Projet mobile

AdMoApp

Résumé

Ce document comprend tous les aspects concernés par l'implémentation et la réalisation d'une application Flutter. Cette application est réalisée dans le cadre d'un projet pour le cours de *AdMoApp*, l'un des cours Master du *MSE HES-SO*, et doit inclure quelques contraintes définies à l'avance par les professeurs.

L'application doit être *context-aware*, c'est-à-dire qu'elle doit adapter son comportement selon des données contextuelles de son environnement. L'utilisation d'un capteur embarqué dans l'appareil est également obligatoire. L'application doit être une application mobile développée avec le Framework *Flutter*, qui utilise le langage *Dart*.

But de l'application

L'application devra permettre de scanner des QR-codes par le biais de l'appareil photo de l'appareil mobile. Avant de proposer à l'utilisateur ou l'utilisatrice d'ouvrir le lien correspondant, l'application ira vérifier sur un site d'analyse de virus nommé cloudmersive.com afin d'attester la sécurité du lien. Ce service propose une API, qui sera utilisée dans ce but. Une fois l'analyse obtenue, un résumé des informations liées au contenu scanné sera affiché, à côté d'un bouton permettant d'ouvrir ledit contenu. Un historique des scans passés sera aussi accessible.

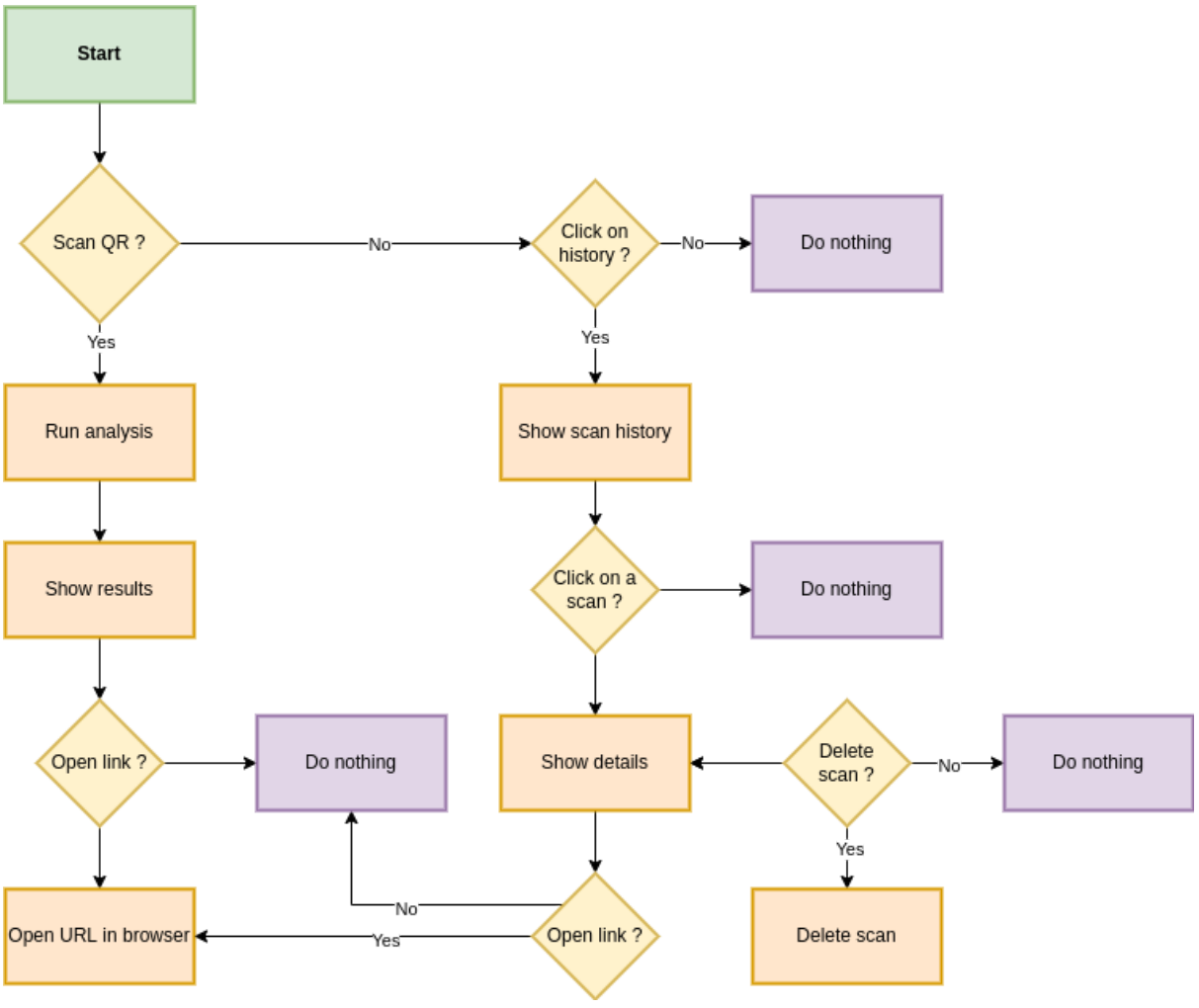
Ingénierie logicielle

Ce chapitre comprend plusieurs aspects liés à la réalisation logicielle de l'application.

Conception

La figure ci-dessous montre les différents choix et actions possibles lors de l'utilisation de l'application.

Nous avons conçu cette application afin de limiter au maximum les interactions nécessaires à l'utilisateur ou utilisatrice, tout en restant cohérents et logiques au niveau des interactions.



Technologies utilisées

Voici une liste de toutes les technologies utilisées pour notre application.

Flutter

Flutter est un kit de développement logiciel (SDK) d'interface utilisateur open-source créé par Google. Il est utilisé pour développer des applications pour Android, iOS, Linux, Mac, Windows, Google Fuchsia et le web à partir d'une seule base de code.

https://flutter.dev/

mobile_scanner

