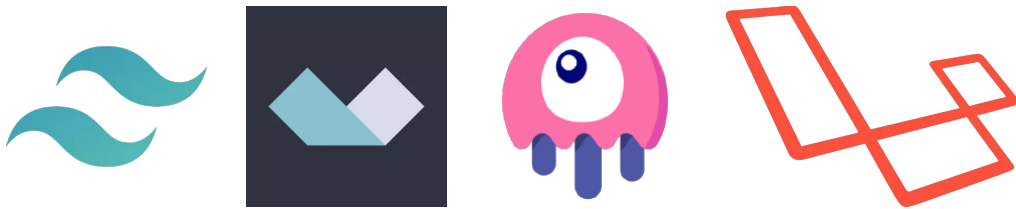


Documentation de Podz

Application web de publication de podcasts

Podz



Projet TPI - 2022

Samuel Roland, SI-MI4A

Table des matières

• Analyse préliminaire	3
◦ Introduction	3
◦ Glossaire	3
◦ Objectifs	4
◦ Planification initiale	5
• Analyse / Conception	7
◦ Technologies utilisées	7
◦ Outils d'aide	8
◦ Modèle Conceptuel de Données	9
◦ Modèle Logique de Données	11
◦ Maquettes	12
◦ Stratégie de test	19
▪ Où sont écrits les tests ?	19
▪ Les données de tests	19
▪ Comment lancer les tests ?	21
◦ Planification	22
◦ Dossier de conception	24
▪ Résumé des podcasts	24
▪ Visibilité des épisodes	24
▪ Traduction	24
▪ Vues de Jetstream	24
▪ Routes	24
▪ Upload d'un fichier audio pour la création d'un épisode	25
▪ Suppression d'un épisode	26
▪ Éléments réutilisables	26
• Réalisation	28
◦ Dossier de réalisation	28
◦ Construction de la documentation	29
◦ Résultats des tests effectués	30
▪ Couverture des tests	31
• Conclusions	33
◦ Erreurs restantes	33
◦ Objectifs atteints / non-atteints	33
◦ Difficultés particulières	34
◦ Points positifs / négatifs	35
◦ Bilan personnel	35
◦ Suites possibles pour le projet	36
◦ Remerciements	36
• Annexes	37
◦ Résumé du rapport du TPI	37
◦ Sources – Bibliographie	37
◦ Journal de travail	37
◦ Manuel d'installation	37
◦ Archives du projet	37

Analyse préliminaire

Introduction

Podz est une application web de publication de podcasts, développée pour le TPI de Samuel Roland en SI-MI4A. Les auteurs peuvent créer des podcasts, publier et gérer des épisodes, planifier la publication d'épisodes dans le futur et les cacher. L'application est basée sous Laravel 9 et ne part pas d'un projet existant.

Glossaire

- **AJAX**: Asynchronous JavaScript and XML: méthode pour faire des requêtes HTTP en arrière-plan dans un navigateur afin d'éviter un rechargement complet de la page
- **BDD**: Behaviour Driven Development
- **Blade**: moteur de vue de Laravel
- **CSS**: Cascading Style Sheets
- **Framework**: ensemble de bibliothèques et de conventions qui donnent un cadre pour développer une application
- **HTML**: Hypertext Markup Language
- **IDE**: Integrated Development Environment
- **JS**: JavaScript
- **MCD**: Modèle Conceptuel de Données
- **MLD**: Modèle Logique de Données
- **MVC**: Modèle Vue Contrôleur
- **PHP**: PHP Hypertext Preprocessor
- **POO**: Programmation orientée objet
- **RSS**: RDF Site Summary ou Really Simple Syndication: système de flux web pour diffuser du contenu (articles, podcasts, ...)
- **Sprint**: court cycle de travail (1-4 semaines) avec une revue du travail à la fin du cycle, généralement utilisé dans la méthode Scrum.
- **SQL**: Structured Query Language
- **Stack**: ensemble cohérent de technologies pour un but donné
- **Starter kit**: kit de démarrage permettant de sauter les premières étapes
- **TALL**: TailwindCSS - AlpineJS - Livewire - Laravel : stack de 4 frameworks web

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 besoin de les implémenter.

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.

En plus de cela, le travail sera évalué sur les 7 points spécifiques suivants:

1. Modélisation des données pertinentes (types, tailles, associations).
2. Respect du modèle MVC.
3. Ergonomie de l'interface utilisateur.
4. Gestion des erreurs de saisie des utilisateurs.
5. Respect des normes d'écriture de code.
6. Utilisation d'un SCM type git avec commits atomiques, petits et fréquents.
7. Lecture audio du podcast bien réalisée.

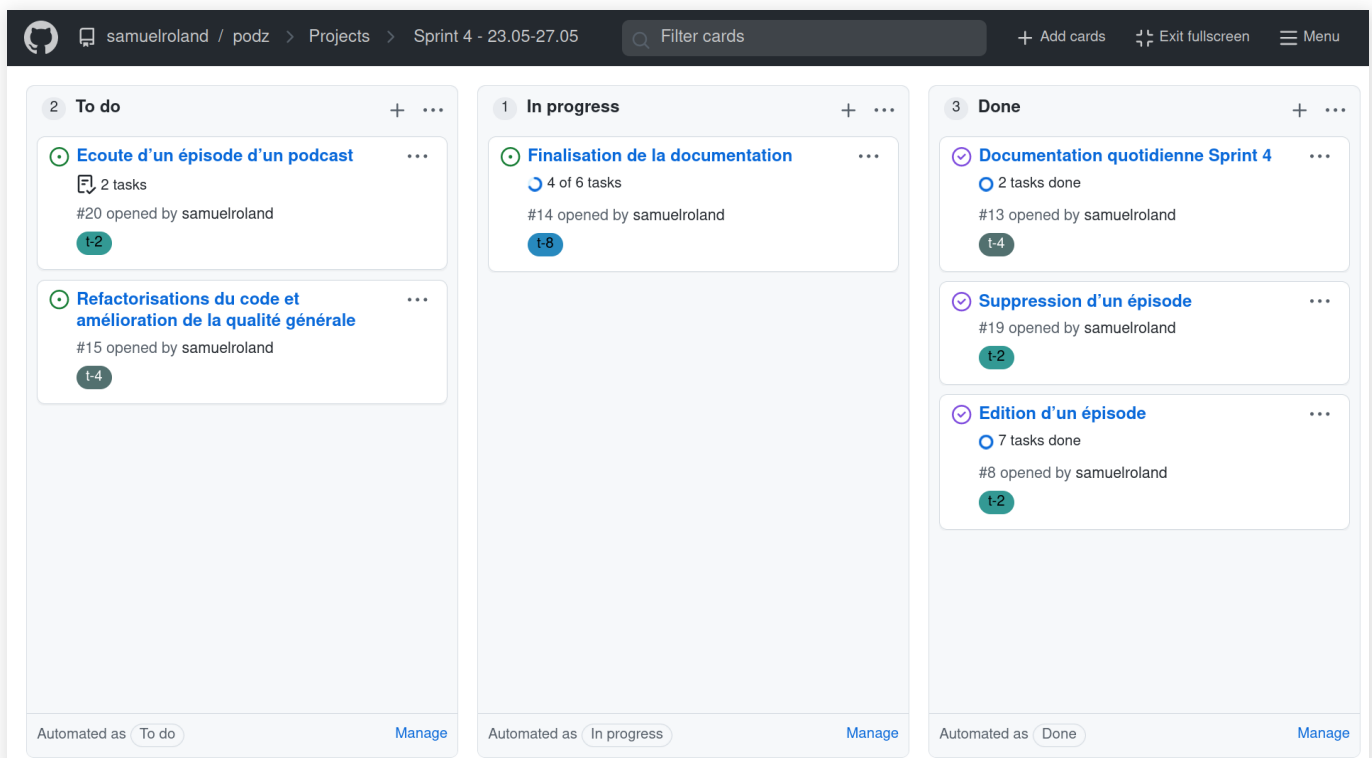
Planification initiale

Le projet n'a pas de méthode de gestion de projet formel, mais plutôt une adaptation de la méthode Scrum (je travaille en Sprint et mon chef de projet vient de faire des retours 1 fois par cycle). Je ne voulais pas partir avec des gros outils comme IceScrum, j'ai préféré partir sur GitHub Projects et gérer des Issues dans des Kanbans. Les étiquettes des Issues indiquent le temps estimé (ex : **t-3** = temps estimé de 3h). Le projet se découpe en 5 sprints, la majorité durent 1 semaine, entre le 02.05.2022 et le 31.05.2022. Comme demandé par l'expert 1, une tâche de documentation quotidienne (avec 4 cases à cocher pour les 4 jours de travail) existe pour chaque sprint (ce qui donne 1h par jour).

