

Laravel : Broadcasting partie 1 (public channel)

1.Introduction

Il y a de plus en plus d'applications web qui utilisent le **Websocket** permettant d'afficher les informations aux utilisateurs en temps réel. Etant l'une des frameworks PHP les plus en vogue actuellement, Laravel offre avec "Laravel Echo", un système de "**broadcasting** d'évènement" qui donne la possibilité, coté client, de facilement se souscrire à des évènements qui sont diffusés en Back-End via des **canaux** ou **channels** par le serveur. Le coté client réagit vis-à-vis de ces évènements sans que celui-ci ait besoin de rafraichir la page. Ce TP va vous guider pas à pas à utiliser Laravel Echo et Pusher.

2.CONFIGURATION

On va avoir besoin du service de **broadcasting** de Laravel, il faut donc activer le service provider correspondant dans `/config/app.php` en décommentant cette ligne:

```
<?php
App\Providers\BroadcastServiceProvider::class
```

Cette classe va ajouter les routes nécessaires au broadcasting d'évènements et ensuite charger les routes qui se trouvent dans le fichier `routes/channels.php`

Dans le fichier `config/broadcasting.php`, on a les configurations du broadcasting de Laravel. Laravel support par default les drivers suivants: **pusher**, **redis**, **log**, **null**


Le driver log nous permet de diffuser un évènement dans le fichier log de Laravel, utilisé surtout à des fins de test et de débogage.

Mais nous allons utiliser pusher parce que pusher propose un service gratuit et c'est surtout le plus facile à utiliser.


Dans le fichier `.env`, modifier le `BROADCAST_DRIVER=log` en `BROADCAST_DRIVER=pusher`.

Dans la configuration de connexion de pusher `config/broadcasting.php`, n'oublier pas de mettre `encrypted` à `false` pour que Pusher accepte aussi les requêtes http et pas seulement des https. (A noter que ceci est seulement pour les environnements de développement et n'est surtout pas recommandé en production).


On doit créer un compte sur dashboard.pusher.com et créer notre première application.





Welcome to Pusher Channels!

Name your app 

Blog


Select a cluster 

eu (EU (Ireland)) 


☐ Create apps for multiple environments? 

Choose your tech stack (optional)

Front end

JQuery 

Back end

Laravel 

Create app

Cancel

Après la création du compte, on arrive à l'écran de génération d'application pusher, il suffit de nommer notre app et cliquer sur "Create my app"

Ensuite, on est redirigé sur le dashboard de notre app. Sur ce dashboard on a un onglet *App Keys* qui contient la configuration de notre app.

App keys

If a token is compromised, you can create a new key/secret pair.
You should delete the old token once you've updated your app.

[Overview](#)[Getting Started](#)**| App Keys**[Stats](#)[Debug Console](#)[Error Logs](#)[Webhooks](#)[Collaborators](#)[App Settings](#) Created 2 minutes ago Copy

```
app_id = "1556712"  
key = "d060438970a2a9259582"  
secret = "1ec10896fec0ed46f0bf"  
cluster = "eu"
```

[Create new key and secret](#)

Dans notre projet Laravel, on doit installer les dépendances de pusher

```
composer require pusher/pusher-php-server
```

Et y renseigner les configurations suivantes (dans le .env):

```
PUSHER_APP_ID= 1556712
```

```
PUSHER_APP_KEY= d060438970a2a9259582
```

```
PUSHER_APP_SECRET= 1ec10896fec0ed46f0bf
```

```
PUSHER_APP_CLUSTER=eu
```

Pour terminer la configuration, il faut installer les diffuseurs javascript coté client, notamment **laravel-echo** et **pusher-js**.

```
npm install laravel-echo pusher-js
```

3. UTILISATION

3.1 DIFFUSION DU MESSAGE DU COTÉ SERVEUR / PUSHER

On aura besoin d'un évènement Laravel à diffuser. Créons un évènement *ProfileUpdated* qui va, soit disant, s'exécuter lorsque le profil d'un student est modifié.

```
php artisan make:event ProfileUpdated
```

Par défaut, l'évènement *ProfileUpdated* est un évènement *standard* coté serveur avec lequel on peut brancher plusieurs *listeners*.

Pour en faire un évènement diffusable coté client, il faut que celui-ci implémente l'interface "**ShouldBroadcast**" déjà importer avec:

```
<?php
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
```

La déclaration de la classe *ProfileUpdated* deviendra donc :

App\Events;

```
<?php
class ProfileUpdated implements ShouldBroadcast
```

Par défaut, on a la fonction *broadcastOn* qui retourne un *PrivateChannel* nommé "channel-name".

Un "PrivateChannel" (`Illuminate\Broadcasting\PrivateChannel`) est un channel qui ne peut être écouté que par des utilisateurs authentifiés et autorisés.

