P_appro-1

Robustiano Lombardo – CID4A

Lausanne

22 janvier – 15 mars

Karim Bourahla

Table des matières

1. Présentation du projet

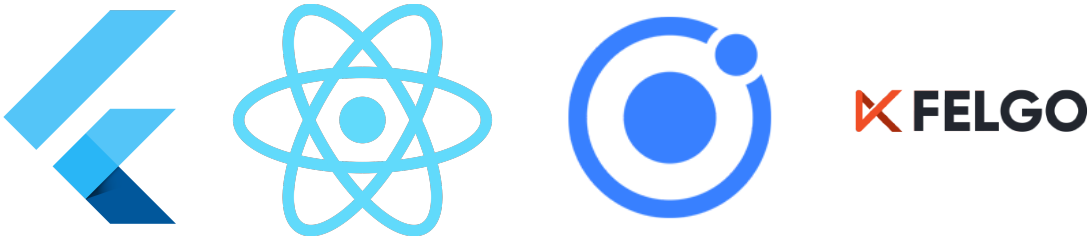
Le projet consiste en la conception et réalisation d'une application web et mobile qui est disponible sur iOS et Android. Cette application est un gestionnaire de contacts qui dispose de fonctionnalité CRUD standard pour gérer les contacts et un outil de recherche.

L'application doit inclure des tests avec les technologies suivantes : Flutter qui va servir de base pour l'interface utilisateur et Larvel en mode API pour le traitement des données.

Il y aurait d'autres technologies pour créer cette application telle que React native qui est très efficace pour le développement et la maintenance car le code qui génère l'application et le même pour iOS et Android, pour le côté web il faut par contre l'adapté. De plus vous pouvez ajouter du code Java ou Swift si besoin.

Ionic est aussi un très bon outil pour cette application, celui-ci s'apparente plus à du développement web. C'est un framework qui permet de travailler avec la technologie JavaScript que vous désirez, il s'intègre de manière transparente a des frameworks telle que Angular, React et Vue, vous pouvez aussi ne pas utiliser de frameworks cela fonctionne aussi.

Il existe aussi Felgo qui bien sûr vous permet de faire des application Cross-Platform il est basé sur le framework Qt. Il n'est pas dans le même style que les deux précédents, vue qu'il utilise Qt sont code et principalement du c++. Environ 90% du code génère la même application le reste sert à l'adaptation pour chaque plateforme.



2. Principes UI/UX exploités

2.1. Décrire les principes

Les différents principes UX exploités sont la compréhension de l'utilisateur et concevoir pour celui-ci. Le premier est toujours la première étape, il servira de base pour le reste cela implique de comprendre leurs besoins, attentes et comportement. Le deuxième principe vient dès que vous êtes sûr que le premier correct. Vous devez concevoir votre application en fonction de leurs besoins et de leurs attentes. Cela signifie créer une application qui facile à utiliser, qui répond aux besoins des utilisateurs et qui offre une expérience agréable.

Les différents principes UI exploités sont la clarté et la simplicité, et la cohérence. L'un des principes fondamentaux de l'UI est la clarté et la simplicité. L'interface utilisateur doit être claire et facile à comprendre. Cela signifie utiliser des éléments de conception clairs, des couleurs contrastées pour la lisibilité, et une hiérarchie visuelle claire pour guider les utilisateurs à travers l'application. La cohérence est un autre principe clé de l'UI. Votre interface utilisateur doit être cohérente à travers tous votre site. Cela signifie utiliser les mêmes éléments de conception, les mêmes couleurs et les mêmes polices à travers les pages.

