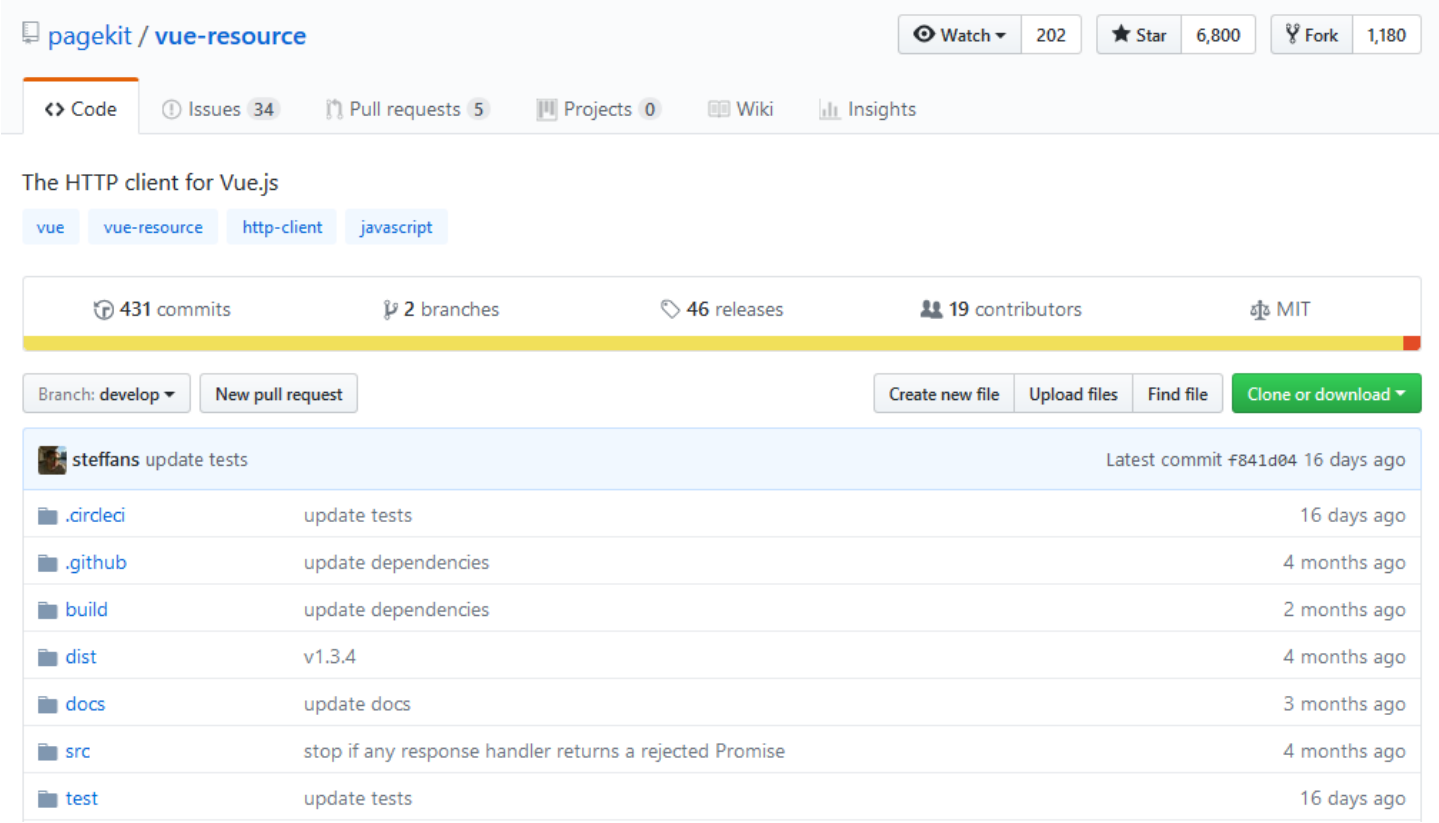
On va donc passer côté client maintenant…

Commencez par générer les modules avec NPM :

```
npm install
```

Vue-resource

Par défaut **Vue.js** n'est pas équipé pour gérer des requêtes HTTP alors on va installer [ce package](#) :



