Dossier Projet

**Philippe ARNAL**

**Application EasyFit**

***Outil de gestion de salles de sport***

**URL du site :**  <https://ecf-studi-easyfit.herokuapp.com/>

Pour se connecter en tant que :

**administrateur** login : administrateur@exemple.com Mdp : administrateur

**partenaire** login : partenaire@exemple.com Mdp : partenaire

**structure** login : structure@exemple.com Mdp : structure

**Table des matières**

[1 Liste des compétences couvertes par le projet 3](#_Toc128414634)

[2 Résumé du projet 4](#_Toc128414635)

[3 Spécifications fonctionnelles du projet 5](#_Toc128414636)

[4 Spécifications techniques du projet, 6](#_Toc128414637)

[4.1 liste des soft 6](#_Toc128414638)

[4.2 Choix des technologies 7](#_Toc128414639)

[5 Préparation du projet (compétence AT1.1) 8](#_Toc128414640)

[5.1 Diagramme de cas d’utilisation 8](#_Toc128414641)

[5.2 Méthode Merise 9](#_Toc128414642)

[5.3 Diagrammes de séquence 11](#_Toc128414643)

[5.4 Wireframe du projet 12](#_Toc128414644)

[5.5 Charte graphique 13](#_Toc128414645)

[6 Initialisation du projet dans Symfony (compétence AT2.1) 14](#_Toc128414646)

[6.1 Quelques mots sur le modèle MVC de Symfony : 15](#_Toc128414647)

[6.2 Quelques mots sur github 15](#_Toc128414648)

[7 Déploiement 16](#_Toc128414649)

[8 Elaboration du projet 17](#_Toc128414650)

[8.1 Mise en forme du projet (compétence AT1.2) 17](#_Toc128414651)

[8.2 Création des entités (compétence AT2.2) 18](#_Toc128414652)

[8.3 Création de la base de données et des entités en SQL (compétences AT2.1 AT2.3) 19](#_Toc128414653)

[8.4 Création des fixtures 20](#_Toc128414654)

[8.5 Création du formulaire de connexion (compétences AT1.4, AT2.4) 20](#_Toc128414655)

[8.6 Création de la page d’accueil 21](#_Toc128414656)

[8.7 Création des pages et de leurs contrôleurs (compétences AT1.4, AT2.4) 21](#_Toc128414657)

[8.8 Ajout d’un utilisateur (compétence AT2.2) 21](#_Toc128414658)

[8.9 Bouton de détail 21](#_Toc128414659)

[9 Fonctions dynamiques (compétences AT1.3 AT2.2 AT2.3) 22](#_Toc128414660)

[9.1 Activation et désactivation en un click 22](#_Toc128414661)

[9.2 Recherche dynamique avec suggestion 22](#_Toc128414662)

[9.3 Suppression d’un utilisateur via une modale 22](#_Toc128414663)

[10 Référencement naturel  (SEO) 23](#_Toc128414664)

[11 Tests 24](#_Toc128414665)

[13 Veille effectuée sur les vulnérabilités de sécurité (compétence AT2.3) 25](#_Toc128414666)

[14 Situation de travail ayant nécessité une recherche à partir de site anglophone 27](#_Toc128414667)

[14.1 Extrait du site anglophone 27](#_Toc128414668)

[14.2 Traduction en français de l’extrait 28](#_Toc128414669)

[15 Conclusion 29](#_Toc128414670)

[16 ANNEXES 30](#_Toc128414671)

# Liste des compétences couvertes par le projet

AT1 - Développer la partie front-end d’une application web ou web mobile en intégrant les recommandations de sécurité.

AT1.1 - Maquetter une application

AT1.2 – Réaliser une interface utilisateur web statique et adaptable

AT1.3 - Développer une interface utilisateur web dynamique

AT1.4 - Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce

AT2 - Développer la partie back-end d’une application web ou web mobile en intégrant les recommandations de sécurité.

AT2.1 - Créer une base de données

AT2.2 - Développer les composants d’accès aux données

AT2.3 - Développer la partie back-end d’une application web ou web mobile

AT2.4 - Elaborer et mettre en œuvre des composants dans une application de gestion de contenu ou e-commerce

Le tableau ci-dessous permet de trouver rapidement les chapitres dans lesquels ces compétences sont particulièrement détaillées.

|  |  |
| --- | --- |
| Compétence | Chapitre |
| AT1.1 | §5 |
| AT1.2 | §8.1 |
| AT1.3 | §9 |
| AT1.4 | §8.4 §8.6 |
| AT2.1 | §6 §8.3 |
| AT2.2 | §8.2 §8.7 §9 |
| AT2.3 | §8.3 §9 §10 |
| AT2.4 | §8.4 §8.6 |

# Résumé du projet

Dans le cadre de de ma formation Graduate Développeur mobile Android chez STUDI, j’ai eu à concevoir une application web complète.

L’application consiste en un outil de gestion de salles de sport.

L’interface permet :

* à une équipe d’administration de gérer :
  + des administrateurs,
  + des franchises avec leurs options globales,
  + des structures avec leurs options,
* à une franchise de lister ses structures et ses options globales,
* à une structure de lister ses options.

Les fonctions principales sont les suivantes :

* En tant qu’administrateur l’utilisateur peut :
  + créer, lister, éditer et supprimer les informations de la base de données,
  + effectuer des recherches et des tris dynamiques sans que les pages soient rechargées,
  + activer, désactiver un utilisateur en un click,
* Le système doit:
  + gérer l’accès à certaines page en fonction des utilisateurs ou refuser l’accès si l’utilisateur est désactivé.
  + demander confirmation pour chaque suppression ou désactivation.
  + envoyer des Emails de confirmation à chaque création ou modification de compte.

Après avoir maquetté l’application pour que l’interface soit responsive et mobile-first, j’ai développé l’outil en toute autonomie, en veillant à intégrer les recommandations de sécurité.

J’ai utilisé pour cela les technos : Symfony, SymfonyCLI, Composer, Doctrine, Stimulus, WebPack Encore, Stimulus, Twigg, Mailer, Bootstrap, Laragon, npm, git et Heroku.

# Spécifications fonctionnelles du projet

**SF1 :** Gérer les partenaires => *L'équipe technique (Administrateur)*

* Ajouter un partenaire : Nom + Adresse Email
* Créer un compte lié : Adresse Email + MdP sécurisé
* Activer un partenaire
* Désactiver un partenaire (avec confirmation)

**SF2** : Gérer les structures => *L'équipe technique (Administrateur)*

* Ajouter une ou plusieurs structures à un partenaire : Adresse postale + Adresse Email gérant
* Créer un compte lié : Adresse Email gérant + MdP sécurisé
* Activer une structure en 1 click
* Désactiver une structure en 1 click (avec confirmation)

**SF3** : Gérer les options => *L'équipe technique (Administrateur)*

* Activer/désactiver une option de fonctionnalité globales sur la page partenaire
* Activer/désactiver une option de fonctionnalité structure
* Par défaut, les option structure sont celles des partenaires

**SF4** : Se connecter => *L'équipe technique (Administrateur), le partenaire, la structure*

* Si le partenaire est actif, il peut se connecter avec Adresse Email partenaire + MdP sécurisé et voir l'ensemble de ses structures avec leurs options et ses options globales.
* Si la structure est active, elle peut se connecter avec Adresse Email gérant + MdP sécurisé et voir ses options.