Dates des sprints:

- **Sprint 1:** du 02.05.2022 au 06.05.2022
- **Sprint 2:** du 09.05.2022 au 13.05.2022
- **Sprint 3:** du 16.05.2022 au 20.05.2022
- **Sprint 4:** du 23.05.2022 au 27.05.2022
- **Sprint 5:** du 30.05.2022 au 31.05.2022

Voici à quoi ressemble mes kanbans pour chaque Sprint:



La planification initiale rendue le premier jour dans un document séparé avait une mise en page peu pratique, j'ai donc repris les données et j'ai changé l'affichage pour plus de lisibilité. L'ordre des tâches est le même qu'il y avait dans les colonnes Todo sur GitHub au début du projet.

ID	Sprint	Titre	Estimé
#1	S1	Réunion avec expert 1 et chef de projet	2h
#3	S1	Mettre en place Laravel dans un nouveau repository	2h
#22	S1	Faire la planification	3h
#18	S1	Créer le MCD et documenter les spécificités	3h
#2	S1	Coder les migrations Laravel et générer le MLD	3h
#4	S1	Faire les maquettes et commencer la documentation avec le canva	3.5h
#10	S1	Documentation quotidienne Sprint 1	4h
#5	S2	Consultation de la liste des podcasts	3h
#6	S2	Création d'un nouveau podcast, édition d'un de ses podcasts existant	5h
#9	S2	Affichage de la liste des épisodes avec toutes les données liées (auteur)	4h
#7	S2	Ajout d'un nouvel épisode	5h
#11	S2	Documentation quotidienne Sprint 2	4h
#21	S3	Consultation du détail d'un podcast (visiteur)	3h
#8	S3	Edition d'un épisode	2h
#19	S3	Suppression d'un épisode	2h
#12	S3	Documentation quotidienne Sprint 3	4h
#20	S4	Ecoute d'un épisode d'un podcast	2h
#15	S4	Refactorisations du code et amélioration de la qualité générale	4h
#14	S4	Finalisation de la documentation	8h
#13	S4	Documentation quotidienne Sprint 4	4h
#17	S5	Relecture finale de tous les documents	3h
#16	S5	Préparation du rendu et impression documents finaux	4h

Analyse / Conception

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 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 simples aux problèmes récurrents (traductions, authentification, gestion des dates, ...). Tout ceci simplifie beaucoup le développement d'applications web en PHP une fois qu'on est 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 PHP pour avoir des parties réactives sur le frontend (normalement codées 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). 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 fournir des classes "composants". C'est très puissant pour construire rapidement des interfaces, en écrivant quasiment jamais de CSS pur. Pour faire du responsive c'est très pratique parce qu'il suffit d'utiliser un préfixe d'écran devant n'importe quelle classe pour utiliser des media queries. Par exemple, on peut utiliser `md:text-white` pour dire que le texte est blanc sur les écrans medium et au-dessus.

Divers:

- **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.

Outils d'aide

Pour m'aider dans mon développement, j'ai utilisé différents outils, ils ne sont pas requis pour travailler sur Podz, mais peuvent être très utiles:

- **Clockwork**: paquet Composer et extension web pour debugger les performances, les requêtes SQL, voir le temps d'exécution, ... Le paquet Composer est déjà installé.

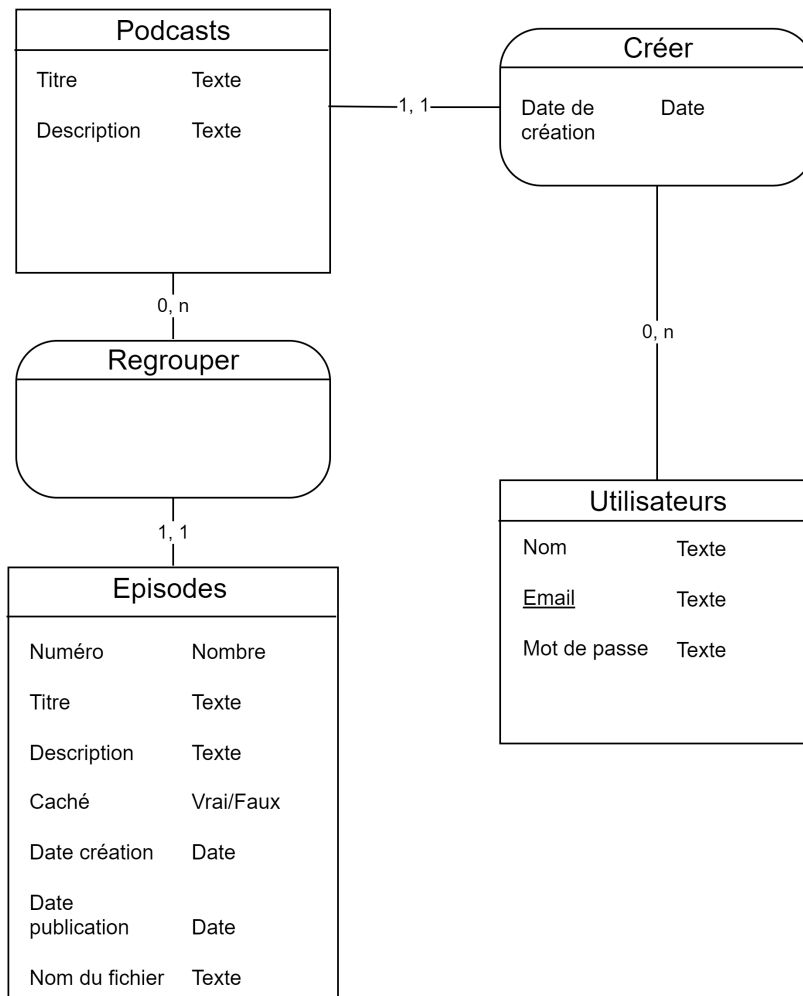
Path Controller	Status	Time Database	Model	Query	Duration
GET /livewire/livewire.js?id=c69dLivewireJavaScriptAssets@source	200	85 ms 0 ms	-	SELECT * FROM `sessions` WHERE `id` = 'cHVH8Sy2eM31mVvfDFEigzfVol9jFwoajJT60JFZ' LIMIT 1	1.41 ms
GET /PodcastController@index	200	97 ms 8 ms	Podcast	SELECT * FROM `podcasts`	0.34 ms
GET /PodcastController@index	200	98 ms 10 ms	Episode	SELECT * FROM `episodes` WHERE `episodes`.`podcast_id` in (1, 2, 3, 4, 5) ORDER BY `number` DESC	0.44 ms
GET /PodcastController@index	200	113 ms 11 ms	User	SELECT * FROM `users` WHERE `users`.`id` in (1, 2, 3)	0.34 ms
GET /PodcastController@index	200	97 ms 10 ms	Episode	SELECT * FROM `episodes` WHERE `episodes`.`podcast_id` = 1 and `episodes`.`podcast_id` IS not NULL and `released_at` < '2022-05-28 09:50:41' and `hidden` = 0 ORDER BY `number` DESC	0.38 ms
			Episode	SELECT * FROM `episodes` WHERE `episodes`.`podcast_id` = 2 and `episodes`.`podcast_id` IS not NULL and `released_at` < '2022-05-28 09:50:41' and `hidden` = 0 ORDER BY `number` DESC	0.38 ms
			Episode	SELECT * FROM `episodes` WHERE `episodes`.`podcast_id` = 3 and `episodes`.`podcast_id` IS not NULL and `released_at` < '2022-05-28 09:50:41' and `hidden` = 0 ORDER BY `number` DESC	0.36 ms
			Episode	SELECT * FROM `episodes` WHERE `episodes`.`podcast_id` = 4 and `episodes`.`podcast_id` IS not NULL and `released_at` < '2022-05-28 09:50:41' and `hidden` = 0 ORDER BY `number` DESC	0.31 ms

- **Laravel Valet**: fait tourner des serveurs web avec Nginx les rendant accessibles via des domaines en `.test`. Ce qui me permet de faire tourner mon serveur sous `podz.test` en HTTPS sans avoir besoin de me soucier de démarrer et d'arrêter ce serveur ni de gérer plusieurs ports quand plusieurs serveurs sont allumés. L'outil fonctionne pour MacOS, mais des forks pour **Windows** et **Linux** existent également. Attention à bien suivre la procédure d'installation pour ne pas être coupé d'internet à cause du DNS local mal configuré.

```
[sam@sx]~/code/podz% valet links
+-----+-----+-----+-----+
| Site   | SSL | URL                               | Path                               |
+-----+-----+-----+-----+
| apdebug | X   | https://apdebug.test             | /home/sam/code/apdebug          |
| podz   | X   | https://podz.test                | /home/sam/code/podz             |
+-----+-----+-----+-----+
```


Modèle Conceptuel de Données

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



Spécificités 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, 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 n'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

