

# Documentation

# Castopod

- [Présentation du projet Castopod](#)
- [Installation avec Docker](#)
- [Présentation Codelgniter](#)
- [Organisation des fichiers](#)
- [Configuration de Codelgniter](#)
- [Présentation des outils externes](#)
- [Fonctionnement de Castopod](#)
  - [Routes](#)
  - [Contrôleurs](#)
  - [Vues](#)
  - [Fonctionnalités supplémentaires](#)
  - [Authentifications et autorisations](#)
  - [Fichiers](#)
  - [Podcast et Épisodes](#)
- [Ajout de Polytech Nantes](#)
  - [Création d'un flux général pour une instance Castopod](#)
  - [Connexion à Nextcloud](#)
  - [Insertion d'image flexible](#)
  - [Sondage](#)



# Présentation du projet

## Castopod

**Castopod** est une plateforme d'hébergement de podcasts unique en son genre, car elle est open-source, auto-hébergée et décentralisée. Développée par **Ad Aures** en mars 2019, elle offre aux utilisateurs un contrôle total sur leurs données et leur contenu.

## Fonctionnalité principale

Castopod est une plateforme de publication de podcasts qui offre la possibilité de créer et diffuser des séries audio, découpées en épisodes. Cette plateforme permet la **monétisation** des podcasts afin de rémunérer les créateurs et les hébergeurs. Castopod se distingue par son intégration au réseau social **Fediverse**, favorisant une communauté active et engagée. La plateforme dispose d'un **système de gestion des utilisateurs** avec différents rôles, et offre des **statistiques d'écoute** détaillées pour suivre l'audience des podcasts..

Techniquement, Castopod est une plateforme développée en **PHP 8.1**, utilisant le framework **CodeIgniter 4** pour une architecture robuste et flexible. La base de données est gérée par **MySQL**, assurant un stockage fiable et performant des contenus. Vous pouvez retrouver plus d'informations sur le fonctionnement de CodeIgniter sur [la page de documentation](#) dédiée.

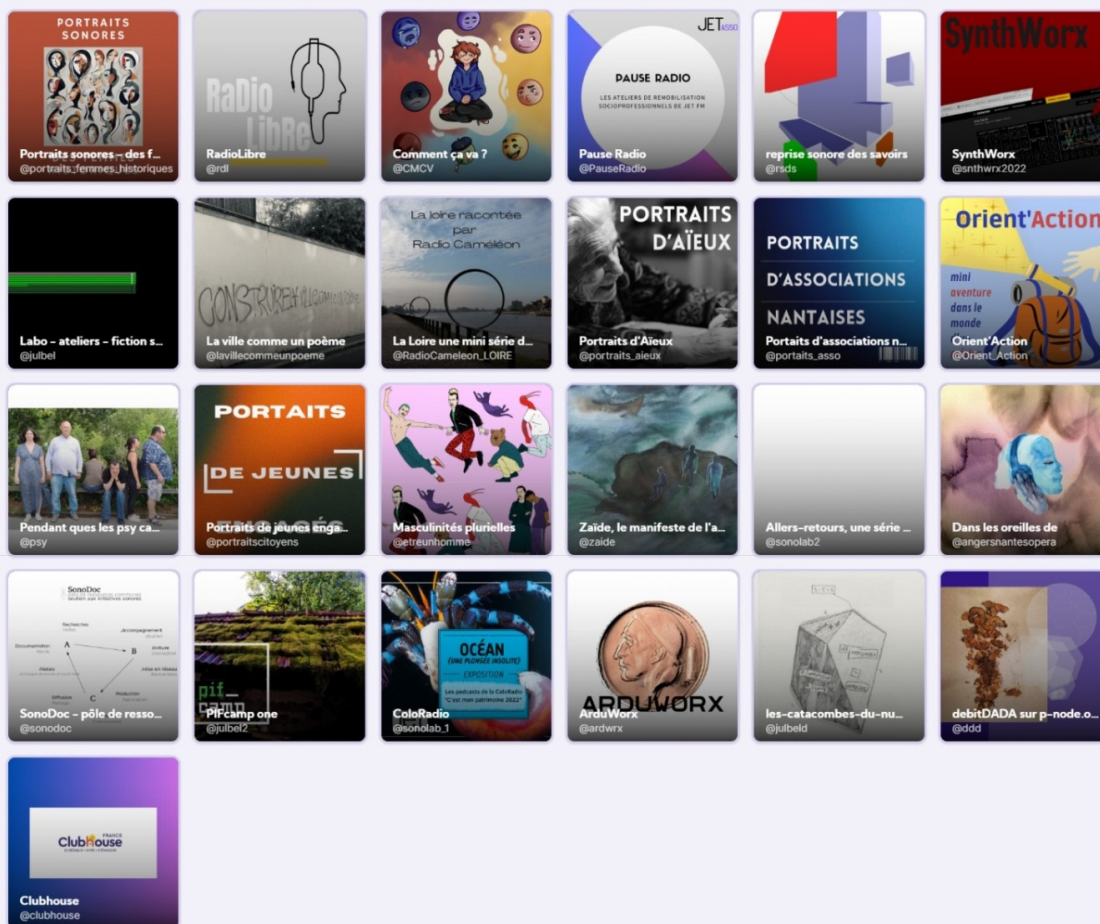
## Organisation de Castopod

**Page d'accueil** : Castopod présente une page d'accueil qui regroupe tous les podcasts publiés sur le serveur. Prenons l'exemple du [Castopod](#) de JetFM.



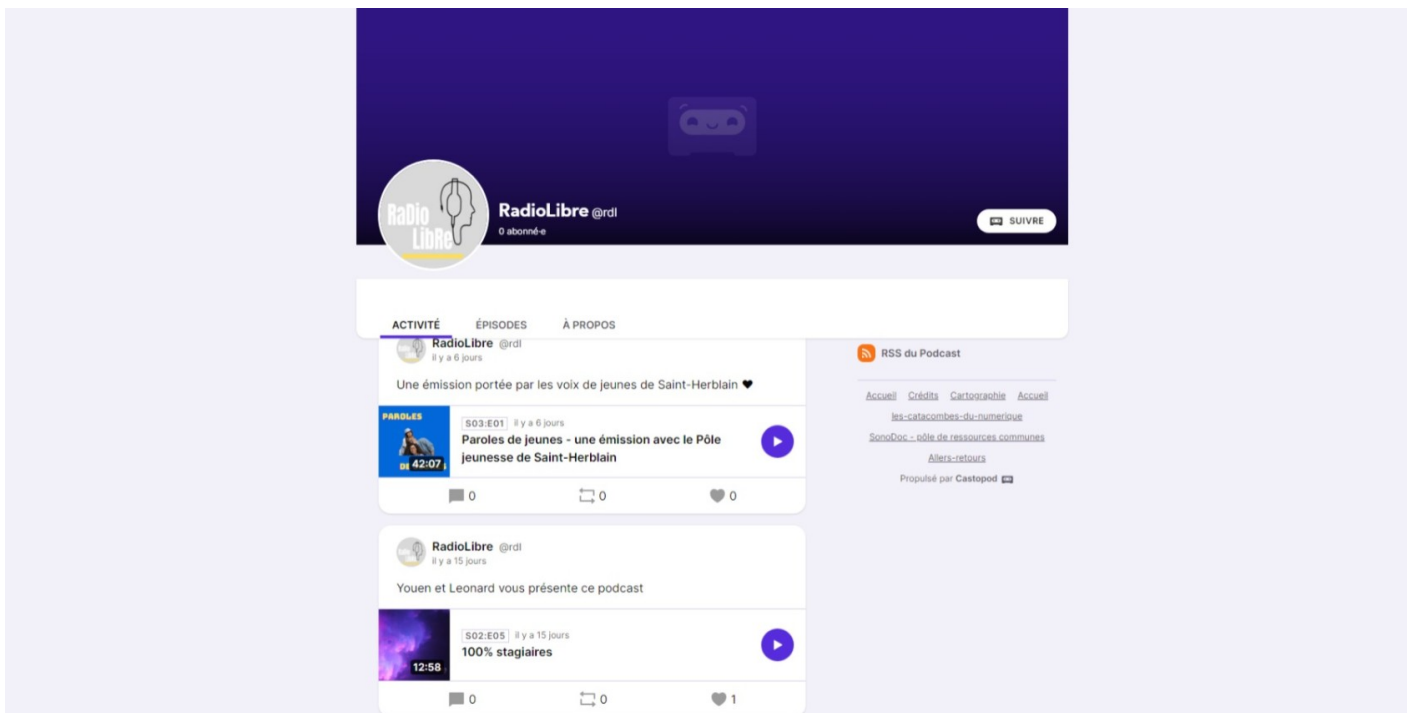
Tous les podcasts (25)

Trier par

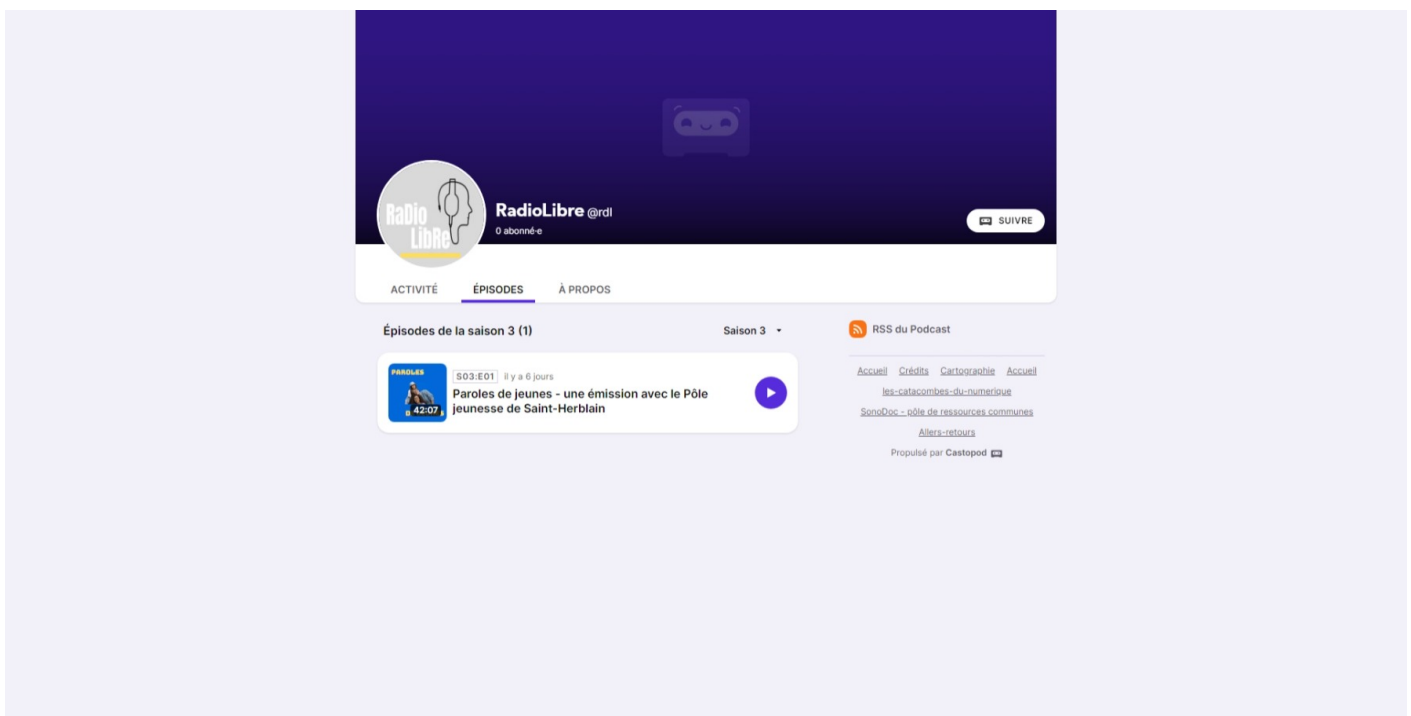


**Page d'un podcast :** En sélectionnant un podcast, vous accédez à sa page dédiée. C'est là que vous pouvez choisir l'épisode que vous souhaitez écouter. Vous pouvez retrouver ci-dessous une page d'exemple pour un podcast hébergé par JetFM.





**Épisode :** Les épisodes peuvent être organisés en saisons pour une meilleure structuration. Les fichiers audio des épisodes sont hébergés sur le serveur de JetFM.



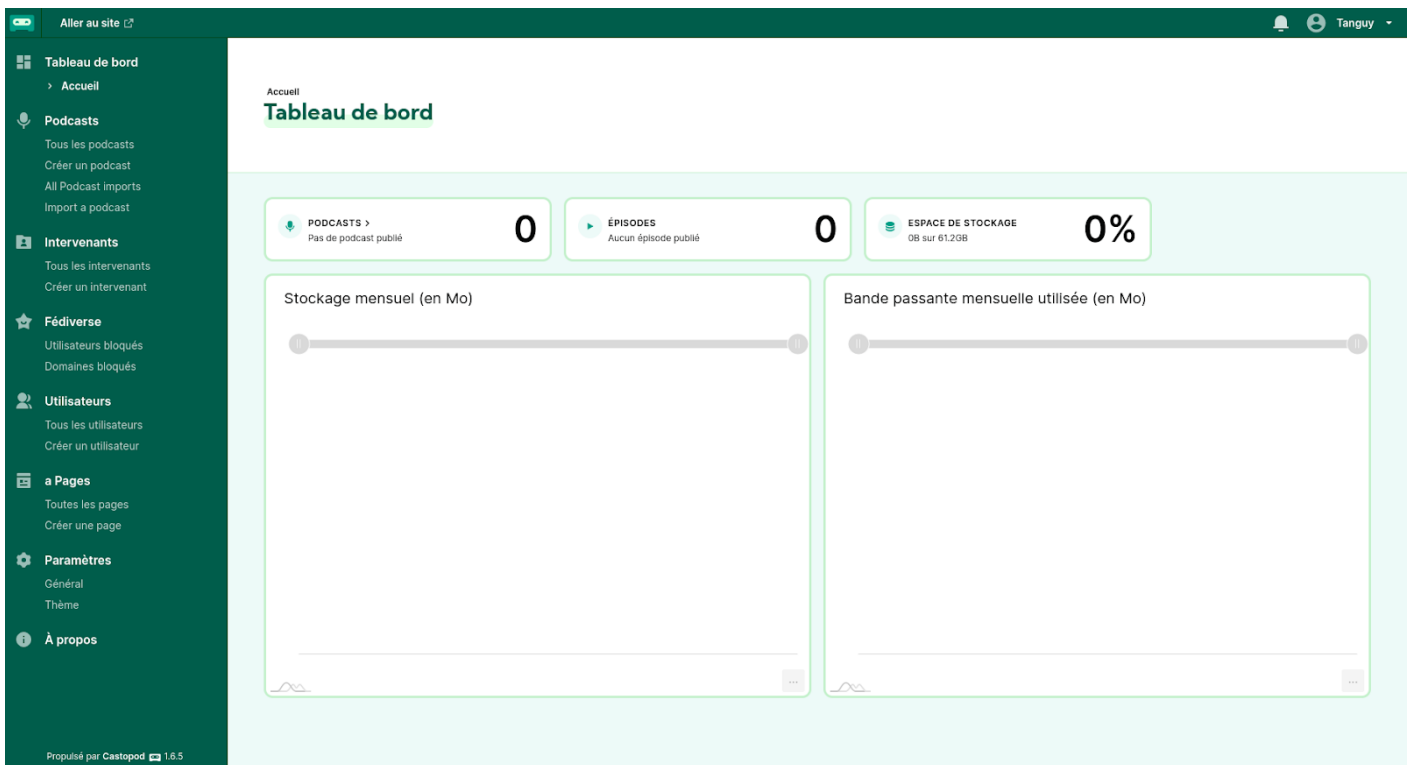
# Gestion de podcast

Pour la gestion des podcasts, Castopod propose une interface utilisateur regroupant la plupart des informations et des fonctions importantes pour la gestion. On peut y retrouver :

- Les **statistiques** sur les podcasts publiés



- La **gestion** des différents utilisateurs du site
- La possibilité d'**importer** et d'**exporter** des podcasts
- Des **informations** sur la machine qui **héberge** l'outil



La page d'accueil de Castopod est accessible à tous, permettant l'écoute des podcasts publiés. Cependant, la création et la modification de podcasts sont soumises à un système de rôles. Deux niveaux de permissions existent :

- Pour la gestion du **site** :
  - **Super administrateur** : Contrôle total sur Castopod.
  - **Manager** : Modification du contenu de Castopod.
  - **Podcasteur** : Utilisateur général de Castopod.
- Pour la gestion d'un **podcast** :
  - **Administrateur** : Contrôle total du podcast.
  - **Éditeur** : Gestion du contenu et des publications du podcast.
  - **Auteur** : Gestion du contenu, mais pas de la publication.
  - **Invité** : Auditeur général du podcast.

