Master 1 Informatique

Rapport Programmation mobile

Université Lumière Lyon 2 – Année universitaire 2024/2025

Pierrick Dennemont
Loïc Babolat

Table des matières

Introduction

Notre projet, nommé GiFinder, permet à l'utilisateur de rechercher et d'enregistrer des GIF directement à travers notre application, grâce à l'utilisation de mots-clés. On entend par « enregistrer » la possibilité de sauvegarder les GIF sélectionnés dans la galerie photo de l'utilisateur, en cliquant simplement sur le GIF souhaité. Ainsi, la sauvegarde des recherches effectuées par l'utilisateur au sein de l'application n'est pas étudiée.

L'enjeu principal repose sur l'utilisation d'une API, reconnue comme étant l'une des plus importantes concernant le domaine des GIF. Celle-ci se nomme Giphy et est disponible à l'adresse suivante : GIPHY - Be Animated. L'utilisation d'une telle API permettra d'une part un choix riche et varié pour l'utilisateur mais aussi empêchera de stocker l'ensemble de la base de données des GIF en local, ce qui est essentiel pour optimiser notre projet.

Difficultés rencontrées

Dans cette section, nous proposerons de revenir plus en détail sur les aspects difficiles à intégrer à notre application.

Chargement d'un GIF

Au premier abord, on pourrait penser que l'utilisation d'un GIF avec le langage programmation Swift serait une tâche relativement facile, grâce aux nombreux widgets prédéfinis accessibles sur XCode. On pense notamment aux ImageView, permettant d'incruster rapidement et efficacement une photo sur une vue. Néanmoins, gérer l'utilisation des GIF a été plus difficile que prévue et cela peut s'expliquer grâce au type de fichier. En effet, un GIF est en quelque sorte à mi-chemin entre une photo et une vidéo. Un GIF est par définition dynamique mais comporte peu d'images, ce qui ne permet pas de les qualifier véritablement de vidéos. L'extension « .gif » confirme l'originalité de ce type de format.

Ainsi, la réflexion concernant l'intégration d'un tel type de fichier n'est pas anodine. Malheureusement, nos compétences dans ce langage de programmation ne nous permettaient pas de créer un algorithme permettant l'utilisation de GIF. Notre choix s'est alors porté sur l'utilisation d'un repository GitHub, permettant l'incrustation d'un GIF avec le widget ImageView. Plusieurs dépôts ont été considérés, on peut citer le projet « Gifu », celui-ci semblant populaire pour ses performances techniques. Néanmoins, nous n'avons pas réussi à l'utiliser correctement dans le temps imparti. Giphy, quant à eux, propose un environnement dédié (Giphy SDK), propice à l'utilisation de leur API, or celui-ci est bien trop complexe à manier. On a finalement utilisé un projet GitHub disponible au lien suivant : *https://github.com/kiritmodi2702/GIF-Swift/tree/master*. Ce dépôt, certes moins robuste et sophistiqué que ceux énoncés précédemment, a le mérité d'être simple d'utilisation et ne nécessite pas de dépendances, ce qui permettra par la suite des erreurs concernant des fichiers manquants. Pour utiliser les fonctions pour les GIF, il suffit d'ajouter dans un projet XCode le fichier « iOSDevCenters+GIF.swift » du projet Github mentionné. Cela étant dit, on peut désormais se focaliser sur d'autres aspects de l'application

Utilisation d'une TableView

Maintenant que nous sommes en mesure d'afficher correctement des GIF, il est primordial de se concentrer sur l'affichage. En effet, il serait judicieux pour l'utilisateur d'avoir une liste de GIF lorsque celui-ci effectue une recherche par mots-clés, afin de trouver le GIF qui lui correspond. Gérer les listes peut se faire facilement avec l'utilisation d'un tableau (« Array »), or on ne connaissait pas de widgets permettant d'exploiter cela. Pour résoudre ce problème, on a dû utiliser des sources extérieures, telles que Stack Overflow, Youtube ou encore la documentation Apple pour les développeurs, et apprendre à utiliser des widgets non cités pendant les TDs. Notre choix s'est porté vers un « TableView », ressemblant en quelque sorte à un tableau. Un autre widget « CollectionView » était utilisable pour notre application, or son utilisation semblait plus complexe. Celui-ci est un aspect à envisager afin d'améliorer notre application.

