Ecole Pratique des Hautes Etudes Commerciales

EPHEC

Institut d'Enseignement Supérieur Economique de Type Court

QUEL FRAMEWORK CHOISIR POUR UNE APPLICATION MOBILE MULTIPLATEFORME DE GESTION DE DETTE?

Florian GÉRARD

Rapporteur(s)

Travail de Fin d'Études présenté
en vue de l'obtention du diplôme de
BACHELIER EN INFORMATIQUE DE GESTION

Année académique 2018-2019

Remerciements

Durant mon parcours scolaire j'ai eu la chance de croiser plusieurs personnes m'ayant aidé et soutenu, voici l'occasion de les remercier.

Merci au corps enseignant de l'Institut Paul Lambin (où j'y ai commencé mes études de développeur) et de la promotion sociale de l'EPHEC pour les cours et les conseils transmis. Un remerciement plus particulier à monsieur Jean-Paul Hecquet, professeur du cours de projet d'intégration, pour m'avoir guidé sur ce TFE en me recadrant sur les points importants.

Petite pensée à toute l'équipe Contraste Consulting, entreprise où j'ai vraiment pu commencer ma carrière de développeur entouré de personnes bienveillantes et généreuses en partage de connaissances. Merci à Maxime Debersaques pour tout l'apprentissage et la confiance ainsi qu'à Flavio Occhipinti pour les débats animés sur les meilleurs frameworks JavaScript existant et enfin à Nicolas Gielen pour assurer le backend. Je n'oublie pas l'IT Support pour sa bonne humeur et pour avoir fait fonctionner l'imprimante !

Je tenais à ne pas oublier les membres du groupe YAWE (joyeux lurons sachant parfois être sérieux) pour les relectures toujours utiles voire même nécessaires de mes travaux.

Et enfin, merci spécial à mon meilleur ami Thibault Vanwersch et mon père Luc Gérard pour leurs soutiens inconditionnels au cours de mes études semé d'embûches.

Table des matières

Introduction

Pour ce travail de développement, je dois m'attaquer à deux problématiques.
Premièrement, comment gérer au mieux mes dettes et créances vis à vis de mon entourage.
Deuxièmement, quel est le meilleur framework pour en faire une application disponible sur les appareils de la marque Apple et ceux utilisant Android.

Money And Relationship eXperience

Quand nous avons des interactions sociales, il y a souvent un moment où la question de l'argent arrive : j'ai payé un sandwich à un ami, un collègue m'a prêté 10 euros ou j'ai assumé l'addition d'un restaurant en famille et chacun devra me rembourser sa part.
Et bien évidemment, viendra toujours le moment du remboursement! Mais comment se souvenir de tout ça? La mémoire? On oublie vite ce genre de choses, surtout le montant exact! Une feuille de papier? Elle va vite se perdre!
C'est pour ce type d'occasions qu'une application sur téléphone se révèle utile.
Une fois dans l'application, nous pouvons créer des relations (les personnes qui nous doivent l'argent ou à qui l'on doit de l'argent), et attacher à ces relations des paiements.
Il sera possible d'activer un rappel à un paiement. Celui-ci permettra qu'on reçoive une notification à la date voulue indiquant qu'il serait temps que le remboursement se fasse.
Ce rappel pourrait être mis par défaut en fonction du type de relation de la personne : on pourrait par exemple imaginer qu'on laisse 7 jours avant d'obtenir le rappel pour une relation de type collègue mais qu'on attende 30 jours si c'est pour la famille.
Il me tient également à cœur que cette application soit Open-Source et qu'elle soit donc exploitable par un autre développeur, le plus rapidement et facilement possible! Ce qui fait que j'utiliserais un ORM (Object-relational mapping) afin de ne pas être dépendant d'une base de données particulière.

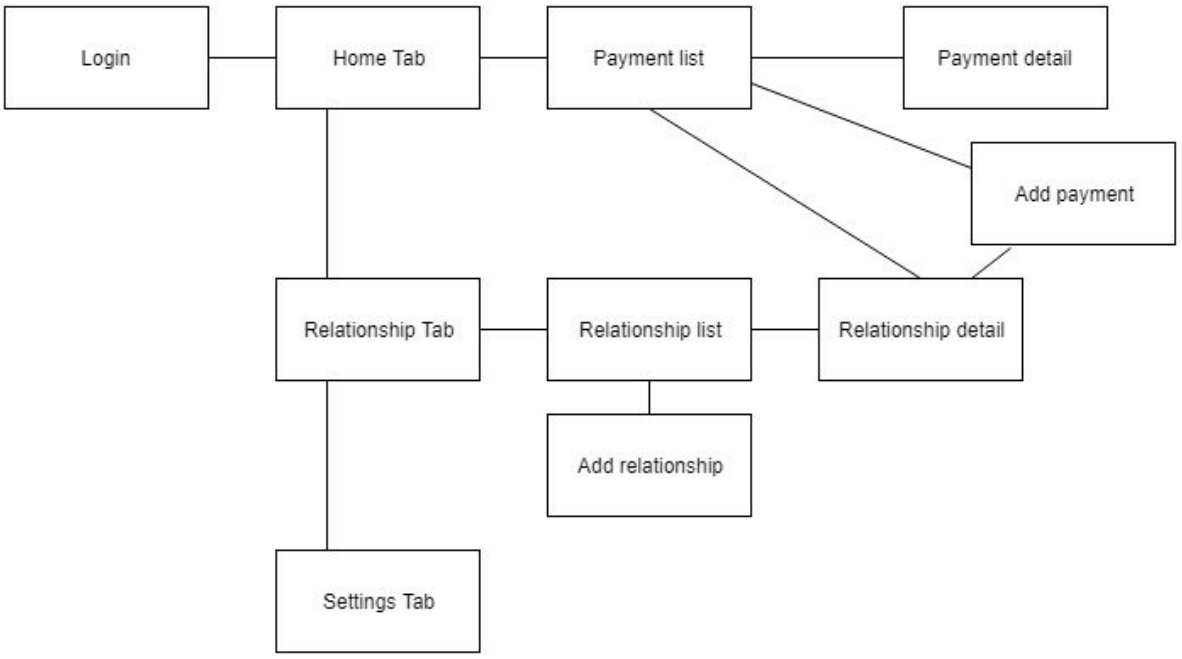
Problématique de la technologie à utiliser pour développer sur support smartphone

Savoir développer des applications pour plateforme mobile, ce n'est pas se préparer pour l'avenir mais c'est être prêt pour le présent. À présent, et ce depuis 2017, le nombre de personne utilisant un smartphone pour aller sur internet est supérieur que celui sur pc. Passer à côté du développement pour mobile revient à ignorer la majorité des utilisateurs d'appareils numériques.
Mais nous voilà avec un nouveau problème! Les parts de marchés entre les différents acteurs de système d'exploitation sont divisés entre iOS d'Apple (22,85%) et Android de Google (74,45%) (d'après les chiffres de janvier 2019) et ne pas

développer pour une de ces deux plateformes c'est se fermer à beaucoup de potentiels clients. Surtout dans le cadre de l'Amérique du Nord où les deux concurrents sont à 49% de part de marché chacun! Malheureusement ces deux OS ont deux langages de programmation natif différents doublant ainsi le travail de développement pour le Front-End. Si nous voulons donc toucher le plus de personne possible avec notre application, nous faudrait-il une équipe avec des informaticiens maîtrisant Objective-C ou Swift pour les appareils à la pomme et Java avec le framework Android pour les smartphones ou tablettes disposant de l'OS Android? Pas spécialement!
La potentielle solution? Le développement d'application Hybride! Le principe, coder dans un seul langage (accompagner d'une bibliothèque ou boîte à outils) qui va être ensuite être traduit en langage natif à la compilation permettant ainsi d'avoir en un seul travail une application mobile tournant sur plusieurs types d'appareil.
Mais des outils pour faire cela, il y en a énormément, mon but ici est de vous présenter les plus pertinents et de vous conseiller sur lequel serait le meilleur (le tout était souvent très subjectif restons honnête).

Description de l'application

Schéma de navigation des pages



Les écrans

Tabulation:

Une fois la connexion faite, nous sommes redirigé sur la page d'accueil et avons comme menu une tabulation en bas vers la page d'accueil (celle de base), des options et la page des relations.

Écran d'accueil:

La liste des 5 paiements non remboursés les plus vieux.
Chaque élément de la liste est cliquable pour accéder à ses détails.
Un bouton "ajouter" sera mit en bas pour ajouter un nouveau paiement.
route utilisée: getPayment

Écran des options:

Dans les options, l'utilisateur pourra changer la langue de l'application (l'anglais et le français seront proposés de base mais d'autre langues seront proposables facilement par la suite).
Cela sera également ici que nous pourrons ajouter des types de relations.

Écran des relations:

La liste des relations de l'utilisateur.
Chaque élément de la liste affiche le nom et le type de relation. On peut aussi accéder aux détails de chaque relation via cet écran.
On pourra directement ajouter une relation sur cet écran.
Un clic sur une relation permettra d'aller dans ses détails.
routes utilisées: getAllRelation , addRelation

Détails d'une relation:

Y sera affiché de manière éditable le nom de la relation et son type.
Une liste d'historique des paiements sera présente avec en premier ceux non remboursé.
Chaque élément de la liste sera cliquable pour en voir les détails.
Un bouton "ajouter" sera mis en bas pour créer un nouveau paiement lié à cette relation.
On pourra également depuis ici supprimer la relation
Routes utilisées: getRelation , updateRelation , deleteRelation

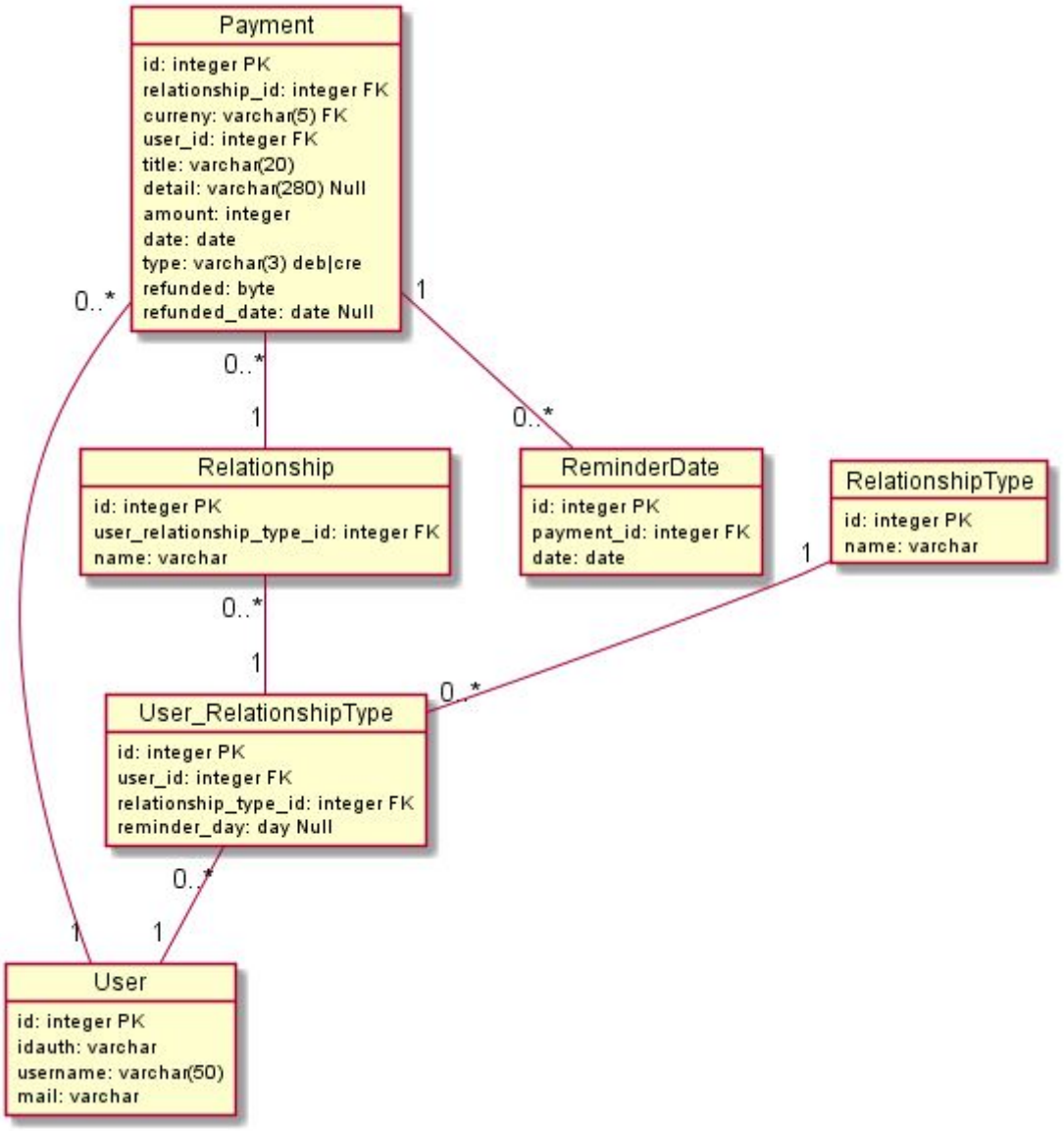
Détails d'un paiement:

Dans les détails seront présentes les informations du nom de la relation, le titre du paiement ainsi que le détail si présent, le montant avec la devise, la date de création ainsi que la date du remboursement si il a été remboursé.
Un bouton "remboursé" sera présent pour indiquer que le paiement est remboursé. S'il a déjà été remboursé, une icône verte sera présente.
On pourra également via cet écran ajouter un rappel.
Routes utilisées: getPayment , refundedPayment , getReminder , addReminder

Nouveau paiement:

Cet écran peut être appelé de différents endroits : depuis l'écran d'accueil, depuis l'écran récapitulatif des paiements et depuis le détail d'une relation.
Sur cet écran sera disponible un formulaire afin d'enregistrer un nouveau paiement.
Si nous venons de la page de détail d'une relation, le champ relation du formulaire sera automatiquement rempli avec la catégorie liée à cette relation.
Routes utilisées: addPayment , getAllCurrencies , getAllRelationship

Modèle de la base de données

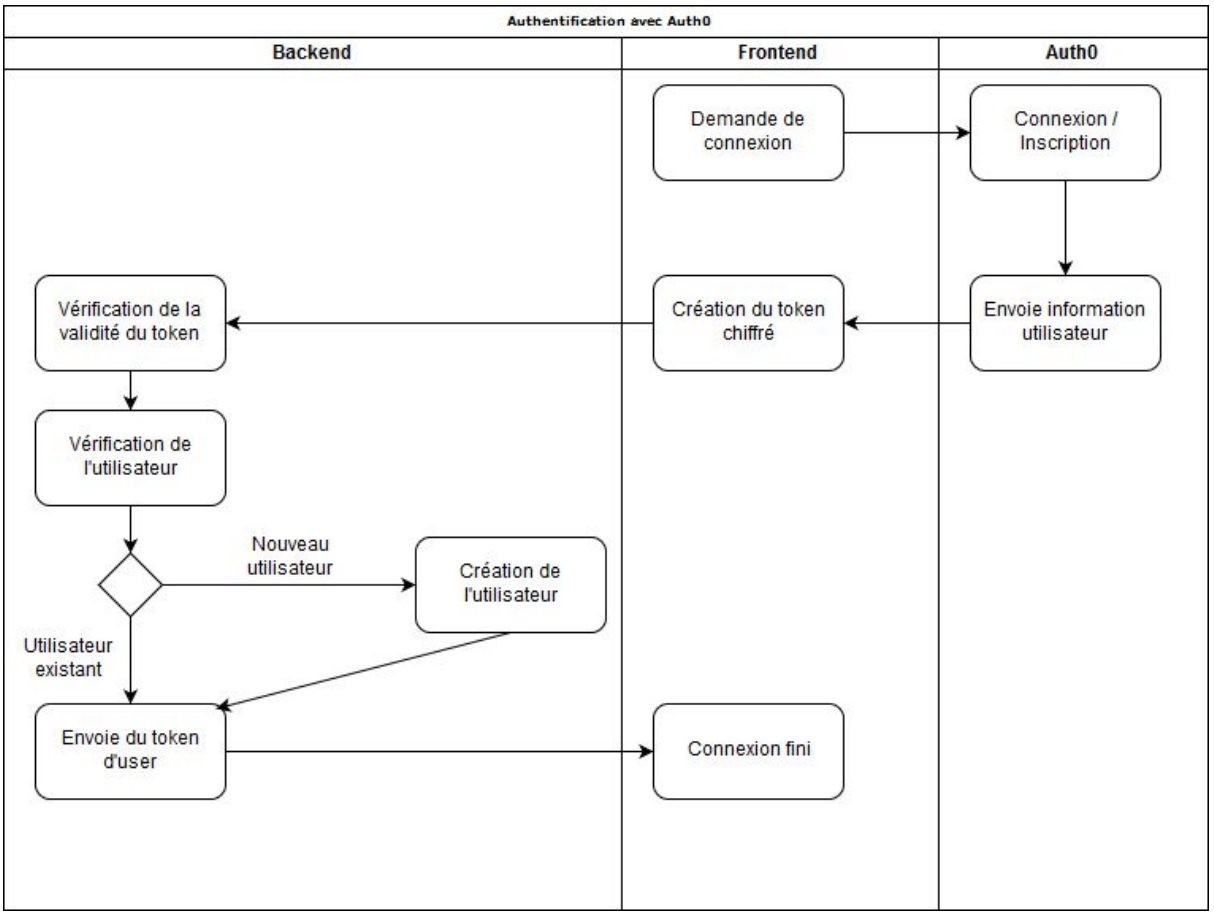


Règles

- Un utilisateur doit posséder un nom, une adresse mail et un ID venant d'Auth0. Il peut avoir plusieurs types de relations ainsi que plusieurs devises. L'ID venant d'Auth0 est unique.
- Un type de relations doit posséder un nom et peut appartenir à plusieurs utilisateurs. Le nom est unique.
- Dans la table de liaison entre un utilisateur et un type de relations, l'ID de l'utilisateur et l'ID du type forment une clé unique.
- Une relation doit avoir un nom ainsi qu'être liée à un unique utilisateur et à un unique type de relations.
- Une devise doit posséder un nom de maximum 50 caractères et un label de 3 caractères. Elle peut appartenir à plusieurs utilisateurs. Le nom est unique.

- Un paiement doit posséder une unique devise et appartient à une relation unique, il doit posséder un montant, une date, un type qui sera soit “det” pour dette soit “cre” pour créance ainsi qu'une date de remboursement.
- Un rappel de paiement doit appartenir à un paiement et doit posséder une date.
- Un rappel de paiement par type de relations doit appartenir à un type de relation et doit posséder un nombre de jours.
- Un type de relations ne peut pas être supprimé si une relation la possède.

L'authentification



Pour s'authentifier à l'application, on passera via le service Auth0 (https://auth0.com/), ce qui permettra d'utiliser au choix de l'user son compte Google, Facebook ou Hotmail. L'authentification classique en avec adresse e-mail et mot de passe sera également possible.
Auth0 me renvoie un token avec contenant comme information un ID d'utilisateur que je traite ensuite. Je regarde dans ma table “user” si une personne avec cette ID d'Auth0 existe. Si c'est le cas il est redirigé vers la page d'accueil. Si le web service le repère en tant que nouvel ID, on lui demandera plus d'informations pour finaliser son inscription.
Passer via le service Auth0 permet à l'utilisateur de passer par un de ses comptes pour se connecter (méthode de plus en plus populaire). De plus, la gestion du mot de passe est faite par un service reconnu et permet également d'avoir un rappel par

mail de ce genre d'informations (ne possédant pas de serveur mail, j'en avais besoin).
Pour la sécurité du backend, un token chiffré reprenant les informations d'Auth0 sera envoyé depuis le frontend, afin de les comparer avec le Json reçu afin d'éviter les faux comptes ne passant pas via mon compte Auth0.

Les types de relations et currency

Un user peut rajouter un type de relation pour ses relations. Quand celui-ci le fait, le web service vérifie d'abord que ce type existe. Si c'est le cas, on rajoute dans la base de donnée une liaison dans User_RelationshipType, sinon on ajoute d'abord ce nouveau type de relations dans RelationshipType et ensuite on fait la liaison. Cela évite d'avoir des doublons dans RelationshipType.
Grâce à cela l'user peut se voir proposer des types de relations quand il en veut un nouveau, sans pour autant avoir la liste de toutes les RelationshipType de tout le monde. Cela fait aussi un gain de mémoire pour la base de données.
Le principe sera le même pour les Currency.

Routes API du web service

Voici les descriptions de toutes les routes du web service. Cette documentation servira à ce que n'importe quel projet Front-End puisse l'utiliser facilement.

Généralités : toutes les routes (excepté /login) doivent avoir dans le header la propriété "Api-Token" qui sera le token reçu par la route /login. Ceci me permettant de sécuriser les routes, mais également de faire transiter automatiquement des informations comme l'ID de l'user (utile lors de nombreuse requête).
Les requêtes et les réponses seront envoyées en Json.
Si il y a une erreur dans le backend, celui-ci renverra un Json avec la propriété message qui est un tableau de string décrivant les erreurs (car il pourrait y avoir plusieurs soucis à afficher).

login

Les détails de fonctionnement de cette route se trouvent dans le chapitre "L'authentification"

POST	login

Request	{
	user_id varchar
	name varchar
	}
	l'user_id étant le token renvoyé par Auth0
	un token sera envoyé avec, mais pas le même que les autres routes

Response	{
	api-token varchar
	}

updateName

PUT	user

Request	{
	name varchar
	}

Response	{
	name varchar
	}

allRelationshipType

GET	relationshipType
Request	

Response	liste de tous les types de relations existantes dans la base de données
	{
	id integer
	name varchar
	}

allRelationshipTypeForUser

GET	user/relationshipType
Request	

Response	liste des types de relations liées à l'utilisateur courant
	{
	id integer
	relationship_type object {
	id integer
	name integer
	}
	}

addRelationshipTypeForUser

POST	user/relationshipType

Request	{
	name varchar
	}

Response	{
	id integer
	name varchar
	}

deleteRelationshipTypeForUser

DELETE	user/relationshipType/{id}
Request	
Response	

getAllRelationship

GET	user/relationship query paramètre: name, relationship_type
Request	possibilité de trier les relations par type de relations ou de faire une recherche sur le nom

Response	liste des relations de l'utilisateur courant
	détail de la relation
	{
	id integer
	name varchar
	user_relationship_type object {
	relationship_type object {
	name varchar
	}
	}
	}

getRelationship

GET	user/relationship/{id}
Request	

Response	détail de la relation
	{
	id integer
	name varchar
	user_relationship_type object {
	relationship_type object {
	name varchar
	}
	}
	}

addRelationship

POST	user/relationship

Request	{
	name varchar
	relationship_type_id integer
	}

Response	la nouvelle relation
	{
	id integer
	name varchar
	user_relationship_type object {
	relationship_type object {
	name varchar
	}
	}
	}

updateRelationship

PUT	user/relationship/{id}
Request	{ name varchar

	relationship_type_id integer }

Response	la relation mise à jour
	{
	id integer
	name varchar
	user_relationship_type object {
	relationship_type object {
	name varchar
	}
	}
	}

deleteRelationship

DELETE	user/relationship/{id}
Request	

Response	la relation supprimée
	{
	id integer
	name varchar
	relationship_type object {
	name varchar
	}
	}

allCurrency

GET	currencies
Request	

Response	Liste de toutes les currencies
	{
	id integer
	name varchar
	label varchar
	}

allCurrencyByUser

GET	user/currencies
Request	

Response	Liste de toutes les currencies utilisées par l'utilisateur courant
	{
	id integer
	user_id integer
	currencies_id integer
	currency object {
	id integer
	name varchar
	label varchar
	}
	}

addCurrency

POST	user/currencies

Request	{
	name varchar
	label varchar
	}

Response	le currency nouvellement ajouté

getAllPayment

GET	user/payment query paramètre: type, refunded, relationship_id, number_row
Request	query optionnel pour filtrer le type (deb ou cre) ou si c'est déjà remboursé ou non (par défaut, prend tout) ainsi que le relationship_id si on veut tous les paiements liés à un utilisateur number_row permet de sélectionner combien on veut en afficher
Response	liste des paiements de l'user courant : { id integer amount integer title varchar description varchar

	date date
	type varchar
	refunded byte
	refunded_date date
	relationship object {
	id varchar
	name varchar
	}
	user_currency object {
	id integer
	currency object {
	id integer
	name varchar
	label varchar
	}
	}
	}

getPayment

GET	user/payment/{id}
Request	

Response	un paiement retrouvé via son ID :
	{
	id integer
	amount integer
	title varchar
	description varchar
	date date
	type varchar
	refunded byte
	refunded_date date
	relationship object {
	id varchar
	name varchar
	}
	user_currency object {
	id integer
	currency object {
	id integer
	name varchar
	label varchar
	}
	}
	}

addPayment

POST	user/payment

Request	{
	amount integer
	title varchar
	description varchar (optionnel)
	type varchar (deb || cre)
	date date
	user_currency_id integer
	relationship_id integer
	user_id integer
	}