Ce système permet de contrôler finement les accès et les permissions sur Castopod, garantissant la sécurité et la confidentialité des contenus.

## Sources

- [Site de Castopod](#)
- [Wikipédia de Castopod](#)



- [Page Castopod de JetFm](#)



# Installation avec Docker

## Instructions de Configuration

### 1. Prérequis

Pour commencer, assurez-vous d'avoir **Docker** installé sur votre système.

Vous n'avez pas besoin de connaissances préalables sur Docker pour suivre les prochaines étapes. Cependant, si vous souhaitez utiliser votre propre environnement, n'hésitez pas à le faire !

Ensuite, **clonez** le projet Castopod en utilisant la commande suivante :

```
git clone https://code.castopod.org/adaures/castopod.git
```

Créez un fichier `.env` avec la configuration minimale requise pour connecter l'application à la base de données et utiliser Redis comme gestionnaire de cache :

```
CI_ENVIRONMENT="development"
# Si défini sur développement, vous devez exécuter `npm run dev` pour démarrer le serveur d'assets statiques
vite.environment="development"

# Par défaut, cela est défini sur vrai dans la configuration de l'application.
# Pour le développement, cela doit être défini sur faux car c'est
# dans un environnement local
app.forceGlobalSecureRequests=false

app.baseURL="http://localhost:8080/"
media.baseURL="http://localhost:8080/"

admin.gateway="cp-admin"
auth.gateway="cp-auth"

database.default.hostname="mariadb"
database.default.database="castopod"
```



```
database.default.username="castopod"
```

```
database.default.password="castopod"
```

```
cache.handler="redis"
```

```
cache.redis.host = "redis"
```

```
# Vous ne souhaitez peut-être pas utiliser Redis comme gestionnaire de cache
```

```
# Commentez/supprimez les deux lignes ci-dessus et décommentez
```

```
# la ligne suivante pour le cache par fichier.
```

```
#cache.handler="file"
```

N'oubliez pas d'ajouter le référentiel que vous avez cloné à Docker Desktop dans Paramètres > Ressources > Partage de fichiers.

## 2. Développement à l'intérieur du Conteneur d'Application avec VSCode (Recommandé)

Si vous utilisez VSCode, vous pouvez configurer un conteneur de développement préconfiguré.

Pour ce faire, installez l'extension [VSCode Remote - Containers](#)

et exécutez la commande suivante :

```
Ctrl/Cmd + Shift + P > Ouvrir dans le conteneur
```

La fenêtre de VSCode se rechargera à l'intérieur du conteneur de développement. Attendez plusieurs minutes lors du premier chargement car il construit tous les services nécessaires.

Le conteneur de développement démarrera en exécutant le serveur PHP de Castopod. Pendant le développement, vous devrez démarrer le serveur de développement de [Vite](#) pour compiler le code TypeScript et les styles :

```
# exécutez le serveur de développement de Vite
```

```
npm run dev
```

Si le serveur PHP de Castopod ne fonctionne pas, vous pouvez le redémarrer en utilisant les commandes suivantes :

```
# exécutez le serveur Castopod
```

```
php spark serve - 0.0.0.0
```



Vous êtes maintenant prêt ! ☑

Vous êtes maintenant à l'intérieur du conteneur de développement, vous pouvez utiliser la console VSCode (Terminal > Nouveau terminal) pour exécuter n'importe quelle commande :

```
php -v

# Composer est installé
composer -V

# pnpm est installé
pnpm -v

# git est installé
git version
```

Pour plus d'informations, consultez les Conteneurs distants VSCode.

## 2-bis. Développement Hors du Conteneur d'Application(sans VScode)

Si vous préférez ne pas utiliser le devcontainer de VSCode, vous pouvez démarrer les conteneurs Docker manuellement. Accédez au dossier racine du projet et exécutez les commandes suivantes :

```
# démarre tous les services déclarés dans le fichier docker-compose.yml
# l'option -d démarre les conteneurs en arrière-plan
docker-compose up -d

# Voir tous les processus en cours d'exécution (vous devriez voir 3 processus en cours d'exécution)
docker-compose ps

# Alternativement, vous pouvez vérifier tous les processus Docker
docker ps -a
```

La commande `docker-compose up -d` démarrera 4 conteneurs en arrière-plan :

- `castopod_app` : un conteneur basé sur PHP avec les exigences de Castopod installées
- `castopod_redis` : une base de données Redis pour gérer les requêtes et le cache des pages



- `castopod_mariadb` : un serveur MariaDB pour les données persistantes
- `castopod_phpmyadmin` : un serveur phpMyAdmin pour visualiser la base de données MariaDB.

Exécutez n'importe quelle commande à l'intérieur des conteneurs en les préfixant par `docker-compose run --rm app` :

```
# use PHP
docker-compose run --rm app php -v

# use Composer
docker-compose run --rm app composer -V

# use pnpm
docker-compose run --rm app pnpm -v

# use git
docker-compose run --rm app git version
```

## 3. Commencez à Développer

Vous êtes maintenant prêt à commencer le développement. Utilisez la console VSCode pour exécuter des commandes telles que `php -v`, `composer -V`, `npm -v`, ou `git version` selon vos besoins.

Pour voir vos modifications, rendez-vous sur :

- `http://localhost:8080/` pour le site web Castopod
- `http://localhost:8080/cp-admin` pour l'interface d'administration de Castopod :
  - Adresse e-mail : **admin@castopod.local**
  - Mot de passe : **castopod**
- `http://localhost:8888/` pour l'interface de phpMyAdmin :
  - Nom d'utilisateur : **castopod**
  - Mot de passe : **castopod**

Pour des information complémentaire visiter la [documentation Castopod](#)



# Présentation CodeIgniter

## Histoire de CodeIgniter

### **2006 : Création du framework**

Créé par Rick Ellis, PDG d'EllisLab, CodeIgniter est né le 28 février 2006. Issu d'ExpressionEngine, un CMS développé par la même société, CodeIgniter visait à offrir une alternative simple et performante pour le développement d'applications web.

### **2006-2019 : Mise à jour fréquente de CodeIgniter**

Au fil des années, CodeIgniter a connu une adoption rapide et une popularité croissante au sein de la communauté PHP. Sa simplicité d'utilisation, sa documentation claire et sa large communauté de contributeurs ont largement contribué à son succès.

### **Versions majeures et évolutions notables :**

- **2007** : Version 1.0, introduction du pattern MVC et de la bibliothèque de base de données
- **2008** : Version 1.5, ajout de la gestion des sessions, des helpers et de la validation de formulaires
- **2011** : Version 2.0, refonte majeure du framework avec l'intégration de l'autoload, des hooks et des tests unitaires
- **2012-2019** : Versions 2.1 à 3.1, enrichissement continu avec de nouvelles fonctionnalités et améliorations de performance

### **2020 : Nouvelle ère avec CodeIgniter 4**

Sortie le 24 février 2020, cette version majeure représente une refonte complète du framework. CodeIgniter 4 s'appuie sur les principes fondamentaux de ses prédécesseurs tout en intégrant des technologies modernes et des pratiques de développement actuelles.

### **Points clés de la version 4 :**

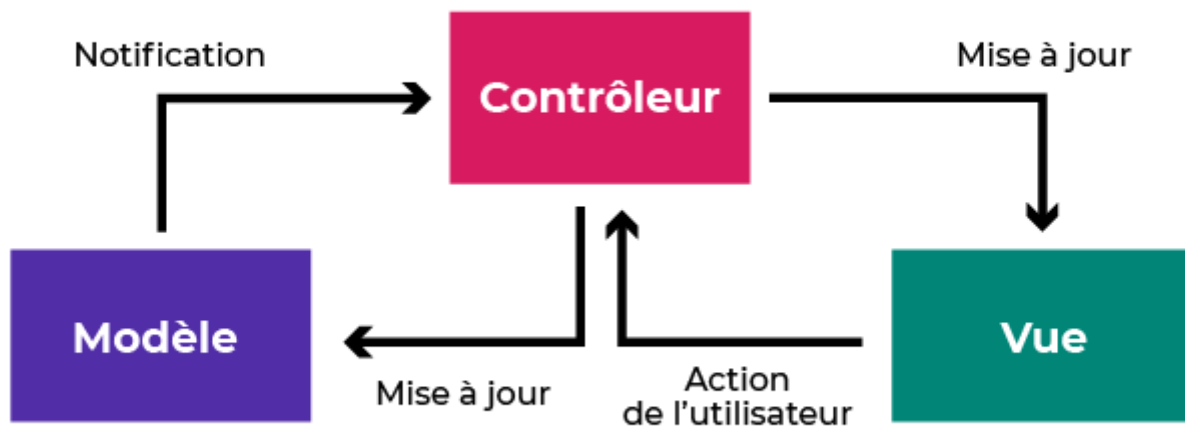
- Architecture MVC améliorée: meilleure séparation des couches et structure plus flexible
- Performances optimisées: code plus léger et utilisation efficace des ressources
- Nouvelles fonctionnalités: support de PSR-4, gestion des routes améliorée, sécurité renforcée
- Compatibilité PHP 7+: utilisation des dernières fonctionnalités du langage PHP



C'est justement cette version 4 de CodeIgniter qui est utilisé pour le développement de la plateforme Castopod.

# Présentation de l'architecture MVC

L'architecture MVC (Model-View-Controller) est un modèle de conception logicielle largement utilisé pour le développement d'interfaces utilisateur. Ce modèle vise à séparer les différentes préoccupations d'une application en trois composants distincts :



## 1. Modèle (Model) :

Le modèle représente la couche métier de l'application. Il encapsule la logique métier, les données et les règles de fonctionnement. Le modèle est responsable de la gestion des données, des calculs et de la validation des informations.

## 2. Vue (View) :

La vue est la couche de présentation de l'application. Elle est responsable de l'affichage des données et de l'interaction avec l'utilisateur. La vue utilise les données du modèle pour générer l'interface utilisateur et reçoit les interactions de l'utilisateur pour les transmettre au contrôleur.

## 3. Contrôleur (Controller) :

Le contrôleur agit comme un intermédiaire entre le modèle et la vue. Il reçoit les requêtes de l'utilisateur, les traite en utilisant le modèle et met à jour la vue en conséquence. Le contrôleur est responsable de la gestion du flux de l'application et de la coordination des interactions entre le modèle et la vue.



# Organisation d'un projet Codeigniter

## Dossier principal :

Le point d'entrée de votre application Codeigniter se trouve dans le dossier principal. Il contient les fichiers suivants :

- `index.php` : Démarre l'application et charge le framework Codeigniter.
- `config.php` : Contient les configurations de base de l'application (par exemple, la base de données, les routes).
- `autoload.php` : Détermine les classes et les helpers à charger automatiquement.

## Structure MVC :

- **Dossier app**: Contient les différents composants de l'architecture MVC :
  - `Controllers` : Contient les contrôleurs de l'application. Chaque contrôleur est responsable d'une partie de la logique métier et de l'interaction avec l'utilisateur.
  - `Models` : Contient les modèles de l'application. Chaque modèle représente une entité métier et encapsule la logique d'accès aux données.
  - `Views` : Contient les vues de l'application. Chaque vue est responsable de l'affichage d'une partie de l'interface utilisateur.

## Fichiers importants :

- `Routes.php` : Définit les règles de routage des URL vers les contrôleurs et les actions.
- `Application.php` : Classe principale de l'application qui gère le flux de l'application.
- `Controller.php` : Classe de base pour tous les contrôleurs.
- `Model.php` : Classe de base pour tous les modèles.
- `View.php` : Classe de base pour toutes les vues.

## Exemple de structure d'un projet basique :

```
app/  
├─ Controllers/  
│   └─ Home.php  
├─ Models/  
│   └─ User.php  
└─ Views/  
    ├─ home/  
    │   └─ index.php  
    └─ user/  
        └─ profile.php  
system/
```



# CodeIgniter et Castopod

Vous pouvez retrouver une explication détaillée de l'organisation des fichiers pour Castopod dans la section [Organisation des fichiers](#). Cette section explique comment les fichiers sont organisés selon l'architecture MVC pour le développement de Castopod.

## Source

- [Documentation officielle de CodeIgniter](#)
- [Page Wikipédia de CodeIgniter](#)



# Organisation des fichiers

Castopod s'appuie sur le framework CodeIgniter, qui possède un modèle MVC consistant à diviser les différentes parties d'un projet sous trois formes :

- Les vues
- Les contrôleurs
- Les modèles

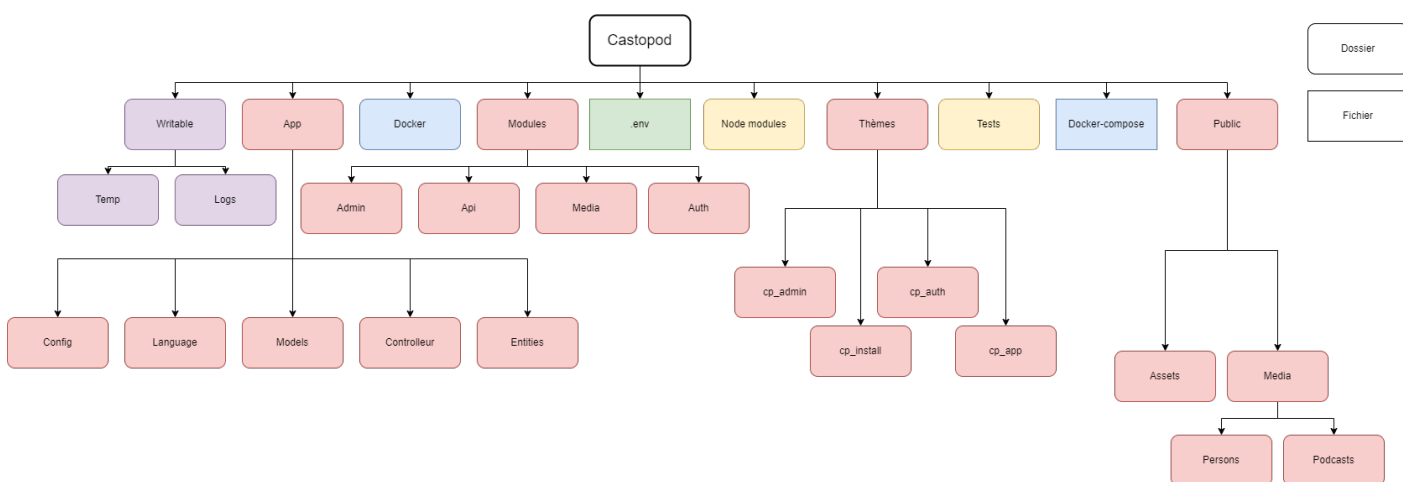
Il existe également les entités qui sont des classes permettant de manipuler plus facilement des objets comme des utilisateurs ou des fichiers.

Pour en savoir plus sur ce modèle, nous vous invitons à vous diriger vers [la page correspondante](#) à l'explication de CodeIgniter ainsi qu'à la [documentation officielle](#).

