MONA : une application mobile iOS sur l'art public à Montréal

par

Paul CHAFFANET

PROJET PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC PROJET EN GÉNIE DES TECHNOLOGIES DE L'INFORMATION
M. Ing.

MONTRÉAL, LE 12 DÉCEMBRE 2019

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE RAPPORT DE PROJET A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

Professeur Alain April, directeur de projet
Département de Génie Logiciel et TI à l'École de technologie supérieure

Abdelaoued Gherbi, président du jury
Département de Génie Logiciel et TI à l'École de technologie supérieure

REMERCIEMENTS

Je remercie Lena Krause, étudiante à la maîtrise en histoire de l'art à l'Université de Montréal, pour m'avoir proposé ce projet et pour avoir grandement contribué à son développement par l'organisation d'activités de rencontres avec les utilisateurs.

Je veux également remercier Gilbert Fortin et Roberto Martinez, étudiants à l'école de design de l'Université de Montréal, qui ont apporté une contribution très importante à l'édifice de cette application en concevant les prototypes d'interfaces usager, les icônes et les badges de l'application.

Je désire remercier Guy Lapalme, ancien professeur au département d'informatique et de recherche opérationnelle de l'Université de Montréal, pour avoir continué à suivre très attentivement ce projet d'application mobile durant ma maîtrise. Guy Lapalme m'a guidé durant le développement de l'application par la dispense de ses conseils avisés et m'a permis d'améliorer l'application en étant mon meilleur testeur et utilisateur.

J'adresse également mes remerciements les plus sincères à Suzanne Paquet, directrice du département d'histoire de l'art et d'études cinématographiques de l'Université de Montréal, pour avoir cru en ce projet.

Enfin, je souhaite remercier Alain April, mon directeur de recherche, pour son encadrement et ses commentaires pour la rédaction du rapport de ce projet.

MONA : une application mobile iOS sur l'art public à Montréal

Paul CHAFFANET

RÉSUMÉ

MONA est un projet d'application mobile iOS développée en Swift qui permet aux utilisateurs de partir à la découverte de la collection d'art public de la ville de Montréal et de l'Université de Montréal de manière ludique. L'application a pour ambition de proposer une nouvelle forme d'interaction avec les œuvres d'art public de Montréal et d'obtenir des données sur l'appréciation de ces œuvres pouvant être utiles à la recherche universitaire en art.

Une première version de MONA, non distribuée sur l'App Store, avait déjà été développée. Le rapport suivant porte sur un projet de maîtrise visant à développer une nouvelle version de cette application beaucoup plus performante et avec un tout nouveau désign en ayant comme but la distribution de l'application sur l'App Store.

Le premier chapitre de ce rapport est une revue de littérature qui permet au lecteur de mieux comprendre le contexte dans lequel s'inscrit cette application. Ce chapitre traite la question de l'impact de cette application, des données sur lesquelles elle s'appuie, du fonctionnement de la plateforme visée (iOS) et de la documentation utilisée pour son développement. Le deuxième chapitre décrit les objectifs et besoins de l'application. Les troisième et quatrième chapitres de ce rapport décrivent respectivement la phase de développement du serveur Web et de l'application. Chacun de ces chapitres décrit l'implémentation de la première version et identifie les problèmes posés par cette implémentation. Puis ils détaillent les objectifs et l'implémentation de la nouvelle version. Le cinquième et dernier chapitre présente les objectifs atteints et les pistes d'amélioration pour le serveur Web et l'application.

Mots-clés: iOS, application mobile, art public, Montréal

MONA : an iOS mobile application sur on public art in Montréal

Paul CHAFFANET

ABSTRACT

MONA is an iOS mobile application project developed in Swift that allows users to discover the public art collection of the City of Montreal and the Université de Montréal in a fun way. The application aims to propose a new form of interaction with Montreal's public artworks and to obtain data on the appreciation of these works that can be useful for university art research.

A first version of MONA, not distributed on the App Store, had already been developed. The following report is about a master project to develop a new version of this much more powerful application and with a brand-new design with the goal of distributing the application on the App Store.

The first chapter of this report is a literature review that provides the reader with a better understanding of the context of this application. This chapter discusses the impact of this application, the data on which it is based, the operation of the targeted platform (iOS) and the documentation used for its development. The second chapter describes the objectives and needs of the application. The third and fourth chapters of this report describe the development phase of the Web server and the application respectively. Each of these chapters describes the implementation of the first version and identifies the problems caused by this implementation. Then they detail the objectives and implementation of the new version. The fifth and last chapter presents the objectives achieved and the areas for improvement for the Web server and the application.

Keywords: iOS, mobile application, public art, Montreal

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

Page

LISTE DES ALGORITHMES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

AÉÉHAUM	Association des Étudiantes et Étudiants en Histoire de l'Art à l'Université de Montréal
API	Application Programming Interface
CC	Creative Commons
iOS	iPhone Operating System
JSON	JavaScript Object Notation
DIRO	Département d'Informatique et de Recherche Opérationnelle
ÉTS	École de Technologie Supérieure
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MoMa	The Museum of Modern Art (New York City (NY), États-Unis)
MeT	The Metropolitan Museum of Art (New York City (NY), États-Unis)
MIME	Multipurpose Internet Mail Extensions
MVC	Modèle-Vue-Contrôleur
OKF	Open Knowledge Foundation
PNG	Portable Network Graphics
QR	Quick Response Code
REST	REpresentational State Transfer
SHA	Secure Hash Algorithm
SQL	Structured Query Language

UdeM	Université de Montréal
URL	Uniform Resource Locator
WWDC	Worldwide Developer Conference

INTRODUCTION

MONA est un projet d'application mobile qui permet aux utilisateurs de partir à la découverte de la collection d'art public de la ville de Montréal et de l'Université de Montréal (UdeM). Le but, pour l'utilisateur, est de prendre en photo les œuvres d'art localisées par l'application afin de pouvoir les ajouter à sa propre collection, et ainsi gagner des récompenses, par exemple des badges. Après avoir pris en photo une œuvre, l'application permet de noter sur une échelle de 1 à 5 l'appréciation de cette œuvre d'art, puis de la commenter.

Ce projet d'application a été imaginé par Lena Krause [1], étudiante à la maîtrise en histoire de l'art à l'UdeM. Il a pour ambition de proposer aux utilisateurs un accès à l'art public et un moyen d'expression au sujet de ces œuvres, tout en collectant, pour les initiateurs de ce projet, une multitude de données utiles pour l'étude des publics de l'art à l'ère numérique. La collecte des données permet d'accumuler des photos, des notes et des commentaires provenant du public utilisateur.

Ce projet d'application mobile a été initié à l'hiver 2018 par le groupe de recherche Art+Site [2] à l'aide de la professeure Suzanne Paquet, directrice du Département d'Histoire de l'Art et d'Études cinématographiques de l'UdeM et chercheuse principale de ce groupe de recherche. Il a été mené en collaboration avec le Département d'Informatique et de Recherche Opérationnelle (DIRO) de l'UdeM et supervisé par Lena Krause. Alors étudiant au baccalauréat informatique à l'UdeM, j'ai participé au développement d'une première version de cette application sur la plateforme iOS dans le cadre d'un projet informatique crédité. Ce projet incluait également la participation d'Émile Labbé pour le développement de l'application sur la plateforme Android, et de Vincent Beauregard pour le développement de l'API sur un serveur du DIRO.

Gilbert Fortin et Roberto Martinez, étudiants à l'École de design de l'UdeM, ont rejoint l'équipe lors de la session d'été 2018 durant laquelle nous avons travaillé à proposer un nouveau design pour l'application et une série de badges d'arrondissements.

Bien que le nouveau design de l'application ne fût pas implémenté à la session d'automne 2018, nous avons tout de même organisé avec l'Association des Étudiantes et Étudiants en Histoire de l'Art à l'Université de Montréal (AEEHAUM) un test bêta de l'application iOS parallèlement à l'activité de la rentrée 2018 des étudiants. Au cours de cette activité, les étudiants ont pu nous aider dans l'identification des problèmes et ont pu nous fournir des pistes d'amélioration pour cette première version. Ce test bêta fut aussi l'occasion pour les étudiants d'explorer et de découvrir le campus de l'UdeM et ses œuvres d'art public. À la suite du succès de cette activité, l'AEEHAUM a remporté le deuxième prix du concours pour l'organisation des activités de la rentrée de l'année 2018 [3].

Le projet a ensuite repris de plus belle à partir de l'hiver 2019. Émile Labbé et Vincent Beauregard ayant quitté le projet, Matija Dabić, étudiant au baccalauréat mathématiques-informatique à l'UdeM, a rejoint le projet à titre de développeur Android tandis qu'Abdelhakim Qbaich, étudiant en baccalauréat informatique à l'UdeM, a refait l'API de MONA initiée par Vincent Beauregard, en utilisant le cadriciel « Laravel ».

Pour ma part, j'avais l'intention de poursuivre le développement de l'application sur la plateforme iOS dans le cadre de mon projet de maîtrise appliquée à l'École de Technologie Supérieure (ÉTS) à partir de la session d'été 2019. Les objectifs fixés pour ce projet d'application sont d'implémenter les besoins qui n'ont pu être comblés par la première version de l'application, d'intégrer les nouveaux changements dans le design, d'adapter l'application à la nouvelle API et de construire une version plus complète et performante de l'application, avec ultimement comme but d'aboutir à la publication de l'application sur l'App Store.

CHAPITRE 1

Revue de littérature

Dans un premier temps, nous étudierons l'impact des nouvelles technologies numériques sur le monde de l'art dans le but de mieux comprendre le contexte et la portée dans lesquels s'inscrit l'application MONA. Après avoir traité cette question, nous verrons sur quel type de données l'application MONA s'appuie pour remplir son objectif de découverte de l'art public à Montréal. Dans un troisième temps, le fonctionnement de la plateforme iOS visée par cette application sera décrit. En particulier, nous verrons quels sont les cadriciels de la plateforme qui a permis le développement de MONA. Enfin, le dernier sous-chapitre détaillera la documentation qui aura été nécessaire pour le développement de l'application.

1.1 Art et numérique

Il serait tout d'abord intéressant de se demander quel impact les nouvelles technologies numériques, telles qu'Internet ou les téléphones intelligents, ont-elles eu sur le domaine de l'art dans nos sociétés. Olivier Donnat juge que [4], malgré une évolution des conditions d'accès à la culture (c.-à-d. l'art, la musique, le cinéma, etc.) se caractérisant par « la dématérialisation des contenus », « la généralisation de l'Internet à haut débit », et l'augmentation de la présence des ordinateurs, tablettes et téléphones intelligents dans les foyers, la révolution numérique n'a eu qu'un impact relativement limité sur les pratiques culturelles en France. Certes, l'avènement d'Internet au milieu des années 90 et du téléphone intelligent au milieu des années 2000 a constitué des technologies disruptives indéniables à la source de déséquilibres économiques majeurs. Prenons pour exemple l'arrivée d'iTunes au début des années 2000 sur le marché de la musique. La plateforme iTunes a transformé le mode de consommation de la musique, en passant d'un acte d'achat physique de disque à celui d'un achat numérique. Puis, s'en est suivie, à la fin des années 2000, l'arrivée d'acteurs du « streaming » musical, tel que Spotify qui, à leur tour, ont bouleversé l'équilibre économique et le mode de consommation de la musique en place jusqu'alors. Pour autant, ces nouvelles technologies n'ont fait que prolonger la tendance d'une écoute de la musique, qui est à la hausse depuis les années 60 grâce à l'arrivée

de la chaîne haute-fidélité et du baladeur. Pour en revenir à l'art, on peut constater que l'intérêt pour l'art est relativement stable depuis les années 70 et n'a pas changé avec la révolution numérique, bien que les modes de consommation aient pu évoluer [5].

En effet, Internet a permis de fournir un accès facilité à une multitude de collections d'œuvres d'art [6] grâce à des plateformes comme Google Arts & Culture [7], grâce aux données mises à disposition par des musées comme le MoMa et le MeT [8]. Les réseaux sociaux, comme Facebook et Instagram, sont également des plateformes qui ont impacté le monde de l'art, facilitant la diffusion d'œuvres d'art, la construction et la gestion d'une communauté pour les artistes et les musées [9]. Une communauté qui est d'ailleurs souvent jeune et peu encline à se rendre dans les galeries d'art ou aux expositions [10]. Instagram constitue aussi un tremplin pour de jeunes artistes qui peuvent créer leur propre galerie d'art virtuelle à peu de frais dans le but de percer dans le monde de l'art en attirant l'œil des exposants. Enfin, les réseaux sociaux, et Internet de manière générale, sont de nouveaux outils qui permettent aux amateurs d'art de partager leur expérience durant une exposition ou une visite de musée, et d'augmenter l'engagement des visiteurs par une interactivité accrue [11; 12]. Bien que les réseaux sociaux facilitent le processus de découverte d'œuvres, le rapport physique aux œuvres d'art reste essentiel pour pouvoir les apprécier à leur juste valeur [13].

Les téléphones intelligents sont également un nouvel outil issu de la révolution numérique qui a impacté le monde de l'art. En témoigne le Mobile Art, un mouvement artistique contemporain qui cherche à utiliser un mode d'expression artistique parmi toutes les fonctionnalités du téléphone intelligent, telles que la photographie, la vidéo ou la réalité augmentée [14]. Grâce à leurs fonctionnalités de localisation et à des codes QR, les téléphones intelligents offrent aussi la possibilité de fournir une information contextuelle propre à chaque œuvre [8]. C'est ainsi que le MoMa s'appuie sur cet outil en proposant une application mobile à télécharger afin de disposer d'un guide audio durant la visite du musée.

L'Internet ainsi que le téléphone mobile permettent au domaine de l'art de valoriser et démocratiser leurs collections d'art. Comme chez les artistes ou les musées, la révolution

numérique dans le domaine de l'art souffre d'importantes disparités. Bien que des artistes ou des musées célèbres comme le MoMa sachent tirer leur épingle du jeu, il n'est pas facile pour les artistes ou les structures moins réputés d'obtenir une meilleure exposition. Par exemple, le Musée des beaux-arts de Montréal possède une page Facebook de 140 000 adeptes contre près de 2M d'adeptes pour le MoMa [8].

C'est donc sur cette toile de fond que s'inscrit l'application MONA. Souhaitant inciter l'utilisateur à découvrir la collection d'œuvres d'art public de la ville de Montréal, cette application cherche à encourager un mode d'expression artistique par la photographie dans le but de constituer pour l'utilisateur sa propre collection d'œuvres d'art. L'utilisateur peut évaluer son expérience en ajoutant une note et un commentaire sur chaque œuvre. Il dispose d'information contextuelle, tel que l'artiste ou la date d'une œuvre, grâce à son positionnement sur une carte géolocalisée. L'application cherche également à encourager l'utilisateur à étoffer sa collection d'œuvres par l'obtention de badges.

Bien qu'il existe de nombreuses définitions et interprétations sur ce qu'est l'art public [15; 16], nous nous appuierons sur la définition d'art public énoncée par Art Public Montréal, car celle-ci correspond à la vision de l'art public de la ville de Montréal: « le terme art public désigne l'ensemble des œuvres d'art situées dans des lieux d'accès public, extérieur ou intérieur. Les œuvres d'art public sont pérennes et installées dans des aires publiques communes. […] Les œuvres peuvent être liées à leur site ou non, s'adapter à leur environnement en étant en harmonie ou en contraste avec celui-ci, selon l'intention de l'artiste. » [17]. Cette définition d'art public permet de regrouper entre autres le mobilier urbain (c.-à-d. statues, bancs, etc.) ou des éléments qui participent à l'embellissement d'un espace public tel que les murales.

Classée 2^{e} meilleure ville au Canada pour les artistes selon Zolo [18] et 5^{e} meilleure ville pour le « street art » au monde selon USA Today [19], Montréal est une ville qui se veut être un hub artistique. Accueillant le plus grand évènement de « street art » d'Amérique du Nord, le Festival Mural attire pas moins d'un million de visiteurs [19]. La ville de Montréal soutient par ailleurs activement le développement de sa collection d'œuvres d'art public grâce à son

Bureau d'art public [20]. Montréal a ainsi tout intérêt à faire valoir ses atouts dans l'art grâce à de nouvelles technologies (comme les applications mobiles) afin de rester une place forte dans le domaine de l'art urbain. Ainsi, l'application MONA permet de valoriser et démocratiser l'accès à ces œuvres d'art. De plus, grâce à l'évaluation des œuvres faites par les utilisateurs (c.-à-d. la note et le commentaire), MONA permet une collecte de données intéressante dans le but d'améliorer continuellement ce musée à ciel ouvert et de servir la recherche universitaire dans le domaine de l'art.

1.2 Données ouvertes (de l'anglais « open data »)

Selon la définition proposée par l'Open Knowledge Foundation (OKF), « open data and content can be freely used, modified, and shared by anyone for any purpose » [21]. La ville de Montréal a mis en place une politique d'ouverture des données qui respecte cette définition. Cette politique vise à constituer un « vecteur de développement économique » et d'innovation ouverte et à améliorer la transparence de l'action publique pour les citoyens [22].

MONA s'appuie sur un ensemble de données sur les œuvres d'art public mis à disposition par la ville de Montréal [23]. Cet ensemble est disponible sous une licence d'utilisation Creative Commons 4.0 Attributions (BY) qui respecte la définition de données ouvertes [24]. Cette licence d'utilisation nous permet de partager ces données (copie, distribution et communication des données) et de les adapter (en les ajoutant à d'autres ensembles de données par exemple), avec pour seule obligation d'en attribuer la paternité à la ville de Montréal.

1.3 Le système d'exploitation iOS

iOS est le système d'exploitation propriétaire développé par Apple pour ses appareils mobiles et dont la première version a été publiée en 2007. Ce système d'exploitation est dérivé du système d'exploitation macOS et fait donc parti de la famille des systèmes d'exploitation Unix. Il est programmé en C, C++, Objective-C et Swift et est disponible pour l'iPhone et l'iPad [25].

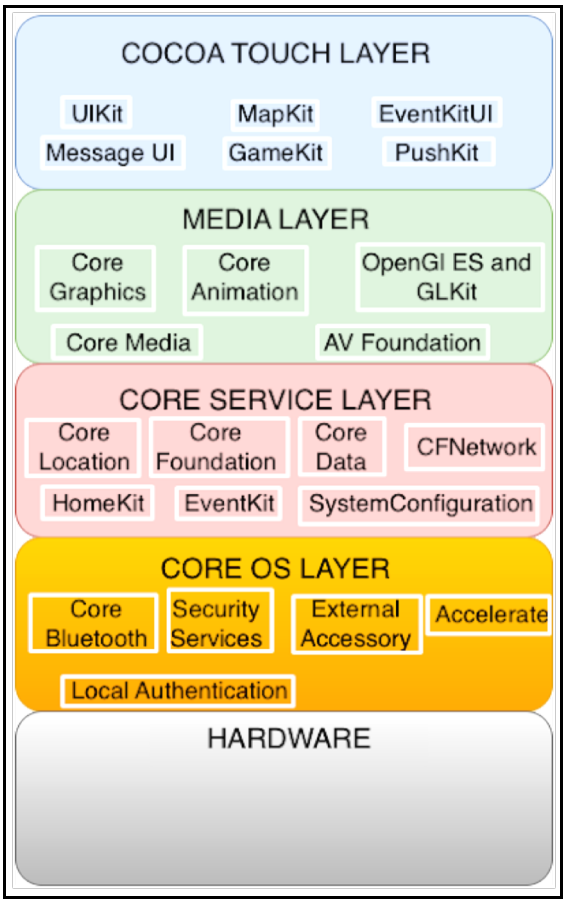


Figure 1.1: Architecture haut-niveau d'iOS [26]

L'architecture d'iOS est organisée autour d'une pile de quatre couches de cadriciels ou d'API [26; 27; 28]:

- **« Cocoa Touch »**: dérivée de la couche « Cocoa » de macOS, cette couche est la plus haute de l'architecture iOS. Elle fournit un ensemble de cadriciels et API qui permettent de définir l'interface graphique d'une application et les réponses aux évènements. MONA utilise notamment deux cadriciels de la couche « Cocoa Touch ». 1) « UIKit », qui fournit une architecture de fenêtres et de vues pour l'implémentation d'une interface et une infrastructure de gestion d'évènements, tels que des évènements tactiles. 2) « MapKit », un cadriciel permettant d'intégrer et d'interagir avec une carte contenant des annotations.
- **Media**: la couche « Media » est la seconde couche de l'architecture iOS. Elle fournit un ensemble de cadriciels et d'API qui permettent de lire, enregistrer ou éditer des

données audiovisuelles. Elle permet également la création d'animation et le rendu graphique en 2D ou 3D. Le cadriciel « AVFoundation », qui appartient à cette couche, a ainsi été utilisé dans MONA afin de pouvoir accéder à l'appareil photo et permettre la prise de photo des œuvres d'art. MONA utilise également le cadriciel « Core Animation » pour créer certaines animations de vues;

- **« Core Services »**: est la troisième couche, plus profonde, de l'architecture iOS. « Core Services » fournit un ensemble de services fondamentaux pour le développement d'application iOS. Les cadriciels et API des couches « Cocoa Touch »et « Media » s'appuient sur les cadriciels et API de cette couche pour proposer leurs services. Par exemple, « MapKit » utilise les services fournis par « Core Location » qui permettent d'avoir accès à la localisation de l'appareil. Parmi les cadriciels de cette couche, MONA utilise notamment « Core Data » pour la gestion et la persistance des données de l'application, et le « Grand Central Dispatch » pour l'exécution concurrente de tâches avec un processeur multicœur;
- **« Core OS »**: est la quatrième couche, la plus profonde, de l'architecture iOS. Cette couche fondamentale, sur laquelle s'appuient les couches supérieures, propose des services de sécurité (par exemple: chiffrement/déchiffrement…) et de réseau (par exemple: Bluetooth…). Elle permet également d'effectuer des tâches de calculs de manière performante (par exemple: des opérations complexes impliquant des vecteurs).

Au-delà de la couche « Core OS », on retrouve le système d'exploitation iOS qui gère entre autres l'interaction avec les pilotes matériels, les différents systèmes de fichier et la communication entre les différents processus.

En 2019, iOS est présent sur environ 22% des appareils mobiles dans le monde et sur environ 53% des appareils mobiles au Canada d'après GlobalStats [29]. L'« App Store » d'Apple distribue plus de 2 200 000 applications [30]. La plateforme iOS est donc une plateforme incontournable lorsque l'on envisage le développement d'une application mobile au Canada.

1.4 Développement d'une application iOS

Il existe deux façons de développer une application iOS, du type de MONA, qui peut être distribuée sur l'App Store:

- **Développement natif**: la première façon est de développer une application en code natif. Apple permet de programmer une application avec le langage de programmation Objective-C ou Swift et le logiciel Xcode. Le principal avantage de cette approche est la performance des applications développées pour cette plateforme et la documentation abondante pour le développement d'application iOS native;
- **Développement hybride**: la deuxième façon est de développer une application multi plateforme pour iOS et Android. Le principal avantage de cette approche est de n'avoir à travailler que sur un seul code source qui fonctionnera sur les deux plateformes. En revanche, la performance des applications développées avec cette approche est impactée. Il existe plusieurs cadriciels qui utilisent différentes stratégies pour permettre le développement d'application mobile multi plateforme. Par exemple, « Cordova » et « Ionic » utilisent une vue Web pour exécuter du code HTML/CSS/JavaScript. Cette vue Web est ensuite empaquetée dans une application native iOS et Android. Malheureusement, ce type de stratégie offre un accès limité aux API de chaque plateforme. « Xamarin », quant à lui, est un cadriciel de développement multi plateforme qui propose une stratégie différente. Il suggère de développer une application en C#, et le cadriciel se charge de traduire les éléments d'interface en code natif propre à chaque plateforme. Il laisse le choix aux développeurs d'utiliser du code natif à chaque plateforme pour créer des éléments d'interfaces plus personnalisés. La logique de l'application pourra toujours être partagée entre Android et iOS. Cette approche est bien plus performante que celle proposée par « Ionic », car elle fait directement appel à du code natif, bien qu'ayant un niveau d'abstraction supérieur par rapport à du code natif pur. Elle donne également un accès complet aux API, car on peut utiliser le code natif directement dans l'application.

Lorsque ce projet d'application a été lancé, au début de la session d'hiver 2018, nous avons été confrontés à la question du choix entre un développement natif ou un développement hybride de l'application. Bien que l'approche d'un développement multi plateforme semble séduisant au premier abord, il fallait prendre également en compte le contexte dans lequel nous nous trouvions, c'est-à-dire:

- Nous étions trois développeurs. À ce moment-là, personne n'avait d'expérience avec la plateforme iOS et les langages Objective-C et Swift et nous avions tous les trois une expérience très modeste en Java/Android;
- Nous avons testé pendant une semaine « Xamarin » et nous nous sommes rapidement rendus compte qu'une connaissance du langage natif était nécessaire pour permettre un développement multi plateforme. De plus, l'application étant essentiellement orientée vers l'affichage d'informations, le partage de code entre les deux plateformes iOS et Android aurait été relativement limité;
- Étant trois développeurs, l'organisation la plus naturelle et efficace était d'affecter un développeur au « back-end », un développeur à la plateforme Android, et un développeur à la plateforme iOS;
- Enfin, la documentation abondante pour le développement d'application native par rapport au manque de documentation pour les cadriciels nous inquiétait également pour aboutir à une première version viable de l'application.

J'ai donc été affecté au développement de MONA sur la plateforme iOS, bien que n'ayant au début du projet aucune connaissance d'Objective-C ou de Swift, et encore moins des cadriciels et API créés par Apple. Il a donc été nécessaire de me former tout au long de mon projet informatique en hiver 2018 et tout au long de ce projet de maîtrise. Pour cela, je me suis appuyé sur de nombreux supports.

On peut se former aux rudiments et spécificités du langage Swift grâce à sa documentation disponible en ligne [31]. Le tutoriel d'introduction au développement d'applications natives iOS d'Apple [32] ainsi que le cours en ligne de l'Université Stanford « Developing iOS 11 Apps with Swift » [33] sont d'excellentes références pour débuter le développement

d'application iOS. Les tutoriels vidéo, comme la chaîne « Youtube Lets Build That App » [34], sont également très utiles pour apprendre des points plus précis du développement d'application iOS.

La majeure partie du développement de l'application pour ce projet de maîtrise a lieu tout au long de la session d'été 2019. Durant cette période, il a été nécessaire d'approfondir les connaissances sur certains cadriciels utilisés dans cette application. Naturellement, la documentation d'Apple [35] a été une source d'information essentielle, bien qu'elle se révèle parfois incomplète. C'est pour cette raison qu'il est souvent nécessaire de recourir aux archives de documentation d'Apple [36], plus complète que la documentation actuelle, bien qu'écrite en Objective-C. Les vidéos des WWDC d'Apple [37] permettent d'apprendre l'utilisation de certains cadriciels, comme « Core Data », où ce sont les ingénieurs qui ont développé ces cadriciels qui viennent présenter eux-mêmes la manière de les utiliser.

Des sites Internet indépendants d'Apple peuvent être aussi d'une grande aide durant le développement d'une application. Le site « Medium » [38] est un lieu de partage contenant de nombreux articles de blogues de développeurs professionnels qui détaillent de bonnes pratiques ou des retours d'expériences dans le développement iOS. Il existe également des sites Internet qui proposent gratuitement ou contre un paiement des tutoriels sur le développement d'application iOS. Comparés à la documentation d'Apple, ces tutoriels ont l'avantage d'entrer plus spécifiquement dans la mise en pratique des cadriciels iOS par l'exemple. Dans cette catégorie de sites Internet, on retrouve « Raywenderlich » [39] qui offre de nombreux tutoriels de qualité, et qui incluent des livres tutoriels [40] pour approfondir la maîtrise de ces cadriciels. Citons également « App Coda » [41; 42] et « Hacking With Swift » [43] qui entrent dans cette catégorie. Les livres constituent aussi un support intéressant pour apprendre le développement d'application iOS [44; 45]. Ceux-ci étant écrits par des développeurs iOS reconnus, ils peuvent se révéler être de meilleure qualité que les sites Internet sur certains aspects. Pour conclure cette liste de références, citons tout de même « StackOverflow » [46], un outil d'aide incontournable pour résoudre des problèmes variés avec Swift et iOS. Bien que cette source

soit de qualité variable, elle est souvent d'une grande aide pour résoudre des problèmes précis (par exemple: renseigner sur un bug d'un cadriciel) lors du développement d'une application.

L'autoformation n'a pas été chose facile malgré l'abondance de ressources. Ne connaissant personne dans mon entourage direct (c.-à-d. amis, professeurs ou étudiants) qui maîtrisait le développement iOS, j'ai principalement procédé par essais-erreur. Cela a particulièrement été vrai pour l'apprentissage de « Core Data » qui est un cadriciel réputé difficile à maîtriser et qui m'a pris quelques semaines pour en comprendre la logique et l'utilisation.

1.5 Conclusion

Au cours de ce chapitre, nous avons vu que les nouvelles technologies numériques n'ont pas impacté fondamentalement la propension à la consommation de l'art, mais qu'elles ont en fait évolué les modalités. Internet facilite l'accès et le partage de l'art sur les réseaux sociaux tel qu'« Instagram », tandis que les téléphones intelligents constituent un moyen d'expression artistique, notamment par la photographie. Montréal est une ville importante dans le monde de l'art. Sa collection d'œuvres d'art public, et en particulier ses murales, est mondialement reconnue. Afin de consolider sa position de leadeur dans le domaine de l'art public, Montréal a tout intérêt à favoriser et à accompagner l'émergence de ces nouveaux modes de consommation artistiques. Dans ce contexte, l'application MONA valorise cette collection en en démocratisant l'accès grâce à la localisation géographique des œuvres. L'utilisateur peut constituer sa collection personnelle d'œuvres d'art en les prenant en photo et est encouragé à exprimer sa vision artistique par la mise en abîme.

MONA s'appuie sur les données ouvertes mises à disposition par la ville de Montréal pour la localisation des œuvres. D'une certaine manière, on peut dire que MONA démontre toute la pertinence de la politique de données ouvertes de la ville puisqu'elle permet effectivement l'émergence de projet d'utilité citoyenne.

Nous avons également vu au cours de ce chapitre que la première version du prototype d'application MONA a été développée pour la plateforme iOS. Ce système d'exploitation est organisé sur quatre couches de cadriciels et d'API qui fournissent un environnement de développement complet pour le développement d'application. MONA utilise abondamment ces cadriciels et API mis à disposition pour le développement d'interfaces, d'animations, la gestion des évènements, la localisation, la persistance des données, etc.

Enfin, nous avons vu qu'il existe deux manières de développer une application iOS: 1) le développement hybride; ou 2) le développement natif, et l'organisation de l'équipe et notre expérience de développeur se prêtaient à privilégier le développement natif. Bien que n'ayant aucune connaissance initiale sur le développement iOS, de nombreuses ressources (documentation, sites Internet, livres, vidéos…), d'origines diverses et variées, m'ont aidé au cours de la phase de développement de MONA.

CHAPITRE 2

Analyse des besoins

Dans un premier temps, nous allons définir l'objectif principal visé par le projet d'application MONA, puis dans un second temps nous procéderons à une analyse des besoins qui ont été définis en concertation avec Lena Krause [47].