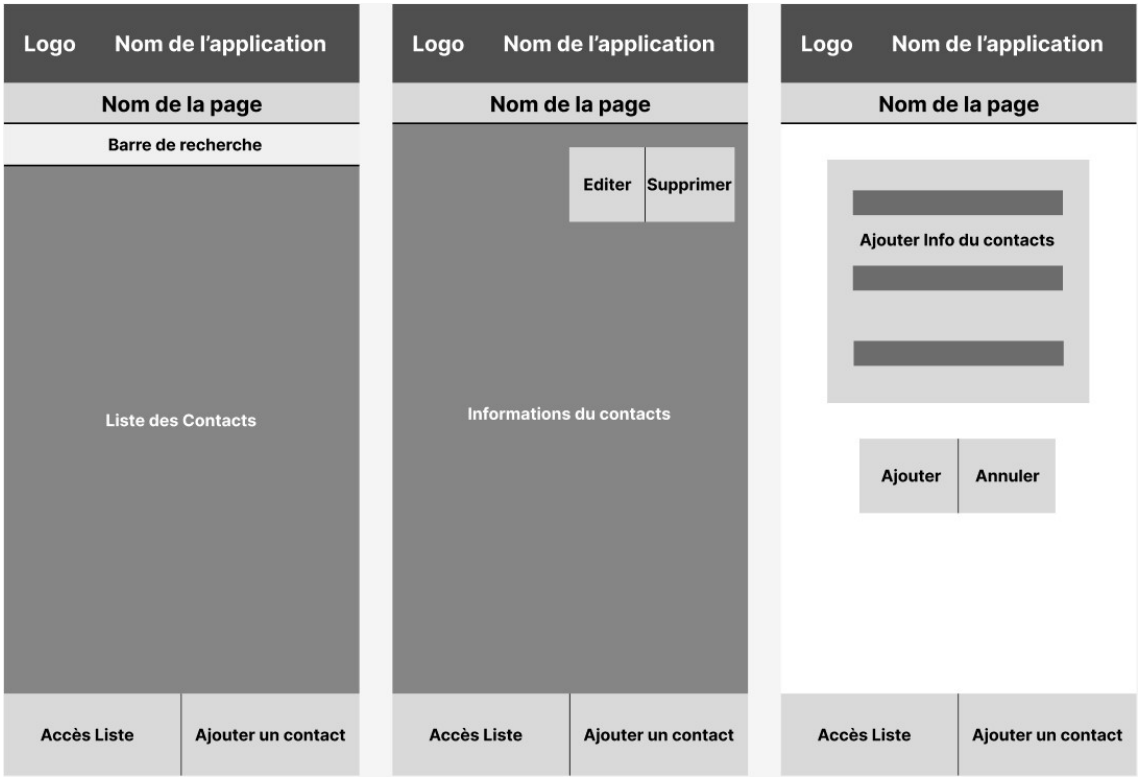
2.2. Material Design

Material Design est un système de conception construit et pris en charge par les concepteurs et développeurs de Google. C'est ici que Flutter a accès à toutes ces icones, polices d'écriture ou encore ces couleurs.

2.3. Maquette

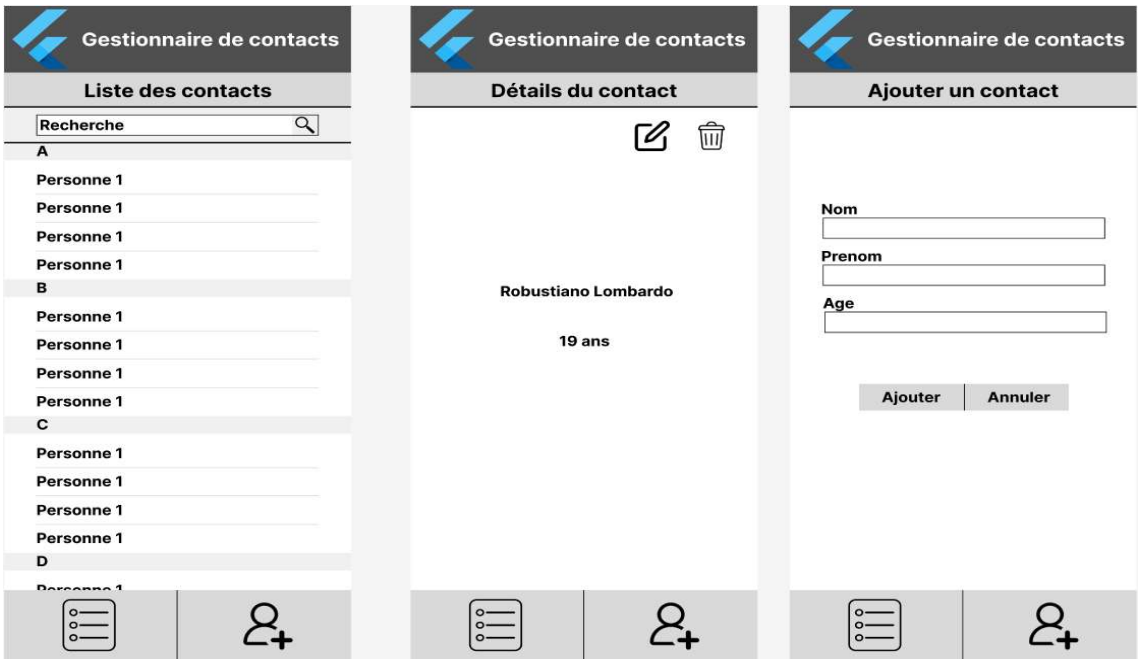
2.3.1. Zoning

Le zoning est la première partie pour la création d'une maquette, il s'agit de découper visuellement la page en plusieurs zones en fonction des besoins. Voici le zoning qui a été valider par le client :