Une fois ce modèle bien compris, nous pouvons donc analyser la structure de Castopod, qui possède ces différents éléments. Ici, nous nous concentrons sur l'organisation des fichiers en [mode développement](#), nous ne nous attardons pas sur les fichiers en mode production qui peuvent présenter des différences.

## Structure générale

Voici la structure générale du projet Castopod, qui sera détaillé plus en profondeur ensuite :



Vous remarquerez ici que nous nous sommes concentré sur une partie des fichiers, et que nous n'allons pas décrire l'ensemble des fichiers/dossiers. Beaucoup de dossiers sont liés à CodeIgniter ou à Git, mais ils n'ont normalement pas besoin d'être configuré. Cependant, si vous souhaitez

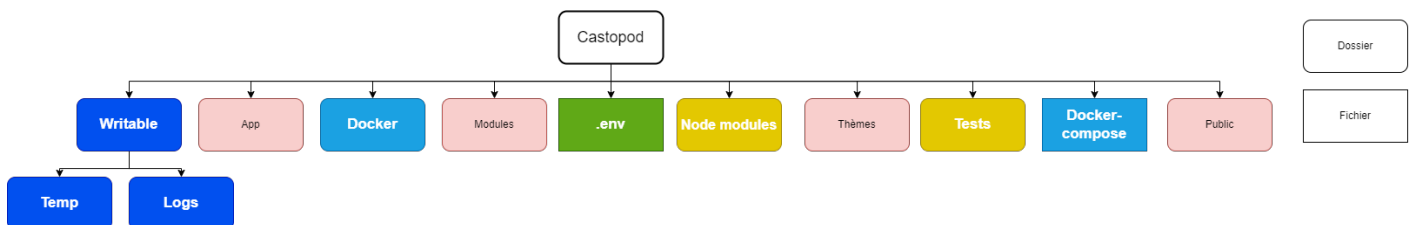


malgré tout vous pencher sur la configuration de Codelgniter, nous vous invitons à accéder [au chapitre expliquant la configuration du projet](#).

De plus, la description des dossiers et fichiers peuvent changer en fonction des versions du projet, nous nous appuyons ici sur la version 1.9.0 du 31 janvier 2024.

# Configuration et mise en place du projet

Nous allons décrire dans cette partie les dossiers et fichiers permettant l'installation et le paramétrage du projet Castopod.



Ces différents éléments permettent au projet de fonctionner selon l'envie de l'utilisateur, mais peuvent tout à fait être laissé par défaut si le développeur souhaite travailler uniquement sur les pages Web du projet.

## Docker

Pour gérer Docker, Castopod s'appuie sur des fichiers décrivant les différents conteneurs à construire pour gérer le projet.

On a un fichier nommé *docker-compose.yml* qui décrit les conteneurs à construire pour le projet, qui sont au nombre de 5 :

- Le conteneur Castopod : Il va contenir l'ensemble des fichiers de Codelgniter qui vont permettre de gérer le front-end et le back-end du projet
- Le conteneur Redis : C'est un système de gestion de base de données. Il stocke les données en mémoire vive, ce qui le rend extrêmement rapide pour les opérations de lecture et d'écriture.
- Le conteneur MariaDB : On retrouve la base de données générale du projet. Contrairement à Redis, elle est utilisée pour gérer des ensembles de données complexes avec des relations entre les tables.
- Le conteneur PHP My Admin : Ce conteneur va permettre d'avoir une gestion graphique de la base de données, en accédant à la page *localhost:8888* (modifiable dans le *.env* du



projet)

- Le conteneur S3 Mock : C'est un outil qui simule le service de stockage d'objets d'Amazon Web Services (AWS) appelé Amazon S3

Ce fichier va permettre de gérer les ports sur lequel les conteneurs vont être reliés à la machine hôte, de gérer différents paramètres des applications, les commandes qui vont être lancés au démarrage et le nom des volumes.

On a également un dossier Docker, qui contient un fichier dans le sous-dossier développement, qui va décrire la construction du conteneur de Castopod.

Pour en savoir plus sur la gestion des fichiers Docker, nous vous invitons à lire [la documentation officielle de Docker](#), qui contiendra plus d'informations sur le déploiement des conteneurs.

## Composer

Composer est un logiciel gestionnaire de dépendances libre écrit en PHP. Castopod s'appuyant sur de nombreuses bibliothèques pour fonctionner, Composer lui permet lors de son déploiement d'installer ces bibliothèques.

Pour cela, les bibliothèques PHP nécessaire sà Castopod sont listés dans un fichier nommé *composer.json*, qui indique également les versions à installer de ces bibliothèques, et qui créé des alias pour des commandes liées à ces bibliothèques dans la partie *scripts* du fichier.

Ces bibliothèques seront ensuite installées dans le dossier **vendor** une fois cette commande utilisée :

```
composer install
```

Encore une fois, pour avoir de plus ample informations, nous vous renvoyons vers [la documentation de Composer](#) qui vous décrira son fonctionnement.

## Node

Node est, à l'instar de Composer, un gestionnaire de paquet mais en javascript, qui permet l'installation de bibliothèques en JS lors du déploiement du projet, ainsi que la gestion côté serveur.

L'ensemble de ces paquets sont détaillés dans le fichier *package.json*, ainsi que les alias qui pourront être utilisé via [npm](#). Il permet par exemple l'utilisation de Vite, une bibliothèque permettant le développement en local de site internet.



Le dossier **node\_modules** contient l'ensemble des bibliothèques JS installées, qui peuvent être téléchargées et installées via la commande :

```
npm install
```

## Fichier *.env*

Ce fichier va décrire la manière donc le projet doit fonctionner. C'est un fichier phare du projet, qui est directement lié à CodeIgniter car c'est un fichier que l'on retrouve sur les projets utilisant ce framework.

Dans ce fichier, vous allez pouvoir indiquer des paramètres pour le fonctionnement de Castopod, comme :

- Le type de déploiement (production ou développement, ce qui va jouer les performances du projet)
- L'URL du projet (celle de l'application, et celle où les médias seront enregistrés)
- Les informations de la base de données
- Le préfix pour les enregistrement dans la base de données
- Les identifiants SMTP pour la gestion des emails
- La gestion d'une [API REST](#) sur Castopod
- Des configurations pour les conteneurs du projet, tels que S3 ou Redis

Si vous souhaitez avoir plus d'informations sur ce fichier, nous vous invitons à vous diriger vers [la documentation de CodeIgniter](#), qui décrit plus en profondeur les possibilités de ce fichier.

## Tests

Dans ce dossier, vous allez retrouver les différents tests qui seront effectués durant le développement du projet. Pour cela, Castopod s'appuie sur la bibliothèque [PHP Unit](#) qui permet d'effectuer des tests unitaires sur des projets PHP.

Dans ce dossier, vous pouvez configurer votre propre test dans ce dossier, et des tests sont déjà effectués par défaut par PHP Unit. Le fichier *README* du dossier vous explique les étapes à effectuer pour lancer les tests, ou pour créer les votre directement.

Il existe deux manières de lancer ces tests, selon votre système d'exploitation :

```
./phpunit # Sur Linux et Mac  
vendor\bin\phpunit # Sur Windows
```



Et vous pouvez spécifier un dossier particulier pour lancer les tests, en rajoutant en paramètre de la commande l'emplacement du dossier.

## Writable

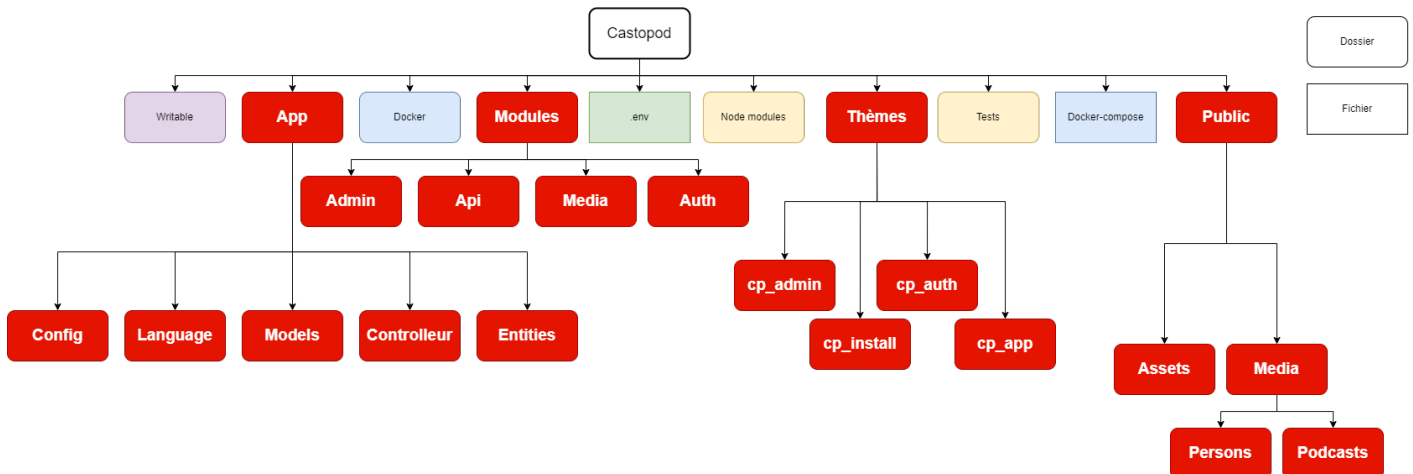
Ce dossier permet d'écrire temporairement des données sur le conteneur Castopod et d'accéder à des logs.

On retrouve dedans un dossier **temp**, qui peut être utilisé pour stocker des fichiers temporairement, ainsi qu'un dossier uploads qui lui peut être utilisé pour un stockage plus long. Attention, ce n'est cependant pas ici que Castopod enregistre les médias liés aux podcasts ou aux utilisateurs. Eux sont enregistrés dans le dossier **public** dont nous parlerons plus loin.

Enfin, on retrouve également des logs, dans le dossier **logs**, ainsi que des logs des sessions où l'on va retrouver les variables des différentes sessions dans le dossier **session**.

## Développement de Castopod

Nous allons maintenant vous décrire l'architecture des dossiers et fichiers qui permettent d'éditer le front-end ainsi que le back-end des différentes pages de Castopod.



Comme expliqué au départ, vous allez retrouver la structure MVC dans cette organisation. Ainsi, si vous n'êtes pas sûr d'avoir bien compris en quoi cela consistait, nous vous invitons à aller sur [Présentation CodeIgniter](#), la page décrivant ce modèle.

## Public

Le dossier **public** est utilisé pour stocker les fichiers audios et les images utilisés par Castopod.



Dans le dossier général, on retrouve les bannières ainsi que l'icone de Castopod, et un filtre robots pour l'indexation sur des sites externes (dans le fichier *robots.txt*).

Dans le fichier *.well-know/GDPR.yml*, on retrouve une description de [la politique RGPD](#) utilisée dans Castopod.

Dans **assets**, on retrouve différents fichiers svg pouvant être utilisés par les pages de Castopod.

Et enfin, dans **media**, on retrouve les fichiers médias utilisés pour la gestion des utilisateurs et des podcasts. Dans **podcasts**, on va par exemple retrouver les fichiers audios liés aux épisodes, et les illustrations du podcast et des épisodes.

# Thèmes

Dans le dossier **themes**, on va retrouver les vues réparties en 4 catégories :

- **cp\_admin** : Les pages accessibles depuis la page d'administration. Par défaut, cette page est située à *URL/cp-admin*
- **cp\_app** : Les pages accessibles aux utilisateurs classiques, qui souhaitent écouter les podcasts ou réagir avec
- **cp\_auth** : La page de connexion, qui apparait lorsque l'on souhaite interagir ou accéder à la page d'administration
- **cp\_install** : La page d'installation, qui est utilisé lors de la mise en place de Castopod et qui permet de finaliser l'installation et de créer un compte super-administrateur pour la première fois. Cette page est accessible par défaut à *URL/cp-install*

Vous pourrez remarquer que certains fichiers sont préfixés avec `_`. Ces fichiers sont des éléments qui sont présent sur plusieurs pages, comme par exemple le bandeau d'administration présent sur le gauche des pages de *cp\_admin* :



Pour éviter de devoir recopier ce bandeau, il y a un fichier nommé *\_sidebar.php* qui est inséré dans toutes les vues de *cp\_admin*.



# App

Dans le dossier **app**, on retrouve les contrôleurs, modèles et entités qui peuvent être utilisés par tous les utilisateurs (et pas seulement par les administrateurs). C'est dans ce dossier que la gestion de la page d'accueil et de l'écoute des épisodes est décrite.

Tout d'abord, nous avons les routes qui sont décrites dans `./Config/Routes.php`. On retrouve par exemple la route vers la page d'accueil décrite au début :

```
$routes->get('/', 'HomeController', [  
    'as' => 'home',  
]);
```

Nous décrivons le fonctionnement des routes sur Castopod dans [cette partie de la documentation](#).

Nous avons ensuite :

- Les modèles présents dans le dossier **Models**
- Les entités dans le dossier **Entity**
- Les contrôleurs dans le dossier **Controllers**.

Enfin, pour permettre à Castopod d'être utilisable dans plusieurs langues, on retrouve dans le dossier **Language** la traduction des différents textes des pages.

## Modules

Dans le dossier **modules**, on va retrouver les contrôleurs qui sont utilisés pour un cas spécifique, qui ne concerne pas tous les utilisateurs (contrairement au dossier **app** vu juste auparavant).

Dans ce dossier, on va retrouver différents dossiers liés à des services. On peut notamment retrouver comme dossiers :

- Le dossier **Admin**, qui décrit les routes et les contrôleurs du panneau d'administration
- Le dossier **API**, qui permet d'avoir un système API Rest pour accéder aux informations des podcasts depuis un service extérieur
- Le dossier **Auth**, qui permet de gérer l'authentification et les pages accessibles pour un utilisateur connecté
- Le dossier **Media**, qui permet de gérer l'enregistrement des fichiers médias en paramétrant par exemple le dossier d'enregistrement, et qui gère aussi l'enregistrement sur la base de données des informations des médias

Il existe d'autres dossiers qui permettent de gérer d'autres services de Castopod, comme l'installation (**Install**) ou les podcasts premium (**PremiumPodcasts**).



Les dossiers possèdent une structure similaire avec un dossier **Config** qui décrit les routes, et généralement un autre fichier pour paramétrer le service. Et on retrouve également un dossier **Controllers**, qui va gérer le fonctionnement de ce service.

Il peut ensuite y avoir un dossier **Models** lorsque le service a besoin de communiquer avec la base de données, et un dossier **Entity** lorsque l'on souhaite pouvoir manipuler une classe liée à ce dossier.

Enfin, comme avec le dossier **app**, on retrouve un dossier **Language** lorsque le service peut afficher une page, pour gérer la traduction de cette page dans différentes langues.



# Configuration de CodeIgniter

La plupart des fichiers de configurations du projet se situe dans le dossier **App**, dans le sous dossier **Config**. Ici, on va retrouver des fichiers qui permettent de paramétrer le fonctionnement de Castopod.

## App.php

Tout d'abord, nous avons le fichier *App.php*, dans lequel vous pouvez paramétrer :

- La racine de l'URL : *\$baseUrl*
- La langue par défaut du site : *\$defaultLocale*
- Activer la reconnaissance automatique de la langue grâce aux en-têtes des requêtes : *\$negotiateLocale*
- Les langues supportées : *\$supportedLocales*
- Le nom du site : *\$siteName*
- La description du site : *\$siteDescription*
- L'emplacement des icones : *\$siteIcon*
- La limite de stockage : *\$storageLimit*

D'autres paramètres peuvent également être modifiés mais nous avons listés les principales ici.

## Cache.php

Vous allez pouvoir dans ce fichier configurer la gestion des caches, notamment le serveur **Redis** (*\$memcached*), et l'emplacement des fichiers caches en local (*\$file*).