2.1 Objectif principal de l'application

L'objectif de MONA est de permettre à un utilisateur de créer sa collection personnelle d'œuvres d'art public de la ville de Montréal grâce à leur localisation géographique sur une carte interactive. L'utilisateur peut exprimer sa fibre artistique en photographiant les œuvres d'une manière originale et peut communiquer son expérience, son point de vue ou son interprétation des œuvres qu'il a collectionnées par une note ou un commentaire. Les réseaux sociaux permettent de partager des photographies des œuvres et créent un lieu de débat sur l'art public montréalais. MONA vise aussi à inciter un utilisateur à découvrir les œuvres qui l'entourent grâce à un système de récompenses par badges.

Il s'agit avant tout pour MONA de stimuler la curiosité et l'intérêt pour l'art public montréalais, en favorisant, d'une part, l'exploration et la découverte de la richesse artistique de Montréal, et d'autre part, en laissant aux utilisateurs la liberté de communiquer leur appréciation et leur interprétation des œuvres.

2.2 Description des besoins

L'application devrait se diviser en quatre sections principales:

1) **Œuvre du jour**: le but de cette section est de suggérer la visite d'une œuvre particulière à l'utilisateur. Cette œuvre devrait être choisie à partir du serveur. Par exemple, si un certain jour, on fêtait les 100 ans de la production d'une œuvre, on pourrait choisir à

partir du serveur de mettre en avant cette œuvre. L'œuvre du jour pourrait aussi être choisie aléatoirement;

2) **Liste des œuvres**: le but de cette section est de permettre de découvrir et d'explorer l'ensemble de la collection de MONA sous la forme de listes triables. On y verrait les œuvres classées selon leur titre, leurs artistes, leur arrondissement, leur catégorie, leurs techniques ou leurs matériaux. Les listes d'œuvres devraient pouvoir être triables par ordre alphabétique, par date de production ou par distance par rapport à la position de l'utilisateur. Chaque élément de ces listes devrait permettre l'ouverture de la fiche d'une œuvre;
3) **Carte**: cette section permet à l'utilisateur de connaître sa position et la localisation des œuvres sur une carte interactive. La carte devrait permettre de distinguer et de filtrer les œuvres ciblées, les œuvres collectionnées et les œuvres non collectionnées. En cliquant sur une icône représentant une œuvre, l'utilisateur devrait accéder à la fiche technique correspondante à l'œuvre;
4) **Collection**: cette section contient la collection d'œuvres d'un utilisateur. Elle contient toutes les œuvres qui ont été prises en photo par l'utilisateur. Une œuvre prise en photo est une œuvre considérée comme collectionnée.

L'application devrait également remplir plusieurs fonctionnalités importantes:

- **La recherche d'œuvres**: l'utilisateur devrait pouvoir effectuer des recherches par mots-clés afin de trouver des œuvres, des artistes, des matériaux, ou encore des arrondissements. Cette fonctionnalité permettrait de retrouver des œuvres ou des artistes plus rapidement que par le parcours de liste;
- **La consultation de la fiche d'une œuvre**: la fiche d'une œuvre devrait afficher toutes les informations liées à une œuvre, en particulier son titre, son ou ses artistes, sa date de production et sa localisation. La fiche devrait aussi permettre à un usager de prendre en photo l'œuvre pour l'ajouter à sa collection personnelle. Elle devrait également permettre à l'utilisateur de l'ajouter à sa liste d'œuvres ciblées. La fiche d'une œuvre devrait afficher la photo de l'œuvre si celle-ci a été prise en photo. Elle ne pourrait afficher qu'une unique photographie, poussant ainsi un utilisateur à faire preuve de

créativité et d'inventivité pour « choisir l'angle dans lequel il souhaite immortaliser sa visite » [47]. L'utilisateur peut ensuite compléter la fiche d'une œuvre avec une note (sur une échelle de 1 à 5) et un commentaire reflétant son appréciation et son expérience personnelle vis-à-vis de l'œuvre;

- **La collecte de données**: l'application devrait pouvoir téléverser les données (comme les photographies, les notes et les commentaires) des utilisateurs sur les œuvres sur un serveur de l'UdeM. Ces données devraient ensuite être facilement accessibles sur une interface d'administration. L'une des retombées importantes de ce projet d'application mobile est de permettre un accès facilité à ces données pertinentes pour la recherche universitaire dans le domaine de l'art. Les données diffusées sur les réseaux sociaux (comme Facebook, Twitter, Instagram, etc.) par l'utilisateur devraient aussi pouvoir être collectées (comme les photographies et les commentaires);
- **Une liste d'œuvres ciblées**: lorsqu'un utilisateur découvre une œuvre qui l'intéresse, il devrait pouvoir l'ajouter à cette liste en la ciblant. Cette liste permettrait à un utilisateur de retrouver plus facilement les œuvres qui ont capté son intérêt et qu'il souhaitait visiter plus tard, à la manière d'une liste de souhaits;
- **Un système de récompense par badges**: une récompense, comme un badge, pourrait être obtenue dépendamment du nombre d'œuvres visitées dans une catégorie ou dans un arrondissement, ou tout simplement du nombre total d'œuvres collectionnées dans l'application. Ce système vise à rendre l'utilisation de MONA plus ludique, tout en fidélisant les utilisateurs qui cherchent à accumuler le plus de récompenses possibles;
- **Être bilingue**: l'application vise le public montréalais et devrait donc être en français et en anglais;
- **Authentification**: pour des raisons de sécurité, l'utilisateur devrait pouvoir créer un compte qui permet de chiffrer et d'authentifier les requêtes effectuées vers le serveur Web, et de récupérer ses données. Un compte pour chaque utilisateur faciliterait également le droit à l'oubli des données en pouvant identifier quelles données appartiennent à quel utilisateur;

- **Notifications**: l'application devrait pouvoir envoyer des notifications à un utilisateur lorsqu'il est positionné à proximité d'une œuvre ou pour lui rappeler d'ouvrir l'application;
- **Réseaux sociaux**: l'application devrait permettre de partager la photographie d'une œuvre sur les réseaux sociaux. Il serait idéal de pouvoir récupérer les informations comme les commentaires liés à ce partage de photographie.

Nous avons vu que l'objectif principal de MONA est d'inciter l'utilisateur à partir à la découverte des œuvres d'art public de Montréal. L'application devrait être organisée en 4 sections : une section « Œuvre du jour », une section « Œuvres » (les listes), une section « Carte » et une section « Collection ». L'application devrait permettre aux utilisateurs d'ajouter une œuvre à une liste de souhaits, de la prendre en photo, de la noter, de la commenter et de la partager sur les réseaux sociaux. Cette application requiert aussi que l'on implémente une gestion de l'authentification et des méthodes de collectes des données des utilisateurs pour la recherche universitaire. Elle devrait être également capable de récompenser les utilisateurs avec des badges. Idéalement, l'application doit pouvoir générer des notifications et être bilingue.

CHAPITRE 3

Le serveur Web

Nous allons d'abord expliquer la conception et l'implémentation de l'ancienne version du serveur Web dont Vincent Beauregard était le responsable [48]. Ensuite, nous procéderons à l'identification des problèmes de l'ancienne version. Puis, nous détaillerons les objectifs qui ont été définis à partir des problèmes identifiés. Enfin, nous expliquerons la conception et l'implémentation de la nouvelle version du serveur Web développée par Abdelhakim Qbaich [49].

3.1 Ancienne version

Le serveur utilisé est la propriété du département d'informatique et de recherche opérationnelle (DIRO) de l'Université de Montréal et utilisent les technologies PHP et MariaDB (MySQL). Ce choix s'explique par le fait que ces technologies sont déjà disponibles sur les serveurs du DIRO.

3.1.1 Importation des données

Une requête « createJson » permet de générer un fichier JSON qui contient l'ensemble des données disponibles sur les œuvres d'art. Lors de l'exécution de cette requête, les données sur les œuvres d'art sont d'abord extraites des diverses sources de données ouvertes telles que l'ensemble de données sur les œuvres d'art publiques de Montréal [23] et l'ensemble de données sur les murales de Montréal. [50]. Puis ces données suivent un processus de normalisation et sont fusionnées pour générer un nouveau fichier JSON.

3.1.2 Interface de programmation

Les requêtes HTTP vers l'API s'effectuent via l'URL http://www-etud.iro.umontreal.ca/~beaurevg/ift3150/server/. Elles sont de type GET ou POST et

s'exécutent à l'aide des paramètres tels que, par exemple, « request », « IDOeuvre » ou « comment ». Par exemple, si l'on souhaite ajouter un commentaire à propos d'une œuvre, on peut former cette requête de la façon suivante :
http://www-etud.iro.umontreal.ca/~beaurevg/ift3150/server/?request=addComment&username=nakwenda&password=123456&IDOeuvre=45&comment=bonjour

Les requêtes disponibles pour cette API sont résumées dans la liste suivante :

1. « createJson »: cette requête déclenche l'extraction et le « parsing » des données ouvertes d'art public de la ville de Montréal et crée un fichier unique fichier JSON nommé « artpubMural.json » qui contient toutes les données sur les œuvres d'art public;
2. « loadJson1 »: cette requête retourne le fichier « artpubMural.json » contenu sur le serveur. Un exemple de donnée JSON pour une œuvre d'art issue du fichier « artpubMural.json » est présenté à l'annexe I;
3. « createUser »: les paramètres de la requête sont « username » et « password ». Elle permet d'ajouter un nouvel utilisateur à la base de données. Elle retourne une réponse JSON avec les champs « successful » et « erreur ». Si la requête de création a été effectuée avec succès, alors le champ « successful » a la valeur 1 et le champ « erreur » est nul. Si elle échoue, alors le champ « successful » a la valeur 0 et le champ « erreur » contient la raison de l'erreur, par exemple « username already exists »;
4. « logUser »: les paramètres de la requête sont « username » et « password ». Elle permet à un utilisateur de se connecter. Elle retourne une réponse JSON avec les champs « successful » et « erreur ». Si la connexion a réussi, le champ « successful » a la valeur 1 et le champ « erreur » est nul. Si elle échoue, par exemple parce qu'un nom utilisateur existe déjà dans la base de données, alors le champ « succesful » vaut 0 et le champ « erreur » contient la raison de l'échec de la requête, par exemple « password is wrong »;

5. « addComment »: les paramètres de la requête « username », « password », « IDOeuvre », et « comment ». Elle permet à un utilisateur d'ajouter un commentaire à une œuvre dans la base de données du serveur;
6. « addNote »: les paramètres de la requête « username », « password », « IDOeuvre », et « note ». Elle permet à un utilisateur d'ajouter une note à une œuvre dans la base de données du serveur;
7. « getComment »: les paramètres de la requête « username », « password » et « IDOeuvre ». Elle retourne le commentaire le plus récent produit par un utilisateur à propos d'une œuvre;
8. « getNote »: les paramètres de la requête « username », « password » et « IDOeuvre ». Elle retourne la note la plus récente produite par un utilisateur à propos d'une œuvre;
9. « getCritic »: les paramètres de la requête « username », « password » et « IDOeuvre ». Elle retourne la critique la plus récente produite par un utilisateur à propos d'une œuvre sous la forme d'un objet JSON contenant les champs « IDOeuvre », « comment », et « note »;
10. « getUserData »: les paramètres de la requête « username » sont « password ». Elle retourne toutes les critiques les plus récentes qu'un utilisateur a pu produire pour chaque œuvre sous la forme d'un tableau JSON où chaque élément est représenté comme les objets JSON retournés par la fonction « getCritic »;
11. « addPicture »: les paramètres de la requête « username », « password », « IDOeuvre » et le fichier image. Elle permet à un utilisateur d'ajouter la photo d'une œuvre. Les photos sont conservées dans un dossier « uploads » et le lien vers le fichier est conservé dans la base de données du serveur. L'implémentation de la procédure de traitement de cette requête y est décrite à l'annexe II.

Une base de données relationnelle qui suit le schéma de données présenté à la figure 3.1 ci-après est utilisée pour la sauvegarde des photos et des critiques des œuvres. Le script de création des tables est présenté à l'annexe III.

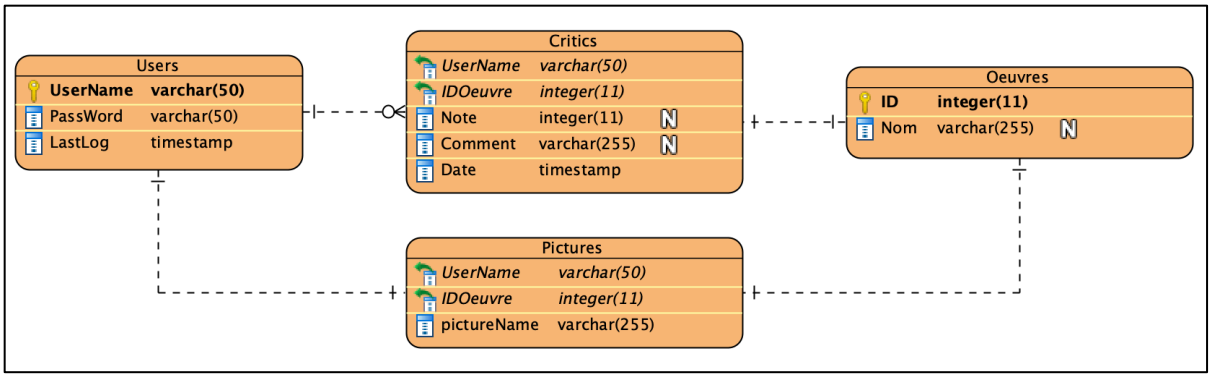


Figure 3.1: Schéma de la base de données (ancienne version)

Cette base de données permet de conserver les noms et mots de passe des utilisateurs dans une table « Users » et les œuvres du fichier JSON dans une table « Œuvres ». La table « Critics » est une table de liaison entre les tables « Users » et « Œuvres » qui conserve une note et un commentaire pour chaque couple utilisateur-œuvre. La table « Pictures » est aussi une table de liaison qui permet de sauvegarder le nom d'une photographie pour chaque couple utilisateur-œuvre. Toutes les photographies téléversées sur le serveur Web sont sauvegardées sur disque dans un dossier « uploads », c'est pourquoi il est nécessaire de conserver le nom de l'image pour retrouver une photographie dans ce dossier.

3.2 Identification des problèmes

L'ancienne version du serveur Web fonctionne bien pour récupérer les données sur les œuvres et pour téléverser les photos, notes et commentaires. Mais cette version souffre malgré tout de plusieurs problèmes:

- Un manque de fiabilité des traductions des matériaux et des techniques. En effet, certains matériaux ou certaines techniques en français pouvaient avoir de multiples traductions en anglais. Un important travail de nettoyage des données est nécessaire. C'est un des problèmes des données ouvertes: celles-ci peuvent être potentiellement de mauvaises qualités;
- Pas d'utilisation du protocole HTTPS ce qui ne permet pas de chiffrer les communications avec le serveur. Cela est très problématique en cas d'attaque du Man in the middle;

- Mots de passe non cryptés dans la base de données. Les mots de passe sont conservés en clair au lieu de procéder à un hachage et salage des mots de passe pour préserver la sécurité des mots de passe;
- Pas de système d'authentification par jeton;
- Il n'est pas possible pour les chercheurs de l'Université de Montréal de récupérer facilement les données sur les critiques et les photos des œuvres via une interface graphique d'administration;
- Pas de protection contre les injections SQL et les attaques par déni de service.

3.3 Définition des objectifs

À partir des problèmes cités, nous pouvons établir les objectifs suivants pour la nouvelle version du serveur Web :

- Améliorer la cohérence des données fournies par l'API, en particulier en ce qui concerne les techniques et les matériaux;
- Sécuriser les requêtes client-serveur avec HTTPS et un système d'authentification par jeton;
- Sécuriser les mots de passe de la base de données avec une fonction de hachage;
- Implémenter un système de gestion de l'authentification par jeton;
- Fournir une interface d'administration permettant un accès facile aux données sur les œuvres d'art comme les photos, les notes et les commentaires des œuvres;
- Se protéger des injections SQL et des attaques par déni de service.

3.4 Nouvelle version

La nouvelle version du serveur Web utilise les mêmes technologies que l'ancienne version, à savoir MariaDB (MySQL) et PHP. En revanche, la nouvelle version de l'API repose sur un des cadriciels les plus couramment utilisés en PHP dans le domaine des applications Web: le cadriciel « Laravel ». Ce choix s'explique par le fait qu'il permet de concevoir un serveur Web rapidement en suivant une architecture MVC.

3.4.1 Importation des données

Dans l'ancienne version du serveur Web, les données sont extraites des différentes sources de données ouvertes, puis normalisées pour générer immédiatement un fichier JSON à chaque requête « createJSON ». La requête « loadJson1 » peut servir ensuite à obtenir le fichier JSON hébergée sur le serveur.

Dans la nouvelle version, les données sont extraites des différentes sources de données, puis elles sont normalisées à l'aide d'expressions régulières avant d'être sauvegardées dans la base de données. La génération du fichier JSON est effectuée dynamiquement à chaque demande d'accès aux œuvres d'art à l'aide de la base de données. Ce fichier est ensuite mis en cache, offrant ainsi un accès plus rapide aux requêtes suivantes.

La figure 3.2 présentée ci-après montre l'évolution de la base de données comparativement au schéma de l'ancienne version du serveur présentée à la figure 3.1. On peut constater dans cette figure que la base de données est désormais beaucoup plus complète et flexible pour la gestion des œuvres et des utilisateurs.

Les scripts de création des tables de la base de données ont également évolué. Plutôt que d'utiliser un unique script SQL de création de tables comme le montre l'annexe III, les tables sont désormais créées en PHP avec les méthodes de la classe « Schema ». En effet, le cadriciel « Laravel » fournit une API unifiée de manipulation de tables de base de données, nommée « Schema », supportant différents systèmes de gestion de bases de données, comme « MySQL » ou « SQLite ». L'annexe IV montre comme exemple la création de la table d'œuvres d'art avec la classe « Schema ».

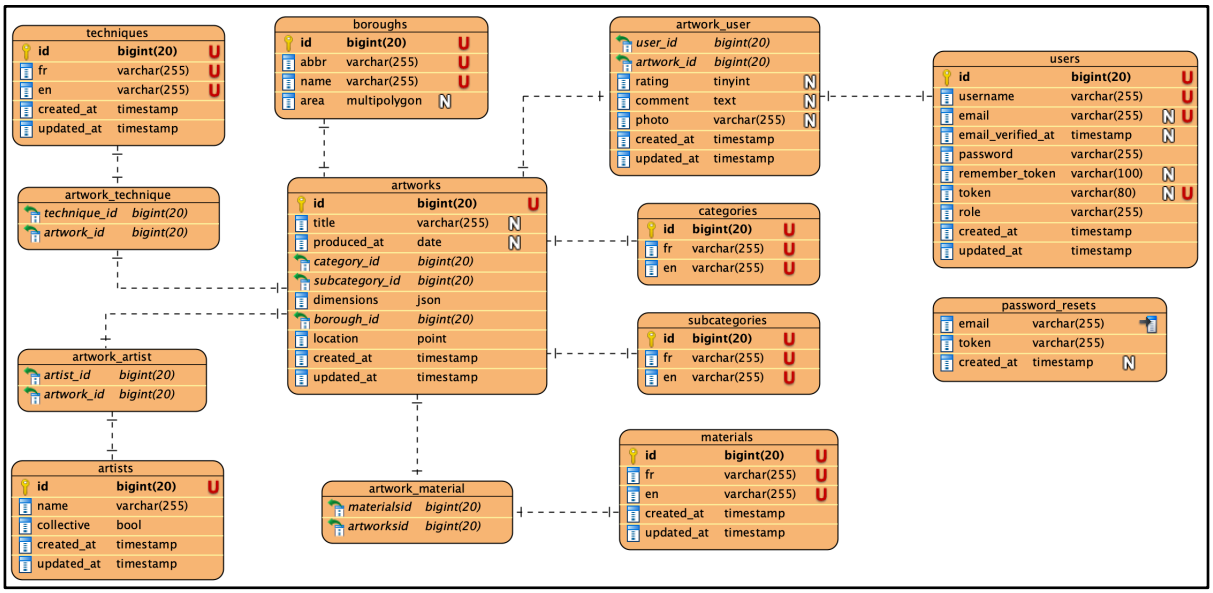


Figure 3.2: Schéma de la base de données (nouvelle version)

3.4.2 Interface de programmation

La nouvelle version du serveur Web est accessible à partir de « picasso.iro.umontreal.ca ». Alors que « www-ens.iro.umontreal.ca » utilisait seulement le protocole HTTP pour ses communications avec le client, cette nouvelle version utilise désormais le protocole HTTPS pour chiffrer ses communications.

La nouvelle interface de programmation se conforme au style d'architecture logicielle REST qui spécifie un ensemble de contraintes pour une API. Les requêtes et réponses sont chiffrées à l'aide du protocole HTTPS, et encodent le type MIME « application/json » dans leur entête. De plus, leur structure est conforme à la spécification JSON API. Le tableau 3.1 ci-dessous liste les chemins d'accès aux différentes requêtes disponibles avec la nouvelle interface de programmation.

Méthode	**URI**	**Middleware**
GET|HEAD	api/artists	api, json
GET|HEAD	api/artists/{artist}	api, json
GET|HEAD	api/artworks	api, json
GET|HEAD	api/artworks/{artwork}	api, json
POST	api/login	api, json, guest
POST	api/logout	api, json, auth:api
POST	api/register	api, json, guest
GET|HEAD	api/user	api, json, auth:api
GET|HEAD	api/user/artworks	api, json, auth:api
POST	api/user/artworks	api, json, auth:api
GET|HEAD	api/user/artworks/{artwork}	api, json, auth:api
PUT|PATCH	api/user/artworks/{artwork}	api, json, auth:api
DELETE	api/user/artworks/{artwork}	api, json, auth:api

Tableau 3.1: Liste des chemins d'accès aux requêtes supportées par l'API (nouvelle version)

Toute requête qui utilise le « middleware » « auth:api », du cadriciel « Laravel », nécessite l'authentification préalable de l'usager pour être exécutée. En effet, « auth:api » est responsable de la validation du jeton d'accès d'un utilisateur. L'algorithme 3.1 ci-après présente quelques exemples de requêtes vers la nouvelle API.

```
curl -d "username=nenda&password=machin" \
https://picasso.iro.umontreal.ca/~mona/api/login

{
  "token": "WVeh1vn2x055qj9o083Z7IYq4M68uZ2Q1vVHxsS5lGqr46S3Li7pXDQDGh1h "
}

curl https://picasso.iro.umontreal.ca/~mona/api/artists/25

{
  "data": {
             "id": 25,
             "name": "Mosaika",
             "collective": false
           }
}


curl -G -d
"api_token=WVeh1vn2x055qj9o083Z7IYq4M68uZ2Q1vVHxsS5lGqr46S3Li7pXDQDGh1h " \
https://picasso.iro.umontreal.ca/~mona/api/user

{
  "id": 18,
  "username": "nenda",
  "email": "machin@truc.ca",
  "email_verified_at": "2019-10-10 00:01:00",
  "role": "admin",
  "created_at": "2019-10-10 00:00:00",
  "updated_at": "2019-10-10 14:30:00"
}
```

Algorithme 3.1: Exemples de requêtes de l'API (nouvelle version)

L'utilisation du protocole HTTPS permet de sécuriser l'authentification des clients contre, par exemple, une attaque Man-in-the-Middle. La fonction de hachage « bcrypt » est utilisée pour hacher les mots de passe reçus lors de l'inscription ou de la connexion d'un utilisateur. Seules les valeurs de hachage des mots de passe sont conservées dans la base de données. Le choix de la fonction « brcrypt » est justifié par le fait que cette fonction est efficace pour se protéger des attaques par force brute ou par tables en arc-en-ciel.

Un client peut se connecter au serveur via une requête de connexion « api/login » qui contient ses identifiants. Un jeton (« token ») d'authentification pour la session est généré aléatoirement, puis est haché avec la fonction de hachage SHA-256 avant d'être sauvegardé dans la base de données. Comme la génération du jeton est aléatoire, la fonction SHA-256 est considérée comme amplement sécuritaire contre des attaques par force brute ou par tables en

arc-en-ciel. En réponse à sa requête de connexion, le client reçoit un jeton d'authentification (« api_token ») qu'il peut passer en paramètre d'une requête nécessitant une authentification. Un système de permissions a également été conçu via la création de rôles « admin » et « user » pour les utilisateurs. Seuls les utilisateurs ayant pour rôle « admin » peuvent accéder à l'interface d'administration décrite à la section 3.4.3.

Les photos des œuvres produites par les usagers sont sauvegardées sur disque sur le serveur (similairement à l'ancienne version). L'URL qui permet d'accéder au fichier image est conservée dans la base de données. L'annexe V permet d'illustrer les changements dans l'algorithme de stockage des données comparativement à l'algorithme de stockage de la photo de l'ancien serveur de l'annexe II.

3.4.3 Interface graphique d'administration

Un utilisateur doit avoir un rôle « admin » pour accéder à l'interface d'administration. Ce rôle est attribuable à un utilisateur en modifiant directement la base de données avec une requête SQL, ou plus facilement avec la commande décrite par l'algorithme 3.2 ci-dessous.

```
php artisan user:role <username> admin # Remplacer <username> par un nom.
```

Algorithme 3.2: Attribuer un rôle d'administrateur à un utilisateur (nouvelle version)

L'interface graphique d'administration est divisée en deux parties:

1. Une section « Œuvres d'art » qui est une liste de toutes les œuvres d'art comme le montre la figure 3.3. Cette section permet notamment à un administrateur d'accéder aux données telles que les photos, les statistiques sur les notes et les commentaires récoltés chez tous les utilisateurs pour une œuvre comme le montre la figure 3.4;

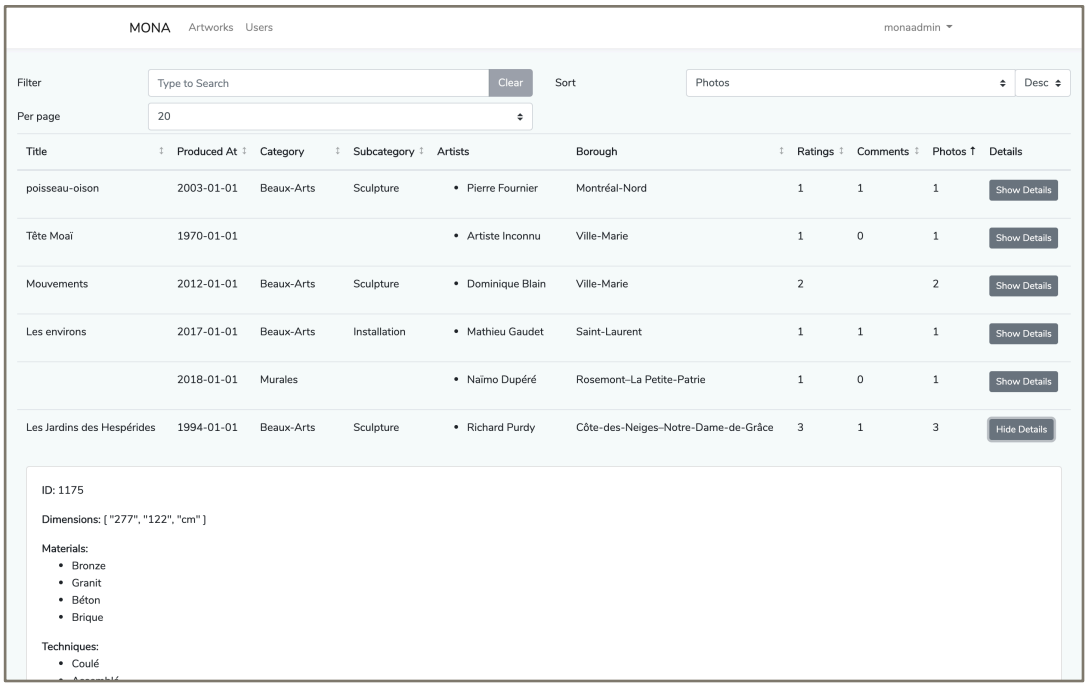


Figure 3.3: La section « Œuvres d'art » de l'interface d'administration (nouvelle version)

Figure 3.4: Détails des données récupérées pour une œuvre d'art dans la section « Œuvres d'art » de l'interface d'administration (nouvelle version)

2. Une section « Utilisateurs » (anonymisés) comme le montre la figure 3.5. Cette section permet à un administrateur d'accéder aux données telles que les photos, les notes et les commentaires pour toutes les œuvres collectionnées par un utilisateur comme le montre la figure 3.6.

Ces deux sections permettent de filtrer et de trier les données, et d'afficher un nombre paramétrable de données par page.