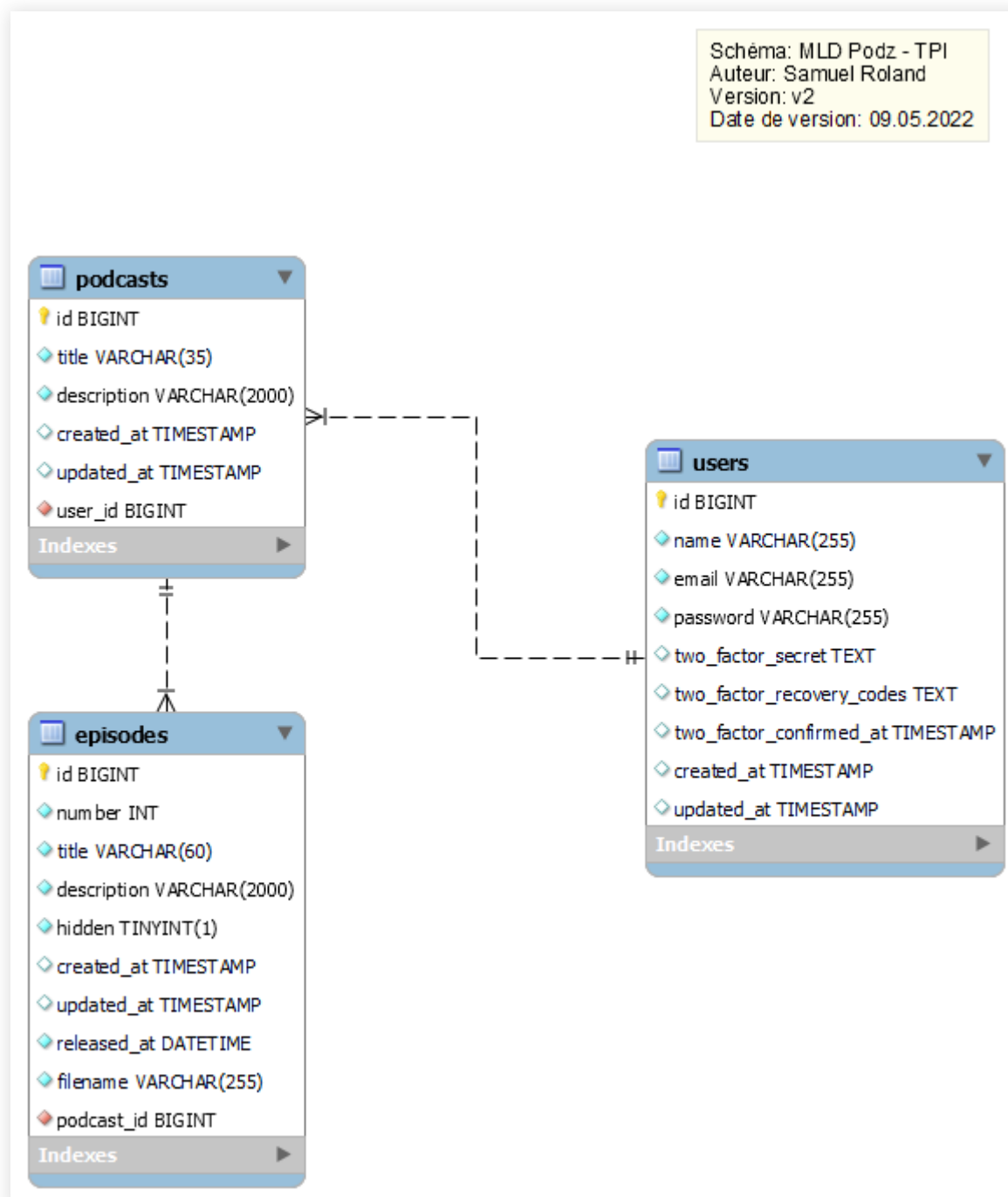
dans le futur une publication.

- 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.

Spécificités 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".

Modèle Logique de Données

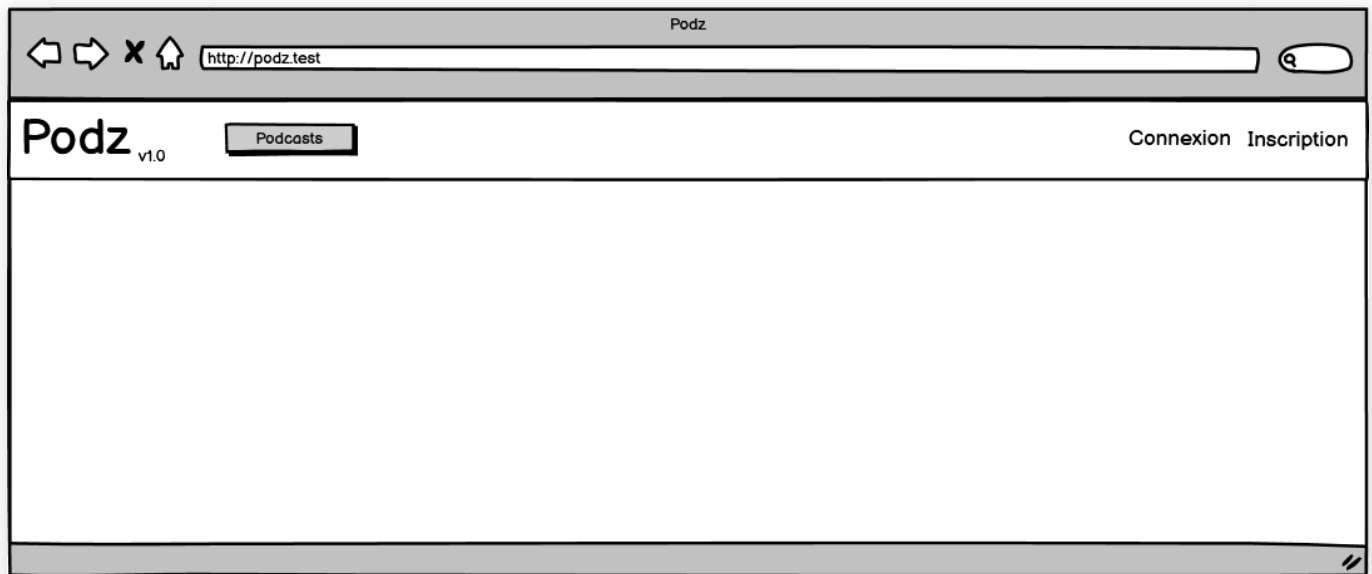


Ce MLD n'a pas été fait à la main mais a été rétro-ingéniéré depuis la base de données, après avoir codé les migrations. Certains champs (`two_factor_*`) sont créés par une migration générée par Jetstream, je n'en ai pas besoin mais je ne vais pas les retirer pour ne pas risquer de casser certaines vues 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, je ne les utilise pas directement.

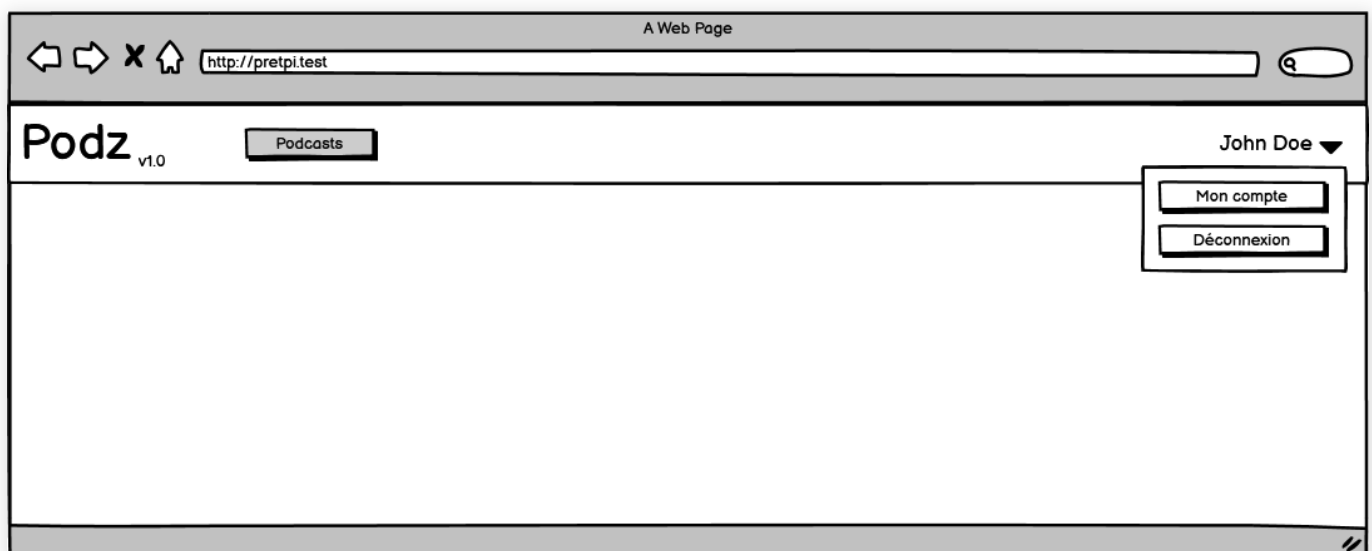
Les champs `created_at` et `updated_at` sont gérés automatiquement par Laravel (grâce au timestamps activés dans la migration), je n'utilise que le `created_at` en lecture seulement.