Alors encore une petite commande :

```
npm install vue-resource
```

On va déclarer ce composant dans notre fichier **resources/assets/js/app.js** :

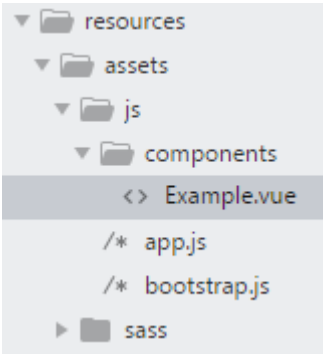
```
window.Vue = require('vue');

var VueResource = require('vue-resource');
Vue.use(VueResource);

Vue.http.headers.common['X-CSRF-TOKEN'] = document.head.querySelector('meta[name="csrf-token"]').content;
```

*La dernière ligne a pour objectif de créer un header pour la protection **CSRF**, normalement **axios** devrait le faire mais pour une raison que j'ignore ça ne fonctionne pas*

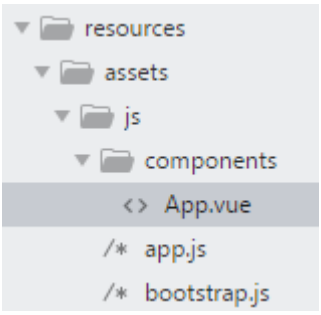
Maintenant on va pouvoir créer des requêtes HTTP vers notre serveur et recevoir des réponses, en gros communiquer !

La liste des pensées

Par défaut on a un composant nommé **Exemple** :



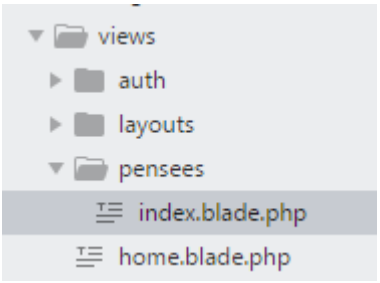
On va changer son nom en l'appelant par exemple **App** :



Et donc renseigner en conséquence le fichier **resources/assets/js/app.js** qui doit donc au final contenir ce code :

```
require('./bootstrap');

window.Vue = require('vue');
var VueResource = require('vue-resource');
Vue.use(VueResource);

Vue.component('app', require('./components/App.vue'));

const app = new Vue({
    el: '#app'
});
```

On va avoir besoin d'une vue de départ :



Avec ce simple code pour charger le composant de Vue.js :

```
@extends('layouts.app')

@section('content')
  <div id="app">
    <app></app>
  </div>
@endsection
```

Une liste simple

Et pour le composant **App.vue** on va commencer par ce code :

```
<template>
    <div class="container">
        <div v-for="pensee in pensees">
          <h4>{{ pensee.user.name }}</h4>
          <p>{{ pensee.text }}</p>
          <p>{{ pensee.created_at }}</p>
        </div>
    </div>
</template>

<script>
    export default {
      resource: null,
      data () {
        return {
          pensees: {}
        }
      },
      mounted () {
        this.resource = this.$resource('/pensees{/id}')
        this.resource.get().then((response) => {
          this.pensees = response.body
```