## Email.php

Vous allez pouvoir dans ce fichier configurer grâce à un serveur mail l'envoi des emails lors de la création d'un compte, ou lors d'une réinitialisation de mot de passe.

Vous allez pouvoir configurer le protocole à utiliser via la variable *\$protocol* : mail | sendmail | smtp



Ensuite, si vous choisissez SMTP, vous allez pouvoir rentrer les différents paramètres de votre serveur pour permettre l'envoi.

## Exceptions.php

Dans ce fichier, vous allez pouvoir configurer la page à afficher lors d'une erreur (lien manquant, fonction non présente dans un contrôleur, etc.) dans la variable `$errorViewPath`.

## Images.php

Les images utilisée par Castopod sont en partis décrites ici. On va retrouver les dimensions des images avec leur correspondance, ainsi que l'emplacement des images utilisées par défaut par Castopod, comme pour les utilisateurs (`$avatarDefaultPath`) ou pour les bannières (`$podcastBannerDefaultPaths`).

## Routing.php

Ici, on retrouve la liste de tous les fichiers possédants des routes, dans la variable `$routeFiles`. On retrouve notamment tous les fichiers de route des différents modules.

Il est aussi indiqué le dossier à utiliser par défaut pour les contrôleurs, dans la variable `$defaultNamespace`, ainsi que le contrôleur par défaut dans `$defaultController`.

Enfin, on retrouve le nom de la méthode qui est utilisé par défaut dans les routes, dans la variable `$defaultMethod`. Nous vous expliquons à quoi correspond cette méthode dans la partie concernant [les routes de Castopod](#).

## Security.php

Dans ce fichier, on retrouve différents paramètres concernant la sécurité de Castopod. Castopod utilise des protections pour éviter [les attaques de type CSRF](#). Dans ce fichier, on peut configurer les méthodes utilisées pour se protéger.

## ViewComponents.php



Enfin, dans ce dernier fichier de configuration, on peut indiquer l'emplacement des vues du projet. Par défaut, toutes les vues sont situées dans le dossier **themes/cp\_**, mais on peut modifier ou rajouter des emplacements ici.



# Présentation des outils externes

Dans cette partie, nous vous présentons les différents outils externes du projet, qui n'avaient pas besoin chacun d'une page dédié.

## Nextcloud

Nextcloud est un logiciel d'hébergement de fichiers publié pour la première fois en juin 2016. C'est un logiciel libre, qui permet d'accéder à ces fichiers via une interface web, et qui propose également de nombreuses fonctionnalités visant à son utilisateur de s'organiser et de collaborer avec d'autres utilisateurs.

L'objectif de Nextcloud est de proposer une solution simple d'auto-hébergement, qui soit Open Source pour permettre à l'utilisateur de contrôler ses données. En faisant un logiciel facile à installer et à utiliser, Nextcloud souhaite ainsi s'adapter à un public très large. De nombreuses fonctionnalités permettent à l'utilisateur de garder un confort d'utilisation, comme l'édition des fichiers, le préchargement de ses fichiers, le partage simple, ou encore la possibilité d'utiliser plein d'appareils différents pour accéder à ses données (téléphone, ordinateur, web, ...)

C'est ce logiciel qui est utilisé par Jet FM pour l'hébergement des podcasts. Pour une question législative, Jet FM doit sauvegarder durant plusieurs années les podcasts qui sont produits. Pour cela, l'association garde l'ensemble de ses podcasts sur de serveurs présents directement dans leur local, et utilise Nextcloud pour gérer cet hébergement. Cependant, Castopod ne propose pas de lier un compte Nextcloud avec un compte Castopod, et empêche donc l'utilisation direct des fichiers présents sur ces serveurs. Jet FM est donc actuellement dans l'obligation de dupliquer les fichiers pour pouvoir les publier également sur le Castpod de Jet FM.

## Bookstack

BookStack est une plateforme de gestion de documentation qui a vu le jour pour la première fois en 2015. Conçu comme un logiciel libre, BookStack offre la possibilité de rédiger des livres sous formes de chapitres et de pages à travers une interface web conviviale. Doté de multiples fonctionnalités, l'objectif principal de BookStack est de fournir une solution d'auto-hébergement simple, conforme aux principes du code source ouvert, offrant ainsi à l'utilisateur un contrôle total



sur ses données.

En mettant l'accent sur une installation et une utilisation aisées, BookStack vise à s'adresser à un large public. Ses fonctionnalités variées, telles que l'édition de fichiers, le préchargement, le partage facile et la possibilité d'accéder aux données depuis différents appareils (téléphone, ordinateur, web, etc.), garantissent une expérience utilisateur optimale.

C'est ce logiciel que nous avons choisi d'utiliser pour écrire notre documentation, car il laissait la possibilité d'écrire en Markdown (ce que nous recherchions), et son système de chapitres permet aussi de bien organiser notre documentation.

# Croppie

Croppie est une bibliothèque JS permettant l'intégration d'une interface pour rogner une image. Pour l'installer, on peut soit passer par NPM, soit ajouter ces éléments à la page :

```
<link rel="stylesheet" href="croppie.css" />
<script src="croppie.js"></script>
```

Une fois installé, on peut utiliser un objet nommé Croppie qui va permettre d'afficher l'interface en paramétrant la taille de l'image voulue (viewport), et la taille de l'interface (boundary):

```
var croppie = new Croppie($imagePreview, {
  viewport: { width: 200, height: 200, type: 'square' },
  boundary: { width: 300, height: 300 },
});
```

Et une fois cette variable croppie créée, il suffit d'utiliser la fonction result() pour récupérer l'image dans les nouvelles dimensions :

```
new Promise(resolve => {
  croppieInstance.result({
    type: 'blob',
    format: 'jpeg',
    size: 'original',
  }).then(blob => {
    const file = new File([blob], 'cropped_image.jpeg', { type: 'image/jpeg' });
    resolve(file);
  });
});
```



# Sources

Nextcloud :

- Documentation de Nextcloud : [Nextcloud](#)
- Page Wikipédia de Nextcloud : [Nextcloud - Wikipédia](#)

Bookstack :

- Documentation de Bookstack : [Docs · BookStack](#)
- Page Wikipédia de Bookstack : [BookStack - Wikipedia](#)

Croppie :

- Site officiel de Croppie : [Croppie - a simple javascript image cropper - Foliotek](#)
- Github de Croppie : [Foliotek/Croppie: A Javascript Image Cropper \(github.com\)](#)



# Fonctionnement de Castopod



# Routes

Comme expliqué auparavant, Castopod s'appuie sur un modèle MVC qui permet d'accéder à des pages du site grâce à des contrôleurs qui vont charger des vues. Si vous n'avez pas compris un seul mot, nous vous renvoyons vers l'article qui correspond à [l'explication de ce principe](#).

Le projet est divisé en deux dossiers principaux, le dossier **App** et le dossier **Modules**, le premier étant plutôt destiné aux pages qui vont être accessibles à n'importe qui, tandis que le deuxième est destiné à une utilisation particulière comme de l'administration ou de l'authentification. Là aussi, si vous n'avez pas tout compris, nous vous redirigeons vers l'article correspondant à [l'organisation du projet](#).

## Définition d'une route

Pour développer sur la manière dont les routes sont agencées dans le projet, nous allons nous intéresser dans un premier à la structure d'une route. Prenons par exemple la route vers la page d'accueil (accessible à l'emplacement *app/Config/Routes.php*) :

```
$routes->get('/', 'HomeController', [  
    'as' => 'home',  
]);
```

Cette configuration indique quatre éléments :

- La route est accessible via le protocole GET (et non POST ou un autre)
- Elle concerne l'index '/', c'est-à-dire la racine du projet, lorsque l'on tape juste l'URL de l'hébergement (par défaut *localhost:8080*)
- Elle fait appel au contrôleur **HomeController**, avec la fonction particulière (il n'y a pas de fonction d'indiquée)
- Le nom de cette route est **home**. On pourra donc utiliser ce nom si on souhaite configurer une redirection dans une autre route ou un contrôleur.

La gestion de l'affichage est ensuite gérée par le contrôleur associé à la route, ici **HomeController**, qui en fonction de la fonction appelée par la route pourra aller chercher la vue recherchée.



Si maintenant on souhaite avoir une route sur l'URL avec comme chemin */health*, qui permettra de savoir si Castopod rencontre un problème ou non.

La structure sera assez similaire, il faudra juste changer l'URL correspondant à la route, et indiquer une fonction particulière du contrôleur :

```
$routes->get('/health', 'HomeController::health', [  
    'as' => 'health',  
]);
```

Avec cette route, lorsque l'utilisateur accèdera à la page *URL/health*, cela chargera la fonction *health()* du contrôleur **HomeController**.

Enfin, il est possible d'indiquer un paramètre à donner au contrôleur lors de la connexion, en utilisant le caractère \$.

On peut appliquer des filtres à ce paramètre en indiquant son type, et en définissant ce type au début du fichier *route*.

Imaginons que la fonction *health(int \$id)* du contrôleur nécessite un paramètre numérique strictement inférieur à 1000. Nous pourrions donc commencer par créer un filtre **nombre**, et appliquer ce filtre au paramètre indiqué par la route :

```
$routes->addPlaceholder('nombre', '[0-9]{1,3}');
```

Nous avons ici créer un filtre **nombre**, qui indique que chaque caractère doit être compris entre 0 et 9, et la taille de la chaîne de caractère ne peut pas être plus grand que 3 (donc numériquement qui soit inférieure ou égale 999).

Pour créer la route, il suffira donc d'indiquer le paramètre lors de la déclaration :

```
$routes->get('health/(:nombre)', 'HomeController::health/$1');
```

## Définition d'un groupe

Avec un ensemble de routes, il est possible de les regrouper sous un groupe. Cela peut être pratique si ces routes possèdent un attribut commun, comme le fait qu'elles nécessitent un même paramètre ou qu'elles font parties d'un endroit particulier sur le site (comme la partie administration par exemple).

Prenons comme exemple l'accès à une page de podcast pour un auditeur lambda. Pour accéder à la page de ce podcast, Castopod utilise le nom du podcast qui est une chaîne de caractère de taille maximale de 32. Pour cela, lorsque l'utilisateur clique sur un podcast, on accède à la page du podcast sous la forme *URL/@nom\_du\_podcast*.



Dans les routes, cela est représenté par un groupe de route, qui récupère ce premier paramètre de l'URL (*@nom\_du\_podcast*) via un filtre, et qui développe ensuite quel contrôleur il doit appelé en fonction de la page de ce podcast.

Pour être plus concret, voici ce que fait concrètement Castopod. Tout d'abord, il crée un filtre **podcastHandle** pour le nom du podcast :

```
$routes->addPlaceholder('podcastHandle', '[a-zA-Z0-9\_]{1,32}');
```

Une fois ce filtre créé, un groupe est indiqué pour regrouper toutes les routes possibles lorsqu'un podcast a été sélectionné par l'utilisateur :

```
$routes->group('@(:podcastHandle)', static function ($routes): void {  
    ...  
})
```

On a regroupé ici tous les liens qui commence par un *@nom\_du\_podcast*, donc dès que l'utilisateur voudra accéder à une page qui commence par *URL/@nom\_du\_podcast/...*, Castopod ira récupérer les routes dans ce groupe.

Imaginons que l'utilisateur souhaite accéder à la page d'un podcast, l'URL sera juste *URL/@nom\_du\_podcast*. Cela signifie donc que l'utilisateur accède à la racine de ce groupe, et comme précédemment, pour indiquer la racine, on utilise l'index '/'.

Cela donnera donc sous forme de routes :

```
$routes->group('@(:podcastHandle)', static function ($routes): void {  
    $routes->get('/', 'PodcastController::activity/$1', [  
        'as'          => 'podcast-activity',  
    ]);  
})
```

Ici, Castopod utilise le contrôleur **PodcastController** pour gérer l'affichage de la page, via la fonction *activity()*. Et si on souhaite accéder à une autre page du podcast, comme par exemple la page liée aux informations supplémentaires (la page à propos) via le lien *URL/@nom\_du\_podcast/about*, comme précédemment, il suffira de rajouter une route dans le groupe :

```
$routes->group('@(:podcastHandle)', static function ($routes): void {  
    $routes->get('/', 'PodcastController::activity/$1', [  
        'as'          => 'podcast-activity',  
    ]);  
})
```



```
$routes->get('about', 'PodcastController::about/$1', [  
    'as' => 'podcast-about',  
]);  
}
```

Enfin, il est possible dans les groupes d'indiquer également l'espace de travail (c'est-à-dire l'emplacement) des contrôleurs à utiliser dans les routes. Par exemple, pour indiquer que les routes liées à l'authentification (qui dans l'exemple commencent par *URL/authentification/...*) utiliseront les contrôleurs du dossier **modules/Auth/Controllers**, on peut ajouter ce paramètre :

```
$routes->group(  
    'authentification',  
    [  
        'namespace' => 'Modules\Auth\Controllers',  
    ],  
    static function ($routes): void {  
        ...  
    }  
);
```

Pour aller plus en profondeur dans la gestion des routes sur CodeIgniter, nous vous invitons à lire [la documentation liée au routing](#) qui développe sur les autres paramètres possibles pour une route. La gestion des différents paramètres est ensuite gérée par les contrôleurs, ainsi que la gestion de l'affichage de la page. Nous détaillons ce fonctionnement sur [cette page](#).



# Contrôleurs

Une fois les routes créées, on peut donc utiliser les contrôleurs pour faire le lien avec les vues du projet.

Chaque contrôleur utilise un **namespace** qui correspond à un emplacement dans le système de fichiers. Cela va lui permettre de savoir de connaître les noms des routes de son dossier, et ainsi d'y faire référence. Par exemple, les contrôleurs présents dans le dossier **modules/Admin** vont indiquer être dans ce namespace, et pourront ainsi faire références aux routes de ce dossier. Cela permet de bien diviser le projet, pour éviter des manipulations entre des contrôleurs qui ne sont pas censés communiquer, et pouvoir ainsi augmenter la sécurité.

## Affichage d'une vue

Les contrôleur sont des classes héritants d'un contrôleur nommé **BaseController**, héritant lui-même d'un contrôleur nommé **Controller**, qui lui provient directement de CodeIgniter. **BaseController** est une classe abstraite, qui va permettre de charger des éléments si besoin et ainsi d'améliorer les performances. Son utilisation est détaillé dans [la documentation de CodeIgniter](#) mais elle n'a pas beaucoup utilisée par Castopod.

Si on reprend l'exemple de la page d'accueil, nous avons donc besoin d'afficher une page avec la liste des podcasts disponibles, qui sont triés suivant un certain ordre. Pour cela, on va avoir besoin de savoir quelles sont les podcasts inscrits dans la base de données, et donc utilisés le modèle associé. Pour les podcasts, le modèle est **PodcastModel**, et pour savoir comment les triés, un simple paramètre dans la requête permettra de savoir comment l'utilisateur veut trier les podcasts.

Pour rappel, la route était :

```
$routes->get('/', 'HomeController', [  
    'as' => 'home',  
]);
```

Il n'y a pas de fonction particulière du contrôleur qui a été appelée par la route. Pour définir la fonction par défaut du contrôleur, il suffit de déclarer une fonction nommée *index()*. Ensuite, pour la signature de la fonction, on a plusieurs solutions, qui sont généralement :

- Un objet string avec la fonction *view()*, qui attendra le nom d'une vue



- Un objet `RedirectResponse`, avec la fonction `redirect()->route()`, qui attendra lui le nom d'une route

Les objets qui peuvent être renvoyés proviennent du dossier `CodeIgniter\HTTP`, et peuvent être importés à l'aide du mot-clé **use**.