Maquettes

Le gabarit est déjà designé par Jetstream. Voici ce que voit un visiteur (déconnecté):



Et maintenant ce que voit un auteur (connecté):



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

- Page Connexion
- Page Inscription
- Page Liste des podcasts
- Page Page Détails d'un podcast
 - Vue visiteur
 - Vue Détails et édition pour auteur
- Page Création d'un podcast

Page Connexion

Podz

Podcasts

ConnexionInscription

Podz

Nom d'utilisateur ou email

Mot de passe

Login

Page Inscription

Podz

Podcasts

ConnexionInscription

Podz

Nom d'utilisateur

Email

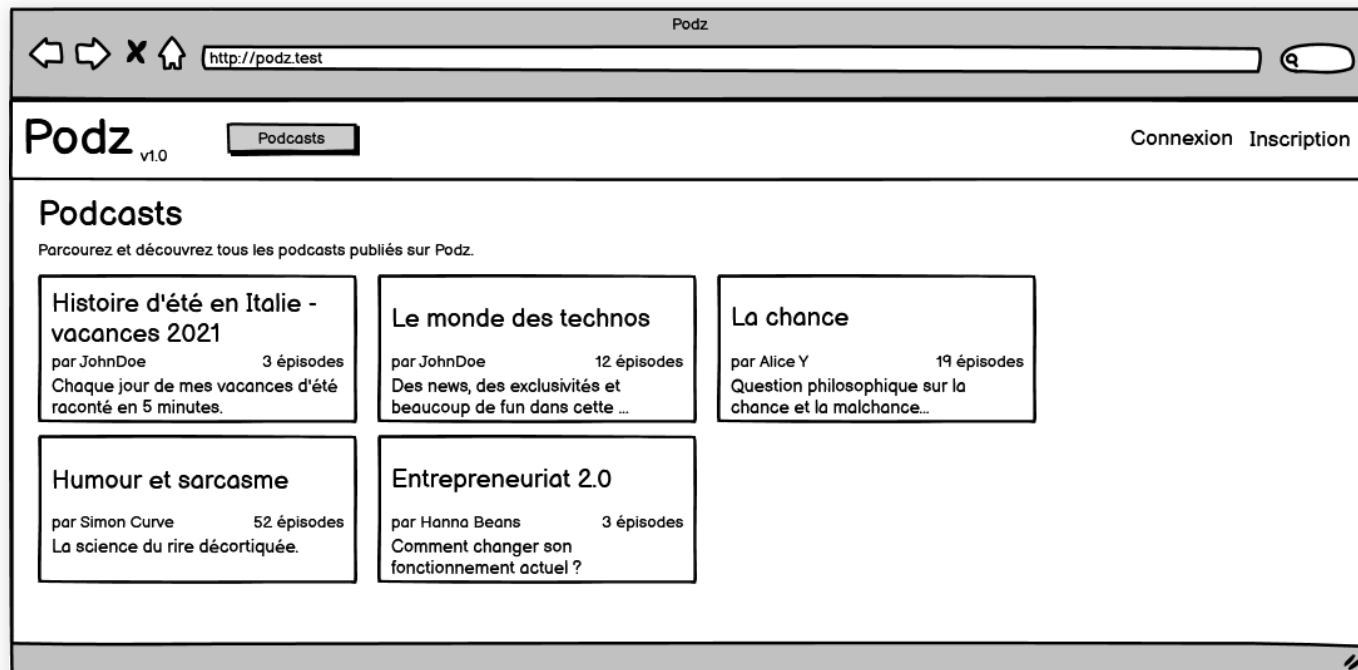
Mot de passe

Confirmer le mot

Créer un compte

Page Liste des podcasts

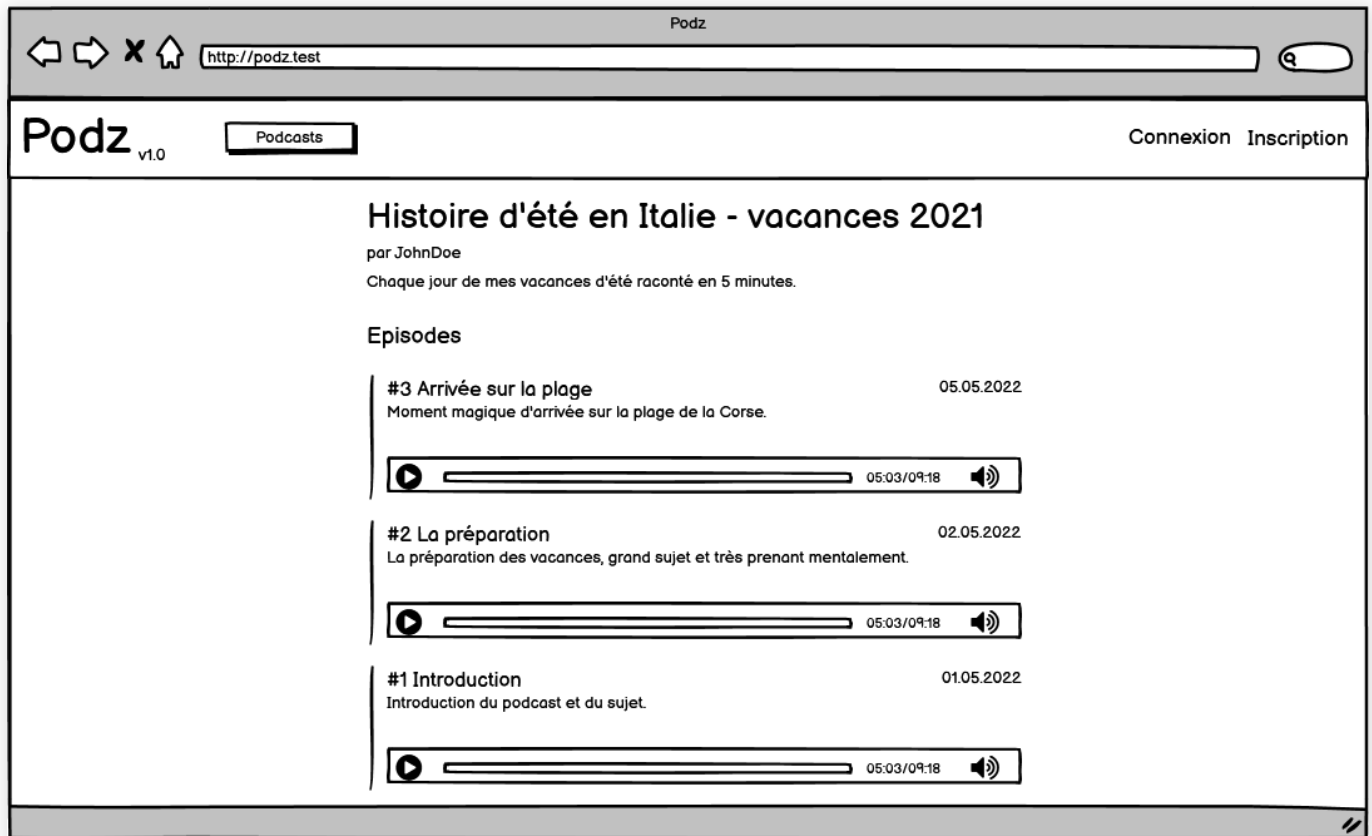
Cette page est visible publiquement et c'est la page par défaut de l'application, on y accède également via le bouton "Podcasts" en haut à gauche. On peut cliquer sur un podcast pour accéder à ses détails.