**SF5** : La notification par Email => *L'équipe technique (Administrateur), le partenaire, la structure*

3 types d'Email automatiques sont envoyés par l'application :

* Lorsqu'un partenaire a été créé, envoie des informations permettant au partenaire de se connecter. Demander un nouveau MdP à la première connexion.
* Lorsqu'une structure a été créé :
  + envoie des informations permettant au gérant de se connecter.
  + envoie au partenaire l'information de la création de la nouvelle structure.
* Lorsqu'une ou plusieurs options ont été changées :
  + envoie à la structure et au partenaire si option structure
  + envoie au partenaire si option globale

**SF6** : La confirmation de sécurité => *L'équipe technique (Administrateur)*

* A chaque modification ou suppression, l'application affiche un message demandant la confirmation de l'utilisateur. S'il ne valide pas alors l'action est annulée.

**SF7** : La recherche dynamique => *L'équipe technique (Administrateur), le partenaire, la structure*

* Une barre de recherche permettra à l'utilisateur de trouver rapidement un partenaire en tapant les premières lettres de son nom. Sans recharge de la page.
* Trier la liste des partenaires ou structures afin de présenter uniquement les actifs ou désactivés. Sans recharge de la page.

# Spécifications techniques du projet,

## liste des soft

**Editeur:** VSCode avec

* Symfony 6
* Symfony CLI 5
* Composer (php) ou Yarn (javascript)
* WebPack Encore
* Doctrine ORM 2
* Moteur de template Twig
* Mailer
* Faker
* sensiolabs/security-checker
* …

**FrontEnd :**

* HTML5
* SCCS
* Bootstrap 5.2
* JavaScript
* jQuery
* Stimulus + stimulus-use

**BackEnd :**

* Laragon :
  + Apache 2.4,
  + Nginx,
  + MySQL 5.7,
  + PHP 8.1.10,
  + Redis,
  + Memcached,
  + Node.js 16.17.0

**Deployment :**

* Heroku via GitHub

## Choix des technologies

**Bootstrap** :

Le framework css bootstrap permet de gagner du temps car il rassemble plusieurs composants prêts à l’emploi aussi bien pour la mise en forme css que dans l’animation et l’interactivité du contenu javascript. Des formulaires, des boutons, des menus de navigation…

Les avantages : un site web responsive, des thèmes utilisables directement, compatibilité avec de nombreux navigateurs.

Les variables sass permettent de modifier rapidement les thèmes.

Bootstrap me permet de me concentrer sur la partie fonctionnelle car je n’ai pas beaucoup d’aisance sur la partie graphique.

**Symfony :**

Le projet a été réalisé avec Symfony 6

* pour travailler sur un framwork MVC. Dans symphony :
  + Model = Repository
  + View = Template
  + Controller = Controller
* pour sa facilité de mise en œuvre et sa documentation parfaite
* pour son intégration de la sécurité avec le security bundle
* pour travailler sur les dernières versions, le monde de l’informatique évoluant rapidement.

**ORM Doctrine** :

Grace à Doctrine on peut gérer aisément nos tables grâce aux trois classes PHP suivantes :

* Création des Entity (tables)
* Utilisation du manager de Doctrine pour gérer les lignes de la base de données en surveillant les entrées pour éviter les injections SQL.
* Utilisation du repository pour faire les requêtes.

**Laragon :**

alternative à XAMPP ou WAMP, il m’a paru plus facile d’accès tout en étant extensible.

**Stimulus :**

Stimulus est une bibliothèque Javascript.

les contrôleurs Stimulus offrent une structure claire et simple pour la logique de comportement et peuvent améliorer les performances des pages web.

# Préparation du projet (compétence AT1.1)

La création du code est grandement facilitée avec la création en amont des documents ci-dessous :

* + Planning avec Trello (non détaillée dans ce rapport),
  + Liste des fonctionnalités ([§3](#_Spécifications_fonctionnelles_du)),
  + Diagramme de cas d’utilisation,
  + Dictionnaire des données (non détaillée dans ce rapport),
  + Les modèles de données issus de la méthode Merise (MCD, MLD, MPD)
  + Diagrammes de séquence,
  + Wireframe Mobile,
  + Wireframe Desktop,
  + Charte graphique,
  + Mock-up.

## Diagramme de cas d’utilisation



## Méthode Merise

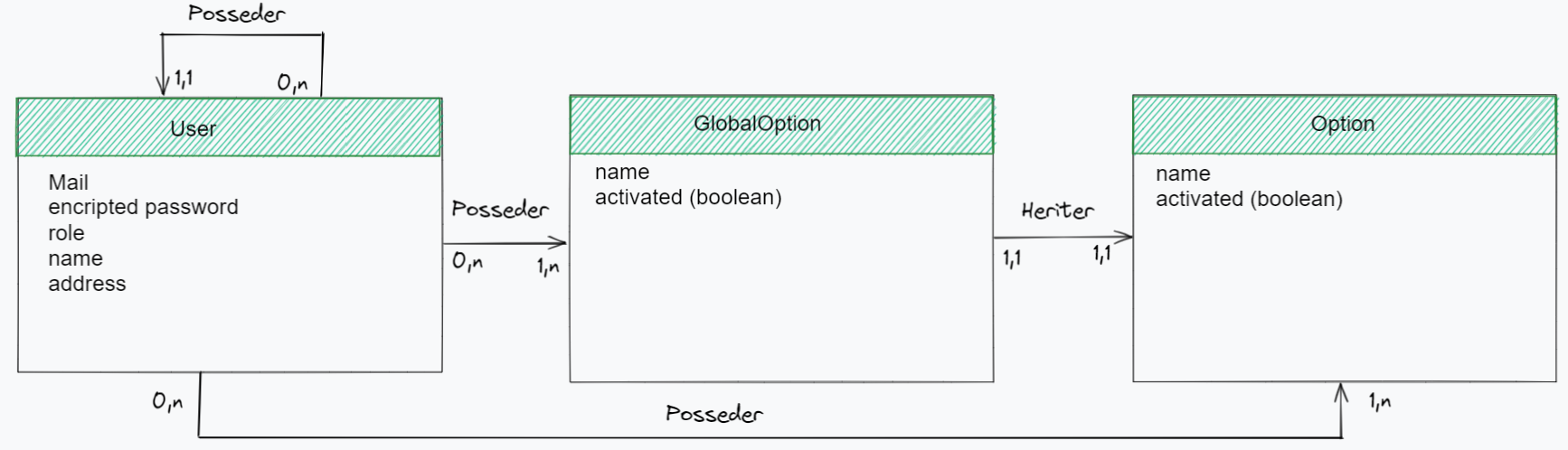
La méthode Merise une méthode d’analyse, de conception et de réalisation de système d’information basée sur un modèle Entité-Association.

Je compliqué mon code en voulant limiter le nombre d’entités. J’ai créé une seule entité User unique pour les trois utilisateurs : administrateur, partenaire et structure en leur donnant un rôle différent. Je ne recommencerai pas, la gestion des rôles a été beaucoup plus compliquée ensuite.

L’entité User est donc enfant d’elle-même car une structure est enfant d’un partenaire.

Lorsqu’un partenaire est supprimé, ses structures sont supprimées en cascade.

