
GAUTHIER Baptiste
Dossier de projet professionel
Titre Concepteur Développeur d'Applications
Site Web Synergie Family
Application Radio Grenouille

Sommaire

Liste des compétences du référentiel…… 3

Projet Synergie Family :

- Présentation du projet …… 3
- Spécifications fonctionnelles…… 4
- Fonctionnalités du projet…… 5
- Spécifications techniques…… 7
    - Technologies utilisées
    - Structure du projet
    - Organisation du travail
    - Utilisation de git
    - Conception de la base de données
- Réalisation et extraits de code……12
    - Fonctionnement d'un controller
    - Développer des composants d'accès aux données
    - ORM Doctrine et liaisons entres les tables
    - Gestion des migrations
    - Mise en place des Data Fixtures
    - Utilisations des variables d'environnement associé à un service
    - Travail sur le SEO
    - Optimisation du front-end : mise en place de WebPack

- Veille sur les vulnérabilités de sécurité…… 17

Projet Application Radio Grenouille

- Présentation du projet…… 20
- Réalisation et extraits de code
    - Maquettage de l'application
    - Utilisation des composants
    - Cycle de vie d'un composant
    - Création de l'API
    - Swagger
    - Test unitaire
- Recherche à partir d'un site anglophone

Annexes…… 35

Liste des compétences du référentiel couvertes par les projets :

Les projets couvrent les compétences ci-dessous :

"1. Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité" :

- Maquetter une application
- Développer une interface utilisateur de type desktop
- Développer des composants d'accès aux données
- Développer la partie front-end d'une interface utilisateur web
- Développer la partie back-end d'une interface utilisateur web

"2. Concevoir et développer la persistance des données en intégrant les recommandations de sécurité" :

- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données

"3. Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité" :

- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
- Concevoir une application
- Développer des composants métier
- Construire une application organisée en couches
- Développer une application mobile
- Préparer et exécuter les plans de tests d'une application
- Préparer et exécuter le déploiement d'une application

Présentation du projet Synergie Family :

Implantée dans les Bouches du Rhône, en Ile de France, à Lyon et au Cameroun, Synergie Family est une start-up associative d'environ 580 salariés spécialisée dans la conception et la mise en œuvre de projets sportifs, culturels et socio-éducatifs. Que ce soit au sein des centres sociaux qu'elle gère en délégation de service public ou dans le cadre d'actions auprès de bailleurs sociaux, Synergie Family œuvre pour l'innovation sociale dans les quartiers et l'épanouissement de leurs habitants, particulièrement les plus fragilisés.

Synergie Family est une start-up associative qui s'implique avec force et conviction depuis plus de 10 ans, dans le développement d'actions éducatives au bénéfice de l'enfance, de la jeunesse et de la famille. Leur investissement dans le temps de loisirs s'inscrit dans une

démarche d'épanouissement de l'individu dans sa globalité. Apporter des savoirs et des compétences par l'intermédiaire d'activités ludiques, telle est leur ambition.

Le site possède plusieurs fonctionnalités : un back office permettant de mettre à jour le contenu, un système de candidature et d'offre d'emploi, une inscription newsletter connecter à une API, un formulaire de contact, un système de blog filtré.

Spécifications fonctionnelles :

Périmètre du projet :

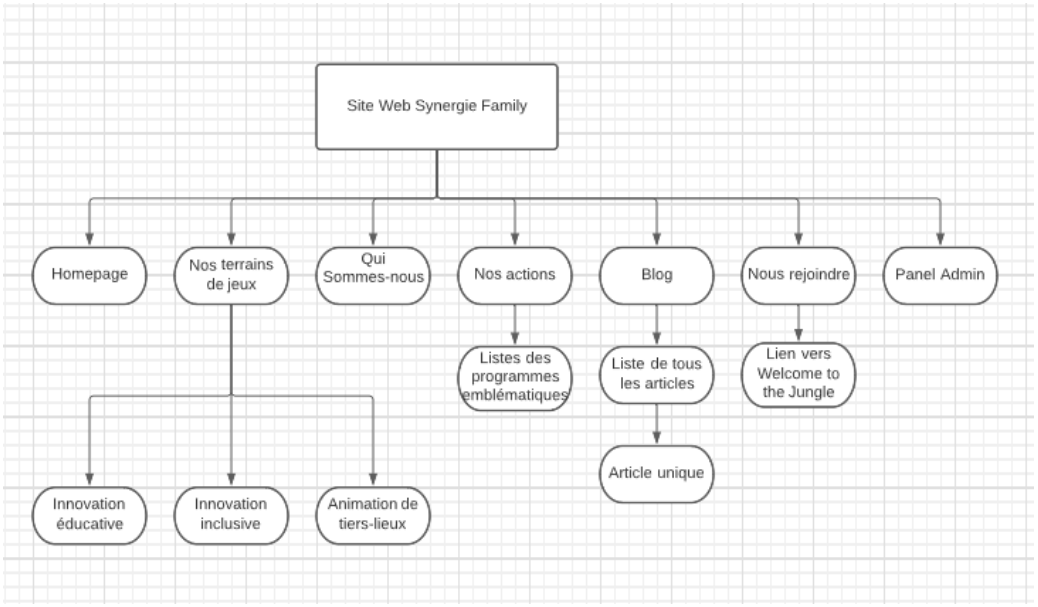
Ce projet a été réalisé au sein de mon entreprise durant ma période d'alternance. Il s'agit de la refonte complète du site web de l'entreprise. Le site est composé d'une partie statique qui présente toute l'activité de Synergie Family ainsi que leurs programmes emblématiques. Il y a également une partie dynamique avec un blog et une rubrique offres d'emploi. Plusieurs sections du site sont également dynamiques tel que des carousel, témoignages, vignette d'articles et chiffres clés.

Le but du projet était donc de créer une vitrine pour l'entreprise afin d'expliquer de manière simple les différentes actions de Synergie. L'association a pour particularité d'avoir beaucoup de projets phare et d'en accueillir de nouveaux régulièrement. Ainsi, le contenu du site est amené à souvent être modifié, il est donc nécessaire d'avoir une bonne partie de contenu dynamique.

Arborescence du site

Le site web est composé de plusieures page :

- Une page d'accueil
- Trois page présentant chaques terrains de jeux :
    - Innovation éducative
    - Innovation inclusive
    - Animation de tiers-lieux
- Une page Nos actions regroupants les programmes emblématiques
- Une page Qui sommes-nous ?
- Une page Blog
- Une page Contact
- Une page Nous rejoindre qui redirige vers les offres d'emploies
- Un espace administrateur
- Une page mentions légales



Description des fonctionnalité du projet

Module Connexion

Pour accéder au back office l'utilisateur doit passer par une page de connexion. Le système de login est paramétré avec le bundle Security de symfony, accompagné des plusieurs rôles : un rôle admin, un rôle éditeur qui à accès à la modification de contenu du site et toute la partie blog, un rôle RH qui peut gérer toute la partie consacré au offres d'emploi.

Une partie blog

Plusieurs articles sont disponibles sur le site. Ils peuvent être ajoutés et mis à jour depuis le back office. Un système de filtres en Ajax à été mis en place pour améliorer l'expérience utilisateur. Il y a la possibilité de filtrer par catégorie, sélectionner les différents tags, accompagné d'une barre de recherche et d'un système de pagination.

Un panel administrateur

Depuis l'url /admin l'utilisateur peut se connecter au back office. Ce panel administrateur à été réalisé avec le bundle EasyAdmin de Symfony. J'ai donc mis en place un crud de mes entités. Plusieurs fonctionnalités sont présentes : ajout/suppression/modification, possibilité d'effectuer des actions selon les rôles, affichage des données différentes selon les actions, système de filtres et barre de recherche, importations d'images et de pdf, mise en place du dashboard avec une documentation.

Contenu dynamique

On retrouve beaucoup de contenu dynamique sur le site : carrousel, article interview, témoignages, bannière, slider et kpi. Le tout peut être modifié directement depuis le back office.

Une partie job et candidature

Une feature pour le moment qui n'est pas mise en place car l'entreprise dépend encore de welcome to the jungle, mais elle à pour but par la suite d' être directement implanté sur le site. On retrouve donc un page avec la liste des offres que l'on peut filtrer selon le type de contrat, le lieu, le télétravail ou non. Un page spécifique pour voir tous les détails d'une offre. Un formulaire pour postuler à l'offre en question.

Une formulaire de contact

Un formulaire de contact connecté au SMTP de sendinblue. Le formulaire envoie un mail à une adresse dédiée, les informations de l'utilisateur ainsi que le contenu du mail sont également stockées en base de données.

Une formulaire d'inscription à la newsletter

Synergie Family à choisis Sarbacane pour toute la gestion d'emailing. Ainsi un formulaire est disponible directement sur le site permettant de s'abonner à la newsletter. Le formulaire communique directement avec l'API de sarbacane et est ajouté à la liste de contact de l'outil.

