

Documentation de Podz

Application web de publication de podcasts

Podz

Projet TPI - 2022

Samuel Roland

Table des matières

Analyse préliminaire

- Analyse préliminaire
 - Introduction
 - Objectifs
 - Planification initiale
- Analyse / Conception
 - Concept
 - Technologies utilisées
 - Base de données: MCD
 - Base de données: MLD
 - Maquettes
 - Choix de conception
 - Stratégie de test
 - Où sont écrits les tests ?
 - Couverture des tests
 - Résultats des tests
 - Prérequis pour lancer les tests
 - Comment lancer les tests ?
 - Planification
 - Dossier de conception
 - Upload d'un fichier audio d'épisode
 - Composants réutilisables
- Réalisation
 - Dossier de réalisation
 - Description des tests effectués
 - Erreurs restantes
 - Liste des documents fournis
- Conclusions
- Annexes
 - Résumé du rapport du TPI / version succincte de la documentation
 - Sources – Bibliographie
 - Journal de travail
 - Manuel d'Installation
 - Archives du projet

Introduction

Podz est une application web de publication de podcasts, pour le projet de TPI de Samuel Roland. Les auteurs de podcasts peuvent créer des podcasts, publier des épisodes, planifier la publication d'épisodes dans le futur et cacher des épisodes. L'application est basée sous Laravel 9 et part de zéro.

Objectifs

Voici la liste des objectifs à atteindre, tirée du cahier des charges:

Fonctionnalités générales (reprises d'un ancien projet)

- Création de comptes utilisateurs.
- Authentification des utilisateurs.
- Il n'y aura pas de partie back-office ni de rôle administrateur.

Ces fonctionnalités sont implémentées par Jetstream, je n'ai donc pas eu besoin de m'en occuper.

Fonctionnalités détaillées selon le type d'utilisateur

- En tant que visiteur (personne non authentifiée) :
 - Consultation de la liste des podcasts.
 - Consultation du détail d'un podcast : épisodes
 - Ecoute d'un épisode d'un podcast.
- En tant qu'utilisateur authentifié, en plus des fonctionnalités accessibles à tout visiteur :
 - Création d'un nouveau podcast, édition d'un de ses podcasts existant.
 - Sur l'un de ses podcasts :
 - Affichage de la liste des épisodes avec toutes les données liées.
 - Ajout d'un nouvel épisode.
 - Edition d'un épisode.
 - Suppression d'un épisode.

Planification initiale

Analyse / Conception

Concept

TODO ????

L'application tourne en PHP sur un serveur Apache. Elle utilise une base de données MySQL pour stocker ses données.

Technologies utilisées

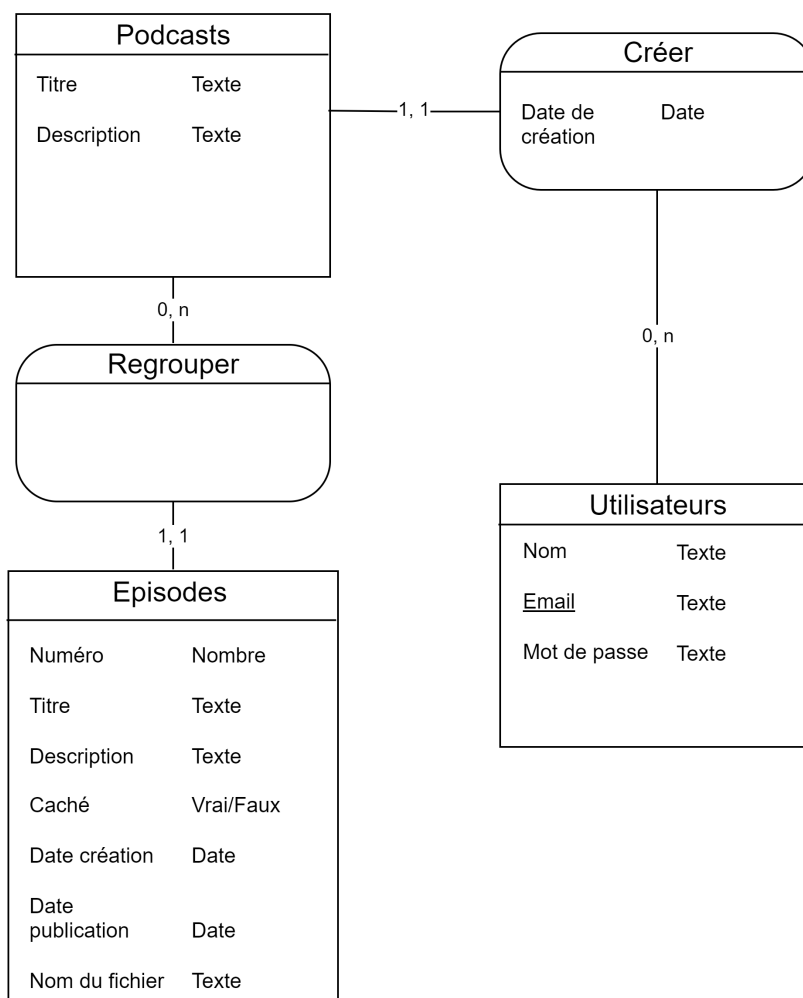
J'ai choisi la stack **TALL** (*TailwindCSS - AlpineJS - Livewire - Laravel*) pour ce projet, car je suis à l'aise avec ces 4 frameworks et parce qu'ils permettent d'être assez productif pour développer une application web.