MCD : Modèle conceptuel de donnée

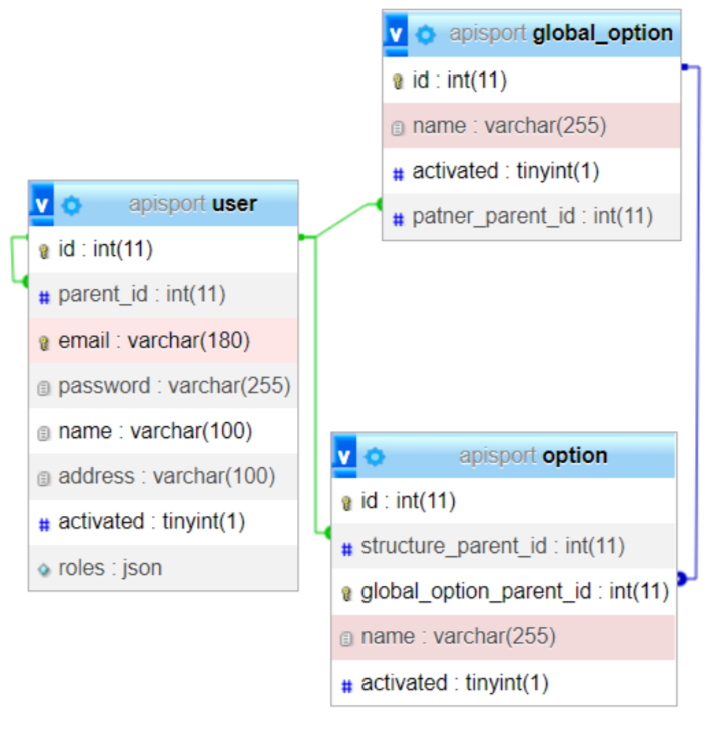


MLD : Modèle Logique de données

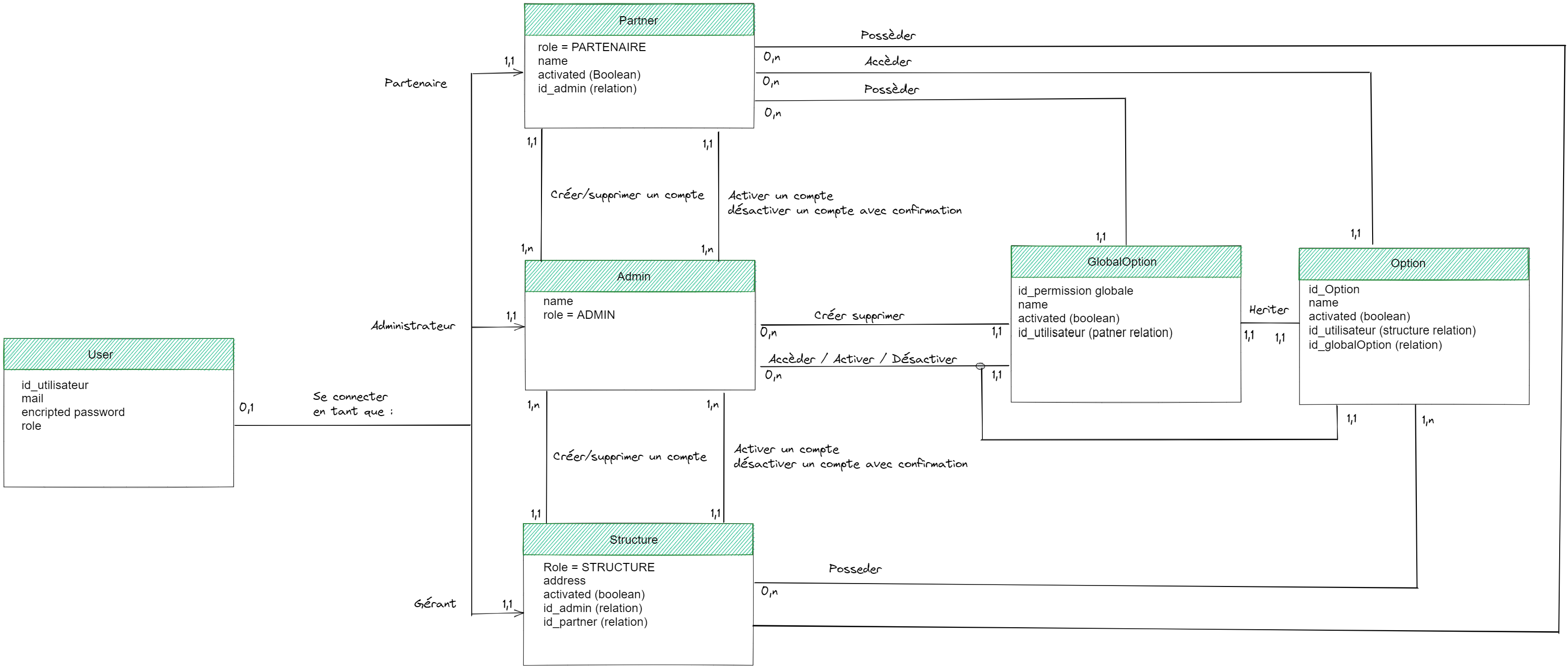
Une image contenant texte

Description générée automatiquement

MPD : Modèle Physique de données



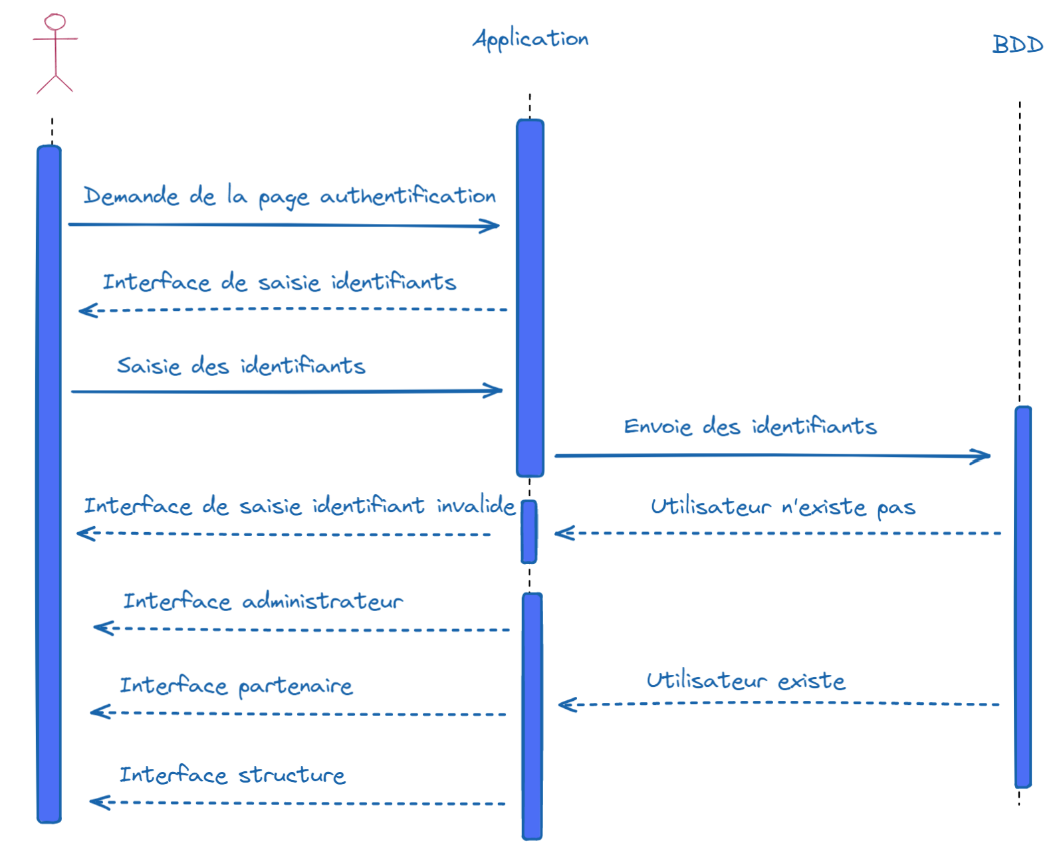
Modèle Conceptuel de données détaillé avec les rôles utilisateurs :

