Rapport final du projet IFT-3150

Hiver 2025

Vi Phung

20178538

1 Introduction

La maison MONA est un organisme sans but lucratif qui a pour mission d'inviter à des rencontres avec l'art, de créer un espace commun pour les échanges et de faire résonner les sens et les sensibilités de chacun.

Le projet **MONA** est une application mobile développée par ***La maison MONA*** qui permet de découvrir des œuvres d'art publiques à Montréal. L'application utilise la géolocalisation pour permettre à l'utilisateur de découvrir des œuvres d'art à proximité et de les ajouter à sa collection.

Ce projet a été créé en 2016 par *Lena Krause* dans le cadre d'un cours d'informatique de l'Université de Montréal. Depuis lors, plusieurs générations d'étudiants se sont succédé, sous la direction du professeur *Guy Lapalme* et de *Lena Krause*, pour poursuivre le développement de cette application. Au cours du semestre d'hiver 2025, j'ai eu le plaisir de prendre part à ce projet dans le rôle de développeur serveur. Les langages utilisés sont surtout **PHP**, avec le framework **Laravel**.

Mon projet est sous la supervision du professeur **Guy Lapalme** et de **Fabian Bastin**.

Lien vers le page des rapports :

https://kvpl7102.github.io/IFT3150-MONA-Rapports/

2 Changements effectués pendant mon projet

Résumé du projet

Durant ce projet, j'ai rejoint l'équipe de développement serveur de l'application MONA. Mon mandat peut se diviser principalement en trois phases :

- **Installation et Familiarisation avec la structure du serveur :** exploration approfondie du code existant, prise en main des outils utilisés (Laravel, Docker, Git), ainsi que compréhension du fonctionnement général de l'infrastructure du serveur.
- **Génération de la documentation API :** utilisation de l'outil Scramble pour automatiser la création et la mise à jour régulière de la documentation OpenAPI des routes existantes, afin de faciliter leur compréhension et leur utilisation par les autres membres de l'équipe.
- **Ajout de fonctionnalités CRUD aux routes de l'API :** implémentation progressive de nouvelles opérations CRUD manquantes (create, read, update, delete) sur les ressources principales de l'application (artworks, places, heritages, artists), améliorant ainsi les capacités fonctionnelles et la maintenabilité du code serveur.

2.1 Installation et Familiarisation avec la structure du serveur

Au début de mon mandat, l'une des premières tâches consistait à installer et configurer localement le serveur de l'application MONA. Cette étape impliquait notamment l'utilisation des technologies telles que Laravel, PHP et Docker. Pour ce faire, j'ai suivi les instructions transmises par Corelie, qui travaillait également sur ce projet dans le cadre du cours IFT3150 précédemment.

Durant l'installation et la configuration initiale, j'ai rencontré des difficultés et des conflits liés à la version de PHP. Pour résoudre ces problèmes, j'ai sollicité l'aide de Simon, développeur serveur lui-même chez MONA, avec qui j'ai continué à collaborer étroitement par la suite. Grâce à son assistance, j'ai pu installer et configurer toutes les dépendances nécessaires au bon fonctionnement du serveur sur mon poste local.

Une fois l'environnement opérationnel, Simon m'a présenté l'infrastructure générale du serveur MONA. Il m'a guidé dans la migration des bases de données vers mon poste local, m'a expliqué le processus pour effectuer des requêtes API, et m'a enregistré en tant qu'administrateur sur le serveur. Cette introduction m'a permis d'acquérir une compréhension claire et approfondie de l'infrastructure existante, facilitant grandement les tâches futures que j'allais entreprendre.

Pendant ce temps d'installation, j'ai aussi essayé de me familiariser avec le code. Je me suis renseigné sur le fonctionnement de Laravel, Vue.js et Docker en lisant la documentation et en regardant des tutoriels puisque c'est ce qui est utilisé pour l'interface et que je n'avais aucune expérience dans ce domaine.