Ici, le contrôleur renvoie deux types différents en fonction de la réponse du modèle de podcast, si un seul podcast est récupéré alors on chargera directement la page du podcast, sinon on affichera la vue nommée **home** :

```
namespace App\Controllers;

use App\Models\PodcastModel;
use CodeIgniter\HTTP\RedirectResponse;

class HomeController extends BaseController
{
    public function index(): RedirectResponse | string
    {
        // Paramètres possibles du GET
        $sortOptions = ['activity', 'created_desc', 'created_asc'];
        // Récupérer le paramètre, ou mettre par activité récente le tri
        $sortBy = in_array($this->request->getGet('sort'), $sortOptions, true) ? $this->request->getGet(
            'sort'
        ) : 'activity';

        // Récupérer les podcasts
        $allPodcasts = (new PodcastModel())->getAllPodcasts($sortBy);

        // Regarder s'il n'y a qu'un podcast, et dans ce cas renvoyé sur la page
        // du podcast directement
        if (count($allPodcasts) === 1) {
            return redirect()->route('podcast-activity', [$allPodcasts[0]->handle]);
        }

        // Création d'un objet data, qui va contenir les podcasts récupérés
        $data = [
            'podcasts' => $allPodcasts,
            'sortBy' => $sortBy,
        ];
    }
}
```



```
// Charger la vue avec l'objet data
return view('home', $data);
}
}
```

Vous remarquerez qu'un objet *data* a été créé avec deux paramètres, *podcasts* qui contient la liste des podcasts récupérés et *sortBy* qui contient la manière dont les podcasts sont triés. Dans la vue **home**, on pourra directement utiliser ces valeurs en faisant référence aux paramètres. Pour cela, on pourra directement utiliser les variables *podcasts* et *sortBy* dans la vue.

# Utilisation des paramètres

## Paramètres d'un groupe

Nous avons vu auparavant comment il était possible d'indiquer des paramètres à un contrôleur en prenant l'exemple d'un groupe podcast. Pour rappel, l'URL ressemblait donc à cela : *URL/@nom\_du\_podcast*.

Nous allons maintenant voir comment est géré ce paramètre pour le contrôleur lié au podcast. Pour cela, le contrôleur implémente une fonction nommée *\_remap*. Cette fonction va permettre de récupérer les paramètres donnés dans l'URL lorsqu'une fonction de ce contrôleur est appelée.

Si on reprend notre exemple, on veut donc récupérer le nom du podcast. Il n'existe pas de route qui appelle le contrôleur sans avoir le nom du podcast dans l'URL, il faudra donc lever une erreur dans le cas où aucun paramètre n'est donné.

Ensuite, on peut également vérifier si ce podcast existe dans la base de données. Pour cela, un appel au modèle peut être effectué en cherchant dedans s'il existe un podcast avec ce nom. Dans le cas où un podcast est trouvé, on peut utiliser une entité **Podcast**, qui contiendra les informations sur le podcast recherché (son identifiant, son titre, sa description, etc.)

Ce podcast pourra ensuite être enregistré dans un attribut du contrôleur, pour pouvoir par la suite manipuler ce podcast pour récupérer les différentes informations, et ainsi les afficher dans la vue.

Tout ça, c'est exactement ce que fait le contrôleur **PodcastController**, en redéfinissant la méthode *\_remap()* :

```
public function _remap(string $method, string ...$params): mixed
{
    if ($params === []) {
        throw PageNotFoundException::forPageNotFound();
    }
}
```



```

if (
    ! ($podcast = (new PodcastModel())->getPodcastByHandle($params[0])) instanceof Podcast
) {
    throw PageNotFoundException::forPageNotFound();
}

$this->podcast = $podcast;

unset($params[0]);

return $this->{$method}(...$params);
}

```

Une fois cette méthode écrite, on peut aisément utiliser la variable *\$this->podcast* pour gérer le podcast sélectionné par l'utilisateur dans la suite du contrôleur.

Pour donner un exemple simplifié, pour afficher la page avec tous les épisodes d'un podcast, le contrôleur **PodcastController** utilise donc cette variable en créant une nouvelle variable *\$data* :

```

$data = [
    'podcast' => $this->podcast,
    'episodes' => (new EpisodeModel())->getPodcastEpisodes(
        $this->podcast->id,
        $this->podcast->type,
    ),
];

```

Et affiche ensuite la page avec les données chargées :

```

return view('podcast/episodes', $data);

```

Si vous allez voir le contrôleur, vous verrez que nous avons simplifié les fonctions pour présenter directement l'utilisation des paramètres. Le contrôleur ajoute en effet un système pour gérer l'enregistrement d'une requête à la page d'un podcast pour pouvoir faire de la statistique derrière (et plus précisément le non-enregistrement lorsque l'utilisateur est connecté, pour ne pas influencer ces statistiques par un administrateur), une gestion de cache pour améliorer les performances, et la gestion des années et des saisons sélectionnées par l'utilisateur.

## Paramètres d'une requête



Il existe un deuxième type de paramètres, ceux qui sont liés directement à la requête. Lorsque vous faites une requête GET avec votre navigateur, vous pouvez indiquer des paramètres dans l'URL pour permettre de passer des informations à la page. Par exemple, pour la page *monsie.fr/mapage?nombre=1&lettre=a*, la page aura 2 paramètres qui sont nombre et lettre, qui seront respectivement égaux à 1 et a.

Si vous n'avez pas tout compris, nous vous conseillons de lire [cet article](#) du site IONIS qui explique les paramètres de requête.

CodeIgniter permet de récupérer ces paramètres via la méthode *getGet(nomVariable)* qui est lié à l'objet **Incoming Request**. On peut également récupérer des paramètres d'autres types de requêtes comme avec la méthode POST, avec la fonction *getPost(nomVariable)*. Comme décrit dans [la documentation de CodeIgniter](#), voici les possibilités :

```
// Tiré de la documentation de CodeIgniter

// the URI path being requested (i.e., /about)
$request->getUri()->getPath();

// Retrieve $_GET and $_POST variables
$request->getGet('foo');
$request->getPost('foo');

// Retrieve from $_REQUEST which should include
// both $_GET and $_POST contents
$request->getVar('foo');

// Retrieve JSON from AJAX calls
$request->getJSON();

// Retrieve server variables
$request->getServer('Host');

// Retrieve an HTTP Request header, with case-insensitive names
$request->header('host');
$request->header('Content-Type');

// Checks the HTTP method
$request->is('get');
$request->is('post');
```



Il existe également un service associé à la gestion des requêtes, qui permet d'utiliser la même fonction via [la fonctionnalité de Services](#) de CodeIgniter :

```
Services::request()->getGet('nomVariable')
```

Maintenant que nous avons vu comment était géré les routes par les contrôleurs, nous pouvons nous pencher sur le dernier maillon de la chaîne, c'est-à-dire [les vues](#).



# Vues

Pour terminer cette explication du lien entre les routes, les contrôleurs et les vues, nous allons développer la manière dont sont gérées les vues par Castopod.

## Tailwind

Castopod utilise pour ses vues [le framework Tailwind](#) qui est un utilitaire CSS open-source qui va permettre de simplifier la gestion du style des pages. Vous pouvez retrouver le fichier de configuration à la racine de Castopod, dans le fichier *tailwind.config.js*. Dans ce fichier, on peut notamment avoir la liste des couleurs pour chaque thème, ou des classes par défaut pour certains éléments HTML.

Cette utilisation de Tailwind est directement lié au fonctionnement de Vite dans le mode développement. C'est pour cela que vous retrouverez au début de chaque vue ce morceau de code permettant entre autre de pouvoir utiliser Tailwind :

```
<?= service('vite')
  ->asset('styles/index.css', 'css') ?>
<?= service('vite')
  ->asset('js/app.ts', 'js') ?>
```

Une fonction *service* est utilisée ici, et pour mieux comprendre l'utilisation de cette fonction que vous retrouvez avec d'autres arguments dans les vues, nous vous invitons à vous rendre sur [la page où nous détaillons l'utilité de cette fonction](#).

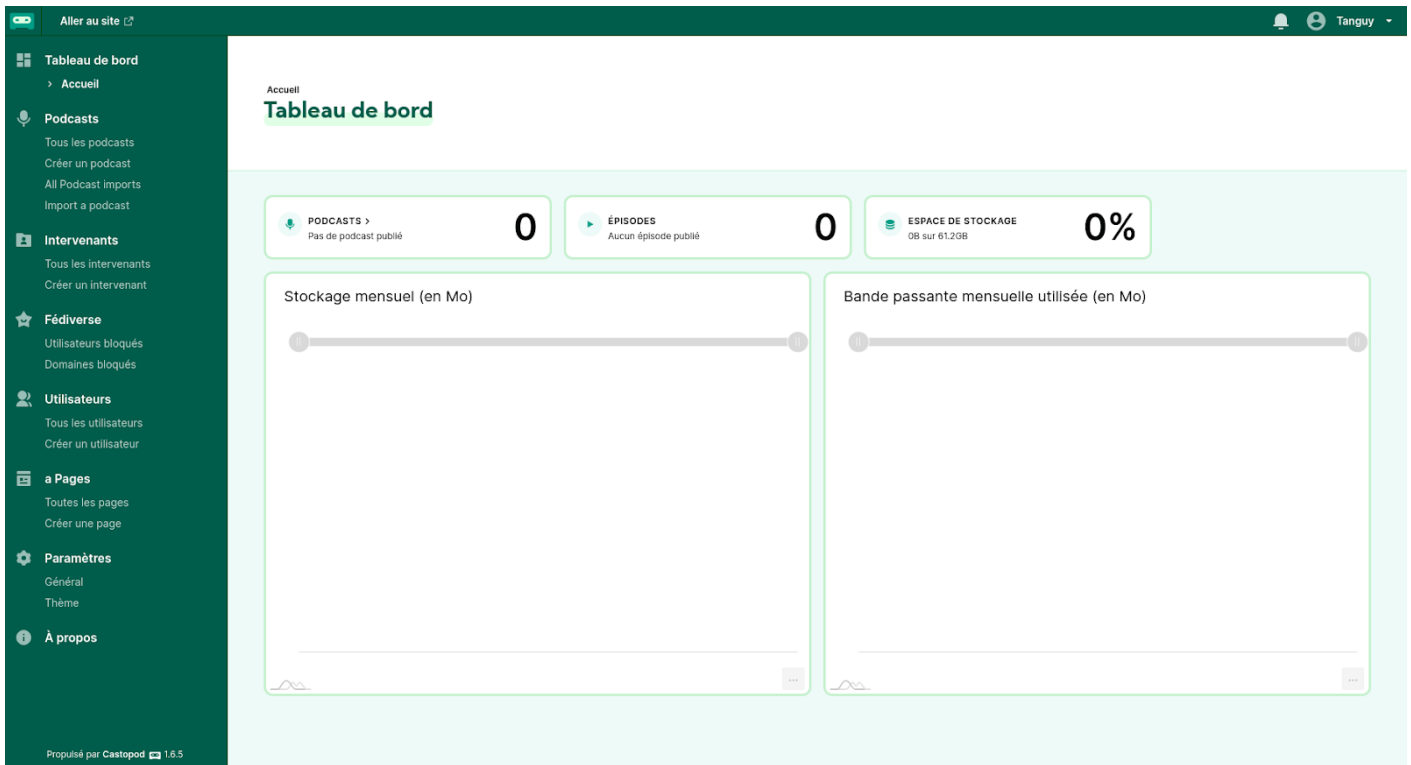
## Layout

Les vues sont écrites en PHP, ce qui permet d'utiliser des fonctions de PHP et de CodeIgniter dans leur fichier. Cela permet par exemple d'utiliser un système de mise en page à l'aide de composant qu'on peut réutiliser dans plusieurs pages.

Prenons par exemple la page d'administration. Quel que soit la page sur laquelle vous êtes, vous retrouvez :



- Un bandeau supérieur, dans lequel vous avez le logo de Castopod, une redirection sur le site principal, etc.
- Un bandeau latéral à gauche, dans lequel vous retrouvez les différentes pages d'administration



Pour éviter de devoir réécrire dans chaque vue des pages d'administration, on va utiliser ce qui s'appelle des *layouts*, en écrivant des morceaux de page réutilisable. Pour les pages d'administration, les deux bandeaux sont `_partials/_nav_header.php` pour le bandeau supérieur et `_partials/_nav_aside.php` pour le bandeau latéral.

Il suffit ensuite d'indiquer sur les vues d'utiliser ces composants à l'aide de la fonction `include(nomLayout)`.

Pour ces deux composants, on aura donc dans la vue :

```
<?= $this->include('_partials/_nav_header') ?>
<?= $this->include('_partials/_nav_aside') ?>
```

Si vous allez voir la vue **dashboard**, vous ne verrez pas tout de suite ces deux lignes, elles sont en fait incluses dans un composant plus général, nommé `_layout`, qui va gérer d'autres éléments que l'on retrouve sur toutes les pages d'administrations (comme le titre de la page par exemple).

Si vous souhaitez ajouter des composants à votre vue en plus des *layouts* inclus, vous devrez les placer dans une section à part à l'aide de la fonction

```
<?= $this->section('nonSection') ?>*
```



Par exemple, pour la vue de création d'épisode depuis la page d'administration, voici ce que nous avons :

```
<?= $this->extend('_layout') ?>

<?= $this->section('title') ?>
<?= lang('Episode.create') ?>
<?= $this->endSection() ?>

<?= $this->section('pageTitle') ?>
<?= lang('Episode.create') ?>
<?= $this->endSection() ?>

<?= $this->section('content') ?>
    // Formulaire de création d'épisode
<?= $this->endSection() ?>
```

On peut voir que la page indique le titre de la page dans la section *title*, mais si vous vous souvenez bien de *\_layout*, nous avons indiqué que le titre de la page était géré dans ce composant.

En vérité, il va générer le titre de la page à partir d'un texte fixe (qui est 'Castopod Admin' par défaut), ainsi que de la section de la page qui l'appelle. Cela se fait avec la fonction *renderSection*. Voici ce qu'on retrouve dans le fichier *\_layout* :

```
<title><?= $this->renderSection('title') ?> | Castopod Admin</title>
```

# Variable

Nous avons précédemment qu'il était possible de [charger une vue avec des données à l'aide du contrôleur](#), voyons maintenant comment utiliser ces données dans la page.

Si on reprend la page d'accueil, dans le contrôleur nous avons cette fonction pour charger la page :

```
namespace App\Controllers;

use App\Models\PodcastModel;
use CodeIgniter\HTTP\RedirectResponse;
```



```

class HomeController extends BaseController
{
    public function index(): RedirectResponse | string
    {
        ...

        // Création d'un objet data, qui va contenir les podcasts récupérés
        $data = [
            'podcasts' => $allPodcasts,
            'sortBy'  => $sortBy,
        ];

        // Charger la vue avec l'objet data
        return view('home', $data);
    }
}

```

On voit ici qu'une variable *\$data* est créé avec deux attributs : *podcasts* et *sortBy*. Le nom de ces attributs est important car c'est ce nom qui pourra être utilisé pour référer directement à leur valeur dans la vue.

Ainsi, pour afficher les différents podcasts récupérés par le contrôleur, il suffit de faire une boucle sur tous les podcasts (en vérifiant au passage que cette variable n'est pas vide, ce qui signifierai qu'aucun podcast n'a encore été créé. Voici ce qu'on trouve dans la vue correspondant à la page d'accueil de Castopod :

```

<div class="grid gap-4 mt-4 grid-cols-cards">
    <?php if ($podcasts): ?>
        <?php foreach ($podcasts as $podcast): ?>
            // Gestion de chaque podcast trouvé
        <?php endforeach; ?>
    <?php else: ?>
        // Gestion si aucun podcast n'a été trouvé
        <p class="italic"><?= lang('Home.no_podcast') ?></p>
    <?php endif; ?>
</div>

```

Pour rappel, *\$podcasts* contenait la liste des podcasts créés, sous forme de l'entité *Podcast*. Donc chaque élément de cette liste contenait les informations du podcast, c'est-à-dire le nom, l'identifiant, le lien, etc.