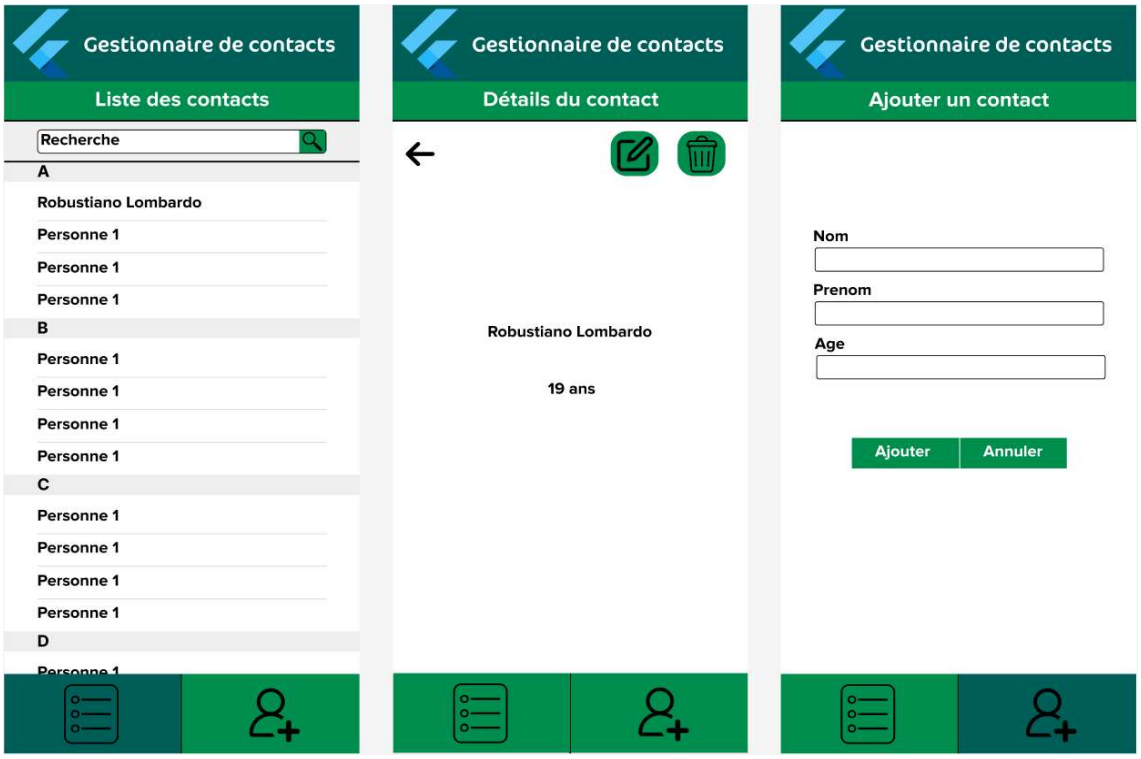
2.3.2. Wireframe

Le Wireframe est la prochaine étape, la maquette va commencer à ressembler au produit final. Ici le plan est de réaliser la maquette finale mais sans couleurs juste avec des nuances de gris et une police d'écriture standard. Voici le Wireframe valider par le client :