Page Détails d'un podcast

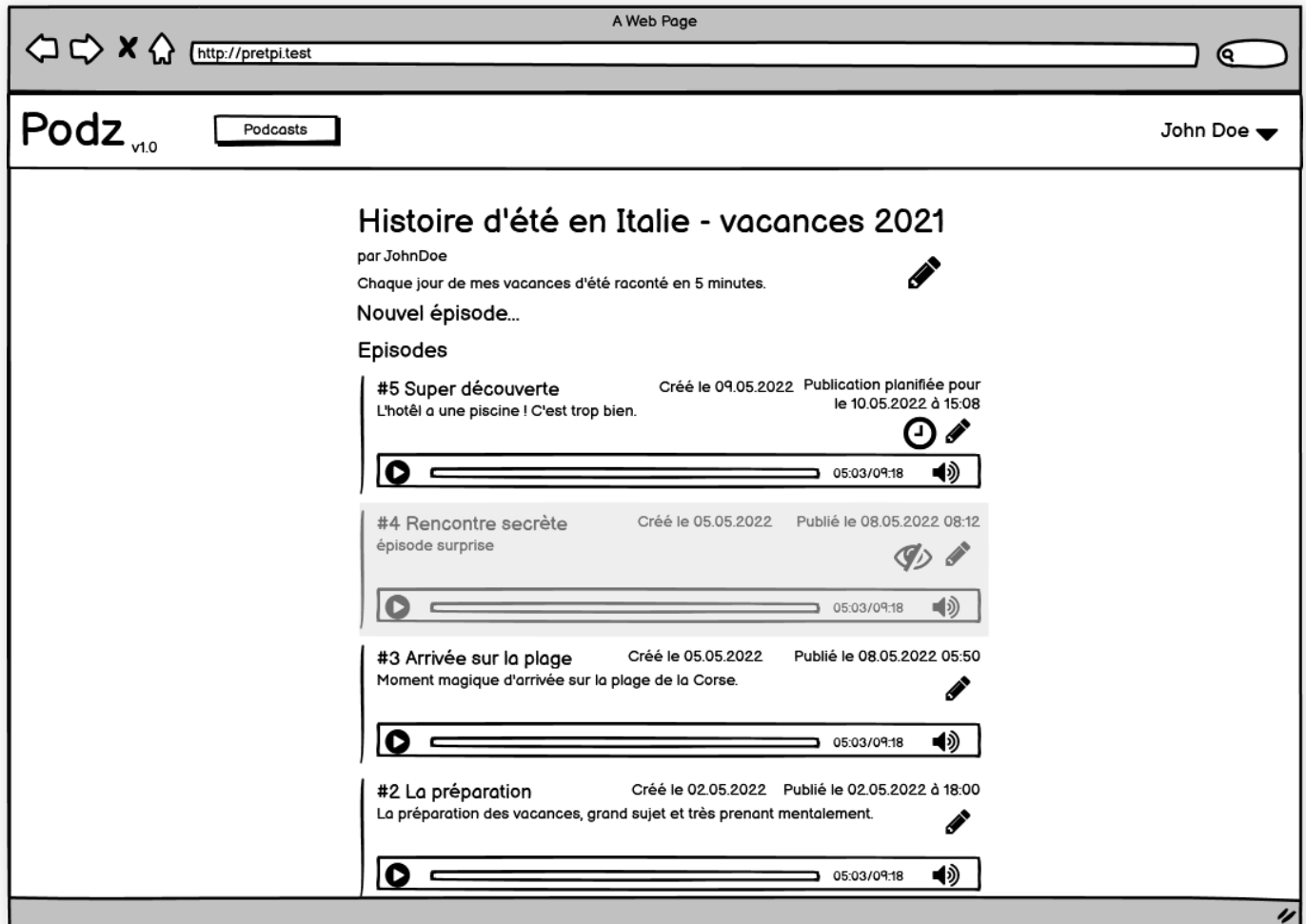
Vue visiteur

Les visiteurs ne voient que les épisodes qui sont visibles et ils ne voient que le numéro, le titre, la description, l'audio et la date (mais sans l'heure et la minute de publication).



Vue Détails et édition pour auteur

L'auteur voit toutes les informations de ses podcasts contrairement au visiteur. L'auteur a une vue visiteur sur les podcasts qui ne lui appartiennent pas. Nous sommes le 09.05.2022 dans cette maquette, l'épisode 4 est caché et le 5 est planifié pour le 10.05.2022 à 15:08. L'épisode 4 est caché parce que l'auteur a décidé après coup de le remettre en privé. Voici l'apparence de la page quand un auteur la charge.



Quand l'auteur clique sur les icônes d'édition, des formulaires s'affichent pour les éléments sélectionnés afin de permettre l'édition ou la suppression. Quand on clique sur [Nouvel épisode...](#) (voir maquette précédente), le formulaire de création apparaît juste en dessous. On peut éditer plusieurs éléments à la fois, il n'y aura pas de problèmes puisque la page ne se rafraîchit pas mais est découpée en plusieurs composants Livewire.

A Web Page

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 ☐ Caché

Fichier audio (mp3, m4a)

Aperçu

Episodes

#4 ☒ Caché

Aperçu

#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.

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.

A Web Page

http://pretpi.test

Podzv1.0

Podcasts

John Doe ▼

Créer un podcast

Titre

Description

Créer

Stratégie de test

Cette section concerne la manière dont est testé Podz durant le projet. Je teste manuellement les fonctionnalités dans mon navigateur (Firefox) et j'écris aussi des tests automatisés avec PHPUnit (un framework PHP de tests). La plupart des fonctionnalités sont couvertes par ces tests automatisés et quand cela n'est pas le cas, je regarde à la main si cela fonctionne.

La stratégie de développement est le BDD (Behaviour 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. J'ai fait quelques tests unitaires mais la majorité sont des tests fonctionnels. Toute la suite de tests est lancée très fréquemment (plusieurs fois par jour) pour s'assurer qu'une nouvelle fonctionnalité n'a pas cassé une autre en chemin.

Où sont écrits les tests ?

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

Les données de tests

Des factories et le seeder ont été codés pour ne pas avoir à rentrer des valeurs à la main. Dans mon seeder `DatabaseSeeder` je génère peu d'éléments (minimum de 2) pour les tests automatisés, afin d'accélérer l'exécution. Je génère plus d'éléments pour l'application locale afin d'avoir une situation plus réaliste dans le navigateur. Dans `EpisodeFactory`, j'ai fait en sorte que les épisodes soient toujours visibles et publiés dans le passé (afin d'éviter des tests qui plantent à cause de cette partie aléatoire non supportée). Quand les tests doivent avoir des épisodes cachés (pour tester les cas de visibilité), ils en créent eux-mêmes quelques-uns avant.

Etant le choix par défaut dans Laravel, j'ai utilisé le paquet Faker dans mes factories pour générer différents types de données. Le texte généré est en Lorem Ipsum. Ce qui est pratique comparé à l'écriture de données manuelles, c'est qu'on peut avoir des textes très longs permettant de valider dans nos interfaces que les valeurs extrêmes sont correctement affichées.

Exemple de données fictives générées par Faker:

title	description	hidden
Sit labore qui illum nobis	Similique quisquam corporis aut est omnis odio illo. Sit magni aut qui nisi adipisci dignis...	0
Et est magni dolor sed	Maxime pariatur ducimus vitae est ducimus. Rerum possimus deserunt quos aut itaque....	1
Et atque fugiat est quia quam	Tenetur mollitia autem omnis quibusdam quo et qui.	0
Qui sed voluptatum ut esse	Totam eveniet dolore impedit omnis. Hic id veritatis veritatis quia commodi amet officia....	0
Nulla odio omnis harum porro omnis	Quia exercitationem est repellat culpa. Ad et et eos architecto officiis fuga non. Fugiat m...	0
Amet velit dolorum quae	Aut sit porro facilis ratione dicta delectus aut pariatur. Soluta et asperiores provident fug...	0
Sunt est non voluptates	Quas dolores voluptatum quia ipsam dolor sint. Est enim est saepe quia fugit. Expedita e...	0
Ad alias totam aut aut saepe	Sint rerum unde dicta sed. Sunt libero deserunt voluptates. Aut sit ducimus ipsam dolori...	1
Dolores ipsum quasi ut nam	Ratione omnis aut aspernatur. Consequatur esse explicabo ab quis eum consequatur. A...	0

Avant chaque test, on retourne à l'état initiale grâce au trait `RefreshDatabase`. Puis le seeder `DatabaseSeeder` s'exécute grâce au `$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 est nécessaire d'avoir mis en place le projet et d'avoir l'extension PHP SQLite tout d'abord. Ensuite, il y a différentes manières de lancer les tests dans un terminal dans le dossier du projet:

- `php artisan test`
- `./vendor/bin/phpunit`
- `phpunit` (seulement si phpunit a été installé séparément/globalement)

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

Vous pouvez passer des paramètres à `phpunit` (fonctionne aussi avec la commande `php artisan test`).

Exemples:

1. pour exécuter seulement 1 test nommé `test_podcasts_page_exists` on peut filtrer:
`php artisan test --filter test_podcasts_page_exists`
2. pour exécuter une classe de tests donnée:
`php artisan test tests/Feature/PodcastsTest.php`
3. pour exécuter les tests d'un dossier:
`php artisan test tests/Unit`

Je recommande de configurer un raccourci clavier dans votre IDE pour lancer les tests. J'ai utilisé ce réglage de raccourci dans VSCode pour lancer tous les tests lors d'un `ctrl+t ctrl+t`

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

Planification

La liste des tâches est la même qu'au départ, les estimations n'ont pas été modifiées. Afin de comparer ce qui avait été prévu et ce qui s'est réellement passé finalement, j'ai rajouté quelques colonnes. Tout le tableau est ordonné par la date d'achèvement des tâches, ce qui explique que ce n'est pas exactement le même ordre que la planification initiale. **S-d** signifie **Sprint de départ** et **S-f** signifie **Sprint final** (est différent pour les tâches achevées en retard ou en avance). Le Delta est le résultat de Temps estimé - Temps passé. Ce calcul n'a pas été fait pour les tâches des "Documentation quotidienne" car ce n'est pas un temps estimé mais planifié.

