Rabot Dutilleul

Logiciels

- PHP 7.4
- MySQL 8
- Symfony 5.2
- SASS
- Webpack encore
- RabbitMQ

Conventions

PHP

Nous suivons les conventions de la norme PHP Standards Recommendations, notamment :

- PSR-1: Basic Coding Standard
- PSR-12: Extended Coding Style

En complément de ces conventions, nous ajoutons un point : l'écriture des noms de variables se fait en **snake case**

Javascript

Nous suivons la convention de codage issue d'Airbnb, mais modifiée pour mettre l'indentation à 4 espaces.

https://github.com/PortailPro/javascript

Nous passons par Webpack Encore pour la gestion des assets, donc des polyfills sont facilement activables en fonction des navigateurs que nous visons.

C'est pourquoi, dans l'idéal nous évitons de passer par jQuery. La plupart des fonctions disponible dans jQuery sont natives dans la version ES6 de Javacript.

Donc, sauf cas exceptionnel, nous nous basons uniquement sur des bibliothèques de type `Vanilla Javascript` : en pur Javascript sans autre dépendance.

Afin de structurer au maximum notre code et de n'inclure que le code nécessaire dans chaque page, nous allons diviser le code en module que nous allons incorporer en fonction de ce qui est nécessaire pour chaque page via webpack.

Nous utilisons la structure suivante :

```
- /assets
  | - /js
      | - /components
          | - /notification.js
          | - /alert.js
      | - /views
          | - /homepage
              | - /dashboard.js
          | - /login
              | - /form.js
      | - /app.js
```

Dans cet exemple de structure, nous avons :

- `app.js` : le fichier qui sera intégré dans toute l'application. Il se basera sur les différents `components` pour initialiser les différentes actions
- `notification.js`, `alert.js` : ce sont des composants du système. Il sont censé gérer une seule fonctionnalité et seront appelé soit par `app.js` soit par des Javascript dédiés à certaines page. Le but étant de réduire la taille et la complexité du fichier `app.js`. Si une fonctionnalité a vocation à se retrouver sur plusieurs page, alors, cette fonctionnalité doit être mise en place de un `component` et inclus via le fichier `app.js`
- `views/homepage/dashboard.js`, `views/login/form.js` : ces fichiers sont dédiés à une page spéfique. Ils seront inclus dans la page grâce aux `entries` Webpack. Ce sont des fichiers contenant des fonctionnalités spécifiques à une page.

CSS

L'ensemble de notre code CSS et en fait généré à partir de fichier SASS. C'est Webpack qui se chargera de convertir les fichiers en fichiers CSS.

Il est déconseillé d'utiliser les id pour appliquer un style. Cela rendant plus difficile la surcharge.

L'ensemble des noms de classe CSS doivent être écrits en anlais, en minuscule et si c'est un mot composé spéraré par un tiret `-`. Par exemple : `primary-block`

Il est conseillé de décrire une fonctionnalité plus qu'un comportement : il est préférable d'appliquer une classe `construction-site-title` à un élément que : `padding-30 big-text`

La structure de nos fichiers de style est similaire à ce que nous faisons pour les fichiers Javascript.

Nous distinguons les composants (mise en forme générale) des spécificités (design spécifique à une page)

Pour les composants, nous allons les découper par type de fonctionnalité, par exemple :

- la gestion des tableaux
- la gestion des formulaires
- la gestion des polices de caractères
- la gestion des notifications
- ...

Tout cela sera ensuite intégré à un fichier `app.sass` .

Pour le design spécifique à une page, nous placerons le code dans un fichier SASS à part que nous incluerons via Webpack et ses `entries`

Exemple de structure pour le CSS :

```
- /assets
  | - /css
      | - /components
          | - /_notification.sass
          | - /_alert.sass
      | - /views
          | - /homepage
              | - /dashboard.sass
          | - /login
              | - /form.sass
      | - /app.sass
```

Git

- le dépôt principal s'appelle l' `upstream`
- chaque développeur possède un fork de l `upstream` appelé `origin`
- chaque fonctionnalité doit être réalisée sur une branche à part et partir de l' `upstream/master`
- le nom des branches doit suivre la structure suivante :
    - `features/nom-de-la-fonctionnalité` : en cas de nouvelle fonctionnalité
    - `fix/nom-du-fix` : en cas de correction de bug
- chaque branche est poussée sur l' `origin` du développeur et un merge request est fait sur la branche `upstream/master`

Gestion des assets

Structure de Symfony

Utilisation de Doctrine

Gestion des requêtes

Toutes les requêtes doivent être définies dans un repository. Il est interdit de faire une requête en dehors des repositories.

Tous les paramètres doivent être échappés, il est donc interdit de faire des concaténations à l'intérieur d'une clause `where` par exemple, il faut passer par les `setParameter` ou `setParameters` .

ENUM

Doctrine ne gère pas nativement les types `ENUM`, mais il est possible de les déclarer via une classe spécifique et un paramétrage. Lorsque nous avons une donnée qui doit être de type `ENUM`, il faut donc la déclarer comme telle.