Petits aperçus de ces frameworks

- **Laravel**: un framework PHP basé sur le modèle MVC et en POO. Laravel donne accès à beaucoup de classes et fonctions très pratiques, d'avoir une structure imposée, d'avoir des solutions toutes faites pour beaucoup de problèmes récurrents (traductions, authentification, gestion des dates et énormément d'autres). Tout ceci simplifie énormément le développement d'applications web en PHP une fois qu'on ait à l'aise avec les bases.
- **Livewire**: un framework pour Laravel permettant de faire des composants fullstack réactifs. L'idée est d'utiliser la puissance de Blade et du PHP pour avoir des parties réactives sur le frontend (normalement codé en Javascript) sans devoir coder des requêtes AJAX.
- **AlpineJS**: un petit framework Javascript relativement simple à apprendre, utilisée ici pour gérer certaines interactions que Livewire ne permet pas, ou qui concernent des états d'affichage (là où des requêtes sur le backend seraient inutiles notamment). Les composants s'écrivent inline (sur les balises HTML directement). Très pratique pour afficher un dropdown, faire une barre de progression, ...
- **TailwindCSS**: un framework CSS, concurrent de Bootstrap mais centré autour des propriétés CSS (en ayant des classes utilitaires - "utility-first") au lieu de tourner autour de composants. C'est très puissant pour construire rapidement des interfaces, en écrivant quasiment jamais de CSS pur et pour faire du responsive c'est très pratique parce qu'on peut préfixer toutes les classes par `md:` par ex. afin dire que la classe ne s'applique que sur les écrans medium et au dessus.
- **Jetstream**: Un starter Kit Laravel mettant en place les fonctionnalités d'authentification, tels que la connexion, la création de compte, la gestion du compte et beaucoup d'autres. L'option Livewire a été utilisée.

Base de données: MCD

Schéma: MCD Podz - TPI
Auteur: Samuel Roland
Version: v2
Date de version: 09.05.2022



En dehors des champs évidents, voici quelques aspects techniques qui demandent des explications.

Dans Episodes:

- Les combinaisons du Numéro et du podcast lié, ainsi que le titre et le podcast lié, sont uniques (exemple: on ne peut pas avoir 2 fois un épisode 4 du podcast "Summer stories", et on ne peut pas avoir 2 fois un épisode nommé "Summer 2020 review" du podcast "Summer stories").
- La date de création est définie par la date de création de l'épisode sur la plateforme (avec l'upload du fichier), peu importe ses autres informations (la publication ou l'état caché n'a

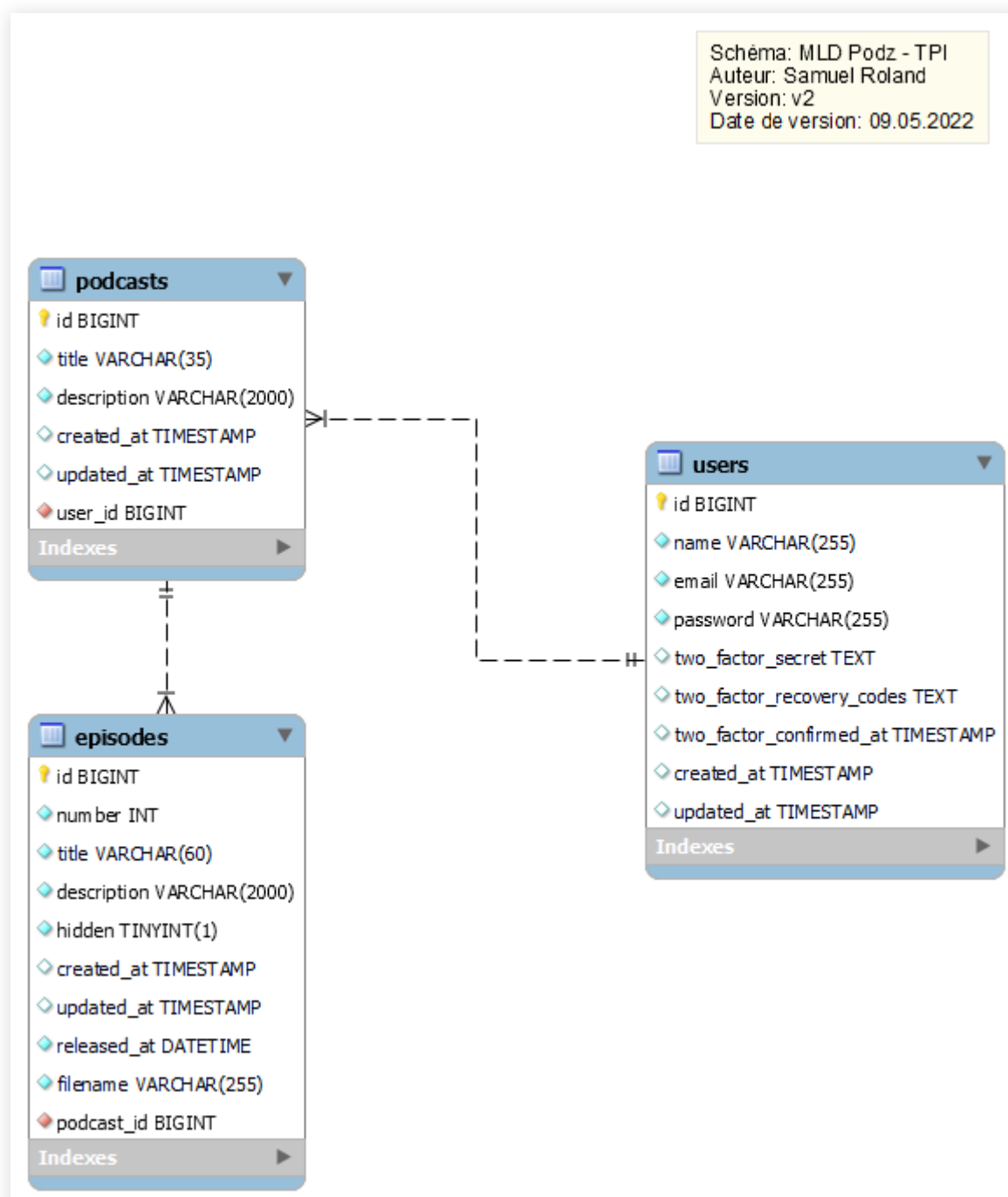
pas d'influence sur cette date). Cette date ne change jamais et est affichée qu'à l'auteur.