Response	le paiement nouvellement créé voir getPaymentDetail

refundedPayment

PUT	user/payment/refunded/{id}

Request	Aucune information n'est envoyée en plus de l'ID du paiement qui est à présent remboursé
	La date du remboursement se fait dans le backend
	{
	}

Response	le paiement à présent remboursé voir getPaymentDetail

updatePayment

PUT	user/payment/{id}

Request	{
	title varchar
	description varchar (optionnel)
	user_currency_id integer
	relationship_id integer
	amount integer
	}

Response	le paiement mis à jour voir getPaymentDetail

deletePayment

DELETE	user/payment/{id}
Request	
Response	

getReminderDate

GET	user/reminderdate
Request	

Response	Toutes les dates de rappel de paiement (avec le paiement) de ceux qui sont non-remboursés
	{
	id integer
	date date
	payment object {
	voir getPayment
	}
	}

postReminderDate

POST	user/reminderdate

Request	{
	paymentId integer
	date date
	}

Response	Le reminder date nouvellement ajouté voir getReminderDate

deleteReminderDate

DELETE	user/reminderdate/{id}

Les technologies

Base de données

Ma base de données sera une relationnel MySQL. Je dispose déjà d'un hébergement web chez OVH, je l'exploite donc pour éviter des surcoûts. De plus, MySQL répond parfaitement à mes besoins de stocker de données, pas besoin de plus.
Je dois juste faire attention car je ne possède qu'une seule base de données, partageant ainsi tous mes projets. Pour ne pas m'y perdre, je nomme chacune de mes tables avec l'abréviation du projet. Ici, ça sera "marx", ainsi par exemple, la table des users s'appellera "marx_user".

Backend

Toujours dans l'optique d'éviter de payer pour rien, j'utilise mon serveur OVH pour y faire tourner un webservice. Dans mon abonnement je ne peux faire tourner que du PHP, c'est donc dans cette famille de langage que j'ai cherché un framework pour faire une API.
Mes besoins sont simples, pouvoir interagir avec ma base de données le plus facilement possible (l'utilisation d'un ORM sera donc souhaité) et communiquer avec des JSON.

Il existe deux grands frameworks reconnu en PHP, Symfony et Laravel.
Les deux sont assez lourd, ils sont fait pour également afficher des sites webs (ce dont je n'ai pas du tout besoin).
Heureusement, l'équipe de Laravel à sorti une variante plus légère, utile pour uniquement faire des API. Parfait, c'est ce dont j'ai besoin.
Lumen est son nom. Et en ce qui concerne l'ORM, l'équipe de Laravel vient à nouveau à mon secours en ayant créé Eloquent.

Quels frameworks front-end tester et comment?

Les choix

Des boîtes à outils pour faire des applications hybride, il y en a à la pelle, je dirais même qu'il y en a de trop. L'impression que chaque développeur veut monter le sien pour la reconnaissance se fait vite ressentir et on se retrouve avec beaucoup de fork de projet open-source dans l'univers JavaScript. J'ai donc dû faire un choix en fonction de ceux apportant une réelle plus-value dans ce monde fort morcelé du développement mobile hybride et j'en ai tiré trois frameworks populaires.
Parmi mes choix, deux frameworks JavaScript vont ressortir car excepté Xamarin de Microsoft il n'existe que cela pour faire des applications multiplateformes.

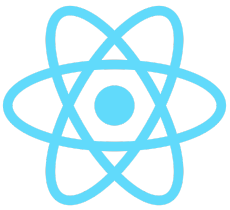
Ionic

Pionnier en la matière, je ne pouvais donc pas passer à côté de celui qui a popularisé le développement multiplateforme en JavaScript.
Basé sur Angular (qui est un coup de cœur personnelle du Front-End), Ionic à 37 000 fans sur Github, forké 12 000 fois et ayant 293 contributeurs (chiffre du 26 février 2019). Ionic a sorti sa première version en 2013.

Xamarin

Derrière Xamarin se cache une histoire un peu complexe. De base Xamarin existait afin de faire tourner des applications .Net de Microsoft sur les distributions Unix (GNU/Linux et MacOS) et s'est transformé petit à petit en framework C# afin de faire de l'Android et de l'iOS en un seul code avec les outils Microsoft. Celui-ci a d'ailleurs racheté le projet (et le personnelle derrière) en 2016 pour y apporter son soutien financier et logistique.

React Native



Soutenue par Facebook, la première version de React Native est sortie mars 2015 et a su vite se faire apprécier grâce aux fans de base de React JS sur lequel ce framework en JavaScript se base. 74 000 fans sur Github et avec 1800 contributeurs le 26 février 2019, React Native est le mastodonte du développement iOS / Android actuellement et reste un incontournable à tester.

Autre?

En testant les trois frameworks cités précédemment, j'explore essentiellement le JavaScript avec Angular et React (qui sont les 2 frameworks du moment pour faire du Front-End) et je n'oublie pas de jeter un œil à ce que propose Microsoft (qui reste un des plus gros acteurs de l'informatique) dans ce secteur.
Mais il existe bien plus de bibliothèque qu'on pourrait tester.
PhoneGap lancé par Adobe reposant aussi sur le JavaScript, Onsen UI, Framework Seven, Quasar également en JavaScript sans oublier NativeScript qui monte de plus en plus en force et qui nous permet de faire du mobile en Vue JS (concurrent à Angular et React).

Méthode pour éprouver les frameworks

Pour savoir à quel point ces frameworks sont agréables à utiliser, je testerais plusieurs éléments importants à mes yeux.

- L'initialisons d'un projet et la clarté de la documentation
- La gestion de l'affichage, comment montrer les données, interagir avec eux et utiliser un formulaire
- La pagination, comment est géré le passage d'une page à l'autre
- L'interaction avec le backend, y récupérer des informations et en envoyer
- Savoir y gérer plusieurs langues
- Afficher des notifications
- Avoir un rendu graphique le plus beau possible
- Comment déployer l'application sur smartphone

Je me baserais essentiellement sur la documentation officielle de chaque framework pour développer l'application car selon moi un framework efficace est aussi celui qui a la meilleure documentation possible, c'est un critère essentiel à mes yeux.

Testons les applications hybrides

Avant-propos

L'environnement de test

Afin d'éprouver nos applications que nous développons, nous bénéficions d'émulateur de smartphone. Les plus reconnus sont fourni par l'ADM (Android Device Manager) de Google disponible via l'Android Studio (IDE de Google en collaboration avec JetBrains) mais également via Visual Studio.
Nous pouvons également tester directement sur notre propre appareil Android si il est relié au pc. Ceci est plus performant (un émulateur est très lourd en ressource de mémoire vive) mais nous permet pas de tester sur plusieurs types d'écrans.

Précisions importante, je n'ai pas testé les frameworks dans leurs versions iOS. Certes je fais un travail de veille technologique pour applications multiplateformes et cela serait donc requis que je teste le tout sur plusieurs plateformes, cependant pour pouvoir tourner mon code sur les appareils Apple, je dois posséder justement un Mac (pour la compilation et également l'émulation) ce que je ne détiens pas.

Pour les déploiements, je ne ferais donc cela qu'en Android. Je serais à même de fournir un APK (Android Package, un format de fichier pour exécuter des installations sur appareil Android, équivalent du .EXE de Windows) mais je ne pourrais pas déployer cela sur le Google Play Store (application où la grande majorité des personnes téléchargent leurs logiciels) car cela est payant (25 dollars à vie) et que je ne possède pas de carte de crédit pour ça. Précisons tout de même que pour le market d'iOS, il faut débourser 99 dollars par an.
Le soucis en ne passant pas via un market d'application est que l'utilisateur va devoir dans les options de son appareil autorisé manuellement les applications externes ce qui peut l'effrayer (un message d'avertissement ne rassurant jamais apparaît).

Vue que je testerais sur du un émulateur Android, j'installerais Android Studio me permettant d'installer automatiquement et un émulateur mais également les outils Java qui compileront mon code, c'est à dire le JDK (Java Development Kit) et l'Android SDK (Software development kit) sans quoi je ne saurais rien produire.

En ce qui concerne les outils de développement, pour Xamarin j'utiliserais l'IDE Visual Studio 2017 qui est fourni par Microsoft et j'exploiterai Visual Code qui est un excellent éditeur de texte se transforment aisément en IDE en y ajoutant quelques plugins.

Le JavaScript

Au vue de la multiplicité des frameworks JavaScript, je trouvais important de vous présenter avant ce langage et son évolution.

1996, ce langage sort avec le navigateur Netscape 2 (actuelle Firefox) et il apporte un gros changement au développement de page web, l'interactivité avec l'utilisateur sans devoir recharger la page. Cela peut être en gérant des formulaires plus efficacement en indiquant des messages d'erreurs directement ou en intégrant des animations plus fantaisiste.
Pour ce faire, le code JavaScript nous permet d'interagir avec le DOM (Document Object Model) en y modifiant l'HTML (langage balisé formant nos page web) et changé à la volé celui-ci.

Ainsi, ce langage est vu comme une aubaine et permet d'avoir des sites dynamiques sans devoir tout le temps rafraichir la page.
Mais comme souvent dans le monde informatique, tout n'est pas rose! Le JavaScript est un langage interprété, cela veut dire que chaque navigateur va le lire comme il l'entend ce qui va poser des soucis de compatibilité de code et arrachera les cheveux de nombreux développeur qui vont devoir travailler à ce que leurs sites fonctionnent partout. ECMAScript va apparaître en 1997 pour normaliser ce langage auprès des distributeurs de navigateur. Sans réel grand succès.
Une solution est apparue avec des bibliothèques traduisant le JavaScript normalisé en fonction du navigateur où est interprété notre code (cela s'appelle le Polyfill). Le plus populaire fut le JQuery lancé en 2006. Bien évidemment cela rend les programmes moins performant mais fort heureusement les navigateurs ont de plus en plus tendance à respecter l'ECMAScript.

Ce langage a également la caractéristique de ne pas être typé. On ne définit pas à l'avance si une variable sera un string, un chiffre, un boolean ou tout autre type pouvant existant.

Le JavaScript à prit un tournant majeur en 2009 quand NodeJS est présenté par son créateur Ryan Dahl. NodeJS permet tout simplement de pouvoir exécuter du JS en dehors du navigateur! Pour ce faire il se base justement sur le moteur JavaScript de Google, V8.
Cela permet par exemple de faire du rendu serveur en JavaScript (du backend par exemple) et a surtout permis l'émergence de framework tel Angular dont on en parlera plus tard avec Ionic mais également d'outil de testing comme Karma ou Jasmine.
Autre avancé non négligeable, l'apparition de NPM (Node Package Manager) qui permet le partage en une ligne de commande de bibliothèques (équivalent de NuGet pour C# ou Composer pour PHP). C'est devenu un élément indispensable en JavaScript.

Ionic

Ce framework Cross-Plateforme à sorti sa première version stable en 2013 en se voulant respectueux des standards du web mais pour faire du mobile.

Avant de parler à proprement dit d'Ionic, je vais devoir introduire Angular et Cordova car il les utilise tous les deux afin de permettre aux développeurs de réaliser des projets d'application mobile.