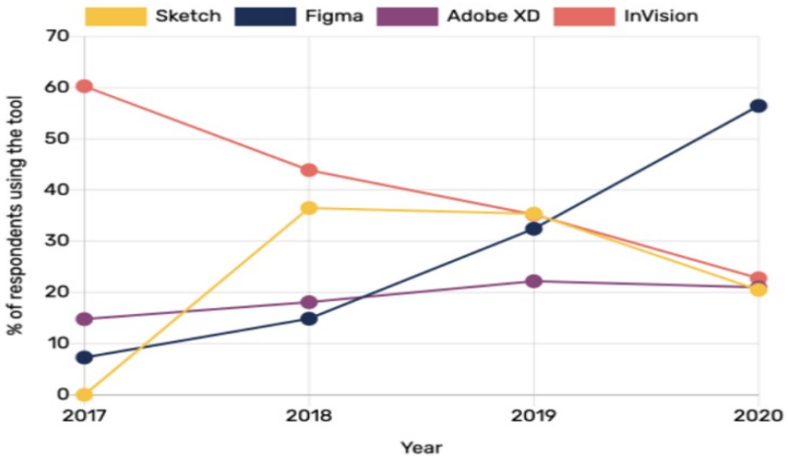
2.3.3. Mockup

Après avoir fait valider le Wireframe par le client, la partie dernière partie de la maquette peut commencer. Il faut reprendre le Wireframe et lui ajouter les couleurs voulu et la bonne police, de plus il faut ajouter les fonctionnalités des éléments, par exemple un lien ramène sur une autre page, les boutons qui ajoute un élément. Les couleurs et polices sont celles de Retraites Populaires et voici la maquette finale valider par le client :



2.3.4. Figma

L'outil utilisé pour créer cette maquette et Figma, celui-ci est très simple à prendre en main et très complet pour le design, en plus il permet aussi de pouvoir ajouter des actions lorsqu'on clique sur un bouton par exemple. Vous pouvez récupérer le css de votre élément, ce qui vous permet de gagner du temps. De plus vue que la plateforme a beaucoup d'utilisateur, il y a des templates créer par cette communauté et une grande majorité de ces templates sont gratuits.



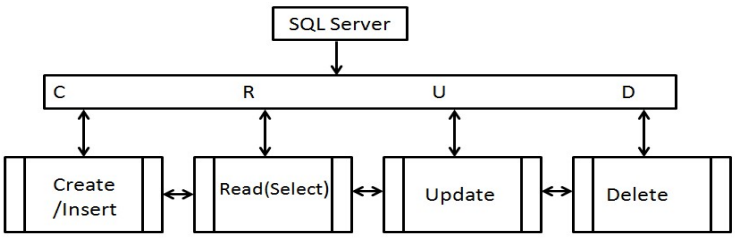
2.3.5. Flutter Flow

Flutter dispose d'un outil de conception ce nommant Flutter Flow, il aide les développeurs à créer des applications encore plus rapidement grâce au développement d'application assisté par l'IA. L'outil intègre l'IA dans son éditeur de code personnalisé pour une génération de code et des suggestions. Il est possible de récupérer le code générer par l'application pour le reprendre dans votre code.

3. Description de l'architecture API

L'API qui est utilisé est un **API REST**. Tout d'abord qu'est-ce que REST, c'est un ensemble de contraintes architecturales et il peut être implémenter de différentes manières. Lorsqu'une requête client est effectuée via cette API, elle transfère une représentation de l'état de la ressource au demandeur. Ces informations sont fournies via HTTP sous format JSON.

L'API doit pouvoir gérer des requêtes CRUD. Le CRUD comporte 4 type de requête POST, GET, PUT et DELETE, ce qui fait une pour chaque fonctionnalité du CRUD.

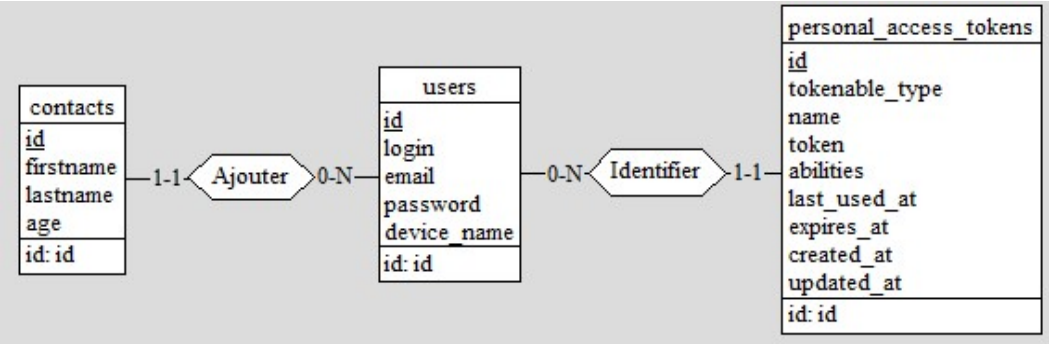


Comme présentez sur le schéma au-dessus, l'application Flutter va envoyer sa requête au serveur de l'application Laravel est celle-ci va traiter sa requête et l'envoyer au serveur SQL. Après ça, l'information fait le chemin inverse pour transmettre l'état de la requête, voir s'il y a eu un problème, si la requêtes c'est bien exécuté ou encore s'il y a des données à retourner comme une liste de contact par exemple.