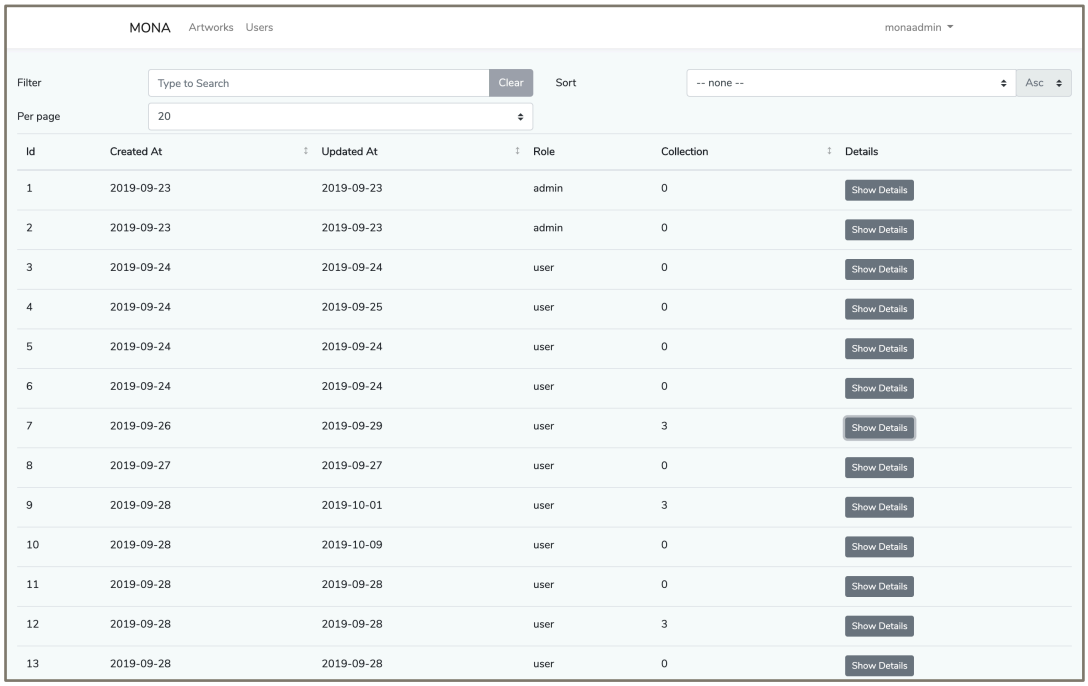


Figure 3.5: La section « Utilisateurs » de l'interface d'administration (nouvelle version)

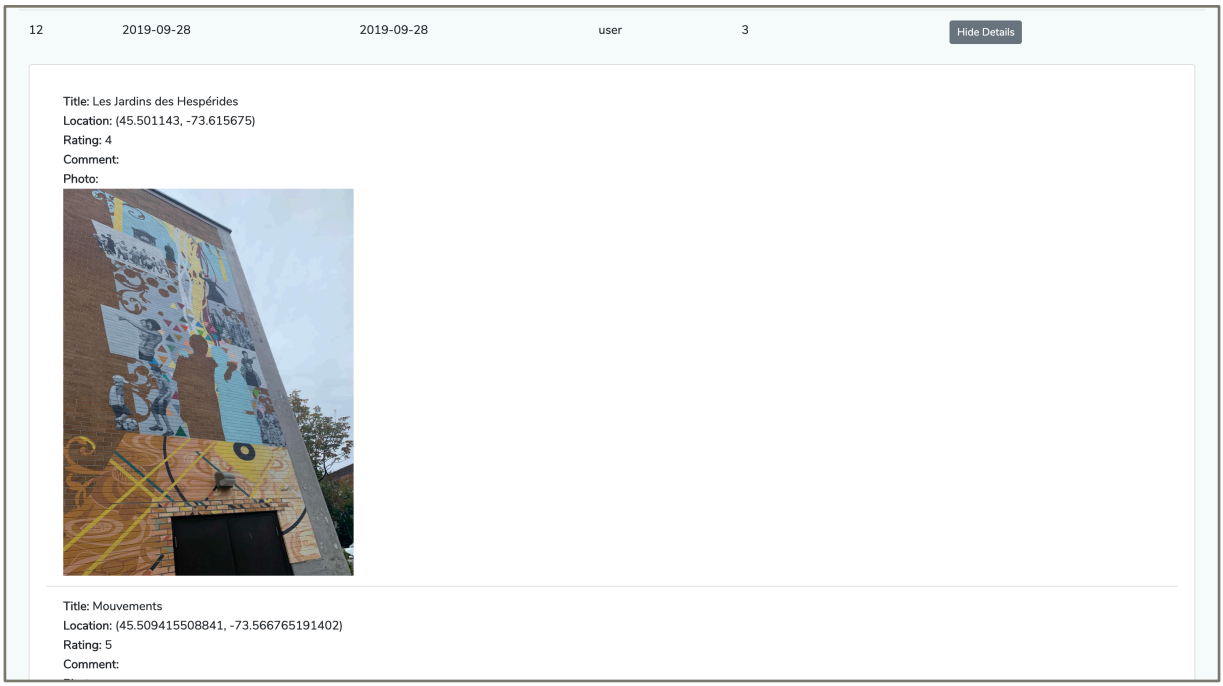


Figure 3.6: Détails des données récupérées sur toutes les œuvres par un utilisateur dans la section « Utilisateurs » de l'interface d'administration (nouvelle version)

CHAPITRE 4

L'application

Nous allons d'abord expliquer la conception et l'implémentation de l'ancienne version de l'application [51]. Puis, nous procéderons à l'identification des problèmes de l'ancienne version. Ensuite, nous détaillerons les objectifs qui ont été définis à partir des problèmes identifiés. Enfin, nous expliquerons la conception et l'implémentation de la nouvelle version de l'application.

4.1 Ancienne version

Dans ce sous-chapitre, le fonctionnement global de l'application dans sa version initiale est présenté. Ceci permet de pouvoir mieux comprendre et apprécier les modifications apportées lors de ce projet de recherche appliquée de maîtrise. Cette version de l'application MONA utilise l'API décrite au sous-chapitre 3.1.

4.1.1 Le modèle

Suivant une architecture MVC, un ensemble de classes regroupées dans un dossier nommé « Model » ont été conçues. Les propriétés et les relations de ces classes sont décrites au diagramme de classes de la figure 4.1.

Ce modèle permet d'accéder à partir d'une œuvre d'art (c.-à-d. une instance de classe « Artwork ») à l'ensemble de ses propriétés, comme ses matériaux ou son arrondissement. On remarque qu'il est également possible d'accéder facilement à la liste des œuvres qu'une instance de classe contient, comme « Category » ou « Technique ».

Afin que l'application puisse être bilingue, les classes telles que « Material » ou « Technique » disposent d'une propriété « names » de type dictionnaire. Par exemple, une instance de « Category » pourrait avoir une propriété « names » de la forme « [.en : "Beaux-Arts", .fr :

"Fine Arts"] » où « .en » et « .fr » sont des valeurs d'un type « enum » « Language » pour désigner l'anglais et le français. De cette manière, cette propriété facilite l'obtention et l'affichage de chaînes de caractères dans la langue de l'usager.

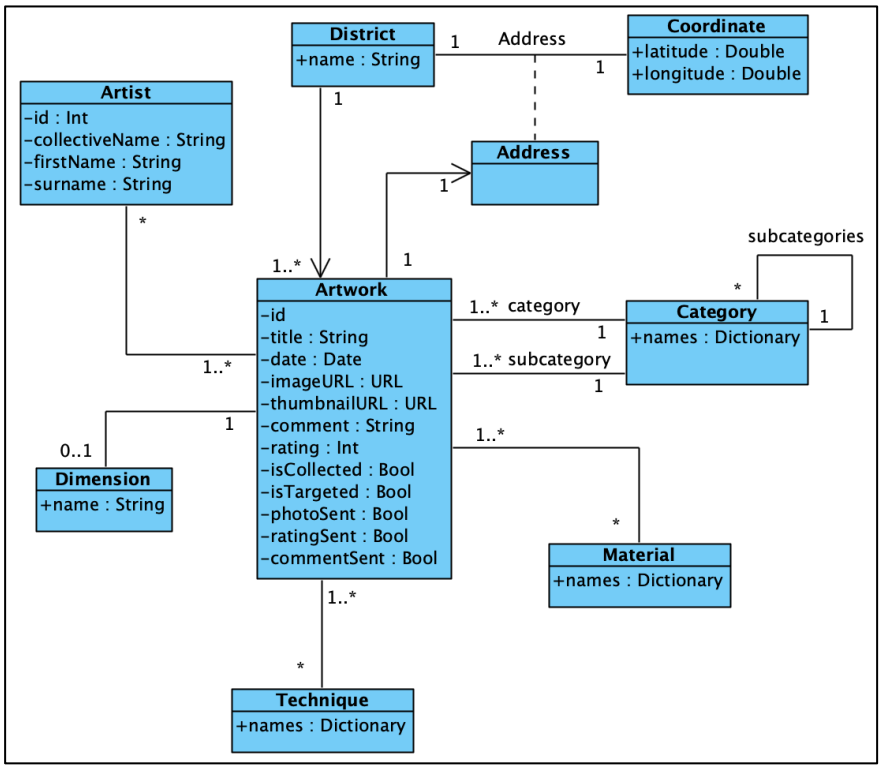


Figure 4.1: Diagramme de classes du modèle (ancienne version)

4.1.2 Gestion des données

Toutes les classes décrites à la figure 4.1 implémentent un protocole nommé « NSCoding » qui permet à leurs instances d'être encodées et décodées dans le but d'être archivées avec la classe « NSKeyedArchiver » du cadriciel « Foundation ». Cette dernière est un encodeur qui permet de stocker un graphe d'objets dans un fichier à une URL donné. L'archivage d'instances de ces classes permet ainsi d'opérer la persistance des données de l'application. L'annexe VI montre comme exemple la façon dont la classe « Coordinate » et le protocole « NSCoding » sont implémentés.

Dans l'ancienne version de l'application, la gestion de la persistance des données incombe à une unique classe nommée « DataManager » qui permet d'interagir plus facilement avec « NSKeyedArchiver ». Comme le décrit l'annexe VII, la classe « DataManager » est un singleton qui, lors de son initialisation, crée un ensemble de dossiers et de fichiers pour les œuvres d'art, les badges, ou encore les images (c.-à-d. les photos des œuvres d'art). La classe « DataManager » contient aussi des procédures de sauvegarde et de chargement des données du modèle. Par exemple, l'algorithme 4.1 est un algorithme d'archivage qui permet de faire persister une liste d'œuvres d'art dans un fichier à un contenue dans « artworksFile.path ».

```
    func saveArtworks(artworks: [Artwork]) -> Bool {
        let isSuccessfulSave = NSKeyedArchiver.archiveRootObject(artworks,
toFile: artworksFile.path)
        if isSuccessfulSave {
            log.info("Artworks successfully saved.")
        } else {
            log.error("Failed to save artworks data. Maybe two files have the
same name.")
        }
        return isSuccessfulSave
    }
```

Algorithme 4.1: Archivage des œuvres (ancienne version)

Par ailleurs, « DataManager » contient une liste de toutes les instances essentielles au fonctionnement de l'application, comme les instances de « Artwork », de « Category », de « District », etc., après que celles-ci aient été chargées en mémoire.

4.1.3 Gestion des photos

Chaque photo liée à une instance de « Artwork » est sauvegardée au format PNG à une URL générée à partir de la propriété « id » d'un « Artwork ». Le singleton « DataManager » contient aussi les procédures de sauvegarde et de chargement des photos des d'œuvres d'art prises par l'utilisateur.

4.1.4 Gestion des opérations réseau

Un « package » nommé « HermesNetwork » (https://medium.com/@danielemargutti/network-layers-in-swift-7fc5628ff789) a été choisi pour effectuer les requêtes HTTP, le « parsing » des réponses JSON du serveur, et pour permettre une gestion facilitée des erreurs.

Pour créer une requête HTTP avec « HermesNetwork », il faut créer une classe qui hérite de « JSONOperation<T> » et définit une méthode d'initialisation spécifiant le type de méthode HTTP, les paramètres et la procédure à exécuter lors de la réception de la réponse au format JSON. Par exemple, « GetArtworksOperation » est le nom de la classe permettant de créer une requête HTTP de type « GET » destinée à l'API du serveur Web et dont l'exécution permet d'obtenir le fichier « artpubMural.json » hébergé sur le serveur. Son implémentation y est présentée à l'algorithme 4.2. À la réception du fichier au format JSON, celui-ci est traité à l'aide de la méthode statique « load(list : JSONArray) » de la classe « Artwork ». Cette méthode statique permet d'obtenir une liste d'instances de « Artwork » qui sera archivée à l'aide des procédures de sauvegarde de la classe « DataManager ».

```
import SwiftyJSON
import HermesNetwork

class GetArtworksOperation : JSONOperation<[Artwork]> {

    override init() {
        super.init()
        self.request = Request(method: .get, endpoint: "/server/")
        self.request?.fields = ["request":"loadJson1"]
        self.onParseResponse = { json in
            return Artwork.load(list: json.arrayValue)
        }
    }

}
```

Algorithme 4.2: Classe « GetArtworksOperation » pour créer une requête de téléchargement du fichier « artpubMural.json » (ancienne version)

Les autres requêtes HTTP qui ont été implémentées de manière similaire sont les classes « AddCommentOperation », « AddNoteOperation », « AddPhotoOperation » (voir Annexe VIII), « CreateUserOperation » et « LogInUserOperation ».

4.1.5 La section « Œuvres »

La section « Œuvres » donne le choix à l'utilisateur d'afficher la liste des œuvres d'art, la liste des artistes, la liste des catégories ou la liste des arrondissements avec un bouton de sélection de liste situé dans la barre de navigation, comme le montre la figure 4.2 ci-dessous.

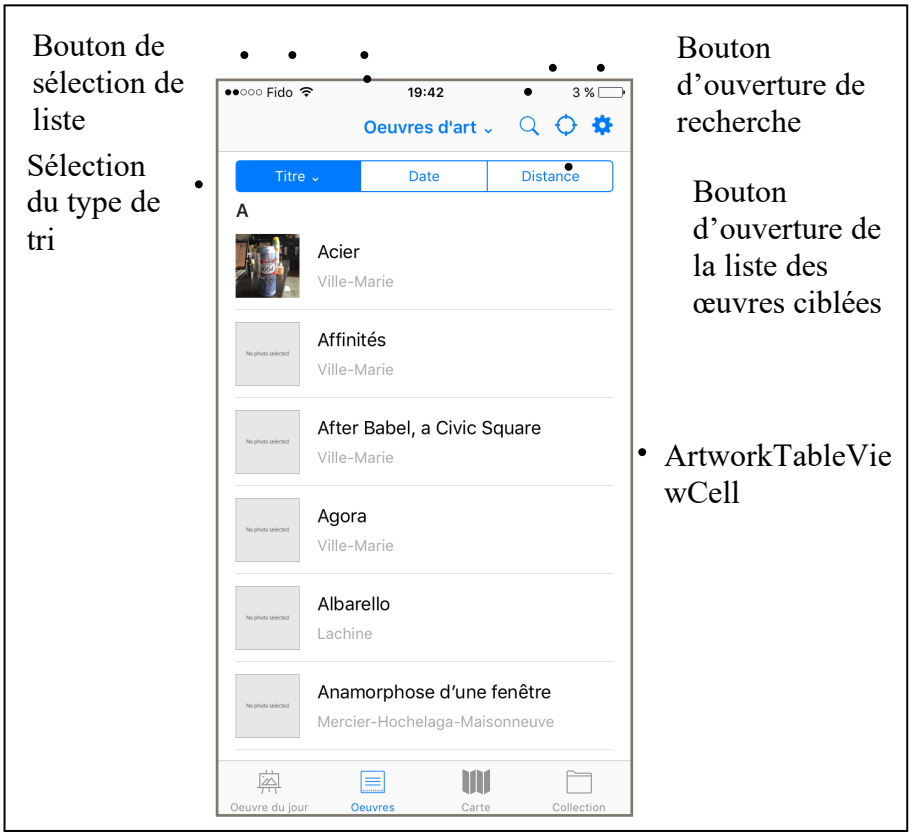


Figure 4.2: Liste des œuvres d'art (ancienne version)

La liste d'œuvres d'art peut être triée de manière ascendante ou descendante selon le titre, la date de production de l'œuvre ou selon la distance entre la position de l'usager et la position

de l'œuvre. En revanche, la liste de catégories, d'artistes ou d'arrondissements ne permet pas de tri et est triée par défaut de manières alphabétique.

Pour faciliter le parcours de cette liste, parfois très longue, d'œuvres ou d'arrondissements, un index a été ajouté à la droite de la liste afin de permettre le saut de section en section, comme le montre la figure 4.3 ci-dessous. Un « package » appelé « MYTableViewIndex » a été utilisé pour mettre en place cet index.

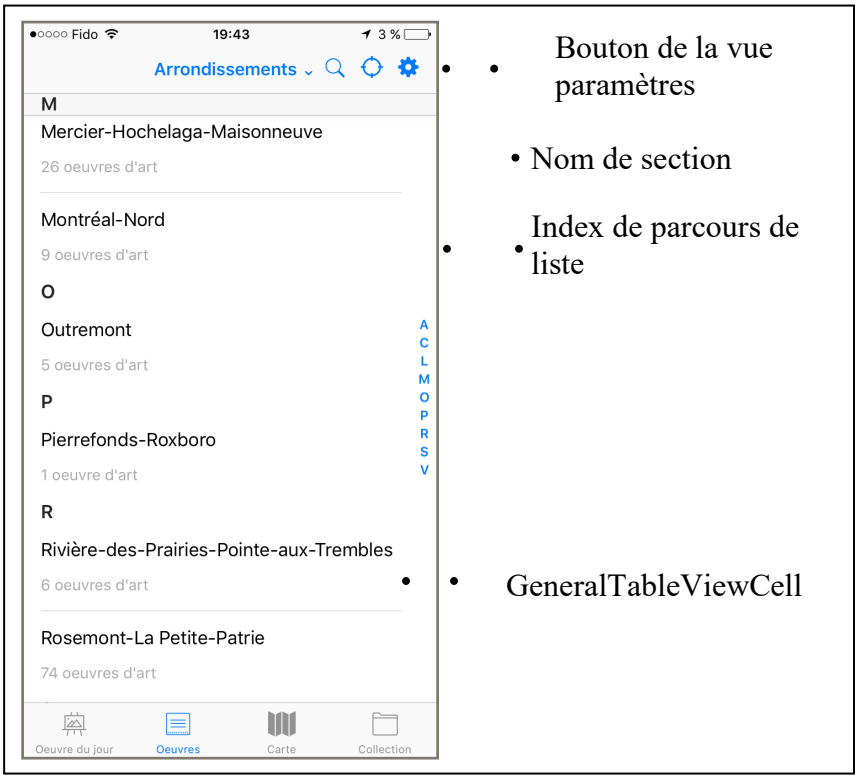


Figure 4.3: Liste des arrondissements (ancienne version)

Deux prototypes d'éléments de liste ont été conçus pour être réutilisés dans plusieurs vues de l'application, par exemple dans l'affichage des résultats d'une recherche ou dans la liste d'œuvres:

1. « ArtworkTableViewCell »: cette classe permet d'instancier un élément de liste qui affiche une vignette avec un titre et un sous-titre. Ce prototype est utilisé lors de

l'affichage d'une œuvre d'art dans une liste, avec comme titre: le titre de l'œuvre et comme sous-titre, l'arrondissement dans lequel est localisée cette œuvre;

2. « GeneralTableViewCell »: cette classe permet d'instancier un élément de liste qui affiche un titre et un sous-titre. Ce prototype est utilisé dans l'affichage d'un artiste, d'une catégorie ou d'un arrondissement, avec comme titre : le nom de l'artiste, de la catégorie ou de l'arrondissement, et comme sous-titre le nombre d'œuvres d'art liées à cet artiste, cette catégorie ou cet arrondissement.

4.1.6 Fiche d'une œuvre

Un clic sur un « ArtworkTableViewCell » ouvre une vue de type « ArtworkDetailsViewController ». Cette classe contrôle l'affichage de la fiche d'une œuvre d'art, comme la description de l'œuvre et la localisation de l'œuvre sur une mini-carte, et les actions possibles à partir de cette fiche, comme la prise de photographie, l'ajout à la liste de souhaits (aussi nommé « liste des œuvres ciblées), la notation et le commentaire d'une œuvre comme le montre l'annexe IX. Après la prise d'une photographie, celle-ci est sauvegardée au sein de l'application à l'aide de la classe « DataManager ». Chaque édition d'une note ou d'un commentaire sur une œuvre est collectée par le serveur Web à l'aide des requêtes HTTP « AddCommentOperation » et « AddRatingOperation ».

4.1.7 La section « Carte »

La section « Carte », présentée à la figure 4.4, utilise le cadriciel « Core Location » pour localiser l'utilisateur sur une carte ainsi que pour afficher toutes les œuvres d'art à leurs coordonnées géographiques sous la forme d'épingles. Un clic sur une épingle ouvre une annotation qui permet d'accéder à la fiche d'une œuvre (voir section 4.1.6). Les œuvres d'art sont triables sur la carte selon qu'elles soient collectionnées ou ciblées (c.-à-d. ajoutées à la liste de souhaits).

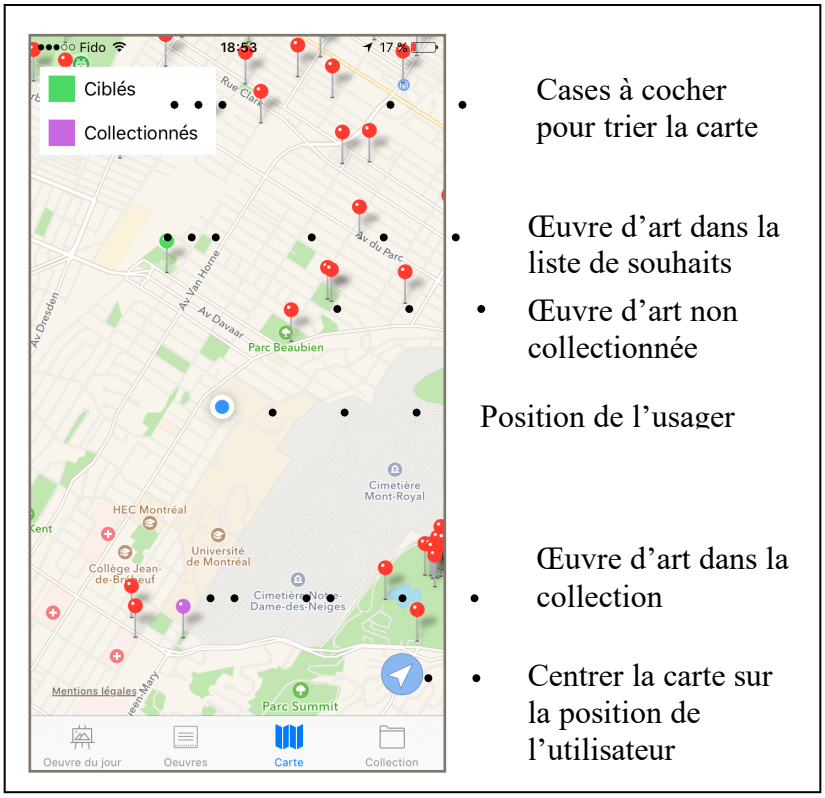


Figure 4.4: La section « Carte » (ancienne version)

4.1.8 La recherche

On peut rechercher une œuvre d'art en cliquant sur le bouton ressemblant à une loupe comme le montre la figure 4.2, ce qui permet d'ouvrir la vue de recherche d'œuvres d'art comme le montre la figure 4.5.

Une classe « SearchResultsController » est responsable de l'affichage des résultats de la recherche correspondants au patron de texte contenu dans la barre de recherche. Les résultats sont organisés en quatre sections: 1) « Œuvres d'art », 2) « Artistes », 3) « Arrondissements » et 4) « Catégorie » et sont actualisés à chaque édition des termes de la recherche. Pour limiter la longueur de la liste des résultats, les sections ne peuvent contenir plus de trois résultats. On doit cliquer sur un nom de section pour lister l'ensemble des résultats contenu par cette section.

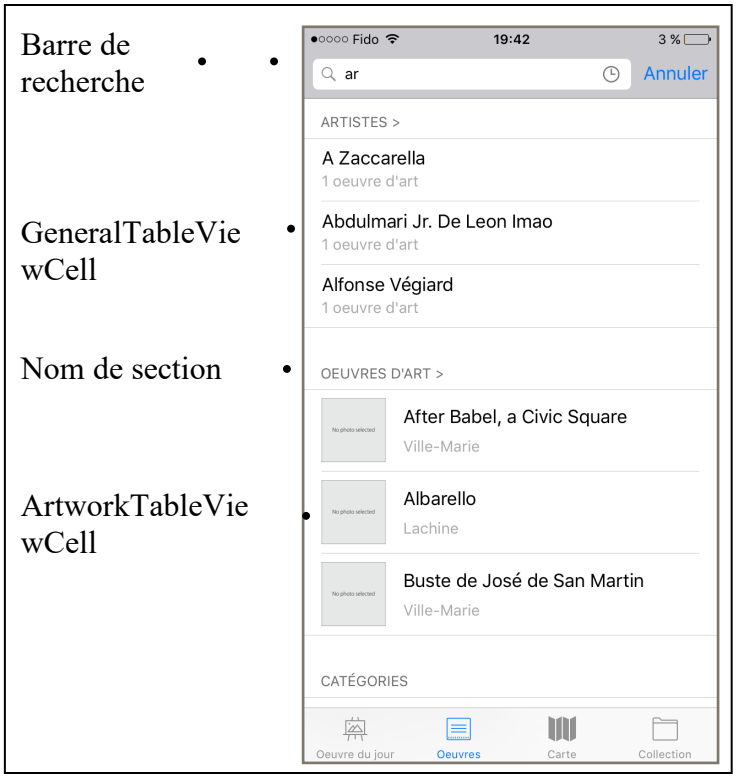


Figure 4.5: La recherche (ancienne version)

4.1.9 Gestion des langues

L'application devant être disponible en français et en anglais, toutes les chaînes de caractères comme les messages d'erreurs ou les noms des listes devaient être traduits en français et en anglais.

Pour résoudre ce problème, il suffit de créer un fichier « .strings » (voir algorithme 4.3 et 4.4 ci-dessous) pour chaque langue souhaitée. En fonction de la langue paramétrée par l'usager sur son téléphone, la classe « NSLocalizedString » issue du cadriciel « Foundation » retourne alors la chaîne de caractères correspondante à la langue de l'usager comme le montre l'algorithme 4.5. Enfin, la propriété « names » décrite à la section 4.1.1 permet d'obtenir facilement la version française ou anglaise du nom d'une catégorie, d'une technique ou encore d'un matériau dépendamment de la langue de l'usager.

```
"hintUsernameLabel_UsernameChoiceViewController" = "Choose a username.";
"usernameTextField_UsernameChoiceViewController" = "Alphanumeric and allowed
from %1$d to %2$d characters.";
"validButton_UsernameChoiceViewController" = "Done";
"usernameAlreadyExists_UsernameChoiceViewController" = "Username already exists.
Please choose another username.";
```

Algorithme 4.3: Extrait du fichier « .strings » anglais

```
"hintUsernameLabel_UsernameChoiceViewController" = "Choisissez un pseudo avant
de continuer.";
"usernameTextField_UsernameChoiceViewController" = "Alphanumérique et de %1$d à
%2$d caractères.";
"validButton_UsernameChoiceViewController" = "Valider";
"usernameAlreadyExists_UsernameChoiceViewController" = "Le nom d'utilisateur
existe déjà. Veuillez choisir un autre nom d'utilisateur.";
```

Algorithme 4.4: Extrait du fichier « .strings » français

```
let hintUsernameLabelText : String =
NSLocalizedString("hintUsernameLabel_UsernameChoiceViewController", comment: "")
```

Algorithme 4.5: Utilisation de « NSLocalizedString »

4.1.10 Autres besoins

La liste d'œuvres d'art ciblées par l'usager a été créée sur le modèle de la liste d'œuvres décrite à la figure 4.2 et se contente d'afficher seulement les œuvres dont la propriété « isTargeted » d'une instance de « Artwork » à la valeur « true ». Cette liste d'œuvres d'art ciblées est accessible avec le bouton en forme de cible visible à la figure 4.2.

La section « Collection », que l'on peut voir à la figure 4.6, fonctionne similairement à la liste d'œuvres d'art ciblées : elle utilise le modèle de liste d'œuvres de la figure 4.2 et affiche

uniquement les œuvres (instances de « Artwork ») dont la propriété « isCollected » possède la valeur « true ».

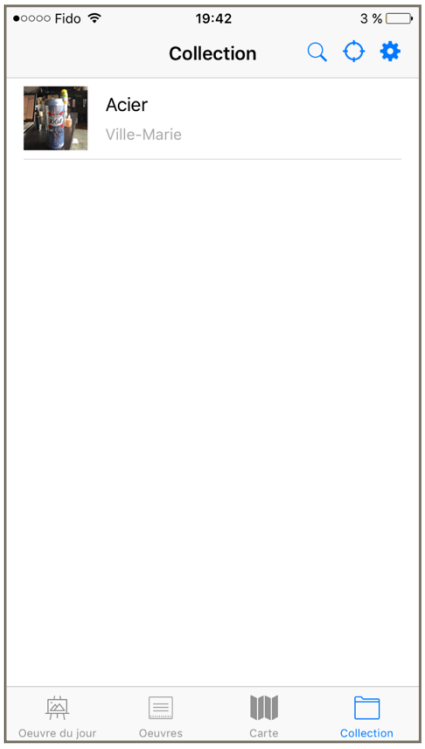


Figure 4.6: La section « Collection » (ancienne version)

Par ailleurs, une classe nommée « UserDefaultsManager » est responsable de la conservation des préférences de l'utilisateur. La vue qui permet à l'utilisateur de paramétrer ses préférences s'ouvre en cliquant sur le bouton en forme de rouage présenté à la figure 4.3. Les paramètres disponibles sont les suivants:

1. Choix de la langue de l'application: français ou anglais;
2. Choix de la méthode de partage des données: désactivé, wifi seulement, cellulaire et wifi.

Enfin, lors du lancement de l'application, un utilisateur peut se créer un compte ou s'y connecter. Le « storyboard » pour l'authentification d'un usager est présenté à l'annexe X.

4.2 Identification des problèmes

Il est nécessaire d'identifier les problèmes posés par l'ancienne version afin de pouvoir développer les objectifs pour ce projet de recherche appliquée à la maîtrise.

Dans l'ancienne version de l'application, la classe « NSKeyedArchiver » du cadriciel « Foundation » est utilisée sauvegarder les données du modèle (voir section 4.1.1). Bien que cette solution pour la persistance des données soit viable, elle est beaucoup moins performante que l'utilisation du cadriciel « Core Data » pour le chargement et la sauvegarde des données.

Les opérations réseau sont dépendantes des « packages » « HermesNetwork » et « Alamofire » pour l'exécution des requêtes HTTP et de « SwiftyJSON » pour le « parsing » des réponses JSON. Bien que ces « packages » proposent de nombreuses fonctionnalités, ils peuvent néanmoins poser des problèmes de performance. Par exemple, une requête HTTP simple, créée avec « Alamofire », peut induire des vérifications non nécessaires qui pénalisent la performance d'une requête lors de son initialisation ou de son exécution. Les « packages » peuvent aussi poser des problèmes de maintenance. Par exemple, un certain temps peut passer avant qu'un défaut tapi dans un « package » ne soit corrigé par l'équipe de développement responsable de la maintenance de ce « package ». Considérons aussi un autre exemple: si une nouvelle version d'iOS est mise à jour, l'équipe de développement chargée de la maintenance d'un « package » peut prendre un temps non négligeable à adapter un « package » à une nouvelle version d'iOS. En conséquence, ceci peut ralentir le développement et la maintenance de l'application. En réalité, la décision d'utiliser ou non un « package » réside dans sa capacité à résoudre plus de problèmes qu'il n'en crée. Or, dans le cas de MONA, les opérations réseau étant relativement simples, et les besoins de performance et de maintenance élevés, il conviendrait de réduire la dépendance à ces « packages » en utilisant le cadriciel préconisé par Apple pour les requêtes HTTP, à savoir la classe « URLSession » du cadriciel « Foundation ».

Il a été possible d'observer, à diverses occasions, des tests d'usagers utilisant l'ancienne version de MONA. Ce fut notamment le cas dans le cadre de l'activité d'intégration organisée

par l'AÉÉHAUM à la rentrée de septembre 2018 où une dizaine d'utilisateurs issus de filières universitaires en histoire de l'art ont pu tester l'application. Au cours de ces tests d'usagers, il a été possible de tirer les enseignements suivants:

- Les utilisateurs avaient souvent de vieux iPhone comme l'iPhone 4S. La taille d'écran de 3.5 pouces de ce téléphone, une taille assez petite comparativement aux modèles de téléphones plus récents (par exemple, l'iPhone XS Max sorti en 2018 a une taille d'écran de 6.5 pouces), n'affichait pas correctement certaines interfaces. De plus, la dernière version majeure d'iOS est actuellement iOS 13 alors que la dernière version majeure disponible pour l'iPhone 4S est iOS 9. Le développement de MONA doit ainsi être rendu disponible sur un éventail assez large de versions d'iOS afin de rejoindre le public d'utilisateurs ciblés, mais cela implique de ne pas pouvoir utiliser des cadriciels à jour ou des cadriciels plus récents. Enfin, la puissance de ces téléphones pouvait être parfois bien trop faible pour utiliser des fonctionnalités gourmandes en puissance de calcul et en mémoire, comme la section « Carte » qui doit gérer l'affichage de centaines d'annotations ou le parcours de liste de centaines d'œuvres;
- Quelques utilisateurs semblaient troublés par certaines incohérences dans le fonctionnement de l'application : par exemple, certains utilisateurs trouvaient qu'il était étrange de pouvoir paramétrer l'utilisation des données directement dans l'application au lieu de le faire dans l'application « Paramètres » sur iOS. D'autres utilisateurs trouvaient aussi qu'il était frustrant de ne pas pouvoir accéder dans l'application « Photos » aux photos prises dans l'application MONA pour les conserver, les retoucher ou les partager. En effet, les photos des œuvres d'art collectionnées sont accessibles uniquement dans l'application MONA;
- Très souvent, les utilisateurs ne remarquaient pas qu'il était possible de changer de liste à l'aide du bouton de sélection de liste situé dans la barre de navigation comme le montre la figure 4.2;
- Enfin, les utilisateurs avaient beaucoup de difficulté à cliquer de manière précise sur les boutons situés dans la barre de navigation en haut à droite comme le montre la figure 4.2. Les trois boutons étaient bien trop rapprochés (c.-à-d. recherche, liste d'œuvres ciblées, paramètres).