Un GIF serait donc affiché sur une cellule du « TableView ». Il semble alors possible de créer un code réutilisable pour chaque cellule du tableau, permettant d'une part l'affichage du GIF concerné mais aussi la gestion de l'enregistrement du GIF, en fonction de la case sélectionné. Cela passe par la mise au point d'une cellule « prototype ». Celle-ci contient un widget de type « ImageView » où sera affiché le GIF. Il reste cependant à coder tout le processus de gestion de GIF en fonction de la cellule, sans parler de la connexion à l'API, qu'on détaillera plus bas.

Comme dit plus tôt, il semble évident que le code pour chaque cellule sera le même. Pour cela, on doit faire passer la cellule prototype de notre « TableView » sous un contrôleur de type « UITableViewCell », celui-ci permettant la gestion des cellules. L'utilisation d'un tel contrôleur était quelque chose de difficile à appréhender, car on n'avait pas imaginé la possibilité qu'un contrôleur puisse être utilisé avec autre chose qu'une vue. Le principal problème n'a pas été l'implémentation mais le fait de trouver la solution répondant le plus à nos besoins, d'où l'utilisation de sources extérieures à celles des TDs.

Utilisation d'une API

Une bonne connaissance de l'utilisation d'une API est indissociable du projet, même si cela n'a pas été abordé en cours. Heureusement, l'API Giphy est bien documentée et couramment utilisée, ce qui permet de la manier relativement facilement. La principale difficulté était de comprendre comment envoyer une requête HTTP en Swift avec des paramètres (pour la gestion des mots clés) mais aussi de gérer la réponse de la requête. Encore une fois, beaucoup de ressources concernant cela sont disponibles sur Internet, il suffisait juste de les implémenter selon nos besoins

Afin de gérer le contenu de la réponse HTTP, on a décidé de créer une structure, identique à celle indiqué dans la documentation Giphy, en prenant le soin de conserver uniquement les attributs intéressants. Cette solution est certes opérationnelle, mais on est certain que celle-ci n'est pas la plus optimisée. En effet, l'intégration d'un GIF a demandé une structure complexe, ce qui peut être évitable selon nous avec une bonne compréhension du langage Swift. Il est sûrement possible de gérer plus efficacement les requêtes HTTP avec des fonctions préétablies ou encore d'autres types d'objets, pouvant par exemple exploiter le JSON.

Ajout de nouveaux GIFs au scroll

Lors d'une recherche faite par l'utilisateur pour trouver un GIF, nous ne chargeons que 15 GIFs, ce chiffre ayant été décidé arbitrairement. Or, dans de nombreux cas, il existe plusieurs centaines de GIFs disponible pour une seule recherche. Nous avons donc ajouté une fonctionnalité qui permet de recharger 15 nouveaux GIFs lorsque l'utilisateur atteint le bas du scroll. Pour cela, il a fallu utiliser un listener qui est appelé lorsque l'utilisateur scroll, puis en comparant la position du scroll à la taille total de l'élément, on peut savoir si l'utilisateur a atteint le bas de la page.

Dans le but de rendre un projet qui ne fait pas de répétition de code, nous avons voulu créer une classe qui permette de gérer les appels à l'API Giphy. Pour cela, nous nous sommes inspirés de projets GitHub qui utilise des APIs, principalement celui-ci *: https://github.com/fsalom/ios-api-asyncawait-swift*, notre version étant largement simplifié pour nos besoins et pour une bonne lisibilité.

Téléchargement des GIFs

Enfin, la dernière partie de notre application, après avoir chargé et afficher les GIFs : les télécharger. Nous nous étions mis dans l'idée que l'utilisateur devait pouvoir télécharger les GIFs vers sa galerie. Il a donc fallu passer par 2 étapes, tout d'abord le téléchargement du GIF en local, puis le transfert du GIF depuis les fichiers télécharger vers la galerie. Pour cela, nous avons utilisé des fonctions natives au développement pour iOS, avec une fonction qui permet d'écrire le fichier en local puis une fonction du framework Photos qui permet de transférer le fichier vers la galerie.