Package Flutter qui apporte un scanner universel de codes-barres et de codes QR pour Flutter basé sur MLKit. Ce package est actuellement disponible sur Android, iOS, MacOS et le Web.

https://pub.dev/packages/mobile_scanner

provider

Package Flutter qui permet la gestion de données au sein de l'application. Un Provider propose une enveloppe autour des widgets *InheritedWidget* pour les rendre plus faciles à utiliser et plus réutilisables. Il permet une allocation/disposition simplifiée des ressources, des chargements paresseux, une simplification de mise en place ainsi qu'une évolutivité accrue pour les classes dont le mécanisme d'écoute croît de manière exponentielle en complexité.

https://pub.dev/packages/provider

http

Package Flutter qui permet d'effectuer des requêtes HTTP, selon un principe de *Futures* (permet d'exécuter des travaux de manière asynchrone afin de libérer les autres threads qui ne doivent pas être bloqués).

Ce paquet contient un ensemble de fonctions et de classes de haut niveau qui permettent de consommer facilement des ressources HTTP. Il est multiplateforme et prend en charge le mobile, le bureau et le navigateur.

https://pub.dev/packages/http

drift

Package Flutter qui est une bibliothèque de persistance réactive pour *Flutter* et *Dart*, construite au-dessus de *sqlite*. Elle est flexible, modulaire, *typesafe*, rapide, réactive et multi-plateforme.

https://pub.dev/packages/drift

sqlite3_flutter_libs

Package Flutter nécessaire pour l'utilisation de *Drift*. Apporte diverses bibliothèques utiles lors de l'utilisation de *sqlite.*

https://pub.dev/packages/sqlite3_flutter_libs

path_provider

Package Flutter pour trouver les emplacements couramment utilisés sur le système de fichiers.

https://pub.dev/packages/path_provider

path

Package Flutter complet et multiplateforme de manipulation de chemins pour *Dart*. Il fournit des opérations communes pour manipuler les chemins : joindre, diviser, normaliser, etc.

https://pub.dev/packages/path

flutter_localizations

Librairie Flutter qui permet de définir des textes différents selon la langue utilisée par l'utilisateur ou l'utilisatrice.