Intégration d'un player youtube

Plusieurs vidéos youtube sont directement intégrées sur le site. Pour l'affichage de ces iframes l'API youtube est directement intégré afin de pouvoir facilement manipuler les iframes.

Spécifications techniques du projet

Stack du projet :

Pour la partie back-end :

Le projet utilise le framework Symfony 6 et php 8.
La base de données est de type MySQL

Pour la partie front-end :

Le projet utilise les technologies Javascript (Vanilla) et CSS.

Comme outil de versionning j'ai utilisé GIT et l'application GitHub Desktop.

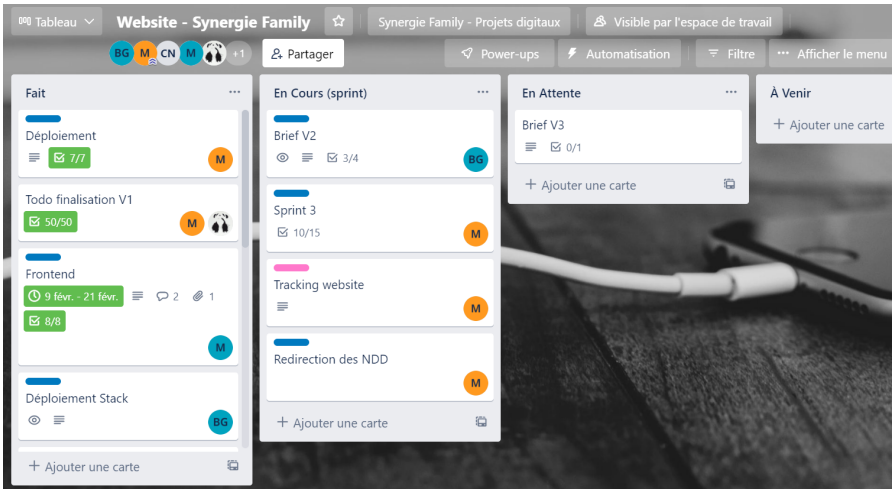
J'ai utilisé Laragon comme environnement de travail local.

Mon éditeur de texte est Visual Studio Code, accompagné des extensions : Live Sass Compiler, Live Server, Php Intelephense, php DocBlocker, php namespace resolver, todotree.

Organisation et environnement de développement

L'équipe chargée du projet du développement du site est composée de 6 personnes au total. Caroline, Laetitia sont du pôle communication et Mily une graphiste freelance. Concernant la partie développement, Michael le CDO avait le lead sur le projet, Malik un développeur freelance s'est occupé du front et j'ai réalisé la partie back end.

Nous avons donc mis en place plusieurs outils. Dans un premier temps figma, ou l'on retrouve toute la maquette du projet avec plusieurs commentaires sur certaines sections. Un trello divisé en plusieurs colonnes : Fait, en cours, en attente, boîte à idée (comm et dev), ressources et questions. On retrouve dans ces colonnes plusieurs cartes, c'est ici que je retrouve le listing de toutes les tâches à faire pour ma feature.



Au sein du pôle informatique nous travaillons en méthode agile. Tous les matins, un daily pour suivre l'avancée du projet. Chaque personne explique de manière très brève ce qu'il a fait hier et ce qu'il compte faire aujourd'hui. Ainsi on peut facilement suivre l'avancement et soulever les problèmes rencontrés. Un point hebdo plus conséquent tous les mercredis avec l'équipe au complet pour suivre l'avancement de chacun sur tous les projets.

Nous avons également mis en place Slack, permettant d'avoir un chat instantané. Mon tuteur étant souvent en réunion, les daily se font également sur slack. Si je rencontre un problème bloquant ou une éventuelle questions, j'utilise également ce moyen de communication pour le tenir au courant au plus vite.

Concernant l'utilisation de git, nous avons un gitlab consacré au projet. Une branche est créée pour une feature. Une fois la feature terminée, je crée une merge request, qui doit être validée par le lead dev. On réalise une code review ensemble puis on effectue le merge dans une branche consacré au sprint. Une fois que tout est fonctionnel on peut merge vers le main, puis envoyer en production.
Pour les commits je précise s'il s'agit d'un fix ou d'une feature. Je suis le plus explicite possible pour faciliter la lecture de la merge request par la suite.

Nous utilisons également confluence pour la documentation du projet. Tout ce que j'ai vu durant les codes reviews m'a permis de faire une documentation sur les bonnes pratiques. J'ai également écrit plusieurs docs sur l'utilisation des bundles, les informations sur le projet etc. Le but étant dès l'arrivée d'un nouveau développeur sur le projet, il est directement tenu au courant des informations importantes, sans forcément devoir trop solliciter les équipes. Par exemple, le développeur front freelance avec qui j'ai travaillé était novice sur Symfony et Webpack. Je lui ai donc transmis toute ma documentation sur l'installation du framework et ses concepts.

Jira Software

Nous avons également utilisé Jira sur un autre projet. Un logiciel avec un gestion de ticket, possibilité de faire des sprint et de voir le backlog. Chacun s'empare du ticket et peut définir le statut : à faire, en cours, prêt pour MR ou terminé. Durant le développement du ticket on documente en 4 étapes : Le **sujet** de la tâche à accomplir (rempli par le responsable de projet), la **documentation** (toutes les tâches réalisées, screenshots et explications), les **étapes des tests** et les **étapes de mise en production** (seulement si nécessaire).
Une fois cette documentation terminée, on peut directement la migrer sur Confluence pour avoir une trace écrite de toutes les opérations réalisées sur le ticket.

Exemple d'un ticket jira :

Projets / SyGTA / GTA-29

Erreur sur le total tableau des activités

Joindre | Créer une sous-tâche | Associer un ticket

Description

Erreur sur le total global du tableau des activités intervenant. Il ne prend pas en compte les HC.

Il faut vérifier les cas de production de ce bug et corriger au besoin.

Documentation

Tableau global des activités :

- Changement CSS : padding thead, bg color lors du hover sur chaque ligne et réduction taille des colonnes → tableau donc plus petit, margin auto pour le centré. Total global en bold + entête plus visible
- Inversion des colonnes : HC est désormais placé avant le total global mis en dernière position

Le hover concerne tous les tableaux d'activités (à garder ou pas ?)

Les temps sont séparés (AFFICHAGE SELEUMENT) :

- Perisco = Total perisco - HC

data → **totalUserDurationHoursSchools** (perisco) = persico + HC

Num. mat.	Animateur	Lieux	REUNION	FORMATION	AUTRES	ABS	Total Périsco	Total ACM	HC	Total global (H + HC)
10927	ABDALLAH DJAHA Kadafi	PETIT BOSQUET + Annexe (ELEM) GRANDE BASTIDE CAZAULX (MAT+ELEM) MONTOLIVET (ELEM) Les Caillols	4	0	0	1	93.25	45.00	0	**142.25**
11018	ABDOU Zahed	(ELEM) SAINT JULIEN 2 (ELEM)	2	0	0	1	52.75	0.00	6.75	**61.50**

Tester le code

Se connecté via un compte administrateur et parcourir le tableau d''activité

J'ai également dû faire la présentation de mon back office au service RH et communication. Une documentation est également à disposition sur le dashboard du back office. Cela m'a appris aussi à collaborer avec des personnes qui n'ont pas forcément de compétences techniques dans le domaine et donc devoir vulgariser les informations pour qu'elles soient comprises de tous.

Utilisation de git

Concernant les commit j'utilise deux préfixe : **[feature]** ou **[fix]**. Cela permet de savoir directement de quoi il s'agit. Je détaille par la suite au maximum toutes les tâches réalisées pour facilité par la suite la lecture de la merge request.

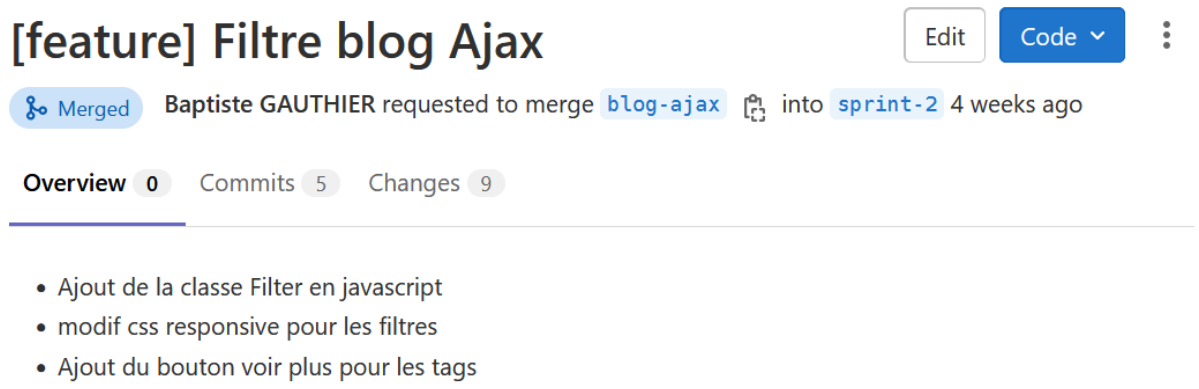
```
commit 67c944bd96e16499c602fcb1b393ce0752bf4c1c
Author: Baptiste <baptiste.gauthier@synergiefamily.com>
Date:   Tue May 24 17:12:44 2022 +0200

    [fix] smtp + crud author

    - Ajout envoie de mail sur le formulaire de contact (ajouter le
      username/pass sendinblue dans le .env)
    - install bundle sendinblue
    - Suppression crud author
    - Ajout texte nos programmes éducatifs
```

Dans le cas oú l'on doit rapidement changer de branche, sur un hotfix par exemple, alors que notre code n'est pas propre pour réaliser un commit j'ai utilisé deux méthodes. La première est de commit malgré tout, et d'effectuer un rebase avec le prochain commit pour lier les deux et avoir un seul commit propre. La seconde est d'utiliser git stash, ce qui me permet de mettre mes changements de côté puis les appliquer lors de la reprise du développement de ma feature.