Problèmes non résolus concernant l'utilisation de l'API

Comme dit précédemment, notre application est fonctionnelle. Néanmoins, certains problèmes persistent dans notre code. On peut en citer deux majeurs. L'un concerne le changement des préférences des utilisateurs dans les réglages. En effet, l'utilisateur doit avoir autorisé GiFinder à accéder à la galerie photo. Cependant, à chaque changement de préférence concernant les autorisations de notre application, l'utilisation de notre application n'est pas possible. Celle-ci est comme remise à « zéro » et la console affiche la ligne suivante : « Terminated due to signal 9" printout ». On a essayé de trouver des solutions pour ce problème, ce qui n'a pas été concluent. Le problème n'a pas encore été résolu sur Stack Overflow (cf objective c - Correct way to handle change in settings for IOS - Stack Overflow), ce qui montre la pertinence de le mentionner.

Enfin, plusieurs messages d'avertissements s'affichent dans la console lors de l'utilisation de notre application. Cela concerne l'utilisation de l'API. Néanmoins, cela ne compromet pas le bon déroulement de notre programme.

Partage des tâches

Cette section est uniquement à titre indicatif, car on considère avoir travaillé de façon équitable sur le projet. En effet, celui-ci nous intéresse tous deux grandement, car Swift est un langage que nous n'avions pas eu l'occasion d'apprendre auparavant. De plus, l'idée de notre application nous stimulait à travailler dessus, celle-ci nous semblant originale. Nous aspirons tous deux à rejoindre le Master CIM l'année prochaine, ce qui témoigne notre envie de nous consacrer à l'application GiFinder.

L'ensemble du travail à été fait selon la méthode de « Pair Programming ». Cela consiste à travailler par binôme sur uniquement un ordinateur. Ainsi, un membre du binôme codait l'application tandis que l'autre fournissait une recherche active sur la façon d'implémenter le code, tout en surveillant le travail de l'autre. Les rôles étaient souvent inversés pour que chacun puisse contribuer de manière globale au projet.

Même si notre projet « GiFinder » a été codé entièrement avec ces méthodes, on peut tout de fois soulever des thématiques du projet où la participation d'un membre a été plus importante que l'autre. En effet, Loïc a su davantage agir lors de l'intégration de l'API Giphy. D'un autre côté, Pierrick a réussi à intégrer l'utilisation d'une « TableView » dans le projet. Cela ne revient pas dire que l'un a contribué plus que l'autre. Il s'agit uniquement d'un aspect du code où l'un a acquis une meilleure compréhension que l'autre, suite aux recherches effectuées. En effet, on faisait en sorte de chercher entre chaque session de travail de la documentation et une façon d'avancer sur le projet, ce qui nous permettait lors de chaque session de travail de comparer nos sources, et de créer le code à partir de celles-ci. En d'autres termes, on se partageait nos connaissances à chaque début de réunion pour pouvoir travailler plus efficacement. Les lacunes d'un membre étaient donc compensées par les connaissances plus poussées de l'autre membre et inversement. Ainsi, on est tous deux conscients d'avoir travaillé de façon équivalente sur ce travail.

Jeux d'essais

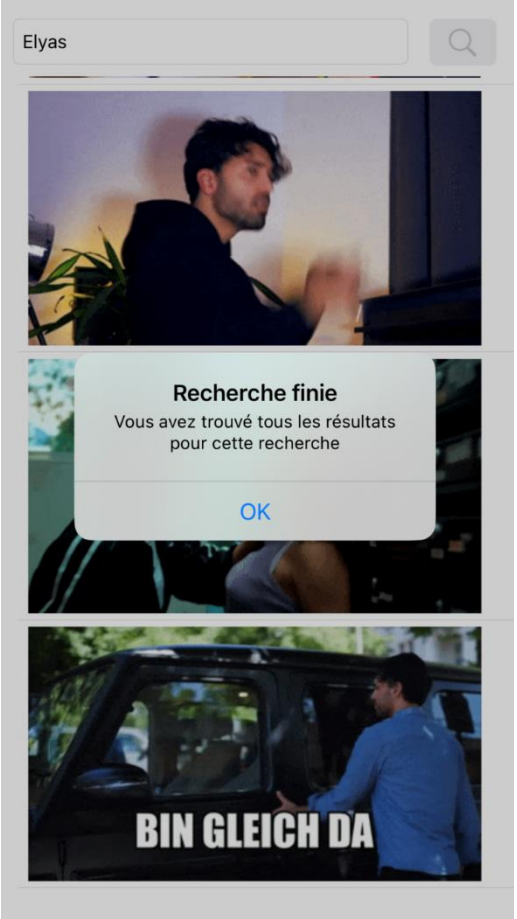

Fig. 1 : Affichage lorsque la fin de la recherche a été atteint

Fig. 2 : Affichage lorsque l'utilisateur essaie de télécharger sans les accès aux photos