Apporté avec le *framework Flutter*.

intl

Pacakge Flutter qui fournit des fonctions d'internationalisation et de localisation, notamment la traduction des messages, les pluriels et les genres, le formatage et l'analyse syntaxique des dates et des nombres, et le texte bidirectionnel.

https://pub.dev/packages/intl

Implémentation

Notre implémentation a été inspirée des exemples, concepts et manières de faire présentés en cours. Nous avons fait attention de respecter l'architecture conseillée par les professeurs et avons séparé un maximum nos parties de codes en *widgets* lorsque possible.

Database

Notre stockage persistant a été réalisé avec *Drift*. Nous n'avons besoin que d'une seule table, qui stocke le résultat de l'API *Cloudmersive*. Voici la classe qui nous permet de récupérer les données :

```
@DriftDatabase(tables: [ScansTable])
class MyDatabase extends _$MyDatabase {
 MyDatabase() : super(_openConnection());
```

```
@override
int get schemaVersion => 3;

Stream<List<ScansTableData>> get allScans => select(scansTable).watch();

Future<List<ScansTableData>> getScan(int id) {
  return (select(scansTable)..where((tbl) => tbl.id.equals(id))).get();
}

Future<int> deleteScan(int id) {
  return (delete(scansTable)..where((tbl) => tbl.id.equals(id))).go();
}

Future<int> saveScan(ScansTableCompanion td) {
  return into(scansTable).insertOnConflictUpdate(td);
}
}
```

La méthode allScans retourne un Stream grâce à la méthode *.watch()* placée en fin d'appel. Cela nous permet d'actualiser les listes basées sur ces données automatiquement.

La méthode saveScan utilise une classe générée par *Drift* et basée sur la définition de la table correspondante. Cette classe "*Companion*" permet de ne pas définir tous les champs d'un enregistrement en base de données, ce qui évite de définir une valeur pour la colonne *id*, qui est auto-incrémentée.

LocalAPI

Cette classe comprend l'instance unique de la base de données implémentée avec *Drift* et rend disponible ses méthodes dans l'application.

```
class LocalApi {
final MyDatabase _db;

LocalApi({required MyDatabase myDatabase}) : _db = myDatabase;

Future<void> deleteResponse(int id) async {
  await _db.deleteScan(id);
}

Future<void> saveResponse(ScanReqResponse newScan) async {
  await _db.saveScan(ScansTableCompanion(
      url: Value(newScan.url),
      websiteThreatType: Value(newScan.websiteThreatType),
      httpCode: Value(newScan.httpCode),
      cleanResult : Value(newScan.cleanResult),
      virusFoundCount: Value(newScan.virusFound.length)));
}

Stream<List<ScansTableData>> getAllResponses() => _db.allScans;
```

```
 Future<List<ScansTableData>> getResponseById(int id) => _db.getScan(id);

}
```

Dans le cas de la sauvegarde d'un nouveau scan avec la méthode *saveResponse*, une corrélation est faite entre la classe retournée suite à l'appel à l'API *Cloudmersive* et la classe présentée précédemment, utilisée par *Drift*.

L'implémentation de cette classe permet d'éviter de réimplémenter un *repository* si l'on change de système de stockage.

Repository

Notre repository permet à tout l'application de pouvoir accéder aux données persistantes de l'application. Dans notre cas, nous utilisons l'API dite locale, qui récupère les informations depuis *Drift*.

```
class ResponsesRepository {
 const ResponsesRepository(LocalApi localApi) : _localApi = localApi;

 final LocalApi _localApi;

 // Call local API
 Stream<List<ScansTableData>> getAllResponses() =>
_localApi.getAllResponses();

 Future<void> saveResponse(ScanReqResponse scan) =>
_localApi.saveResponse(scan);

 Future<void> deleteResponse(int id) => _localApi.deleteResponse(id);

 Future<List<ScansTableData>> getResponseById(int id) =>
_localApi.getResponseById(id);
}
```

Dans notre cas, nous n'avons qu'une source de données à disposition, ce qui rend cette mise en place peu utile. Mais dans une réalisation plus conséquente, une telle mise en place est conseillée.