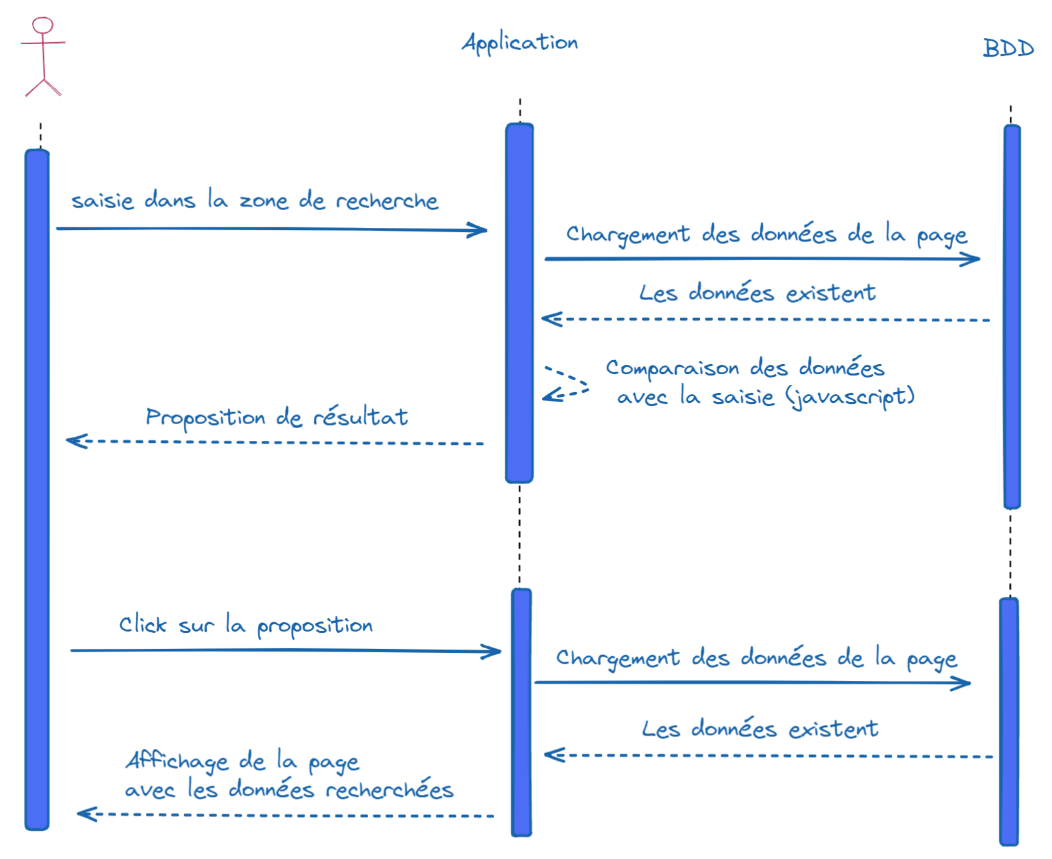


## Diagrammes de séquence

Deux diagrammes de séquence sont présentés ici à titre d’exemple.

Diagramme de séquence authentification

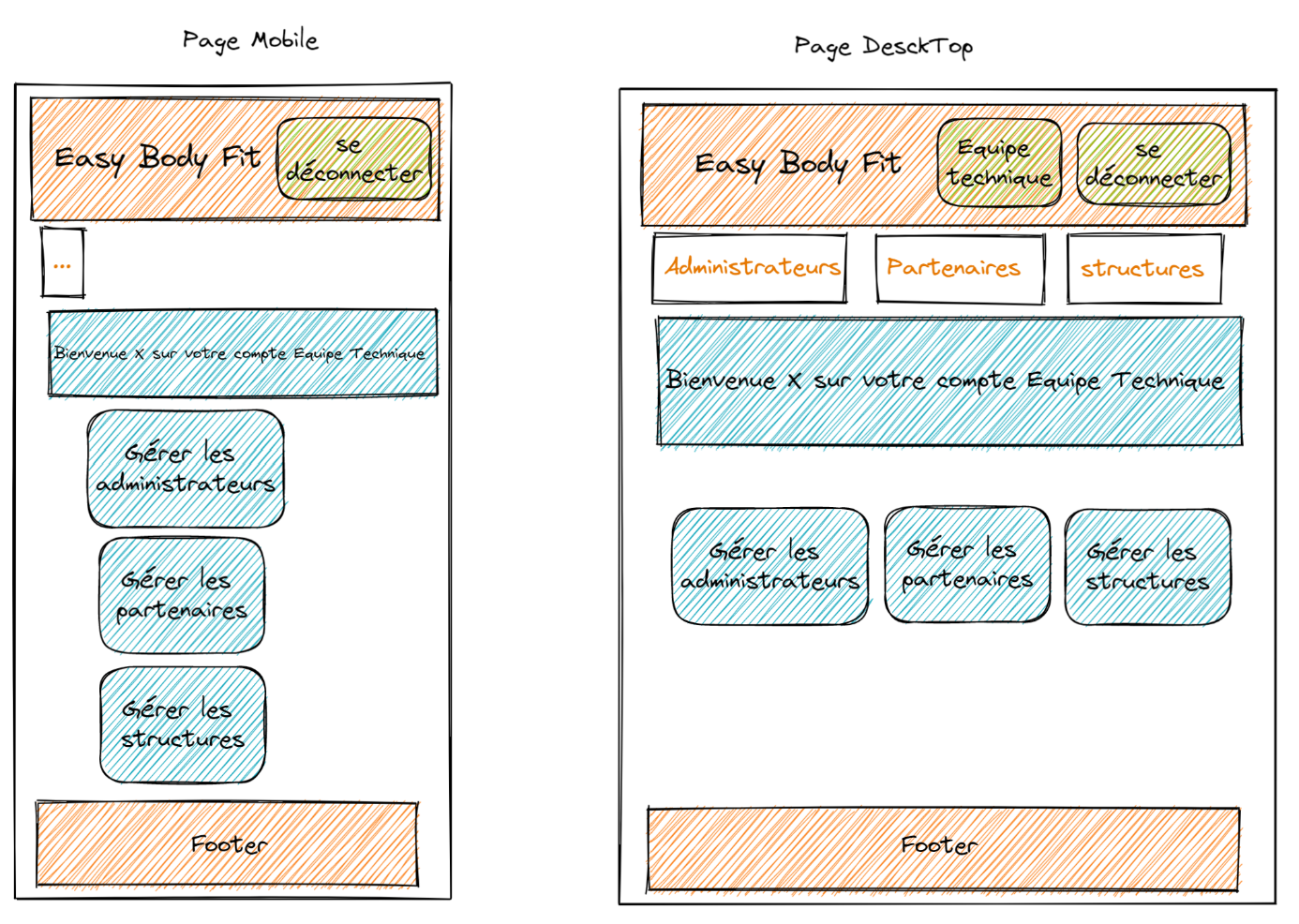


Diagramme de séquence Recherche partenaire ou structure

## Wireframe du projet

Le wireframe général est présenté ci-dessous pour démontrer la philosophie responsive et mobile first de l’application.

Un wireframe plus détaillé montrant la séquence des ouvertures de page en desktop est présenté en [Annexe 1.](#_ANNEXE_1_:)



## Charte graphique

La charte graphique de l’espace d’administration est un rappel de la charte graphique de la marque.

Celle-ci est composée :

* de couleurs évoquant le dynamisme et la sportivité à base d'orange et de gris,
* d’une fonte lisible et conviviale

**Une image contenant texte

Description générée automatiquement**

# Initialisation du projet dans Symfony (compétence AT2.1)

Une part importante du projet est la configuration de Symfony et de la base de données.

* Création du nouveau projet : ***symfony new my\_project --webapp***
* Composer permet de gérer les composants utiles à mon application ( library) et de les garder à jour, ils sont alors visible dans le fichier composer.json.
* Yarn permet de garder à jour les composants javascript.
* Installation de Saas, Bootstrap, Javascript, JQuery, Stimulus.
* Avec le WebPack Encore, les fichiers CSS, SCSS et JS sont contenus dans un répertoire Assets. WebPack va ensuite les compiler dans le repertoire Public Build pour l'utilisation en production.