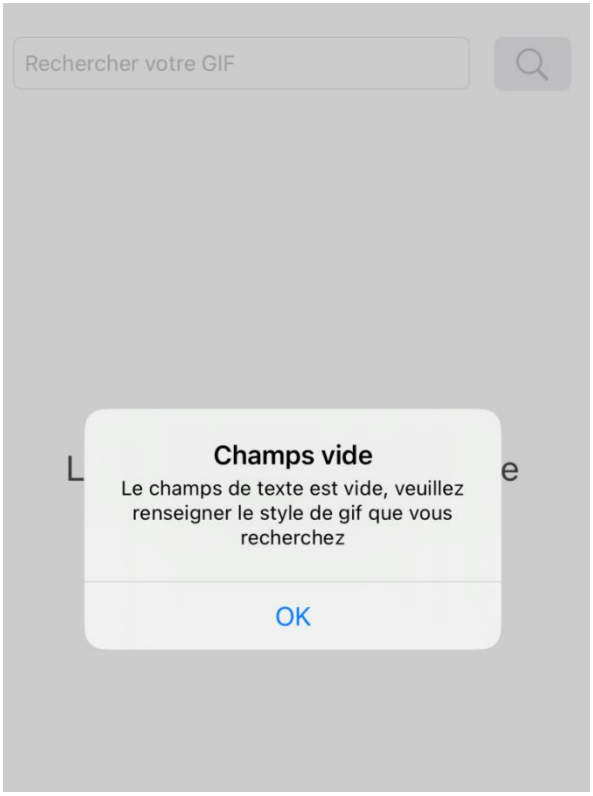

Fig. 3 : Recherche sans mots clés

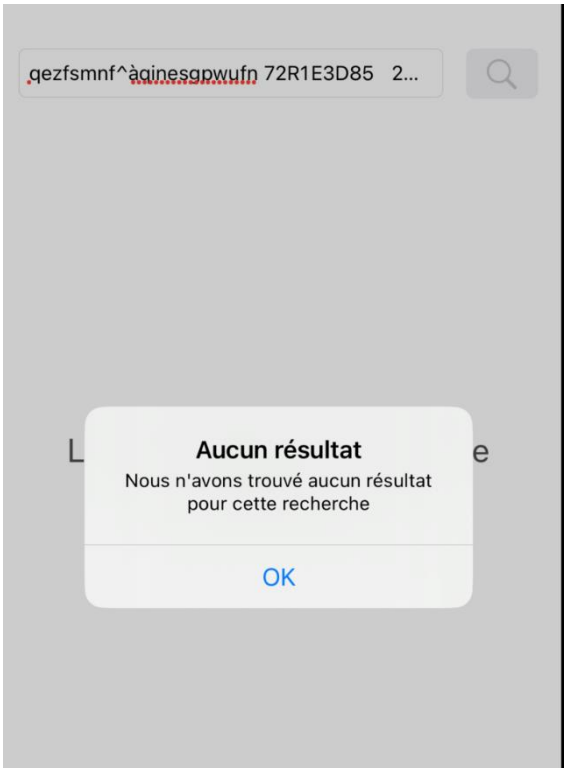

Fig. 4 : Recherche sans résultats

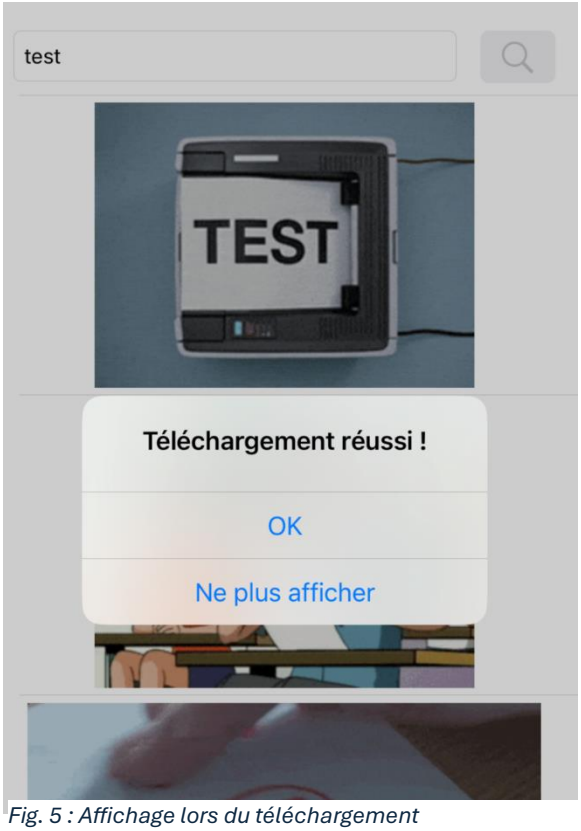

Fig. 5 : Affichage lors du téléchargement

Perspectives d'améliorations

Même si on reste satisfaits de notre rendu, il n'en reste pas moins que celui-ci ait été élaboré avec un temps restreint. Ainsi, plusieurs points n'ont pas pu être abordé, par manque de temps. Dans cette dernière section, on se propose d'éclaircir des perspectives d'amélioration, afin d'apporter un point de vue réflexif sur notre travail.

Tout d'abord et comme mentionné précédemment, il aurait été plus judicieux d'utiliser un widget de type « Collection View » pour l'interface de GiFinder. Cela permettrait d'obtenir une véritable galerie de GIF pour l'utilisateur. En effet, même si l'utilisation d'un « TableView » s'intègre bien à nôtre projet, il n'est pas très agréable pour l'utilisateur d'avoir accès aux GIF de cette façon. Un GIF prend, dans notre cas, l'entièreté d'une colonne, ce qui pourrait être évité avec une « Collection View », en permettant l'incrustation de GIF sur plusieurs lignes et colonnes.

De plus, on peut souligner le manque d'originalité comparé à ce qui existe déjà. En effet, même si notre projet reste singulier conformément aux autres projets suggérés, il n'en reste pas moins une fonctionnalité omniprésente dans les applications actuelles. Celle-ci est systématiquement disponible sur le clavier Google par exemple, ce qui rend l'utilisation d'une application dédiée aux GIF obsolète. Encore une fois, on ne cherchait pas dans ce projet à « réinventer la roue » mais simplement à fournir une idée stimulante à réaliser et dont on peut être contents du résultat, ce qui est le cas avec GiFinder. Pour compléter notre application, on pourrait par exemple intégrer de nouvelles fonctionnalités, afin de se démarquer de ce qui est proposé actuellement. Par exemple, on pourrait suggérer un jeu, similaire à un Quizz, pour que l'utilisateur trouve le mot-clé correspondant au GIF affiché, parmi un ensemble de proposition, afin de laisser la possibilité à l'utilisateur de découvrir de façon naturelle des nouveaux GIF qu'il n'a pas l'habitude de rencontrer. Cet objectif est atteignable à notre niveau, mais n'était pas envisageable sur la période de développement.