2.2 Génération de la documentation des APIs

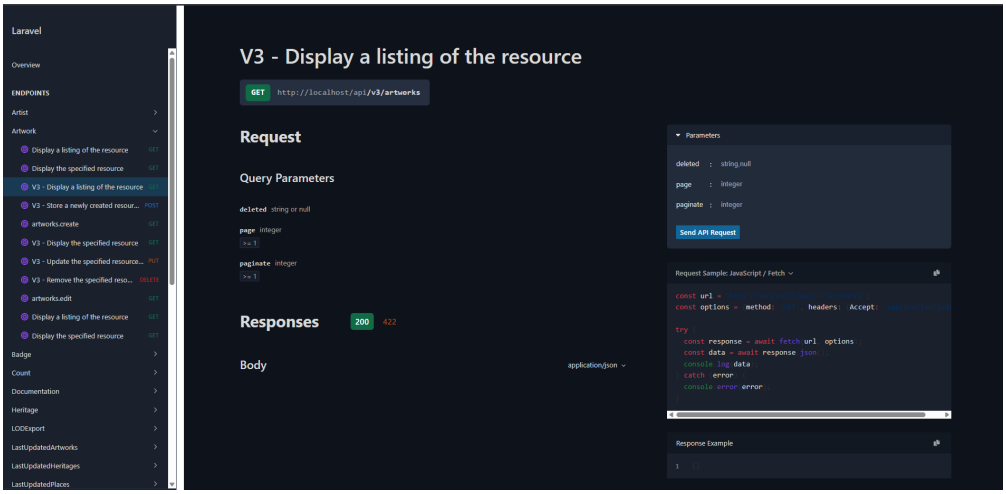
Après l'installation réussie de l'environnement serveur, j'ai entamé mon travail concret avec les APIs. Mon intérêt principal portait sur les appels API déjà existants du serveur MONA, notamment pour les ressources telles que *Artist, Artwork, Heritage* et *Place*. Pour mieux comprendre les fonctionnalités disponibles, j'ai passé du temps à étudier attentivement la documentation interne de ces APIs disponible sur le wiki du projet.

J'ai débuté en créant une pull request sur le dépôt `mona-server`, intitulée *"Generation of APIs documentation and code linting"*. L'objectif principal était de générer automatiquement une documentation claire des routes API existantes en utilisant Scramble, un outil facilitant la création de documentations OpenAPI directement à partir du code source.
Ce processus peut être divisé en plusieurs étapes distinctes :

Génération initiale de la documentation OpenAPI : L'objectif ici était de produire automatiquement la documentation OpenAPI pour toutes les routes API déjà existantes. L'outil utilisé était Scramble (`https://scramble.dedoc.co/`). J'ai rencontré initialement quelques difficultés lors de son installation, principalement dues à des conflits de versions PHP avec celles utilisées dans le projet MONA. Ce problème a été finalement résolu en utilisant les commandes Laravel Sail fournies par Docker. Une fois installé, Scramble génère automatiquement la documentation OpenAPI et l'affiche via les routes `/docs/api` (interface utilisateur) et `/docs/api.json` (format JSON).

Ajout des commentaires PHPDoc : Pour enrichir et compléter la documentation générée, j'ai ajouté des commentaires au format PHPDoc dans le code. Ces commentaires fournissent des informations supplémentaires essentielles à la compréhension des routes et de leurs fonctionnalités. Pour déterminer précisément quelles informations étaient nécessaires, je me suis basé sur les recommandations du wiki interne (`https://github.com/MaisonMONA/mona-server/wiki`). Ce travail m'a pris un temps considérable en raison du nombre important de routes à documenter.

Après la finalisation de cette étape, voici un aperçu de la documentation disponible via l'interface à l'URL `/docs/api` :



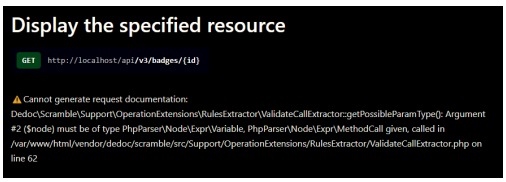
Pendant l'ajout des commentaires PHPDoc, j'ai rencontré plusieurs erreurs qui affectaient la génération de la documentation :

Erreur 1 : Blocs de commentaires indésirables

— **Problème :** Certains blocs de code commentés (par exemple, une boucle `foreach` dans `UserPlaceController`) apparaissaient dans la documentation générée.
— **Résolution :** Supprimer ou déplacer ces commentaires pour éviter leur interprétation en tant que documentation.