Certains besoins explicités pour l'application n'ont pas été comblés ou ne l'ont été que de manière partielle:

- La section « Œuvre de jour » n'a pas été implémentée;
- La section « Collection » montrée à la figure 4.6 est identique à la liste des œuvres montrée à la figure 4.2 : cette section, en plus de ne pas rendre agréable le parcours de cette collection censée être semblable à une galerie d'art personnelle, peut être facilement confondue par l'utilisateur avec la liste des œuvres;
- Le système de récompenses par badge a été partiellement implémenté: seuls des noms de badges et quelques valeurs cibles ont été enregistrés. En revanche, l'utilisateur n'était pas alerté lorsqu'il gagnait un badge. De plus, aucune interface n'avait été conçue afin de permettre à l'utilisateur de parcourir sa collection de badges;
- Les traductions pour les techniques ou les matériaux manquaient de cohérence. En effet, certains noms de matériaux ou de techniques ne disposaient pas d'une traduction en anglais, ou étaient traduits par deux mots anglais distincts (par exemple, la technique « boulonnée » se retrouvait traduite par « screwed » ou « bolted » selon l'œuvre). Étant donné le manque de qualité des données pour les techniques ou les matériaux, il n'est pas possible de parcourir la liste des œuvres selon le matériau ou la technique utilisée;
- Le système d'authentification n'est pas sécuritaire puisque les requêtes HTTP d'authentification ne sont pas chiffrées;
- Les fonctionnalités de partage et de commentaires des photos d'œuvres d'art sur les réseaux sociaux n'ont pas été implémentées;
- Pas de système de notification mis en place;
- Enfin, l'application n'est pas diffusée sur l'App Store alors que c'est un objectif incontournable de ce projet.

On peut également déplorer l'absence de tests unitaires et d'intégrations dans le logiciel, mais qui s'explique par le fait qu'il reste à déterminer ce que signifie « tester une application mobile iOS » dans le contexte d'une application qui affiche essentiellement des œuvres d'art.

Enfin, tout au long de ce projet d'application et au fil des réunions de l'équipe, les besoins pour l'application se sont raffinés, précisés ou parfois complexifiés. Certaines fonctionnalités de l'application ont parfois été redéveloppées. Ainsi, le code de l'application évoluant, il pouvait être rendu confus ou mal conçu à certains endroits.

4.3 Définition des objectifs

À partir des problématiques qui ont été identifiées à la section 4.2, on peut fixer pour le projet d'application les objectifs suivants:

- Implémenter une solution de persistance des données plus performante;
- Refaire la partie réseau en adaptant l'application à la nouvelle API tout en en profitant pour réduire la dépendance aux « packages »;
- Adapter l'application à la nouvelle API pour implémenter les listes de matériaux et les listes de techniques, le téléchargement et le téléversement des données et une gestion de l'authentification plus sécuritaire;
- Implémenter un nouveau design pour l'application plus utilisable pour les utilisateurs et plus performant tout en s'assurant de son fonctionnement sur les différentes tailles d'iPhone;
- Supprimer la vue des « Paramètres » de l'application par souci de cohérence avec la plateforme iOS;
- Lors de prise de photo, la photo doit être enregistrée dans l'application « Photos » et ne plus être emprisonnée dans l'application;
- Retravailler la section « Collection »;
- Implémenter une section « Œuvre du jour »;
- Implémenter un système de badges;
- Implémenter un système de gestion de l'authentification;
- Implémenter un système de notification;
- Proposer des fonctionnalités de partage de photo des œuvres;
- Assurer une bonne performance générale de l'application (c.-à-d. un usage de la mémoire et une réactivité améliorée);

- Réorganiser le code pour le rendre plus clair et plus maintenable;
- Mettre en place des tests unitaires pour détecter les problèmes dans l'application et faciliter la maintenance;
- Mettre en place une documentation complète de l'application pour faciliter la reprise du projet par d'autres développeurs à l'avenir (l'application se voulant libre de droits).

Ultimement, l'objectif final de ce projet est de s'assurer de sa disponibilité sur l'App Store.

4.4 Nouvelle version

4.4.1 Conception

Un ensemble d'enseignements provenant de la version initiale du prototype de l'application, permettent de concevoir une nouvelle interface plus esthétique et plus centrée sur les besoins de l'utilisateur. La phase d'idéation et de conception d'interfaces a débuté en dessinant les interfaces sur papier. L'avantage du papier est qu'il est facile et peu coûteux de concevoir des interfaces et de les prétester auprès des utilisateurs. La figure 4.7 permet d'avoir un aperçu de la phase d'idéation et de conception de l'interface des badges sur papier.

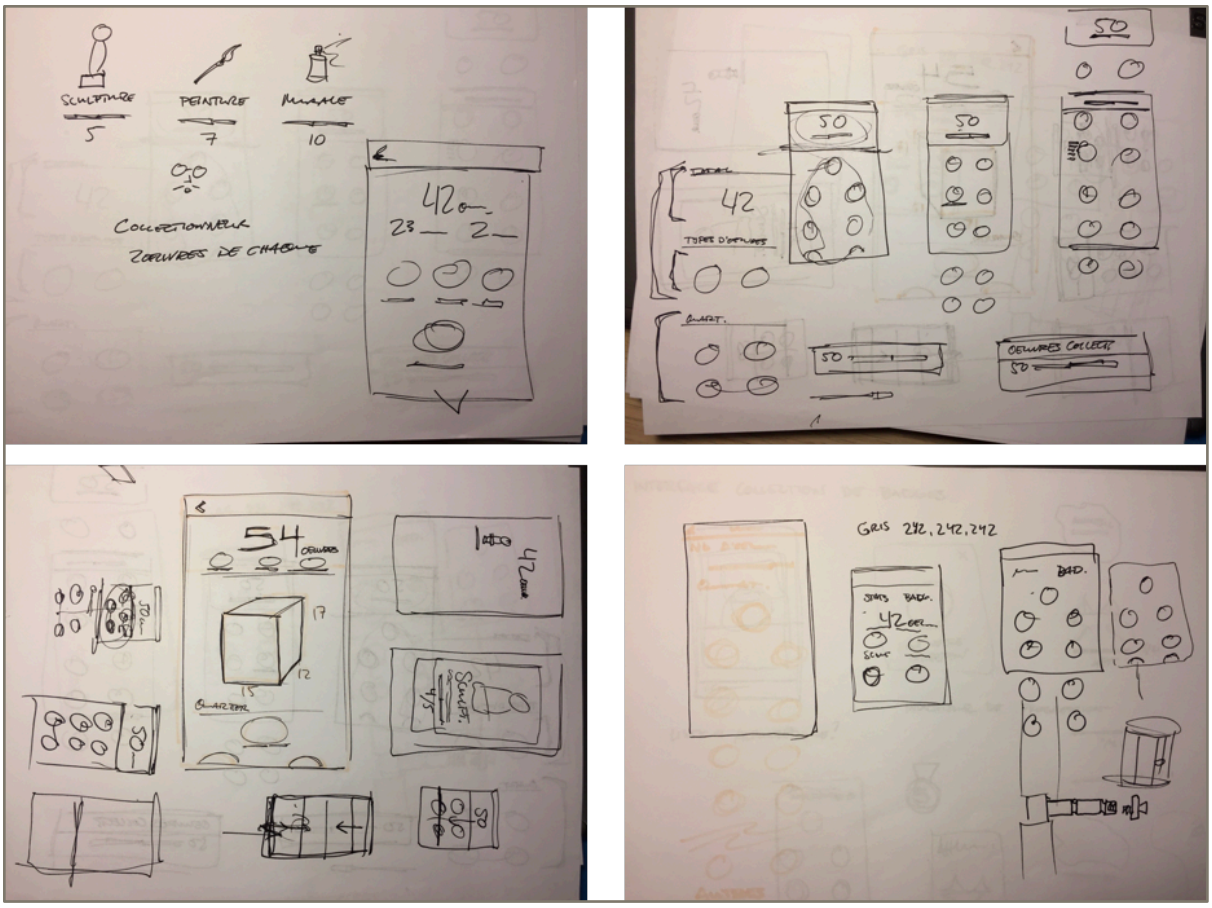


Figure 4.7: Prototypage papier de l'interface des badges

Dans une seconde phase, les prototypes d'interface, les icônes et le logo de l'application ont été dessinés avec le logiciel Adobe Illustrator après la prise en compte de rétroactions des utilisateurs. Cette phase a permis d'obtenir un aperçu des interfaces au plus fidèle à ce qui sera réellement implémenté. Les annexes XI, XII, XIII et XIV montrent les interfaces telles qu'elles ont été dessinées dans Adobe Illustrator. La figure 4.8 montre le logo dessiné pour l'application, alors que la figure 4.9 montre les icônes de section de l'application dans l'ordre « Œuvre du jour », « Œuvres », « Carte » et « Collection ». Enfin, la figure 4.10 montre l'ensemble des badges d'arrondissement dessinés pour l'application, tandis que la figure 4.11 montre l'ensemble des badges numéraires.

Figure 4.8: Logo MONA

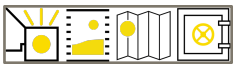


Figure 4.9: Icônes de section

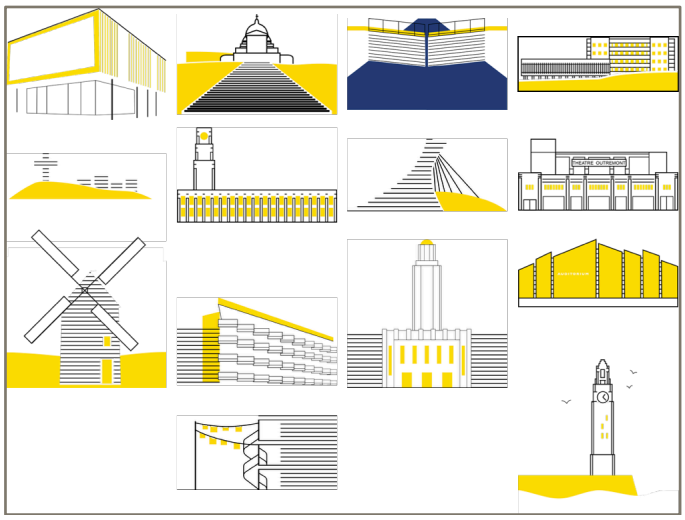


Figure 4.10: Badges d'arrondissements

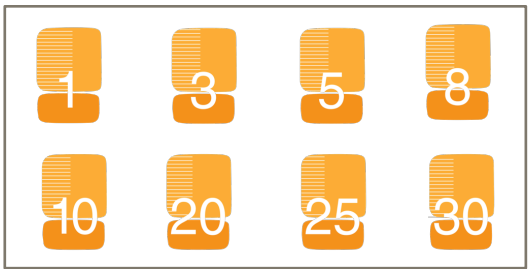


Figure 4.11: Badges numéraires

Concernant la programmation, la gestion des données sera effectuée à l'aide du cadriciel « Core Data » pour des raisons de performance. L'application doit aussi s'adapter aux

changements de serveur Web (tels que décrits à la section 3.4) et implémenter une toute nouvelle interface. Ces nombreux changements impliquent une modification d'environ 80% de la version initiale du prototype de l'application. Cependant, certains algorithmes pourront être réutilisés dans cette nouvelle version de l'application, comme les algorithmes de tri des œuvres par exemple. Enfin, la flexibilité et la complétude du modèle de données conçu à la section 4.1.1 font que ce modèle de données ne nécessite pas de modifications majeures pour les propriétés et les relations entre entités.

4.4.2 Implémentation

Cette section du rapport présente la phase d'implémentation de la nouvelle version de l'application MONA.

4.4.2.1 Le modèle

Le modèle de données présenté à la section 4.1.1 (voir figure 4.1) a été repris dans l'outil de conception de la base de données du cadriciel « Core Data », moyennant quelques modifications comme le montre la figure 4.12 ci-après.

Rappelons que l'application doit être multilingue. Des entités, comme les catégories ou les techniques, peuvent avoir un nom différent selon la langue. Ainsi, le modèle de données présenté à la section 4.1.1 (voir figure 4.1) décrit la classe « Category » ou « Technique » avec une propriété « names » de type « Dictionary » où la clé est de type « Language » et la valeur est la chaîne de caractères correspondant à la langue. Or, le cadriciel « Core Data » ne permet pas de créer une entité ayant une propriété de type « Dictionary ». Pour contourner ce problème, les entités « Badge », « Technique », « Material », « Category », « Subcategory » hériteront de « LocalizableEntity ». Une instance de « LocalizableEntity » a une propriété « localizedNames » de type « Set<LocalizedString> ». Une instance de « LocalizedString » a une propriété « language » qui définit la langue de la chaîne de caractères « localizedString ». Ainsi les entités héritant de « LocalizableEntity » ont une propriété « localizedNames » qui

contient les noms multilingues associés à ces entités et qui contournent le problème de ne pas pouvoir utiliser de propriétés de type « Dictionary ».

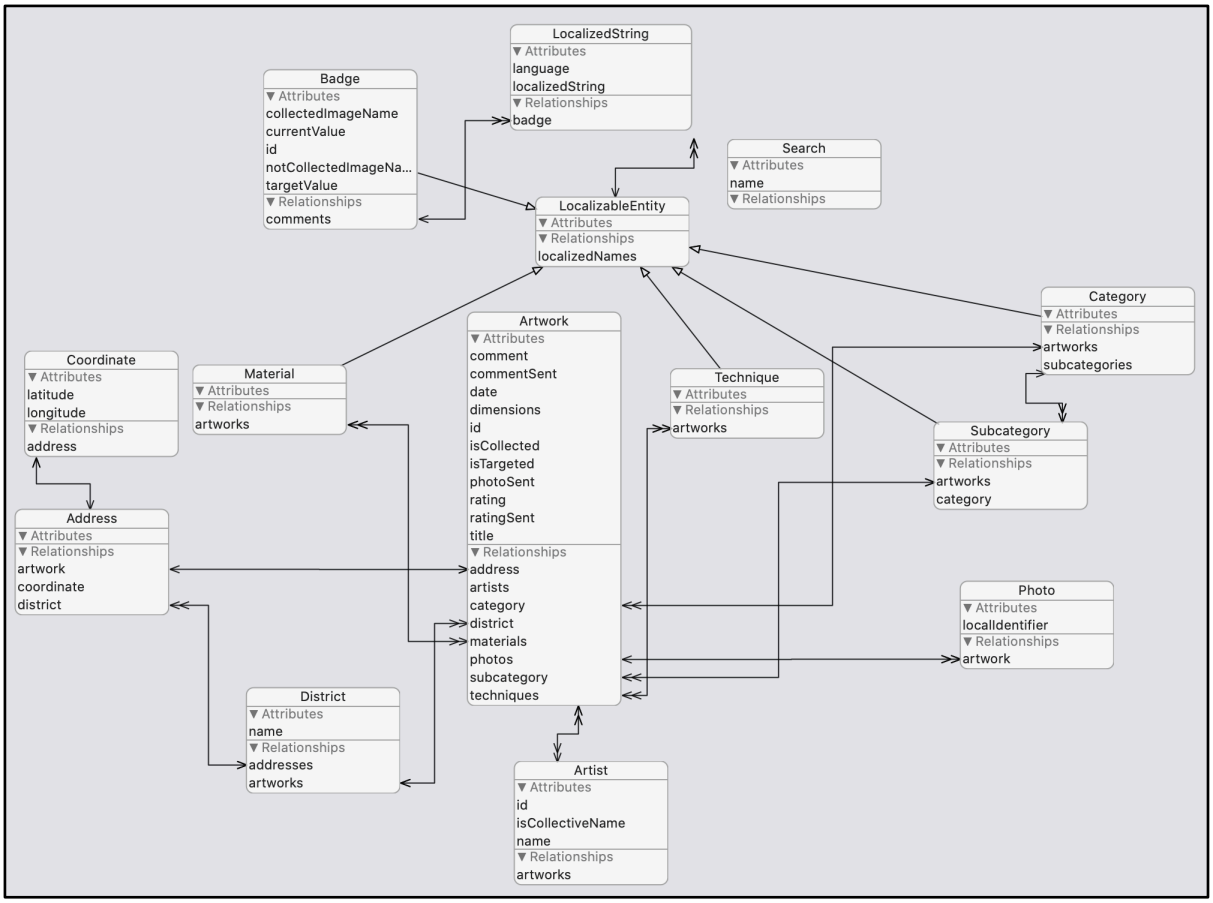


Figure 4.12: Diagramme d'entités conçu pour « Core Data » (nouvelle version)

Le modèle de données de la figure 4.12 accueille également une nouvelle entité « Subcategory » qui facilite les manipulations dans la base de données comparativement à une entité récursive sur « Category ».

Une entité « Badge » est aussi une nouvelle entité du modèle de données. Un badge possède une propriété « collectedImageName » qui désigne le nom du fichier image qui doit être affiché pour représenter le badge dans l'application quand celui-ci est collectionné. Il a aussi la propriété « notCollectedImageName » qui désigne le nom du fichier image du badge quand il n'est pas collectionné. De plus, un badge a la propriété « currentValue », qui représente le

nombre d'œuvres déjà collectionnées pour ce badge, et la propriété « targetValue », qui représente le nombre d'œuvres qu'il faut collectionner pour obtenir le badge. Lorsque « currentValue » est supérieur ou égal à « targetValue », alors le badge est considéré comme collectionné. Une entité « Badge » hérite de « LocalizableEntity » et peut donc avoir plusieurs noms en fonction de la langue. Enfin, un badge a une propriété « comments » qui permet d'afficher un message lorsqu'un badge est obtenu en fonction de la langue.

Notons enfin l'apparition de l'entité « Search » qui permet de conserver un historique des recherches de l'utilisateur.

Xcode fournit un outil de génération de classes, à partir du diagramme de base de données présenté à la figure 4.12. Par exemple, l'algorithme 4.6 montre la classe « Material » générée à partir du diagramme de Xcode avec ses propriétés et ses accesseurs.

```
import Foundation
import CoreData

@objc(Material)
final public class Material: LocalizableEntity { }

extension Material {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<Material> {
        return NSFetchRequest<Material>(entityName: "Material")
    }

    @NSManaged public var artworks: Set<Artwork>
}
// MARK: Generated accessors for artworks
extension Material {
    @objc(addArtworksObject:)
    @NSManaged public func addToArtworks(_ value: Artwork)

    @objc(removeArtworksObject:)
    @NSManaged public func removeFromArtworks(_ value: Artwork)

    @objc(addArtworks:)
    @NSManaged public func addToArtworks(_ values: NSSet)

    @objc(removeArtworks:)
    @NSManaged public func removeFromArtworks(_ values: NSSet)
}
```

Algorithme 4.6: Classe « Material » générée par Xcode à partir du modèle

Un « NSManagedObject » désigne un objet pouvant être géré par « Core Data ». Toutes les classes du modèle générées héritent de « NSManagedObject ». Comme le montre l'algorithme 4.7 ci-dessous, la classe « LocalizableEntity » générée hérite de « NSManagedObject », et « Material » hérite donc de l'implémentation de « NSManagedObject ».

```
import Foundation
import CoreData

@objc(LocalizableEntity)
public class LocalizableEntity: NSManagedObject {
}

extension LocalizableEntity {
    @nonobjc public class func fetchRequest() ->
NSFetchRequest<LocalizableEntity> {
        return NSFetchRequest<LocalizableEntity>(entityName:
"LocalizableEntity")
    }

    @NSManaged public var localizedNames: Set<LocalizedString>
}

// MARK: Generated accessors for localizedNames
extension LocalizableEntity {
    @objc(addLocalizedNamesObject:)
    @NSManaged public func addToLocalizedNames(_ value: LocalizedString)

    @objc(removeLocalizedNamesObject:)
    @NSManaged public func removeFromLocalizedNames(_ value: LocalizedString)

    @objc(addLocalizedNames:)
    @NSManaged public func addToLocalizedNames(_ values: NSSet)

    @objc(removeLocalizedNames:)
    @NSManaged public func removeFromLocalizedNames(_ values: NSSet)
}
```

Algorithme 4.7: Classe « LocalizableEntity » générée par Xcode à partir du modèle (nouvelle version)

Il est possible d'ajouter de nouveaux comportements aux classes générées. L'algorithme 4.8 ci-après montre par exemple une méthode qui exécute une requête permettant d'obtenir toutes les instances « Material » qui respectent un prédicat passé en paramètre.

```
    @nonobjc public class func fetchRequest(predicate: NSPredicate?, context:
NSManagedObjectContext) -> [Material] {
        do {
            let fetchRequest : NSFetchRequest<Material> =
Material.fetchRequest()
            fetchRequest.predicate = predicate
            let fetchedResults = try context.fetch(fetchRequest)
            return fetchedResults
        }
        catch {
            log.error("Fetch task failed: \(error)")
            return [Material]()
        }
    }
```

Algorithme 4.8: Obtenir toutes les instances de « Material » selon un prédicat (nouvelle version)

4.4.2.2 Gestion des données

La gestion des données s'effectue à l'aide du cadriciel « Core Data » qui permet de gérer les objets du modèle dans l'application. Ce cadriciel fournit un ensemble de méthodes permettant de gérer un graphe d'objets et le cycle de vie des objets. « Core Data » gère également la persistance de ces données. La figure 4.13 présente un aperçu du schéma du fonctionnement de ce cadriciel.

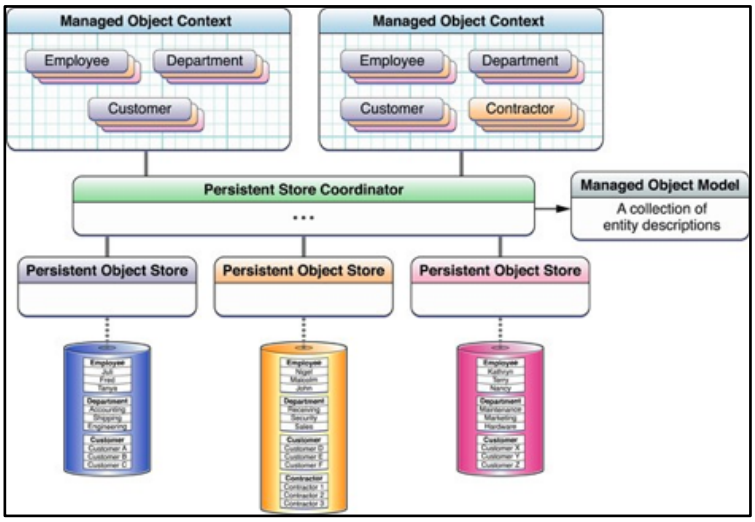


Figure 4.13: Schéma du fonctionnement de « Core Data »

La classe « NSPersistentStore » permet de lire et écrire des données dans un magasin selon un type de stockage choisi, comme SQLite ou XML. La classe « NSPersistentStoreCoordinator » s'initialise avec le modèle défini dans la section 4.4.2.1, ce qui permet à une instance de cette classe de coordonner les lectures et écritures des données des entités dans les magasins de données.

Une instance de « NSManagedObjectContext » permet de garder la trace de multiples « NSManagedObject ». Rappelons que toute entité du modèle hérite de « NSManagedObject ». Les objets du modèle sont insérés dans un contexte grâce au coordinateur qui permet d'extraire les entités des magasins de données pour les insérer dans un contexte. Les objets peuvent aussi être créés, mis à jour ou supprimés dans un contexte et la sauvegarde d'un contexte déclenche l'écriture de ces changements dans les magasins de données.

L'application interface le cadriciel « Core Data » avec un singleton nommé « CoreDataStack ». Ce singleton garde une référence sur une instance de « NSPersistentCoordinator » initialisée avec le modèle décrit en 4.4.2.1 et qui joue le rôle de coordonnateur. Ce coordonnateur garde trace d'un unique magasin de données au format SQLite pour stocker les données du modèle. « CoreDataStack » garde aussi une référence sur une instance de « NSManagedContext » qui joue le rôle de contexte principal de l'application. Ce contexte permet à l'application de gérer les objets du modèle, comme les instances de « Artwork ».

Afin que l'application se lance très rapidement, le magasin de données vient précharger avec les œuvres d'art. Si Internet n'est pas accessible, l'application peut fonctionner avec les données déjà présentes. En revanche, si Internet est accessible au lancement de l'application, une requête est effectuée à l'API pour télécharger le dernier fichier JSON des œuvres d'art. Après la phase de « parsing » du fichier JSON, on insère dans un second contexte toutes les œuvres d'art déjà enregistrées. Si de nouvelles œuvres d'art apparaissent dans le JSON, il suffit de créer de nouveaux objets « Artwork », tandis que si ces œuvres existent déjà, elles sont

mises à jour avec les nouvelles données. À la fin de cette phase, le second contexte est sauvegardé ce qui déclenche l'écriture en magasin des mises à jour des objets du modèle.

Les préférences sont gérées avec une classe nommée « UserDefaults » qui est une interface programmatique du cadriciel « Foundation ». « UserDefaults » est préconisé pour faire persister les préférences d'un usager sous la forme de paires clé-valeurs. Cette classe est utilisée dans l'application pour stocker le jeton d'authentification de session et la langue préférée de l'usager.

4.4.2.3 Gestion des photos

La version précédente de l'application sauvegardait les photos prises par l'utilisateur dans le système de fichiers interne à l'application au format PNG. Cette méthode rendait les photos inaccessibles à l'utilisateur en dehors de « MONA » et avait provoqué la frustration des usagers. En effet, les utilisateurs s'attendent typiquement à retrouver leurs photos prises sur les différentes applications de leur iPhone dans l'application « Photos ». Cette application est normalement l'application qui permet à un utilisateur d'iPhone de gérer sa bibliothèque de photos. Dans cette nouvelle version de l'application, toutes les photos prises par l'utilisateur dans l'application « MONA » seront maintenant accessibles dans « Photos ».

Pour interagir avec la bibliothèque de photos d'un iPhone, il faut utiliser le cadriciel « PhotoKit ». Un singleton nommé « MonaPhotosAlbum » est utilisé dans l'application pour faciliter les procédures d'insertion et d'extractions des photos avec ce cadriciel. Au lancement de l'application « MONA », le singleton « MonaPhotosAlbum » est initialisé et crée un album photo « MONA » dans l'application « Photos ».

Lors de la procédure de sauvegarde d'une photo dans la bibliothèque de photos de l'utilisateur à l'aide de « PhotoKit », il est nécessaire de récupérer la valeur de la propriété « localIdentifier » de la photo, qui est une chaîne de caractères identifiant de manière unique une photo, pour en garder la trace. À l'aide de cette chaîne de caractères, il est possible d'insérer dans le contexte principal une entité « Photo » (voir le modèle à la figure 4.12 dans

la section 4.4.2.1) et l'ajouter à l'instance de « Artwork » pour laquelle la photo a été prise. Puis on sauvegarde avec « CoreDataStack » le contexte afin de faire persister les changements dans la base de données locale. Plus tard, il est possible d'extraire une photo de la bibliothèque de photos de l'utilisateur avec la propriété « localIdentifier » grâce à l'algorithme 4.9 présenté ci-dessous pour afficher la photo d'une œuvre collectionnée.

```
func fetchAsset(withLocalIdentifier localIdentifier: String) -> PHAsset? {
    let fetchOptions = PHFetchOptions()
    fetchOptions.includeAllBurstAssets = false
    fetchOptions.includeHiddenAssets = true
    fetchOptions.includeAssetSourceTypes = [.typeCloudShared, .typeUserLibrary,
.typeiTunesSynced]
    let result = PHAsset.fetchAssets(withLocalIdentifiers: [localIdentifier],
options: fetchOptions)
    return result.firstObject
}
```

Algorithme 4.9: Procédure d'extraction de photo (nouvelle version)

Au lancement de l'application « MONA » ou lorsque l'application revient en avant-plan, une procédure vérifie que l'utilisateur n'a supprimé aucune photo de sa bibliothèque, auquel cas une œuvre est toujours considérée comme collectionnée. Alternativement, à la place de la photo sur la fiche d'une œuvre, on indique que la photo a été supprimée.

4.4.2.4 Gestion des badges

Les données des badges sont incluses directement dans le code sous la forme d'une liste de paires clés-valeurs comme le montre l'algorithme 4.10 ci-dessous. À l'installation de l'application et à la suite de son lancement, cette liste est parcourue et les données sur les badges sont extraites pour être insérées dans le contexte principal de l'application sous la forme d'entité « Badge ». Puis ces badges sont inscrits dans la base de données à l'aide du singleton « CoreDataStack » qui procède à la sauvegarde du contexte.

```
"name" : [
    Language.en : "Côte-des-Neiges–Notre-Dame-de-Grâce",
    Language.fr : "Côte-des-Neiges–Notre-Dame-de-Grâce"
],
"currentValue" : 0,
"targetValue" : 10,
"collectedImageName" : "Cote-des-Neiges - Normal",
"notCollectedImageName" : "Cote-des-Neiges - Gris",
"comment" : [
    Language.fr : "Bravo, vous avez collectionné 10 œuvres dans Côte-des-Neiges-
Notre-Dame-de-Grâce. En avez-vous profité pour visiter l'Oratoire St-Joseph?",
    Language.en : "Well done, you collected 10 artworks in Côte-des-Neiges-
Notre-Dame-de-Grâce. Have you visited St Joseph's Oratory while you're at it?"
]
```

Algorithme 4.10: Données des badges (nouvelle version)

La vérification des badges gagnés est effectuée à chaque fois qu'un utilisateur prend une photo d'une œuvre. L'algorithme 4.11 ci-dessous montre la procédure de vérification des badges gagnés. Chaque badge obtenu est ensuite affiché après la prise de photo.

```
let notCollectedBadges = AppData.badges.filter { !$0.isCollected }
for notCollectedBadge in notCollectedBadges {
    if notCollectedBadge.text == artwork.district.text {
        notCollectedBadge.currentValue += 1
    }
    else if ["1", "3", "5", "8", "10", "15", "20", "25",
"30"].contains(notCollectedBadge.collectedImageName) {
        notCollectedBadge.currentValue += 1
    }
}
let newlyCollectedBadges = notCollectedBadges.filter { $0.isCollected }
```

Algorithme 4.11: Vérification des badges collectionnées (nouvelle version)

4.4.2.5 Gestion des opérations réseau

Comme le montre l'algorithme 4.12 ci-après, l'interaction avec le serveur Web est implémentée à l'aide d'un singleton nommé « MonaAPI » permettant d'exécuter quatre requêtes HTTP:

1. « register » : une méthode qui crée et exécute une requête HTTP permettant de procéder à l'inscription d'un nouvel utilisateur avec un nom d'utilisateur « username », un

courriel optionnel et un mot de passe « password ». La réponse à cette requête est un jeton qui est stocké dans les préférences avec « UserDefaults »;

2. « login » : une méthode qui crée et exécute une requête HTTP qui permet de connecter un nouvel utilisateur avec un nom d'utilisateur « username » et un mot de passe « password ». La réponse à cette requête est un jeton d'authentification qui est stocké dans les préférences avec « UserDefaults »;
3. « artwork » : une méthode qui crée et exécute une requête HTTP permettant de téléverser une note, un commentaire ou une photo en utilisant un jeton d'authentification;
4. « artworks » : une méthode qui crée et exécute une requête HTTP qui permet de télécharger le fichier JSON du serveur.