Concernant les merges request j'indique dans le titre la feature principale puis je liste les différentes tâches à vérifier.

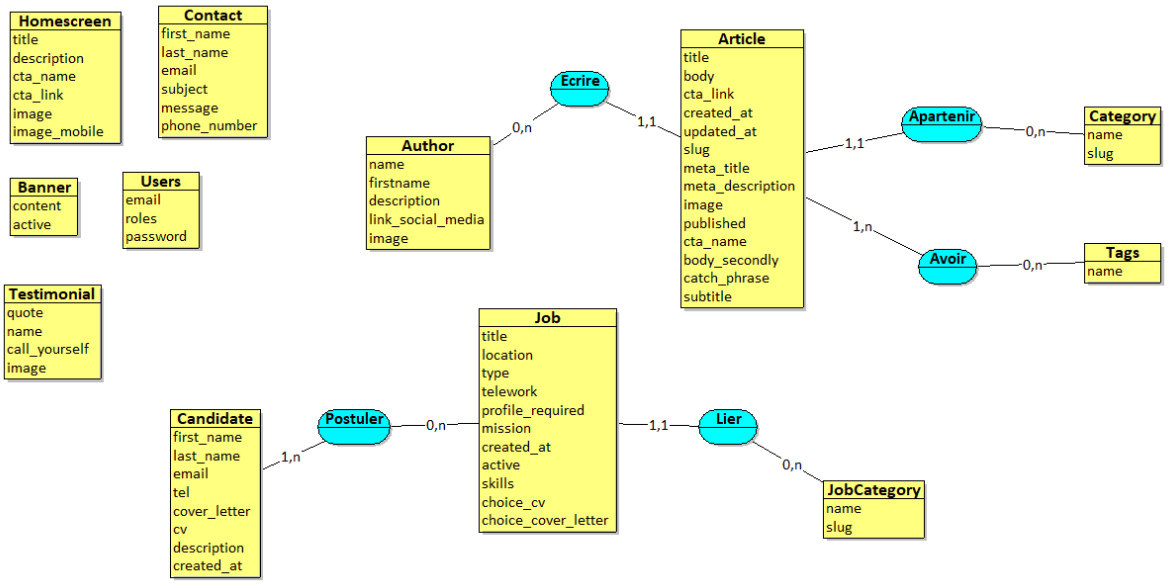


Structure du projet :

Le projet est organisé en plusieurs dossier :

- le dossier **src** dans lequel on retrouve :
    - Un dossier Controller avec l'ensemble des Controller de l'application
    - Un dossier Repository
    - Un dossier pour les formulaire
    - Un dossier pour les Fixtures
    - Un dossier pour les Services
- le dossier **public** dans lequel il y a l'index.php, htaccess et tous les fichiers uploader
- le dossiers **assets** avec les fichiers javascript, les images, les pdf et les fichiers style
- le dossier **vendor** où l'on retrouve tous les bundles
- le dossier **migrations** avec l'ensemble des migrations
- le **templates** avec tous les fichiers twig

Conception de la base de donnée :



Concernant la base de données, j'ai dans un premier temps réaliser le modèle conceptuel de données. Ma base se compose donc d'un table article, category, job, candidate, tags, jobcategory. Pour les liaisons on retrouve une OneToMany entre article et category, une autre avec author et une entre job et jobcategory. On a également deux ManyToMany entre article

et tags et entre job et candidate. Il y a donc deux tables de liaisons pour stocker les clés étrangères de chaque table, comme on le voit dans le MLD (*voir en annexes*).

On retrouve également cinq autres tables qui sont sans liaisons, servant seulement à stocker des données et rendre du contenu dynamique sur le site, modifiable via le back office.

A l'aide de l'ORM doctrine je crée alors toutes les entités et les différentes liaisons.
Je crée ma base de données puis j'exécute une migration, la base de données est à présent opérationnelle.

Réalisation et extraits de code

Fonctionnement des Controller de l'application

Une controller est une porte d'entrée. Il va seulement s'occuper d'appeler les différents modèles dont il a besoin pour répondre à une problématique et les transmettre à la vue par la suite. Les routes pour les pages dites plus statiques vont être à l'intérieur du PageController. Les pages dynamiques tel un système de blog, un page job ou un formulaire de contact, vont avoir un controller dédié.

Développer des composants d'accès aux données

Symfony se base sur le modèle MVC. Durant le développement du site de Synergie Family, j'ai réalisé un blog accompagné d'un système de tri en Ajax.

Lors de la création d'une entité, un Repository est directement créé. C'est cette classe qui va gérer toutes les intéractions avec la base de données. Si l'on veut persister, mettre à jour ou récupérer des données depuis la base, c'est ici que l'on va écrire nos différentes méthodes.

Le Controller lié à une route peut alors faire appel à un repository. On exécute alors la fonction du repository pour récupérer les données. L'objet récupéré est alors transmis à notre vue twig grâce à la fonction **return**. On peut alors afficher les données sur notre page web.

Chaque repository hérite d'une classe générique qui possède des méthodes simples permettant de rapidement récupérer des données. Par exemple, la méthode *findAll()* est disponible pour tous les repository et permet de renvoyer tous les éléments d'une entitée. Il en existe d'autres qui permettent de filtrer un peu plus en profondeur, mais cela reste très générique.

Si l'on souhaite faire une requête un peu plus complexe, c'est à nous de créer notre méthode. Symfony propose un objet : le Query Builder. Il permet de faire des requêtes SQL en utilisant la programmation objet. On retrouve donc des méthode *select()*, *where()*, *join()*, *limit()* etc. Le véritable avantage de cette méthode par rapport au SQL classique, c'est qu'il permet d'exécuter certaines requêtes selon des conditions et donc faire des requêtes plus

modulables (selon un rôle par exemple on peut très facilement demander à ne pas envoyer les mêmes données).
J'ai utilisé cette méthode pour filtrer les articles de blog. Si un champ est null ou vide, la requête n'est pas la même.

```php
/**
 * Récupère les articles en lien avec une recherche
 * @return PaginationInterface
 */
public function findSearch(SearchData $search) : PaginationInterface {

    $query = $this
                ->createQueryBuilder('a')
                ->select('a')
                ->addSelect('t')
                ->join('a.tags' , 't')
                ->orderBy('a.createdAt' , 'DESC')
                ->where('a.published = TRUE');

    if($search->q !== null || $search->category !== null || $search->tags !== null) {
        $query = $query
                    ->where('a.title LIKE :q')
                    ->andWhere('a.published = TRUE')
                    ->setParameter('q', "%{$search->q}%");
```

```php
class ArticleController extends AbstractController
{
    #[Route('/article', name: 'article', options: ['expose' => true])]
    public function index(ArticleRepository $repository , Request $request): Response
    {
        $data = new SearchData();
        $data->page = $request->get('page' , 1) ;
        $form = $this->createForm(SearchForm::class, $data);
        $form->handleRequest($request);

        // tableau php qui renvoie les data en fonction de la recherche effectuée
        $article_search = $repository->findSearch($data);

        if($request->isXmlHttpRequest()) {
            return new JsonResponse([
                "uri" => $request->getRequestUri(),
                "content" => $this->renderView('article/article_search.html.twig' , ['article_search' => $article_sea
            ]);
        }

        return $this->render('article/index.html.twig' , [
            'article_search' => $article_search,
            'form' => $form->createView(),
        ]);
    }
```

Avec l'injection des dépendances, j'indique à mon controller qu'il a besoin d'appeler l'ArticleRepository. J'appelle donc la fonction *findSearch()*, puis dans la fonction render je transmet le résultat dans ma vue twig.