- La date de publication peut être dans le passé ou mais aussi dans le futur. Si elle est dans le futur, l'épisode n'est pas encore publié (jusqu'à la date définie). Ceci permet de programmer dans le futur une publication.
- Le champ Caché est par défaut à Faux et n'a pas d'effet dans ce cas. S'il est Vrai, l'épisode ne sera pas visible dans les détails du podcast.

Dans Podcasts:

- La combinaison du titre et de l'auteur est unique. Exemple: Michelle ne peut pas publier 2 podcasts s'appelant "My story", par contre Michelle et Bob peuvent chacun publier 1 podcast nommé "My story".

Base de données: MLD



Ce MLD n'a pas été fait à la main mais a été rétro-ingéniéré depuis la base de données, après avoir codé les migrations. Certains champs sont créés par une migration générée par Jetstream, je n'en ai pas besoin mais je ne vais pas les retirer au risque de casser certaines parties existantes. Ce MLD omet volontairement les tables générées par Laravel et propres à chaque application Laravel (`sessions`, `migrations`, ...), une partie provient de migrations créées par Jetstream. Ne vous étonnez donc pas de trouver d'autres tables dans la base de données, car je ne les utilise pas directement.

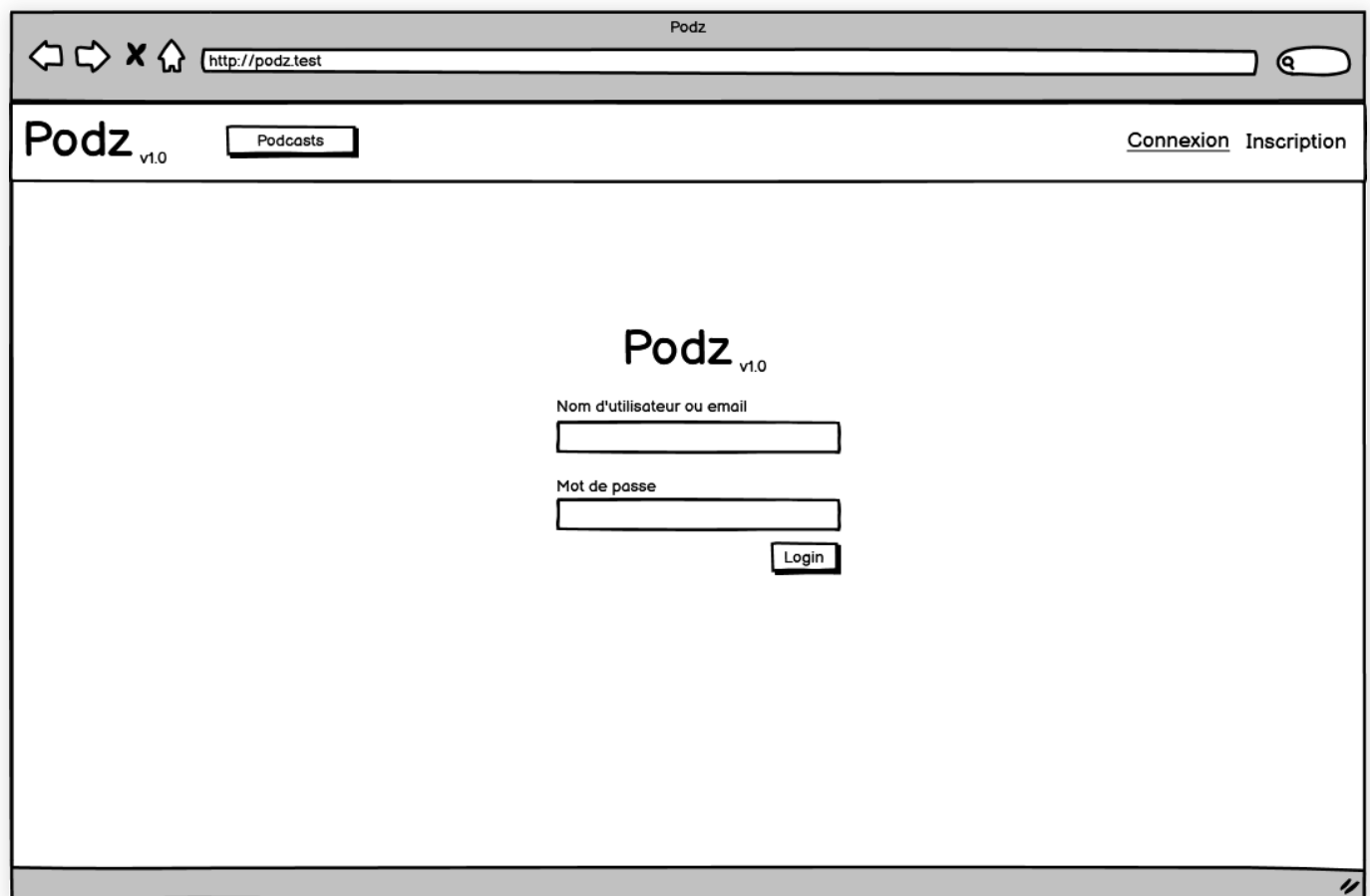
todo: documenter spécificités. Les champs `created_at` et `updated_at` sont gérés automatiquement par Laravel, je n'utilise que le `created_at` en lecture seulement.