On peut directement accéder à ces informations en indiquant l'attribut à accéder dans la variable.



Pour accéder au nom du premier podcast par exemple, on pourra y accéder comme cela :

```
$podcasts[0]->title
```



# Fonctionnalités supplémentaires

Dans cette page, nous allons développer certaines fonctionnalités utilisées par Castopod, mais qui ne nécessite pas une page complète d'informations.

## Services et Config

### Services

Les services sont des fonctions qui vont pouvoir être utilisées par n'importe quelle vue ou contrôleur, et qui sont définis dans une classe ou un composant du projet. Pour les fichiers de CodeIgniter, ces services sont généralement situés dans le fichier *Config/Services.php*. C'est par exemple le cas de Vite, qui permet l'intégration de Tailwind à l'aide de son service (situé dans *App/Libraries/Vite/*) avec ce code présent sur toutes les vues, qui permet de récupérer les fichiers CSS et JS du projet :

```
<?= service('vite')
    ->asset('styles/index.css', 'css') ?>
<?= service('vite')
    ->asset('js/app.ts', 'js') ?>
```

Un autre exemple de service, avec le service **settings**, qui permet d'accéder entre autre aux valeurs du fichier de configuration général *App/Config/App.php*. C'est avec ce service par exemple que l'on peut connaître la couleur de thème voulue par l'utilisateur.

Pour cela, deux fonctions sont proposées par ce service, la fonction *get* et *set*. Dans le fichier de configuration, la couleur du thème est dans la variable *theme*, ainsi les fonctions seront :

```
// Pour récupérer la couleur du thème
service('settings')
    ->get('App.theme')

// Pour mettre la variable $theme comme couleur de thème
```



```
service('settings')
->set('App.theme', $theme);
```

Pour connaître la liste des variables que l'on peut paramétrer via le service settings, nous vous invitons à lire la page correspondant à [la configuration de Castopod](#) dans la partie **App**.

## Config

A l'instar des services, si vous fouillez dans les fichiers de Castopod, vous pourrez parfois tomber sur des lignes PHP qui ressemble à cela :

```
namespace Modules\Admin\Config;
...
config(Admin::class)->gateway
```

Cela permet de réutiliser un attribut commun, ici par exemple l'emplacement URL des pages d'administrations. Codelgniter va ici remplacer cette partie de code par l'attribut *gateway* de la classe Admin du dossier **Config**.

Si nous allons à cette emplacement, c'est-à-dire *modules/Admin/Config/Admin.php*, nous avons ce fichier :

```
class Admin extends BaseConfig
{
    // Attribut gateway de la classe Admin
    public string $gateway = 'cp-admin';
    ...
}
```

Codelgniter va donc remplir *config(Admin::class)->gateway* par *cp-admin*.

Pour conclure, Config est un mot-clé qui va plus être utilisé pour accéder à des fonctions ou variables liées à la configuration initiale du projet, tandis que Services est liée à des objets qui vont permettre d'utiliser des fonctionnalités spécifiques de l'application.

## Helper

Les **Helpers** sont des fonctions qui ressemblent beaucoup à ce que peut proposer les Services. La différence est que les Helpers vont permettre l'utilisation de fonctions globales tandis que les Services sont des composants spécifiques préchargés qui encapsulent des fonctionnalités



avancées pour des parties spécifiques de l'application dans CodeIgniter.

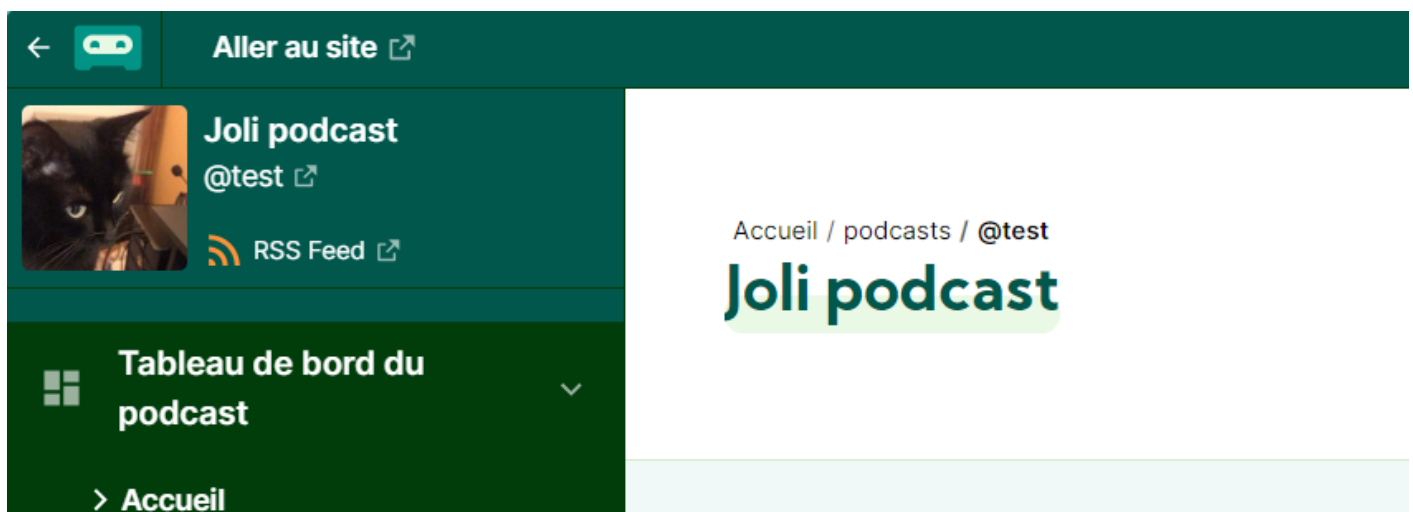
Les Helpers qui peuvent être utilisés sont écrits dans le fichier *App/Config/Autoload.php*, dans la variable **\$helpers**. On y retrouve par exemple pour Castopod un Helper pour la gestion des authentifications. Mais on peut également intégrer un Helper directement dans un contrôleur, à l'aide de la fonction *helper()*, dans laquelle on va indiquer les Helpers que l'on souhaite utiliser.

Castopod utilise un framework de CodeIgniter pour gérer l'authentification et les autorisations (accessible dans le dossier *vendor/codeigniter4/shield/src/Authentication*) et qui propose un Helper pour savoir si un utilisateur est authentifié. On retrouve une fonction *loggedIn* d'ailleurs dans la page principale du site, qui permet de savoir si la vue doit afficher un bandeau d'administration :

```
<?php if (auth()->loggedIn()): ?>
    <?= $this->include('_admin_navbar') ?>
<?php endif; ?>
```

# Breadcrumb

Breadcrumb (ou en français un Fil d'Ariane) est un utilitaire qui va permettre d'afficher la hiérarchie de la page actuelle :



Ici par exemple, le breadcrumb est *Accueil/Podcasts/@test*.

Pour gérer cela, Castopod utilise une bibliothèque écrite par **Ad Aures** qui permet de générer cette liste. Pour cela, la vue utilise une fonction *render\_breadcrumb()* qui va générer la liste en fonction de l'emplacement de la page en utilisant la fonction *render()* de la classe **App/Librairies/Breadcrumb**.

Le texte qui sera affiché, correspondant à la page, sera celui indiqué dans le dossier **Language** dans le fichier **Breadcrumb.php** de la langue correspondant à l'utilisateur.

Cette bibliothèque utilise deux fichiers :



- Le fichier helper : App/Helper/breadcrumb\_helper.php
- Le fichier de la classe : App/Librairies/Breadcrumb.php

Et c'est le service *breadcrumb* qui va permettre de relier les deux.

# Rules

Lorsque des données sont transmises via un formulaire, les données vont être vérifiées pour qu'elles respectent certaines règles. Par exemple, lors du téléversement d'un épisode, le fichier audio doit avoir une extension mp3 ou m4a, et la couverture de l'épisode doit être carré, avec une dimension minimale de 1400\*1400px.

C'est le contrôleur associé à la création d'un épisode qui va vérifier si ces conditions ont bien été respectées. Cela va pouvoir être fait grâce à la méthode *validate()*, qui va avoir en paramètre la liste des règles qui doivent être respectées, qui va renvoyer un booléen en fonction du respect des règles et un *validator* qui va contenir le message d'erreur.

Si on reprend notre exemple, cela ressemblera à ça :

```
$rules = [  
    'audio_file'      => 'uploaded[audio_file]|ext_in[audio_file,mp3,m4a];',  
    'cover'          =>  
    'is_image[cover]|ext_in[cover,jpg,jpeg,png]|min_dims[cover,1400,1400]|is_image_ratio[cover,1,1]',  
];  
  
// On revient en arrière avec un message d'erreur  
if (! $this->validate($rules)) {  
    return redirect()  
        ->back()  
        ->withInput()  
        ->with('errors', $this->validator->getErrors());  
}
```

Ce concept est expliqué plus en profondeur dans [la documentation de CodeIgniter](#).

# Components

En regardant les formulaires dans les vues de Castopod, vous remarquerez peut-être une manière un peu particulière d'écrire ces formulaires. Par exemple avec la page de connexion au site :



...

```
<?= $this->section('content') ?>
```

```
<form actions="<?= url_to('login') ?>" method="POST" class="flex flex-col w-full gap-y-4">
```

```
<?= csrf_field() ?>
```

```
<Forms.Field
```

```
    name="email"
```

```
    label="<?= lang('Auth.email') ?>"
```

```
    required="true"
```

```
    type="email"
```

```
    inputmode="email"
```

```
    autocomplete="username"
```

```
    autofocus="autofocus"
```

```
/>
```

```
<Forms.Field
```

```
    name="password"
```

```
    label="<?= lang('Auth.password') ?>"
```

```
    type="password"
```

```
    inputmode="text"
```

```
    autocomplete="current-password"
```

```
    required="true" />
```

...

```
<Button variant="primary" type="submit" class="self-end"><?= lang('Auth.login') ?></Button>
```

```
</form>
```

```
<?= $this->endSection() ?>
```

...

Vous aurez remarqué que les différentes entrées du formulaire sont sous la forme *Forms.Field*, et qu'à la fin il y a un composant nommé *Button*. Cela est possible grâce à l'utilisation de Composants, appelés **Components** dans CodeIgniter.

Tous les composants HTML que vous pouvez utiliser dans une vue CodeIgniter sont décrits dans le dossier **App/Views/Components**. Cela peut être pratique pour rendre vos pages plus lisibles et plus simples.



Vous pourrez également retrouver dans ce dossier un sous-dossier **errors** contenant les pages d'erreurs lorsqu'une page est inaccessible ou qu'un utilisateur n'a pas les droits nécessaires pour y accéder.



# Authentifications et autorisations

La partie gestion des connexions et déconnexions des utilisateurs à la page d'administration est gérée à l'aide du *Shield* d'authentification de Codelgniter. Ce *shield* est un framework développé par Codelgniter pour l'authentification et l'administration sur un projet Codelgniter. Pour avoir plus de détail, vous pouvez vous rendre sur [la page de documentation](#) de Codelgniter.

## Modèles associés

Pour commencer, voyons comment sont enregistrés les utilisateurs et leurs informations dans la base de données.

Codelgniter divise la gestion des données en deux tables :

- Une table *User*, gérée par le modèle *UserModel*, qui va contenir les informations liés sur l'activité du compte (sa dernière connexion, quand est-ce qu'il a été créé, etc.)
- Une table *Auth\_identities* qui va enregistrer les identifiants de connexion des comptes, en hashant le mot de passe. Cette table est gérée par le framework directement, ainsi que le liant entre les deux tables (lorsqu'un utilisateur est créé sur *User*, le framework va créer cette utilisateur dans cette table également).

Pour la gestion des autorisations, la table *Auth\_groups\_users* va liée l'identifiant des utilisateurs à leurs droits sur le projet. Pour Castopod, il existe quatre groupes :

- *podcaster* : le groupe par défaut lorsqu'un utilisateur est créé
- *superadmin* : le groupe possédant tous les droits sur le projet
- *guest* : identique à *podcaster*, mais qui n'a pas accès au panneau de configuration
- *admin* : le groupe possédant tous les droits sur un podcast

Tous ces groupes sont gérés dans le fichier *Modules/Auth/Config/AuthGroups.php*.

## Configuration du framework



Castopod utilise principalement la connexion à l'aide d'un email et d'un mot de passe. Cette connexion est gérée par des identifiants sessions. Lorsque l'utilisateur va se connecter à Castopod, si les données indiquées sont bonnes, Castopod va créer un identifiant unique et va le sauvegarder pendant un certain temps, et cette variable s'appelle une session. Castopod va ensuite renvoyer l'identifiant à l'utilisateur, qui va être enregistré dans un cookie sur son ordinateur, et qui va être utilisé pour se connecter à la page voulue.

Pour empêcher un visiteur non connecté d'accéder à certaines pages, on va s'appuyer sur ce principe de session, en utilisant le fichier *App/Config/Filters.php*. Ce fichier est utilisé pour pouvoir modifier la demande d'un utilisateur en fonction de certains paramètres. Par exemple, pour gérer la connexion à la page d'administration, nous avons ceci :

```
public function __construct()
{
    parent::__construct();

    $this->filters = [
        'session' => [
            'before' => [config('Admin')->gateway . '*'],
        ],
    ];
}
```

On peut voir que lorsque l'utilisateur souhaite accéder à une page d'administration, identifier par *config('Admin')->gateway . '\*'*, s'il n'est pas connecté, il sera redirigé vers une page de connexion.

Dans le fichier *Modules/Auth/Config/Auth.php*, vous allez pouvoir configurer les liens de redirection lorsque l'utilisateur se sera connecté, à l'aide de la variable *\$redirects* :

```
public array $redirects = [
    'register'      => '/',
    'login'         => '/',
    'logout'        => 'login',
    'force_reset'   => '/',
    'permission_denied' => '/',
    'group_denied'  => '/',
];
```

Vous pouvez également configurer la reconnexion lorsque l'utilisateur est déjà connecté dans le constructeur de l'objet, dans la variable du même nom.

Vous allez pouvoir configurer les vues associées à chaque page du framework dans la variable *views* du même fichier. Vous pouvez également indiquer des routes qui seront reliées aux



contrôleurs associés à l'aide du fichier *AuthRoutes.php*.  
Par exemple, pour le projet Castopod, nous avons ceci :

```
public array $views = [
    'login'          => 'login',
    'register'       => 'register',
    'layout'        => '_layout',
    'action_email_2fa'    => 'email_2fa_show',
    'action_email_2fa_verify' => 'email_2fa_verify',
    'action_email_2fa_email' => 'emails/email_2fa_email',
    'action_email_activate_show' => 'email_activate_show',
    'action_email_activate_email' => 'emails/email_activate_email',
    'magic-link-login'    => 'magic_link_form',
    'magic-link-message'  => 'magic_link_message',
    'magic-link-email'    => 'emails/magic_link_email',
    'magic-link-set-password' => 'magic_link_set_password',
    'welcome-email'      => 'emails/welcome_email',
];
```

Nous voyons que pour la page de connexion, c'est la route *login* qui est associée. Et si nous allons voir dans le fichier *AuthRoutes.php*, nous retrouvons le contrôleur associés :

```
class AuthRoutes extends ShieldAuthRoutes
{
    public array $routes = [
        'login' => [
            ['get', 'login', 'LoginController::loginView', 'login'],
            ['post', 'login', 'LoginController::loginAction'],
        ],
        ...
    ];
}
```

C'est le contrôleur *LoginController* qui permet d'obtenir la vue associée via la méthode *loginView()*

Enfin, il est possible dans le fichier *Modules/Auth/Config/Auth.php* de paramétrer le cookie qui va enregistrer l'identifiant de l'utilisateur, en modifiant la variable *sessionConfig*. Pour l'instant, cette variable n'a pas été modifiée, donc sa configuration est celle par défaut dans le fichier *Vendor/CodeIgniter4/shield/src/Auth.php* :



```
public array $sessionConfig = [  
    'field'           => 'user',  
    'allowRemembering' => true,  
    'rememberCookieName' => 'remember',  
    'rememberLength'   => 30 * DAY,  
];
```

Lorsque la configuration est celle par défaut, les variables possèdent les valeurs du fichier présent dans ce dossier. Mais pour modifier ces valeurs, il suffit alors de paramétrer les fichiers du dossier *Modules* dans le projet de Castopod.