Commençons avec Apache Cordova qui est le moins le connu.
Celui-ci permet d'utiliser des fonctionnalités natives du téléphone (la caméra, le gps, afficher des notifications, …) et est compatible avec de nombreux système d'exploitation mobile (Android et iOS bien évidemment mais également Windows Phone, Blackberry et Symbian l'ancien OS de Nokia). Cordova utilise les technologies web pour fonctionner.

En ce qui concerne Angular, framework développé par Google depuis 2009, il apporte essentiellement une structure, une architecture aux applications webs où chaque composant (on ne parle plus de page) aura son propre HTML pour la forme, son JavaScript qui sert de contrôleur et son style CSS (avec toujours la possibilité d'avoir un CSS global mais c'est n'est pas recommandé).
Cet outil de Google utilise depuis 2016 TypeScript qui est une surcouche de JavaScript permettant de typer les variables (les définir comme chaîne de caractère, nombre, …) et ainsi permettre une meilleur lisibilité du code.

Angular utilise des directives pour structurer l'HTML. Ceci permet de changer dynamiquement le DOM via certains attribut.
Ainsi nous pouvons directement mettre des conditions (if) ou des boucles (for) dans notre fichier HTML

```
<ion-spinner *ngIf="isLoading" item-start name="dots"></ion-spinner>
```

```
<ion-list *ngIf="list && !isLoading">
  <button
    ion-item
    *ngFor="let item of list"
    (press)="delete(item)"
  >
    {{ item.relationship_type.name }}
  </button>
</ion-list>
```

Dans cet exemple, la directive *ngIf permet d'afficher une animation d'attente si la variable (définie dans le fichier JavaScript) est à True

Nous pouvons également mettre plusieurs conditions facilement comme pour la liste qui ne sera affichée uniquement si isLoading est False et que la "list" (variable toujours gérée en JavaScript) est également présente.
Le *ngFor lui servira à créer plusieurs bouton (un par élément de la liste) et chaque donnée de la liste prendra le nom de variable item comme désigné ainsi. Cela servira pour afficher la propriété relationship_type.name à chaque occurrence de bouton et surtout lui lier comme il faut le bon objet à supprimer.
Pour lier une méthode à un événement (un clic, une longue pression, ...) il suffit d'indiquer à quoi nous voulons réagir (ici une longue pression sur l'élément bouton) et lier à la méthode JavaScript (delete() dans le cas présent) avec, s'il le faut des paramètres.

Autre grand point important d'Angular, le two-way data-binding. C'est le fait de refléter automatiquement les changements de variables entre le contrôleur (en Typescript) et la vue (en HTML). Principe très puissant réduisant le coût de développement car nous ne devons plus manuellement changer la valeur d'une variable dans l'affichage si elle est changée dans le javascript et vice versa.
Parlons également de l'injection de dépendance, un design pattern où nous créons des singletons (aussi appelé service) envoyant des données aux différents composant. Cela évite la duplicité du code et en isolant les services (utilisés pour appeler l'API) les tests automatiques sont simplifiés.

Ionic allie ainsi la puissance de Cordova pour interagir avec les plateformes mobiles et la structure de développement d'Angular afin d'avoir une architecture cohérente limitant la répétition de code et permettant également aux habitués du web avec Angular de vite retrouver leurs pratiques.

Teston dès à présent les joies du développement avec Ionic

L'initialisation

Pour commencer un projet Ionic, nous devons passer par la ligne de commande. En installant via NPM Ionic de manière globale ses outils, une floppé de commandes nous sont fournis (on appelle cela la CLI). La commande "ionic start" nous fait apparaître un questionnaire demandant le nom du projet et nous demande ensuite de sélectionner un template pour commencer. Les templates nous permet d'avoir déjà un squelette de navigation en nous proposant! un menu apparaissant sur demande sur le côté ou le menu en bas de page avec des icônes. Ionic nous propose également le template "super" qui fournit de nombreux exemple et bonne pratique pour le développement, très utile quand on débute et que nous sommes perdu.
En ce qui concerne l'architecture fourni de base, nous irons vite l'améliorer car au début Ionic ne fait pas la différence entre les pages et les simples composants. Mais lorsque nous générons automatiquement ceux-ci (merci au CLI) un dossier pages et composant apparaîtront automatiquement (alors autant le faire directement).
La documentation sur le site d'Ionic est également bien fournie avec à chaque fois des exemples de codes facilement visible et le template "super" lors d'une création de projet aide à avoir une belle vue des possibilités.

L'affichage et la gestion d'un formulaire

Utilisant Angular, nous développerons le style via le CSS, l'interaction en TypeScript et la forme dans un document HTML mais en employant des balises propre à Ionic. Nous pouvons toujours nous servir de balise propre à HTML mais cela est fortement déconseillé.
Chaque composant Ionic commence avec ion dans le nom et permet d'avoir directement des éléments que les utilisateurs mobiles sont habitués à voir. Citons par exemple ion-fab, qui permet d'avoir un bouton rond avec une icône restant toujours au même endroit de la page (très souvent en bas à droite) ou ion-datetime qui fait apparaître un sélectionneur de date que l'on peut dérouler pour choisir le jour voulu.
En ce qui concerne la structure des pages, Ionic cherche toujours à se rapprocher du web et propose ainsi qu'on puisse mettre un header et un footer si besoin. Le component ion-content est là pour y mettre le contenu de la page. Pour gérer la position des éléments, Ionic recommande grandement d'utiliser le système des grilles. Ceci a été popularisé par Bootstrap, bibliothèque CSS, et nous permet de voir l'affichage comme un damier de jeu d'échec où nous pouvons ainsi positionner et délimiter les éléments bien plus facilement.
Comme vue précédemment, nous avons des directives que nous pouvons placer directement dans les balises HTML de nos composants.
Cela permet de facilement afficher ou cacher des élément de la page sous certaines conditions.

Explorons maintenant la gestion d'un formulaire. Angular nous fournit le service FormBuilder nous permettant via une variable de décréter les noms de champs qu'on a besoin et d'y ajouter une validation

```
this.relationshipForm = this.fb.group({
  name: ["", Validators.required],
  relationshipTypeId: ["", Validators.required]
});
```

Dans l'exemple nous avons un formulaire pour une relation qui va avoir un nom (requis) et un id de type de relation (également requis).

```
<form [formGroup]="relationshipForm">
```

Dans l'HTML, via la balise form (non spécifique à Ionic ou Angular) on lui lie la variable.

```
<ion-select
  interface="popover"
  formControlName="relationshipTypeId"
>
  <ion-option
    *ngFor="let item of relationshipTypeList"
    [value]="item.id"
    >{{ item.relationship_type.name }}</ion-option
  >
</ion-select>
```

Nous lions ensuite chaque élément de la variable de formulaire à des components Ionic permettant de former un formulaire. Ici une liste d'élément à choisir. La directive fromControlName permet de savoir à quoi lier l'élément dans le formulaire. Et cela se fait automatiquement grâce au two way data-binding! Je n'ai pas besoin de rajouter un événement qui à chaque sélection change la variable ou à chaque entrée clavier renouvelle le contenu de cette variable dans le cadre d'un champ de texte.

```
<button
  type="submit"
  color="secondary"
  [disabled]="!relationshipForm.valid"
>
```

Une autre grande puissance des formulaires en Angular, c'est via sa validation, ma variable possède plusieurs propriété afin de vérifier des conditions, comme ici je peux rendre un bouton indisponible si le formulaire n'est pas valable. Très pratique pour empêcher d'actionner des événements (comme un ajout dans la base de données) si les conditions ne sont pas requises et pour également indiquer des messages d'erreurs ou de préventions.

La pagination

Ionic propose deux manière de gérer la pagination des pages.

La première n'est nul autre que la gestion à la Angular, le routing en y important le module Router. Pour aider à comprendre ce principe, il faut voir notre application comme un site web (cela tombe bien, c'est souvent ce qu'essaie de faire Ionic) avec différente URL. Nous définissons dans le module de base de l'application toutes les routes existantes et les lions à un component pour qu'il sache quoi afficher à tel ou tel appel de route. L'avantage de cette manière de faire est l'architecture de l'application qui est visible plus facilement et rapidement! Nous pouvons également gérer des redirections dans certains cas que nous pouvons programmer (pratique pour une gestion des droits efficaces).
Afin de naviguer entre les pages (les écrans!) nous pouvons dans l'HTML ajouter la directive routerLink qui prendra en paramètre la route voulu dans les boutons ou dans les balises <a> servant de base au web pour naviguer.

La seconde solution repose sur un principe plus orienté application mobile où l'on y définie qu'elle est la page racine de l'application et à tel ou tel ouverture de nouvelle page, si celle-ci remplace l'ancienne dans la racine ou si elle s'ajoute dans la pile de page ouverte.
Nous passons via le service Nav du paquet Ionic et sommes obligés pour le début de définir un des composant page comme racine de l'application. Pour ajouter une page dans la pile, nous utiliserons la méthode push avec en paramètre le composant voulu. Pour revenir en arrière, ça sera la méthode pull, cela fonctionne comme une liste dans la plupart des langages! La différence entre changer la page racine et empiler les pages, c'est que si nous nous trouvons sur une page racine et nous cliquons sur le bouton retour des smartphones Android ou Windows Phone, nous quittons l'application tandis qu'avec une pile, ce même bouton physique servira

simplement de pull. Le désavantage de cette solution est qu'au niveau du code l'architecture des pages est moins lisible.

L'interaction avec une API

Afin de récupérer des informations du backend, ou d'y en envoyer, nous utilisons le service HttpClient fourni de base par Angular. Nous y bénéficions des méthodes get, post, put et delete permettant d'exécuter ces requêtes vers un web service et nous renvois un observable avec lequel nous traitons les informations de manière asynchrone.
Ces méthodes prennent en paramètres la route du service web ainsi que de nombreux paramètres, comme des headers (dont j'ai besoin pour y faire transiter le token d'authentification) ou un body utile pour envoyer les données relatif aux requêtes POST et PUT.
Profitons également d'Angular pour utiliser les HttpInterceptor. Ceci nous permet d'intercepter toutes les requêtes vers un web service que nous faisons avant et après l'envoie de celui-ci. Dans notre cas, il sera utile en ajoutant à chacune de nos requêtes HTTP un header contenant le token d'authentification de la personne (excepté bien évidemment pour la connexion).

Le multilinguisme

Simplicité, c'est ainsi que je définirais l'internationalisation dans Ionic et dans Angular de manière plus globale.
Il existe une bibliothèque populaire qu'est ngx-translate qui nous facilite grandement la tâche.
Les dictionnaires de traductions sont des fichiers json tous placés dans le même dossier. Nous nommons chaque fichier de dictionnaire par l'abréviation du code langue IETF (https://fr.wikipedia.org/wiki/Mod%C3%A8le:Code_langue) de la langue utilisé. Dans mon cas j'ai donc un fichier fr.json et en.json. Ces fichiers json fonctionne en clé valeur, nous définissons une clé (qui nous sert d'identifiant unique) qui sera la même dans toutes les langues et une valeur qui sera la traduction propre à chaque langue.

```
"General": {
  "Welcome": "Bienvenue",
  "Add": "Ajouter",
  "Update": "Mettre à jour",
  "Cancel": "Annuler",
  "Confirm": "Confirmer",
  "AreYouSure": "Êtes vous sur?",
  "Error": "Erreur",
  "InWorking": "En cours de construction",
  "Date": "Date",
  "Both" : "Les deux"
},
```

```
"General": {
  "Welcome": "Welcome",
  "Add": "Add",
  "Update": "Update",
  "Cancel": "Cancel",
  "Confirm": "Confirm",
  "AreYouSure": "Are you sure?",
  "Error": "Error",
  "InWorking": "In working",
  "Date": "Date",
  "Both" : "Both"
},
```

Une fois la bibliothèque installé, nous devons préciser dans le app.module l'emplacement des traductions. Il est conseillé de les mettre avec les assets et faire

un dossier nommé i18n (Internationalization). C'est une convention et lorsque nous mettons notre application en production, tous les fichiers fixes (images et dictionnaires) seront ainsi au même endroit0
L'initialisation se fait dans le component (controlleur) global de l'application, app.component.ts, avec le service Translate fourni par la bibliothèque, nous lui précisons les langues présentes dans un tableau de string ("en" et "fr" pour nous), indiquons la langue par défaut (si une clé manque dans une langue, l'application ira chercher la traduction de la langue mise par défaut).
Pour afficher une traduction, dans l'html il suffit d'y indiquer la clé voulu avec le pipe "translate"

```
{{ "General.Add" | translate }}
```

Si nous voulons une traduction dans un controlleur javascript, c'est un peu plus compliqué, cela se passe en asynchrone où nous devons accéder au service Translate, à sa fonction get qui prend en paramètre la clé de traduction voulu (en string). C'est pour moi le seul point noir, devoir passer en asynchrone juste pour récupérer un string.
Le changement de langue est tout aussi simple, nous appelons une fois de plus le service Translate et lui indiquons que nous utilisons à présent une autre langue que celle par défaut en passant le code IETF correspondant. Ainsi, les valeurs changent automatiquement dans l'HTML. Pour les traductions dans les contrôleurs, nous devons recharger ces parties manuellement quand un changement est repéré.

Afficher des notifications

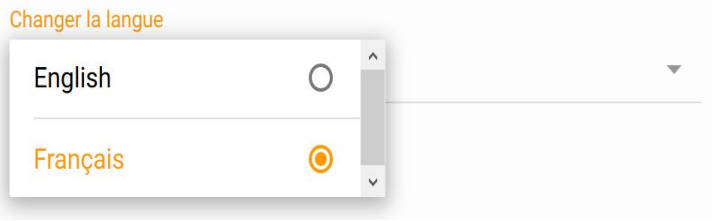
Ionic fournit des accès natif aux propriétés smartphone tel la caméra, le bluetooth ou les notifications via Cordova.
Dans le cas des notifications, la bibliothèque ionic-native fournit directement un module LocalNotifications qui nous permettra de réaliser notre but.
La première chose à faire pour afficher à l'utilisateur une notification est de voir si nous en avons la permission (en effet, sur nos smartphone de base les applications n'ont aucun droit natif, c'est à l'utilisateur d'autoriser tel ou tel accès). Si nous ne l'avons pas, nous pouvons afficher un pop-up le demandant, sinon, nous pouvons commencer à construire l'objet localNotification qui va prendre obligatoirement trois propriété. Premièrement un id servant à le retrouver si nous voulons l'effacer sous conditions. Deuxièmement le texte à afficher et enfin un trigger pour l'enclencher. Mais nous pouvons également lui mettre un son spécifique ou une icône particulière (par défaut cela sera celle de notre application).

Essayons d'être présentable

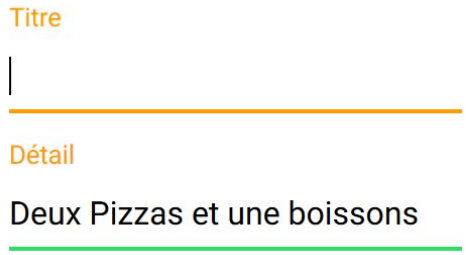
Si vous êtes un développeur web (ce qui est mon cas), vous serez comme un poisson dans l'eau dans Ionic.
Tout le design se passe via des propriété CSS classiques, aucun piège.
Ionic propose également de base une palette de couleurs prédéfini dans des variables pour les éléments primaires, secondaires, de danger, réussi, blanc et noir qui automatiquement auront un dégradé et que nous pouvons changer facilement via cet outil https://ionicframework.com/docs/theming/color-generator de générateur

de couleur nous montrons en direct à quoi cela ressemblerait sur notre application. Autre avantage, les composants d'Ionic se base essentiellement en terme de design sur Material Design, qui est un ensemble de règles esthétique proposé par Google depuis 2014 et qui sans être officiel dans le milieu du mobile, à prit une place considérable dans le coeur des développeurs mais aussi des utilisateurs.

Ainsi, il est très aisé de rapidement proposer un rendu tape à l'oeil sans se prendre la tête et surtout sans être un professionnelle du design (ce qui est rarement notre cas en développement)
Voici par exemple une liste déroulante pour sélectionner un item



Ou deux champs de textes, avec le label qui se met à la place du texte tant que celui-ci est vide pour ensuite se déporter au-dessus.



Ce sont tous des petits détails rendant la partie design du développement Ionic un plaisir.
Notons tout de même un désavantage à ce genre de pratique, cela peut amener à ce que toutes les applications se ressemblent trop.

Déploiement

Nous avons deux manière de faire.

Si nous possédons Android Studio (ce qui est souvent le cas pour avoir les outils de compilation et d'émulation) nous pouvons ouvrir le dossier Android générer par Ionic et de là compiler notre produit en .apk directement.

L'autre option est de passer via les lignes de commande.
"ionic cordova build android --prod --release" nous permet de générer un apk qui se situera dans le dossier "platforms/android/build/outputs/apk". Malheureusement cela n'est pas tout, il faut également signer l'exécutable (sinon la plupart des appareils refuseront de l'installer) en lançant la commande java "keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000". Nous avons déjà trouvé plus intuitif comme procédé.

Autres points importants?

Un avantage d'Ionic dans le développement est qu'on peut également tester notre produit sur un navigateur web. Bien évidemment on ne peut pas tester les accès natifs aux smartphones mais cela s'avère très utile pour voir le rendu graphique implémenté et y apporter des modifications plus rapidement sans passer par un émulateur (et la compilation qui va avec) plus lourd. C'est également un avantage si on souhaite déployer une application smartphone et sa version web!

Ionic propose également des outils payants. Nous avons AppFlow permettant de faire du DevOps en ayant un dashboard pour automatiser des tests avant chaque release, en fournissant un Git et des outils de travaux collaboratif .
Il vend également un IDE fait maison (Ionic Studio) où nous pouvons créer une interface sans toucher en code et proposant des facilités propres à Ionic.

Le 23 janvier, Ionic à fait son annonce de la sortie de sa version 4.
Comme souvent lors de ce genre d'événement, les performances des outils y sont améliorées et de nouveaux composants sont sorti afin de nous faciliter la tâche.
Mais le clou du spectacle fut l'annonce de la prise en charge (en beta pour le moment) des bibliothèques ultra populaires que sont React JS et Vue JS pour ceux qui ne veulent pas d'Angular.

Xamarin

Au début, Xamarin était une entreprise fondée en 2011 qui a créée Mono, projet rendant les applications C# multiplateforme (très apprécié par les développeurs Linux et Apple). Et au fur et à mesure de l'avancement du projet, ils ont lancé Mono Android et MonoTouch (pour iOS) et unifient ces plateforme de développement en 2013 afin de pouvoir avoir des écrans utilisables sur Android et iOS avec le moins de code possible. C'est en 2016 que Microsoft, afin de plus être à la traîne dans le monde du mobile, racheta cette entreprise afin de l'intégrer dans Visual Code (pour ne plus avoir besoin de l'IDE maison de Xamarin) et surtout rendre le projet Open-Source!

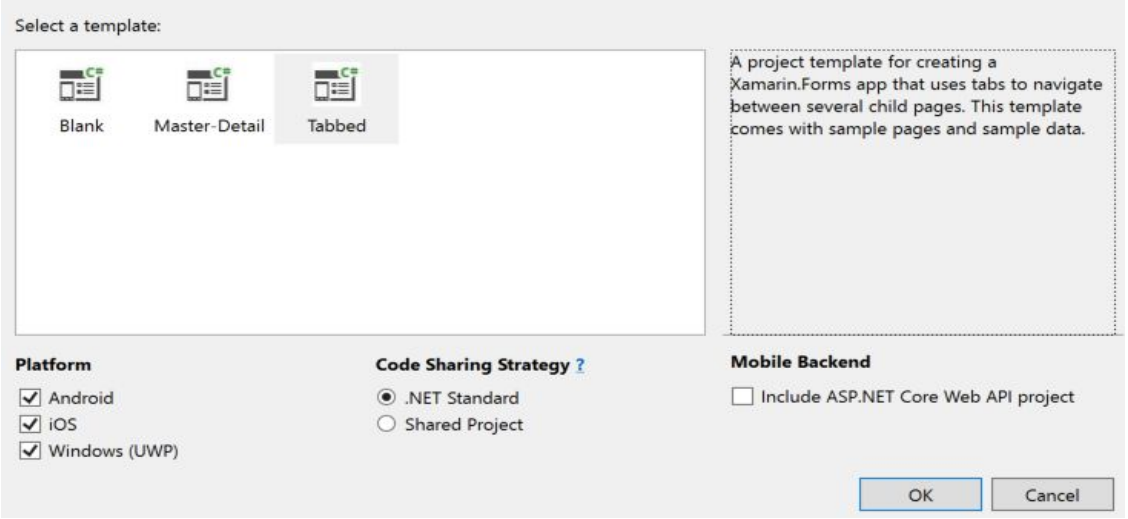
Xamarin utilise donc C# et le framework .NET de Microsoft afin de faire tourner des applications sur les appareils mobiles. Les habituées des technologies Microsoft peuvent à présent faire des applications smartphones sans apprendre tout un nouveau langage depuis zéro. Pour la partie affichage pure, Microsoft a développé XAML qui tout comme HTML est une surcouche d'XML afin de faire des interfaces graphiques. XAML est aussi utilisé pour faire du développement d'application UWP (Universal Windows Platform) ce qui permet aux personnes ne jurant que par Microsoft de garder leurs marques.

Microsoft recommande d'utiliser l'architecture MVVM, Model–view–viewmodel, bien que nous pouvons faire à notre guise.
Le modèle représente un objet avec de réel données dedans. La vue elle permet d'avoir un affichage. Quant au viewmodel il sert à automatiser les données entre un model et une vue et changer en direct si besoin (nous appelons cela le binding)

L'initialisation

Microsoft oblige, c'est via Visual Studio que l'on va démarrer le projet. Il est certes possible de le faire ailleurs et jouer avec la CLI ou gérer les fichiers manuellement, mais cela serait une perte de temps. Au lancement du projet, après sélection de l'option Cross Platform project, nous pouvons choisir pour quelle plateforme sera notre application (Android, iOS et Windows, il n'est pas obligé de sélectionner les trois) mais surtout décider si nous commençons avec une application vide, une vue avec pagination d'un menu déroulant ou avoir un menu fixe en haut ou bas de page permettant de passer de l'une à l'autre en un clic.

Visual Studio nous propose aussi de lier si nous le souhaitons directement un web service (en ASP.NET, le framework web de Microsoft).



Une fois la solution initialisé, Visual Studio nous fournit un projet de base et un projet par plateforme ciblé. Tout se joue dans le projet racine, les autres étant là si nous avons besoin de code spécifique à une seule des plateformes.

Les dossiers pour respecter le MVVM sont présent avec déjà des exemples si nous n'avons pas sélectionner un projet vide.
L'exécution se fait en choisissant entre un émulateur Android, Windows Phone, iPhone (non disponible pour moi) ou en mode desktop. Le mode desktop sera souvent utilisé car plus rapide que lancé sur un émulateur.
En ce qui concerne la documentation, nous avons à faire à Microsoft et sa documentation fouillis qu'on retrouver sur plusieurs pages différentes et souvent incomplète et avec parfois des exemples plus à jours (j'ai eu quelques fois le soucis d'avoir une propriété dans un exemple n'existant plus!).
Xamarin à un site Xamarin University avec des vidéos explicatives et parfois le code source qui va avec. Une section payante est également présente pour y avoir des certifications ou la possibilité de parler à des experts pour nous aider.

L'affichage et la gestion d'un formulaire

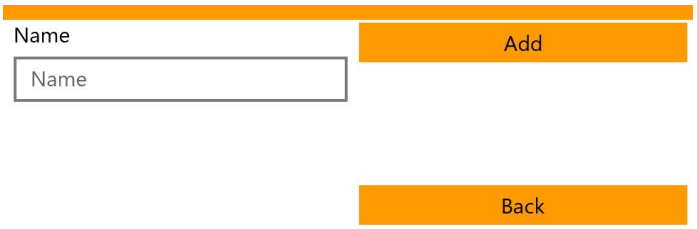
Le XAML se limite à ce que veut bien nous fournir Microsoft et nous voilà déjà embêté par les restrictions. Nous n'avons par exemple pas de Check ou Radio Button. Soit nous chipotons en créant du code nous même pour que cela y ressemble soit on espère trouver notre bonheur dans les librairies fournies par la communauté. Comparer à ce que nous fournit de base Ionic, la limitation se fait vite ressortir. Limitation assez incompréhensible à mes yeux non seulement pour une si grosse entreprise qu'est Microsoft mais aussi comparer à un projet UWP (pour faire des applications Windows 8 et 10), qui se base également sur XAML et qui lui est très complet)

Pour la séparation du code, un fichier XAML va se diviser en deux, la partie pure XML où nous y indiquerons les balises représentant nos éléments et la partie CS (C#, le langage phare de Microsoft) qui lui contiendra le code d'interaction (avec les boutons ou tout simplement l'initialisation de la page) et la liaison avec le vue-model.
Concernant le vue-model (qui est un autre fichier CS), on y trouvera les variables utilisées dans l'affichages. Chaque variable public devra avoir son équivalent en privé (par convention on y met un '_' devant le même nom) et les variables public auront aussi un getteur pointant vers son équivalent privé afin de s'afficher et un setteur pour modifier son contenu privé. Je trouve ce genre de procédé fort lourd en développement de devoir ainsi faire des répétitions.
La page en elle-même se forme dans la balise ContentPage qui contient d'abord des propriétés utiles pour les importations (utile pour le multilinguisme) et ensuite la balise ContentPage.Content où nous y disposerons les éléments. Nous aurons comme choix pour la dispositions de faire des grilles (Grid) afin d'être précis dans la répartition visuel soit une pile d'élément (StackLayout) s'affichant les uns en dessous des autres. Évoquons aussi l'AbsoluteLayout (où nous fixons les éléments) et la ScrollView qui elle n'est qu'une liste.
Concernant la grille, son positionnement est différent d'Ionic où nous mettons des rangs que l'on découpe ensuite pour former un rendu responsif. Ici nous définissons au tout début de la grille le nombre de colonnes et de rangées et indiquons la taille à chaque fois. La valeur * elle permet de prendre toute la place possible et si plusieurs éléments ont cette valeur, ils se partageront équitablement cette place.
Après avoir défini notre grille, pour mettre les éléments à l'intérieur, nous devons leurs indiquer à quelle place de la colonne et de la rangées ils sont avec les propriétées Grid.Row et Grid.Column (attention comme un tableau on part de zéro et ça sera d'ailleurs la valeur par défaut).
Je n'ai pas l'habitude de ce genre de positionnement. Cela m'a semblé au départ assez peu intuitif pour voir quel élément est à quelle place. Mais finalement cela s'avère assez efficace et pas prise de tête.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <StackLayout Grid.Column="0">
        <Label Text="Name"/>
        <Entry Text="{Binding Name}" Placeholder="Name"/>
    </StackLayout>

    <Button Text="Add" Clicked="Add_Button_Clicked" VerticalOptions="Start"
        BackgroundColor="{StaticResource Primary}" Grid.Column="1"/>

    <Button Text="Back" Clicked="Back_Button_Clicked" VerticalOptions="Start"
        BackgroundColor="{StaticResource Primary}" Grid.Row="1" Grid.Column="1"/>
</Grid>
```



Faisons maintenant un formulaire. Comme évoqué un peu plus haut, on remarque dès le départ que nous ne disposons pas de Radio et Check Button pour faire des sélections simples et multiples. Si nous voulons juste sélectionner un élément d'une liste il y a le Picker qui est présent pour nous (cela affichera un pop up avec la liste voulu) mais cela reste assez pauvre.
Pour une saisie de texte, nous avons la balise Entry auquel on lira (le fameux binding) une variable.
Malheureusement contrairement à Ionic, il n'y a pas d'objet formulaire rendant le tout plus simple. Je ne peux donc de manière efficace cacher le bouton "Valider" du formulaire tant que les conditions ne sont pas remplies.
Je me contente alors de laisser le bouton tout le temps cliquable et faire les vérifications quand l'utilisateur pressera le bouton en affichant un message d'erreur si besoin.

```
<StackLayout IsVisible="{Binding IsBusy, Converter={Helpers:InverseBoolConverter}}">

    <Label Text="Adding a Relationship"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="CenterAndExpand" />
    <Entry Text="{Binding Relationship.Name}"
       Placeholder="Relationship type name" />

    <Picker
        Title="Select a relationship type"
        ItemsSource="{Binding RelationshipTypeList}"
        ItemDisplayBinding="{Binding RelationshipType.Name}"
        SelectedItem="{Binding SelectedRelationshipType}"
        />

    <Button Margin="0,10,0,0"
        Text="Add"
        Clicked="Add_Clicked"
        BackgroundColor="{StaticResource Primary}"
        VerticalOptions="CenterAndExpand"
        TextColor="White" />
</StackLayout>
```

La pagination

En partant d'un projet avec un menu, la pagination est d'une grande simplicité. nous avons la page principal qui reprend le menu et un container pour y mettre les pages qui seront choisi via le menu. A la sélection d'une nouvelle page, celle-ci va se mettre en tant que RootPage (objet de la page principal) et faire également

changer le titre du menu. Si nous voulons une page qui se met au-dessus de l'écran principal, cela fonctionne comme une pile via l'objet Navigation où l'on lui précise la page à afficher au-dessus de tout. Cette page peut se retirer en l'appelant manuellement toujours grâce à l'objet navigation ou automatiquement avec le bouton physique retour des téléphones voir encore avec le bouton formant une flèche vers la gauche en haut de l'écran.

Un projet avec les tabulations pour changer de page principal? Encore plus simple! Dans notre page principal nous mettons le XAML TabbedPage avec ses enfants qui seront des onglets avec un titre, une icône si besoin et la page correspondante.

```
<NavigationPage Title="Relationship Type">
    <NavigationPage.Icon>
        <OnPlatform x:TypeArguments="FileImageSource">
            <On Platform="iOS" Value="tab_about.png"/>
        </OnPlatform>
    </NavigationPage.Icon>
    <x:Arguments>
        <views:RelationshipTypesPage />
    </x:Arguments>
</NavigationPage>
```

L'interaction avec une API

Le traitement des requêtes se fera dans le view-model d'une page (limitons le code dans la partie CS des fichiers view). Celui-ci appellera le service voulu (qui est une classe à part créée par nos soins).
Pour appeler mon webservice j'utilise la librairie (non inclus de base) RestSharp qui me fut conseillé par des collègues. Celle-ci réduit drastiquement le nombre de ligne à faire (et réduisant aussi par la même occasion le risque d'erreur). La librairie me fournit l'objet RestClient qui prendra en paramètre l'url de base. Ensuite vient l'objet RestRequest où je peux y inclure le type de méthode (GET, POST, …) mais également le body et le header.
Il me reste à utiliser la méthode Execute de RestClient prenant en paramètre le RestRequest conçu juste avant et je reçois ainsi une RestReponse contenant la réponse du webservice.
Il est également possible de réaliser tout cela en asynchrone, ce qui est toujours conseillé.

Le multilinguisme

Pour rendre son application multilingue, Microsoft à sa bibliothèque interne Globalization qui nous y aide.
Nous devons mettre des fichiers 'resx' dans un dossier Localization à la racine du projet. Ce sont des fichiers qu'on peut éditer en xml ou via Visual Studio qui nous met à disposition un éditeur. Comme souvent, cela fonctionne avec une clé qui sera la même dans toutes les langues et une valeur qui elle changera en fonction du fichier 'resx' que l'on édite.

	Name	Value
▶	ChooseLang	Changez votre langue
	HelloWorld	Bonjour le monde
	Home	Accueil
	Relationship	Relation
	RelationshipType	Type de relation
*		

A la racine du dossier Views, nous devons créer un fichier TranslateExtension où nous y initialiserons le tout en précisant la langue par défaut. Et dans chaque fichier XAML de vue où nous aurons besoin d'une traduction, nous indiquerons dans le ContentPage l'assembly utilisé et le label voulu sera récupéré en indiquant la clé du dictionnaire.

Attention cependant, à l'affichage Xamarin prendra soit une langue qu'on lui impose soit la langue du téléphone si elle est parmi celle que l'on propose. On peut changer en cours d'exécution de l'application la langue mais elle ne se changera pas automatiquement partout, il faut à chaque fois recharger les pages déjà ouvertes avant (ou simplement relancer l'application).

Afficher des notifications

Fonctionnalité pourtant importante pour une application sur mobile, il n'est pas aisé d'afficher une simple notification avec Xamarin. Tout d'abord, les notifications doivent être gérées à part pour iOS et pour Android! Ce qui enlève l'intérêt du cross platform.
Heureusement nous pouvons comme souvent compter sur la communauté des développeurs open-source. Il existe un plugin nuget, "Xam.Plugins.Notifier", qui permet de faire cela très simplement. Il suffit de faire appel à l'objet CrossLocalNotifications, de lui assigner un titre et un corps de message et l'affaire est jouée! Nous pouvons également lui définir un délai avant d'apparaître.
C'est assez dommage que Xamarin ne propose pas tout simplement cela de base.

Essayons d'être présentable

Dans le fichier App.xaml (point d'entrée de l'application) nous pouvons y définir un un dictionnaire de clé / couleur pour réutiliser dans les éléments la même couleur (et la changer qu'une seule fois si besoin) et nous pouvons également définir des propriétés globales afin d'être utilisable partout. Si nous voulons par exemple que tous nos boutons aient un margin de 10 pixels, nous mettons le code <Thickness x:Key="ButtonMargin">10</Thickness> avec comme clé "ButtonMargin" et en appliquant à mes boutons Margin="{StaticResource ButtonMargin}" il prend la valeur voulu.

```
<Application.Resources>
    <ResourceDictionary>
        <!--Global Styles-->
        <Color x:Key="NavigationPrimary">#FF9A00</Color>
        <Color x:Key="Primary">#FF9A00</Color>
        <Color x:Key="Secondary">#0B72FF</Color>
        <Style TargetType="NavigationPage">
            <Setter Property="BarBackgroundColor" Value="{StaticResource NavigationPrimary}" />
            <Setter Property="BarTextColor" Value="White" />
        </Style>
    </ResourceDictionary>
</Application.Resources>
```

Je préfère tout de même la méthode du web où en une seule classe permet de définir plusieurs valeurs. Je suis de plus habités aux propriétés HTML/CSS mais cela n'est pas non plus insurmontable d'avoir un rendu correct en Xamarin. Le style donné par défaut aux divers éléments est dans le rendu standard d'Android ou iOS.

Déploiement

Voilà un point fort de Xamarin. Avec les outils très clair de Visual Studio, il suffit de faire un clic droit la solution du projet Android, sélectionner archiver ce qui va générer l'APK. Il est également possible de signer l'application et de créer une clé de signature aisément via l'IDE. Et enfin, cerise sur le gâteau, si nous avons un compte Google pour cela, nous pouvons directement publier notre application sur le Google Play!

Autres points importants?

Un point qui m'a beaucoup dérangé dans mon développement avec Xamarin, c'est le manque de clarté de la documentation.
Première, on ne sait pas toujours où se trouve tous les éléments. Il y a plusieurs pages de documentation (que ce soit la Xamarin University ou la Doc Microsoft). On ne sait pas toujours où donner de la tête.
Ensuite, les exemples non à jours. J'ai eu quelques fois des exemples de la documentation officiel qui si je la suivais strictement me faisait des erreurs car tel ou tel propriété n'existe plus!
Nous ne sommes jamais informés non plus des importations (use) à devoir utiliser avec certains outils (pour les notifications par exemples).
C'est un soucis que j'ai toujours eu avec les diverses documentations de Microsoft et cela est pénalisant.

Autre point à signaler, Microsoft base sa communication promotionnelle sur le fait qu'en un seul code nous allons avoir une application sur trois plateformes différentes (iOS, Android et Windows pour rappel). Parfait, c'est ce que nous cherchons!
Cependant souvent dans le documentation il y a des exemples de codes spécifiques pour tel ou tel plateforme et ce afin d'avoir une application plus optimisé pour l'un ou l'autre appareil. Ce qui est bien d'un côté mais respecte plus du coup le principe d'un seul code pour les contrôler tous.
Et au niveau du style, il faut préciser certaines choses spécifiquement dans les dossiers Android et iOS. Les couleurs des menus par exemple, il faut également les

faire changer et dans nos fichiers de configuration de base mais aussi préciser les bonnes valeurs dans le fichier colors d'Android. Ce sont des petits détails, mais je trouve cela à nouveau pénalisant.

J'ai également pu tester Visual Studio 2019. J'ai créé un projet Xamarin vie cette mise à jour et à présent la version UWP n'est plus prise en charge directement. Cela veut dire que je ne peux plus tester facilement mon application comme un exécutable Windows mais je dois dorénavant d'office lancer un émulateur Android ou iOS. Processus bien plus lourd rendant les tests plus lents!

Cependant, Xamarin à sa version 4 en bêta depuis quelques temps et semble aller dans le bon sens, déjà avec une pagination mieux gérée via des routes (comme avec Ionic) et plus de moyen pour l'affichage. Tout n'est peut-être pas perdu.

Petit élément perturbant à préciser tout de même, Une page en Xamarin s'appelle une "Form". Oui, comme un formulaire en français. Ma recherche de tutoriel pour faire un formulaire en Xamarin fut donc compliqué à cause de cela.

React Native

Tout comme Ionic, React Native se base sur un framework web déjà existant pour former des applications mobiles. Cependant, React Native va puiser dans les ressources de React JS. Ces deux outils sont tous les deux maintenus par Facebook. C'est depuis 2013 que React JS existe, le principe de cette bibliothèque est de créer des composants le plus réutilisable possible et le but est de pouvoir utiliser React indépendamment du reste d'une application ce qui veut dire qu'il n'y a pas des règles ou une structure prédéfini à avoir pour l'utiliser (au contraire d'Angular) afin d'être le plus flexible possible. C'est également pour cela que React JS fonctionne en JavaScript sans la surcouche TypeScript. L'avantage de ceci est que React est exploitable sur un projet déjà existant sans devoir tout refaire, on peut se permettre de l'intégrer étape par étape. Cela a permis son adoption très rapide dans le coeur des professionnelles qui ainsi ne devaient pas repartir de zéro.
Comme dit précédemment, React fonctionne avec le principe des composants, pour en créer, il nous faut une classe JavaScript étendant la classe component de base de React. Cette classe aura un state qui est l'initialisation des variables (sous forme de json) et un render, qui retourne un HTML.