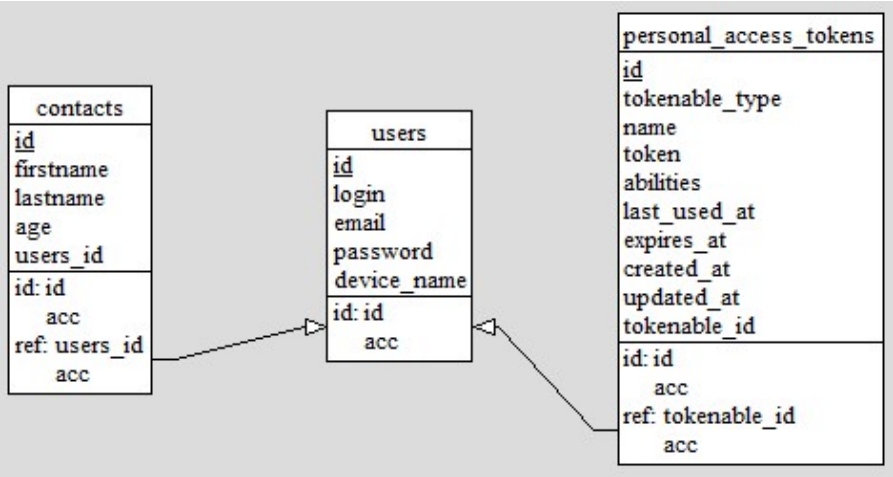
4. Conception de la base de données

La base de données a été conçu de cette manière. Il faut 3 tables pour que l'application fonctionne, la première qui est contacts, ici tous les contacts vont être enregistré avec l'id de l'utilisateurs pouvoir savoir à qui apprend chaque contact. La table users contient les informations de connexion des utilisateurs. La dernière qui est personal_access_tokens va contenir les tokens de connexion des utilisateurs leur permettant d'accéder à leurs données.

Voici le MCD :



Voici le MLD :



5. Mise en place de l'environnement

5.1. Environnement de développement Flutter

L'installation de Flutter et une installation standard que vous pouvez retrouver sur le site officiel de Flutter. Vous aurez juste besoin de rajouter deux commandes à la fin de l'installation, celle-ci servent pour la partie API avec Laravel : `flutter pub outdated` et `flutter pub add dio`

Dans le fichier pubspec.yaml ajouter les lignes `shared_preferences: ^2.0.9` et `device info: ^2.0.3` sous **dependencies** ensuite vous pouvez exécuter le commande : `flutter pub get`

Pour le développement, VS code est un éditeur de code très connu et qui a tout ce qu'il faut pour développer du Flutter. Des extensions sont nécessaire pour le bon fonctionnement du code et aussi dans le but d'aidé le développeur dans sa tâche. Les extensions sont les suivantes :

Dart est indispensable car c'est le langage de programmation d'une application flutter.

Flutter qui sert à créer les différents widgets pour créer les interfaces utilisateur.

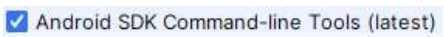
Flutter Widget Snippets sert à générer certain éléments grâce à des raccourcis par exemple : **Erreur ! Signet non défini.** qui va permettre de générer toutes l'architecture pour créer un nouveau widget.

Si le code dart n'est pas formaté correctement et est donc illisible, essayer d'aller dans les paramètres de VS code et chercher ce paramètre :

S'il est coché, décocher et coché le. Maintenant votre problème est réglé.

5.2. Emulateur

5.2.1. Installation

Pour l'émulateur, Android Studio permettra de faire tous ces dons vous avez besoin sur un smartphone Android bien évidemment. L'installation standard suffit et il faut juste rajouter un paramètre à rajouter dans les paramètres le Android SDK Command-line Tools (latest) sous la rubrique Languages & Frameworks > Android SDK > SDK Tools et installé 

5.2.2. Virtual Device

Après avoir installé l'émulateur, il faut créer un smartphone virtuel. Il faut mettre en place ce smartphone, le **Medium Phone** est comme son nom l'indique de taille moyenne pour avoir une bonne base sur la quel faire l'affichage. Pour la version de l'OS, **UpsideDownCake** est une des dernières versions d'Android à ce jour. Voilà tout pour les caractéristiques du smartphone.