Ces fichiers Build seront "lus" dans la page principale base.html.twig grâce aux fonctions d'affichage Twig : *encore\_entry\_link\_tags()* et *encore\_entry\_script\_tags()*.

* Installation de Laragon, Node.js, Symfony CLI.
* Lancement du serveur local et écoute du build avec Watch,
* Création de la base de données,
* Configuration du fichier .env.local avec la référence à la base de données,
* Supprimer l’APP\_SECRET et la référence de base de donnée locale sur le fichier .env destiné à l’environnement de production.
* Création du dépôt GitHub et initialisation du repository.
* Créer le lien entre Github et l’éditeur de code pour faciliter les commit et push.

## Quelques mots sur le modèle MVC de Symfony :

Le Model-View-Controller (MVC) est un modèle architectural utilisé dans symfony comme dans d’autres framework pour organiser le code d'une application web. Il sépare les données (modèle), l'interface utilisateur (vue) et la logique de contrôle (contrôleur) en trois couches distinctes. Ces couches permettent de rendre le code plus facile à maintenir et à étendre car les couches sont indépendantes.

Dans Symfony, les routes de l'application sont gérées par des contrôleurs, qui décident quoi afficher à l'utilisateur en fonction des données reçues. Les données sont gérées par les modèles, qui peuvent être des entités Doctrine ou des objets métier. Les modèles sont généralement utilisés pour interagir avec la base de données et fournir les données nécessaires pour les vues. Les vues sont les templates Twig qui affichent les données à l'utilisateur.

En résumé, dans Symfony, les routes sont gérées par des contrôleurs qui utilisent les données gérées par les modèles pour afficher les vues à l'utilisateur.

Par exemple pour le formulaire d’authentification :

1. Le contrôleur récupère la requête
2. Le contrôleur va venir transmettre les infos de l’utilisateur à la couche modèle
3. Le modèle vérifie que les données existent dans la BDD
4. Le modèle renvoie les données de l’utilisateur en question au contrôleur
5. le contrôleur transmet ces informations à la vue
6. la vue affiche à l’utilisateur

## Quelques mots sur github

Lors de la phase de développement, Github facilite particulièrement la gestion des versions de notre projet. Un enregistrement régulier de son travail permet de mieux comprendre son évolution et de revenir en arrière en cas de problème.

Pour un nouveau projet :

le dépôt git créé avec la commande ***git init*** à la racine de notre projet,

Une nouvelle version de nos fichiers est créée avec les commandes ***git add*** et ***git commit***

Le dépôt local est synchronisé ensuite avec le dépôt distant github avec la commande ***git remote add***. On peut ensuite envoyés les commits vers le dépôts distant avec un ***git push***.

***git push origin X*** permet de spécifier la branche X sur laquelle on veut ajouter les commits.

En travaillant seul il est facile de créer une nouvelle branche pour de nouvelle fonction puis faire un ***git rebase*** lorsque l’on veut appliquer les commits de la branche en question sur la branche master (ou main).

Lorsque l’on travaille à plusieurs, il est préférable d’effectuer au préalable un ***git fetch*** pour récupérer les modifications du dépôt distant, réparer les conflits éventuels***,*** puis fusionner les branches avec un ***git merge***. La commande ***git pull*** permet de faire les deux actions en même temps.

# Déploiement

Le déploiement du projet est effectué le plus tôt possible pour corriger les pertes d’information au fur et à mesure.

Les étapes sont résumées ici :

* Création du compte
* Installation de Heroku CLI.
* Création de l’application Heroku.
* Configuration de la variable d’environnement à PROD et la DATABASE\_URL.
* Ajout de l’add-ons pour la base de données.
* Création et édition du fichier Procfile à la racine du projet.
* Synchronisation avec Github pour l’api soit mise à jour à chaque push.

# Elaboration du projet

## Mise en forme du projet (compétence AT1.2)

La mise en forme et le responsive du projet est principalement effectuée avec Bootstrap.

Sur les petits écrans, certains détails de la bannière comme le slogan, le nom et le rôle de l’utilisateur connecté ne sont pas afficher et la barre de navigation est réduite.



Une image contenant texte

Description générée automatiquement

Pour compléter, j’ai ajouté quelques media queries comme dans les exemples ci-dessous pour limiter la taille du texte et supprimer les sous-titre et renseignements sur l’utilisateur connecté lors de lecture sur mobile.

/\* Le sous-titre et les renseignements de l'utilisateur connecté disparaissent pour écrans xs (<768px) \*/

@media screen and (max-width:768px) {.roleDisplay {display: none;}}

/\* font-size des tableaux réduite pour écrans xs et sm (<992px) \*/

@media screen and (max-width:992px) { .tableDisplay { font-size: 0.7rem;}}

Nota : Le développement **mobile first** voudrait que seulement les grands formats soient sujet à des média queries avec *min-with*. Je ne manquerai pas de me tenir à cette notion lors d’un prochain projet.

L’appli est basée sur un espace d’administration de données avec potentiellement de nombreuses données. Cette administration se fait en général sur un pc. Il m’a donc semblé pertinent de garder la présentation des données sous forme de tableau.

Néanmoins, pour assurer la lecture sur des petits écrans, j’ai inséré mes commandes de tableaux dans une div ayant la classe bootstrap « table-responsive ». Ceci permet de scroller facilement d’un coup de pouce le tableau pour atteindre les colonnes non visibles.





<div class="table-responsive">

<table class="mt-5 table tableDisplay">

…

</table>

</div>

## Création des entités (compétence AT2.2)

La gestion des entités est facilitée par l’utilisation de l’ORM Doctrine.

Grace à Doctrine, on peut gérer nos tables grâces aux trois classes PHP suivantes :

* Création des Entity (tables)
* Utilisation du manager de Doctrine pour gérer les lignes de la base de données en surveillant les entrées pour éviter les injections SQL.
* Utilisation du repository pour faire les requêtes.

Il faut ensuite migrer les données sur la base de données grâce aux fichiers de migration.

### Création de l’entité User

Avec l’utilisation des rôles de l’utilisateur l’entité User regroupe dans ce projet les administrateurs les partenaires et les structures.

L’entité User est donc enfant d’elle-même car les structures sont des entités enfants des partenaires.

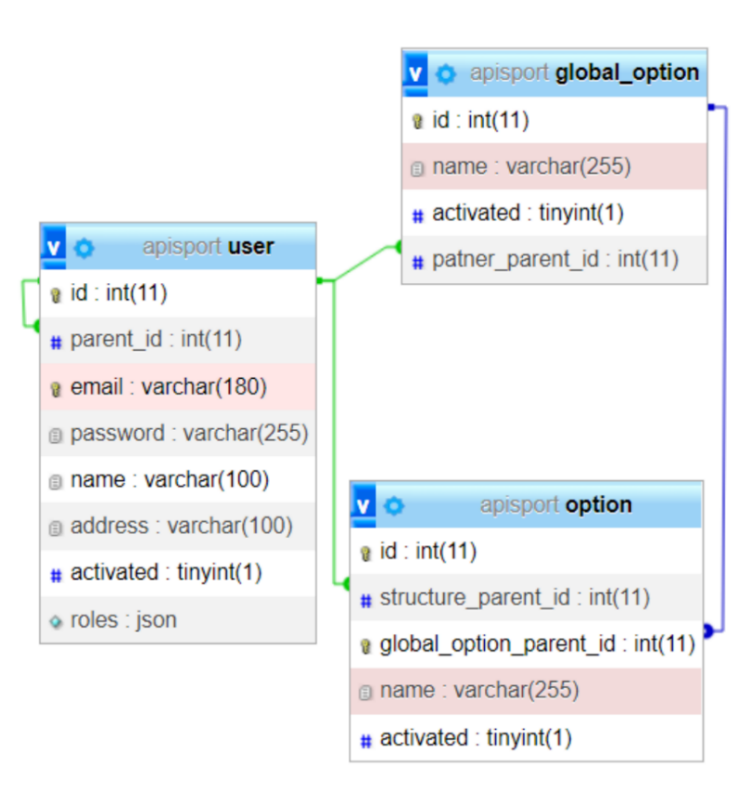
### Création des entités GlobalOption et Option

L’entité GlobalOption est enfant de partenaire.