ID	S-d	S-f	Titre	Estimé	Passé	Delta	Fin
#1	S1	S1	Réunion avec expert 1 et chef de projet	2h	3.5h	-1.5h	03.05.2022
#3	S1	S1	Mettre en place Laravel dans un nouveau repository	2h	1h	1h	03.05.2022
#22	S1	S1	Faire la planification	3h	5h	-2h	03.05.2022
#2	S1	S1	Coder les migrations Laravel et générer le MLD	3h	1h	2h	03.05.2022
#18	S1	S1	Créer le MCD et documenter les spécificités	3h	1.5h	1.5h	03.05.2022
#4	S1	S1	Faire les maquettes et commencer la documentation avec le canva	3.5h	4h	-0.5h	06.05.2022
#5	S2	S1	Consultation de la liste des podcasts	3h	3h	0h	06.05.2022
#10	S1	S1	Documentation quotidienne Sprint 1	4h	3.5h		06.05.2022
#9	S2	S2	Affichage de la liste des épisodes avec toutes les données liées (auteur)	4h	10h	-6h	12.05.2022
#6	S2	S2	Création d'un nouveau podcast, édition d'un de ses podcasts existant	5h	5h	0h	13.05.2022
#11	S2	S3	Documentation quotidienne Sprint 2	4h	1h		19.05.2022
#21	S3	S3	Consultation du détail d'un podcast (visiteur)	3h	2h	1h	22.05.2022
#7	S2	S4	Ajout d'un nouvel épisode	5h	16h	-11h	23.05.2022
#12	S3	S4	Documentation quotidienne Sprint 3	4h	3.5h		23.05.2022
#8	S3	S4	Edition d'un épisode	2h	8h	-6h	23.05.2022
#19	S3	S4	Suppression d'un épisode	2h	2h	0h	23.05.2022
#13	S4	S4	Documentation quotidienne Sprint 4	4h	5h		28.05.2022
#20	S4	S5	Ecoute d'un épisode d'un podcast	2h	1h	1h	30.05.2022
#14	S4	S5	Finalisation de la documentation	8h	13h	-5h	31.05.2022

ID	S-d	S-f	Titre	Estimé	Passé	Delta	Fin
#17	S5	S5	Relecture finale de tous les documents	3h	1h	2h	31.05.2022
#15	S4	S5	Refactorisations du code et amélioration de la qualité générale	4h	0.5h	3.5h	31.05.2022
#16	S5	S5	Préparation du rendu et impression documents finaux	4h	1h	3h	31.05.2022
			<i>Tâches diverses</i>		6h		
			Totaux	77.5h	97.5h		

Tâches diverses contient toutes les activités qui ne sont pas reliés à des Issues sur GitHub, ce comptage se base sur le journal de travail (voir les entrées qui n'ont pas de tâche assignée). Ceci inclut les visites de M. Hurni et des experts et la résolution de petits bugs.

Analyse des différences

Quand on compare le temps estimé et passé on voit que j'ai sur-estimé certaines tâches simples, et que j'ai beaucoup sous-estimé les tâches plus complexes et longues. À partir du sprint 3, presque toutes les tâches ont été terminée un ou deux sprints plus tard. Les 2 tâches les plus sous-estimées sont "Ajout d'un nouvel épisode" et "Finalisation de la documentation". Je n'avais pas imaginé avoir autant de peine pour la création d'épisode, et qu'il y avait autant de choses à expliquer dans la documentation.

Je m'en suis rendu compte tard, mais mon sprint 4 était prévu sur toute la semaine alors que le jeudi et vendredi étaient fériés. Si on regarde mon journal de travail, on voit que je n'ai pas réussi à faire de la documentation tous les jours. Dans ce tableau, il y a aussi des petits bouts de documentations écrits pour les fonctionnalités en tant que tels dont le temps est compté avec celles-ci. J'étais très concentré sur le code en sprint 2 et j'ai fait moins de documentation que le reste des sprints. A la fin j'avais du retard sur les finitions du code et surtout sur ma documentation, j'ai donc décidé de faire quelques heures à la maison.

Dossier de conception

Résumé des podcasts

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

Pour qu'un épisode soit visible publiquement il faut que sa date de publication soit dans le passé et que son état Caché soit Faux. Si cette condition n'est pas vraie, l'épisode n'est visible que par l'auteur. Si on regarde en détail le code et les routes, on s'aperçoit que les fichiers étant sur le disque public, il n'y a pas d'autorisations appliquée au chargement des fichiers audios. Ainsi si on mémorise le nom du fichier audio, et que l'épisode devient ensuite invisible, on pourra toujours accéder publiquement via le lien d'accès direct (ex:

<https://podz.test/storage/episodes/UyJ7nE5TewwbjXRAhrmWX6Ht45.ogg>). Cette sécurité n'était pas demandée donc je ne l'ai pas implémentée mais cela pourrait être une idée d'amélioration. Pour corriger ceci, il faudrait bouger les épisodes dans le disque `local` qui n'est pas publiquement accessible, et "streamer" les fichiers audio via une route dédiée de notre application, de sorte à pouvoir appliquer un contrôle des droits d'accès et bloquer l'accès du fichier audio sur un épisode caché si ce n'est pas l'auteur.

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 quelques messages d'erreurs que j'utilisais.

Vues de Jetstream

Le `navigation-menu.blade.php` a été modifié afin d'avoir les bons boutons. Le logo de Jetstream était modifiable dans 3-4 fichiers différents, j'ai préféré regrouper le tout dans `logo.blade.php` afin de centraliser. Le logo utilise la couleur `green` définie dans `tailwind.config.js`. Le gabarit `layouts.guest` a été supprimé au profit d'un seul gabarit `layouts.app`, le menu de navigation s'adapte pour si on est connecté ou non.

Routes

J'ai suivi les conventions des noms et URLs des routes comme pour les contrôleurs ressources (je n'en ai pas utilisé dans ce projet).

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Tiré de la [documentation de Laravel](#)

Upload d'un fichier audio pour la création d'un 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 et dans la taille maximum pour l'upload de fichiers temporaires de Livewire. Ces 2 paramètres dans la configuration de PHP (fichier `php.ini`) doivent être augmentées au-dessus de 150MB: `upload_max_filesize` et `post_max_size`.

Les fichiers audios sont stockés dans `storage/app/public/episodes` c'est-à-dire dans le dossier `episodes` du dossier `public` avec un nom aléatoire unique.

Suppression d'un épisode

J'ai surchargé la méthode `delete` dans `Episode.php` afin d'ajouter la suppression du fichier en même temps que la suppression de l'enregistrement. J'ai mis le tout dans une transaction pour éviter d'avoir l'incohérence du fichier qui existe sur le disque mais il n'y a plus d'épisode lié dans la base de donnée. Cette transaction n'empêche pas d'avoir l'incohérence inverse, puisque la suppression sur le disque n'est pas une requête SQL (et ne peut pas être rollback).

```
public function delete()
{
    DB::transaction(function () {
        //Delete file on disk first
        Storage::disk('public')->delete($this->path);

        //Then delete the record in db
        parent::delete();
    });
}
```

Éléments réutilisables

Le composant Field

Un composant Blade permettant d'abstraire les éléments communs à tous les champs de formulaire: l'affichage du label, le design basique et l'affichage des erreurs de validations.

Propriétés du composant

Nom	Type	Requis	Description
<code>name</code>	String	Oui	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	Non	Nom du label au-dessus du champ.
<code>type</code>	String	Non	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	Non	Un placeholder qui est ajouté directement sur le champ.
<code>cssOnField</code>	String	Non	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 pour tout le composant. 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.store') }}" 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>
```

Classes CSS et couleurs

J'ai défini 3 nouvelles couleurs Tailwind, qu'on peut utiliser partout où les couleurs fonctionnent avec TailwindCSS (`border-green` , `text-lightblue` , `bg-blue` , ...)