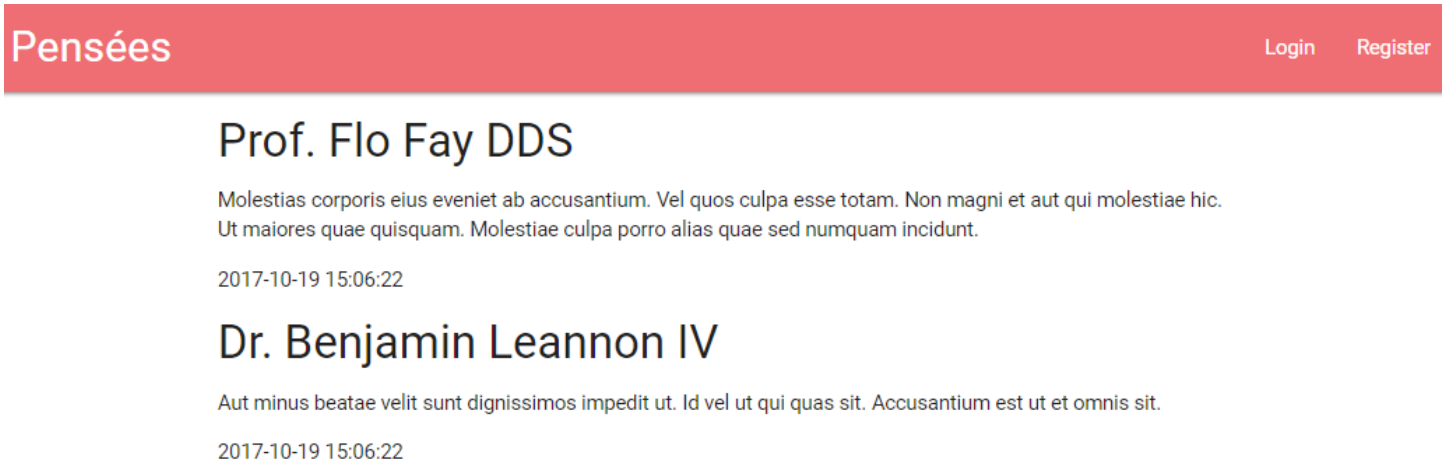
```
        })
      }
    }
</script>
```

Au niveau des **data** une simple variable **pensees** pour contenir les pensées.

Lorsque le composant est prêt (**mounted**) on lance la requête GET **/pensees** et on met le résultat dans **pensees**.

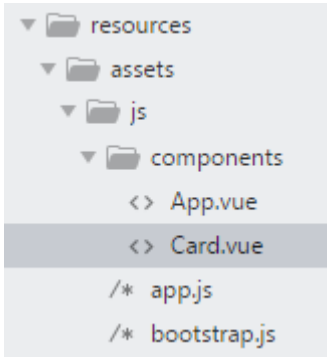
Dans le template la directive **v-for** se charge de générer le HTML.

Si tout se passe bien vous devez obtenir cet aspect :



On améliore l'aspect

On va un peu améliorer l'aspect obtenu. On va créer un composant **Card** :



Avec ce code :

```
<template>
  <div class="card red lighten-2">
```

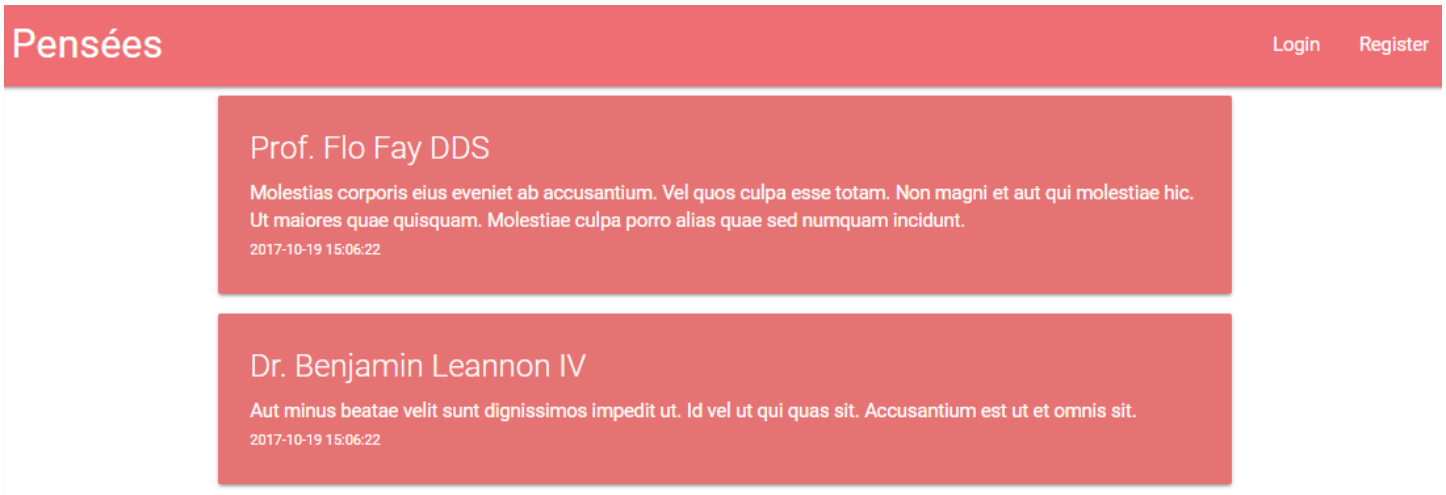
```
    <div class="card-content white-text">
      <span class="card-title">{{ name }}</span>
      <p>{{ text }}</p>
      <p><small>{{ date }}</small></p>
    </div>
  </div>
</template>

<script>
  export default {
    props: ['name', 'text', 'date']
  }
</script>
```