Lorsqu’un partenaire est supprimé ses options globales sont supprimées en cascade.

L’entité Option est enfant de GlobalOption (pour que les options soient issues des options globales) et de Structure.

Lorsqu’une Structure est supprimée ses options sont supprimées en cascade.



## Création de la base de données et des entités en SQL (compétences AT2.1 AT2.3)

Doctrine automatise la création de la base de données, des tables et les requêtes associées. Voici les commandes SQL qu’il faudrait exécuter pour notre projet.

### Création de la base de données apisport

CREATE DATABASE apisport

DEFAULT CHARACTER SET = 'utf8mb4';

USE apisport;

### Création des tables User, GlobalOption et Option

CREATE TABLE user (

id INT PRIMARY KEY AUTO\_INCREMENT NOT NULL,

parent\_id INT DEFAULT NULL,

email VARCHAR(180) NOT NULL,

password VARCHAR(255) NOT NULL,

name VARCHAR(100) DEFAULT NULL,

address VARCHAR(100) DEFAULT NULL,

activated BOOLEAN DEFAULT FALSE,

roles JSON NOT NULL,

UNIQUE (email),

FOREIGN KEY (parent\_id) REFERENCES User(id) ON DELETE CASCADE

);

CREATE TABLE globalOption (

id INT PRIMARY KEY AUTO\_INCREMENT NOT NULL,

patner\_parent\_id INT DEFAULT NULL,

name VARCHAR (255) DEFAULT NULL,

activated BOOLEAN DEFAULT FALSE,

FOREIGN KEY (patner\_parent\_id) REFERENCES User(id) ON DELETE CASCADE

);

CREATE TABLE options (

id INT PRIMARY KEY AUTO\_INCREMENT NOT NULL,

structure\_parent\_id INT DEFAULT NULL,

global\_option\_parent\_id INT DEFAULT NULL,

name VARCHAR (255) DEFAULT NULL,

activated BOOLEAN DEFAULT FALSE,

FOREIGN KEY (structure\_parent\_id) REFERENCES User(id) ON DELETE CASCADE,

FOREIGN KEY (global\_option\_parent\_id) REFERENCES GlobalOption (id)

);

### Insertion de données dans la table

Par exemple, l’instruction pour insérer un utilisateur ayant le rôle administrateur dans la table sera :

INSERT INTO User (email,password,name,activated,roles) VALUES ('administrateur@exemple.com','administrateur','Administrateur', '1', '[“ROLE\_ADMINISTRATEUR”]’);

### Lecture des données

Par exemple, pour afficher le nom et le mail de tous les utilisateurs ayant le rôle administrateur :

SELECT name, email

FROM User

WHERE roles LIKE '["ROLE\_ADMINISTRATEUR"]'

ORDER BY name

## Création des fixtures

Afin de faciliter les tests et permettre la démonstration de l’application, j’ai insérer dans mon code les fixtures contenant les différents cas d’utilisation.

Pour cela, j’ai utilisé les bundle doctrine-fixture et fakerphp

J’ai insérer le chargement de ces fixtures dans les scripts du fichier composer.json pour que la base de données soient remise à jour à chacun des nouveaux push dans github.

Ce script sera nécessairement supprimé lors de la remise de l’application au client final et seul le compte administrateur indispensable à la première connexion subsistera.

## Création du formulaire de connexion (compétences AT1.4, AT2.4)

***symfony console make:auth***

* Mise à jour du fichier :
  + config/packages/security.yaml (avec le nom de l’Authenticator et une route pour le logout)
* Crée les fichiers suivants :
  + src/Controller/SecurityController.php (la route)
  + src/security/usersAuthentificator (qui gère toute l’authentification en sécurité)
* Retourne un passeport

## Création de la page d’accueil

Suivant l’utilisateur connecté, l’application donnera accès :

* Pour un administrateur => à tous les détails administrateur, partenaire ou structure et à leurs modifications. Les pages concernées ont toutes l’URL commençant par : /administrateur/ interdites aux autres utilisateurs.
* Pour un partenaire => au listing de ses structures et de ses options globales.
* Pour une structure => au listing de ses options uniquement.

Si l’utilisateur a été désactivé, un message d’alerte le lui signale et il n’a pas accès à ses données.

## Création des pages et de leurs contrôleurs (compétences AT1.4, AT2.4)

J’ai eu la mauvaise idée de n’utiliser qu’une seule page pour lister toutes les entités : administrateur, partenaire et structure. Elle devient compliquée à reprendre par un autre développeur. Il faudra corriger cela avec un peu de temps.

## Ajout d’un utilisateur (compétence AT2.2)

Au click du bouton Ajouter, un contrôleur Creation est appelé pour donner à l’utilisateur un accès à un formulaire. Il transportera l’Id de façon à coupler la structure avec le partenaire parent. Le formulaire sera différent suivant l’objet à créer (administrateur, partenaire ou structure).

En retour du formulaire, si l’enregistrement sur la base de données est bien effectué, le contrôleur retournera à l’affichage de la liste.

## Bouton de détail

Au click d’un bouton détail, le contrôleur Liste est appelé avec l’idParent pour afficher le détail de l’élément sélectionner :

Si le parent est une franchise => la liste des structures et des options globales

Si le parent est une Structure=> la liste des options uniquement

# Fonctions dynamiques (compétences AT1.3 AT2.2 AT2.3)

## Activation et désactivation en un click

Création d’un fichier javascript qui va écouter un input de type checkbox.

Au click, la fonction Javascript va :

* Récupérer la variable Id de la ligne en question.
* Activer le contrôleur Activation en injectant la variable Id.

Le contrôleur Activation va alors :

* Permuter la valeur booléenne de la propriété ‘activé’ de l’entité.
* L’enregistrer dans la base de données.
* Revenir à la page d’origine.

## Recherche dynamique avec suggestion

Le but ici est d’éviter que la page soit rechargée à chaque entrée dans le formulaire.

Pour cela, on écoute les données entrées dans le formulaire pour les envoyer dans un contrôleur stimulus. Ce contrôleur Stimulus va charger de manière asynchrone le contrôleur symfony que se chargera d’afficher une vue partielle avec une suggestion.

Si l’utilisateur click sur l’une des suggestions la page est alors rechargées pour afficher le résultat.

Le contrôleur stimulus écoute également un click à l’extérieur du formulaire de recherche, il demande alors à afficher « rien ». Ce qui stop l’affichage de la proposition.

L’explication de la recherche dynamique de suggestion est détaillée en [Annexe 2](#_ANNEXE_2_:)

## Suppression d’un utilisateur via une modale

Pour chaque suppression d’un utilisateur (administrateur, partenaire ou structure) une confirmation est demandée.

L’appui sur un bouton supprimer envoi au contrôleur stimulus l’URL du champ à supprimer.

Le contrôleur ouvre ensuite la vue modal.

Si appui sur le bouton confirmer le contrôleur de suppression est appelé pour supprimer le champ de la base de donnée et revenir à la page en cours.

# Référencement naturel  (SEO)

Voici les étapes principales que j’ai suivi pour optimiser mon code afin de rendre mon site plus visible dans les moteurs de recherche.

* Utiliser une balise titre par page, des méta-description appropriées avec des mots clés pertinents et des URL propres et significatives pour donner une idée claire aux utilisateurs de ce qu'ils peuvent attendre du contenu de la page.
* Le contenu du site doit être de qualité et original.
* Utiliser des balises header hiérarchisées avec un seul titre H1 pour permettre aux moteurs de recherche de comprendre la hiérarchie de l'information sur la page.
* Utiliser les balises HTML à bon escient : header, nav, main, section, article, aside, footer.
* Optimiser les images avec des balises alt décrivant les images pour les utilisateurs et les moteurs de recherche qui ne peuvent pas afficher les images.
* Utiliser des liens internes pour structurer le site : les liens internes permettent aux moteurs de recherche de comprendre la structure du site et de mieux comprendre les pages les plus importantes.
* Faire attention aux erreurs 404 et 301 qui sont renvoyés par les serveurs web lorsqu'une page n'est pas trouvée ou qu'elle a été déplacée. Vérifier les liens externes régulièrement.
* Ajouter un fichier Robot.txt pour gérer le trafic des robots d'exploration.
* Si le site est important ou contient beaucoup de contenu multimédia, créer un fichier sitemap. Il est possible de créer et mettre à jour automatiquement un fichier sitemap.xml avec Symfony.

# Tests

Les tests automatiques vont permettre de savoir rapidement si l’application n’a pas régressé lors par exemple, de l’ajout d’une nouvelle fonction ou à l’intégration des fonctions d’un autre développeur.

Il existe trois sortes de tests :

* Les tests unitaires
* Les tests fonctionnels
* Les tests End to End

Le bundle test-pack est ajouté au composer dans l’environnement de DEV. Il permet de créer des test sur mesure à l’aide de la commande ***make:test***.

Une base de données ***\_test*** image de la base de données existante avec ses tables et ses fixtures est créée.

Un .env.test.local est créé avec les accès à cette base de données.

Ensuite les tests sont lancés avec la commande ***php bin/phpunit.***

A titre d’exemple, j’ai inséré :

* un test unitaire pour vérifier que l’entrée d’un email non valide renvoie bien une erreur.
* Un test fonctionnel pour vérifier l’accès à la première page de connexion et l’envoi du formulaire et l’interdiction d’accès au pages administrateur si l’utilisateur est logué en tant que structure.

Le code de ces tests est présenté en [**Annexe**](#_ANNEXE_4_:) **3**

# Veille effectuée sur les vulnérabilités de sécurité (compétence AT2.3)

J’ai suivi les recommandations de l’Agence nationale de la sécurité des systèmes d'information ANSII : <https://www.ssi.gouv.fr/>

* + Bonnes pratiques : <https://www.ssi.gouv.fr/uploads/2014/10/anssi-catalogue-guides-notes-techniques.pdf>
  + Sécuriser un site WEB : <https://www.ssi.gouv.fr/guide/recommandations-pour-la-securisation-des-sites-web/>

Voici les actions principales mise en œuvre durant mon projet pour limiter les failles de sécurité :

* .env copier en .env.local pour ne pas envoyer ses données sur Github et sur le serveur. Notamment l’APP\_SECRET.
* Gérer l’authentification et les autorisations des utilisateurs à l’aide du composant dédié de Symfony : **security-bundle**
* Gérer l’authentification de manière sécurisée : **Hacher les passwords** et imposer un mot de passe suffisamment long. Ceci afin de protéger le site de l’usurpation de l’identité d’un utilisateur ou mieux d’un administrateur qui permettra par la suite l’accès à une plus grande surface d’attaque.
* **Limiter les privilèges** d’accès aux pages via l’URL à l’aide du fichier config\packages\**security.yaml** :

access\_control:

- { path: ^/administrateur, roles: ROLE\_ADMINISTRATEUR }

- { path: ^/partenaire, roles: ROLE\_PARTENAIRE }

- { path: ^/structure, roles: ROLE\_STRUCTURE }

role\_hierarchy:

ROLE\_ADMINISTRATEUR: ROLE\_PARTENAIRE

ROLE\_PARTENAIRE: ROLE\_STRUCTURE

* Valider les entrées utilisateur. Pour cela, j’ai veillé à insérer dans les formulaires des contraintes sur les entrées avec des expressions régulières pour éviter les **injections SQL** indésirables. J’ai utilisé pour cela les requêtes préparées du manager de l’ORM Doctrine.
* Se protéger contre les attaques Cross-Site Request Forgery (**CSRF**) en mettant en place des jetons anti-CSRF pour les formulaires et de les vérifier avant de traiter les données du formulaire. Un jeton est également ajouté pour la demande de suppression d’un utilisateur.
* Vérifier les données que je récupère dans ma base de données pour éviter les **failles XSS**. L'incorporation de données extérieures est protégée contre les attaques de type injections HTML (Cross-Site Scripting XSS) principalement par la librairie sqlite3 et ses fonctions dédiées.
* La suppression des enregistrements dans la base de données se fait avec la méthode POST moins vulnérable que la méthode GET.
* Configuration de l’infrastructure d’hébergement. Vérifier que l’application est protégée par la mise en place du protocole **HTTPS**.
* Vérification des vulnérabilités de sécurité connues dans les dépendances : *composer require sensiolabs/security-checker* + *Symfony security:check*
* Activer les outils proposés par github pour la détection de faille de sécurité et de divulgation de clefs.

J’ai effectué une analyse de mon site avec l’outil de sécurité web OWASP ZAP et je me suis rendu compte que mon site présentait une faille CSRF au niveau du formulaire de recherche rapide d’un utilisateur. Même si la page de ce formulaire n’est accessible que par un administrateur, il faudra que j’insère un jeton au niveau de ce formulaire.

# Situation de travail ayant nécessité une recherche à partir de site anglophone

La sécurité d’une application venant avant tout de la vérification de l’utilisateur connecté, je me suis penché sur la façon dont Symfony enregistrait l’utilisateur et comment cet enregistrement pouvait être fiabilisé.

Pour cela, je me suis rendu sur la page de la documentation Symfony :

<https://symfony.com/doc/3.3/security/entity_provider.html>

## Extrait du site anglophone

**Understanding serialize and how a User is Saved in the Session**

If you're curious about the importance of the serialize() method inside the User class or how the User object is serialized or deserialized, then this section is for you. If not, feel free to skip this.

Once the user is logged in, the entire User object is serialized into the session. On the next request, the User object is deserialized. Then, the value of the id property is used to re-query for a fresh User object from the database. Finally, the fresh User object is compared to the deserialized User object to make sure that they represent the same user. For example, if the username on the 2 User objects doesn't match for some reason, then the user will be logged out for security reasons.

Even though this all happens automatically, there are a few important side-effects.

First, the Serializable interface and its serialize() and unserialize() methods have been added to allow the User class to be serialized to the session. This may or may not be needed depending on your setup, but it's probably a good idea. In theory, only the id needs to be serialized, because the refreshUser() method refreshes the user on each request by using the id (as explained above). This gives us a "fresh" User object.

But Symfony also uses the username, salt, and password to verify that the User has not changed between requests (it also calls your AdvancedUserInterface methods if you implement it). Failing to serialize these may cause you to be logged out on each request. If your user implements the EquatableInterface, then instead of these properties being checked, your isEqualTo() method is called, and you can check whatever properties you want. Unless you understand this, you probably won't need to implement this interface or worry about it.

## Traduction en français de l’extrait

**Comprendre la sérialisation et comment un utilisateur est enregistré dans la session**

Si vous êtes intéressé par l'importance de la méthode serialize() à l'intérieur de la classe *User* ou la façon dont l'objet *User* est sérialisé ou désérialisé, alors cette section est faite pour vous. Si ce n'est pas le cas, n'hésitez pas à sauter ceci.