Pour commencer, nous allons utiliser le public channel. Dans la fonction *broadcastOn*, on aura:

```
<?php
public function broadcastOn()
{
    return new Channel('profile');
}
```

On va supposer que l'action *update* fait la mise à jour d'un student et exécute l'évènement *ProfileUpdated*.

```
public function update(Request $request, Student $student)
{

    //validation
    $rules=[
        'name'=>['required','max:100'],
        'mail'=>['required','max:100','email'],
        'picture'=>$request->hasFile('picture') ?
['required','max:1024','image'] : "",
        'section'=>['required','max:100'],
    ] ;
    $messages=[
        'name.required'=>'please write your name',
    ];
    $validateForm=Validator::make($request->all(),$rules,$messages);
    if($validateForm->fails()){
        return redirect()->back()->withErrors($validateForm);
    }
    //
    if($request->hasFile('picture')){
        //delete image from server
        unlink('storage/'.$student->picture);
        $student->picture=$request->picture->store('students','public');
    }
    $student=$student->update([
        'name'=>$request->name,
        'mail'=>$request->mail,
        'section'=>$request->section,
        'picture'=>$student->picture
    ]);
    ProfileUpdated::dispatch($student);
    return redirect()->route('students.index')->withSuccess('student
updated successfully');
```

```
}
```

L'évènement *ProfileUpdated* doit donc recevoir un objet Student:

```
<?php

namespace App\Events;



use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;


class ProfileUpdated implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;
    public $student;
    public function __construct(Student $student)
    {
        $this->student=$student;
    }

    public function broadcastOn()
    {
        return new Channel('profile');
    }
}
```

Pour vérifier si tout fonctionne bien, lancer le serveur php (php artisan serve), naviguer vers la route *students.update* et vérifier sur dashboard.pusher.com si l'évènement a été capturé.

Sur dashboard.pusher.com, aller dans l'onglet "Debug console" pour voir la liste des évènements capturés par pusher. Vous devriez voir le student avec l'id 74:

 Event creator 

| EVENT | DETAILS | TIME |
|---|--|--|
| API message | Channel: profile, Event: App\Events\ProfileUpdated | 11:13:31  |
| <pre>{ "student": { "id": 74, "name": "Sara", "picture": "students/38Es1HLrkzCtox9oYRKYwtiab1DPLiBSE9voypf4.png", "section": "info", "mail": "sara@gmail.com", "created_at": "2023-01-31T21:29:27.000000Z", "updated_at": "2023-02-19T11:13:30.000000Z" } }</pre> | | |

3.2 ECOUTE DU MESSAGE DU COTÉ CLIENT

Maintenant qu'on a compris comment diffuser le message sur Pusher, nous allons voir comment écouter sur le channel donné avec Javascript, Laravel Echo, et pusher-js afin de réagir vis-à-vis du message.

Vu qu'on va modifier des fichiers javascript, il est nécessaire d'exécuter la commande de compilation de npm.

```
npm run dev
```

Dans `resources/js/bootstrap.js`, décommenter les lignes correspondantes à Laravel-echo et comme dans `config/broadcasting.php`, mettre `encrypted` à `false`.

// `resources/js/bootstrap.js`

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
  broadcaster: 'pusher',
```



```

    key: import.meta.env.VITE_PUSHER_APP_KEY,
    wsHost: import.meta.env.VITE_PUSHER_HOST ? import.meta.env.VITE_PUSHER_HOST :
`ws-${import.meta.env.VITE_PUSHER_APP_CLUSTER}.pusher.com`,
    wsPort: import.meta.env.VITE_PUSHER_PORT ?? 80,
    wssPort: import.meta.env.VITE_PUSHER_PORT ?? 443,
    forceTLS: (import.meta.env.VITE_PUSHER_SCHEME ?? 'https') === 'https',
    enabledTransports: ['ws', 'wss'],
    cluster:import.meta.env.VITE_PUSHER_APP_CLUSTER, //add this line
  });

```

Ensuite, on va modifier notre template `welcome.blade.php` et y ajouter la directive **@vite** dans le tag head:

```
@vite('resources/js/app.js')
```

```

<body>
  <script>
    window.onload=function(){
      Echo.channel('profile')
        .listen('ProfileUpdated', e => {
alert('profile updated :'+JSON.stringify(e.student.name));

      });
    }
  </script>
</body>

```

- la méthode **channel** correspond au channel publique "Channel". Noter que cette méthode varie selon le type de channel utilisé (ex: private pour les PrivateChannel).
- "profile" est le nom de notre channel dans **app/Events/ProfileUpdated.php**
- "**ProfileUpdated**" est le nom de l'évènement qu'on va écouter, par convention Echo utilise le nom complet de la classe mais on n'a pas besoin de le spécifier parce que Echo va assumer que la classe donnée se trouve dans le namespace **App\Event**

- e est l'évènement qu'on va recevoir. A la réception du message, on va juste afficher un message dans le console ou bien alerte.

Pour voir le résultat, il nous faut 2 navigateurs différents:

- Ouvrez deux navigateurs différents (navigateur A et B) cote à cote
- Dans le navigateur A:
 - naviguer sur la page welcome.
 - vider le contenu de la console.
- Dans le navigateur B:
 - naviguer sur la route update
- Dans la console du navigateur A, vous devriez apercevoir le message comme dans la figure suivante:

