**Les Évènements**

Dans notre cas, nous allons créer un système d’évènement pour notre formulaire de contact, ce qui sera mis sous écouteurs afin de gérer chaque évènement.

1. **Création**

**php artisan make:event ContactRequestEvent**

Cela créera une classe qui sera stockée dans le dossier Events de notre application.

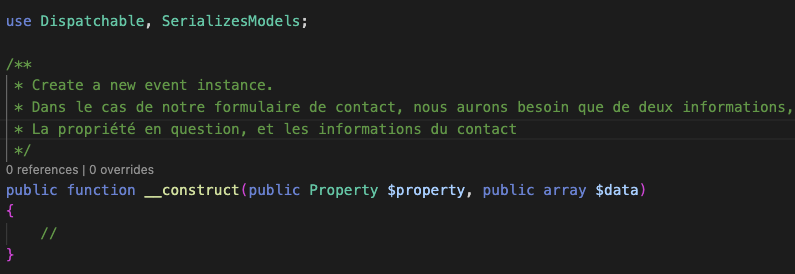
Nous remarquons ici que cela utilise 3 autres :

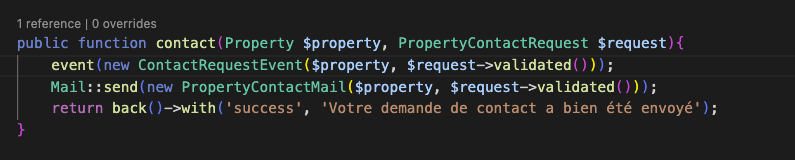
use Dispatchable, InteractsWithSockets, SerializesModels;**traits**

**Dispatchable :** Pour dispatcher les évènements, cela contient des méthodes un peu rapides pour pouvoir émettre l’évènement.

**InteractsWithSockets :** Pour les évènements qui ont attraits aux sockets

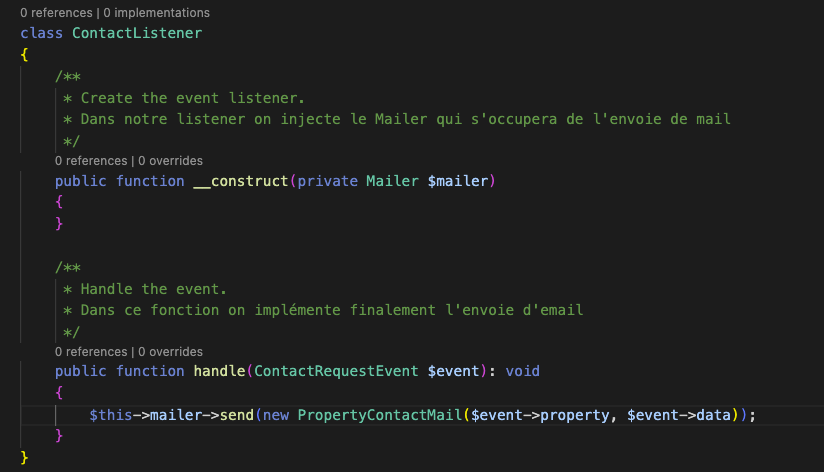
**SerializesModels :** Qui nous montre comment sarialiser les models

***info :***  *Au niveau du constructeur, on lui passe des informations qui seront ensuite sauvegardées dans notre évènement. De ce fait, on sauvegarde les informations les plus minimes pour ne pas que cela prenne trop de place*

Dans notre cas, on pourrait lui dire d’émettre un événement à chaque fois que quelqu’un essaye de nous contacter. Donc dans notre **PropertyController,** on utilise le mot clé **event()** ou l’injection de dépendance en utilisant **Illuminate\Events\Dispatcher **

1. Cela déclenchera l'événement mais ne fera rien. Pour pouvoir gérer cela nous allons implémenter un **listener** qui permettra d’écouter l'événement et déclencher un comportement spécifique à chaque fois que l'événement est déclenché. Il va falloir également supprimer la ligne de code pour l’envoi d’email qui sera gérée finalement gérer par le **listener.**

**php artisan make:listener ContactListener --event=ContactRequestEvent**

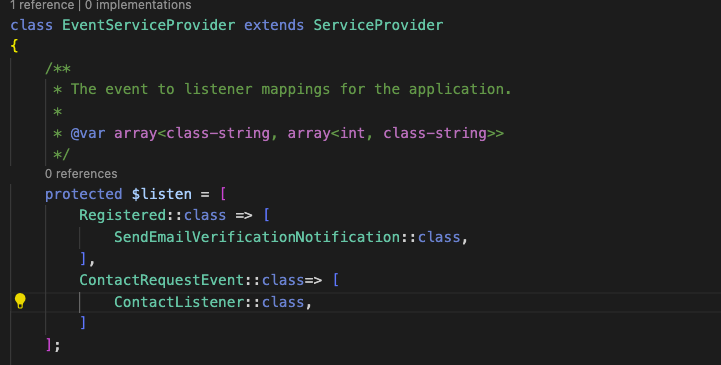
****

Avec cette façon, nous pourrons nous greffer à plusieurs événements et agir en conséquence.

Par exemple, quand l’utilisateur s’inscrit à travers **register du AuthController,** nous pouvons lui envoyer automatiquement un mail.

Après avoir fait cela, nous devons faire le câblage pour indiquer quel listener utiliser pour quel évènement; pour cela on peut modifier le **EventsServiceProvider.**

Dans ce service nous remarquons que nous avons déjà une propriété protégée qui gère **LaravelBreeze** qui gère l’envoi d'e-mail quand un utilisateur s’est inscrit. Nous en ferons de même pour notre formulaire de contact.

**NB:** On peut avoir plusieurs listener pour un événement donné. Sachant qu’un listener peut envoyer un boolean, si un listener renvoie false, la chaîne sera stoppée. 

Cela gère tous les listeners de façon synchrone. Donc si un listener gère les choses de façon lente, l’évènement aura des répercussions; en exemple on pourrait mettre **sleep(3)** dans le handle de notre listener et là nous remarquons cela de façon concrète.

*Nous pouvons de même implémenter un système de file d'attente, en demandant en notre* ***listener,*** *d’implémenter l’interface* ***shouldQueue,*** *sachant que cela demandera d'autres configurations.*

***info:*** *Si on a des tâches un peu plus lentes, il sera préférable de ne pas pas mettre la logique dans un controller mais plutôt de d’émettre un évènement et ce sera aux listeners de se brancher dessus.*

Revenant un peu au **trait Dispatchable,** au lieu d’utiliser le mot clé **event(),** on aurait pu utiliser directement la classe **ContactRequestEvent** et appeler directement les méthodes de ce **trait,** ce qui nous permettrait d'aller un peu plus vite.

**Exemple :** Au cas où on veut déclencher un évènement seulement si une condition est vérifiée:

**ContactRequestEvent::dispacthif(true, $property, $request->validated());**

1. **Les Subscribers**

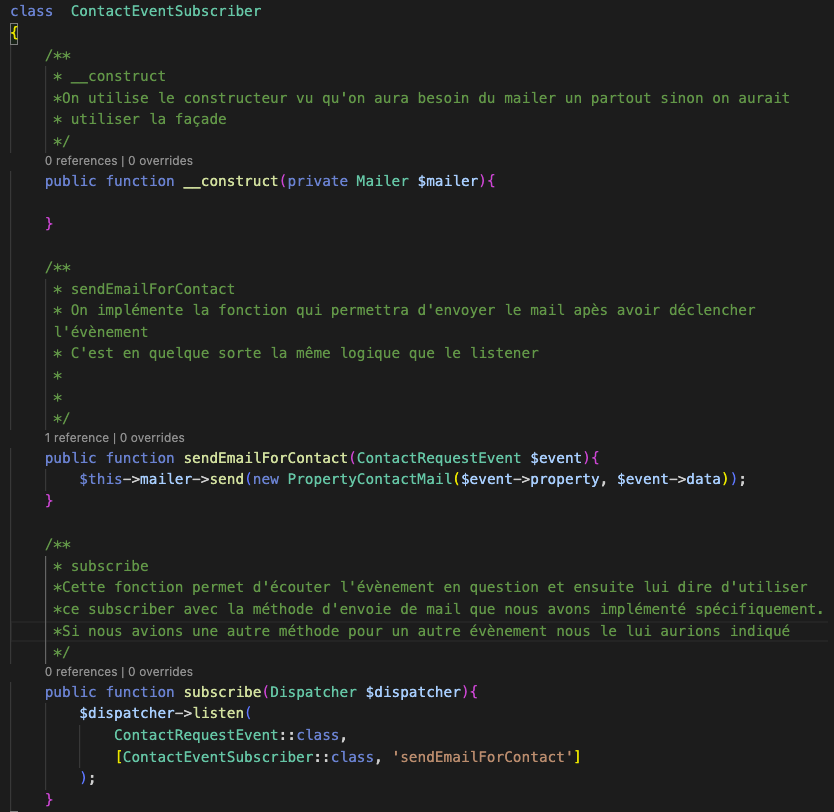
Une autre alternatives aux **listeners** est les **subscribers**, ayant la possibilité de se connecter à plusieurs évènements.

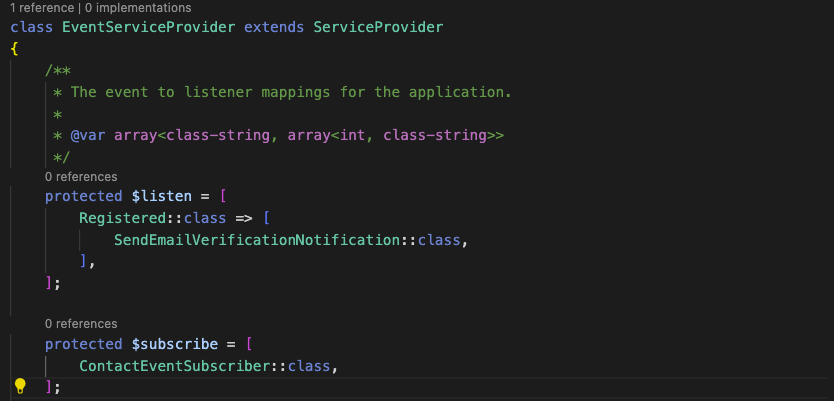
On peut s’imaginer vouloir envoyer des emails dans différentes parties de notre application, plutôt que de créer des **listeners** qui envoyer individuellement ces emails, on peut se créer un **subscriber** qui va s’abonner à plusieurs évènements pour l’envoi des emails pour chacun de ces évènements.

***info:*** *Pour la création d’un subscriber on a pas de commande spécifique en php. du coup, nous allons créer un fichier dans notre dossier Listener qui sera :* **ContactEventSubscriber**

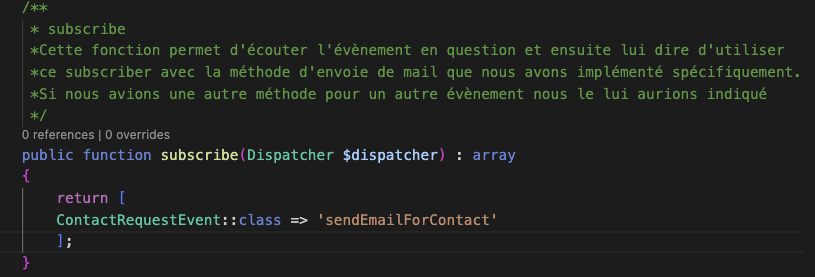
Sur ce dispatcher, on va pouvoir écouter différents évènements et expliquer quelle méthode utiliser, lorsque cet événement est déclenché.

Le code ci-dessous ci💀👇



Après cela, nous devons l’enregistrer dans notre **EventServiceProvider;** à noter que dans le cadre d’un **listener** nous avions utiliser la propriété **$listen**; Dans le cadre d’un subscriber, c’est le **subscriber** lui même qui explique ceux sur quoi il écoute. Donc nous aurons: 

Quand on réactualise notre page on voit bien que l’email est envoyé.

***info:*** *Nous pouvons également simplifier notre* ***subscriber*** *si nous avons plusieurs classes pour la partie envoi et cette méthode ci est appropriée; si nous avons par contre la même classe pour l’envoi nous pouvons simplifier le code en renvoyant un tableau associatif qui va prendre en clé les* ***événements*** *et en valeur la* ***méthode*** *à renvoyer.*

Cela est une approche très intéressante si nous voulons découper le code de notre application de façon structurée et aussi cela nous évite de devoir ajouter manuellement dans notre **EventServiceProvider** une tonne de listeners à ce niveau.

***info :*** *On pourrait également implémenter le* ***ShouldQueue*** *à notre* ***subscriber*** *comme le cas de notre* ***listener.***

**NB:** La grande leçon à retenir dans cette partie est d'émettre des évènements lorsqu’il se passe des choses spécifiques dans notre application même si nous n’utilisons pas les listener là-dessus, cela nous permettrait demain d’ajouter des choses.

**À REGARDER :** SerializesModels