Une fois l'utilisateur connecté, l'objet *User* entier est **sérialisé** dans la session. A la requête suivante, l'objet ***User* est désérialisé.** Ensuite, la valeur de la propriété *id* est utilisée pour interroger à nouveau un nouvel objet *User* à partir de la base de données. Ce nouvel objet *User* est alors comparé à l'objet ***User* désérialisé** pour s'assurer qu'ils représentent le **même** utilisateur. Par exemple, si le nom d’utilisateur dans le 2eme objet *User* ne correspondent pas pour une raison quelconque, par sécurité l'utilisateur sera déconnecté.

Même si tout cela se produit automatiquement, il y a quelques effets secondaires importants.

Tout d'abord, l'interface **Serializable** et ses méthodes serialize() et unserialize() ont été ajoutées pour permettre à la classe *User* d'être sérialisée dans la session. Cela peut être nécessaire ou pas selon votre configuration, mais c'est probablement une bonne idée. En théorie, seule la propriété **id** doit être sérialisée, car la méthode refreshUser() rafraîchit l'utilisateur à chaque requête en utilisant la propriété **id** (comme expliqué ci-dessus). Cela nous donne donc un "nouveau" objet *User*.

Mais Symfony utilise également les username, salt et password pour vérifier que l'utilisateur n'a pas changé entre les requêtes (il appelle également vos méthodes AdvancedUserInterface si vous l'implémentez). Si vous ne les sérialisez pas, vous risquez d'être déconnecté à chaque demande.

Si votre utilisateur implémente EquatableInterface, au lieu de vérifier ces propriétés, alors votre méthode isEqualTo() est appelée et vous pouvez vérifier les propriétés de votre choix.

À moins que vous n’ayez bien compris ce point, vous n'avez probablement pas besoin d'implémenter cette interface ni de vous en soucier.

# Conclusion

Ce travail sur Symfony m’a permis de me rendre compte d’une erreur à éviter lors de mes prochains projets.

A trop vouloir limiter le nombre de pages, de contrôleurs et d’entités, mon code est devenu délicat à reprendre par un autre développeur.

Je me suis attaché à résoudre seul mes erreurs, cela m’a fait perdre trop de temps. Un développeur doit savoir communiquer avec son équipe (en l’occurrence les autres apprenants).

Aussi, en lisant encore et encore la document Symfony, je vois qu’il me reste de quoi optimiser mon code. Cela se fera c’est certain mais avec un peu plus de temps.

Je comprends qu’être développeur, c’est avant tout :

* connaitre des framework et leurs avantages.
* savoir aller chercher rapidement les informations à jour dans les documentations,
* être capable d’utiliser les outils de test et de débogage,
* être curieux et avoir envie d’apprendre,

le reste vient avec l’expérience.

# ANNEXES

### ANNEXE 1 : Wireframe détaillé



### ANNEXE 2 : Explication de la recherche dynamique de suggestion

Page administrateur (listeUser) :

* La page listeUser affiche un formulaire de recherche.
* Le formulaire appel le contrôleur Stimulus **search-preview\_controller** avec en second argument l’URL du contrôleur **SearchController.**
* Chacune des entrées dans ce formulaire appellent la méthode **onSearchInput** du contrôleur Stimulus **search-preview\_controller** grâce à la commandedata-action="search-preview#onSearchInput"

Une image contenant texte

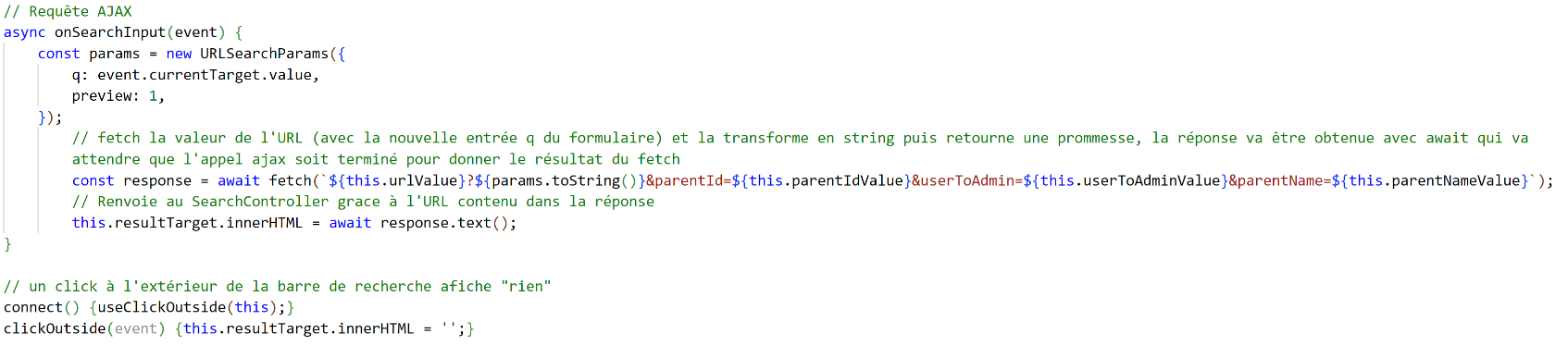
Description générée automatiquement

Controller Stimulus **search-preview\_controller :**

la méthode **onSearchInput** du contrôleur est une méthode asynchrone qui va :

* appeler l’URL du contrôleur **SearchController** (fetch) avec la nouvelle donnée q du formulaire
* Attendre une réponse
* Renvoyer le texte de la réponse (l’URL)
* Appeler cet URL

Le contrôleur écoute aussi un click à l’extérieur du formulaire qui aura pour résultat d’afficher « rien »



Controleur **SearchController**:

* Recherche les entités correspondantes à la recherche dans le model.
* affiche une page partielle **\_searchPreview** avec 2 résultats

Une image contenant texte

Description générée automatiquement

Page partielle searchPreview :

* Si la recherche contient un résultat => affiche les résultats avec un lien qui appel le contrôleur pour afficher les renseignements de ce résultat dans la page listeUser.
* Sinon affiche « Pas de résultat »

Une image contenant texte

Description générée automatiquement

### ANNEXE 3 : Exemple de test unitaire et fonctionnel

**1 Test Unitaire** : vérifier que la validation lors de la saisie d’un Email.

class EmailValidationTest extends TestCase

{

    public function testInvalidEmail()

    {

        $validator = Validation::createValidator();

        $errors = $validator->validate('pasUnEmail', [

            new Email(),

        ]);

        $this->assertGreaterThan(0, count($errors), 'Un Email incorrect peut être rentré !');

    }

}

**2 Test Fonctionnel** : vérifier que si connexion en tant que Structure => accès interdit à une page /administrateur

class StructureConnexionTest extends WebTestCase

{

    public function testLoginStructure(): void

    {

        // Appel d'un client qui agit en tant que navigateur

        $client = static::createClient();

        // aller sur la première page de connection

        $crawler = $client->request('GET', '/');

        // Vérifier l'accès à la page

        $this->assertResponseIsSuccessful();

        // vérifier que le titre H2 est bien égal à "Connectez-vous pour accéder à votre compte"

        $this->assertSelectorTextContains('h2', 'Connectez-vous pour accéder à votre compte');

        // récupérer le formumaire qui a le bouton "Se connecter" avec le crawler qui inspecte tout le DOM

        $submitButton = $crawler->selectButton('Se connecter');

        $form = $submitButton->form();

        // rentrer les données de l'utilisateur structure

        $form["email"] = "structure@exemple.com";

        $form["password"] = "structure";

        // soumettre le formulaire

        $client->submit($form);

        // Vérifier que le statut de réponse à la rediraction de la page est OK

        $this->assertResponseStatusCodeSame(Response::HTTP\_FOUND);

        // aller sur la page accueil administrateur

        $client->request('GET', '/administrateur/accueil');

        // Vérifier que la réponse à bien le statut 403

        $this->assertEquals(403, $client->getResponse()->getStatusCode(), 'Un utilisateur structure peut accéder à une page administrateur');

    }

}