ORM Doctrine et liaisons entres les tables

L'ORM Doctrine est fourni directement avec Symfony. C'est une couche logicielle qui sert à faire le lien entre une base de données relationnelle et un code utilisant la programmation orientée objet. En créant toutes les entités depuis symfony j'ai défini mes différentes liaisons entre mes tables. Dans une relation entre deux tables, il y a toujours deux côtés différents : le côté **prioritaire** et le côté **inversé**. Dans une relation OneToMany, doctrine définit automatiquement la table prioritaire comme celle qui possède la clé étrangère. Par exemple, un article possède une catégorie et une catégorie peut appartenir à plusieurs articles. Ici l'entité article est du côté prioritaire et catégorie du côté inversé. On va plus souvent chercher à connaître la catégorie de l'article en question, plutôt que chercher un article depuis une catégorie. Doctrine par défaut vient bloquer toutes les requêtes du côté inversé pour réduire le nombre de requêtes SQL de notre application. Il est tout à fait possible de modifier le côté de chaque entité ou même d'effectuer les requêtes dans les deux sens si on le souhaite. Lors d'une relation ManyToMany, c'est nous qui décidons quelle entité est prioritaire ou inversée.

On peut modifier tout ça directement avec une annotation sur notre liaison dans le code de l'entité.

src/Entity/Article.php

```php
/**
 * @ORM\ManyToOne(targetEntity=Category::class, inversedBy="articles")
 * @ORM\JoinColumn(nullable=true, onDelete="SET NULL")
 */
private $category;
```

src/Entity/Category.php

```php
/**
 * @ORM\OneToMany(targetEntity=Article::class, mappedBy="category")
 */
private $articles;
```

Doctrine permet de limiter la taille des requêtes selon notre affichage dans twig. En effet, si j'effectue une fonction findAll() sur mon Repository Article, je vais récupérer toutes les informations plus les identifiants des clés étrangères. Par exemple, un article possède un auteur. Je récupérerai donc l'id de mon auteur. En revanche ses informations seront null (nom de l'auteur, prénom de l'auteur etc). Si je désire afficher le nom de mon auteur, c'est tout à fait possible. Dans mon twig je pourrai y accéder avec la variable **article.author.name**. Ainsi doctrine s'adapte et fait directement la jointure pour nous en modifiant la requête selon ce qu'on désire afficher.

Gestion des migrations

Les migrations permettent d'avoir un historique de la base de données du projet. Dès lors qu'une migration est effectuée, elle est directement stockée en base de données dans une table dédiée. Ainsi on peut à tout moment voir notre avancée en comparants les migrations en local sur notre projet et celle dans notre base de données. On peut donc voir le statut actuel des migrations, quelle est la dernière effectuée, s'il y a des migrations manquantes, se déplacer de migrations en migrations pour revenir en arrière etc..

Une bonne pratique que j'ai mis en place et de synthétiser plusieurs migrations en une seule pour une feature. Si durant la feature que je développe j'ai effectué plusieurs migrations : ajout d'un champ en bdd, ajout d'une table, retrait d'un champ, changement de typage etc.. On se retrouve parfois avec beaucoup de migrations pour peu de ligne. Pour avoir un historique propre et ne pas le surcharger, une fois la feature terminer, je reviens en arrière à la dernière migrations avant la feature. Je supprime toutes les migrations effectuées en local, puis j'en génère une nouvelle à partir du nouveau code. Ainsi j'ai une seule migration pour ma feature. Il est donc plus facile pour le responsable de projet de tester le code et de ne pas avoir de conflit avec l'historique en production.

Mise en place des Data Fixtures

Au commencement du développement de l'application j'ai mis en place des fixtures sur le projet. Combiné au package Faker(faux nom, prénom, date, lieux etc…) cela m'a permis de générer plusieurs données fictives et ainsi pouvoir commencer à travailler avec de la donnée. Le véritable avantage et lors de la collaboration sur un projet, notre partenaire développeur n'a plus qu'à exécuter une ligne de commande pour remplir sa base de données. Nous avons le choix d'écrire toutes nos fixtures dans un seul et même fichier, ou bien d'en créer plusieurs et de choisir l'ordre dans lesquelles elles sont exécutées.

Utilisations des variables d'environnement associé à un service

Symfony propose d'avoir plusieurs environnements. Un de test, un de production et un de développement. On a également le env local qui s'applique uniquement sur notre machine. Concernant toutes les clés Api du projet je les déclare dans mon fichier .env sans leur donner de valeur. Je leur affecte une valeur dans mon env local pour réaliser mes développements.

Cela permet de garantir une sécurité et de ne pas laisser les clé API ou les mot de passe en clair dans le fichiers env. On peut également générer des clé Api pour chaque environnement afin de dissocier au maximum les choses selon les environnements. J'ai utilisé cette méthode pour effectuer une inscription de newsletter.

J'ai créé un NewsletterService, que je définit comme service dans le fichier service.yaml. Dans ce fichier je peux également définir les arguments de mon controller. Ainsi je lui définit

toutes mes variables d'environnement : clé API, identifiant du compte API, identifiant de la liste d'utilisateur et identifiants d'un champ personnalisé. Ma classe Newsletter Controller aura directement dans son constructeur ces différentes données. Ainsi on a un code qui fonctionne en local comme en production, il faudra seulement renseigner les bonnes données dans le fichier de configuration .env.

services.yaml

```yaml
App\Service\NewsletterService:
    arguments:
        - '%env(SERBACANE_API_ID)%'
        - '%env(SERBACANE_API_KEY)%'
        - '%env(SERBACANE_API_LIST_ID)%'
        - '%env(SERBACANE_API_DATE)%'
```

.env

```
###> serbacane/newsletter ###
SERBACANE_API_ID=
SERBACANE_API_KEY=
# id de la liste dans laquelle ajouter un nouvel abonné
SERBACANE_API_LIST_ID=
# id du champ personnalisé (date)
SERBACANE_API_DATE=
###> serbacane/newsletter ###
```

NewsletterService.php

```php
<?php

namespace App\Service;

use DateTime;
use DateTimeZone;

class NewsletterService
{
    private $apiId;
    private $apiKey;
    private $apiListId;
    private $apiIdDate;

    public function __construct($apiId, $apiKey, $apiListId, $apiIdDate)
    {
        $this->apiId = $apiId;
        $this->apiKey = $apiKey;
        $this->apiListId = $apiListId;
        $this->apiIdDate = $apiIdDate;
    }
```

Travail sur le SEO

J’ai réalisé quelques petites optimisations sur le SEO. Dans un premier temps, j’ai vérifié toutes les erreurs en passant le site sur le W3C Validator. J’ai travaillé sur le renommage des images, lorsque l’on importe une image dans le back office, elle prend directement le nom du slug. J’ai également beaucoup travaillé sur le bundle LiipImagine.

Il propose de configurer des filtres (cropper les images, pourcentages de qualité, largeur maximum etc), que l’on peut appliquer via un filtre twig sur l'élément directement. Une nouvelle image est alors généré selon nos paramètres et est stockée en cache. On peut donc très facilement réduire le poids des images sur tout le site. Il possède également un deuxième avantage, il peut générer une deuxième fois l’image au format WEBP. Si le navigateur comprend ce format, le chemin de l’image sera automatiquement modifié, sinon il reste sur le format classique.

Optimisation du front-end : mise en place de WebPack

Pour ce projet je ne me suis pas occupé de faire la partie frontend du site (css et javascript du projet). En revanche j’ai mis en place l’utilisation de Webpack sur le projet. En effet, Webpack permet de compiler tous nos fichiers css et javascript en un seul. Ainsi toutes les librairies sont directement prises en charge dans un seul fichier. Il permet également de donner une version à nos fichiers css et javascript et ainsi côté client charger à chaque fois les bons fichiers sans se soucier de problèmes de cache.

J’ai également fait la structure de tous les points d’entrées : quels fichiers globaux sont appelés sur toutes les pages, quels sont les fichiers à charger sur cette page essentiellement etc.

Veille sur les vulnérabilités de sécurité :

Faille XSS

Pour décrire rapidement la faille XSS on peut simplement dire qu'elle consiste à injecter du code malicieux dans une page Web, grâce à une variable faillible (c'est-à-dire non sécurisée ou mal sécurisée) qui sera affichée.

Si l’utilisateur rentre une balise dans un input, elle sera alors interprétée par le navigateur (ex : <script> Je fais ce que jeux ici ! </script>).

Pour la faille XSS il faut procéder à la validation de tous nos inputs. Symfony permet d’appliquer des contraintes directement sur les attribut de notre entité. On peut également spécifier dans l’application la liste de tous les champs utilisés et indiquer leurs contraintes de sécurité.

Injection SQL :

L’injection SQL est un type d’attaque sur une application web qui permet à un attaquant d’insérer des instructions SQL malveillantes dans l’application web, pouvant potentiellement accéder à des données sensibles dans la base de données ou détruire ces données.

Lors d’un formulaire de connexion par exemple, l’utilisateur pourra modifier la requête SQL via l’input de la connexion.

Avec l’utilisation de doctrine on peut échapper les données insérées par l'utilisateur en utilisant la méthode setParameter() de l’objet queryBuilder. Ainsi, les requêtes personnalisées deviennent des requêtes préparées, et évitent les injections SQL.

Faille upload :

Le principe de l'attaque est très simple. Le pirate essaie d'uploader un fichier qui contient du code malveillant ou un code PHP de sa création. Si la faille est là, alors le fichier finira par atterrir sur le serveur. Il suffit ensuite au pirate d'appeler son fichier pour que celui-ci s'exécute.

C’est un risque lorsque l’utilisateur est amené à upload une image, une vidéo ou un CV.

Dans le cadre du projet du site web de Synergie Family, j’ai ajouté un contraintes sur mes fichiers image et pdf, permettant de limiter le poids et de vérifier l’extension.

```php
/**
* @Vich\UploadableField(mapping="article_images", fileNameProperty="image")
* @var File
*/
#[Assert\File(
    maxSize : '2M',
    mimeTypes : ['image/png', 'image/jpeg' , 'image/jpg' , 'image/gif'],
    mimeTypesMessage : 'constraint.mime_type'
)]
private $imageFile;
```

Attaque de force brute :

Consiste via un programme de tester toutes les combinaisons possibles pour trouver un mot de passe et accéder à un compte admin ou utilisateur.

Pour l'utilisateur, il est nécessaire d'utiliser des mots de passe différents sur chaque site afin d'assurer plus de sécurité.

Utiliser des mots de passe complexes. Forcer durant l'inscription l'utilisateur a remplir des mot de passes selon des conditions précises (nombres de caractères, majuscules, chiffres, caractères spéciaux...).

Pour ça on peut utiliser les expressions régulières des contraintes symfony sur le champ password. Le mieux reste encore de bloquer la connexion au bout de plusieurs tentatives ou d'intégrer un captcha.

Projet Radio Grenouille :

Le projet de notre application était de réaliser un app mobile pour la radio marseillaise Radio Grenouille. C'est une radio qui propose un flux radio musical et des programmes dans le domaine des Arts & Cultures, Société & Citoyenneté, Environnement & Ecologie et enfin Jeunesses.

Actuellement la radio ne possède qu'un site web, le but étant d'adapter les principales fonctionnalités aux formats d'une application mobile.

Sur l'application on retrouvera donc principalement : la liste des programmes ainsi que leurs différents podcast associé, avec la possibilité de les réécouter quand on le souhaite. La possibilité d'ajouter en favoris les différents programmes et les retrouvés dans notre bibliothèque. Un système de connexion et d'inscription et un player pour le flux radio en temps réel.

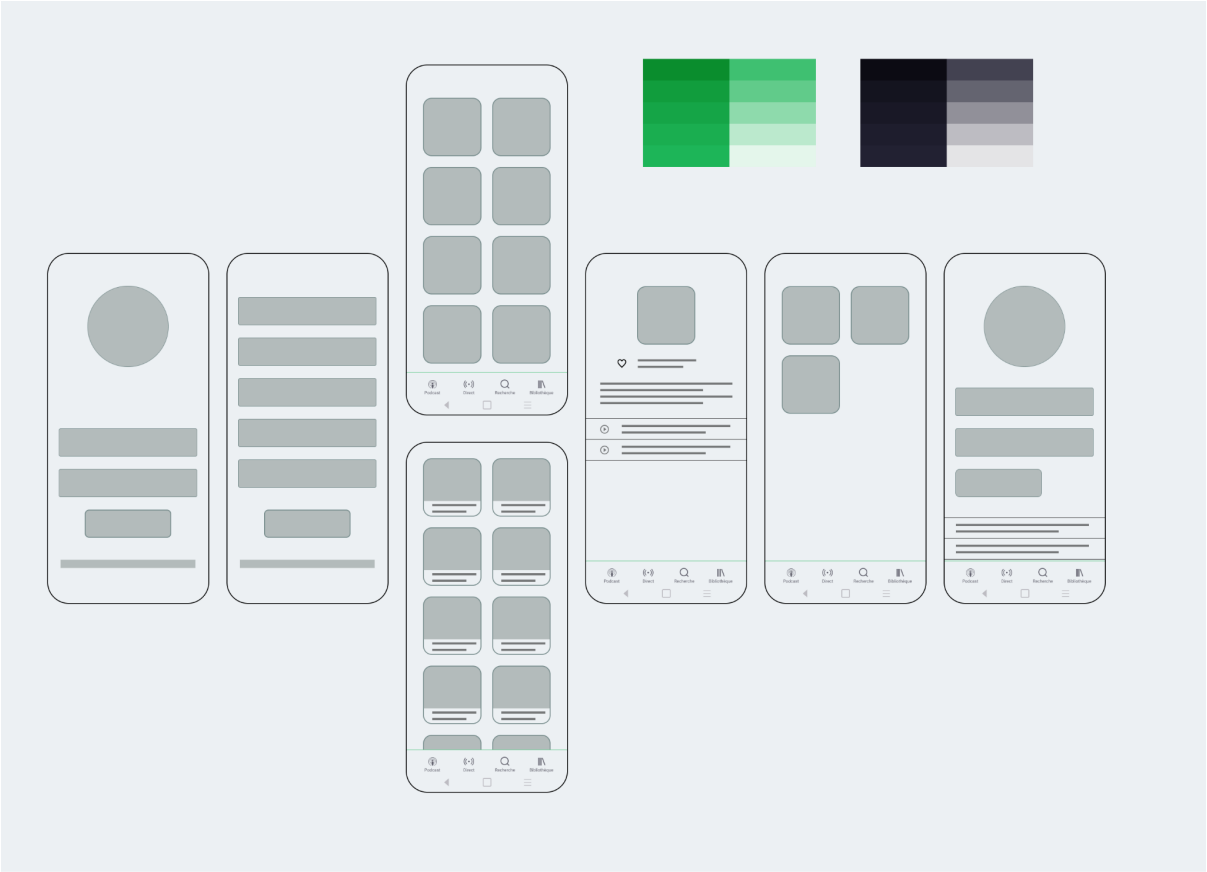
Le partie front-end de l'application à été réalisé avec le framework React Native. Une technologie qui nous permet d'utiliser React avec les fonctionnalités natives d'appareil android et IOS.

Il nous permet donc de créer une application adaptable sur différents supports.
L'application est structuré avec plusieurs dossiers :

- Un dossier **assets** dans lequel sont stockées toutes les images et les font
- Un dossier **src** dans lequel on retrouve plusieurs dossiers :
    - Un dossier **component** qui regroupe tous les composants de l'application
    - Un dossier **screens** avec les différents écrans de la l'application
    - Un dossier **navigation** qui contient le fichiers faisant le lien entre tous les écrans et l'appel vers les composants
    - Un dossier service avec les requêtes vers Transistor, une api tierce vers laquelle on récupère les différents podcast

Maquettage de l’application

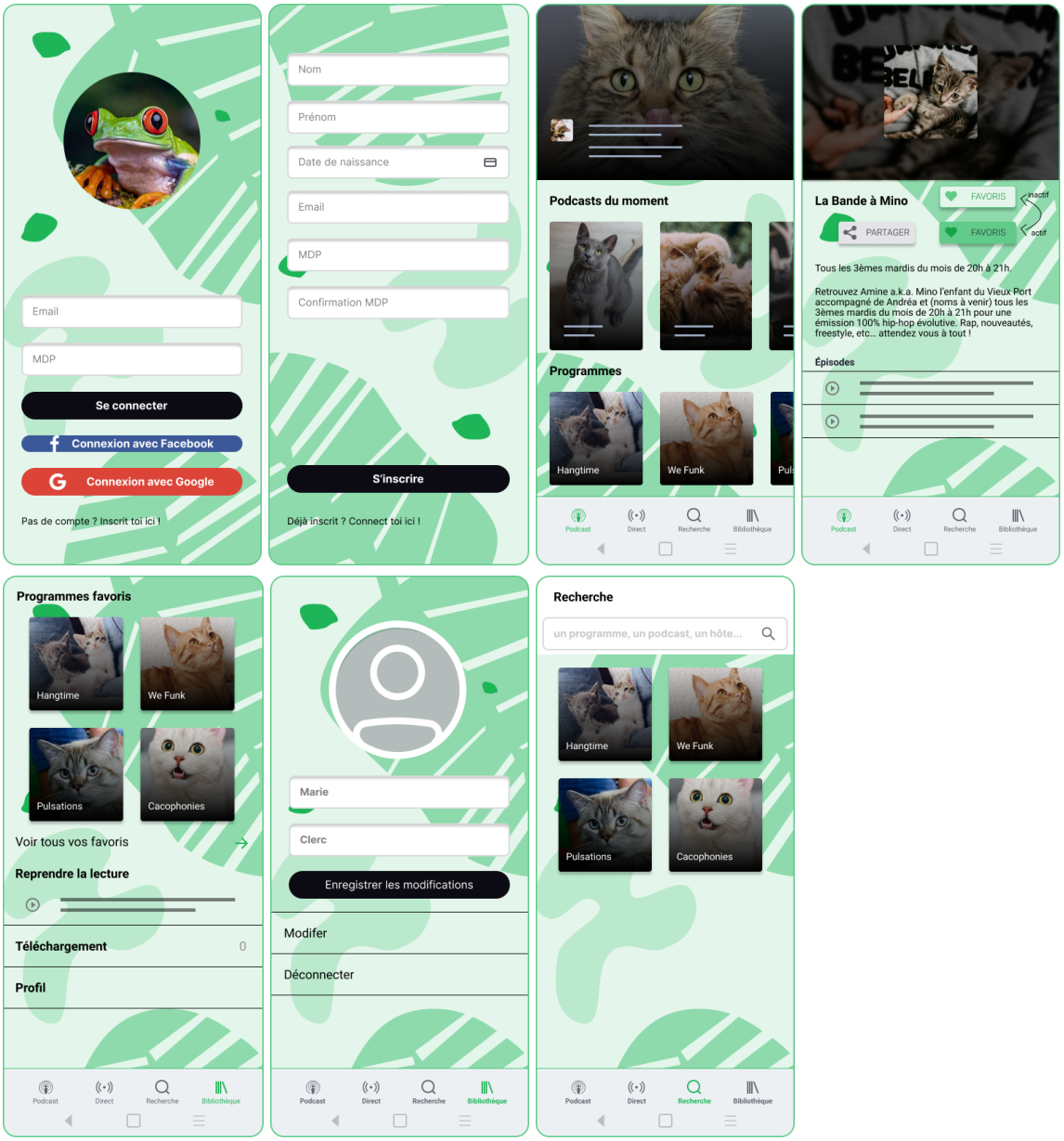
Nous avons utilisé l’outil figma pour réaliser la maquette de notre application.
Nous avons dans un premier temps élaborer le wireframe.



Ces wireframes nous ont permis d’avoir une première ébauche du projet et d’affiner notre design en fonction des fonctionnalités dans la deuxième étape de conception : la maquette.

Les deux premières frames correspondent aux pages d’inscriptions et de connexion, puis les autres correspondent à la navigation parmi les programmes, la lecture d’un programme sélectionné, le profil, la recherche de programme ainsi qu’une navigation.

Dans l’ordre : une page connexion, inscription, accueil, détail programme, programme favoris, profil et recherche.



La maquette réalisée en amont nous à permis de faciliter l'intégration. On peut directement observer les différents composants qui vont être réutilisables. On y retrouve toutes les couleurs et les typographies qui représentent l'identité visuelle imaginée pour radio grenouille. Un prototype à également été réalisé sur figma pour simuler le parcours utilisateur.

Réalisations et extraits de code

Utilisation des composants

L'application est découpée sous forme de composants. Ils sont regroupés dans le dossier src/components. Nous avons privilégié une approche fonctionnelle avec l'utilisation des Hooks.

Pour ma part je me suis occupé de la partie inscription/connexion.
J'ai donc créé deux composants : un Login.js et un Register.js

Exemple du composant Login :

```
import React , {useState, useEffect} from "react";

import { SafeAreaView, StyleSheet, TextInput, Pressable, Text, View, ImageBackground} from "react-native";
```

On retrouve dans un premier temps les différents imports. Ici on importe le useState depuis React ainsi que tous les composants de react native pour construire notre vue.

```
const Login = () => {

  const [login, onChangeLogin] = useState(null);
  const [pass, onChangePass] = useState(null);
  const[message , setMessage] = useState('') ;
  const[style, setStyle] = useState(styles.text_error) ;
```

On déclare ensuite tous nos états, les données qui sont amenées à changer au cours de l'utilisation du composant. On crée donc notre variable ainsi que la fonction qui permet de la modifier. On définit ensuite sa valeur par défaut en paramètre du useState.

```
const checkLogin = () => {

  if(login !== null && pass !== null && login !== '' && pass !== '')
  {
    var myHeaders = new Headers();

    myHeaders.append("Content-Type", "application/json");

    var raw = JSON.stringify({
      "username": login,
      "password": pass,
    });

    var requestOptions = {
      method: 'POST',
      headers: myHeaders,
      body: raw,
      redirect: 'follow'
    };

    fetch("http://localhost:8080/api/login", requestOptions)
    .then(response => response.json())
    .then((result) => {
        if(result.error)
        {
          setMessage('Erreur lors de l\'autentification')
        }
        else {
          setStyle(styles.text_validate)
          setMessage('Connexion réussi !');
        }
    })
    .catch(error => console.log('error', error));
  }
  else {
    setMessage('Veuillez remplir tous les champs.') ;
  }
}
```

On crée une première fonction checkLogin() qui sera appelée à chaque validation du formulaire. Une première vérification est effectuée sur les champs vides.
On fait ensuite une requête vers la route api/login. Selon la réponse renvoyée on valide ou non la connexion.

Côté back end j'ai effectué un authentification json proposé par le bundle security de Symfony. Ainsi en faisant une requête vers la route /api/login un controller est appelé et renvoie une réponse en json.

```
class ApiLoginController extends AbstractController
{
    #[Route('/api/login', name: 'api_login')]
    public function index(#[CurrentUser] ?User $user): Response
    {

        if (null === $user) {
                return $this->json([
                    'message' => 'missing credentials',
                ], Response::HTTP_UNAUTHORIZED);
        }

        return $this->json([
            'user' => $user->getUserIdentifier(),
            'id' => $user->getId(),
            'roles' => $user->getRoles()
        ]);
    }
}
```

Les contraintes de validations des champs login et password on également été mise en place sur l'entité User pour permettre de sécuriser les champs côté back end.

On construit notre vue dans la fonction render en appelant tous les composants React Native que l'on a importé au-dessus. On vient également placer les valeurs de nos input et appeler nos fonctions selon les événements.

```
return (
  <SafeAreaView style={styles.global_container}>
    <ImageBackground source={require('../../assets/bg.svg')}
        resizeMode="cover"
        style={styles.image_bg}>
    <View style={styles.section_img}>
      <View style={styles.container_img}>
          <ImageBackground source={image}
          resizeMode="cover"
          style={styles.image}></ImageBackground>
      </View>
    </View>
    <View style={styles.form}>
      <TextInput
        style={styles.input}
        onChangeText={onChangeLogin}
        value={login}
        placeholder="Login"
      />
      <TextInput
        secureTextEntry={true}
        style={styles.input}
        onChangeText={onChangePass}
        value={pass}
        placeholder="Mot de passe"
      />
      <Pressable
        style={styles.button}
        onPress={() => {
          checkLogin(login,pass);
        }}>
        <Text style={styles.text_button}>Se connecter</Text>
      </Pressable>

      <Text style={style}> {message} </Text>
    </View>

    <Text style={styles.text_msg}>
      Pas de compte ? Inscrit toi ici !
    </Text>
    </ImageBackground>
  </SafeAreaView>
);
```

Dans React Native le style est un objet que l'on vient intégrer à nos composants.
A l'intérieur de celui-ci on peut créer de nouveaux objets et l'appliquer aux composants que l'on souhaite.
Ainsi on découpe le style en plusieurs parties dans un seul et même objet lié à notre composant principal.
Il est donc directement intégré à la suite du fichier.

```
const styles = StyleSheet.create({
  input: {
    height: 48,
    borderRadius : 8,
    marginVertical : 12,
    marginHorizontal: 20,
    borderWidth: 1,
    paddingVertical : 12,
    paddingHorizontal : 16,
    borderColor : '#D9D9D9',
    shadowColor: "#000",
    shadowOffset: {
        width: 0,
        height: 2,
    },
    shadowOpacity: 0.25,
    shadowRadius: 4.84,
    elevation: 5,
    backgroundColor : "white",
  },
```

Cycle de vie d'un composant :

Un composant React possède un cycle de vie. Il est divisé en quatre étapes :

Initialisation : C'est l'étape où le composant est construit avec les Props et l'état par défaut donnés. Cela se fait dans le constructeur d'une classe de composants.

Montage : le montage est l'étape de rendu du JSX renvoyé par la méthode de rendu elle-même.

Mise à jour : La mise à jour est l'étape où l'état d'un composant est mis à jour et l'application est repeinte.

Démontage : comme son nom l'indique, le démontage est l'étape finale du cycle de vie du composant où le composant est supprimé de la page.

React nous donne plusieurs méthodes au sein d'un composant permettant d'interagir à un moment précis de son cycle de vie.

La méthode **componentWillMount()** est appelé juste avant que le composant ne soit monté par le DOM, la méthode **componentWillUpdate()** est appelé lorsque le composant se reconstruit lors de la modification d'un state et la méthode **componentWillUnmount()** permet d'agir lorsque le composant est démonté du DOM, dès lors qu'il est supprimé de la page et marque la fin du cycle de vie.

Toutes ces méthodes sont utilisées avec React lorsque que le composant est sous forme de classe. Dans notre projet tous les composants ont une approche fonctionnelle, c'est là qu'intervient l'utilisation des Hooks.