Et on va changer ainsi le code du composant **App** :

```
<template>
    <div class="container">
        <div v-for="pensee in pensees">
            <card :name="pensee.user.name" :text="pensee.text"
:date="pensee.created_at"></card>
        </div>
    </div>
</template>

<script>
    import Card from './Card'

    export default {
      resource: null,
      data () {
        return {
          pensees: []
        }
      },
      mounted () {
        this.resource = this.$resource('/pensees{/id}')
        this.resource.get().then((response) => {
          this.pensees = response.body
        })
      },
      components: {
        Card
      }
```

```
    }
</script>
```

Le code est mieux organisé et l'aspect plus présentable :



Suppression d'une pensée

Maintenant comment faire si un utilisateur veut supprimer une de ses pensées ? il faudrait prévoir un bouton quelque part qui apparaît que pour le rédacteur de la pensée. Mais pour le moment rien ne nous indique que l'utilisateur actuel est le rédacteur de certaines pensées parce qu'on ne dispose pas de son identifiant…

On va donc modifier la méthode **index** du contrôleur :

```
public function index(Request $request)
{
    $pensees = Pensee::with('user')->oldest()->get();

    $user = auth()->check() ? auth()->id() : 0;

    return response()->json([$pensees, $user]);
}
```

Ainsi on renvoie les pensées et l'identifiant de l'utilisateur en cours (0 s'il n'y en a pas).

Voici le nouveau composant **App** :

```
<template>
    <div class="container">
        <div v-for="pensee in pensees">
```

```
                    <card  :pensee="pensee"  :user="user"
@deletePensee="deletePensee"></card>
        </div>
    </div>
</template>

<script>
    import Card from './Card'

    export default {
      resource: null,
      data () {
        return {
          pensees: [],
          user: 0
        }
      },
      mounted () {
        this.resource = this.$resource('/pensees{/id}')
        this.resource.get().then(response => {
          this.pensees = response.body[0]
          this.user = response.body[1]
        })
      },
      components: {
        Card
      },
      methods: {
        deletePensee (id) {
          this.resource.delete({id: id}).then(response => {
             let index = _.findIndex(this.pensees, function(o) {
return o.id == id; })
            this.pensees.splice(index, 1)
          })
        }
      }
    }
</script>
```

On a le nouveau data **user** pour mémoriser l'identifiant de l'utilisateur.

On envoie dans le composant **Card** l'objet **pensee** et **user**.

On attend un événement (**deletePensee**) de la part de **Card**. Cet événement a pour effet d'activer la méthode **deletePensee** qui reçoit l'identifiant de la pensée. On envoie alors la requête pour supprimer cette pensée sur le serveur et au retour on a la supprime en local.

Voici le nouveau code pour **Card** :

```
<template>
  <div class="card red lighten-2">
    <div class="card-content white-text">
      <span class="card-title">{{ pensee.user.name }}</span>
      <p>{{ pensee.text }}</p>
      <p><small>{{ pensee.created_at }}</small></p>
    </div>
    <div v-if="deletable" class="card-action">
       <a href="#" @click.prevent="deletePensee">Supprimer cette
pensée</a>
    </div>
  </div>
</template>

<script>
  export default {
    props: ['pensee', 'user'],
    computed: {
      deletable () {
        return this.pensee.user_id == this.user
      }
    },
    methods: {
      deletePensee () {
        this.$emit('deletePensee', this.pensee.id)
      }
    }
  }
</script>
```

On reçoit maintenant un objet global **pensee** et aussi **user**.

On fait apparaître le lien de suppression avec un **computed** qui teste l'utilisateur.

Pour la suppression on envoie un événement (**$emit**) au parent **App**.

Ajout d'une pensée

Il faut aussi pouvoir ajouter une pensée. Donc un formulaire pour la saisie réservé au utilisateurs connectés. On va utiliser une fenêtre modale pour le réaliser.

On va ajouter un élément dans le menu et l'activation de la fenêtre modale (**resources/views/layouts/app.blade.php**). Voici le code complet de la vue résultante :

```
<!DOCTYPE html>
<html lang="{{ app()->getLocale() }}">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-
scale=1">

    <!-- CSRF Token -->
    <meta name="csrf-token" content="{{ csrf_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Styles -->
    <link href="{{ asset('css/app.css') }}" rel="stylesheet">
    @yield('css')
</head>
<body>
    <div id="app">

        @auth
```

```
            <ul id="dropdown1" class="dropdown-content">
                <li>
                    <a href="{{ route('logout') }}"
                        onclick="event.preventDefault();
                                 document.getElementById('logout-
form').submit();">
                        Deconnexion
                    </a>

                     <form id="logout-form" action="{{
route('logout') }}" method="POST" style="display: none;">
                        {{ csrf_field() }}
                    </form>
                </li>
                <li>
                      <a class="modal-trigger" href="#ajout">Une
pensée</a>
                </li>
            </ul>
        @endauth

        <nav>
            <div class="nav-wrapper">
                    <a href="{{ url('/') }}" class="brand-
logo"> {{ config('app.name', 'Laravel') }}</a>
                    <a href="#" data-activates="mobile-demo"
class="button-collapse"><i class="material-icons">menu</i></a>
                @guest
                    <ul class="right hide-on-med-and-down">
                            <li><a href="{{ route('login')
}}">Connexion</a></li>
                        <li><a href="{{ route('register')
}}">Enregistrement</a></li>
                    </ul>
                    <ul class="side-nav" id="mobile-demo">
                            <li><a href="{{ route('login')
}}">Connexion</a></li>
                        <li><a href="{{ route('register')
}}">Enregistrement</a></li>
                    </ul>
                @else
                    <ul class="right hide-on-med-and-down">
                         <li><a class="dropdown-button" href="#!"
```

```
data-activates="dropdown1">{{   Auth::user()->name   }}<i
class="material-icons right">arrow_drop_down</i></a></li>
                    </ul>
                @endguest
            </div>
        </nav>
        @yield('content')
    </div>

    <!-- Scripts -->
    <script src="{{ asset('js/app.js') }}"></script>
    <script>
        $(".button-collapse").sideNav()
        $(".dropdown-button").dropdown()
        $('.modal').modal()
    </script>
</body>
</html>
```