5.3. Environnement de dev Laravel

Pour ce projet, **Laravel** va fonctionner sur la version **11**, Il vous faudrait donc au minimum la version **8.2** de **PHP**. Après avoir installé **PHP** vous pouvez prendre **Xampp** qui sers à mettre en place un serveur web local, une installation standard suffi.

Il vous faut aussi installer **Composer**, c'est un gestionnaire de dépendances pour PHP. Ici aussi une installation standard est suffisante. Lorsque vous disposer de tous les programmes cités jusqu'à présent vous pouvez installer **Laravel**. Vous pouvez reprendre le projet qui est sur **Bitbucket**. Si le projet que vous avez pull contient déjà la version 11 de Laravel, vous pouvez sauter au point suivant.

Sinon il vous faut appliquez les modifications suivantes dans le projet Laravel. Pour passer à la version 11, diriger vous dans le fichier **composer.json**, et changer la version de **laravel/framework** pour la **11.0** et de **laravel/sanctum** pour la **4.0**. Ensuite vous devez exécuter les commandes dans cette l'ordre suivant « composer require nunomaduro/collision » -> « composer update » -> « composer autoload ».

Vous pouvez dès à présent installer **Postman** qui est la dernière application dont vous aurez besoin pour développer avec Laravel. Si vous ne connaissez pas Postman, c'est un outil qui vous permet de tester les API.

Vous pouvez finalement ajouter l'extension **PHP Intelephense** dans VS code qui vous aide dans votre développement avec Laravel.

5.4. Versionning avec Git

5.4.1. Bonnes pratiques

La gestion du projet se fait avec git, pour commencer il y a des bonnes pratiques à suivre avec git pour que d'autres développeurs comprennent votre architecture et pour que vous puissiez vous-même vous y retrouver dans le projet. Voici les plus importantes :

L'utilisation de branches, la fonctionnalité la plus importante de git est son modèle de branchement. Ces branches vous permettent de gérer le flux de travail plus rapidement et plus facilement. En divisant vos différentes tâches par type comme **Bugfix**, **Hotfix**, **Release** ou encore **Feature.**

Comme constaté sur le précèdent point, il est important de bien nommée ces branches, pour retrouver dans le branchement quel type de tâches est votre branche vous allez la nommée « **type/ nomdelatâche** » par exemple « **feature/header** » pour la création d'un nouveau header.

Ecrivez des messages de commit utiles ils doivent être simple, résumer votre travail et si possible ne pas dépasser plus de 50 char.

Utiliser les **pull request** a bonne escient. Dans le meilleur des cas il ne faut faire qu'une seul **pull request** par branche, ne fait celle-ci que lorsque vous êtes sûr que votre code est fonctionnel et qu'il ne cause aucun problème sur la branche **dev**.

5.4.2. Structure

La branche **main** est comme son nom l'indique le **main**. La branche principale ou va se retrouver vont se retrouver les branches pour la phase de test est la **dev.** En dessous de ça, vous retrouvez toutes les branches de **features/bugfix** pour le développement de l'application et pour corriger les bugs.

5.4.3. Commandes

Pour se connectez à git il vous faut utiliser les commandes suivantes :

```
PS C:\Users\pg78vgj> git config --global user.email "votre@email.com"
```
Erreur ! Signet non défini.

```
PS C:\Users\pg78vgj> git config --global user.name "votre nom"
```
Erreur ! Signet non défini.

Ensuite les commandes utiles sont la création de branche avec **Erreur ! Signet non défini.** et les commit qui ressemble à `PS C:\Users\pg78vgj> git branch leNomDeVotreBranch` **Erreur ! Signet non défini.**

6. Implémentation de l'application

6.1. Bonnes pratiques d'un projet Flutter

Dans le développement avec Flutter il y a des bonnes pratiques à respecter pour que l'application soit performante et pour que des développeurs qui est déjà fait du flutter puisse s'y retrouver assez vite. Les plus importantes sont les suivantes :

Dans plusieurs cas vous serez amené à rassembler des éléments d'un dans un seul bloc pour gérer ceux-ci. Pour ce genre de cas, si vous avez quelques bases sur Flutter vous allez penser à un **Container**, mais ceux-ci sont plus gourment et vous n'allez pas utilisé ce widget a son plein potentielle. Il vaut plutôt opter pour une **SizedBox** qui a été créer dans cette optique.