Il est à noter que, pour une application qui a pour but d'être déployé, il faudrait réaliser de nombreux autres tests. Notamment sur le problème de la connexion Internet. Effectivement, les ordinateurs MAC étant relié connecté via Ethernet, la connexion est de qualité. Cependant, il arrive souvent que la connexion par wifi ou par réseau mobile soit de mauvaise qualité, il faudrait donc tester la qualité et la rapidité de l'appel API avec une mauvaise connexion pour constater l'expérience utilisateur dans ce cas-là.

Il y a également un autre problème dont nous nous sommes rendus compte le dernier jour du rendu et que nous n'avons pas eu le temps de corriger. Ce problème est que l'API Giphy ne fonctionne plus lorsque l'application est lancée sans XCode, c'est-à-dire lorsqu'on lance l'application directement depuis le téléphone, après l'avoir installé avec XCode. C'est un problème auquel nous n'avions pas pensé, et dont nous avons du mal à déterminer la véritable origine. Cela pourrait venir du fait que les autorisations ne sont plus actives lorsque l'application est lancée sans XCode, et donc que l'application n'est pas autorisée à accéder à Internet. Pour régler ce problème, il faudrait passer du temps sur le debug de l'application, car nous n'avons pas accès à la console lorsqu'on lance l'application sur un téléphone physique. Une des façons de faire serait de déclencher un pop-up d'alerte, ainsi, si l'utilisateur n'est pas connecté à Internet, nous pourrions voir si le problème vient bel et bien de l'accès à Internet.

Enfin, l'expérience utilisateur est perfectible. Cet aspect de notre application a été le dernier considéré lors du développement de GiFinder. On a préféré se focaliser sur quelque chose d'opérationnel, même si cela consistait à laisser de côté certains aspects graphiques.