Déclaration d'un `ENUM` :

```
- /config
  | - /packages
      | - doctrine.yaml
- /src
  | - /DBAL
      | - /Type
          | - /Enum
              | - ConstructionSiteStateType.php
              | - EnumType.php
  | - /Entity
      | - ConstructionSite.php
```

```yaml
# /config/packages/doctrine.yaml
doctrine:
    dbal:
        mapping_types:
            enum: string
        types:
            construction_site_state: App\DBAL\Type\Enum\ConstructionSiteStateType
```

```php
<?php
// /src/DBAL/Type/Enum/EnumType

declare(strict_types=1);

namespace App\DBAL\Type\Enum;

use Doctrine\DBAL\Platforms\AbstractPlatform;
use Doctrine\DBAL\Types\Type;

abstract class EnumType extends Type
{
    protected $name;
    protected $values = [];

    public function getSQLDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        $values = array_map(function ($val) { return "'".$val."'"; }, $this->values);

        return 'ENUM('.implode(', ', $values).')';
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (null === $value) {
            return null;
        }

        if (!in_array($value, $this->values)) {
            throw new \InvalidArgumentException("Invalid '".$this->name."' value.");
        }

        return $value;
    }

    public function getName()
    {
        return $this->name;
    }

    public function requiresSQLCommentHint(AbstractPlatform $platform)
    {
        return true;
    }
}
```

```php
<?php
// App/DBAL/Enum/Type/ConstructionSiteStateType.php
<?php

declare(strict_types=1);

namespace App\DBAL\Type\Enum;

class ConstructionSiteStateType extends EnumType
{
    const OPENED = 'opened';
    const CLOSED = 'closed';

    const AVAILABLES = [
        self::OPENED,
        self::CLOSED,
    ];

    protected $name = 'construction_site_state';
    protected $values = self::AVAILABLES;
}
```

```php
<?php
// App/Entity/ConstructionSite.php
// ...
class ConstructionSite
{
    // ...
    /**
     * @ORM\Column(type="construction_site_state")
     */
    private $state
    // ...
}
```

Suivi des ajouts et modifications

Pour chaque entitée, nous devons pouvoir connaître la date d'ajout et de mise à jour.

Pour cela, nous mettons en place un listener, via le bundle Gedmo, permettant davoir une date de création et une date de dernière modification.

Controllers

Les contrôleurs doivent contenir un minimum de code. Ils doivent uniquement contenir :

- la récupération des données
- l'appel à une classe extérieur permettant d'exécuter le code métier que nous plaçons dans des classes `Manager`
- l'envoi de la réponse HTTP

Par exemple, l'enregistrement d'une données ou la construction d'un formulaire vont se faire dans des classes spécifique (model ou manager pour l'enregistrement, FormType pour la construction d'un formulaire)

Manager

Les classes dites `Manager` sont les points d'entrée pour le code métier. Elle sont situées dans le namespace `App\Manager`

C'est à travers elle que nous allons appeler le code métier (qui peut-être dispatché dans différentes classes et namespace)

Formulaire

Chaque formulaire doit être défini dans une classe de type FormType.

Dans l'idéal, les validations doivent être mises dans la classe associée au formulaire via des annotations.

Si le formulaire ne correspond pas exactement à une entité, alors une classe sera créée spécifiquement pour correspondre au formulaire.

Event

Si une action peut déclencher plusieurs autre action, comme par exemple : l'enregistrement d'un utilisateur, déclenche l'envoi d'un mail à l'utilisateur et l'envoi d'un mail

à un administrateur. Alors, ces actions vont se déclencher via un `Event`.

On créera ensuite les `subscriber` nécessaire pour le traitement post-enregstrement.

Commande

Toutes les commandes créées doivent inscrire tant que la sortie standard (ou sortie d'erreur) que dans des fichiers afin de savoir ce que la commande a fait exactement (que ce soit en cours d'exécution ou pour rechercher d'ancienne exécution de commande)

Un logger spécifique devra étre dédiée aux différentes commandes.

Consumer

Dans le cas de l'utilisation de déport de traitement via RabbitMQ, les consumers doivent être le plus simple possible :

- Récupération des données
- Envoi des données à une commande

Les consumers s'exécutant durant plusieurs heures / jours, il est important de ne mettre aucun code faisant appel à une connexion (base de données par exemple) car celle-ci peut-être coupé et générer des erreurs dans les consumers.

Pour l'envoi des mails, nous passeons par des consumers : voir le composant `messenger` de Symfony

Traduction

L'ensemble de l'application doit pouvoir être traduite. Il est donc essentiel que l'ensemble des chaînes soient traduisibles.

Pour cela, nous utilisons des `clés de traduction`. Les chaînes doivent être mise dans le domaine correspondant au contexte.

Par exemple, les chaînes relatives à un formulaire vont se retrouver dans le domaine `form`, les chaînes destinées aux mails, dans le domaine `mail`, les chaînes destinées à être afficher dans un flashbag dans le domaine `flagbag`

De la même manière, pour un soucis de lisiblité de code, nous regrouperons les chaînes par fonctionnalité, par exemple :

````yml
# translations/form.fr.yml

construction_site:
    label: Titre
    status: Statut
    shop:
        title: Nom du magasin
```

Ainsi, les clés de traduction à utiliser seront : `construction_site.label`