Le hook *useEffect* remplace les méthode ci-dessus, c'est une méthode qui est appelée à chaque nouvel affichage de notre composant. On va donc généralement mettre à l'intérieur notre requête API pour récupérer les données et ainsi être sur des les avoir pour le rendu.

Le useEffect peut être appelé à chaque nouvel affichage, ou seulement lors du premier montage, ou selon un événement précis, comme un clic d'un bouton par exemple. On peut renseigner ça en ajoutant un second paramètre dans le useEffect.
On peut également retourner une méthode dans le *useEffect* pour effectuer le nettoyage du composant.

```
useEffect(()=>{
  fetchPrograms()
},[])
```

Dans le cas de l'image ci-dessus la fonction *fetchPrograms* représente la requete API pour récupérer tous les programmes. Elle sera appelée qu'un seule fois lors du premier montage du composant car on lui renseigne une tableau vide en paramètre (**[ ]**)

Le useState lui permet de déclarer tous les états et de créer des setters associés. Dès lors que l'on appelle un setter, le composants est directement mis à jour avec un nouvel affichage.

L'utilisation des hooks est donc très puissante car ils nous permettent de jouer avec les états et le cycle de vie du composant sans passer par un composant sous forme de classe. On a donc moins de ligne de code et plus de clarté. Le useEffect est directement déclaré dans le composant, ainsi il a directement accès à toutes les props.

Paramétrage de l'Api avec Api platform

Le framework API Platform nous permet de créer une API avec Symfony. Nous avons plusieurs possibilités pour paramétrer notre API. Tous les paramètres se font directement sur l'entité via l'annotation API Ressource. Automatiquement API Plateform nous génère un CRUD pour toutes les entités. Nous avons la possibilité de paramétrer ce que nous renvoie l'API sur plusieurs critères. Nous pouvons effectuer des opérations sur les items (user/{id} par exemple) et sur les collections (/users listes de tous les utilisateurs).

Concernant sur les données que nous renvoie l'API nous pouvons définir quels attributs recevoir en créant des groupes.

Il existe deux contextes pour la sérialisation des données : le context de normalisation et le context de dénormalisation.

Le contexte de normalisation prend en compte toutes les opérations ou l'on passe d'un objet vers du json. C'est la plupart du temps lorsque l'on souhaite récupérer de la données et la récupérer au format json côté front (GET).

A l'inverse le contexte de dénormalisation s'applique lorsque que l'on envoie des données aux formats json et qu'il est nécessaire de le transformer au format objet pour les faire persister en base de données (POST, PUT, PATCH).

Suivant ces différents contextes, les groupes et les choix des items ou collections on peut facilement définir quelles données envoyer ou récupérer selon les opérations de l'utilisateur.

J'ai également ajouter des filtres grace à l'annotation API filter, ce qui nous permet d'ajouter des paramètres dans la route et ainsi récupérer un résultat selon certaines conditions.

```
#[ApiFilter(SearchFilter::class, properties: ['id' => 'exact', 'programs.id' => 'exact'])]
```

```php
#[Groups(['read:User'])]
private $email;

#[ORM\Column(type: 'json')]
#[Groups(['read:User'])]
private $roles = [];

#[ORM\Column(type: 'string')]
#[Assert\NotBlank(
    message: 'Veuillez remplir ce champ.'
)]
private $password;

#[ORM\Column(type: 'string', length: 255)]
#[Assert\NotBlank(
    message: 'Veuillez remplir ce champ.'
)]
#[Groups(['read:User'])]
private $name;

#[ORM\Column(type: 'string', length: 255)]
#[Assert\NotBlank(
    message: 'Veuillez remplir ce champ.'
)]
#[Groups(['read:User'])]
private $surname;

#[ORM\ManyToMany(targetEntity: Program::class, mappedBy: 'users')]
#[ORM\JoinTable(name:'`favoriteBy`')]
#[Groups(['read:User'])]
private $programs;
```

On voit ici tous les attributs qui appartiennent au groupe read:User, ainsi si on récupère la liste des utilisateurs, le mot de passe ne sera pas retourné.

On peut également ajouter un nouvelle route pour une opération spécifique, en la reliant directement à un controller Symfony.
J'ai réalisé un système de like sur mon application, l'utilisateur à la possibilité d'ajouter en favoris un programme en particulier.
J'ai donc dans un premier temps créé mon controller et ma méthode fav.

```php
#[Route('/api/add' , name : 'fav')]
public function fav(Request $request,UserRepository $userRepository ,ProgramRepository $programRepository,  Mar

    $contentBody = $request->getContent();
    $body = json_decode($contentBody);

    $idProgram = $body->id_program ;
    $idUser = $body->id_user;

    $entityManager = $doctrine->getManager();

    $user = $userRepository->find($idUser) ;
    $program = $programRepository->find($idProgram);

    $user->addProgram($program);

    $entityManager->flush();

    return $this->json([
        'add' => 'Votre programme à bien été ajouter !',
    ]);
```

Je l'ajoute par la suite à mon API directement depuis l'entité User via l'annotation Api Ressource.

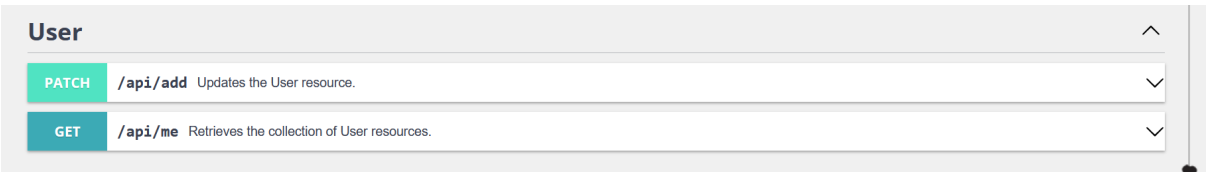
```php
#[ApiResource(
    collectionOperations: [
        "me" => [
            'pagination_enabled' => false,
            'path' => 'api/me',
            'method' => 'get',
            'controller' => MeController::class,
            'read' => false

        ],
    ],

    itemOperations: [
        "addFav" => [
            'controller' => AddFavController::class,
            'path' => '/api/add',
            'method' => 'patch',
            'read' => false
        ]
    ],
    normalizationContext: ['groups' => ['read:User']]


)]
```

Cela me permet également d'ajouter les différentes routes au niveau de Swagger.



Swagger

Nous avons utilisé Swagger pour documenter notre API. Il permet d'avoir la liste des différentes routes de notre application avec une description personnalisée. L'outil permet aussi de tester les différentes routes, et voir quels sont les paramètres attendus dans le body et la réponse retourner en json pour chacunes des routes. C'est donc une partie importante du projet car il permet à nous ou un nouveau développeur sur le projet, d'avoir une vision globale de l'API.

Sécurité de l'API :

Concernant la sécurité de l'API, il existe deux types principaux d'attaque potentielle :

- **Un déni de service** (Denial of Service ou DoS) est une attaque réalisée dans le but de rendre indisponible les services. En effet, une attaque DoS fonctionne en épuisant une ressource limitée dont une API a besoin pour répondre aux demandes légitimes. En inondant une API de fausses requêtes, ses ressources sont bloquées pour répondre à ces requêtes et pas aux autres.
- **Une attaque brute force**, c'est lorsque l'attaquant utilise des outils pour envoyer un flux continu de requêtes à une application ou une API – afin de tester toutes les combinaisons possibles d'un paramètre, via un processus d'essais et d'erreurs pour espérer deviner juste. Les objectifs peuvent être multiples : brute force d'un formulaire d'authentification dans le but de voler un compte, brute force d'un identifiant pour récupérer des données sensibles, brute force d'un secret, etc.

Pour pallier ces attaques, il convient d'implémenter **des mécanismes de Rate Limiting**. Ils permettent de protéger les APIs et les autres services contre une utilisation excessive et abusive, dans le but d'assurer leur disponibilité. Il s'agit d'anticiper le fait qu'un ou plusieurs client(s) – système – peuvent utiliser plus que leur « juste » part d'une ressource, via l'envoi de requêtes. Et en limitant le nombre de requêtes qu'un utilisateur donné est autorisé à envoyer dans un laps de temps défini, on peut réduire le risque d'attaque DoS ou d'attaque brute force.

Pour cela, il existe un bundle Symfony nommé "Rate Limiter" avec une configuration personnalisable du nombre de requêtes autorisé sur une période de temps avec trois stratégies différentes : Fixed Window Rate Limiter, Sliding Window Rate Limiter et Token Bucket Rate Limiter.

Exemple d’une recherche en anglais

Synergie family compte près de 300 animateurs qui travaillent dans plusieurs écoles de Marseille. Il propose des activités ludiques pour les enfants selon plusieurs périodes : le matin avant les cours, le midi et le soir.