Erreur 2 : Problème de type dans ValidateCallExtractor

— **Problème :** L'appel suivant dans des méthodes des controlleurs générait une erreur : `Validator::make(request()->route()->parameters(), [...])`

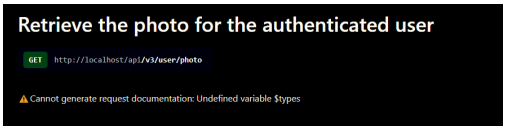


— **Résolution :** Assigner les paramètres de la route à une variable avant la validation :

```
$params = request()->route()->parameters();
Validator::make($params, [
    'artist' => ['integer', Rule::exists('artists', 'id')]
])->validate();
```

Erreur 3 : Variable non définie dans UserPhotoController

— **Problème :** Dans `UserPhotoController`, une erreur « Undefined variable $types » apparaissait.



Code original :

```
$types = ['artwork', 'place', 'heritage'];
Validator::make($request->toArray(), [
    'discovery_id' => ['required', 'integer', 'min:1'],
    'type' => ['required', Rule::in($types)]
])->validate();
```

— **Résolution :** Remplacer l'utilisation de `$types` par une définition directe :

```
Validator::make($request->toArray(), [
    'discovery_id' => ['required', 'integer', 'min:1'],
    'type' => ['required', Rule::in(['artwork', 'place', 'heritage'])]
])->validate();
```

Linting du code : Pour assurer la cohérence et la qualité du code, j'ai installé Duster pour le linting du code PHP et Prettier pour formater les vues.

Pour le PHP, j'ai exécuté les commandes suivantes pour lancer Duster et corriger automatiquement les erreurs de style :

```
./vendor/laravel/sail/bin/sail php ./vendor/bin/duster lint
./vendor/laravel/sail/bin/sail php ./vendor/bin/duster fix
```

Pour les vues (fichiers Blade), après avoir installé Prettier et le plugin dédié (prettier-plugin-blade), j'ai lancé la commande suivante :

```
./vendor/laravel/sail/bin/sail npx prettier --write "resources/views/**/*.blade.php"
```

Gestion des commits avec Git : Après avoir créé ma pull request, j'ai collaboré avec Simon pour réaliser un « rebase » de ma pull request. C'était la première fois que je découvrais cette notion avancée de gestion de versions avec Git. Concrètement, j'ai appris à regrouper mes multiples commits initiaux en seulement trois commits principaux. Cette méthode simplifie considérablement l'historique des modifications apportées. Ainsi, si à l'avenir je dois effectuer des ajustements à ma pull request, je pourrai directement modifier ces commits existants plutôt que d'en créer de nouveaux, améliorant ainsi la lisibilité et l'organisation du projet.

2.3 Ajout de fonctionnalités CRUD aux routes de l'API

Suite à l'approbation et à la fusion de ma pull request par Simon, j'ai poursuivi mon travail sur l'API. J'ai eu de nombreuses discussions avec Simon et Corelie concernant l'ajout de fonctionnalités CRUD supplémentaires aux routes de ressources de l'API. J'ai également approfondi mes connaissances sur la structure des modèles, des ressources et des routes dans le projet Laravel afin de mieux comprendre leur fonctionnement et ce que je devais faire pour ajouter de nouvelles fonctionnalités à une route.

Actuellement, l'API permet principalement d'afficher la liste des ressources ou une ressource spécifique lorsqu'on fournit son identifiant. Mon objectif était d'ajouter davantage de fonctionnalités aux routes, notamment les opérations `store`, `update` et `destroy`. J'ai compilé la liste des fonctionnalités des routes API dans les contrôleurs de ressources pour avoir une vue d'ensemble de ce qui devait être fait.