Main

La classe *Main* est le point d'entrée de l'application. Afin d'éviter de la modifier si nous souhaitons changer un aspect de l'application, nous créons une instance de la classe App.

```
oid main() {
 final localApi = LocalApi(myDatabase: MyDatabase());

 runApp(App(
     repository: ResponsesRepository(localApi),
```

```
    ));
}
```

App

La classe *App* contient les éléments racine de l'application. Elle possède l'instance du *repository* afin de pouvoir accéder aux données, définit les valeurs de texte selon la langue utilisée et met en place la route par défaut. Cette route contient les différents écrans accessibles et affiche l'écran utilisé dans un moment T.

```
class _AppState extends State<App> {
 @override
 Widget build(BuildContext context) {
   return Provider.value(
     value: widget.repository,
       child: MaterialApp(
           localizationsDelegates: const [
             GlobalMaterialLocalizations.delegate,
             GlobalWidgetsLocalizations.delegate,
             GlobalCupertinoLocalizations.delegate,
           ],
           supportedLocales: const [
             Locale('fr', ''), // French, no country code
             Locale('en', ''), // English, no country code
           ],
           routes: {
             HomePage.route: (context) => const HomePage()
           },
           initialRoute: HomePage.route,
         )
     );
 }
}
```

HomePage

Comme expliqué plus haut, une classe contient tous nos écrans de l'application et le bon écran est affiché selon le choix de l'utilisateur. Cela permet aussi de définir la barre de navigation inférieure de l'application ainsi que la top bar.

```
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: const Text("Scans"),
   ),
   body: activeTab == AppTab.scanner
       ? const ScannerScreen()
       : const HistoryScreen(),

   bottomNavigationBar: BottomNavigationBar(
     currentIndex: AppTab.values.indexOf(activeTab),
```

```
      items: const [
        BottomNavigationBarItem(icon: Icon(Icons.qr_code_scanner_rounded),
label: "Scanner"),
        BottomNavigationBarItem(icon: Icon(Icons.view_list_rounded), label:
"History"),
      ],
      onTap: (index) {
        _updateTab(AppTab.values[index]);
      },
    ),
  );
}
```