Maquettes

Pour pouvoir utiliser les fonctionnalités requises, voici la liste complète des pages existantes et leur maquette.

- Page Connexion
- Page Inscription
- Page Liste des podcasts
- Page Détails d'un podcast (visiteur)
- Page Edition des détails d'un podcast (auteur)
- Page Détails d'un podcast (auteur)
- Page Création d'un podcast

Page Connexion



Maquette de la page de connexion (Page Connexion) pour Podz v1.0. L'interface est présentée dans un navigateur web avec l'adresse `http://podz.test`.

Le header de la page contient le logo **Podz** v1.0, un bouton **Podcasts**, et des liens [Connexion](#) et [Inscription](#).

Le contenu principal de la page est centré et comprend :

- Le logo **Podz** v1.0.
- Le texte "Nom d'utilisateur ou email" suivi d'un champ de saisie.
- Le texte "Mot de passe" suivi d'un champ de saisie.
- Un bouton **Login**.

Le footer de la page est gris et contient un logo de développement (//).

Page Inscription

Podz

Podcasts

Connexion

Inscription

Podz

Username

Email

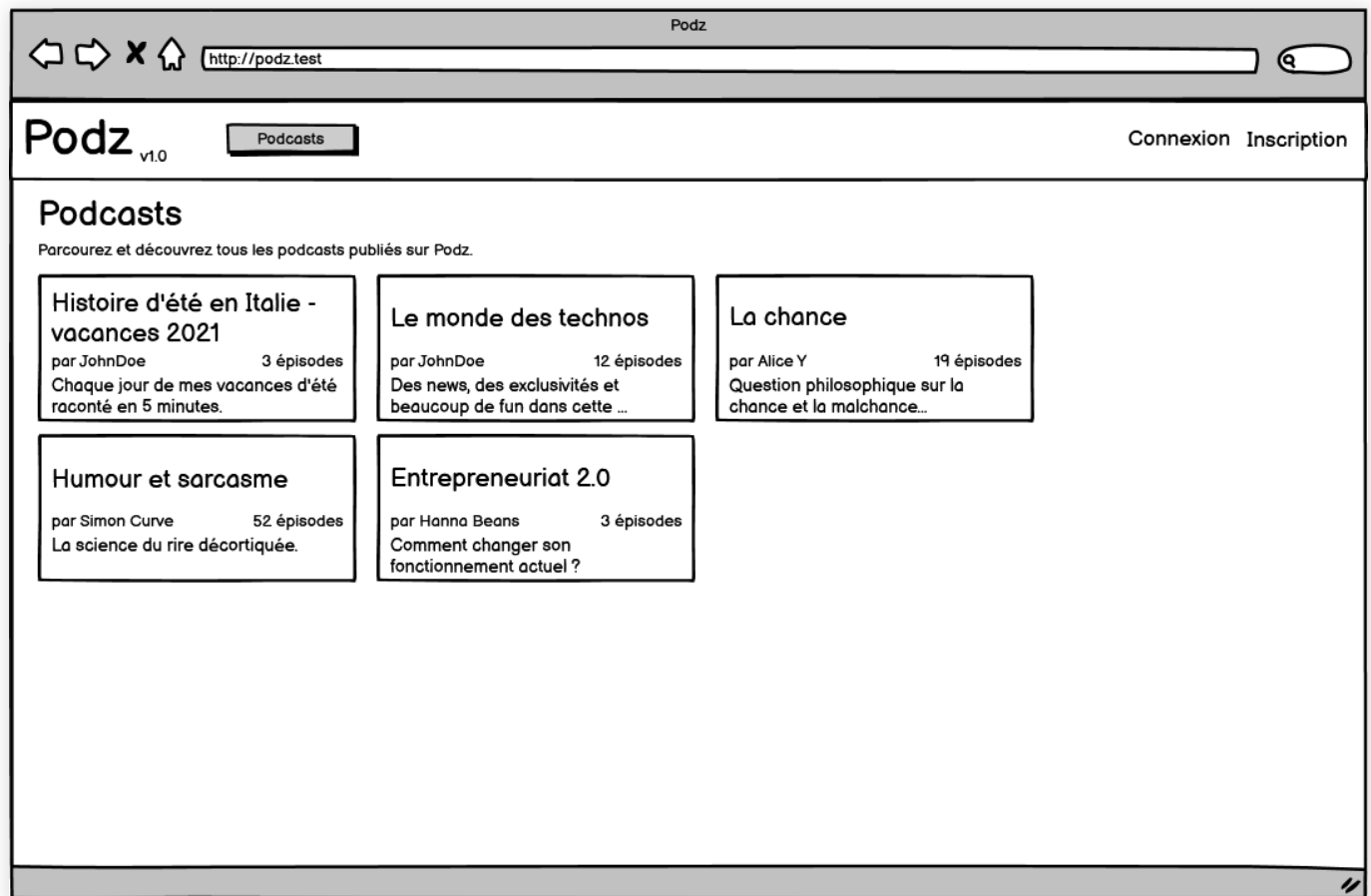
Password

Confirm Password

Register

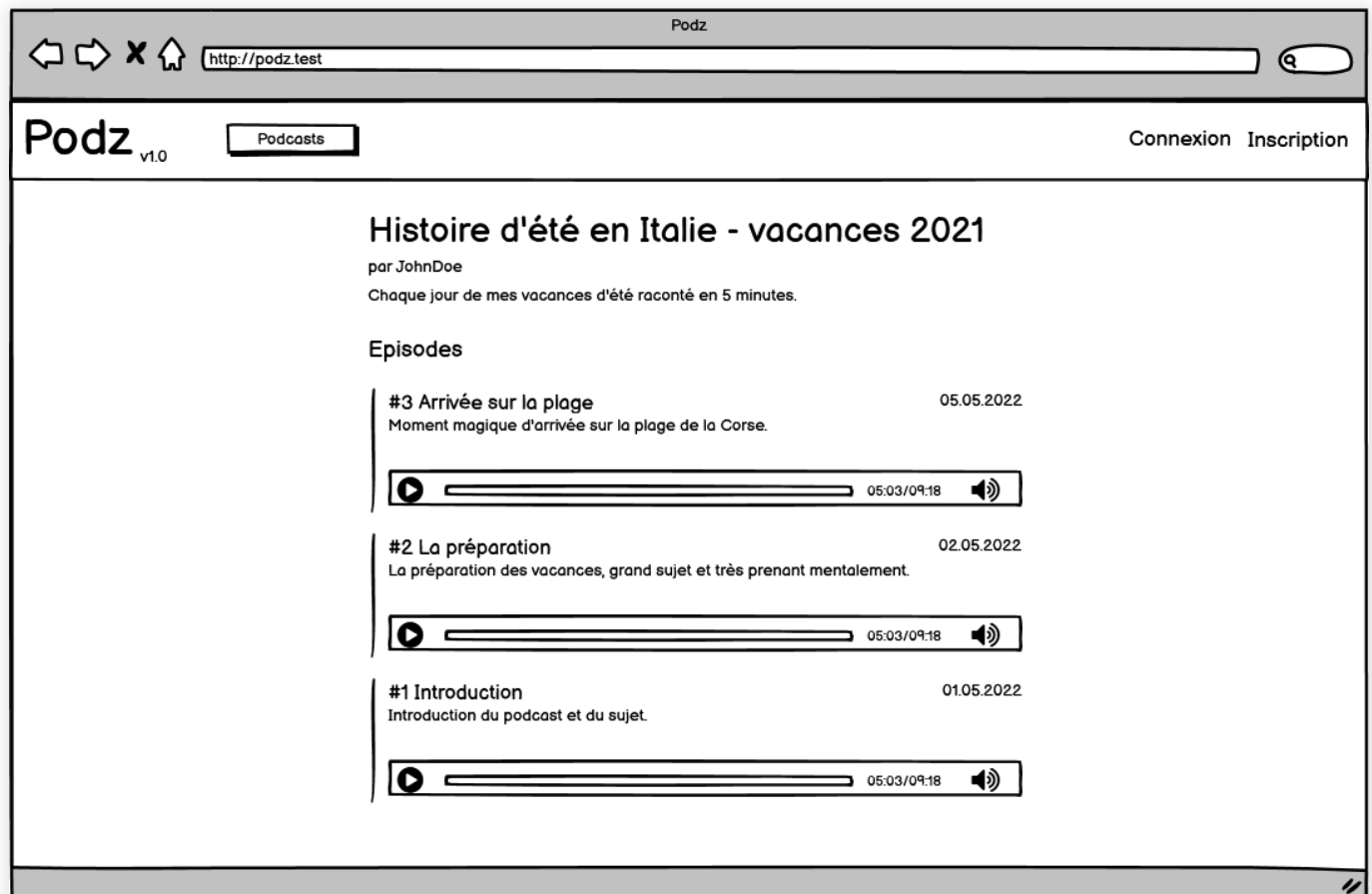
Page Liste des podcasts