L'algorithme 4.12 montre aussi comment le jeton d'authentification, utilisé pour les requêtes vers le serveur Web, est récupéré de l'interface de gestion des préférences « UserDefaults ».

Les requêtes effectuées dans « MONA » implémentent un protocole « HTTPRequest », présenté à l'algorithme 4.13 ci-après. Elle doit spécifier l'URL, le type, les entêtes et le corps de la requête. Ce protocole implique aussi un type associé « Response ». « Response » est en fait un alias au nom d'une classe qui implémente le protocole « HTTPResponse ». Enfin, le protocole implique aussi l'exécution d'une méthode « execute », qui en cas de succès de la requête, « parse » la réponse JSON (un « HTTPDecodableResponse »), et en cas d'échec, signale une erreur (un « HTTPError »).

```
struct MonaAPI {

    static var shared = MonaAPI()
    var baseURLString: String = "https://picasso.iro.umontreal.ca/~mona/api"
    var apiToken : String? = {
        return UserDefaults.Credentials.get(forKey: .token)
    }()

    func artwork(id: Int, rating: Int?, comment: String?, photo: UIImage?,
completion: @escaping (Result<ArtworkRequest.Response.HTTPDecodableResponse,
HTTPError>) -> Void) {
        let session = URLSession.shared
        let request = ArtworkRequest(id: id, rating: rating, comment: comment,
photo: photo)
        request.execute(session: session, completion: completion)
    }

    func artworks(completion: @escaping
(Result<ArtworksRequest.Response.HTTPDecodableResponse, HTTPError>) -> Void) {
        let session = URLSession.shared
        let request = ArtworksRequest()
        request.execute(session: session, completion: completion)
    }

    func login(username: String, password: String, completion: @escaping
(Result<LoginRequest.Response.HTTPDecodableResponse, HTTPError>) -> Void) {
        let session = URLSession.shared
        let request = LoginRequest(username: username, password: password)
        request.execute(session: session, completion: completion)
    }

    func register(username: String, email: String?, password: String,
completion: @escaping (Result<RegisterRequest.Response.HTTPDecodableResponse,
HTTPError>) -> Void) {
        let session = URLSession.shared
        let request = RegisterRequest(username: username, email: email,
password: password)
        request.execute(session: session, completion: completion)
    }
```

Algorithme 4.12: Implémentation de la nouvelle API (nouvelle version)

```
protocol HTTPRequest {
    associatedtype Response : HTTPResponse
    var URL: URL { get }
    var method: HTTPMethod { get }
    var headers: HTTPHeaders? { get }
    var body: Data? { get }

    func execute(session: URLSession, completion: @escaping
(Result<Response.HTTPDecodableResponse, HTTPError>) -> Void)
}
```

Algorithme 4.13: Le protocole « HTTPRequest » (nouvelle version)

Comme le montre l'algorithme 4.14 ci-dessous, le protocole « HTTPResponse » possède deux types associés nommés « HTTPDecodableResponse » et « HTTPErrorDecodableResponse ». Le premier type permet de « parser » la réponse du serveur à une requête réussie, tandis que le second permet de « parser » une réponse du serveur à une requête qui a échoué. Le protocole « Decodable » permet en effet de « parser » facilement une réponse JSON.

```
protocol HTTPResponse {
    associatedtype HTTPDecodableResponse : Decodable
    associatedtype HTTPErrorDecodableResponse : Decodable

    func process(completion: ((Result<HTTPDecodableResponse, HTTPError>) ->
Void)?) -> (Data?, URLResponse?, Error?) -> Void
    func handle(_ result: Result<Data, Error>) -> Result<HTTPDecodableResponse,
HTTPError>
}
```

Algorithme 4.14: Le protocole « HTTPResponse » (nouvelle version)

L'algorithme 4.15, de la page suivante, montre comment le protocole « HTTPRequest » est implémenté pour une requête d'authentification « login ». L'URL de la requête est construite à partir de l'URL de base de l'API auquel est concaténée la chaîne de caractères « /login ». La méthode est de type POST tandis que les entêtes avertissent le serveur Web que le contenu de la requête et que le type de réponse attendu est au format JSON. À l'initialisation de la requête, les paramètres « username » et « password » sont encodés en JSON dans le corps de la requête « body ». L'alias « Response » de la requête est « CredentialsReponse », qui implémente le protocole « HTTPResponse ». Dans « CredentialsResponse », « HTTPDecodableResponse » est l'alias de « TokenDecodableResponse » tandis que « HTTPErrorDecodableResponse » est l'alias de « CredentialsErrorDecodableResponse ».

```
    struct LoginRequest : HTTPRequest {

        typealias Response = CredentialsResponse

        // The url of the request
        var url: URL = URL(string: MonaAPI.shared.baseURLString + "/login")!
        // HTTP method used by the request
        var method: HTTPMethod = .post
        // HTTP Headers of the http request
        var headers: HTTPHeaders? = [
            .accept : "application/json",
            .contentType : "application/json"
        ]
        // HTTP body of the request
        var body: Data?

        // Init the body of the request with the parameters username, email and
password
        init(username: String, password: String) {
            let parameters = [
                "username" : username,
                "password" : password
            ]
            body = try? JSONSerialization.data(withJSONObject: parameters,
options: .prettyPrinted)
        }

        // Execute the request
        // Can be a dataTask, a uploadTask or a downloadTask
        func execute(session: URLSession, completion: @escaping
(Result<Response.HTTPDecodableResponse, HTTPError>) -> Void) {
            let response = CredentialsResponse()
            let task = session.dataTask(with: urlRequest, completionHandler:
response.process(completion: completion))
            task.resume()
        }
    }
```

Algorithme 4.15: Une requête « login » (nouvelle version)

L'algorithme 4.16 ci-dessous montre « TokenDecodableResponse » qui permet de « parser » la réponse JSON à une requête « register » ou à une requête « login » qui a réussi. On obtient de cette façon le jeton d'authentification.

```
//MARK: HTTPDecodableResponse
struct TokenDecodableResponse : Codable {
    let token : String
}
```

Algorithme 4.16: « HTTPDecodableResponse » alias de « TokenDecodableResponse » (nouvelle version)

L'algorithme 4.17 ci-dessous montre « CredentialsErrorDecodableResponse » qui permet de « parser » la réponse à une requête « register » ou à une requête « login » qui a échoué. Par exemple, une requête « login » peut échouer si un utilisateur donne un nom d'utilisateur qui n'existe pas sur le serveur ou un mot de passe incorrect. Le « parsing » de cette réponse permet d'obtenir les détails de l'erreur.

```
//MARK: HTTPErrorDecodableResponse
struct CredentialsErrorDecodableResponse : Codable, CustomStringConvertible {
    let message : String
    let errors : RegisterErrorDetails

    struct RegisterErrorDetails : Codable {
        let username : [String]?
        let email : [String]?
        let password : [String]?
    }

    var description: String {
        var result = message
        if let usernameErrors = errors.username {
            result += "\nusername: " + usernameErrors.joined(separator: ", ")
        }
        if let emailErrors = errors.email {
            result += "\nemail: " + emailErrors.joined(separator: ", ")
        }
        if let passwordErrors = errors.password {
            result += "\npassword: " + passwordErrors.joined(separator: ", ")
        }
        return result
    }
}
```

Algorithme 4.17: « HTTPErrorDecodableResponse » alias de « CredentialsErrorDecodableResponse » (nouvelle version)

D'autres erreurs peuvent subvenir au cours de l'exécution d'une requête. Le type « enum » « HTTPError » (voir algorithme 4.18) a été créé pour prendre en compte les erreurs de type réseau (par exemple, une erreur HTTP 403), les erreurs de décodage (par exemple, le « parsing » JSON de la réponse à la requête HTTP qui échoue, car le serveur a changé le format de la réponse), et « requestError », qui est une requête HTTP dont l'exécution a réussie, mais dont l'API signale qu'une erreur s'est produite (par exemple, un nom d'utilisateur qui n'existe pas).

```
enum HTTPError : Error, LocalizedError {
    case unknownError(error: Error)
    case dataError(dataError: DataError)
    case networkError(networkError: NetworkError)
    case decodingError(decodingError: DecodingError)
    case requestError(httpErrorDecodableResponse: Decodable)

    public var errorDescription: String? {
        switch self {
        case .unknownError(let error):
            return error.localizedDescription
        case .dataError(let dataError):
            return dataError.localizedDescription
        case .networkError(let networkError):
            return networkError.localizedDescription
        case .decodingError(let decodingError):
            return decodingError.localizedDescription
        case .requestError(let httpErrorDecodableResponse):
            return "Invalid request: \(httpErrorDecodableResponse)"
        }
    }
}
```

Algorithme 4.18: Le type « HTTPError » (nouvelle version)

Rappelons que l'application requiert de récupérer les données des utilisateurs sur les œuvres d'art, comme les notes, les commentaires et les photos. Pour remplir ce besoin, une procédure de téléversement de données (par exemple : les notes, les commentaires ou les photos des œuvres) s'effectuent périodiquement toutes les 30 secondes en arrière-plan. Cette procédure consiste à boucler sur toutes les œuvres d'art et à exécuter une requête « artworks » de téléversement pour les œuvres dont les propriétés « ratingSent », « commentSent » ou « photoSent » sont à « false ». Si l'exécution de la requête signale une erreur « HTTPError », comme une erreur réseau, alors les propriétés « ratingSent », « commentSent » et/ou « photoSent » restent à « false ». Dans le cas contraire, le téléversement a réussi et ces propriétés sont mises à « true » pour indiquer qu'il n'est plus nécessaire de téléverser ces données.

4.4.2.6 Organisation de l'application en sections

L'application est organisée en cinq sections: 1) une section « Œuvre du jour »; 2) une section « Œuvres »; 3) une section « Carte »; 4) une section « Collection »; et enfin 5) une section « Plus ».

L’éditeur d’interface d’Xcode facilite la conception d'une interface utilisateur complète sans écrire de code. Il suffit de glisser-déposer des fenêtres, des boutons, des champs de texte et d’autres objets sur le canevas de conception pour créer une interface utilisateur fonctionnelle.

Une application iOS est composée de plusieurs vues à travers lesquelles l’utilisateur navigue. Les relations entre ces vues sont définies par des « storyboards » qui montrent une vue complète du flux de navigation dans l’application. Ces « storyboards » facilitent la création et la conception de nouvelles vues, ainsi que leur enchaînement pour créer une interface utilisateur complète.

Comme le montre la figure 4.14, la classe « UITabBarController » du cadriciel « UIKit » permet d’ajouter cinq boutons de section dans l’application avec pour chacun une icône en format PNG. Chaque bouton de section mène vers un « storyboard » correspondant à celui de la section cliquée.

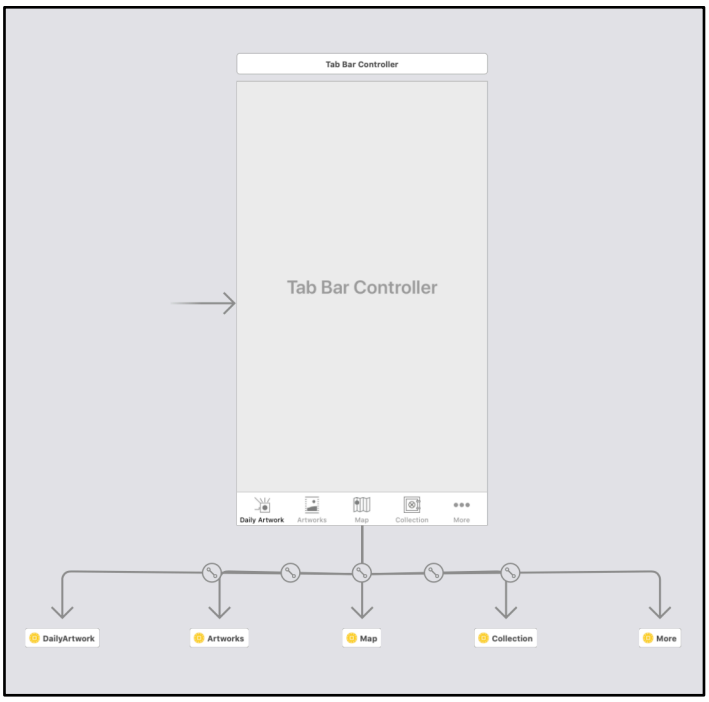


Figure 4.14: « Storyboard » pour l’organisation en sections de MONA (nouvelle version)

4.4.2.7 La section « Œuvre du jour »

La section « Œuvre du jour » a été conçue dans un « storyboard » indépendant à l'aide de l'éditeur d'interface Xcode tel que montré à la figure 4.15 ci-dessous.

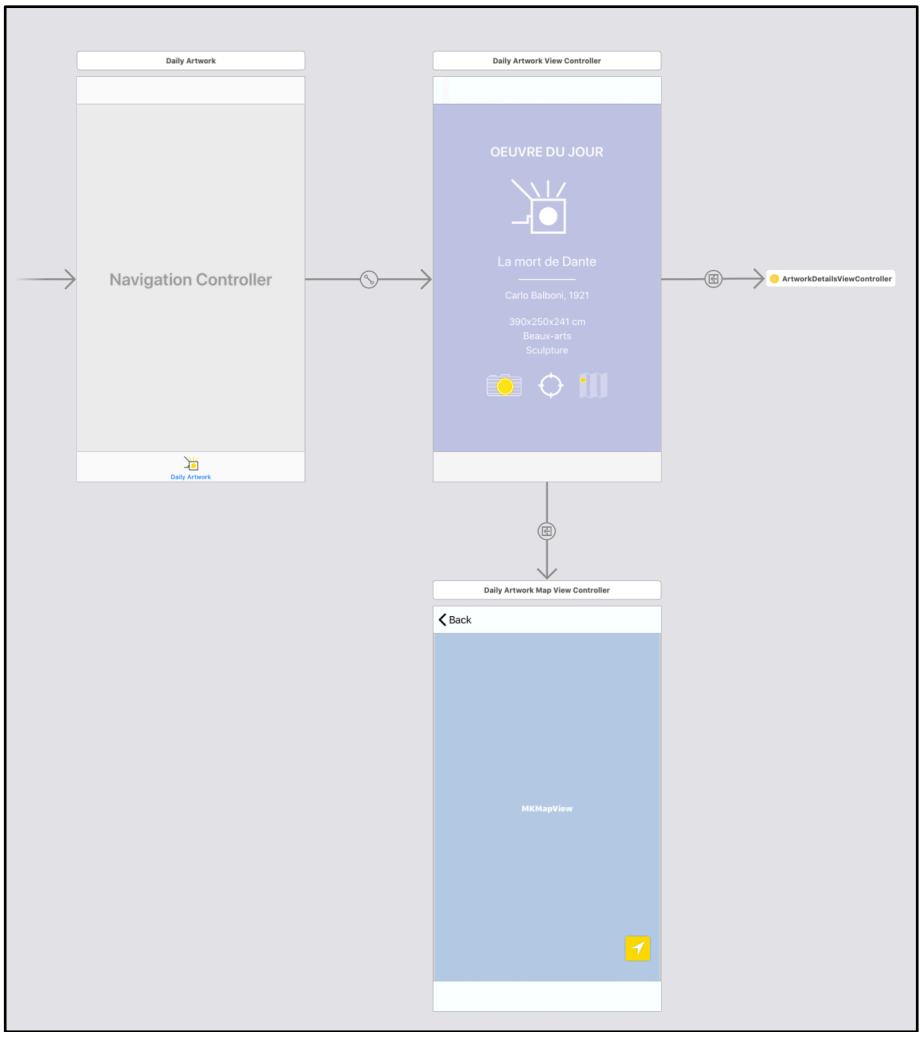


Figure 4.15: « Storyboard » de « Œuvre du jour » (nouvelle version)

Pour permettre une navigation hiérarchisée, la classe « UINavigationController » du cadriciel « UIKit », contrôle un ensemble de vues enfants sur le schéma d'une pile de « First In First Out ». L'utilisateur peut ainsi facilement retourner à une vue précédente à l'aide d'un bouton de retour « Back ».

La classe « DailyArtworkViewController » est la vue racine d'une instance de « UINavigationController » et contrôle l'affichage de l'œuvre du jour, sélectionnée pour le moment de manière aléatoire, avec son titre et sa description. Cette classe contrôle aussi le comportement des boutons comme celui de la prise de photo de l'œuvre, du ciblage de l'œuvre et de l'ouverture de carte. La vue obtenue pour cette implémentation est présentée à l'annexe XVI.

4.4.2.8 La section « Œuvres »

La section « Œuvres » nécessite de nombreuses listes pour être implémentée. La façon la plus commune d'implémenter une vue liste dans iOS est d'utiliser « UITableView » du cadriciel « UIKit ». Une « UITableView » accepte plusieurs prototypes d'élément de liste pour gérer l'affichage des éléments de la liste. Les prototypes suivants ont été créés:

1. « MainTableViewCell »: c'est le prototype (voir Figure 4.16) utilisé pour afficher des noms de liste, comme « Artistes » et « Matériaux » ou pour afficher des noms de catégories et de sous-catégories, « Arts décoratifs » et « Installation », ou des noms d'arrondissement;

Figure 4.16: Le prototype « MainTableViewCell » (nouvelle version)

2. « GeneralTableViewCell »: cette classe permet d'instancier un élément de liste qui affiche un titre et un sous-titre. Ce prototype (voir Figure 4.17) est utilisé lors de l'affichage d'un artiste, d'une catégorie ou d'un arrondissement dans une liste. Par exemple, pour un artiste, le titre serait le nom de l'artiste, et le sous-titre serait le nombre d'œuvres d'art liées à cet artiste. Lorsque l'on clique sur un « GeneralTableViewCell », la liste d'œuvres qui lui est associée s'ouvre;

Figure 4.17: Le prototype « GeneralTableViewCell » (nouvelle version)

3. « ArtworkTableViewCell »: cette classe permet d'instancier un élément de liste qui affiche une vignette, un titre et un sous-titre. Ce prototype (voir Figure 4.1) est utilisé lors de l'affichage d'une œuvre d'art dans une liste, avec comme titre : le titre de l'œuvre et comme sous-titre, l'arrondissement dans lequel est localisée cette œuvre. Lorsque l'on clique sur un élément de liste « ArtworkTableViewCell », l'application ouvre la fiche d'une œuvre « ArtworkDetailsViewController »;

Figure 4.18: Le prototype « ArtworkTableViewCell » (nouvelle version)

Ces trois prototypes d'éléments de liste sont réutilisés dans plusieurs vues de l'application, comme dans l'affichage des résultats d'une recherche, ou dans la liste d'œuvres.

Le « storyboard » de la section « Œuvres » est présenté à la figure 4.19 ci-dessous. Trois classes sont utilisées dans cette section:

1. « MainTableViewController » : gère une liste qui affiche des « MainTableViewCell » pour afficher les noms de catégories par exemple;
2. « ArtworksTableViewController » : gère une liste d'œuvres d'art affichées avec des « ArtworkTableViewCell ». La liste d'œuvres d'art peut être triée de manière ascendante ou descendante selon le titre, la date de production de l'œuvre ou selon la distance entre la position de l'usager et la position de l'œuvre;
3. « GeneralTableViewController » : gère la liste des artistes, des matériaux et des techniques, triée par défaut de manière alphabétique. Lorsque l'on clique sur un élément de cette liste, par exemple un artiste, une « ArtworkTableViewController »

s'ouvre avec les œuvres correspondantes à l'élément choisi, c.-à-d. les œuvres produites par l'artiste dans notre exemple.

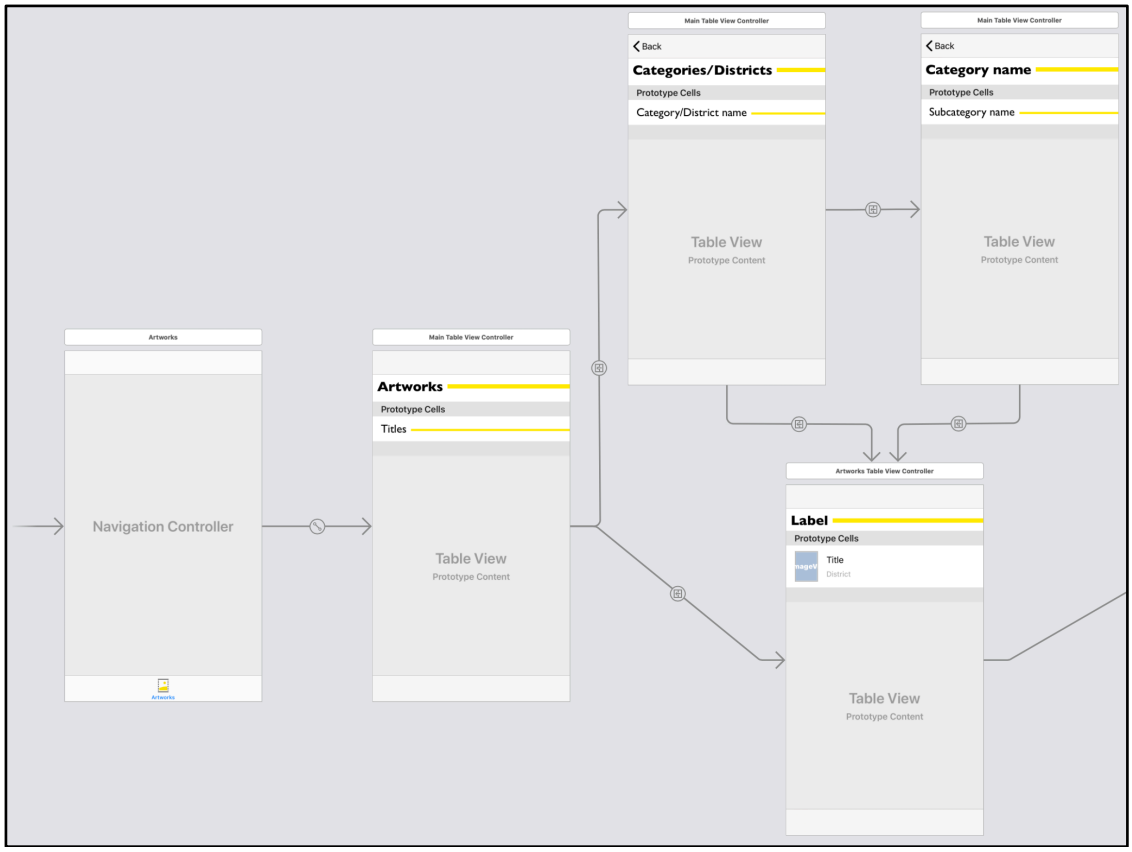


Figure 4.19: « Storyboard » de la section « Œuvres » (nouvelle version)

Comme dans la version précédente de l'application, le parcours de listes très longues, comme la liste des œuvres, est facilité à l'aide d'un index qui a été ajouté à la droite de la liste à l'aide du même package « MYTableViewIndex ».

Le résultat obtenu de l'implémentation des listes est présenté à l'annexe XVII.

4.4.2.9 La fiche d'une œuvre

La fiche de l'œuvre du jour peut être ouverte lorsque l'on clique sur le titre ou la description de l'œuvre. C'est la classe « ArtworkDetailsViewController » qui contrôle l'affichage et le

comportement de la fiche d'une œuvre. Le « storyboard » associé à la fiche d'une œuvre est montré à la figure 4.20 ci-après. La classe « ArtworkDetailsViewController » affiche la photo d'une œuvre si elle est collectionnée. Dans le cas contraire, elle affiche une interface par défaut contenant un bouton de prise de photo, un bouton pour cibler l'œuvre et un bouton d'ouverture de carte. Un « ArtworkDetailsViewController » contient aussi un « UIPageViewController », une classe issue du cadriciel « UIKit », qui permet de contrôler la navigation entre des vues enfants. Ce « UIPageViewController » contrôle les trois vues suivantes :

1. Une vue « ArtworkDetailsDescriptionViewController » qui contient les métadonnées de l'œuvre, comme les noms d'artistes ou la date de réalisation;
2. Une vue « ArtworkDetailsMapViewController » qui permet d'afficher une petite carte avec la localisation de l'œuvre. Un clic sur cette petite carte déclenche l'ouverture d'une grande carte contenant seulement la localisation de cette œuvre;
3. Une vue « ArtworkDetailsCommentRatingViewController » qui permet à un utilisateur d'éditer son commentaire et sa note à propos d'une œuvre.

Le résultat obtenu de cette implémentation est présenté à l'annexe XVIII.

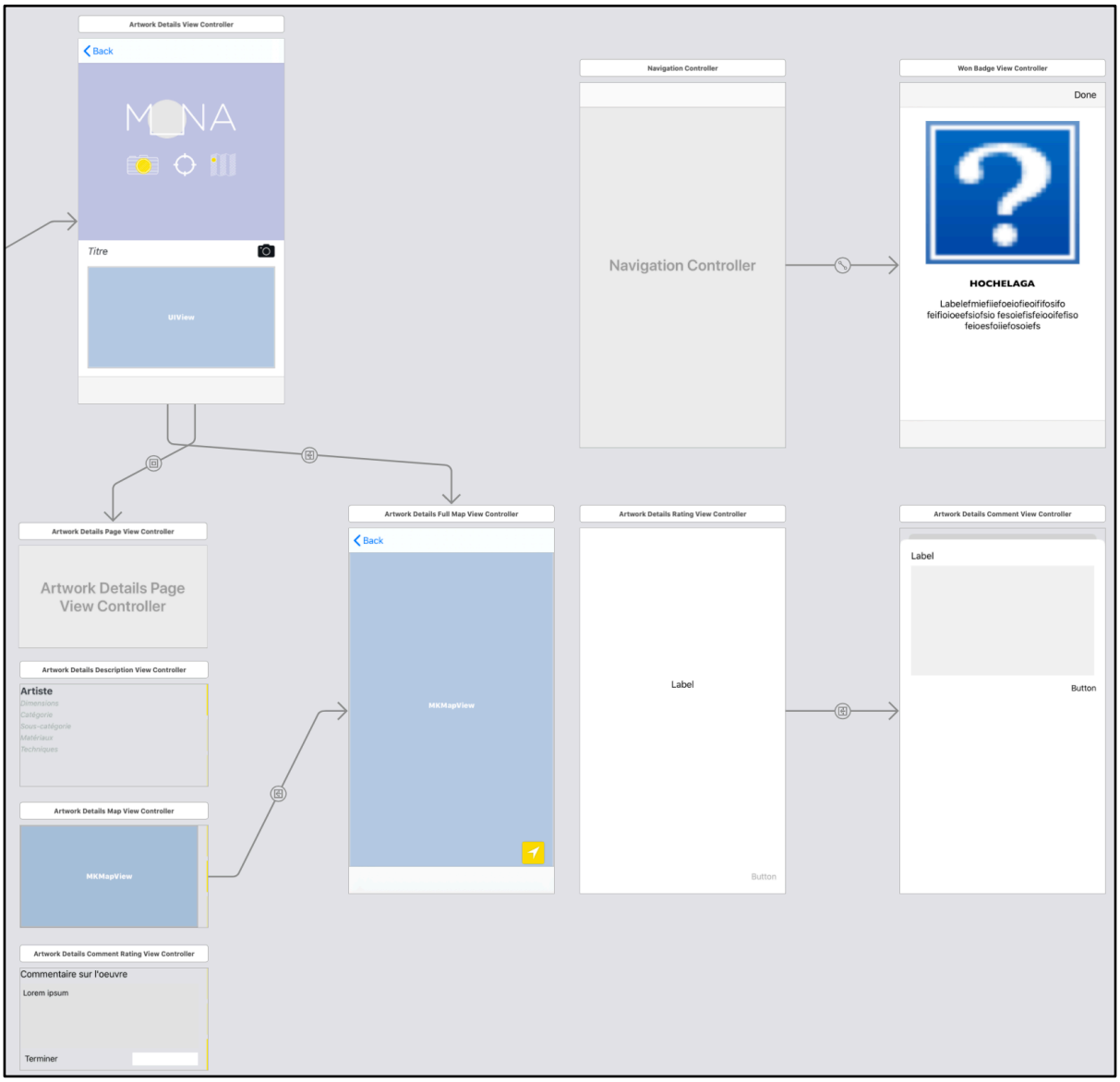


Figure 4.20: « Storyboard » pour la fiche d'une œuvre (nouvelle version)

Le bouton photo permet d'ouvrir l'appareil photo avec une vue « UIImagePickerController ». Cette classe issue du cadriciel « UIKit » fournit une interface simple pour la prise de photo. Après la prise d'une photo, l'application doit la sauvegarder la photo à l'aide du singleton « MonaPhotosAlbum » (voir section 4.4.2.3) et procède à la vérification des badges (voir la section 4.4.2.4 et plus précisément l'algorithme 4.11). Si un ou plusieurs badges sont obtenus, ils sont affichés consécutivement à l'aide du « WonBadgeViewController ». Après l'apparition des badges, un « ArtworkDetailsRatingViewController » est présenté. Il permet à l'utilisateur

d'assigner une note à l'œuvre sur une échelle d'un à cinq. Puis un « ArtworkDetailsCommentViewController » est présenté pour commenter l'œuvre. Après cela, l'utilisateur revient automatiquement à la fiche de l'œuvre et la photo est affichée à la place de l'interface par défaut. La note, le commentaire ainsi que la présentation et les badges obtenus, suite à la prise de photo, sont décrits à l'annexe XIX.

Le bouton en forme de cible de « ArtworkDetailsViewController » ajoute l'œuvre à la liste des œuvres ciblées en mettant sa propriété « isTargeted » à la valeur « true ». Le ciblage d'une œuvre, à partir de sa fiche par défaut (c.-à-d. quand une œuvre est non collectionnée), est aussi présenté à l'annexe XX.

Enfin, le bouton en forme de carte permet l'ouverture d'une carte « ArtworkDetailsFullMapViewController ». Celui-ci ouvre une carte et un bouton situé en bas à droite permet de centrer la vue de la carte sur la position de l'utilisateur. Le fonctionnement de la carte est discuté en détail dans le sous-chapitre suivant (c.-à-d. à la section 4.4.2.10).

4.4.2.10 La section « Carte »

La section « Carte » affiche dans une carte toutes les œuvres d'art à leurs coordonnées géographiques.

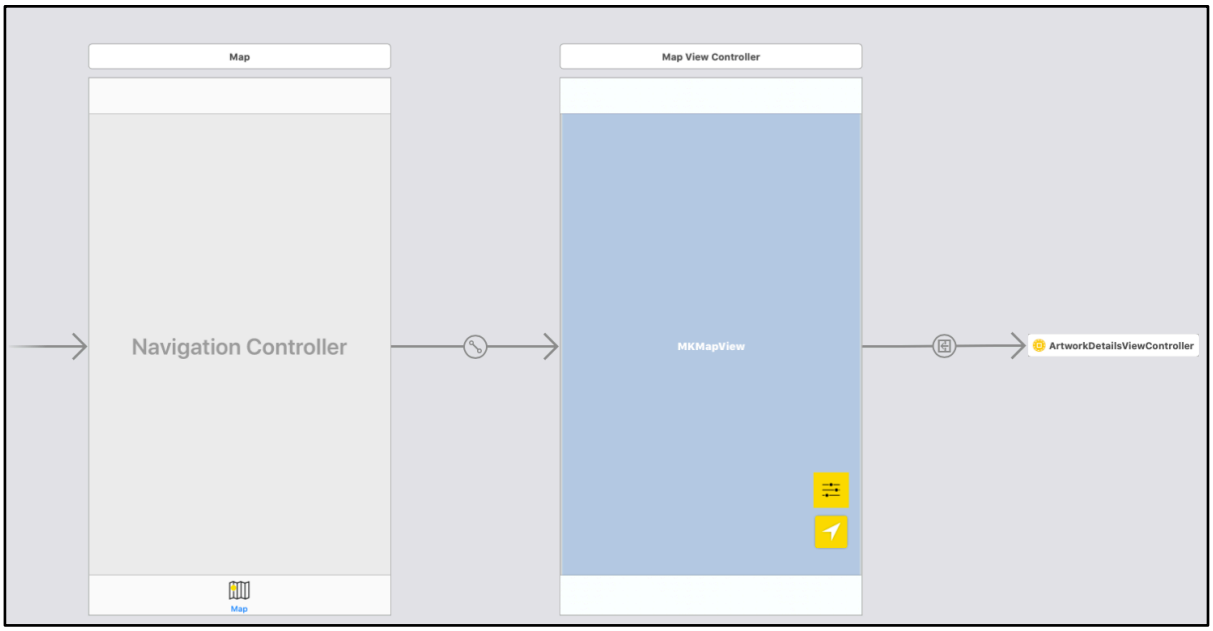


Figure 4.21: « Storyboard » de la section « Carte » (nouvelle version)

Le « storyboard » de la section « Carte » est présenté dans la figure 4.21 ci-dessus. La classe « MapViewController » est utilisée pour gérer l'affichage de la carte et la gestion des boutons de localisation de l'utilisateur et de filtrage des œuvres. Elle s'appuie sur le cadriciel « MapKit » pour afficher une carte « MKMapView ». On peut ajouter à cette carte des annotations (c.-à-d. des icônes qui représentent les œuvres sur la carte) tant que celles-ci sont de type « MKAnnotation ». Pour ce faire, une classe « ArtworkAnnotation » héritant de « MKAnnotation » gère l'affichage des œuvres sur la carte sous la forme d'icônes selon qu'elles soient collectionnées, ciblées ou non collectionnées. Lorsque l'on clique sur une icône représentant une œuvre, une annotation s'ouvre avec le titre de l'œuvre. En cliquant sur l'annotation, l'usager ouvre la fiche d'une œuvre gérée par « ArtworkDetailsViewController ».

« MapViewController » utilise aussi le cadriciel « Core Location » pour localiser l'utilisateur sur la carte. Un bouton permet aussi de centrer la carte sur la position de l'utilisateur.

Enfin, un autre bouton géré par « MapViewController » permet d'ouvrir un popup illustré à la figure 4.22. Ce popup est une vue qui permet de sélectionner ou désélectionner des options de filtrage pour filtrer les œuvres selon qu'elles soient collectionnées, visitées et/ou ciblées. La

classe « MapViewController » gère dynamiquement la mise à jour de la vue « MKMapView » en ajoutant ou supprimant les annotations de la carte au besoin en fonction des options de filtrage choisies.

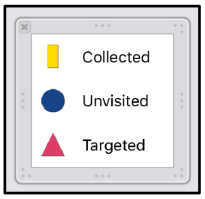


Figure 4.22: Popup affichant des options de filtrage (nouvelle version)

Les vues de la section « Carte » sont présentées à l'annexe XXI.

4.4.2.11 La section « Collection »

Le « storyboard » de la section « Collection » est présenté dans la figure 4.23 ci-après. Les œuvres collectionnées sont affichées dans une instance de « UICollectionView », qui est une classe du cadriciel « UIKit » permettant l'affichage d'une collection d'items. L'agencement de cette collection d'œuvres est contrôlé par une instance de « UICollectionViewLayout » qui permet d'afficher les photos des œuvres collectionnées sous la forme de rangée de quatre photos. Un clic sur une photo de cette collection ouvre la fiche de l'œuvre associée à la photo. L'implémentation de la section « Collection » est présentée à l'annexe XXII.

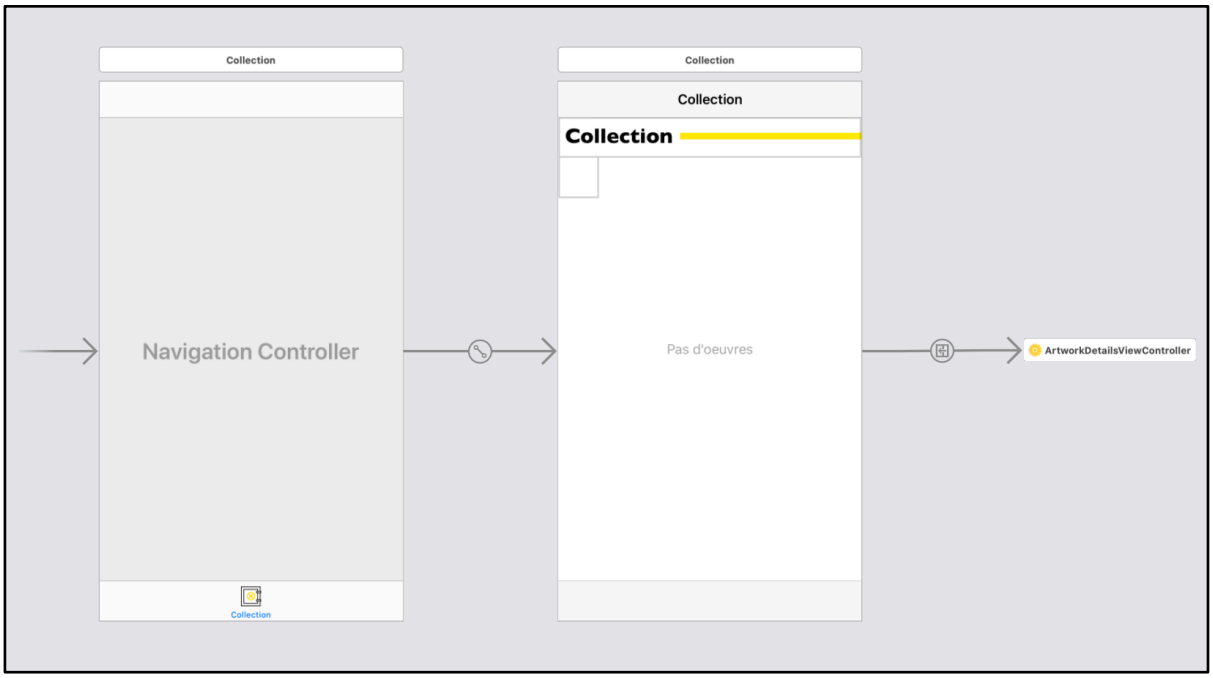


Figure 4.23: « Storyboard » de la section « Collection » (nouvelle version)

4.4.2.12 La section « Plus »

La dernière section de l'application est la section « Plus ». La figure 4.24 ci-après montre le « storyboard » de celle-ci. La classe « MoreTableViewController » contrôle l'affichage et le comportement. Elle contient une vue « UITableView » qui affiche 5 options:

1. L'option « Badges »: la classe « BadgesViewController » contrôle l'affichage et le comportement de la liste des badges. Elle utilise pour cela une vue « UICollectionView » qui affiche les badges par rangées de 2 avec une barre de progression pour collectionner le badge, ainsi qu'un entête affichant le nombre total d'œuvres collectionnées pour chaque catégorie et pour toute l'application. Lorsqu'un utilisateur clique sur un badge, le badge est affiché en grand format avec un texte indiquant le nombre d'œuvres collectionnées pour ce badge (la propriété « currentValue ») et le nombre d'œuvres à collectionner nécessaires à l'obtention du badge (« targetValue »). C'est la classe « BadgeDetailsViewController » qui est responsable de l'affichage au grand format d'un badge;

2. L'option « Ciblées »: la classe « TargetedArtworksTableViewController » affiche les œuvres qui ont été ciblées par l'usager pour les ajouter à cette liste de souhaits. Cette liste utilise une « UITableView » avec le prototype d'élément de liste « ArtworkTableViewCell » pour afficher les œuvres ciblées. Un clic sur élément de cette liste ouvre la fiche d'une œuvre contrôlée par « ArtworkDetailsViewController »;
3. L'option « Confidentialité des données »: cette option ouvre une page qui peut être défilée grâce à la classe « UIScrollView » du cadriciel « UIKit ». Cette page contient un texte qui détaille les conditions auxquelles consent un usager en utilisant l'application en ce qui concerne l'usage de ses données (photo, note et commentaire de l'œuvre en particulier). Ce texte détaille aussi ses droits en la matière;
4. L'option « À propos »: cette dernière option permet d'ouvrir une page « UIScrollView » qui décrit le projet MONA et les objectifs poursuivis par l'application.

La section « Plus » est montrée à l'annexe XXIII.

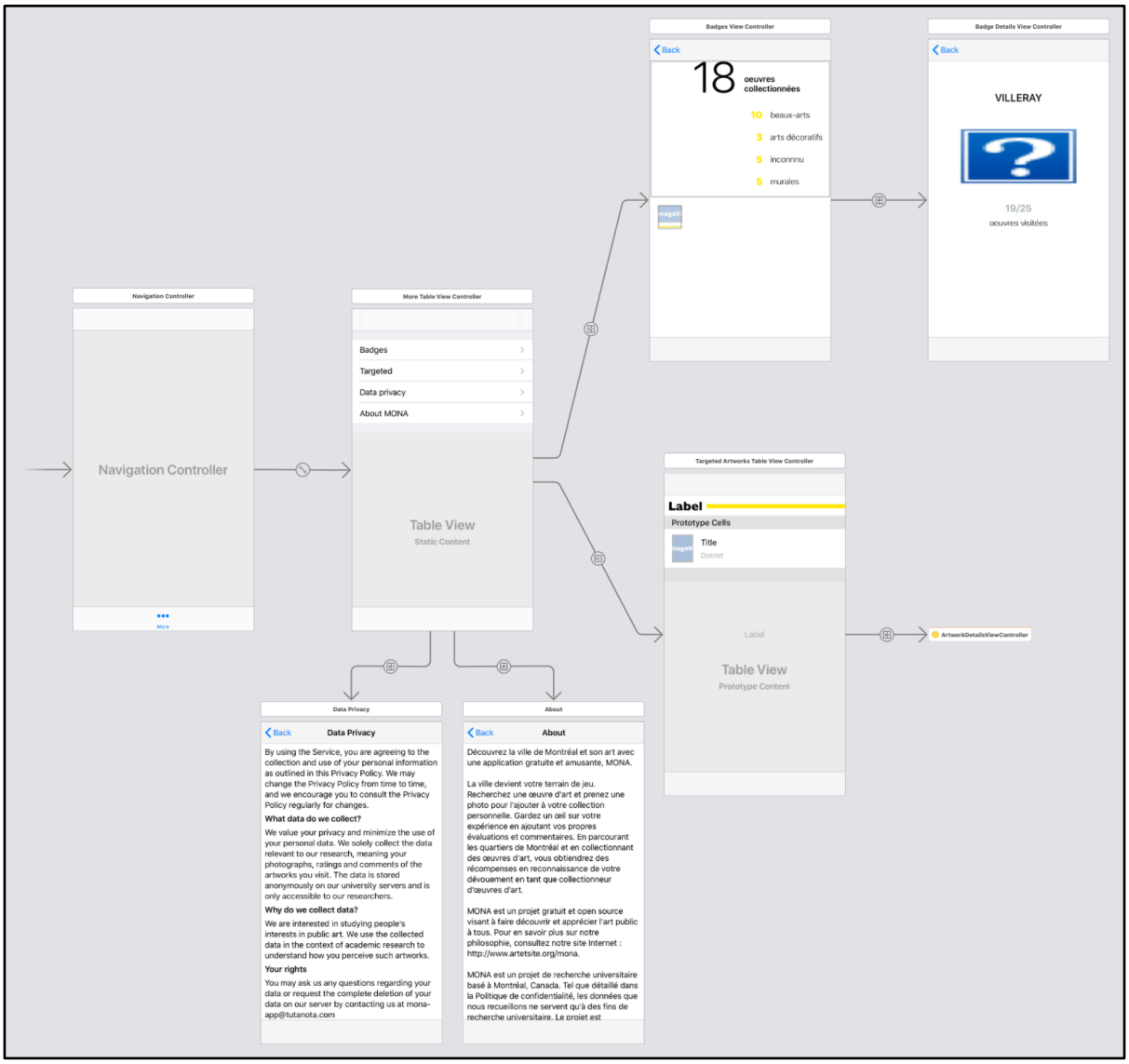


Figure 4.24: « Storyboard » de la section « Plus » (nouvelle version)

4.4.2.13 La recherche

La fonctionnalité de recherche est implémentée avec un « SearchFilterViewController ». Cette classe permet d'afficher un bouton de recherche et un bouton de filtrage, activables ou désactivables au besoin, dans la barre de navigation. Le bouton de recherche permet d'ouvrir l'interface de recherche dans l'application, tandis que le bouton de filtrage est utilisé pour faire

apparaître les options de tri par « Titre », « Date » ou « Distance ». Ces boutons sont tous deux des instances de « UIBarButtonItem », une classe du cadriciel « UIKit » nécessaire pour afficher des boutons dans la barre de navigation.

La classe « SearchFilterViewController » détient une référence sur un « UISearchController » qui affiche un « SearchResultsController » de façon analogue à l'ancienne version de l'application, comme vue dans la section 4.1.8. Comme le montre le « storyboard » de la fonction de recherche dans la figure 4.25 ci-dessous, un clic sur une œuvre ouvre la fiche d'une œuvre « ArtworkDetailsViewController », tandis qu'un clic sur un artiste, une technique, une catégorie ou encore un matériau ouvre la liste d'œuvres correspondante avec la classe « ArtworksTableViewController ».

La partie recherche de l'application est présentée à l'annexe XXIV.

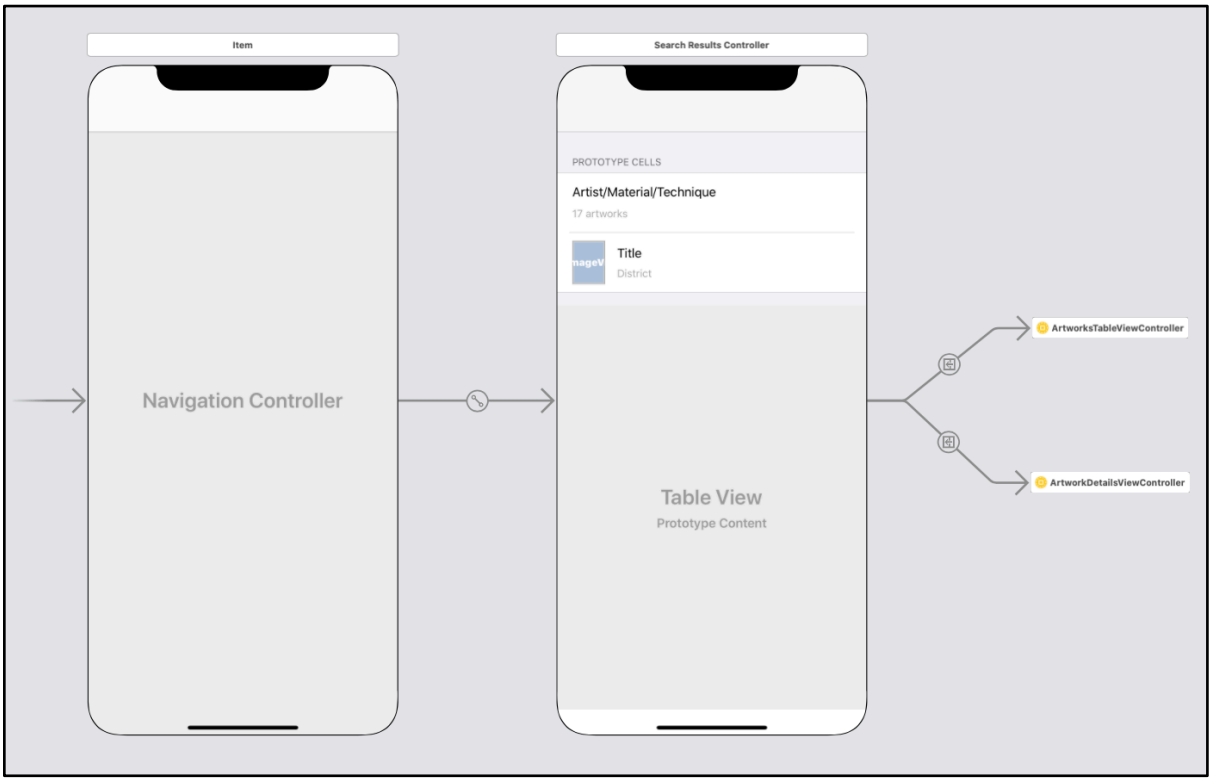


Figure 4.25: « Storyboard » de la fonction de recherche (nouvelle version)

4.4.2.14 Gestion de l'authentification

L'authentification n'a pas été complètement implémentée dans MONA. Au lancement de l'application, on demande à un utilisateur de choisir un nom d'utilisateur (voir Annexe XXV) et un mot de passe est automatiquement généré et sauvegardé dans les préférences avec la classe « UserDefaults ». On utilise ce nom d'utilisateur et ce mot de passe pour obtenir un jeton d'authentification avec une requête HTTP « register » et « login ». Ce jeton d'authentification est alors sauvegardé dans « UserDefaults » et est utilisé pour toutes les requêtes authentifiées vers le serveur Web.

4.4.3 Tests et publication

L'application a été testée tout au long de la phase de développement pour s'assurer de sa qualité autant sur le plan de la conception des interfaces que sur le plan de la performance. Un iPhone 6 avec la version iOS 9.3 a été utilisé pour tester manuellement l'affichage et le comportement de l'application. De plus, le simulateur fourni par Xcode a permis de vérifier que les interfaces conçues s'affichent correctement sur les différentes tailles d'écran d'iPhone disponibles sur le marché. L'application « Instruments », fournie avec Xcode, met à disposition une boîte d'outils permettant de surveiller la performance d'une application iOS. Elle a été utilisée en particulier pour surveiller la performance mémoire (« memory leaks » et objets « zombies ») et la performance énergétique de l'application (consommation de la batterie).

Pour permettre aux développeurs de tester et distribuer leurs applications iOS, Apple propose un coffre d'outils Web qui se nomme App Store Connect. Ce coffre d'outils fournit en particulier l'outil « TestFlight » qui permet aux développeurs d'inviter des usagers, par courriel ou avec un lien de partage, à venir tester une application avant sa sortie officielle sur l'App Store. Pour ce faire, il faut téléverser et soumettre un « build » à la procédure de vérification d'Apple. Cette procédure de vérification, effectuée par Apple et qui est d'une durée d'un à deux jours, vise à vérifier que le « build » soumis sur l'App Store Connect respecte bien les politiques, directives et conditions dictées par Apple (https://developer.apple.com/app-store/review/guidelines). Par exemple, Apple requiert que les applications soient performantes

et respectent certains standards de conception d'interfaces, ainsi que des règles de sécurité et de confidentialité des données. À l'aide de ce processus, « MONA » a été distribuée préalablement en test bêta à une centaine d'utilisateurs, ce qui a permis d'obtenir un certain nombre de commentaires d'utilisateurs concernant des défauts non débusqués par nos essais.

Par ailleurs, une activité de test a été organisée, au début de septembre 2019, avec l'AÉÉHAUM sur le campus universitaire de l'Université de Montréal lors de la journée d'intégration des nouveaux étudiants. Au cours de cette activité, les étudiants ont semblé apprécier le concept et les interfaces de l'application et ont transmis plusieurs suggestions d'améliorations.

Après la réussite de ces activités de tests, qui ont permis de s'assurer de la qualité de l'application, un « build » contenant les derniers correctifs a été soumis à Apple dans le but de distribuer l'application sur l'App Store au Canada seulement (pour le moment). Une seconde procédure de vérification d'une durée d'un à deux jours est nécessaire durant laquelle un employé d'Apple testera réellement l'application pour s'assurer du respect des conditions générales d'utilisation ainsi que de la performance et la pertinence de « MONA ». À la date du 12 septembre 2019, l'application a été acceptée pour être distribuée sur l'App Store et est désormais disponible au Canada (https://apps.apple.com/ca/app/MONA/id1462822498) comme le montre la figure 4.26.

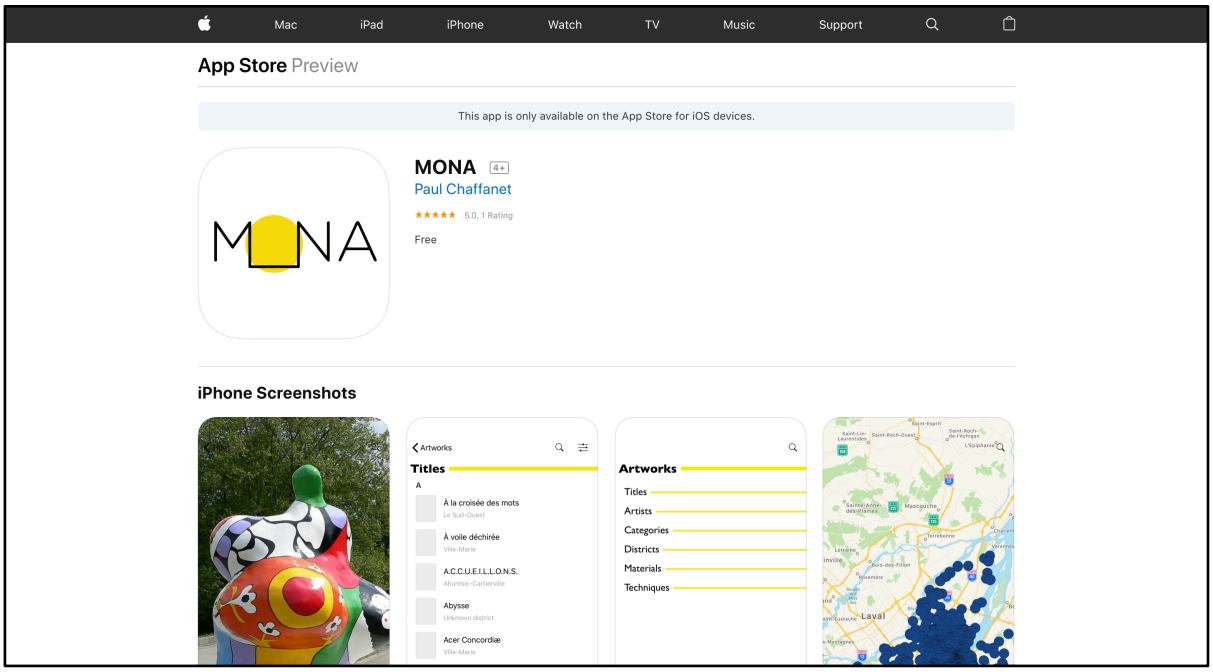


Figure 4.26: L’application MONA publiée sur l’« App Store »

CHAPITRE 5

Résultats

Dans un premier temps, nous verrons quels sont les objectifs qui ont été atteints pour ce projet de recherche appliquée, puis nous verrons les pistes d'amélioration possibles.

5.1 Objectifs atteints

Dans une première partie, nous verrons les objectifs atteints pour le serveur Web, puis nous verrons les objectifs atteints pour l'application.

5.1.1 Le serveur Web

- **Amélioration de la cohérence des données**: la cohérence des données, notamment des techniques et des matériaux, a été améliorée pour permettre un affichage bilingue;
- **Sécurisation des requêtes client-serveur**: le protocole HTTPS et un système d'authentification ont bien été implémentés;
- **Sécurisation des mots de passe**: les mots de passe et les jetons d'authentification sont hachés respectivement avec les fonctions de hachage « bcrypt » et SHA-256 avant d'être stockés;
- **Système d'authentification par jeton**: un système avec jeton a été implémenté pour permettre l'authentification des requêtes HTTP;
- **Interface d'administration**: une interface d'administration fournit un accès facile aux données, comme les photos ou les commentaires des œuvres;
- **Protection contre les attaques informatiques**: la protection contre les injections SQL et les attaques par déni de service est permise par le cadriciel « Laravel » qui fournit une API nommée « Schema » pour interagir de manière sécurisée avec de multiples types de bases de données.

En définitive, tous les objectifs fixés pour le serveur ont été atteints.

5.1.2 L'application

À partir des objectifs qui ont été identifiés à la section 4.3, les objectifs suivants ont été atteints:

- **Solution de persistance des données plus performante**: « Core Data » a été implémenté pour un chargement plus rapide des données. Il en résulte une application plus réactive pour les téléphones moins performants, comme l'iPhone 5;
- **Adapter l'application à la nouvelle API**: la partie réseau de l'application a été refaite avec « URLSession », la classe préconisée par Apple, en permettant d'effectuer des requêtes authentifiées par jeton d'authentification pour le téléchargement et le téléversement de données. Cette nouvelle conception permet de délaisser, au passage, les packages « HermesNetwork » et « Alamofire ». De plus, le décodage des réponses JSON s'effectue avec l'interface « Codable » du cadriciel « Foundation » plutôt que le package « SwiftyJSON ». L'amélioration de la cohérence des données avec la nouvelle API a aussi permis d'afficher la liste des matériaux et des techniques dans un format bilingue. Enfin, les requêtes de téléversement des photos s'effectuent avec succès et peuvent s'afficher sur le serveur dans l'interface d'administration;
- **Nouveau design**: un nouveau design qui s'adapte aux différentes tailles d'écran d'iPhone a été implémenté. D'une part, ce nouveau design améliore la convivialité de l'application en résolvant des problèmes de l'ancien design (comme la difficulté pour les utilisateurs à trouver le bouton de sélection de liste). La vue des « Paramètres » a aussi été supprimée pour un souci de cohérence avec la plateforme iOS. D'autre part, il donne à l'application plus de caractère que l'ancien avec de nouvelles icônes, des images de badges et une carte plus immersive;
- **Photo qui doit être enregistrée dans l'application « Photos »**: la photo s'enregistre dans l'application « Photos » d'iOS dans un souci de cohérence avec la plateforme iOS;
- **Section « Œuvre du jour »**: une section « Œuvre du jour » a été conçue puis implémentée;
- **Section « Collection »**: la section « Collection » a été retravaillée pour proposer une galerie des photos d'œuvres collectionnées;

- **Récompense par badge**: le système de récompense par badge a été implémenté avec une vue qui permet de voir tous les badges existants dans l'application, et des vues qui présentent les badges au moment où ceux-ci sont collectionnés;
- **Assurer une bonne performance générale de l'application**: la performance de l'application a été observée avec l'outil « Instruments » afin de repérer et réparer les goulots d'étranglement. L'utilisation de « Core Data » a permis un chargement plus rapide des données;
- **Améliorer la maintenabilité du code**: la nouvelle version de l'application réduit sa dépendance au code externe en délaissant trois « packages » : HermesNetwork », « Alamofire » et « SwiftyJSON »;
- **Publication de l'application sur l'App Store**: l'application a été rendue disponible sur l'App Store, après avoir réussi le processus de vérification d'Apple.

Pour autant, tous les objectifs n'ont pu être atteints:

- **Système de gestion de l'authentification**: l'authentification a été implémentée de manière partielle. Le jeton d'authentification de l'API est obtenu, pour le moment, avec un mot de passe généré automatiquement dans l'application;
- **Partage des photos**: le partage des photos sur les réseaux sociaux n'est pas possible au sein de l'application. Mais il est toujours possible de partager la photo d'une œuvre à partir de l'application « Photos » qui dispose de fonctionnalités de partage;
- **Notifications**: pas de notifications pour le moment;
- **Mise en place de tests automatisés**: les tests unitaires n'ont pas été implémentés;
- **Documentation**: le code est abondamment commenté, mais certaines fonctions ne sont toujours pas documentées.

5.2 Pistes d'amélioration

Dans cette partie, nous verrons quelles sont les pistes d'amélioration à explorer pour améliorer le serveur Web et l'application.

5.2.1 Le serveur Web

Le serveur Web peut encore être amélioré sur les points suivants:

- Effectuer la traduction en version française de l'interface d'administration;
- Améliorer la gestion de l'authentification en permettant à l'utilisateur de modifier un mot de passe. De plus, un système de gestion de comptes reliés à des réseaux sociaux pourrait être implémenté avec « Laravel Passport » et « Laravel Socialite »;
- Implémenter une validation plus robuste des requêtes vers l'API;
- Fixer une fréquence limite de requêtes pour se protéger des attaques par déni de service;
- Mettre en place des tests unitaires.

5.2.2 L'application

L'application peut encore être améliorée sur les points suivants:

- L'application devrait permettre à un utilisateur de se créer un compte avec mot de passe plutôt que de générer un mot de passe automatiquement. De plus, le jeton d'authentification et le mot de passe devraient être sauvegardés avec le cadriciel « Keychain Services »;
- Ajout des fonctionnalités de partage sur les réseaux sociaux dans l'application, même si celles-ci sont disponibles dans l'application « Photos » d'iOS;
- Notifier l'utilisateur lorsqu'il passe près d'une œuvre;
- Améliorer la précision de la localisation d'un utilisateur. En effet, l'iPhone est très imprécis dans les bâtiments. Bluetooth 5.1 serait une piste à explorer pour résoudre ce problème;
- De nombreux utilisateurs de l'application nous ont dit qu'il arrive que des œuvres localisées dans l'application ne se trouvent pas à la position indiquée. Il faudrait mettre en place un système collaboratif, où les utilisateurs peuvent signaler à d'autres utilisateurs qu'une œuvre ne se trouve pas à la position indiquée;
- Certains utilisateurs nous ont réclamé la possibilité que l'application propose des circuits de visite dans la ville;

- D’autres utilisateurs auraient aimé pouvoir ajouter de nouvelles œuvres à leur collection, même quand celles-ci ne sont pas localisées dans l’application, en créant par exemple leur propre fiche d’une œuvre;
- Des tests automatisés devraient être mis en place pour améliorer la robustesse de l’application;
- Une documentation devrait être faite pour faciliter la maintenabilité de l’application.

CONCLUSION

L'objectif principal de l'application MONA, développée pour la plateforme iOS, est de permettre à un utilisateur d'explorer la collection d'art public de la ville de Montréal grâce à leur géolocalisation, et de créer sa propre galerie d'art en photographiant ces œuvres. L'utilisateur peut exprimer son opinion, par une note et un commentaire, sur chaque œuvre, et est incité à poursuivre son exploration avec un système de récompense avec badges. Pour MONA, le but à atteindre est l'éveil de la curiosité des utilisateurs pour la richesse de l'art public montréalais en favorisant un processus d'exploration, de découverte et de partage.