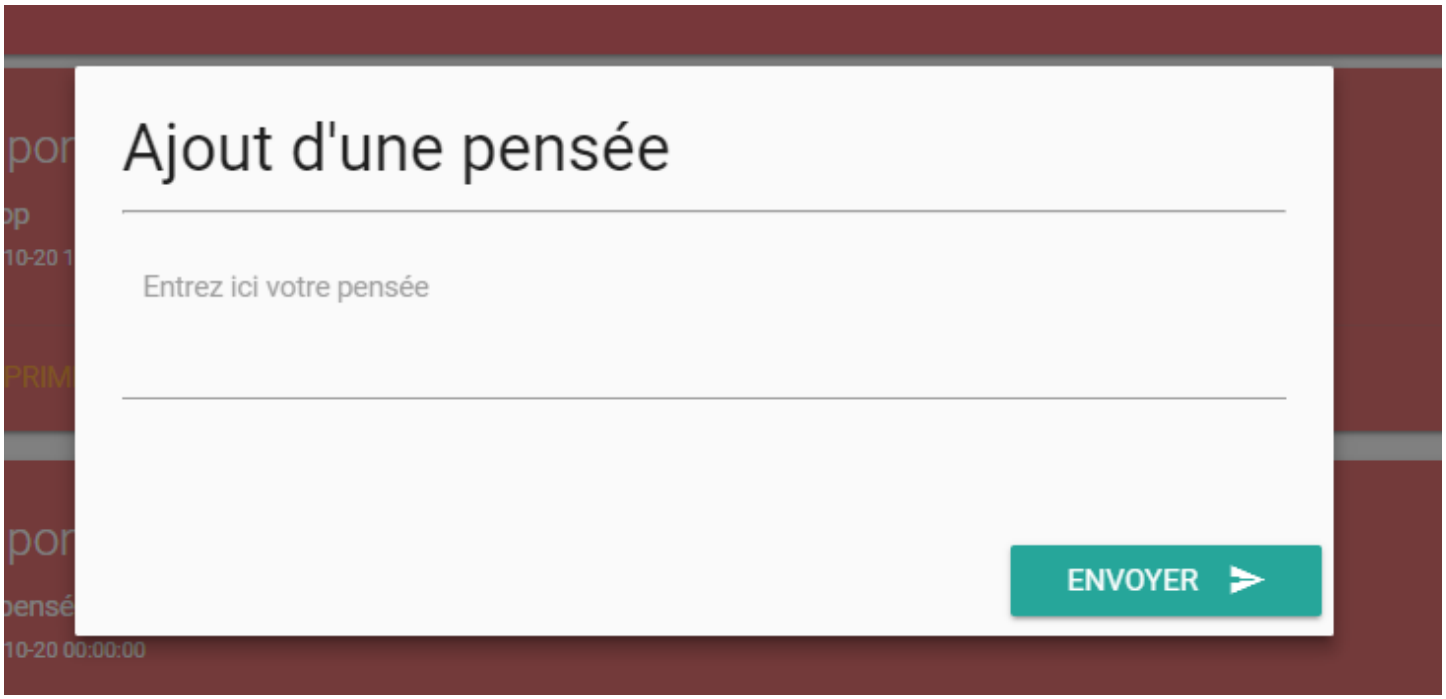
On ajoute le code de la fenêtre modale dans le composant **App** ainsi que sa gestion (mise à jour, validation…) :

```
<template>
    <div class="container">
        <div v-for="pensee in pensees" :key="pensee.id">
                     <card  :pensee="pensee"  :user="user"
@deletePensee="deletePensee"></card>
        </div>
        <div id="ajout" class="modal">
            <form v-on:submit.prevent="addPensee">
                <div class="modal-content">
                    <h4>Ajout d'une pensée</h4>
                    <hr>
                    <div class="input-field col s12">
                          <textarea id="pensee" v-model="texte"
class="materialize-textarea"></textarea>
                           <label for="pensee">Entrez ici votre
pensée</label>
                       <p class="red-text">{{ error }}</p>
                    </div>
                </div>
                <div class="modal-footer">
                     <button class="btn waves-effect waves-light"
type="submit">Envoyer
```

```
                        <i class="material-icons right">send</i>
                    </button>
                </div>
            </form>
        </div>
    </div>
</template>

<script>
    import Card from './Card'

    export default {
      resource: null,
      data () {
        return {
          error: '',
          texte: '',
          pensees: [],
          user: 0
        }
      },
      mounted () {
        this.resource = this.$resource('/pensees{/id}')
        this.resource.get().then(response => {
          this.pensees = response.body[0]
          this.user = response.body[1]
        })
      },
      components: {
        Card
      },
      methods: {
        deletePensee (id) {
          this.resource.delete({id: id}).then(response => {
             let index = _.findIndex(this.pensees, function(o) {
return o.id == id; })
            this.pensees.splice(index, 1)
          })
        },
        addPensee () {
           this.resource.save({ text: this.texte }).then(response
=> {
            $('#ajout').modal('close')
```

```
            this.texte = ''
            this.error = ''
            this.pensees.unshift(response.body)
          }, response => {
            this.error = response.body.errors.text[0]
          })
        }
      }
    }
</script>
```

Et maintenant on peut ajouter des pensées !