Cette page est visible publiquement et c'est la page par défaut de l'application, on y accède également via le bouton Podcasts en haut à gauche. On peut cliquer sur un podcast pour accéder à ses détails.



Page Détails d'un podcast (visiteur)

Les visiteurs ne voient que les épisodes qui sont visibles et qu'une partie de leurs informations. Ils ne voient que le numéro, le titre, la description, l'audio et le date (arrondie au jour).



Page Edition des détails d'un podcast (auteur)

L'auteur d'un podcast peut gérer les détails de son podcast, autant le titre et la description que les détails et la liste des épisodes. Nous sommes le 09.05.2022 dans cette maquette, l'épisode 4 est caché et le 5 est planifié pour le 10.05.2022. Ici l'auteur crée un 5 ème épisode planifiée qui ne sera publié que le lendemain à 15h08. Il peut aussi éditer les anciens épisodes en cliquant sur l'icône de stylo, ce qui passe l'épisode en mode édition (et permet ainsi de modifier).

http://pretpi.test

Podz

v1.0

Podcasts

John Doe ▼

Histoire d'été en Italie - vacances 2021

par JohnDoe

Chaque jour de mes vacances d'été raconté en 5 minutes.

Enregistrer

Nouvel épisode

#5

Super découverte

☐ Caché

10/05/2022 15:08

Description

Fichier audio (mp3, m4a)

Select file...