Une première version non publiée sur l'App Store avait été développée pour l'application. Elle présentait des problèmes de désign et de performance, et ne remplissait pas tous les besoins spécifiés par les utilisateurs. Cette première version faisait appel aux services d'un serveur Web avec une API qui posait plusieurs problèmes de sécurité et de cohérence des données, et qui ne proposait pas d'interface graphique d'administration facile d'accès.

La nouvelle version du serveur Web permet d'améliorer la sécurité et la maintenabilité du serveur avec l'utilisation du cadriciel « Laravel », tout en présentant une plus grande cohérence des données. Cette nouvelle version de l'application implémente un tout nouveau désign qui avait été préalablement prototypé. Elle est plus performante, a une meilleure gestion de la persistance des données à l'aide de « Core Data », une plus grande cohérence de la gestion des photos et des paramètres, et une plus grande maintenabilité en réduisant sa dépendance aux « packages ». Enfin, l'application a réussi le processus de vérification effectué par Apple pour être rendue disponible sur l'App Store. Toutefois, certains besoins n'ont pas pu être remplis, ou ne l'ont été que partiellement, comme le système de gestion de l'authentification et le système de partage des photos des œuvres. D'autres besoins se sont aussi ajoutés en cours de développement, comme la possibilité pour un utilisateur de créer la fiche d'une œuvre. Les besoins remplis partiellement et les nouveaux besoins constituent autant de pistes d'amélioration et de développement pour les versions futures de MONA.

ANNEXE I

Une œuvre au format JSON (ancienne version)


```
{
   "id": 0,
   "Titre": "Source",
   "TitreVar": null,
   "ficher": null,
   "Categorie": "Beaux-arts",
   "CategorieANG": "Fine Arts",
   "SousCategorie": "Sculpture",
   "SousCategorieANG": "Sculpture",
   "Date": "/Date(1291352400000-0500)/",
   "Materiaux": [
      {
         "Nom": "bronze",
         "NomANG": "bronze "
      }
   ],
   "Technique": [
      {
         "Nom": "bronze coulé",
         "NomANG": "cast bronze"
      },
      {
         "Nom": "boulonné",
         "NomANG": "bolted"
      }
   ],
   "Dimension": "549 x 466 x 466 cm",
   "Arrondissement": "Côte-des-Neiges-Notre-Dame-de-Grâce",
   "Latitude": "45.466405",
   "Longitude": "-73.631648",
   "Artiste": [
      {
         "ID": 960,
         "Prenom": "Patrick",
         "Nom": "Coutu",
         "NomCollectif": null
      }
   ]
}
```

ANNEXE II

Algorithme de stockage de photo sur le serveur Web (ancienne version)

```
	function addPicture($conn,$username,$password){
		if(isset($_FILES['file']) and (isset($_GET['IDOeuvre']) or
isset($_POST['IDOeuvre']))){
			if($_SERVER['REQUEST_METHOD'] === 'GET'){
				$IDOeuvre = $_GET['IDOeuvre'];
			} else {
				$IDOeuvre = $_POST['IDOeuvre'];
			}
			$file = $_FILES['file'];
			$fileName=$file['name'];
			$fileTmpName=$file['tmp_name'];
			$fileSize=$file['size'];
			$fileError=$file['error'];
			$fileType=$file['type'];
			$fileExt=explode('.',$fileName);
			$fileActualExt = strtolower(end($fileExt));
			$allowed = array('jpg','jpeg','png');
			if(in_array($fileActualExt,$allowed)){
				if($fileError === 0){
					$fileNameNew =
$username."_".$IDOeuvre.".".$fileActualExt;
					$fileDestination = 'uploads/'.$fileNameNew;
					move_uploaded_file($fileTmpName,
$fileDestination);
					chmod($fileDestination, 0755);
					$req = "INSERT INTO `Pictures` (`pictureName`,
`UserName`, `IDOeuvre`)
						VALUES ('$fileNameNew', '$username',
'$IDOeuvre');";
					$res = $conn->query($req);
					if($res) return returnJson(true,null);
					else return returnJson(true,"already added to
database");
				} else {
					return returnJson(false,"uploading error");
				}
			} else {
				return returnJson(false,"invalid picture extension");
			}
		} else {
			return returnJson(false,"invalid upload - file not
present");
		}
	}
```

ANNEXE III

Création des tables de la base de données sur le serveur Web (ancienne version)

```
CREATE TABLE `Users` (
  `UserName` varchar(50) NOT NULL,
  `PassWord` varchar(50) NOT NULL,
  `LastLog` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`UserName`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 ;

CREATE TABLE `Oeuvres` (
  `ID` int(11) NOT NULL,
  `Nom` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 ;

CREATE TABLE `Critics` (
  `IDOeuvre` int(11) NOT NULL,
  `UserName` varchar(50) NOT NULL,
  `Note` int(11) DEFAULT NULL,
  `Comment` varchar(255) CHARACTER SET utf8 DEFAULT NULL,
  `Date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`IDOeuvre`,`UserName`,`Date`),
  KEY `UserName` (`UserName`),
  CONSTRAINT `Critics_ibfk_1` FOREIGN KEY (`UserName`) REFERENCES `Users`
(`UserName`),
  CONSTRAINT `Critics_ibfk_2` FOREIGN KEY (`IDOeuvre`) REFERENCES `Oeuvres`
(`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE `Pictures` (
  `pictureName` varchar(255) NOT NULL,
  `UserName` varchar(50) NOT NULL,
  `IDOeuvre` int(11) NOT NULL,
  PRIMARY KEY (`pictureName`),
  KEY `UserName` (`UserName`,`IDOeuvre`),
  KEY `UserName_2` (`UserName`,`IDOeuvre`),
  KEY `IDOeuvre` (`IDOeuvre`),
  CONSTRAINT `Pictures_ibfk_1` FOREIGN KEY (`IDOeuvre`) REFERENCES `Oeuvres`
(`ID`),
  CONSTRAINT `Pictures_ibfk_2` FOREIGN KEY (`UserName`) REFERENCES `Users`
(`UserName`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 ;
```

ANNEXE IV

Algorithme de création de la table d'œuvres d'art du serveur Web (nouvelle version)

```php
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateArtworksTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('artworks', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('title')->nullable();
            $table->date('produced_at')->nullable();
            $table->unsignedBigInteger('category_id');
            $table->unsignedBigInteger('subcategory_id')->nullable();
            $table->json('dimensions');
            $table->unsignedBigInteger('borough_id');
            $table->point('location');
            $table->timestamps();

            $table->foreign('category_id')
                  ->references('id')->on('categories');
            $table->foreign('subcategory_id')
                  ->references('id')->on('subcategories');
            $table->foreign('borough_id')
                  ->references('id')->on('boroughs');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('artworks');
    }
}
```

ANNEXE V

Algorithme de stockage de photo, de note et de commentaire du serveur Web (nouvelle version)

```php
namespace App\Http\Controllers\V2\User;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Validator;

class ArtworkController extends Controller
{
    /**
     * Get a validator for an incoming request.
     *
     * @param  array  $data
     * @return \Illuminate\Contracts\Validation\Validator
     */
    protected function validator(array $data)
    {
        return Validator::make($data, [
            /* 'id' => ['required', 'integer'], */
            'rating' => ['integer', 'max:5'],
            'comment' => ['string', 'max:2048'],
            'photo' => ['image', 'max:4096'],
        ]);
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  Request  $request
     * @return void
     */
    public function store(Request $request)
    {
        $this->validator($request->all())->validate();

        $attrs = [];
        if ($rating = $request->rating)
            $attrs['rating'] = $rating;
        if ($comment = $request->comment)
            $attrs['comment'] = $comment;
        if ($photo = $request->file('photo'))
            $attrs['photo'] = $photo->store('public/photos');

        $artworks = Auth::user()->artworks();
        if ($artworks->find($request->id)) {
            $artworks->updateExistingPivot($request->id, $attrs);
        } else {
            $artworks->syncWithoutDetaching([$request->id => $attrs]);
        }
    }
}
```

ANNEXE VI

La classe « Coordinate » (ancienne version)

```
class Coordinate : NSObject, NSCoding {

    //MARK: - Properties
    let latitude : Double
    let longitude : Double

    //MARK: - Initializers
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }

    //MARK: - Types
    struct PropertyKey {
        static let latitude = "latitude"
        static let longitude = "longitude"
    }

    //MARK: - NSCoding
    func encode(with aCoder: NSCoder) {
        aCoder.encode(latitude, forKey: PropertyKey.latitude)
        aCoder.encode(longitude, forKey: PropertyKey.longitude)
    }

    required convenience init?(coder aDecoder: NSCoder) {
        let latitude = aDecoder.decodeDouble(forKey: PropertyKey.latitude)
        let longitude = aDecoder.decodeDouble(forKey: PropertyKey.longitude)
        self.init(latitude: latitude, longitude: longitude)
    }
}
```

ANNEXE VII

Initialisation de « DataManager » (ancienne version)

```
init?() {
        fileManager = FileManager()
        documentsDirectory = fileManager.urls(for: .documentDirectory, in:
.userDomainMask).first!
        searchesFile = documentsDirectory.appendingPathComponent("searches")
        loginFile = documentsDirectory.appendingPathComponent("login")
        artworksFile = documentsDirectory.appendingPathComponent("artworks")
        badgesFile = documentsDirectory.appendingPathComponent("badges")
        guard let artworksFolderUrl = DataManager.createFolder(fileManager:
fileManager, documentsDirectory: documentsDirectory, folderName:
"cachedArtworks")
            else {
                log.error("Unable to create artworksFolder.")
                return nil
        }

        artworksFolder = artworksFolderUrl

        guard let badgesFolderUrl = DataManager.createFolder(fileManager:
fileManager, documentsDirectory: documentsDirectory, folderName: "cachedBadges")
            else {
                log.error("Unable to create artworksFolder.")
                return nil
        }

        badgesFolder = badgesFolderUrl

        guard let imagesFolderUrl = DataManager.createFolder(fileManager:
fileManager, documentsDirectory: documentsDirectory, folderName: "images")
            else {
            log.error("Unable to create imagesFolder.")
            return nil
        }
        imagesFolder = imagesFolderUrl

        guard let thumbnailsFolderUrl = DataManager.createFolder(fileManager:
fileManager, documentsDirectory: documentsDirectory, folderName: "thumbnails")
else {
            log.error("Unable to create thumbnailsFolder.")
            return nil
        }
        thumbnailsFolder = thumbnailsFolderUrl
}
```

ANNEXE VIII

Requête pour stocker une photo sur le serveur (ancienne version)

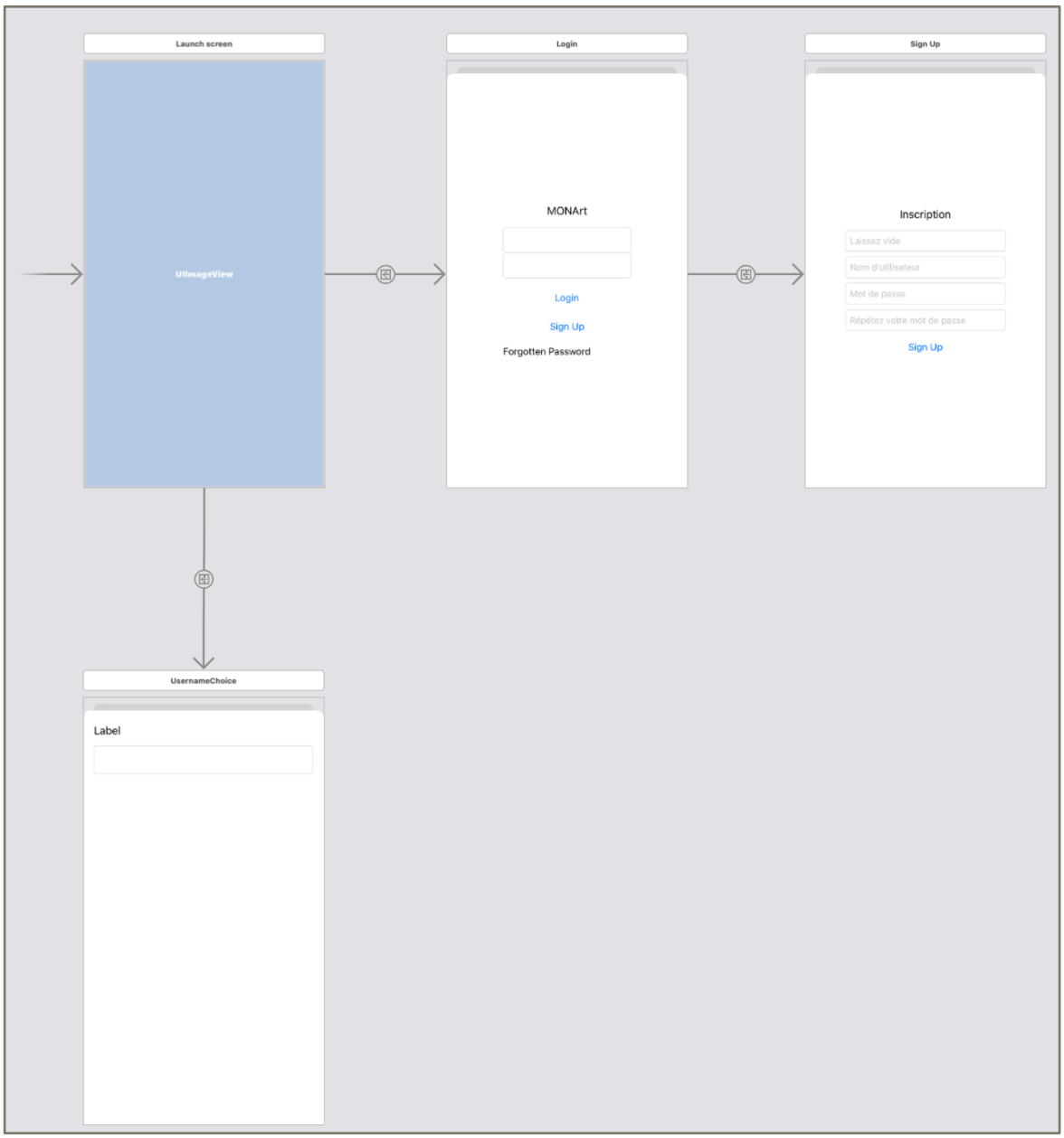
```
import Alamofire

class AddPhotoOperation {

    init(withUsername username: String, withPassword password: String, artwork:
Artwork) {


        let parameters = ["request" : "addPicture", "username" : username,
"password" : password, "IDOeuvre": "\(artwork.id)"]
        let imgData =
UIImageJPEGRepresentation(dataManager.loadImage(forArtwork: artwork)!, 0.8)!

        Alamofire.upload(multipartFormData: { multipartFormData in
            multipartFormData.append(imgData, withName: "file",fileName:
"file.jpg", mimeType: "image/jpg")
            for (key, value) in parameters {
                multipartFormData.append(value.data(using:
String.Encoding.utf8)!, withName: key)
            } //Optional for extra parameters
        },
        to:"http://www-etud.iro.umontreal.ca/~beaurevg/ift3150/server/")
        { (result) in
            switch result {
            case .success(let upload, _, _):
                upload.responseJSON { response in
                    log.info("Request: addPicture, Username: \(username),
ArtworkID: \(artwork.id), Image size: \(Double(NSData(data: imgData).length) /
1024.0) KB")
                    if response.value != nil {
                        artwork.photoSent = true
                        dataManager.saveCachedArtwork(artwork)
                    }
                    log.info("Response: \(response.value ?? "No Value")")
                }

            case .failure(let encodingError):
                log.error(encodingError)
            }
        }
    }
}
```

ANNEXE IX

Fiche d'une œuvre (ancienne version)

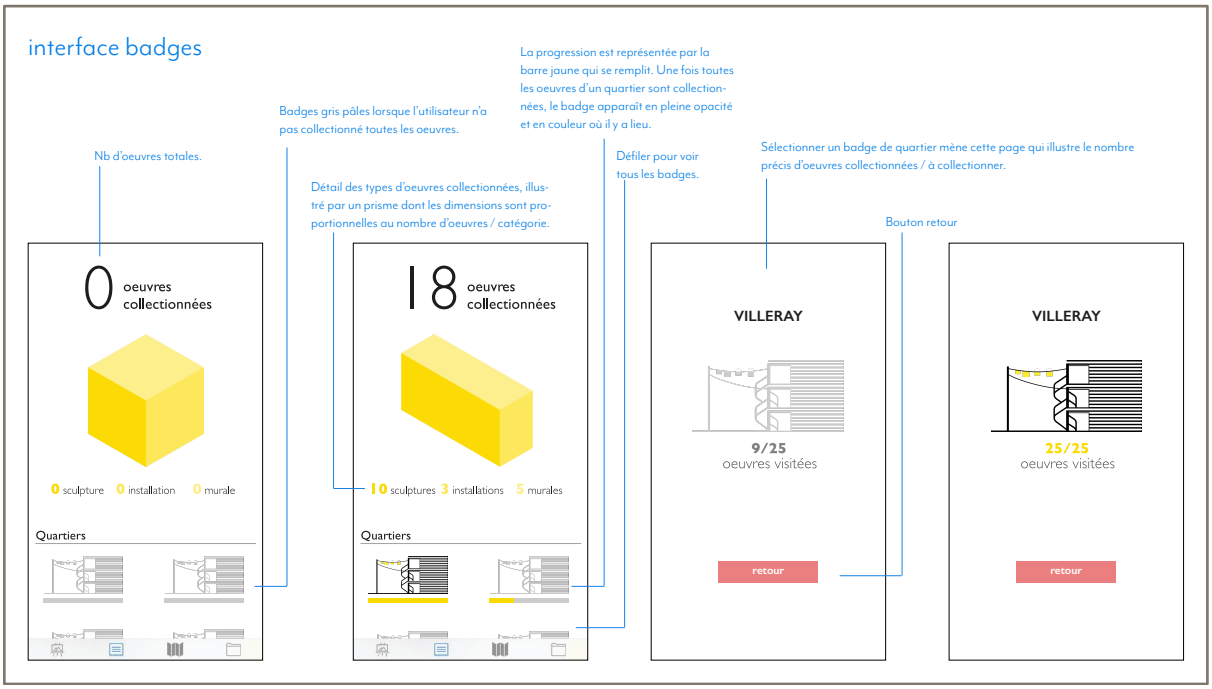
ANNEXE X

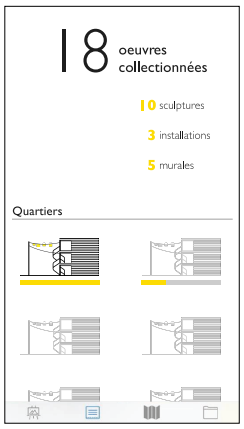
« Storyboard » pour l'authentification (ancienne version)



ANNEXE XI

Prototype de l'interface des badges

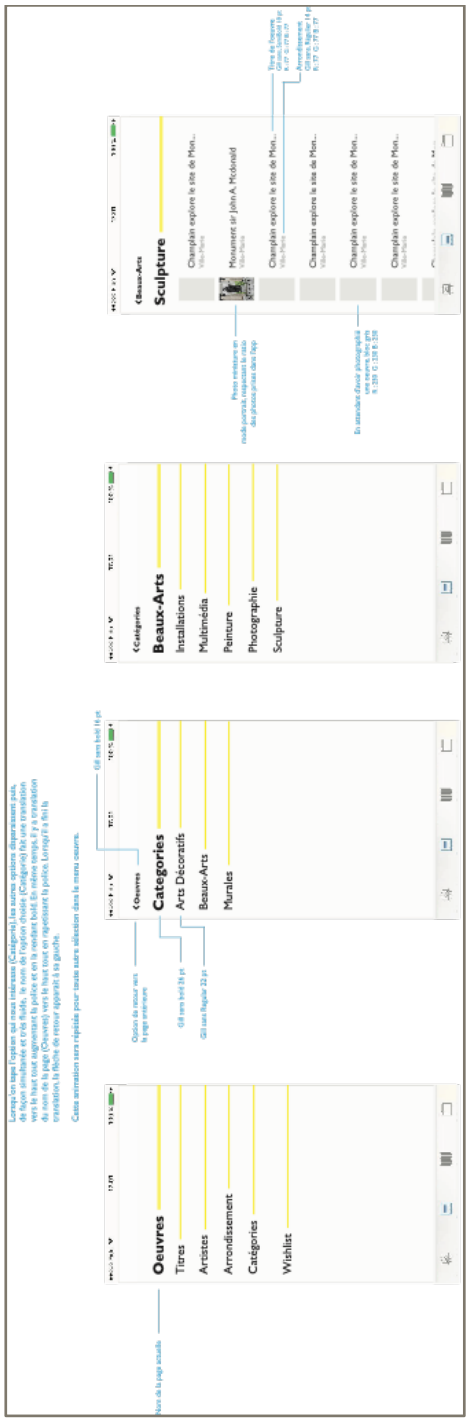




Version simplifiée.
Les mots «sculptures», «installations» et «murales» seraient éventuellement à remplacer par des icônes représentatifs

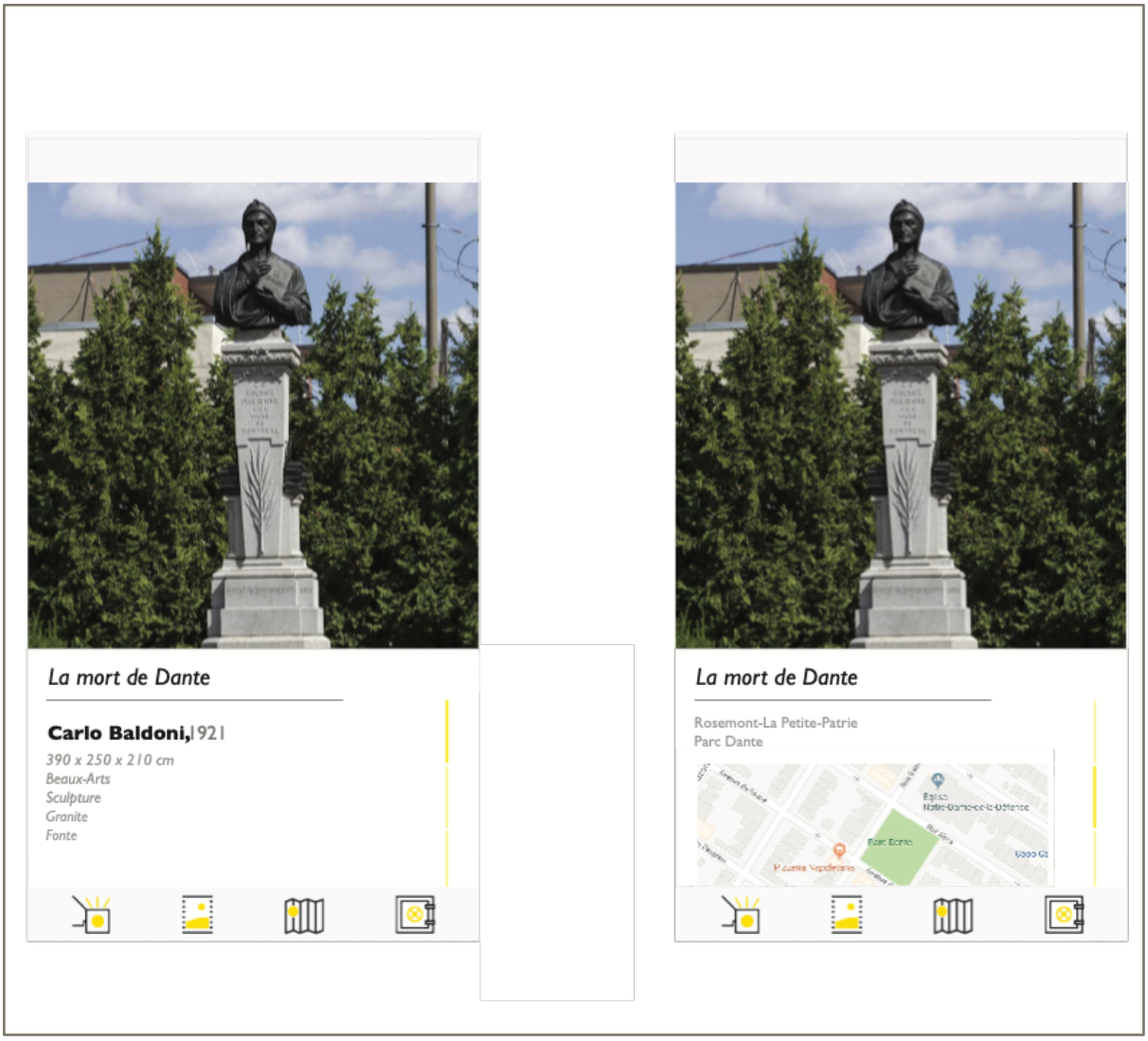
ANNEXE XII

Prototype de l'interface des listes dans la section « Œuvres »



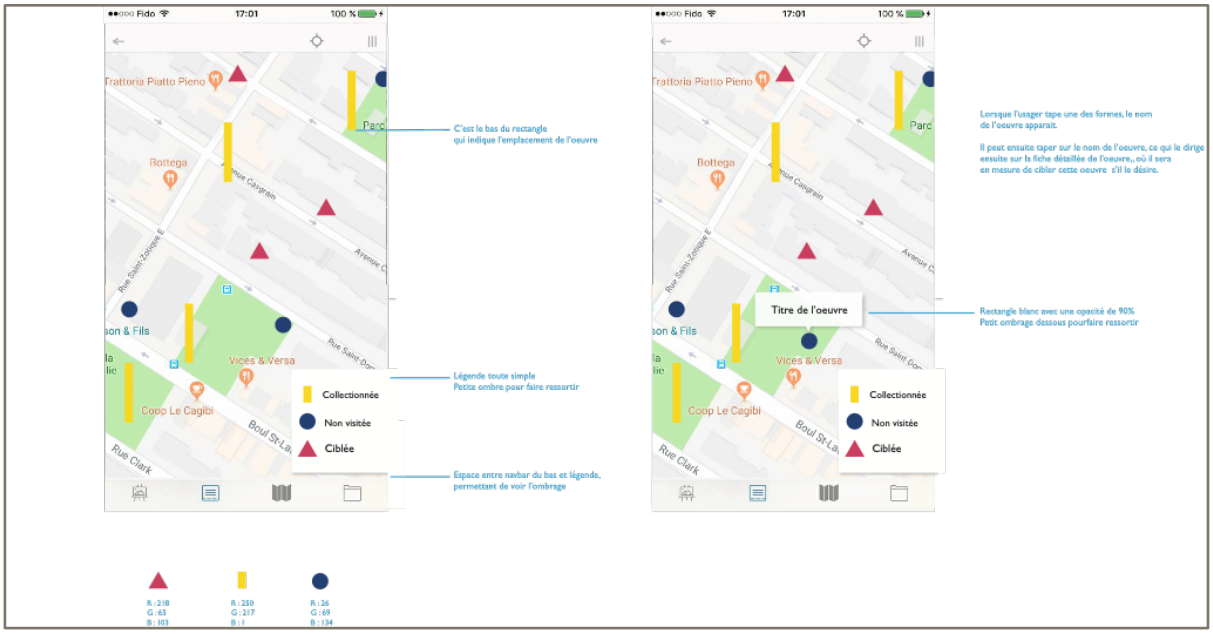
ANNEXE XIII

Prototype de l'interface de la fiche d'une œuvre



ANNEXE XIV

Prototype de l'interface de la section « Carte »



ANNEXE XV

« Parsing » du fichier JSON (nouvelle version)

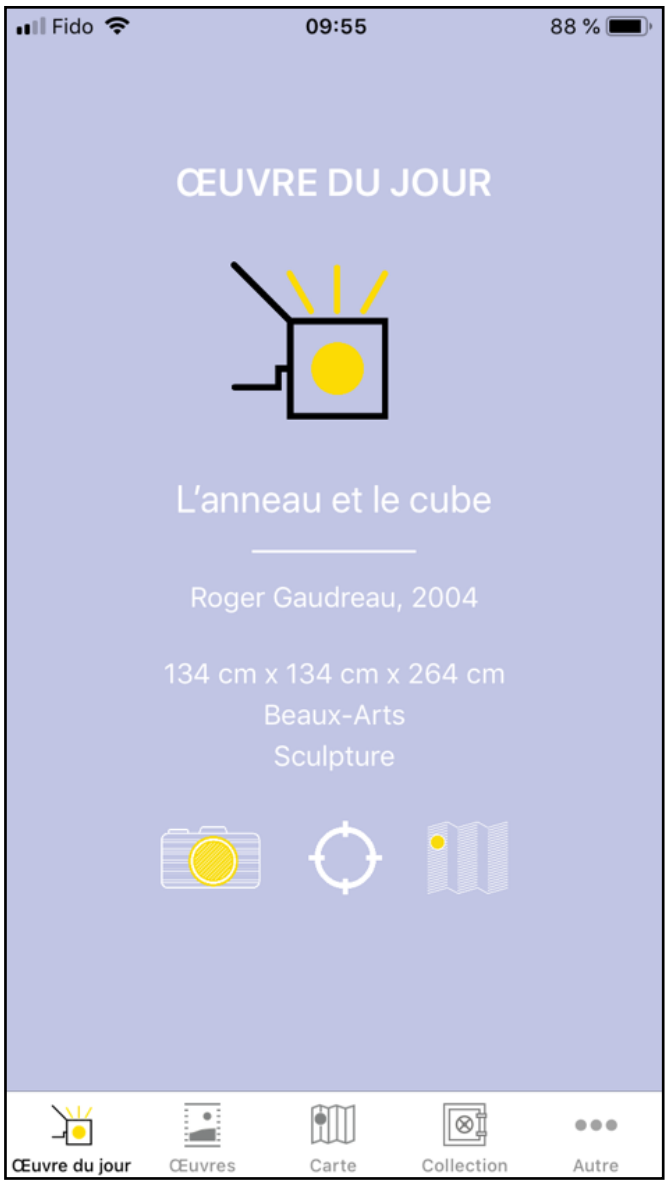
```
struct ArtworkDecodableResponse: Codable {

    enum CodingKeys : String, CodingKey {
        case data
    }

    struct Artwork : Codable {

        enum CodingKeys : String, CodingKey {
            case id
            case title
            case date = "produced_at"
            case category
            case subcategory
            case dimensions
            case materials
            case techniques
            case district = "borough"
            case artists
            case coordinate = "location"
        }

        struct Artist : Codable {

            enum CodingKeys : String, CodingKey {
                case id
                case name
                case isCollectiveName = "collective"
            }

            let id: Int16
            let name: String
            let isCollectiveName : Bool

        }

        struct Category : Codable {

            enum CodingKeys : String, CodingKey {
                case fr
                case en
            }

            var fr: String?
            var en: String?

        }
```

```
        struct Coordinate : Codable {

            enum CodingKeys : String, CodingKey {
                case latitude = "lat"
                case longitude = "lng"
            }

            let latitude: Double?
            let longitude: Double?

        }

        struct Subcategory : Codable {

            enum CodingKeys : String, CodingKey {
                case fr
                case en
            }

            var fr: String?
            var en: String?

        }

        struct Material : Codable {

            enum CodingKeys : String, CodingKey {
                case fr
                case en
            }

            let fr: String?
            let en: String?

        }

        struct Technique : Codable {

            enum CodingKeys : String, CodingKey {
                case fr
                case en
            }

            let fr: String?
            let en: String?

        }

        let id : Int16
        let title : String?
        let date : Date?
        let artists : [Artist]?
        var category : Category?
        var subcategory : Subcategory?
        let dimensions : [String]?
        let materials : [Material]?
        let techniques : [Technique]?
        var district : String?
        let coordinate : Coordinate?
    }
```

```
        var data : [Artwork]?
    }
```