Controllers	Index	Create	Store	Show	Edit	Update	Destroy
Artist (Admin)	✓	✗	✗	✓	✗	✗	✗
Artist (API)	✓	✗	✗	✓	✗	✗	✗
Artwork (Admin)	✓	✓	✓	✓	✓	✓	✗
Artwork (API)	✓	✗	✗	✓	✗	✗	✗
Place (Admin)	✓	✓	✗	✓	✗	✗	✗
Place (API)	✓	✗	✗	✓	✗	✗	✗
Heritage (Admin)	✓	✗	✗	✓	✗	✗	✗
Heritage (API)	✓	✗	✗	✓	✗	✗	✗
Badge (Admin)	✗	✗	✓	✗	✗	✓	✗
Badge (API)	✓	✗	✗	✓	✗	✗	✗
User (Admin)	✓	✗	✗	✓	✗	✗	✗
User (API)	✗	✗	✗	✓	✗	✗	✗

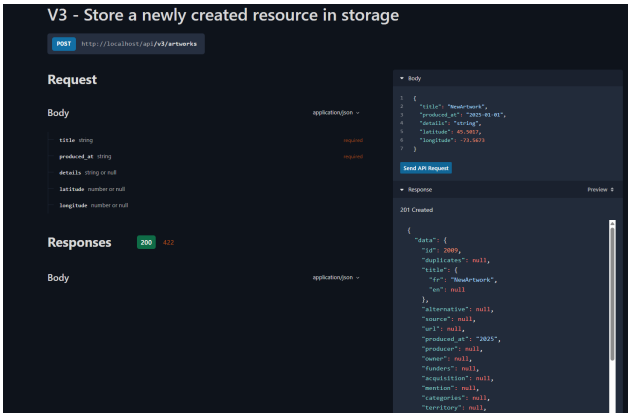
Après discussion avec l'équipe sur les fonctionnalités nécessaires pour chaque route, j'ai commencé la mise en œuvre des fonctionnalités manquantes pour l'API, en me basant sur le modèle de ressources du dépôt mona-server. Cette tâche m'a pris un temps considérable, particulièrement lors de la vérification de la compatibilité de mon code avec notre modèle de ressources existant. Voici les fonctions que j'ai implémentées pour les contrôleurs API de ressources :

Controllers	Index	Create	Store	Show	Edit	Update	Destroy
Artist (API)	✓	✗	✓	✓	✗	✓	✓
Artwork (API)	✓	✗	✓	✓	✗	✓	✓
Place (API)	✓	✗	✓	✓	✗	✓	✓
Heritage (API)	✓	✗	✓	✓	✗	✓	✓
Badge (API)	✓	✗	✓	✓	✗	✓	✓

Les méthodes `create` et `edit` dans Laravel retournent uniquement le formulaire HTML pour la requête, ce qui n'est pas nécessaire pour nos API qui renvoient des données JSON.

Après avoir ajouté les fonctionnalités supplémentaires, j'ai enregistré ces dernières dans la liste des routes du serveur.

Voici un exemple de la nouvelle fonction `store` pour l'API `Artwork` :



J'ai ensuite créé une nouvelle pull request intitulée « Add CRUD features for resource APIs » dans le dépôt `mona-server`.

3 Conclusion

Participer à ce projet au sein d'une équipe aussi agréable et poursuivant un objectif captivant a été pour moi une expérience très enrichissante. Les connaissances et compétences que j'ai pu développer durant cette période constitueront sans aucun doute des atouts précieux pour ma carrière future.

Les principaux défis que j'ai rencontrés étaient liés à mon manque de familiarité avec les technologies utilisées, ainsi qu'au fait d'intégrer un projet déjà en cours de développement. Comprendre l'architecture du serveur s'est révélé particulièrement complexe sans avoir participé à sa configuration initiale. Chaque amélioration ou ajout nécessitait au préalable une recherche approfondie dans la documentation et l'analyse minutieuse du code existant. Cette expérience m'a fait prendre conscience que je serai régulièrement amenée à rejoindre des projets en cours de développement dans ma vie professionnelle, et j'ai désormais une meilleure appréciation de l'importance cruciale d'une documentation claire et complète.

Quelques idées pour la suite

Bien que le serveur fonctionne généralement bien, il y a quelques tâches qu'il serait bien de réaliser :

- Vérifier des nouvelles fonctions des routes APIs
- Travailler encore pour améliorer la documentation des routes API
- Ajouter des pages de vues manquantes pour les routes dans l'interface admin