Aperçu

05:03/09:18

Publier

Episodes

#4

Rencontre secrète

☒ Caché

08/05/2022 08:12

épisode surprise

Aperçu

05:03/09:18

Sauver

#3 Arrivée sur la plage

Créé le 05.05.2022

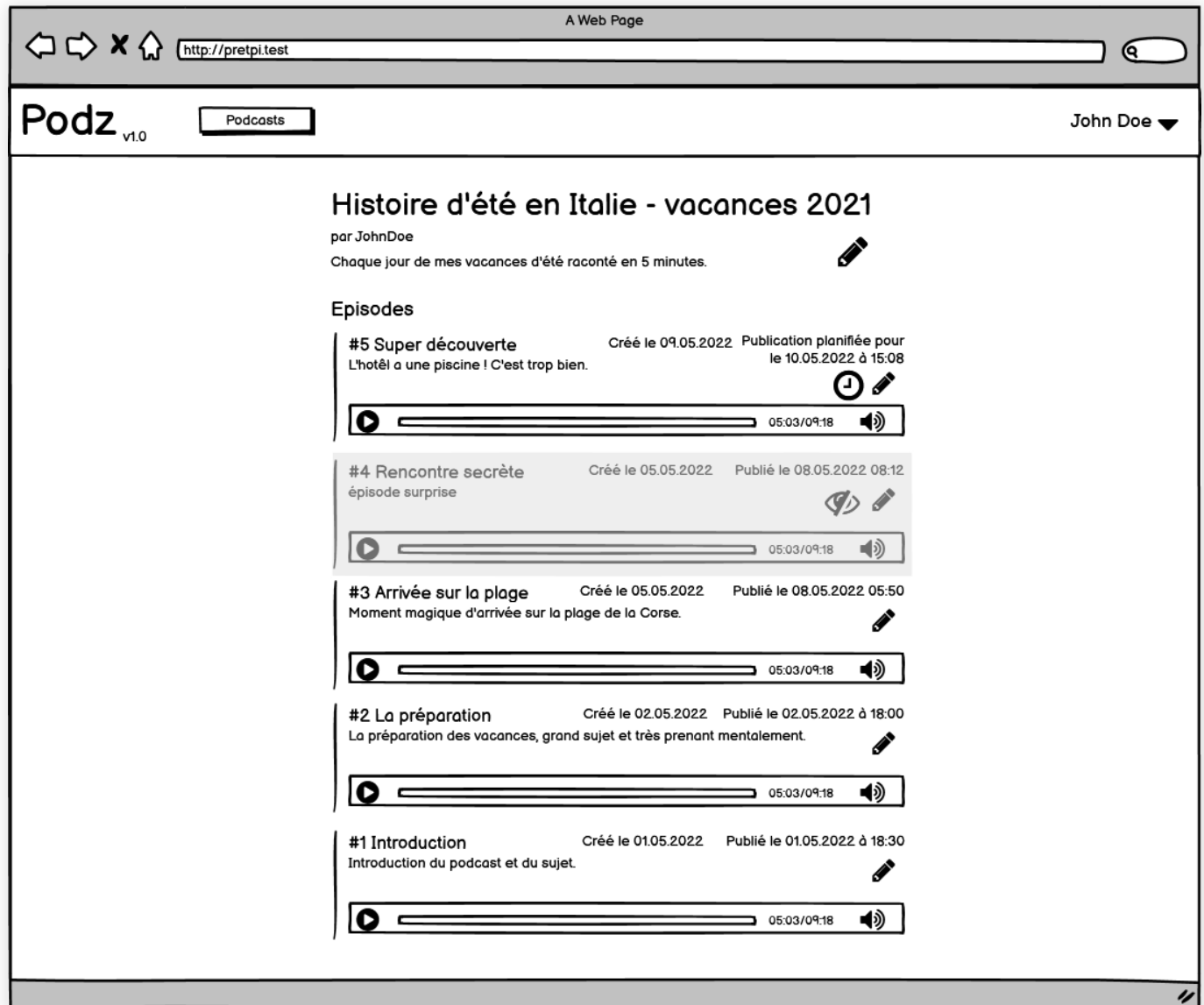
Publié le 08.05.2022 à 05:50

Moment magique d'arrivée sur la plage de la Corse.

05:03/09:18

Page Détails d'un podcast (auteur)

L'auteur voit évidemment toutes les informations de ses podcasts contrairement au visiteur. (Pour les podcasts d'autres auteurs, il voit la vue visiteur). Nous sommes le 10.05.2022 dans cette maquette, l'épisode 4 est caché et le 5 est planifié pour le 10.05.2022. L'épisode 4 est caché parce que l'auteur a décidé après coup de le remettre en privé.



Page Création d'un podcast

Simple formulaire pour créer un nouveau podcast, avec affichage des erreurs en dessous des champs si jamais les valeurs rentrées sont invalides.

The screenshot shows a web browser window with the address bar set to `http://pretpi.test`. The page title is "Podz v1.0". In the top navigation bar, there is a "Podcasts" button and a user profile "John Doe" with a dropdown arrow. The main content area is titled "Créer un podcast". Below this title, there is a "Titre" label followed by a single-line text input field. Below that is a "Description" label followed by a multi-line text area. At the bottom right of the form, there is a "Créer" button.

Choix de conception

Résumé des podcats

Sur la page Podcasts, il y a un résumé des descriptions des podcasts, qui se limitent à 150 caractères (+3 petits points), puisque la description est trop longue pour être affichée entièrement et l'utilisation de `text-overflow: ellipsis` en CSS sur plusieurs lignes n'est pas très simple. Raccourcir en PHP était donc l'autre solution. Un attribut `summary` de la classe `Podcast` permet de récupérer facilement ce résumé. Si la description est plus courte que la limite, la description est utilisée.