Une application à été développée afin de gérer le temps de travail des animateurs. Elle permet de calculer le nombre d'heures effectuées par mois pour un animateur. Ces données sont ensuite exportées en csv, puis importées dans Silae, le logiciel de paiement de la boîte. Les équipes RH et Périscolaire sont les principaux utilisateurs de l’application.

L’application est développée en Angular côté front et Node.js avec le framework Nest.js côté backend accompagné d’une API.

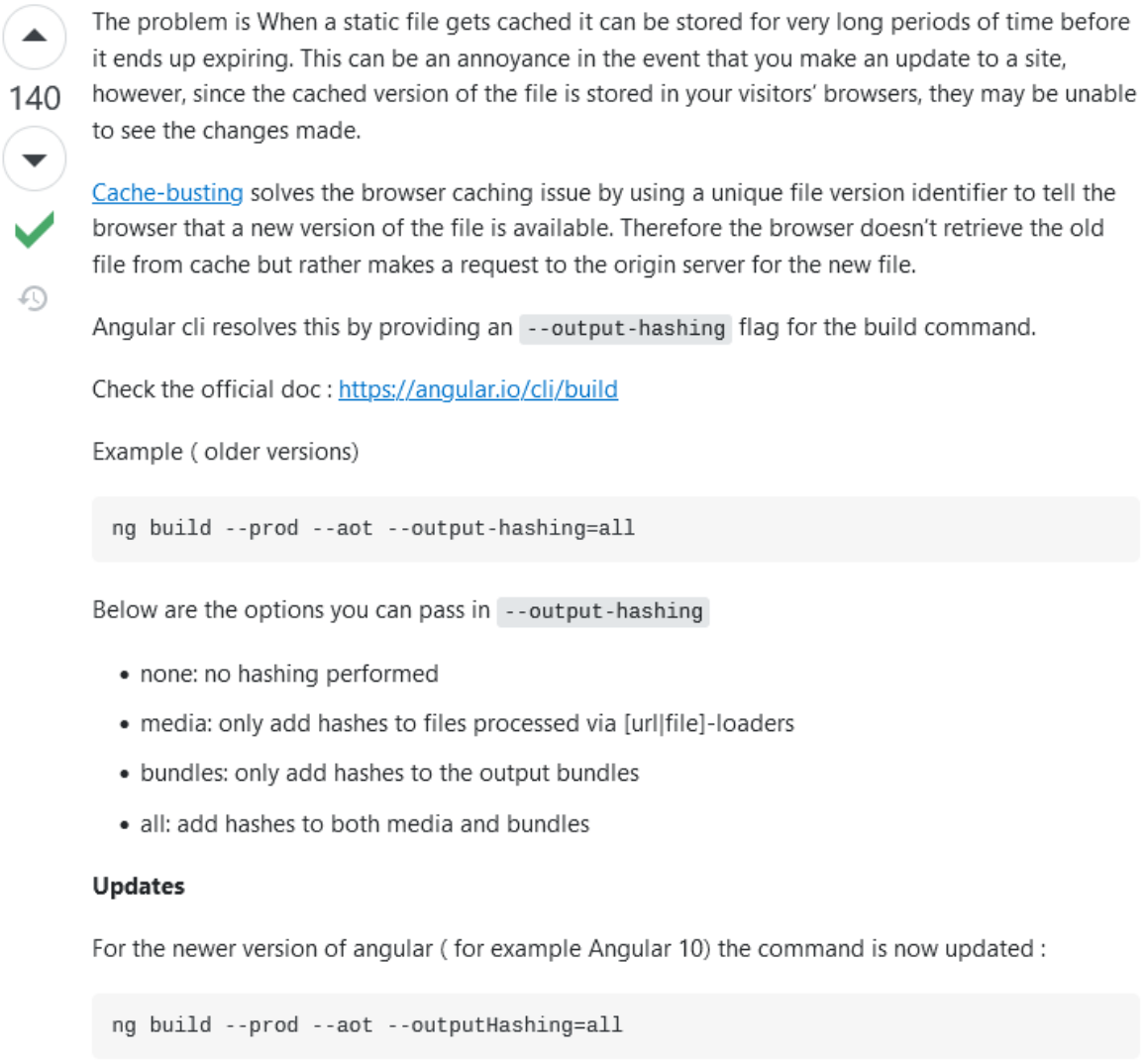
J’ai eu l’occasion de réaliser quelques tickets dessus.

Lors de chaque mise en production, les utilisateurs devaient de leur côté vider le cache. Un problème important car les personnes n'étant pas experts techniques, nous avons remarqué qu’il y avait des problèmes sur l’application et que certaines fonctionnalités ajouter n’était pas présentes. En vidant le cache, cela résout le problème, mais ça reste problématique pour les prochaines mises en production.

J’ai donc taper la recherche suivante “*How can clear cache after new deployment angular*” ?

J'atterris sur les premier lien stack overflow, où la personne rencontres des problèmes très similaires :

Il explique ici que lorsque les développeurs front déploie une nouvelle feature sur la connexion, les utilisateurs sont obligés de passer en navigation privée ou de vider le cache manuellement. Pourquoi certains changements ne sont pas mis à jour automatiquement, qu'est ce qui les différencie d'une autre feature particulière ?



140

The problem is When a static file gets cached it can be stored for very long periods of time before it ends up expiring. This can be an annoyance in the event that you make an update to a site, however, since the cached version of the file is stored in your visitors' browsers, they may be unable to see the changes made.

Cache-busting solves the browser caching issue by using a unique file version identifier to tell the browser that a new version of the file is available. Therefore the browser doesn't retrieve the old file from cache but rather makes a request to the origin server for the new file.

Angular cli resolves this by providing an `--output-hashing` flag for the build command.

Check the official doc : https://angular.io/cli/build

Example (older versions)

```
ng build --prod --aot --output-hashing=all
```

Below are the options you can pass in `--output-hashing`

- none: no hashing performed
- media: only add hashes to files processed via [url|file]-loaders
- bundles: only add hashes to the output bundles
- all: add hashes to both media and bundles

Updates

For the newer version of angular (for example Angular 10) the command is now updated :

```
ng build --prod --aot --outputHashing=all
```

La personne qui répond explique qu'un fichier mis en cache peut rester très longtemps avant d'expirer. Ainsi les utilisateurs garderont dans leurs navigateur l'ancienne version des fichiers mis en cache.

Il explique que pour palier à ce problème il est possible de donner une version à ces fichiers et que par conséquent le navigateur ira toujours chercher le dernier dossier mis à jour.

Il faut donc ajouter le `--output-hashing=all` lors du build.

Une autre personne ajoute aussi que l'on doit configurer notre angular.json.

While the accepted answer above will work, **this adjustment should be made in `angular.json`**, under `configurations` => `<my-env-name>` => `outputHashing` => set to `all` (for production environments).

Simplified example:

```
{
  "projects": {
    "<my-project>": {
      "architect": {
        "build": {
          "configurations": {
            "<my-env-name>": {
              "outputHashing": "all"
            }
          }
        }
      }
    }
  }
}
```

Je vais donc directement regarder mon fichier angular.json à l'intérieur du projet.

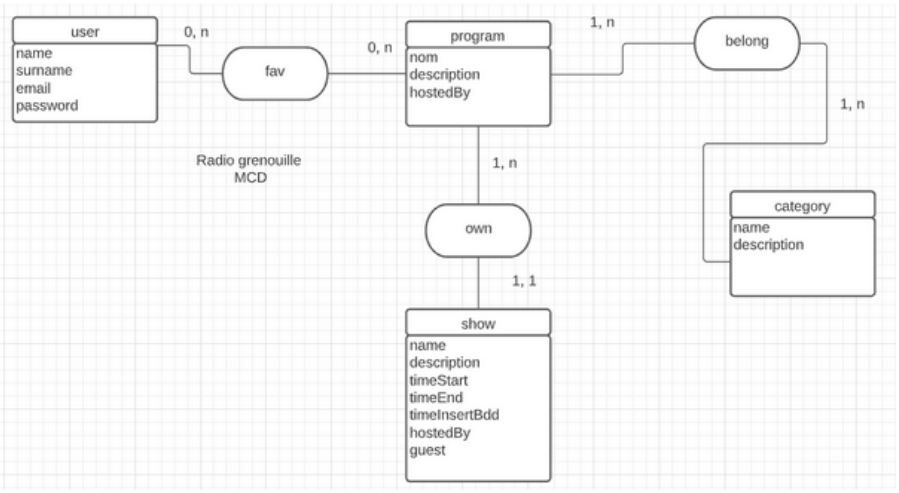
Je remarque alors que tout était déjà bien configuré, seulement l'option `output-hashing=all` étant présente seulement lorsque qu'on exécute la commande *npm run build:prod. (équivalent ng build –prod)*

Toutes les mises en prod jusqu'à maintenant étant effectuées avec un *npm run build* . Les fichiers sur la production n'avaient donc pas de versions.

Depuis nous effectuons la bonne commande et les fichiers sont versionnés.

Annexes :

MCD RADIO GRENOUILLE



MPD RADIO GRENOUILLE