Il faut faire la différence entre un **Widget Stateless** et **Stateful**. Le premier est un élément statique, celui-ci est par exemple utiliser pour les **Text**, **Icon** et **Button**, en bref les widgets qui ne peuvent pas être influencer par les actions de l'utilisateur. Différence au deuxième qui lui est un élément dynamique, qui utiliser pour les **Checkbox**, **Radio Button** ou encore **Slider**, ici cette élément change par rapport à une action de l'utilisateur. Donc lorsqu'on créer des Widget il vaut mieux ce demandez si l'utilisateur peux avoir une influence dessus pour avoir le bon type.

Les constructeurs doivent être des constants pour éviter de recréer des nouveaux **Widgets** à chaque fois ce qui pourrait. Plus il y a de différentes pages plus ce point est important car s'ils ne sont pas des constants à chaque appel ça va recréer un **Widget** au lieu de recharger un déjà existant.

Bien sûr il y a des conventions de nommage qui sont les suivantes :

- Utiliser le **snake_case** pour les librairies, packages, répertoires et les fichiers.
- Les variables privées commencent avec un **underscores**.
- Utiliser le **lowerCamelCase** pour les constants, variables, paramètres.
- Utiliser le **UpperCamelCase** pour les classes, types, énumérateur et les nom d'extensions.

Il faut diviser votre code en plusieurs Widgets le plus que possible, cela peut éviter de recréer des éléments déjà existants et donc d'éliminer le code superflu.

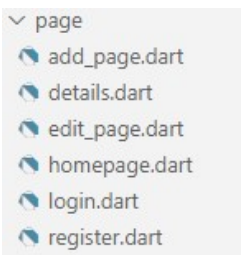
6.2. Installation des packages

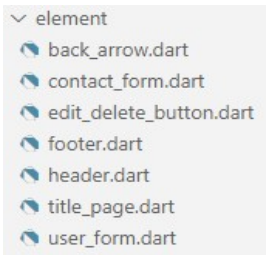
L'installation des packages et assez simple sous l'environnement Flutter, la première étape et bien sûr de l'installer, pour la suite il vous faut l'ajouter le chemin du packages dans un fichier pour pouvoir l'utiliser dans votre application. La gestion de ceux-ci se déroule dans le fichier **pubspec.yaml**. Par exemple si vous voulez utilisez une police d'écriture spécifique qui ne se trouve pas dans les polices dont Flutter dispose vous avez juste à référencer celle-ci dans le fichier. Le projet n'a besoin actuellement que d'assets telle que les images et les polices d'écritures.

6.3. Création des interfaces

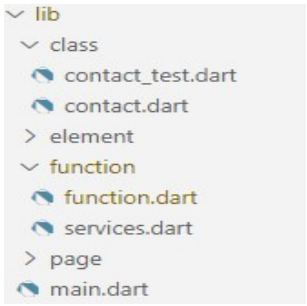
Comme vu dans le point *Maquette* il y a 4 interfaces principales et 2 annexes qui sont dérivé d'une des interfaces principales. Si vous ne vous êtes pas encore identifier ou alors que vous n'avez pas créer un compte vous arrivez sur la page de connexion. Dès que votre authentification s'est faite vous arriver sur la homepage, elle contient la liste des utilisateurs et depuis celle-ci vous pouvez accéder à l'interface détail en cliquant sur un contact et dans depuis le footer vous pouvez accéder à la page d'ajout de contact. A préciser que le footer se retrouve sur toutes les pages. Toutes les interfaces sont divisées en 3 partie le header, le content et le footer. Il n'y a que le content qui est différent sur chaque page.

Le code est fragmenté de manière utile, c'est-à-dire que s'il y a plus de 2 fois le même block il est créé en élément pour pouvoir être utiliser sur plusieurs pages. L'exemple parfait est le header, vu que celui-ci est sur chaque interfaces il faut qu'il vaut mieux le créer une fois et pouvoir l'appeler sur les différentes interfaces.





Les autres fichiers qui sont présent dans ce projet sont le fichier function.dart, qui a pour utilité de contenir la grande majorité des fonctions dont le fichier a besoin pour fonctionner. Le fichier contact.dart qui est la classe qui va contenir tous les contacts enregistrer par l'utilisateur. Et bien sur le fichier main.dart qui lance l'application.