D'autres options de configuration sont possibles sur ce framework, pour en savoir plus nous vous conseillons à nouveau de vous rendre sur [la documentation officielle](#).

## Fonctions du framework

Une fois la configuration terminée, nous allons pouvoir nous attaquer aux fonctions proposées par ce framework. Ces fonctions vont utiliser la classe **Auth**, que nous allons pouvoir appeler à l'aide de la fonction *auth()*.

Voici les différentes fonctions présentes dans la classe **Auth** :

- *attempt()* : Elle va permettre de se connecter, en passant en paramètre un email et un mot de passe (ou d'autres champs si vous avez modifier les champs nécessaires à la connexion). Cette fonction renvoie un objet **Response**, sur lequel on va pouvoir utiliser la fonction *isOk()* pour savoir si la connexion s'est bien passée.
- *check()* : Cette fonction ressemble à la fonction précédente, car elle va permettre de vérifier si les champs indiquées en paramètres correspondent à un utilisateur. La différence est simplement que cette fonction ne va pas connecter l'utilisateur derrière, elle permet simplement de vérifier les informations.
- *loggedIn()* : Elle permet de vérifier si un utilisateur est connecté
- *logout()* : Cette fonction va déconnecter l'utilisateur, et supprimer l'identifiant unique sur le serveur lié à cette utilisateur.
- *forget()* : Cette fonction permet de supprimer tous les identifiants uniques, ce qui amène tous les utilisateurs à se reconnecter.

La classe **Auth** apporte des fonctions supplémentaires liés à la session actuelle, qui sont décrite comme ceci dans la documentation :

```
// Récupérer l'utilisateur actuel  
auth()->user();
```



```
// Récupérer l'identifiant de l'utilisateur actuel
auth()->id();

// ou
user_id();

// Récupérer le 'User Provider' (UserModel par défaut)
auth()->getProvider();
```

Le *User Provider* correspond aux informations de l'utilisateur stockées avec le modèle *UserModel* par défaut comme indiqué.



# Fichiers

La gestion des fichiers se fait à l'aide de la classe **Media** qui est présent dans le dossier *Modules/Media*. Dans ce dossier, vous allez également retrouver l'utilitaire qui sert d'interface à la classe **Media**, FS.

Il existe deux possibilités pour la gestion des fichiers, soit utilisés FS, soit utiliser s3. Ici nous nous concentrerons à l'utilisation de FS car c'est ce système qui est présent par défaut dans Castopod. Mais vous pouvez modifier cela dans la variable *\$fileManagers* du fichier *Modules/Media/Config/Media.php*.

La classe **Media** indique donc les différents emplacements d'enregistrement des fichiers. Nous avons vu auparavant que [les fichiers étaient enregistrés par défaut dans le dossier \*public/media\*](#), mais vous pouvez modifier l'emplacement dans cette classe, ainsi que le nom des dossiers dans lesquels vont être enregistrés les documents des podcasts ou des personnes, dans la variable *\$folders*.

Si nous regardons maintenant la classe **FS**, qui est utilisé par **Media** pour enregistrer ou supprimer des fichiers, les renommer, récupérer leur contenu, etc. Chaque fichier n'est pas identifié par son nom mais par une clé unique. Nous verrons ensuite comment Castopod lie cette clé à son nom.

Voici les fonctions principales de cette classe :

- *save()* : Cette fonction va permettre d'enregistrer un fichier dans le dossier indiqué dans la classe **Media**. Elle attend en paramètre un objet **File**, et la clé du fichier.
- *delete()* : Elle va permettre de supprimer un fichier grâce à la clé.
- *rename()* : Cette fonction va permettre de changer une clé dans une autre clé.
- *getFileContents()* : Cette fonction va retourner le contenu d'un fichier, grâce à la clé indiquée en paramètre de la fonction. Vous pouvez retrouver l'ensemble des possibilités de cette classe dans le fichier d'interface situé dans *Modules/Media/FileManagers/FileManagerInterface.php*.

Nous vous avons parlé de la classe **File**, qui est une classe écrite par Codelgniter qui permet d'interagir avec un fichier. Elle est une classe étendue d'une classe de base de PHP, **SpIFileInfo**, qui permet elle aussi d'interagir avec des fichiers. Nous ne nous attarderons cependant pas sur ces deux classes, pour éviter de créer des confusions et simplifier cette documentation, mais si vous souhaitez plus d'informations sur les possibilités de ces classes, nous vous redirigeons vers [la page de documentation associée de Codelgniter](#).

Toute la gestion de clé est gérée également par Codelgniter, via son système de gestion des fichiers. Lorsqu'un fichier va être envoyé à Codelgniter, lors de l'enregistrement du fichier, une



fonction *getRandomName()* va être appelé pour générer un nom aléatoire à ce fichier. Ce nom est généré à partir du temps et d'un nombre aléatoire généré :

```
public function getRandomName(): string
{
    $extension = $this->getExtension();
    $extension = empty($extension) ? '' : '.' . $extension;

    return Time::now()->getTimestamp() . '_' . bin2hex(random_bytes(10)) . $extension;
}
```

Une fois ce nom généré, le fichier va pouvoir être enregistré dans le bon dossier. Ensuite c'est au niveau de la base de données qu'il va falloir s'intéresser.

Si on prend comme exemple l'ajout d'un épisode, deux tables vont être concernées :

- La table *episodes*, qui contient les informations liées à un épisode. On va retrouver dans les attributs un identifiant nommé *audio\_id*, qui est un entier.
- La table *media*, qui contient l'ensemble des fichiers audios et les images de Castopod. Là dedans, on va retrouver pour chaque fichiers une clé, qui correspond à l'emplacement du fichier (qui est la clé utilisée dans les fonctions citées précédemment), et un identifiant nommé *id*. C'est grâce à cette identifiant que le lien va pouvoir être fait entre l'épisode et ce fichier.



# Podcast et Épisodes

## Podcast

Dans l'organisation de Castopod, un site peut avoir un ou plusieurs podcast(s), qui eux-même pourront accueillir un ou plusieurs épisode(s). Ceux sont les épisodes qui vont être liés aux fichiers audios, un podcast n'étant finalement qu'un regroupement d'épisodes ayants le même thème.

Il existe deux manières de disposer les épisodes, soit sous forme épisodique, ce qui signifie que ceux seront les épisodes les plus récents seront ceux présentés en premier, soit sous forme de série où la présentation sera l'inverse.

Lors de la création d'un podcast, on peut participer au programme Open Podcast Prefix Project, nommé OP3, qui est un service gratuit et open-source d'analyse de podcasts. Vous pouvez retrouver plus de détails sur [leur page directement](#).

On peut également définir le podcast comme *Premium*, ce qui signifie que les épisodes ne seront pas affichés aux visiteurs, il faudra être identifié pour pouvoir accéder aux épisodes (en sachant que lorsqu'un podcast est en *Premium*, cela signifie que les épisodes sont par défaut en *Premium* aussi, mais que cet attribut peut être changé pour chaque épisode).

C'est la table *podcasts* qui contient les données de tous les podcasts dans la base de données. On va retrouver l'ensemble des informations rentrées par l'utilisateur, ainsi que des informateurs sur le créateur du podcast. En effet, on retrouve l'identifiant d'utilisateur pour chaque podcast pour identifier le créateur, ainsi que des informations sur.

La partie Podcast est divisée en deux :

- Tout ce qui va concerner l'affichage pour les visiteurs
- Tout ce qui permet de gérer les podcasts du site pour les administrateurs

Pour la partie des visiteurs, les fichiers sont présents dans le dossier *App*, tandis que pour la partie administration, les fichiers sont présents dans le dossier *Modules/Admin*.

On va retrouver dans les deux un contrôleur *PodcastController*, qui s'appuie sur un modèle interagissant avec la base de données, situé dans *App/Models/PodcastModel.php*.

Les vues sont aussi situées dans deux endroits différents. Pour la partie des visiteurs les vues sont situés dans le dossier *themes/cp\_app/podcast*, tandis que pour la partie administration, elles sont situées dans le dossier *themes/cp\_admin/podcast*.



Enfin, il existe un contrôleur dans la partie administration qui permet de gérer les différentes autorisations sur ce podcast, nommé *PodcastPersonController*. Ce contrôleur communique avec le modèle *PersonModel*, qui va permettre d'enregistrer ou de supprimer un lien entre le podcast et l'utilisateur sur la table *podcasts\_persons* de la base de données.

# Épisodes

On va retrouver dans un style similaire la même organisation pour les épisodes. Une partie pour les visiteurs, situées dans le même sous-dossier que pour les podcasts, et idem pour la partie administration.

Les épisodes vont récupérer leurs informations dans la table *Episodes*, qui contient non seulement les informations rentrées par l'utilisateur mais également les identifiants vers les fichiers audios et les images associés à cet épisode.

Comme pour le podcast, on va retrouver un contrôleur *EpisodePersonController* qui va permettre de gérer les autorisations sur un épisode, grâce aux fonctions *attemptCreate()* et *remove()*. Ce contrôleur communique avec le même modèle que pour les podcasts, mais ce modèle enregistrera les liens sur une table différente nommée *episodes\_persons*.



# Ajout de Polytech Nantes



# Création d'un flux général pour une instance Castopod

Dans la version actuelle de Castopod, il est possible de récupérer le flux RSS d'un épisode pour pouvoir utiliser ce protocole pour la gestion d'outils d'abonnement. Mais il n'est pas encore possible de créer un flux global rassemblant les flux RSS de chaque épisode dans une instance de Castopod. Il faut donc que chaque utilisateur récupère le flux RSS d'un épisode, et qu'il gère de son côté la récupération de multiples flux pour pouvoir être avertie de la sortie d'un nouvel épisode.

C'est dans cette problématique que Jet Asso nous a demandé de faire en sorte d'avoir un flux global RSS contenant l'ensemble des flux de chaque épisode. Et après quelques recherches, nous avons pu trouver un outil adapté pour cette demande : FreshRSS.

**FreshRSS** est un agrégateur de flux RSS open source et auto-hébergé qui vous permet de suivre facilement vos flux RSS préférés à partir d'une seule interface. Nous allons vous montrer comment installer FreshRSS à l'aide de Docker et comment l'utiliser pour suivre les flux RSS du Castopod de l'association Jet.

## Installation de Fresh RSS

### Prérequis

Assurez-vous que Docker est installé sur votre système. Si ce n'est pas le cas, vous pouvez suivre les instructions d'installation de Docker Desktop disponibles sur le site de [Docker](#).

### Installation avec Docker

Ouvrez une fenêtre de terminal, allez dans un dossier où les différentes données de FreshRSS seront stockés, et exécutez la commande suivante pour lancer le conteneur FreshRSS :

```
docker run -d --restart unless-stopped --log-opt max-size=10m \  
-p 8080:80 \  
-e TZ=Europe/Paris \  
-e 'CRON_MIN=1,31' \  
-v freshrss_data:/var/www/FreshRSS/data \
```



```
-v freshrss_extensions:/var/www/FreshRSS/extensions \
--name freshrss \
freshrss/freshrss
```

Dans cette commande, nous spécifions :

- Le port sur lequel FreshRSS sera accessible (8080 dans cet exemple).
- Le fuseau horaire (Europe/Paris).
- La configuration de la tâche cron pour mettre à jour les flux toutes les 30 minutes (à la 1ère et à la 31ème minute de chaque heure).
- Les volumes pour stocker les données de FreshRSS et les extensions.

Une fois la commande exécutée, vous pouvez accéder à FreshRSS en ouvrant votre navigateur et en saisissant `http://localhost:port`.

## Configuration initiale

La première fois que vous accédez à FreshRSS, vous devez effectuer une configuration initiale. Suivez les instructions à l'écran pour configurer votre compte utilisateur et vos préférences.

# Utilisation de FreshRSS

## Abonnement à l'ensemble des flux de Castopod

Pour suivre les flux RSS de l'association Jet, vous devez vous abonner aux flux RSS correspondants. Vous pouvez télécharger le fichier `subscriptions.xml` contenant tous les flux RSS de Castopod en cliquant [ici](#).

Pour importer le fichier subscriptions.xml dans FreshRSS, suivez ces étapes :

1. Cliquez sur l'icône "Gestion des abonnements" dans le menu latéral gauche.
2. Cliquez sur l'onglet "Importer / Exporter".
3. Dans la section "Importer", cliquez sur le bouton "Parcourir" et sélectionnez le fichier subscriptions.xml que vous avez téléchargé précédemment.
4. Cliquez sur le bouton "Importer" pour importer les flux RSS.





Gestion des abonnements



Flux principaux

Flux importants

Articles favoris (0)

Mes étiquettes

Subscriptions

## Importer

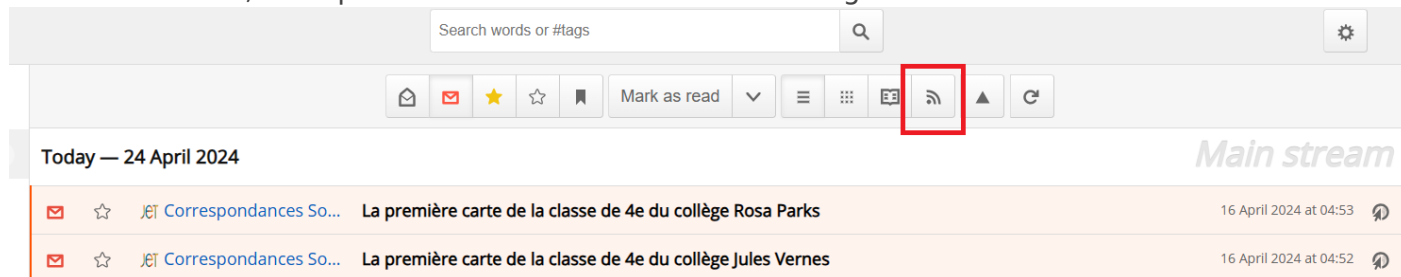
Fichier à importer  
(OPML, JSON ou ZIP)

Choisir un fichier subscriptions.xml

Importer

Une fois que vous avez importé les flux RSS, vous pouvez actualiser le flux, ce qui permettra de charger tous les flux que vous venez d'importer et de les afficher sur la page d'accueil de FreshRSS.

Une fois cela fait, vous pouvez enfin accéder à un flux RSS global :




Et c'est ce lien que vous pourrez partager à vos utilisateurs pour qu'ils puissent accéder aux flux RSS global que vous avez configuré.

## Ajout d'un nouveau flux RSS

Si un nouveau podcast est ajouté à Castopod, celui-ci ne sera pas présent dans le fichier `subscriptions.xml` et vous devrez l'ajouter à la main. Pour ce faire :



1. Cliquer sur le petit  à côté de "Gestion des abonnements"
2. Copier le lien du flux RSS du podcast dans "Ajouter un flux"
3. Choisissez la catégorie
4. Cliquer sur "Ajouter"
5. Dans la page qui s'ouvre, vous pouvez cliquer sur "Valider" directement sauf si vous voulez modifier certains paramètres

Et voilà ! Vous avez importé l'ensemble des flux RSS présents sur le Castopod de l'association Jet (au 22/04/2024).

