

# Fonctionnement de Castopod

- [Routes](#)
- [Contrôleurs](#)
- [Vues](#)
- [Fonctionnalités supplémentaires](#)
- [Authentifications et autorisations](#)
- [Fichiers](#)
- [Podcast et Épisodes](#)

# 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 commencent 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 liée 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](#).

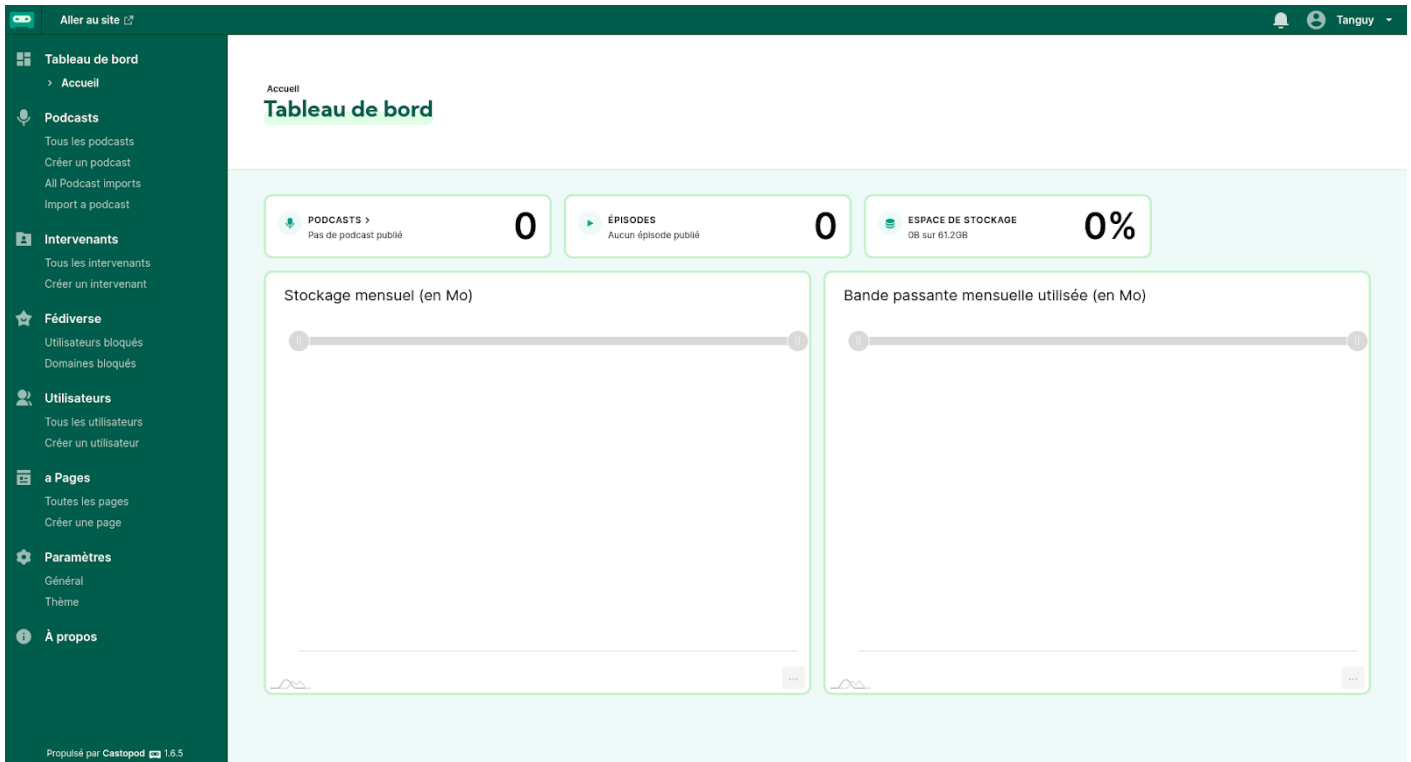
## Layout

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 composants 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 Codelgniter.

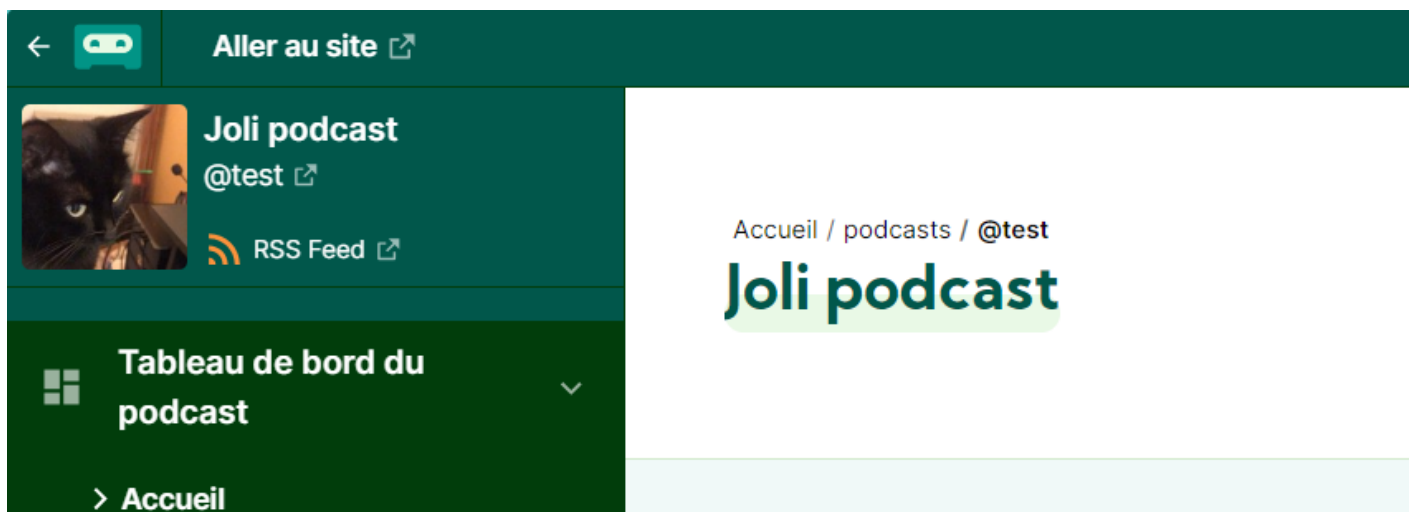
Les Helpers qui peuvent être utilisés sont écrits dans le fichier *App/Config/Autoload.php*, dans la variable **\$helpers**. On y retrouve par exemple pour Castopod un Helper pour la gestion des authentifications. Mais on peut également intégrer un Helper directement dans un contrôleur, à l'aide de la 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 **Breadcrump.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 CodeIgniter qui permet d'interagir avec un fichier. Elle est une classe étendue d'une classe de base de PHP, **SplFileInfo**, 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 CodeIgniter](#).

Toute la gestion de clé est gérée également par CodeIgniter, via son système de gestion des fichiers. Lorsqu'un fichier va être envoyé à CodeIgniter, 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 ayant 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 informations 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 situées 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*.