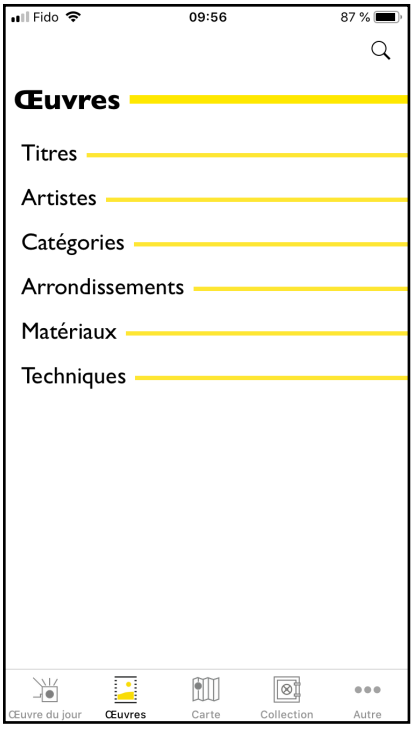
ANNEXE XVI

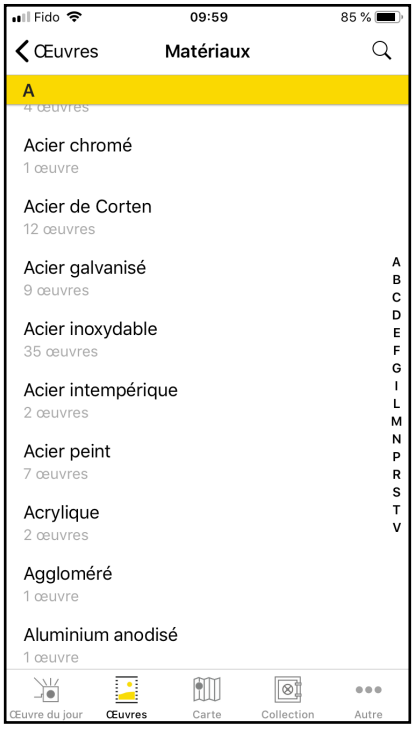
Vue de la section « Œuvre du jour » (nouvelle version)

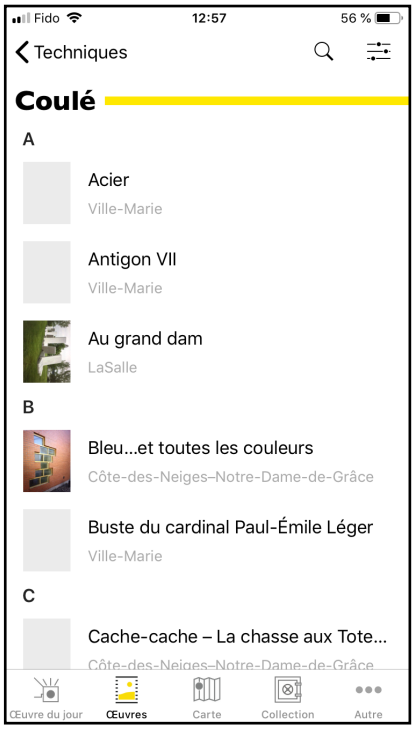


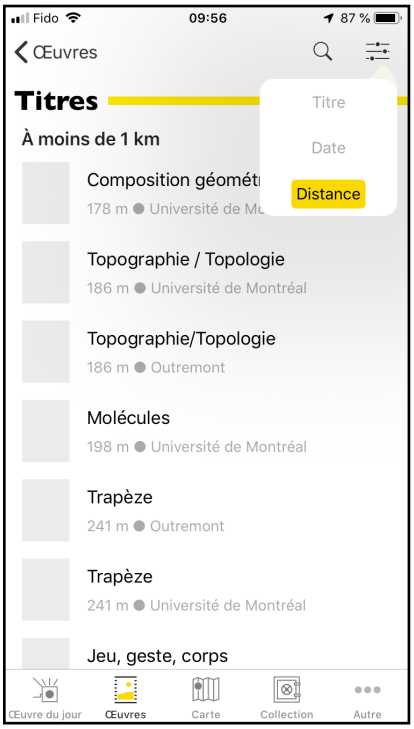
ANNEXE XVII

Vues de la section « Œuvres » (nouvelle version)



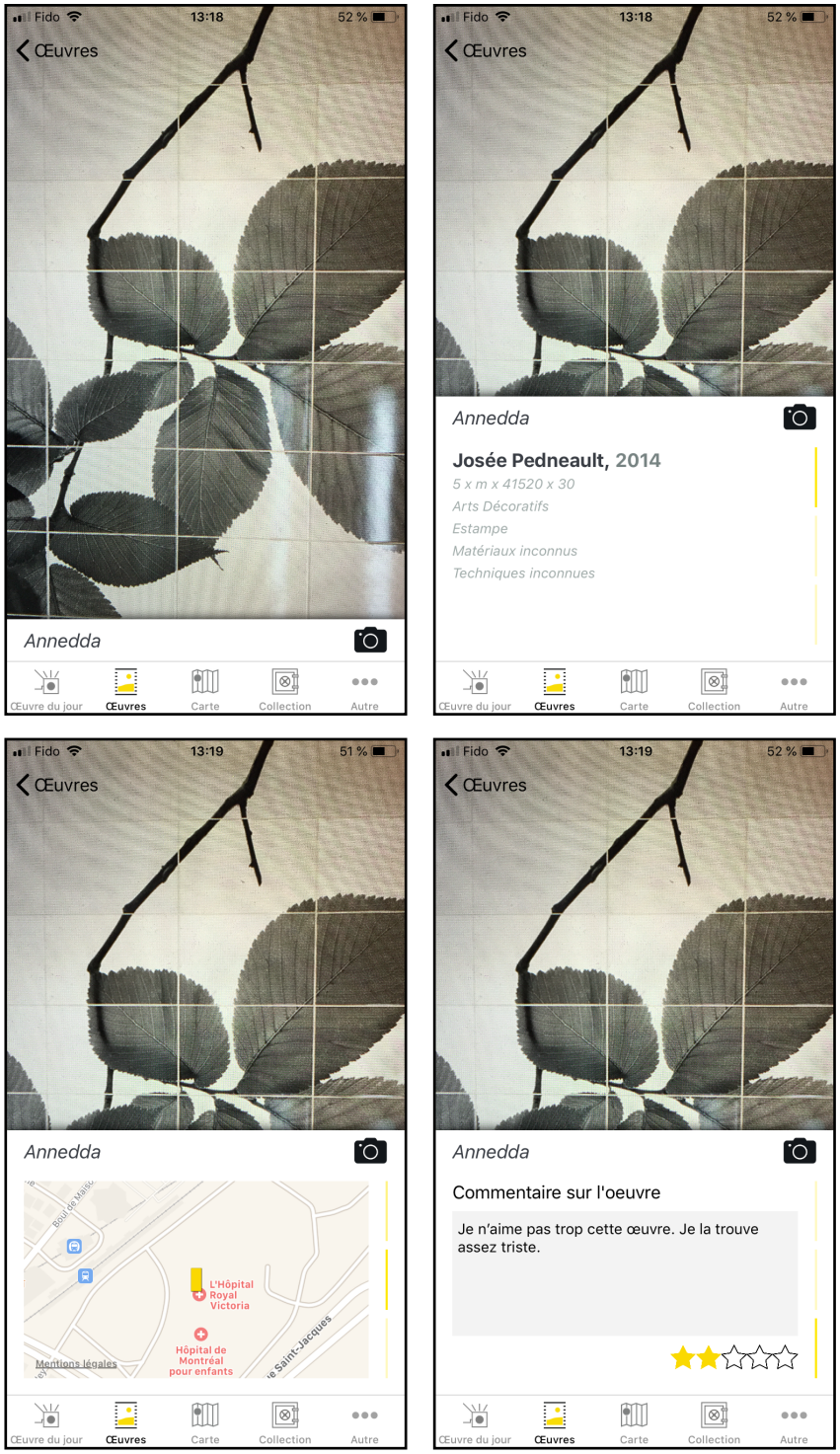






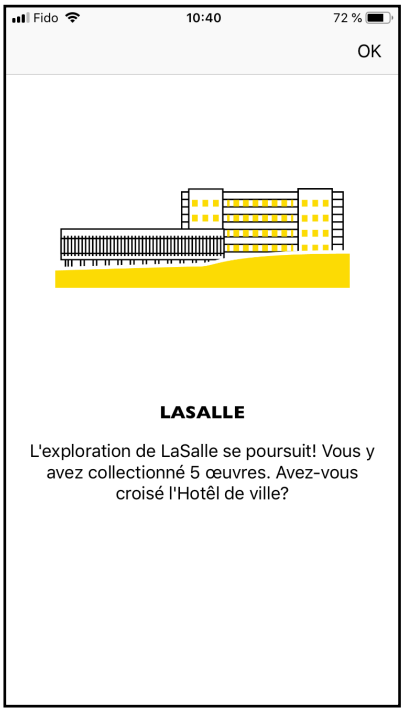
ANNEXE XVIII

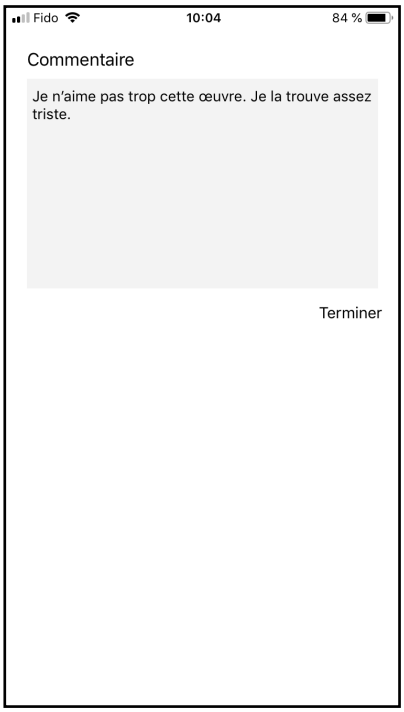
Vue de la fiche d'une œuvre (nouvelle version)



ANNEXE XIX

Vues de l’obtention de badge, la notation et le commentaire d’une œuvre après la prise d’une photo (nouvelle version)



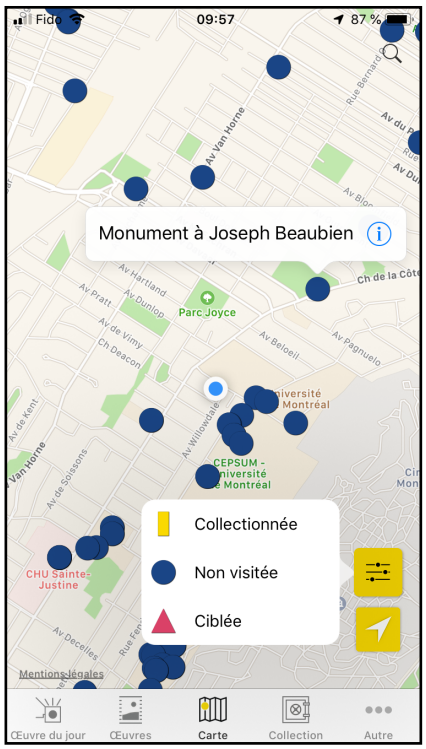


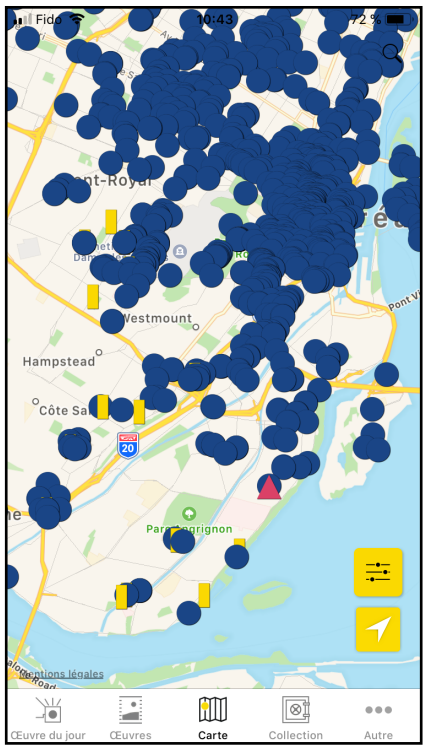
ANNEXE XX

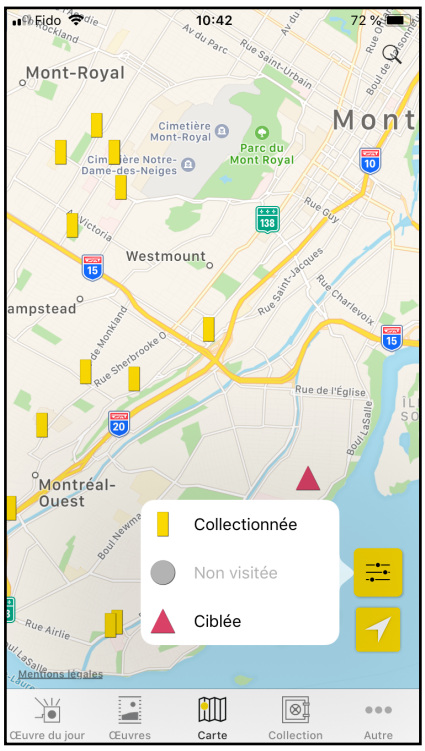
Ciblage d'une œuvre (nouvelle version)

ANNEXE XXI

Vues de la section « Carte » (nouvelle version)

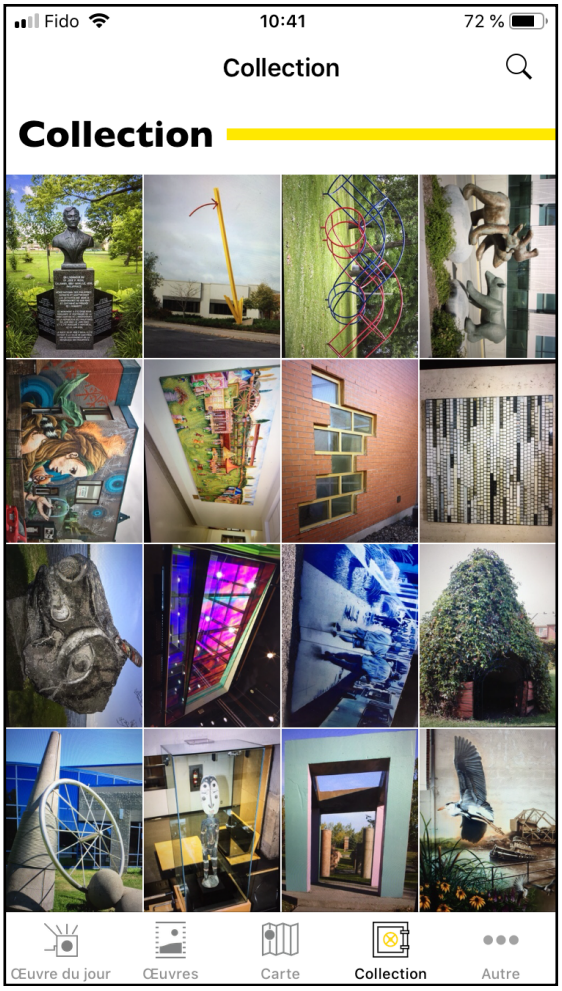






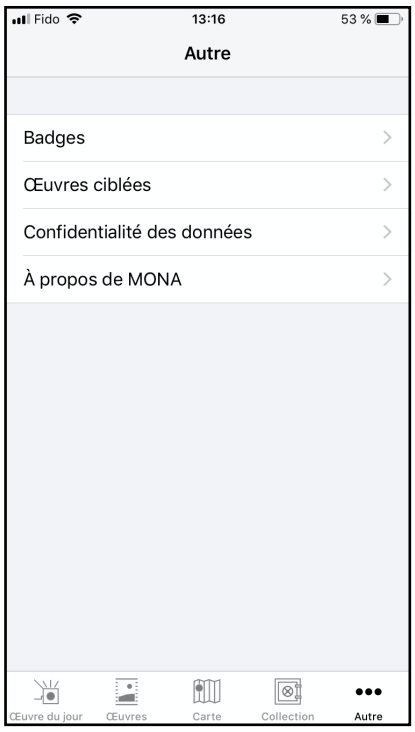
ANNEXE XXII

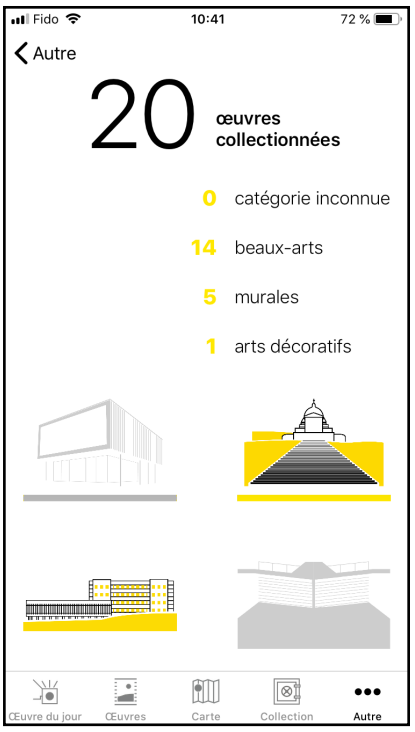
Vue de la section « Collection » (nouvelle version)

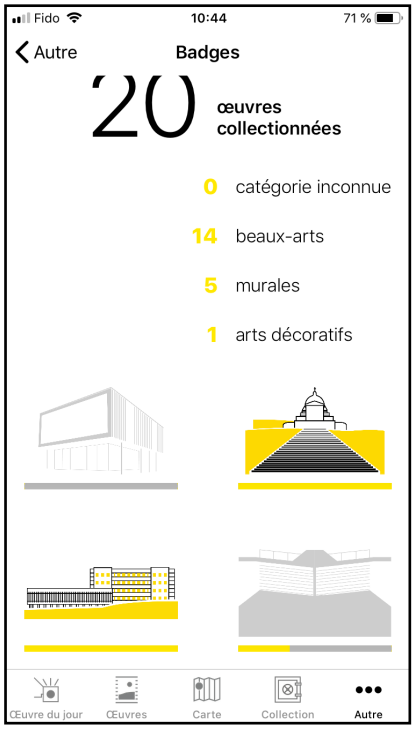


ANNEXE XXIII

Vues de la section « Plus » - Badges (nouvelle version)

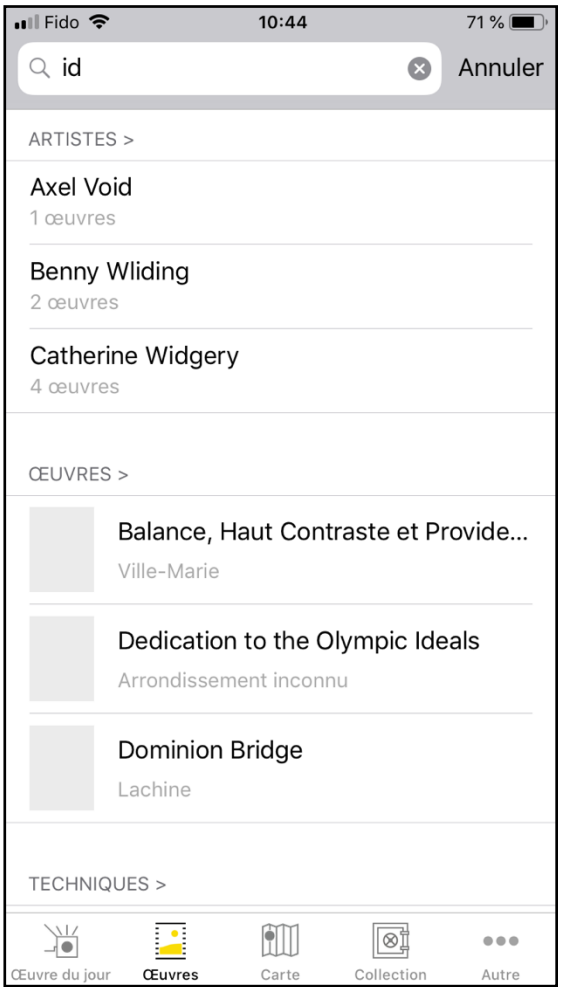






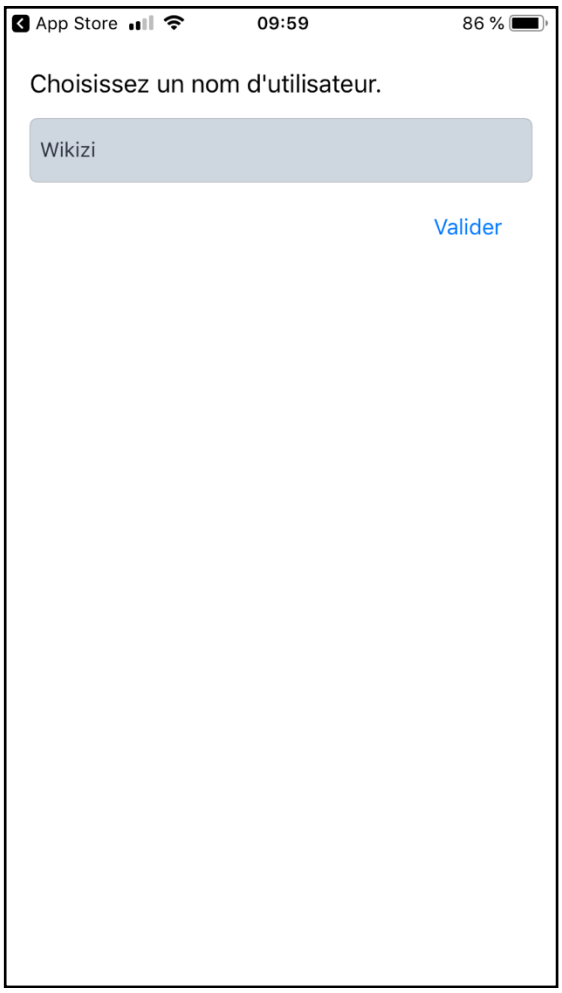
ANNEXE XXIV

Vue de la recherche (nouvelle version)



ANNEXE XXV

Vue du choix de nom d'utilisateur au lancement de l'application (nouvelle version)



LISTE DES RÉFÉRENCES BIBLIOGRAPHIQUES

[1] Krause, L. Lena Krause. Repéré à http://lenamk.site/

[2] Département d'Histoire de l'Art et d'Études Cinématographiques - Université de Montréal. Art et Site. Repéré à http://www.artetsite.org/

[3] O'Meara, G. (2018, 30 octobre). Succès du concours pour des activités d'accueil respectueuses et inclusives. *UdeMNouvelles*. Repéré à https://nouvelles.umontreal.ca/article/2018/10/30/succes-du-concours-pour-des-activites-d-accueil-respectueuses-et-inclusives/

[4] Donnat, O. (2010). Les Pratiques culturelles à l'ère numérique. *Bulletin des bibliothèques de France (BBF),* (5), 6-12. Repéré à http://bbf.enssib.fr/consulter/bbf-2010-05-0006-001

[5] Donnat, O., & Lévy, F. (2007). Approche générationnelle des pratiques culturelles et médiatiques. *Culture Prospective*(2007/3), 1-31. doi:10.3917/culp.073.0001

[6] Boyer, P. (2016, 28 octobre). Quand le numérique bouscule l'art [Billet de blogue]. Repéré à https://www.latribune.fr/opinions/blogs/homo-numericus/quand-le-numerique-bouscule-l-art-610960.html

[7] Google. Google Arts & Culture. Repéré à https://artsandculture.google.com/

[8] Art Media Agency (AMA). (2014, 13 mars). Comment le numérique est en train de métamorphoser le monde de l'art [Billet de blogue]. Repéré à https://www.latribune.fr/blogs/le-blog-sur-le-marche-de-l-art/20140313trib000819757/comment-le-numerique-est-en-train-de-metamorphoser-le-monde-de-l-art.html

[9] Yueria. (2019, 6 mars). Les réseaux sociaux propagent l'art en ligne [Billet de blogue]. Repéré à https://www.evolyon.fr/inspirations-artistiques/art-en-ligne/

[10] Artsper Magazine. Instagram, impact du réseau social sur l'art [Billet de blogue]. Repéré à https://blog.artsper.com/fr/la-minute-arty/instagram-lart/

[11] Suess, A. (2018). Instagram and Art Gallery Visitors: Aesthetic experience, space, sharing and implications for educators. *Australian Art Education, 39*(1), 107-122. Repéré à https://www.academia.edu/36689964/Instagram_and_Art_Gallery_Visitors-_Aesthetic_experience_space_sharing_and_implications_for_educators

[12] Suess, A., & Budge, K. (2018, 31 janvier). Instagram is changing the way we experience art, and that's a good thing. *The Conversation*. Repéré à https://theconversation.com/instagram-is-changing-the-way-we-experience-art-and-thats-a-good-thing-90232

[13] Miller, M. (2018, 21 novembre). Les réseaux sociaux, une aubaine pour les jeunes artistes. *Le Monde*. Repéré à https://www.lemonde.fr/campus/article/2018/11/18/les-reseaux-sociaux-une-aubaine-pour-les-jeunes-artistes_5385034_4401467.html

[14] Desjardins, M.-L. (2019, 14 mars). Quand les technologies mobiles rendent l'art proche et partageur. *Arts Hebdo Médias*. Repéré à https://www.artshebdomedias.com/article/quand-les-technologies-mobiles-rendent-lart-proche-et-partageur-2/

[15] Hein, H. (1996). What Is Public Art?: Time, Place, and Meaning. *The Journal of Aesthetics and Art Criticism, 54*(1), 1-7. doi:10.2307/431675

[16] Université Laval. Qu'est-ce que l'art public ? Repéré à https://www.ulaval.ca/lart-public/quest-ce-que-lart-public/lart-public/definition.html

[17] Art Public Montréal. Qu'est-ce que l'art public ? Repéré à https://artpublicmontreal.ca/a-propos/quest-ce-que-lart-public/

[18] Specht, M. (2017, 17 septembre). 10 best cities for artists in Canada. Repéré à https://www.zolo.ca/news/best-cities-for-artists-canada

[19] Bleiberg, L. (2016, 1er avril). 10Best: Cities to see street art. *USA Today*. Repéré à https://eu.usatoday.com/story/travel/destinations/10greatplaces/2016/04/01/murals-street-art/82466442/

[20] Ville de Montréal. Art Public. Repéré à https://artpublic.ville.montreal.qc.ca/

[21] Open Knowledge Foundation. Open Definition. Repéré à https://opendefinition.org/

[22] Ville de Montréal. *Politique de données ouvertes*. Repéré à http://donnees.ville.montreal.qc.ca/portail/wp-content/uploads/2015/10/Politique-de-donn%C3%A9es-ouvertes.pdf

[23] Ville de Montréal. Portail de données ouvertes - Art public - Information sur les oeuvres de la collection municipale. Repéré à http://donnees.ville.montreal.qc.ca/dataset/art-public-information-sur-les-oeuvres-de-la-collection-municipale

[24] Vollmer, T. (2013, 27 décembre). Creative Commons 4.0 BY and BY-SA licenses approved conformant with the Open Definition [Billet de blogue]. Repéré à https://creativecommons.org/2013/12/27/creative-commons-4-0-by-and-by-sa-licenses-approved-conformant-with-the-open-definition/

[25] Sheikh, A. A., Ganai, P. T., Malik, N. A., & Dar, K. A. (2013). Smartphone: Android Vs IOS. *The SIJ Transactions on Computer Science Engineering & its Applications (CSEA), 1*(4), 141-148. doi:10.13140/RG.2.2.10444.46724

[26] Rupesh. (2017, 27 septembre). iOS Layered Architecture [Billet de blog]. Repéré à https://codeingwithios.blogspot.com/2017/09/ios-layered-architecture.html

[27] Apple. (2012). *iOS Technology Overview*. Repéré à http://pooh.poly.asu.edu/Mobile/ClassNotes/Papers/MobilePlatforms/iOSTechnicalOverview.pdf

[28] Tracy, K. W. (2012). Mobile application development experiences on Apple's iOS and Android OS. *IEEE Potentials, 31*(4), 30-34. doi:10.1109/MPOT.2011.2182571

[29] GlobalStats. Mobile Operating System Market Share Worldwide. Repéré à https://gs.statcounter.com/os-market-share/mobile

[30] Costello, S. (2019, 24 juin). How Many Apps Are in the App Store? *Lifewire*. Repéré à https://www.lifewire.com/how-many-apps-in-app-store-2000252

[31] Swift. Repéré à https://swift.org/

[32] Apple Inc. Start Developing iOS Apps (Swift). Repéré à https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/

[33] Stanford (Producer). (2017). Developing iOS 11 Apps with Swift. [Cours vidéo] Repéré à https://itunes.apple.com/us/course/developing-ios-11-apps-with-swift/id1309275316

[34] Lets Build That App. (n.d.). Home [Chaîne YouTube]. Repéré à https://www.youtube.com/channel/UCuP2vJ6kRutQBfRmdcI92mA

[35] Apple inc. Documentation. Repéré à https://developer.apple.com/documentation

[36] Apple Inc. Documentation Archive. Repéré à https://developer.apple.com/library/archive/navigation/

[37] Apple Inc. Videos. Repéré à https://developer.apple.com/videos/

[38] Medium. Repéré à https://medium.com/

[39] Ray Wenderlich. Repéré à https://www.raywenderlich.com/

[40] Douglas, A., Mora, S., Morey, M., & Rea, P. (2016). *Core Data by Tutorials* (3 ed.): Raywenderlich.

[41] App Coda. Repéré à https://www.appcoda.com/

[42] Ng, S. (2016). *Intermediate iOS 10 Programming with Swift*: AppCoda.

[43] Hacking With Swift. Repéré à https://www.hackingwithswift.com/

[44] Keur, C., & Hillegass, A. (2016). *iOS Programming: The Big Nerd Ranch Guide* (6 ed.): Big Nerd Ranch.

[45] Neuburg, M. (2018). *Programming iOS 12: Dive Deep Into Views, View Controllers, and Frameworks* (9 ed.): O'Reilly Media, Inc.

[46] Stackoverflow. Repéré à https://stackoverflow.com/

[47] Krause, L., & Paquet, S. (2018). *Recherche, analyse et réflexion concernant la médiation de l'art public par le numérique: projet MONA*. Université de Montréal.

[48] Beauregard, V. (2018). *MONA-Serveur*. Université de Montréal.

[49] Qbaich, A. (2019). *Serveur web pour une application de découverte d'art public*. Université de Montréal.

[50] Ville de Montréal. Portail de données ouvertes - Murales subventionnées. Repéré à http://donnees.ville.montreal.qc.ca/dataset/murales

[51] Chaffanet, P. (2018). *Projet d'application mobile iOS: MONArt*. Université de Montréal.