Visibilité des épisodes

Traduction

Pour que les messages d'erreurs soient en français. J'utilise le système d'internationalisation de

Laravel et j'ai défini le français comme langue par défaut et l'anglais comme langue de repli ("fallback language") au cas où quelque chose n'aurait pas été traduit en français. J'ai dupliqué le fichier `lang/fr/validation.php` à partir `lang/en/validation.php` et j'ai traduit les erreurs que j'utilisais.

Stratégie de test

Cette section concerne la manière dont est testé Podz durant le projet et à la fin. Samuel teste manuellement les fonctionnalités dans son navigateur (Firefox) et écrit aussi des tests automatisés avec PHPUnit (un framework PHP pour les tests). La plupart des fonctionnalités sont couvertes par ces tests automatisés et quand cela n'est pas le cas, Samuel regarde à la main si cela fonctionne. Les factories et le seeder écrits sont également très utile pour les tests.

La stratégie de développement est le BDD (Behavior Driven Development). Cela consiste à écrire des tests qui testent le comportement avant de coder, s'assurer que le test plante, puis développer jusqu'à que le test passe. Ensuite on peut refactoriser pour augmenter la qualité tout en s'assurant que cela fonctionne toujours grâce à nos tests.

Toute la suite de tests est lancée très fréquemment pour s'assurer qu'une nouvelle fonctionnalité n'a pas cassé une autre en chemin.

Où sont écrits les tests ?

Tous les tests se trouve dans le dossier `tests` à la racine du repository. Le dossier `Feature` contient les tests fonctionnels, `Unit` les tests unitaires et `Jetstream` les tests créé par Jetstream (ces derniers ont été déplacé de `Feature` afin de ne pas les exécuter constamment).

Couverture des tests

Comme les tests sont écrits et exécutés en PHP, les tests ne peuvent que tester le comportement backend. les interactions frontend ne peuvent pas être testées avec les outils actuels (il faudrait d'autres outils comme Laravel Dusk, Selenium, ...).

Pour la plupart des fonctionnalités, j'ai suivi cette ordre pour décider des tests à écrire et de leur contenu:

1. D'abord écrire un test pour vérifier que la page existe ou que le composant testé est bien chargé dans une des pages.
2. Ensuite tester le comportement idéal (toutes les données valides) pour s'assurer que les données gérées ont bien été modifiées.
3. Ensuite tester les validations des données.
4. Et finalement valider les permissions de visibilité ou d'accès (ex: être sûr qu'un visiteur ne peut pas modifier un épisode).

todo

Ce que les tests ne couvrent pas:

- Validation de la taille maximale d'upload d'un fichier
-

Voici la liste complète des tests, les noms devraient permettre d'avoir une idée de ce qui est testé et quels cas sont couverts.

Tests\Feature\YYY

- podcasts page exists

Résultats des tests

Cette capture montre le résultat des tests exécutés le YYY à YYY. Tous les tests passent. 

todos

Prérequis pour lancer les tests

Il est nécessaire d'avoir mis en place le projet et d'avoir l'extension PHP SQLite.

Avant l'exécution de chaque test, on retourne à l'état d'avant l'exécution du test (grâce au trait `RefreshDatabase`) et le seeder `DatabaseSeeder` s'exécute (`$seed` défini à `true`). Ces

2 configurations sont faites dans `tests/TestCase.php`, ce qui permet au final que tous les tests sont lancés sur une base de données propre et remplie.

Afin de ne pas impacter la base de données de développement, les tests sont lancés sur une base de données SQLite en mémoire. Voici les lignes en bas du fichier de configuration de PHPUnit `phpunit.xml`, qui redéfinit 2 variables d'environnement permettant d'avoir une base de données en RAM.

```
<env name="DB_DATABASE" value=":memory:"/>
<env name="DB_CONNECTION" value="sqlite"/>
```

Comment lancer les tests ?

Il y a différentes manières de lancer les tests dans un terminal:

- `./vendor/bin/phpunit`
- `php artisan test`

Les tests en dehors du dossier `tests/Unit` et `tests/Feature` ne seront pas lancés. Pour lancer les tests de Jetstream si besoin, il faut lancer `php artisan test tests/Jetstream`.

Vous pouvez passer des paramètres à `phpunit` (idem pour la commande `php artisan test`).

Exemples:

- pour exécuter seulement 1 test:
`php artisan test --filter podcasts_page_exists`
- pour exécuter une classe de tests donnée:
`php artisan test tests/Feature/PodcastsTest.php`
- pour exécuter les tests d'un dossier:
`php artisan test tests/Unit`

Je recommande de configurer un raccourci dans votre IDE pour lancer les tests. J'ai utilisé ce réglage de raccourci dans VSCode pour lancer `php artisan test tests/Feature` lors d'un `ctrl+t ctrl+t`

```
{
  "key": "ctrl+t ctrl+t",
  "command": "workbench.action.terminal.sendSequence",
  "args": {
    "text": "php artisan test tests/Feature\u000D"
  }
}
```

Planification

Dossier de conception

Upload d'un fichier audio d'épisode

J'ai décidé de fixer la taille maximum d'upload de fichiers à 150MB. Cette limite est fixée dans l'application, au niveau de la validation à la création d'un épisode. Les 2 valeurs dans la configuration de PHP (fichier `php.ini`) doivent être augmentées au dessus de 150MB: `upload_max_filesize` et `post_max_size`.