```
//Extrait de tailwind.config.js
colors: {
  'green': '#0d9488',
  'blue': '#0d1594',
  'lightblue': '#0d159414',
}
```

Il y a aussi des classes CSS qui peuvent être utilisées pour avoir un design commun à travers l'interface:

- `text-info` : pour les messages d'informations
- `btn` : pour les boutons

Réalisation

Podz est maintenant en version 1 (v1), cette version est affichée à droite du logo. Il n'y a pas d'autres numéros avant.

Dossier de réalisation

Structure du repository

Certains dossiers de Laravel moins pertinents ont été remplacés par des `...`. Seulement les dossiers et les fichiers à la racine sont affichés. Uniquement ceux que j'ai utilisé sont définis.

podz	Racine du repository
└ app	
└ Actions	
└ Fortify	
└ Jetstream	
└ Console	
└ Exceptions	
└ Http	
└ Controllers	Les classes contrôleurs
└ Livewire	Les classes des composants 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 (MCD, MLD)
└ imgs	Les images utilisées 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 écrit dans app.css
└ js	Javascript global écrit dans app.js
└ markdown	
└ views	
└ api	
└ auth	
└ components	
└ layouts	Contient le gabarit app.blade.php
└ livewire	Les vues pour Livewire

			└─ podcasts	Les 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	Le CLI 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

Construction de la documentation

La documentation étant écrite en Markdown, j'ai dû régler plusieurs problèmes pour avoir le même résultat visuel que si j'avais travaillé dans Word.

Pour l'exporter en PDF et avoir cette apparence, j'ai utilisé VSCode et une extension nommée [Markdown PDF](#) (id: [yzane.markdown-pdf](#)), de lancer la palette de commandes (Ctrl + Maj + P), puis de choisir l'action [Markdown PDF: Export \(pdf\)](#). Le résultat sera le fichier [podz-docs.pdf](#) à côté de ce fichier. Même fonctionnement pour le journal de travail et le README s'il y a besoin de les exporter. J'ai dû écrire du CSS [docs/markdown-build/pdf-export.css](#) pour améliorer le design de l'export qui n'était pas très joli. Toutes les configurations pour l'extension sont faites dans le fichier [.vscode/settings.json](#) (en-tête et pied de page, choix du thème du surlignage avec HighlightJS, taille des marges et feuilles de styles).

Comme j'utilise Git, je n'ai pas besoin de garder d'anciennes versions avec un numéro de version choisi, puisque tout l'historique est consultable. Pour retrouver la documentation à une date donnée, il suffit d'aller sur GitHub sous les commits, de prendre le dernier commit avant cette date, de cliquer sur Browse files puis d'aller chercher le documents dans le dossier [docs](#).

Résultats des tests effectués

Cette capture montre le résultat des tests exécutés le 30.05.2022. Tous les tests passent.

```
[sam@sx]~/code/podz% ./vendor/bin/phpunit
PHPUnit 9.5.20 #StandWithUkraine

..... 44 / 44 (100%)

Time: 00:01.310, Memory: 50.50 MB

OK (44 tests, 174 assertions)
```

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

1. Tests\Unit\EpisodeTest

1. path is well built

2. Tests\Unit\PodcastTest

1. podcasts summary is correctly extracted
2. podcasts summary doesnt extract when description length is already good
3. get next number really gives next number

3. Tests\Feature\EpisodeCreationTest

1. podcast details page uses episode creation component
2. podcast details page doesnt use episode creation if not author
3. episode creation works
4. data are correctly validated
5. audio file type is validated
6. default value of the episode are set
7. publishing fails silently if forbidden
8. publishing 2 episodes with same title in a podcast is not possible

4. Tests\Feature\EpisodeDeletionTest

1. episode deletion works
2. episode deletion is only authorized to the author

5. Tests\Feature\EpisodeUpdateTest

1. podcast details page uses episode update component
2. podcast details page doesnt use episode update if not author
3. episode update works
4. data are correctly validated
5. datetime value is set after mount
6. update fails silently if forbidden
7. updating title to another episode title in the same podcast fails

6. Tests\Feature\PodcastCreationTest

1. create a podcast page exists
2. create a podcast page is guarded
3. store route is guarded
4. podcast creation works
5. podcast is not created on invalid request
6. new podcast button is present
7. new podcast button doesnt exist as visitor

7. Tests\Feature\PodcastDetailsTest

1. podcasts details page exists
2. podcast info component is included in the page
3. all information are displayed for the author
4. a message is displayed when no episode is published
5. prefix text of future release date is displayed correctly for author
6. release date displays only date for the public
7. future episodes are not publicly visible
8. past hidden episodes are nt visible for the public
9. only required info are displayed publicly

8. Tests\Feature\PodcastUpdateTest

1. podcast details page contains update component
2. podcast details page doesnt contain update component as visitor and as non author
3. details can be updated
4. details must be valid

9. Tests\Feature\PodcastsTest

1. podcasts page exists
2. the page has title and description
3. all podcasts are displayed with their data

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.

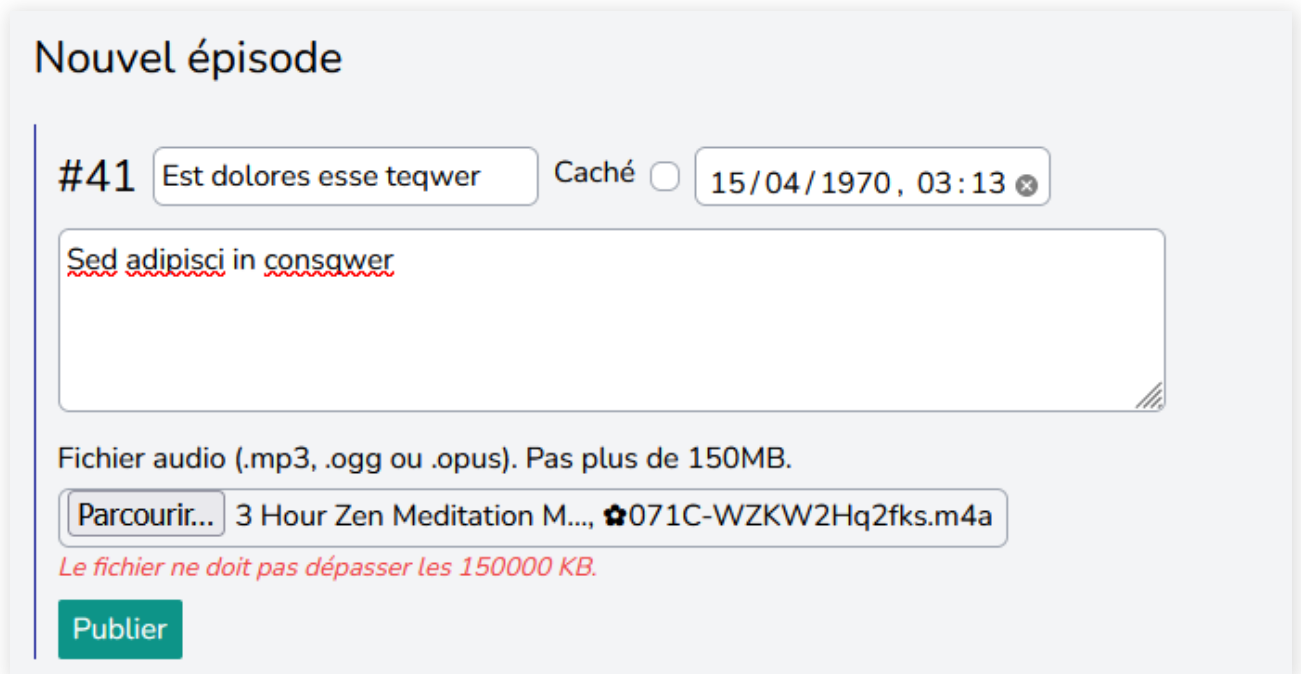
Pour la plupart des fonctionnalités, j'ai suivi cette ordre pour décider des tests à écrire et de leur contenu:

1. D'abord écrire un test pour vérifier que la page existe ou que le composant Livewire testé est bien chargé dans une des pages.
2. Ensuite tester le comportement idéal (avec toutes les données valides).
3. Puis 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 ou ne peut pas voir d'épisode s'il est invisible).

Ce que les tests ne couvrent pas:

- La validation de la taille maximale d'upload d'un fichier pour la création d'épisode

Les tests manuels ont permis de vérifier que cela fonctionnait. Un test manuel avec un fichier mp3 de 170Mo a été fait plusieurs fois afin de vérifier la limite de 150Mo. En voici la démonstration:



The screenshot shows a web form titled "Nouvel épisode". It contains several input fields: a number field with "#41", a text field with "Est dolores esse teqwer", a checkbox labeled "Caché" which is unchecked, and a date/time field with "15/04/1970, 03:13" and a close icon. Below these is a large text area containing the text "Sed adipisci in consqwer" with red wavy lines underneath. Under the text area, it says "Fichier audio (.mp3, .ogg ou .opus). Pas plus de 150MB." followed by a file selection button "Parcourir..." and a file name "3 Hour Zen Meditation M..., 071C-WZKW2Hq2fks.m4a". Below the file name, a red error message states "Le fichier ne doit pas dépasser les 150000 KB." At the bottom left of the form is a green button labeled "Publier".

- Les méthodes `episodes()` et `publicEpisodes()` de `Podcast` en test unitaire
- La présence des icônes, selon la personne connectée (puisque'ils sont en SVG ils n'ont pas de nom et c'est difficile de les identifier)

Conclusions

Erreurs restantes

- Au lancement des tests, les fichiers audios créés ne devraient pas aller dans le dossier `storage/app/public/episodes` mais un faux dossier de stockage (avec `Storage::fake('public');`), mais cela ne marche pas vraiment et je ne sais pas pourquoi.

Objectifs atteints / non-atteints

Tous les objectifs fixés au départ ont été atteints.

Objectif	Atteint ?
En tant que visiteur (personne non authentifiée) :	
• Consultation de la liste des podcasts.	Oui
• Consultation du détail d'un podcast : épisodes	Oui
• Ecoute d'un épisode d'un podcast.	Oui
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.	Oui
Sur l'un de ses podcasts :	
• Affichage de la liste des épisodes avec toutes les données liées.	Oui
• Ajout d'un nouvel épisode.	Oui
• Edition d'un épisode.	Oui
• Suppression d'un épisode.	Oui

Difficultés particulières

- L'upload de fichiers et les tests associés ont été assez difficiles, comme expliqué dans mon journal de travail. Pour comprendre pourquoi les tests ne passaient pas alors que mon code était correct quand on faisait `UploadedFile::fake()->create('audio.m4a', 100, 'audio/mp4')` par ex., j'ai regardé dans le code de la classe `FileFactory` (dans `vendor\laravel\framework\src\Illuminate\Http\Testing\FileFactory.php`) dans mon IDE (en faisant Ctrl+click sur la méthode `create()`) et j'ai trouvé ceci:

```
/**
 * Create a new fake file.
 *
 * @param string $name
 * @param string|int $kilobytes
 * @param string|null $mimeType
 * @return \Illuminate\Http\Testing\File
 */
public function create($name, $kilobytes = 0, $mimeType = null)
{
    if (is_string($kilobytes)) {
        return $this->createWithContent($name, $kilobytes);
    }

    return tap(new File($name, tmpfile()), function ($file) use ($kilobytes, $mimeType) {
        $file->sizeToReport = $kilobytes * 1024;
        $file->mimeTypeToReport = $mimeType;
    });
}
```

On voit que le fichier contient le résultat `tmpfile()` (fonction PHP qui crée des fichiers temporaires), en inspectant avec un éditeur hexadécimal on y trouve une vingtaine d'octets toujours les mêmes, le contenu le correspond donc ni à la bonne taille ni au bon type MIME demandé. Pour que cela fonctionne quand même avec Laravel, la taille et le type MIME - que la classe retourne quand on lui demande - sont définis dans des attributs de la classe. Le problème dans mon application, c'est probablement parce que j'utilise Livewire qui stocke les fichiers dans un dossier temporaire puis les déplacent dans le bon dossier à la sauvegarde. Ce n'est qu'une hypothèse que je n'ai pas pu le vérifier (cela aurait demandé des recherches plus longues) mais j'imagine que l'objet `UploadedFile` final est rechargé ou recréé avec le fichier sur le disque, le type MIME et la taille étant fictifs sont donc perdus durant le processus.

Pour résoudre ce problème, j'ai finalement créé différents vrais fichiers de différents formats dans `storage/app/testing` avec FFmpeg, et créé des fichiers bidons (`test.pdf`), que j'utilise comme fichier à l'upload.

- L'export PDF de mes documentations et la construction des planifications ont été complexes, avec toutes les choses à inclure à inclure et moyens de détourner les contraintes. Pour la planification finale, il y avait beaucoup de valeurs qui devaient être recopiées de GitHub. Au

lieu de tout faire à la main j'ai préféré scripter sa génération. J'ai créé un fichier `planifdata.json` avec les infos des Issues tirées de l'API de GitHub dans lequel j'ai ajouté le temps passé sur chaque tâche (en calculant les sommes des temps indiqués dans mon journal de travail). Mon script fonctionne très bien et est super pratique. J'ai du faire du design de mon document en CSS et parfois écraser le style par défaut de l'extension, cela m'a pris un certain temps.

Points positifs / négatifs

Les tests automatisés sont un point positif du projet, car sont robustes et m'ont beaucoup aidé durant le développement. Au niveau de la planification j'aurai pu mieux gérer mon temps en classe. Parfois je suis resté bloqué sur l'écriture de tests que j'aurai pu outrepasser et d'autres fois j'étais déconcentré et/ou j'aidais des collègues sur Laravel. Mieux avancer et être un peu plus concentré aurait peut-être permis de ne pas avoir trop de retard à la fin. J'aurai aussi pu faire les calculs des totaux de temps de travail pour me rendre compte de mon avance ou retard.

Au niveau de la documentation, faire de la documentation plus régulièrement aurait permis de varier un peu le travail final. Je pense avoir fait une documentation assez qualitative et soignée. J'ai mis plus de détails et de soin dans cette documentation que d'habitude, c'était important pour moi de rendre des documents soignés. Un autre point positif est d'avoir réussi à finir toutes les fonctionnalités demandées.

Bilan personnel

J'ai eu beaucoup de plaisir à développer Podz, surtout avec l'écriture des tests. Contrairement à mon Pré-TPI où je n'avais pas pu terminer le développement et la documentation, je suis plutôt content d'avoir réussi à finir toutes les fonctionnalités demandées dans les temps et d'avoir pu faire correctement la documentation. Je me sens encore plus à l'aise qu'avant pour écrire des tests, même pour des cas plus complexes pour gérer des fichiers et des erreurs. J'ai compris les stratégies de base pour savoir ce qu'on peut tester ou pas, quand je dois en écrire un nouveau je sais donc rapidement quels sont les éléments à inclure. Au passage, j'ai appris que tous les navigateurs ne supportent pas tous les fichiers audio (surtout s'ils sont propriétaires), Firefox par ex. a quelques difficultés avec les fichiers `.m4a`. Comme durant mon Pré-TPI, j'ai eu de la peine avec l'upload de fichiers parce que je n'arrivais pas à écrire des tests corrects. Donc j'ai beaucoup testé à la main et cela devenait vite chronophage. Grâce à l'aide M. Hurni mon chef de projet, j'ai pu changer de stratégie pour ces tests, je serai comment m'y prendre à l'avenir.

Suites possibles pour le projet

De nombreuses fonctionnalités pourraient être implémentées si le projet est réutilisé par quelqu'un d'autre. Voici une petite liste d'idées:

1. Ajouter un flux RSS ce qui permet d'écouter le podcast depuis un lecteur de podcasts (comme Apple Podcasts par exemple). Ce flux pourrait être utilisé pour republier le contenu sur d'autres plateformes (Spotify, Apple Podcasts, Soundcloud, ...).
2. Comme expliqué pour l'upload de fichiers, les fichiers audio pourraient être sécurisés derrière une route Laravel et non un accès direct, en passant par le disque `local`.
3. L'ajout d'images comme pochette de podcasts
4. Ajout de commentaires pour chaque épisode

Et beaucoup d'autres possibilités encore...

Remerciements

J'aimerais remercier M. Hurni pour les retours et les conseils techniques qu'il m'a apportés au Pré-TPI et au TPI. Il a pu répondre à mes nombreuses questions et j'ai senti une vraie progression avec Laravel en général et l'écriture de tests. J'espère avoir pu utiliser au mieux ces feedbacks et continuer de m'améliorer continuellement sur Laravel et les autres frameworks à l'avenir, pour produire du code de qualité et maîtriser de plus en plus ces technologies.

Je remercie aussi Gatien Jayme pour sa relecture de ma documentation.

Annexes

Résumé du rapport du TPI

Le résumé est disponible en document séparé (voir archives) ou directement sur GitHub [en Markdown](#).

Sources – Bibliographie

Pour résoudre mes différents problèmes j'ai surtout utilisé StackOverflow et les documentations officielles des 4 frameworks que j'utilise:

- [Documentation de Laravel](#)
- [Documentation de Livewire](#)
- [Documentation de AlpineJS](#)
- [Documentation de TailwindCSS](#)

J'ai aussi utilisé le site [Mozilla Developer Network](#) comme référence pour le HTML et le CSS.

- **Icônes**: les icônes ont été copié-collées (en SVG) depuis [heroicons.com](#), elles 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, et .mp3 pour la validation lors de la création d'épisode.

Aides humaines

- **M. Hurni**: conseils et retours réguliers, réponses à mes questions.
- **Gatien Jayme**: aide relecture des documents

Journal de travail

Le journal est disponible en document séparé (voir archives) ou directement sur GitHub [en Markdown](#) ou [en PDF](#).

Manuel d'installation

Toutes les informations nécessaires à l'installation du projet se trouve dans le README disponible en document séparé (voir archives) ou sur GitHub [en Markdown](#).

Archives du projet

- [podz-code.zip](#) : repository Git
- [podz-documentation.pdf](#) : cette documentation
- [podz-journal-de-travail.pdf](#) : journal de travail du projet
- [podz-résumé-tpi.pdf](#) : résumé du TPI
- [podz-readme.pdf](#) : le README avec la procédure de mise en place du projet