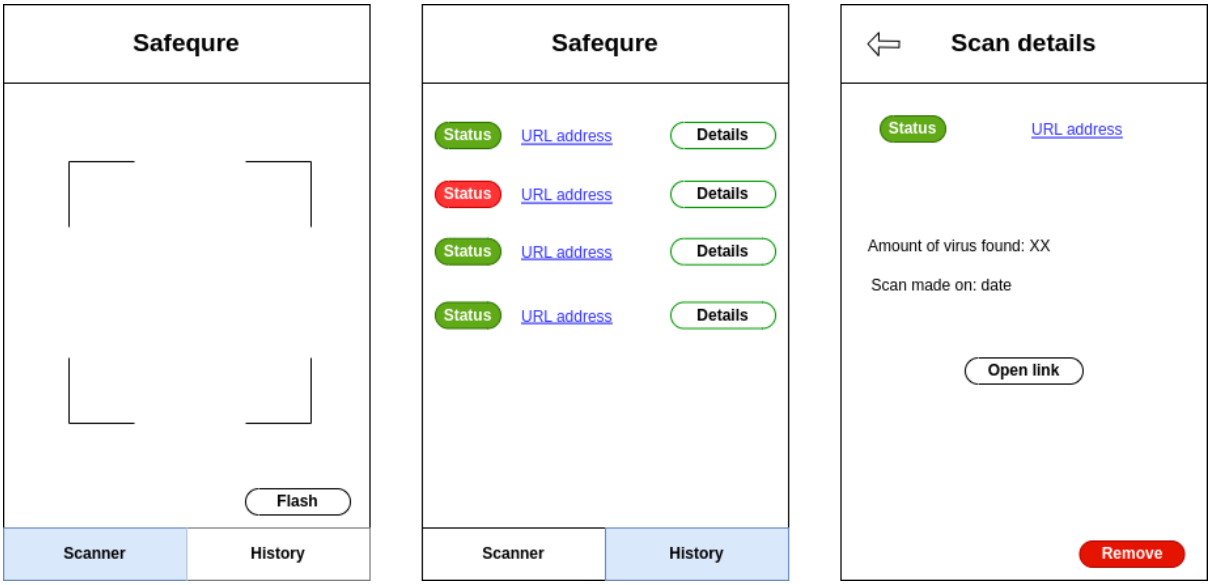
Interfaces graphiques

Ce chapitre va traiter l'aspect graphique de l'application.

Maquettes

Avant de concevoir notre application au niveau de son code, nous avons défini son squelette afin que nous nous rendions compte du travail à effectuer.

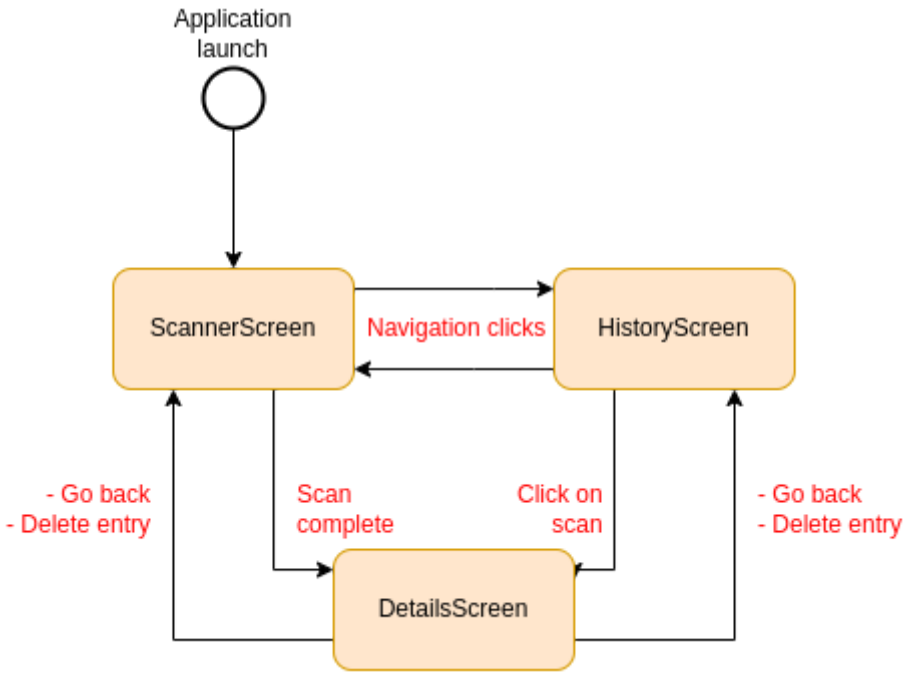
Les maquettes que nous avons réalisées sont sous forme de *wireframes*, qui n'affichent uniquement les éléments fonctionnels sans s'occuper de leur aspect visuel (couleur, style de texte, …).



Nous n'avons pas besoin d'afficher plus d'informations, l'application étant plutôt basique dans ses écrans.

Navigation

Comme nous disposons de plusieurs écrans, nous avons besoin de changer l'affichage entre ces derniers. Pour cela, nous avons mis en place une navigation.



Les textes rouges correspondent à des actions faites par l'utilisateur ou l'utilisatrice, qui résultent en des changements dans la navigation.

Les écrans sont représentés par les cases oranges.