Composants réutilisables

Le composant Field

Un composant blade permettant d'abstraire les éléments communs de tous les champs de formulaire, avec quelques réglages possibles. L'affichage du label, le design basique, l'affichage des erreurs de validations.

Propriétés du composant

Nom	Type	Requis	Description
<code>name</code>	String	X	Le nom technique du champ, utilisé pour l'attribut <code>name</code> de l'input et par le <code>@error()</code> et par la fonction <code>old()</code>
<code>label</code>	String		Nom du label au dessus du champ
<code>type</code>	String		Type de l' <code><input></code> . Par défaut <code>text</code> . Si <code>textarea</code> est donné, une balise <code><textarea></code> est utilisée à la place.
<code>placeholder</code>	String		Un placeholder qui est ajouté directement sur le champ
<code>cssOnField</code>	String		Des classes CSS qui sont ajoutées directement sur le champ

Tous les autres attributs non reconnus sont transférés à la `div` racine du composant, ce qui permet d'ajouter du style ou d'autres attributs HTML. Tous les attributs commençant par `wire:model` sont ajoutés au champ pour permettre l'utilisation de ce composant avec Livewire.

Exemple d'utilisation:

```
<form action="{{ route('podcasts.index') }}" method="POST">
<x-field label="Title" name="title"></x-field>
<x-field label="Description" type="textarea" name="description"></x-field>
<x-field label="Date de naissance" type="date" name="user.date"></x-field>
[...]
```

Un autre exemple d'utilisation dans le cas d'un formulaire géré par Livewire:

```
<div>
  <x-field wire:keyup.enter="update" placeholder="Rentrez un titre court
et marquant." label="Title" name="podcast.title"
wire:model.lazy="podcast.title"></x-field>
  <x-field label="Description" type="textarea" name="podcast.description"
wire:model.lazy="podcast.description">
    </x-field>
  @csrf
  <button wire:click.prevent="update" class="btn mt-
1">Enregistrer</button>
</div>
```

Réalisation

Dossier de réalisation

Structure du repository: Certains dossiers de Laravel moins pertinents ont été remplacés par des `...`. Seulement les dossiers sont affichés et seulement que j'ai utilisé (travaillé dedans) sont définis.

```

podz                                Racine du repository
├─ app
│   ├── Actions
│   │   ├── Fortify
│   │   └─ Jetstream
│   ├── Console
│   ├── Exceptions
│   ├── Http
│   │   ├── Controllers    Les classes contrôleurs
│   │   ├── Livewire
│   │   └─ Middleware
│   ├── Models             Les classes modèles
│   ├── Providers
│   └─ View                Les classes des vues, pour les composants Blade
│       └─ Components
├─ ...
├─ config                  Les fichiers de configuration globaux
├─ database                Tout ce qui concerne la gestion de la base de
données
│   ├── factories          Les factories pour créer des données fictives
│   └─ migrations          Les migrations pour définir la structure des
tables
│   └─ seeders             Les seeders pour remplir la base de données avec
les factories
├─ docs                    Dossier pour stocker les éléments de
documentations (notamment MCD, MLD)
│   ├── imgs              Les images utilisant dans cette documentation
│   ├── models            Les exports des maquettes
│   └─ sources             Les fichiers source binaires des maquettes, MCD et
MLD
├─ lang                    Les fichiers de langues
│   ├── en
│   └─ fr                  Certaines traductions en français
├─ public
├─ resources               Toutes les ressources utiles à générer nos vues
│   ├── css               Style CSS global dans app.css
│   ├── js                Javascript global dans app.js
│   ├── markdown
│   └─ views
│       ├── api
│       ├── auth
│       └─ components

```

		layouts	
		livewire	Les vues pour Livewire.
		podcasts	Vues pour les podcasts
		profile	
		vendor	
		└─ jetstream	Les vues de Jetstream
		└─ ...	
	routes		Configuration des routes dans web.php
	storage		Espace de stockage dédié
	└─ app		Dossier ciblé par le disque "local"
	└─ public		Dossier publiquement accessible et ciblé par le
	disque "public"		
	└─ testing		Fichiers audios de tests pour le développement
	└─ clockwork		
	└─ ...		
	└─ logs		Emplacement de laravel.log
	tests		Tests automatisés
	└─ Feature		Tests fonctionnels
	└─ Jetstream		Tests créés par Jetstream
	└─ Unit		Tests unitaires
		.editorconfig	
		.env.example	Fichier .env d'exemple
		.gitattributes	
		.gitignore	
		.styleci.yml	
		artisan	
		composer.json	Liste des paquets Composer requis
		composer.lock	Liste des paquets Composer installées et leur
	version		
		package-lock.json	Liste des paquets NPM installées et leur version
		package.json	Liste des paquets NPM requis
		phpunit.xml	Fichier de configuration de PHPUnit
		README.md	
		tailwind.config.js	Configuration de Tailwind
		webpack.mix.js	Configuration du build JS et CSS avec Webpack pour
	Mix		

Description des tests effectués

Erreurs restantes

Liste des documents fournis

Conclusions

Annexes

Résumé du rapport du TPI / version succincte de la documentation

Sources – Bibliographie

- Icônes: les icônes ont été copié-collées (en SVG) depuis heroicons.com, elle sont publiées sous licence MIT.
- [Liste des Types de médias, par l'IANA](#). Cette ressource m'a été utile pour trouver les types MIME des fichiers audios .ogg, .opus, .mp3 et .m4a.

Journal de travail

Manuel d'Installation

Archives du projet