Les dates en français

Vous avez sans doute remarqué que le format des dates est celui par défaut qui n'est pas très adapté au français. On pourrait changer ça au niveau de Laravel mais on va le faire avec Javascript.

On va utiliser la superbe librairie moment (l'équivalent de Carbon pour Javascript) :

```
npm install moment
```

Et on l'utilise dans le composant **Card** :

```
<template>
```

```
  <div class="card red lighten-2">
    <div class="card-content white-text">
      <span class="card-title">{{ pensee.user.name }}</span>
      <p>{{ pensee.text }}</p>
      <p><small>{{ date }}</small></p>
    </div>
    <div v-if="deletable" class="card-action">
       <a href="#" @click.prevent="deletePensee">Supprimer cette
pensée</a>
    </div>
  </div>
</template>

<script>
  import moment from 'moment'
  moment.locale('fr')

  export default {
    props: ['pensee', 'user'],
    computed: {
      deletable () {
        return this.pensee.user_id == this.user
      },
      date () {
         return moment(pensee.created_at).format('D MMMM YYYY à
H:mm:ss')
      }
    },
    methods: {
      deletePensee () {
        this.$emit('deletePensee', this.pensee.id)
      }
    }
  }
</script>
```

On importe la librairie :

```
import moment from 'moment'
```

On fixe la locale :

```
moment.locale('fr')
```

On crée un élément calculé :

```
date () {
  return moment(pensee.created_at).format('D MMMM YYYY à H:mm:ss')
}
```

Et on l'utilise dans le template :

```
<p><small>{{ date }}</small></p>
```

Et maintenant on a les dates en bon français :

La pagination

Si on commence à avoir beaucoup de pensées il va nous falloir une pagination :

```
npm install vue-paginate
```