Persona

Les personas sont utilisés dans le marketing et la publicité en créant un persona qui représente un groupe ou un segment de clients afin que l'entreprise puisse concentrer ses efforts. Par exemple, les agences de publicité en ligne peuvent surveiller les photos, l'historique de navigation et les publicités que les internautes sélectionnent ou choisissent de cliquer. Sur la base de ces données, elles adaptent leurs produits à un public ciblé ou décrivent mieux un segment de clientèle en utilisant une approche fondée sur les données.

Dans notre cas, nous avons défini deux personas. Ce sont les deux profils que nous avons identifiés à la suite de nos réflexions.

Antoine Dufour

Antoine a 27 ans et travaille en tant que développeur dans une entreprise de service. Son poste n'est pas à responsabilité et gagne 85'000 CHF par mois.

À côté de son travail, Antoine s'intéresse de près aux nouveautés sur le marché de l'informatique. Sa formation d'informaticien puis son brevet fédéral lui ont appris à toujours s'informer sur ses intérêts, au point que ses amis l'appellent souvent "le geek". En effet, l'un de ses hobbies est de créer des projets personnels, que ce soit des logiciels, des systèmes embarqués ou des interfaces.

Il habite en ville et se déplace en vélo, car l'exercice et le respect du climat sont importants pour lui.

En parallèle à ce premier hobby, Antoine apprécie faire des randonnées, faire du vélo et déguster des bières artisanales. Les activités en plein air lui permettent de déconnecter et de se rapprocher de la nature.

Ses objectifs dans la vie sont d'être indépendant, d'être prudent et de faire tout son possible pour respecter ses principes lorsqu'il prend des décisions.

Il déteste les poivrons, prendre l'avion, les logiciels mal réalisés et faire des heures supplémentaires. Il hait perdre son temps, ce qui le force à optimiser ses actions et de réfléchir aux meilleures décisions possibles.

Chéline Hameaudruz

Chéline a 31 ans et est très active en politique, particulièrement dans sa ville d'origine qui est Genève. Elle y travaille en tant que comptable : on peut la qualifier de carriériste, car Céline est passée cadre en 2019 et gagne plus de 100'000 CHF par mois. Son travail sérieux et rapide lui apporte de nombreuses primes, ce qui la motive au jour le jour.

Son travail est central pour elle : elle ne laisse que peu de temps pour ses activités annexes. Elle pratique la course à pied afin qu'elle dépense son énergie, dont ses réserves sont inépuisables. La course à pied lui permet de s'adapter à ses horaires de travail, car elle peut choisir le lieu et l'heure pour ses entraînements.

Avec sa vie qui défile à 100 à l'heure, Chéline s'en accommode facilement. Elle a essayé, il y a quelques années, d'augmenter son cercle d'amis et de se trouver quelqu'un, mais cette vie-là ne la comble pas. Son ex-copine lui reprochait de ne pas lâcher son travail pour se relaxer : depuis, Chéline n'a plus cherché l'amour.

En 2021, alors en télétravail, l'entreprise de Chéline a subi une cyberattaque et a perdu plusieurs dizaines de milliers de francs. Cela a eu comme effet de sensibiliser les employés à la cyber sécurité, enjeu que Chéline a vite compris et saisi. Ne sachant pas si elle en était le maillon faible, Chéline fait dès lors très attention à ses actions en ligne.

Chéline n'apprécie pas les chiens, n'aime pas partir en vacances et n'aime pas voir à long terme : elle préfère réagir aux situations selon son feeling, ce qui ne l'a encore jamais péjorée.

Finalité

Les deux profils représentent bien les prospects qui nous intéressent, et qui sont susceptibles d'être intéressés par notre projet : la personne technophile qui connaît son milieu, et qui s'arme des outils nécessaires pour se sécuriser, ainsi que la personne ne provenant pas de l'informatique mais qui a le besoin de se sécuriser au quotidien. Dans ces deux cas, l'application doit être utilisable et ergonomique.

Tests

Tests utilisateurs

Score

Discussions

Conclusion

analyse personnelle