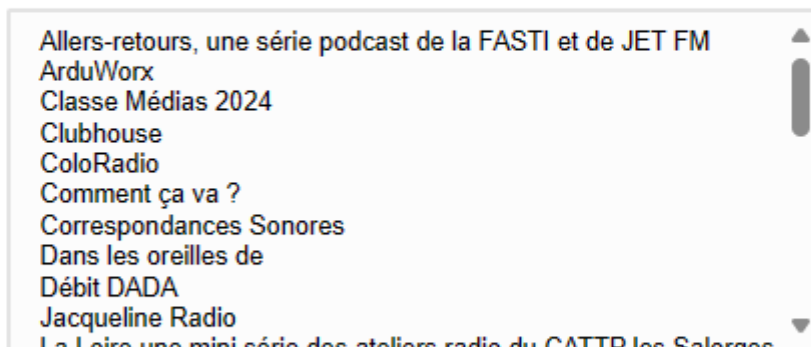
## Ajout d'un utilisateur

Une fois que le compte administrateur est correctement abonné à tous les flux RSS du Castopod de l'association, il est possible de créer de nouveaux comptes pour les personnes intéressées. Pour ce faire, cliquez sur l'icône en forme d'écrou en haut à droite, puis sélectionnez "Gestion des utilisateurs". Vous pouvez alors créer un nouvel utilisateur en remplissant les champs requis.

Pour que le nouvel utilisateur puisse également s'abonner à tous les flux RSS, le compte administrateur peut lui partager son fichier d'abonnement. Pour ce faire, l'administrateur doit retourner dans l'onglet "Importer / Exporter" de la gestion des abonnements et exporter sa liste d'abonnements en décochant les options "Exporter les articles étiquetés" et "Exporter les favoris".

### Exporter

- ☒ Exporter la liste des flux (OPML)
- ☐ Exporter les articles étiquetés
- ☐ Exporter les favoris



Exporter



Le nouvel utilisateur n'a alors plus qu'à importer le fichier généré par l'administrateur en suivant la même méthode que celle présentée précédemment dans la section "Utilisation". Il pourra ainsi s'abonner facilement à tous les flux RSS partagés par l'administrateur.

# Extensions

Nous vous recommandons de vous pencher sur les extensions disponibles pour FreshRSS, que vous pouvez retrouver sur [cette page](#). Chaque extension que vous téléchargez pourra être mis dans le dossier *freshrss\_extensions* du dossier utilisé pour le déploiement du conteneur.

# Sources

- [Github de FreshRSS](#)
- [Site web de FreshRSS](#)



# Connexion à Nextcloud

## Connexion à Nextcloud via Castopod

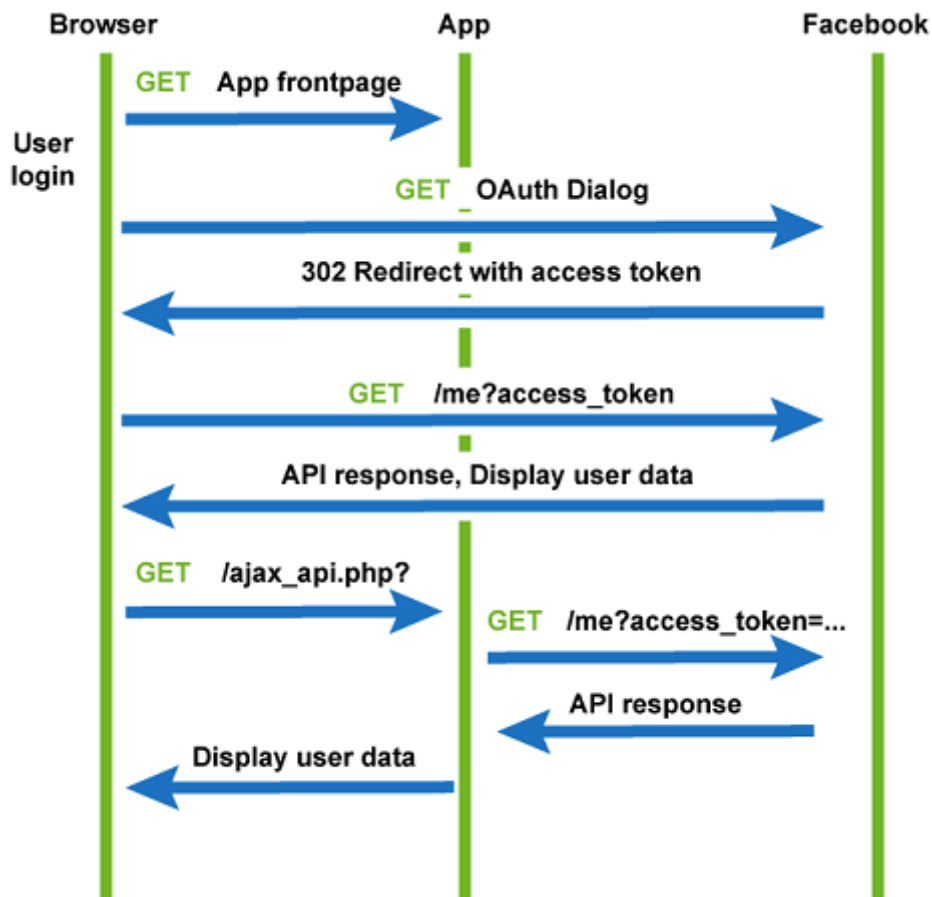
### 1. Objectifs :

Permettre à l'utilisateur de se lier à Castopod son compte Nextcloud, offrant ainsi une expérience de connexion simplifiée et sécurisée.

### 2. Explication :

Pour réaliser cette fonctionnalité, nous avons mis en place un système d'authentification basé sur le protocole **OAuth**, qui implique un échange sécurisé de clés entre le client (Castopod) et le serveur (Nextcloud).





Lorsque l'utilisateur choisit de se connecter via Nextcloud, un contrôleur spécifique est déclenché pour initier le processus. Ce contrôleur effectue une requête GET afin d'afficher la page de connexion de Nextcloud, en incluant des paramètres tels que l'identifiant du client (idClient) et de serveur (idServeur), préalablement configurés dans les paramètres d'administration de Nextcloud.

Une fois que l'utilisateur autorise l'accès sur la page de connexion de Nextcloud, Nextcloud renvoie à Castopod un access\_token. Ce token est utilisé par Castopod pour obtenir une clé API ainsi qu'un refresh\_token. Ce dernier peut être utilisé ultérieurement pour obtenir une nouvelle clé API lorsque celle en cours expirera.

Pour implémenter cette fonctionnalité, plusieurs éléments ont été ajoutés au projet :

- **Contrôleur d'authentification (AuthController) :** Ce contrôleur est responsable de l'interaction avec le protocole OAuth. Il gère les requêtes et les réponses nécessaires à l'authentification via Nextcloud.
- **Entité OAuth :** Cette entité contient les informations requises pour le protocole OAuth, telles que les clés d'identification du client et du serveur, ainsi que des fonctions pour l'utilisation du protocole.



- **Modèle OAuth (OAuthModel) :** Ce modèle permet d'interagir avec la table de la base de données pour stocker et récupérer les informations relatives à l'authentification OAuth.
- **Utilisation de fonctions externes :** Pour garantir l'implémentation, notre projet s'appuie sur des fonctions fournies par un [projet Git de Bahuma20](#). Ces fonctions offrent des interfaces PHP CodeIgniter pour la gestion du protocole OAuth.



# Insertion d'image flexible

## Insertion d'images flexibles dans Castopod

### Objectif :

Permettre à l'utilisateur d'insérer des images dans Castopod, même si elles ne respectent pas les ratios (1:1) et la taille minimale requise (1400px), en fournissant une interface de recadrage et en ajustant automatiquement les dimensions si nécessaire.

### Explication :

A la base, le formulaire de la page envoie les données de l'épisode (dont l'image) à un contrôleur ( `EpisodeController` ) via la route `episode-create` :

```
<form action="<?= route_to('episode-create', $podcast->id) ?>" method="POST" enctype="multipart/form-data" class="flex flex-col w-full max-w-xl mt-6 gap-y-8" id="episode-form">
```

```
$routes->post(
    'new',
    'EpisodeController::attemptCreate/$1',
    [
        'filter' => 'permission:podcast#.episodes.create',
    ],
);
```

Il fallait donc proposer à l'utilisateur cette interface, changer l'image du formulaire avant de l'envoyer, et gérer ce que renvoie le contrôleur.

Pour gérer l'interface, il faut du JavaScript (JS) car c'est du code dynamique, qui va changer tout le temps, contrairement à PHP qui charge juste au début de la page. Le script Croppie permet de faire cette interface, de manière simple.



```

// Fonction pour afficher l'interface lorsqu'une image est insérée
var $uploadImage = document.getElementById('cover');
var $imagePreview = document.getElementById('imagePreview');
var croppie = new Croppie($imagePreview, {
  viewport: { width: 200, height: 200, type: 'square' },
  boundary: { width: 300, height: 300 },
});

$uploadImage.addEventListener('change', function () {
  $imagePreview.classList.remove('hidden');
  var reader = new FileReader();
  reader.onload = function (e) {
    croppie.bind({
      url: e.target.result,
    });
  };
  reader.readAsDataURL(this.files[0]);
});

// Fonction pour obtenir l'image croppée sous forme de fichier
function getCroppedImageAsFile(croppieInstance) {
  return new Promise(resolve => {
    croppieInstance.result({
      type: 'blob',
      format: 'jpeg',
      size: 'original',
    }).then(blob => {
      const file = new File([blob], 'cropped_image.jpeg', { type: 'image/jpeg' });
      resolve(file);
    });
  });
}

```

Pour gérer la taille de l'image, on utilise la fonction `resizeImage`, qui permet de changer la taille d'une image.

Une fois que l'utilisateur soumet le formulaire, nous bloquons l'envoi direct par la page. Étant donné que la nouvelle image est générée par JavaScript, l'envoi direct des données n'est pas possible. Nous sommes donc contraints de gérer l'envoi de la requête en JavaScript et de traiter la réponse. Comme cette réponse ne peut pas être un objet PHP, nous devons la gérer entièrement côté client.



```

document.getElementById('episode-form').addEventListener('submit', async function (event) {
    event.preventDefault();

    var formData = new FormData(this);

    if(document.getElementById('cover').value !== ''){
        // Récupérez l'image croppée sous forme de fichier
        const croppedFile = await getCroppedImageAsFile(croppie);

        // Redimensionnez l'image à une largeur de 1400px si nécessaire
        const resizedFile = await resizeImage(croppedFile, 1400);

        // Ajoutez le fichier redimensionné au formulaire FormData
        formData.set('cover', resizedFile);
    }

    try {
        const response = await fetch("<?= route_to('episode-create', $podcast->id) ?>", {
            method: "POST",
            body: formData,
        });
    }

```

Auparavant, lorsque les données étaient envoyées directement depuis la page sans JavaScript, l'objet renvoyé était un `RedirectResponse`, géré en arrière-plan par un contrôleur. Cependant, dans ce scénario, cette approche n'est pas applicable. Nous devons plutôt utiliser un type d'objet compatible avec JavaScript, comme un JSON. Ce JSON peut prendre deux formes :

- En l'absence d'erreur : il contient un champ "id" qui représente l'URL de l'épisode créé. Dans ce cas, l'URL retournée est chargée.

```

const idUrl = responseData.id;
console.log(idUrl);
// Open the new episode view page
const baseUrl = '<?= base_url('/') ?>'.slice(0, -1);
window.location.href = baseUrl + idUrl;

```

- En présence d'une erreur : un champ error qui contient les erreurs retournées par le contrôleur gérant l'upload de l'épisode. Dans ce cas, la page est rafraîchie avec un champ expliquant l'erreur au début de la page, à l'aide d'une variable `GET error` contenant l'erreur.



```

<?php
// Vérifier s'il y a une erreur, et dans ce cas l'afficher
if (isset($_GET['error'])) {
    // Décoder la chaîne URL avant de la diviser
    $decodedError = urldecode($_GET['error']);

    // Diviser la chaîne en un tableau basé sur le caractère "+"
    $errorMessages = explode('+', $_GET['error']);

    // Supprimer les éléments vides du tableau
    $errorMessages = array_filter($errorMessages);

    // Afficher un div contenant un paragraphe pour chaque élément du tableau
    echo '<div class="flex flex-col gap-x-2 gap-y-4 md:flex-row">';
    echo '<fieldset class="w-full p-8 bg-red-200 border-3 flex flex-col items-start border-red-600 rounded-xl ">';
    foreach ($errorMessages as $errorMessage) {
        echo '<p> Erreur : ' . $errorMessage . '</p>';
    }
    echo '</fieldset>';
    echo '</div>';
}
?>

```

Un problème rencontré en cas d'erreur est la perte des données saisies par l'utilisateur car la page est rechargée. Pour remédier à cela, nous effectuons des vérifications avant d'envoyer les données du formulaire. Nous vérifions que les champs obligatoires sont correctement remplis par l'utilisateur. Si un champ (ou plusieurs) est manquant, une erreur est affichée et la page est automatiquement ramenée à l'emplacement du champ manquant.

```

const audioFile = document.getElementById('audio_file').value;
const title = document.getElementById('title').value;
const description = document.getElementById('description').value;

// Vérifiez si les champs requis sont vides
const emptyFields = findEmptyFields([
    { id: 'audio_file', label: 'Audio File' },
    { id: 'title', label: 'Title' },
    { id: 'description', label: 'Description' }
]);

```



```
if (emptyFields.length > 0) {  
  const errorMessage = `Veuillez remplir les champs obligatoires : ${emptyFields.map(field =>  
field.label).join(', ')}.`;  
  alert(errorMessage);  
  
  // Faites défiler la page jusqu'au premier champ manquant  
  const firstEmptyField = emptyFields[0].id;  
  const element = document.getElementById(firstEmptyField);  
  const elementPosition = element.getBoundingClientRect().top + window.scrollY - window.innerHeight / 2;  
  window.scrollTo({ top: elementPosition, behavior: 'smooth' });  
}
```

En conclusion, pendant que le script JavaScript gère l'envoi du formulaire et la réponse associée, nous désactivons le bouton d'envoi dès que l'utilisateur appuie dessus. Cette désactivation évite que l'utilisateur n'appuie plusieurs fois sur le bouton, ce qui pourrait entraîner l'envoi répété de la même requête. De plus, un indicateur visuel sous forme d'un rond de chargement apparaît pour informer l'utilisateur que le traitement est en cours.

```
// Baisser l'opacité de la page sauf pour le cercle de chargement  
document.getElementById('all').style.opacity = '0.5';  
  
// Désactiver le bouton  
const submitButton = document.getElementById('submit-button');  
submitButton.disabled = true;  
  
// Afficher le cercle de chargement  
const loadingCircle = document.getElementById('loading-circle');  
loadingCircle.style.display = 'block';
```



# Sondage

## Page de Sondage sur Castopod

### Introduction

La page de sondage sur Castopod constitue une fonctionnalité toujours en cours de développement au sein de l'écosystème Castopod. Initialement conçue dans le cadre d'une prise en main du développement sur Castopod, cette fonctionnalité est considérée comme secondaire, sans nécessité d'améliorations majeures étant donné qu'elle n'est pas jugée cruciale pour le client.

### Fonctionnalités Principales

- **Création de Sondages:** Les utilisateurs ont la possibilité de créer des sondages directement depuis l'interface de Castopod.
- **Gestion des Sondages:** La page de sondage permet la gestion des sondages existants, y compris leur édition et leur suppression si nécessaire.
- **Participation aux Sondages:** Les auditeurs peuvent participer aux sondages en votant pour leurs choix préférés.

### Architecture Technique

- **Frontend:** La page de sondage est implémentée en utilisant du HTML, CSS et JavaScript.
- **Backend:** Le backend de la fonctionnalité de sondage est développé en utilisant le framework PHP CodeIgniter, intégré à l'écosystème de Castopod. Il gère la logique métier, y compris la création, la modification et la suppression des sondages, ainsi que la gestion des votes des utilisateurs.

### Limitations Connues

Étant considérée comme une fonctionnalité secondaire, la page de sondage ne bénéficie pas de ressources prioritaires pour son développement et son amélioration.