```
  super(props);
  this.state = {
    relationshipTypeName: "",
    isLoading: false,
    relationshipTypeList: [],
    modalVisible: false
  };
}
```

Contrairement à Ionic et Angular, React ne fait pas de two-way data binding mais uniquement du one-way, cela veut dire que la vue en elle-même réagit aux changements du "contrôleur" mais pas le contraire. Cela est bien évidemment plus performant à l'exécution de base mais au niveau du développement cela est (selon moi) pénalisant car lors d'un formulaire par exemple, si je veux mettre à jour la variable d'un champs texte, je devrais y mettre un listener qui change manuellement cette variable à chaque repérage changement.
React met également toute les éléments d'un composant dans un seul fichier, nous n'avons donc pas un fichier CSS, HTML et JS pour un composant mais un unique fichier JavaScript mettant la logique, le rendu et la style au même endroit. Cela est voulu car un composant se doit d'être le plus petit possible.

```
export default class TitleText extends Component {
  render() {
    return (
      <View style={[styles.container, styles.horizontal]}>
        <Text style={styles.text}>{this.props.text}</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center"
  },
  horizontal: {
    flexDirection: "row",
    justifyContent: "space-around",
    padding: 10
  },
  text: {
    color: Colors.secondaryColor,
    fontSize: 20
  }
});
```

De nouveau je ne suis pas un fan de ce procédé car on peut très rapidement avoir des fichiers très long où on s'y perd rapidement.
React Native, qui a été lancé deux ans après en 2015, va utiliser totalement le même principe que son grand frère React JS ce qui va faciliter grandement le plongeon dans le grand bain du développement mobile pour les fans de React JS.
La différence sera que React Native possède déjà des composants utilisables par nous et au niveau du style, cela ne reprend pas totalement les mêmes propriétés CSS que le web mais tout en restant très proches.

L'initialisation

React étant du JavaScript, c'est également via NodeJS que nous devons exécuter les lignes de commandes afin de mettre en place un projet React Native. La documentation officiel nous propose deux manières de débuter un projet.
Soit nous pouvons passer via l'outil NPM react-native-cli qui va nous mettre un projet sans aucune structure et juste un fichier JavaScript faisant office de première page d'application, soit via un autre outil NPM indépendant, Expo, qui lui nous proposera une base de navigation et trois écrans prédéfinis pour montrer des exemples (mais toujours pas de vrai structure). J'ai utilisé Expo pour la suite de cette découverte car Expo nous fournit des outils en plus pour avoir une application le plus proche possible du natif.
Au premier abord on peut se dire que nous laisser le choix entre deux méthodes d'initialisation est bien (on aime les choix dans la vie généralement) mais le soucis qui apparaîtra rapidement lorsque nous cherchons de l'aide sur internet est que la manière de faire d'Expo et React-Native-Cli sont différentes et certaines applications ou bibliothèques externes ne seront pas utilisables avec l'une des deux méthodes!
De plus le manque de structure (une partie pour les pages en tant que soit, une partie pour les composant réutilisable, pour les services, …) est quelques choses qui m'embête à chaque fois.

L'affichage et la gestion d'un formulaire

En ce qui concerne la mise en place des divers éléments sur une page, React fonctionne avec ce qu'on appelle les Flexbox (qui existe également dans le web via le CSS). Flexbox est pensé pour ne pas devoir se soucier de la grandeur des écrans, ce qui est pratique dans notre monde rempli d'écran de taille différente même pour les appareils mobiles.
Nous devons d'abord indiquer dans une balise view (qu'on peut comparer ça au Div d'HTML) que le style sera un flex et via la propriété flexDirection lui indiquer si cela sera un colonne (valeur par défaut) ou une rangée horizontal. Les éléments à l'intérieur de cette View seront ainsi misent l'un en dessous de l'autre ou l'un à côté de l'autre. Nous avons ensuite la propriété justifyContent pour définir comment se placeront les éléments à l'intérieur. De base ils seront collé l'un sur l'autre en partant de la bordure gauche (flex-start) mais nous pouvons les centrer (center), les faires débuter de la droit (flex-end) mais aussi mettre un espace entre chaque éléments (space-around et space-between).
Voilà ce que nous propose la documentation de React Native et cela s'avère très efficace.
Nous pouvons bien évidemment nous en passer et utiliser les mêmes types de placement que de Xamarin et Ionic (StackLayout ou Grid).

Tout comme Ionic, React va jouer avec des composants dans tous les sens pour former nos pages et ainsi notre application.
Pas de grand dépaysement donc, excepté que, nous devons importer à chaque fois tous les composants utilisés… Même les plus basiques et natif à React tel le composant Button, Text, View, StyleSheet (pour notre CSS), ce qui fait que dès le départ, nous aurons une page avec beaucoup d'importation. Et bien évidemment attention à ne pas les oublier du coup sinon l'application ne fonctionnera pas, ce qui pour moi habitué d'Angular m'a joué beaucoup de mauvais tour.

React utilise donc le One-way-data-binding pour afficher des variables, ce qui veut dire que la vue réagira au changement du contrôleur, mais pas l'inverse.
Pour afficher une variable, nous l'initialisons dans le contrôleur en le passant dans l'objet state. Et dans la vue, il suffira d'aller le chercher toujours dans cet objet state!
En ce qui concerne les méthodes qu'on veut lier à une action, cela reste tout aussi basique. Nous créons donc la fonction dans le contrôleur puis si un composant possède une action (Button et son action onPress) nous faisons une fonction lambda qui appellera la méthode du contrôleur voulu. Nous pouvons également écrire la fonction directement ici mais ce n'est pas propre niveau pratique de développement.

```
<Button
  icon="add-box"
  mode="contained"
  onPress={() => {
    this.setModalVisible(true);
  }}
>
```

Jusque-là, tout est simple et même agréable à utiliser. Mais le One-way-data-binding devient limité en terme de confort quand on veut faire un formulaire car bien souvent on voudrait bien que le contrôleur se met à jour tout seul quand l'utilisateur rentre des informations. Cela est utile pour valider les données écrites et autoriser tel ou tel action (bien souvent appuyer sur le bouton valider qui enverra le formulaire au backend). Pour régler cela nous allons mettre dans les composant de formulaire (TextInput pour un texte par exemple) l'action onChangeText qui appelle la méthode setState (méthode toujours présente dès que notre composant à un state avec des variables dedans) qui avec en paramètre son variable que l'on va changer va mettre à jour le state.

```
<TextInput
  label={i18n.t("General.Name")}
  value={relationshipTypeName}
  onChangeText={relationshipTypeName =>
    this.setState({ relationshipTypeName })
  }
/>
```

Et pour l'exemple du bouton cliquable uniquement quand le formulaire est valide, nous devons dans la propriété disabled du composant Button lui indiquer les règles ou passer via une méthode lui disant si c'est valide ou non, méthode qui sera appelé à chaque changement dans le formulaire.
Ceci n'est pas spécialement compliqué, mais c'est une énorme perte de temps de devoir à chaque formulaire jouer ainsi avec le state! Dans mon exemple d'application j'ai des formulaires de un ou deux champs, aucun soucis. Mais j'ai également un formulaire pour enregistrer un paiement qui contient six champs requis et je vais donc devoir pour chaque champs mettre en place le changement de la vue vers le contrôleur. Selon moi, ce genre de pratique est le plus gros point faible de React et m'empêche de pleinement profiter du framework de Facebook.

La pagination

En ce qui concerne la navigation, React Native va encore puiser dans ce qui a déjà été fait avec React JS et utiliser une bibliothèque commune, React Navigation. Celui-ci va fonctionner d'une manière semblable à ce que l'on fait avec Angular et Ionic. Nous définissons un "stackNavigator" qui contient un json avec comme clefs chaque nom que nous voulons donner à nos routes (servant d'id) et en valeur à quel component chaque nom est lié.
Ceci devra être fait dans le component initiateur de l'application (souvent nommé App) et servira ainsi de containeur pour tous les composants liés aux diverses routes.

L'interaction avec une API

Contrairement à Angular, React ne propose pas d'outils spéciaux pour faire appel à un webservice. Cependant JavaScript a déjà ce qu'il faut pour cela et React ne cherche pas à réinventer la roue à tout prix.
Il est ainsi assez aisé de chercher à communiquer avec un serveur distant en JavaScript via un objet fetch, nous devons de base uniquement lui donner un string comme paramètre qui est l'url que nous souhaitons appeler. Par défaut cela fera une

requête GET. Comme second paramètre nous lui passerons un objet avec comme propriété method qui sera GET, POST, PUT ou DELETE, un body si il faut (dans le cadre du POST et PUT) ainsi qu'un headers.
Ce Fetch peut s'exécuter de manière asynchrone (primordiale de nos jours) et nous pouvons ensuite transformer notre réponse en json.

```
export const getRoute = async route => {
  try {
    const bearer = await AsyncStorage.getItem("Bearer");
    return fetch(`${baseUrl}${route}`, {
      method: "GET",
      headers: {
        Accept: "application/json",
        "Content-Type": "application/json",
        Authorization: `Bearer ${bearer}`
      }
    }).then(response => {
      if (!response.ok) {
        verifyStatut(response);
      }
      return response.json();
    });
  } catch (error) {
    // Error retrieving data
  }
};
```

Le multilinguisme

Continuons les comparaisons avec Ionic en parlant du multilinguisme car il est à peine plus difficile à mettre en place qu'avec Angular.
Tout d'abord, nous aurons à nouveau des fichiers Json qui seront nommés chacun avec l'abréviation de la langue voulu (fr, en, nl, de, …) et des pairs clés - valeurs pour aller chercher les traductions. Jusqu'ici nous sommes en terrain connu.
Lors de mes recherches pour un outil React me permettant de gérer les langues, c'est le module NPM i18n-js qui est souvent ressorti.
Pour l'initialiser, il suffit de lui dire quelle fichier de langue nous voulons qu'il aille chercher dans le dossier "lang" et ensuite toutes les abréviations possible. Pour repérer la langue de l'utilisateur Expo a un objet Localization nous permettant de le retrouver avec la propriété locale. Tout cela est à faire dans le render() de l'application de base

```
render() {
  i18n.locale = Localization.locale;
  i18n.translations = { en, fr };
```

Pour ensuite ressortir la traduction voulu, dans un composant de vue nous devons re importer i18n-js (petit point faible par rapport à Angular où nous ne devons pas nous soucier de cela) et dans le render faire appel à la traduction voulu via la clés qui

nous retournera la valeur du fichier Json de traduction de la langue enregistré dans le i18n.local vue avant.

```
<Text style={styles.helpLinkText}>
  {i18n.t("General.HelloWorld")}
</Text>
```

Si nous voulons changer la langue en cours d'exécution de l'application, cela devient un peu plus complexe. En effet on ne peut pas changer la langue depuis n'importe quel composant mais uniquement depuis le composant conteneur (pour rappel, on l'appel par convention App). Il faudra donc à chaque fois faire appel à celui-ci via une méthode qu'on peut atteindre.

Afficher des notifications

Ça sera via Expo que nous pouvons afficher des notifications. Pour ce faire nous aurons besoin de ses outils Permissions (afin de savoir si l'utilisateur nous autorise à ce qu'on lui montre des notifications sur son appareil Android) et Notifications
Avant donc de mettre en place l'objet Notifications nous allons vérifier si nous avons ces droits d'accès natif. Si ce n'est pas le cas, nous pouvons les demander.

Essayons d'être présentable

Pour modifier le style de notre application, React va se rapprocher de ce qu'on fait sur le web avec des petits changements. Nous aurons dans le component une variable qui aura l'objet StyleSheet.create avec en paramètre un json de clé pour le nom de la classe de style et comme valeur un autre json avec les paramètre de style. Il est bien évidemment possible d'avoir un style global mais avec React on adore jouer le plus possible avec les components.
Pour qu'un élément du DOM adopte le style voulu, on lui met la propriété "style" qui sera le nom de la variable de style et la clé voulu.
Concernant les paramètres de style modifiable, alors qu'en CSS on utilise le kebab-style (font-size, background-color), ici on utilisera le camelCase (fontSize, backgroundColor).
Si nous sommes habités au style du web nous ne serons pas perdu.
Il n'y a malheureusement pas dans la documentation officiel un répertoire de toutes les valeurs de style changeable.

Déploiement

"expo publish", peut-on faire plus simple que cette commande à écrire pour avoir notre exécutable Android?
Il faudra ensuite faire signer l'APK qui a été généré mais cela est du ressort de Google et non de React Native, je vous invite à relire la partie déploiement d'Ionic car cela sera exactement la même méthode.

Autres points importants?

Je n'ai plus de points significatifs à relever concernant React. Durant la rédaction de ce travail il n'y a pas eu de grande annonce ou nouveauté prévu.

React Native et JS va continuer à être apprécié par de nombreuse personnes, probablement être de plus en plus utilisé car cela reste un outil pratique pour faire du multiplateforme.
Saluons tout de même le site de React Native avec un très bon guide pour prendre en mains ce framework et surtout des exemples que l'on peut changer en direct sur le site afin de voir rapidement ce que peut donner tel ou tel propriété!

Comparaisons récapitulatif

Je vais ici présenter sous forme de tableau les points forts et les points faibles de chaque frameworks, le tout ayant bien évidemment une part de subjectivité.

En voici le code couleur:
Très bon, agréable à exploiter
Rien de spécial à redire
J'y ressent des lacunes qui m'ont ennuyé
Catastrophique selon moi, cela m'a dérangé

	Ionic	Xamarin	React Native
Mettre les outils en place			
Initialisation d'un projet			
Qualité de la documentation			
Gérer la mise en forme d'une page			
Afficher une liste d'élément			
Rédiger un formulaire			
La pagination			
Communiquer avec un Webservice			
Multilinguisme			
Les notifications			
Jouer avec le style			
Déployer l'application			

Conclusion

Après ce long travail de veille technologique, je ressors assez déçu du développement hybride, il n'y en a pas un qui me fait dire "Il faut absolument l'utiliser! Il va écraser toute concurrence!", ils ont chacun d'assez gros défaut que ce soit en terme de performance ou de développement. Mais ils ont aussi chacun des choses plaisantes à proposer qui me fait garder espoir dans l'avenir de ce type de développement!
Au final j'en ressors avec deux conclusions

Développer spécifiquement pour mobile est-il toujours utile?

Non! Quand nous faisons un site web de nos jours, une attention particulière sur le responsive design (le fait d'être visible correctement sur tout type d'écran) est de plus en plus souvent porté. Et très souvent cela est suffisant pour les besoins des utilisateurs finaux. Qu'apporte les OS mobiles en plus? La possibilité d'émettre des notifications? C'est le nouveau spam depuis que les mails ne sont plus à la mode (et les navigateurs web permettent cela aussi maintenant). La localisation? Elle est également disponible sur le web.
Le meilleur exemple selon moi sont les applications des médias journalistiques. Qu'avons-nous besoin pour une telle application? Savoir lire des textes, observer des images et parfois des vidéos pour plus d'explication. Un navigateur internet permet il cela? Hé bien oui.
Je ne dis bien évidemment pas que développer distinctement pour mobile est une mauvaise chose, certaines applications sont moins user friendly dans leurs variantes navigateur web (je pense à Instagram pour poster directement une photo ou Skype pour faire des conférences vidéos) mais il est important de se poser la question avant chaque début de projet si oui ou non le développement mobile nous apporte un plus que si nous nous contentons du web.
Car au final, le vrai multiplateforme n'est-il pas le web tout simplement?

Développer en pure natif pour Android et iOS n'est pas toujours une perte de temps

Avant toute chose, rappelons comment se déroule le processus de création d'une application.
Une fois l'idée bien défini, une analyse se fait pour ressortir tous les besoins des fonctionnalités qui seront développées sans oublier la base de données.
Le backend pour interagir entre nos téléphones doit se faire également.
Ainsi que des maquettes à dessiner pour définir à quoi ressemblera la partie visible de l'application.

Tout ce que je viens de citer avant, qu'on développe une application hybride ou natif, ne sera réalisé qu'une seule et unique fois.
On se rend compte dès lors que la programmation de l'interface ne prend pas autant de temps qu'on pourrait le croire! Les applications mobiles étant souvent utile que pour une seule tâche à réaliser.
Rappelons également qu'une application hybride sera moins performante que le natif. Nos smartphones actuel ont beau souvent être plus puissante qu'un ordinateur, il reste beaucoup de téléphone portable milieu ou bas de gamme et l'optimisation doit rester importante.
Les développeurs derrières les frameworks présentés peuvent être les meilleurs du monde, traduire du code JavaScript ou C# vers du Swift ou du Java sera toujours moins optimisé que développer directement en Swift ou Java justement.
Concernant les jeux vidéos sur mobile (qui sont de grosse application donc), il existe des moteurs de jeux (framework spécialisé en jeux vidéos) conçu pour cela et faisant en sorte que cela soit directement multiplateforme et optimisé (nous pouvons citer Unity de Microsoft par exemple)

Scénario de recommandation

Android et iOS

Je suis dans une grosse boite avec beaucoup de développeurs à disposition, je peux donc me permettre de faire deux développements séparés afin d'avoir plus de performance.
C'est selon moi le scénario idéal et ayant pu côtoyer des développeurs et managers mobiles où j'ai travaillé, c'est le scénario que les entreprises tentent d'avoir le plus souvent même si ce n'est pas toujours facile et que si c'est lié à un client il vaut mieux lui expliquer que cela prendra plus de temps et surtout risque d'être plus cher.

Xamarin

J'ai une équipe de développement purement Microsoft dont je ne veux pas perturber les habitudes.
J'aurais du mal à recommander à quelqu'un de neutre niveau technologie d'utiliser Xamarin. La documentation et les guides qui y sont lié sont désastreux et pas à jour. Il y a des manques au niveau de la gestion d'une interface. Et au final Xamarin pousse parfois à faire du code indépendant pour Android et iOS ce qui va à l'encontre de ce que l'on recherche. Mais tout n'est pas à jeter dedans! Et les fans de Microsoft pourront y trouver leurs bonheur!

Ionic ou React Native

Soit j'ai une petite équipe ou moins de temps que prévu et je dois absolument fournir en même temps une application et sur iOS et sur Android.
Soit j'ai une équipe web performante qui veut un peu de changement alors ce choix sera le bon également.
On prendra bien évidemment Ionic si l'équipe est habitué à Angular et React Native si elle utilise usuellement React JS.

Et du coup? Mon choix: Ionic

Tout n'est bien évidemment pas à jeter dans le développement hybride.
Il y a pas mal de cas de figure où utiliser un des frameworks présentés sera une bénédiction comme le montre mes scénarios.
La bataille des langages en général est aussi une question de goût et d'habitude. J'ai commencé ma carrière professionnelle dans le monde JavaScript, j'ai apprécié ce langage via TypeScript et cela m'avantage de pouvoir l'utiliser pour un projet d'application mobile.
Ainsi mon choix se porterait donc sur l'utilisation d'Ionic et lors de mes projets personnelles, ça sera toujours vers lui que je me tournerais.
React Native (et React JS tout simplement) me plait beaucoup moins. Je dirais même que je ne comprends pas du tout l'engouement autour de cela. Mais il faut reconnaître qu'il est bien plus performant que ses concurrents.
En ce qui concerne Xamarin… Il a de trop grand manque pour que je puisse le recommander à quelqu'un, de plus j'ai l'impression qu'on doit se compliquer la tâche par rapport à ce que j'ai l'habitude.

Je pourrais faire la conclusion bateau de ce genre de comparaison: "Il n'y a pas un framework meilleur qu'un autre, il faut choisir en fonction des besoins, de l'équipe et des goûts". Elle est en partie vrai et raisonnable. Cependant je préfère insister sur la nécessité de bien réfléchir au besoin mobile de l'application, ne pas écarter trop vite de le faire en web et si le choix se porte tout de même sur un framework hybride, je vous recommande Ionic, le plus ancien, qui a le plus fait ses preuves et qui apporte la meilleure pérennité avec son mode de développement sorti d'Angular qui a selon moi la meilleure lisibilité, ce qui est primordial sur les projets à long terme. De plus, sa documentation est solide, accompagné de petit tutoriel et son futur me semble radieux en proposant régulièrement des mises à jours.

Source

Code source du projet

Backend: https://github.com/kingdomflo/M-A-R-X_Backend
Frontend: https://github.com/kingdomflo/M-A-R-X_Frontend

Langage - bibliothèques utilisées ou évoquées

PHP: http://php.net/
Framework PHP Lumen: https://lumen.laravel.com/docs/5.7
Doc de l'ORM Eloquent: https://laravel.com/docs/5.7/eloquent
Auth0: https://auth0.com/#/
Auth0 avec Ionic 3: https://auth0.com/docs/quickstart/native/ionic3
Ionic: https://ionicframework.com/ https://github.com/ionic-team/ionic
Token JWT: https://jwt.io/
Librairie PHP pour JWT: https://github.com/lcobucci/jwt/blob/3.2/README.md
Librairie NodeJS pour JWT: https://github.com/auth0/node-jsonwebtoken
Composer: https://getcomposer.org/
NodeJS: https://nodejs.org/en/
Bootstrap: https://getbootstrap.com/
Material design: https://material.io/design/introduction/#principles
RestSharp: http://restsharp.org/
Xamarin plugin notification: https://github.com/edsnider/LocalNotificationsPlugin
Expo pour React: https://expo.io/
React Native: https://facebook.github.io/react-native/
Fetch: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
React Navigation: https://reactnavigation.org/
i18n-js: https://github.com/fnando/i18n-js

Outils utilisés

Postman: https://www.getpostman.com/
Visual Code: https://code.visualstudio.com/
Diagramme UML: http://plantuml.com/
JetBrains: https://www.jetbrains.com/
ADM: https://developer.android.com/studio/run/managing-avds

Autre

Popularité de JavaScript en 2018: https://2018.stateofjs.com/
Part de maché des OS mobiles:
http://gs.statcounter.com/os-market-share/mobile/worldwide
Framework existant:
https://blog.jscrambler.com/10-frameworks-for-mobile-hybrid-apps/
https://www.websoptimization.com/blog/hybrid-mobile-app-frameworks/
JQuery: https://jquery.com/
Xamarin University: https://university.xamarin.com/
Stackoverflow: https://stackoverflow.com/