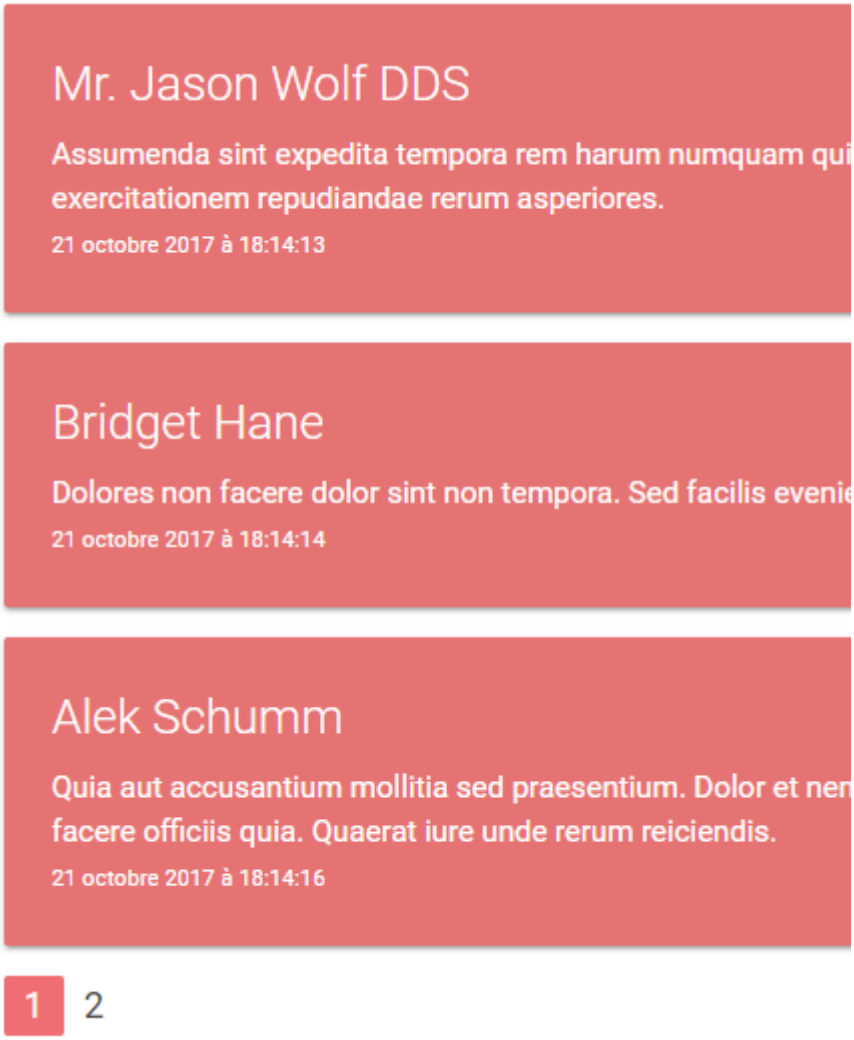
Et voilà le code modifié en conséquence pour le composant **App** :

```
<template>
    <div class="container">
      <paginate name="pensees" :list="pensees" :per="3">
            <li v-for="pensee in paginated('pensees')"
:key="pensee.id">
                  <card :pensee="pensee" :user="user"
@deletePensee="deletePensee"></card>
        </li>
      </paginate>
          <paginate-links for="pensees" :classes="{'ul':
'pagination'}"></paginate-links>
      <div id="ajout" class="modal">
          <form v-on:submit.prevent="addPensee">
              <div class="modal-content">
                  <h4>Ajout d'une pensée</h4>
                  <hr>
                  <div class="input-field col s12">
                        <textarea id="pensee" v-model="texte"
class="materialize-textarea"></textarea>
```

```
                        <label for="pensee">Entrez ici votre
pensée</label>
                    <p class="red-text">{{ error }}</p>
                </div>
            </div>
            <div class="modal-footer">
                 <button class="btn waves-effect waves-light"
type="submit">Envoyer
                    <i class="material-icons right">send</i>
                </button>
            </div>
        </form>
    </div>
  </div>
</template>

<script>
  import Card from './Card'
  import VuePaginate from 'vue-paginate'

  export default {
    resource: null,
    data () {
      return {
        error: '',
        texte: '',
        pensees: [],
        paginate: ['pensees'],
        user: 0
      }
    },
    mounted () {
      this.resource = this.$resource('/pensees{/id}')
      this.resource.get().then(response => {
        this.pensees = response.body[0]
        this.user = response.body[1]
      })
    },
    components: {
      Card
    },
    methods: {
      deletePensee (id) {
```

```
          this.resource.delete({id: id}).then(response => {
             let index = _.findIndex(this.pensees, function(o) {
return o.id == id; })
            this.pensees.splice(index, 1)
          })
        },
        addPensee () {
           this.resource.save({ text: this.texte }).then(response
=> {
            $('#ajout').modal('close')
            this.texte = ''
            this.error = ''
            this.pensees.unshift(response.body)
          }, response => {
            this.error = response.body.errors.text[0]
          })
        }
      }
    }
</script>
```

On a maintenant une pagination :



On pourrait encore améliorer cette application mais on en restera là…

En résumé

Laravel n'impose rien au niveau de la gestion côté client et ce ne sont pas les solutions qui manquent. Toutefois l'installation de Laravel prévoit par défaut une intendance pour Vue.js. On a vu dans ce chapitre que cette intendance est au point et que tout fonctionne de façon efficace. La prise en main de Vue.js est moins difficile que d'autres framework comme Angular.

Reste que le choix des outils de gestion côté client n'est pas évident. Souvent JQuery est amplement suffisant lorsqu'on a juste à générer des requêtes HTTP et à manipuler un peu le DOM. A quel moment devient-il plus intéressant de passer à un framework plus complet ? La réponse n'est pas facile parce que dépendant de

plusieurs facteurs dont la maîtrise qu’on a d’un outil particulier n’est pas le moindre.

Si vous n’avez jamais utilisé Vue.js je vous conseille de l’essayer parce qu’il est simple et puissant et que sa courbe d’apprentissage n’est pas rebutante.