6.4. Test avec une API

Il a fallu tester que l'application puisse se connecter et utilisé des fichier json avant de passer à Laravel. Pour cela, il faut trouver une api qui génère des fichiers json pour afficher des datas de tests. Ici l'application a utilisé l'api **JsonPlaceHolder** pour pouvoir consommé du fichier json. L'utilisation d'API demande de rajouter une petite ligne de commande qui a été référencer dans le point *Environnement de développement Flutter*. Malheureusement, l'application ne peut pas que lire les datas du fichiers json et ne peut pas les modifier ou les supprimer. Donc à ce moment les tests avec l'API ne peuvent pas aller plus loin.

Vous avez besoin d'un fichier qui effectue la connexion vers l'API pour faciliter son utilisation. Ici c'est le fichier remote_service.dart qui prend en charge cette fonctionnalité.

6.5. Gestion des API

Les requêtes sont effectuées avec **Dio** dans Flutter, si vous le désirer vous pouvez utiliser **HTTP** qui fonctionne aussi bien. Dans tous les cas il vous faut configurer vos requêtes, pour cela vous pouvez devez préciser l'url, le headers et le token d'authentification s'il existe. Voici les paramètres :

```
var dio = Dio(BaseOptions(
    baseUrl: 'http://10.0.2.2:8000/api/',
    headers: {
      'accept': 'application/json',
      'content-type': 'application/json',
    },
    responseType: ResponseType.json)); // BaseOptions // Dio
```

Si votre application Laravel est en local, vous devez utiliser cette url avec le port que vous avez défini dans votre application. Sinon utiliser l'url de votre application serveur.

Celui-ci est géré dans le fichier **services.dart** et après vous pouvez faire appel à votre élément Dio dans un autre fichier pour pouvoir effectuer des requêtes http. Par exemple voici à quoi ressemble la requête pour récupérer tous les utilisateurs :

```
Future<List<Contact>> getContact() async {
  String? id = await getUserId();
  Dio.Response response = await dio().get('contact/$id');
  List listcontact = response.data;
  return listcontact.map((contact) => Contact.fromJson(contact)).toList();
}
```

6.6. Gestion de l'authentification

Dès que le token a été créée dans l'application Laravel, il faut que vous le stocker dans l'application Flutter. Pour ça vous pouvez utiliser **SharedPreferences** qui vous permet d'enregistrer dans le stockage de l'application, le token qui vous a été attribuer. Sur le passage vous pouvez aussi stocker l'id de l'utilisateur qui vous sera utile pour vos requêtes.

```
SharedPreferences preferences = await SharedPreferences.getInstance();
preferences.setString('authToken', response.data['token']);
preferences.setString('user_id', "${response.data['user_id']}");
```

Dès que le token est créer il faut l'ajouter dans les paramètres de Dio pour qu'il puisse le transmettre à l'application Laravel. Donc dans le fichier **services.dart** ajouter aux headers votre token :

```
dio.interceptors.add(InterceptorsWrapper(
  onRequest:
      (RequestOptions options, RequestInterceptorHandler handler) async {
    //Ajoutez le token à l'en-tête de la requête si le token est disponible
    if (await doesTokenExist()) {
      String? authToken = await getAuthToken();
      options.headers['Authorization'] = 'Bearer $authToken';
    }
    return handler.next(options); // Continuez le flux de la requête
  },
```

Bien évidemment, si l'utilisateur se déconnecte il faut supprimer le token dans les 2 applications. Du coté Flutter voici la fonction qui s'en occupe :

```
Future<void> removeAuthToken(BuildContext context) async {
  SharedPreferences preferences = await SharedPreferences.getInstance();
  Dio.Response response = await dio().delete('auth/token',);
  preferences.remove('authToken');
  preferences.remove('user_id');
  goToLoginPage(context);
}
```

Faite bien attention à supprimer d'abord envoyer la requête de suppression à l'application Laravel avant de le supprimer, sinon il ne pourra pas supprimer le token dans la DB.

6.7. Implémentation de CSRF

Il y a un problème pour cette partie, vu que l'application serveur ne génère pas de session pour l'application flutter. L'api n'arrive pas à utiliser à générer un token **CSRF**. D'ailleurs l'utilité du CSRF n'est pas sûr car il fait pour protéger les données envoyer depuis un formulaire mais cette fonctionnalité est plus orienté pour un site web qu'une application.

7. Implémentation de l'application serveur

7.1. Création de l'application Laravel

L'application contient 2 modèles **Contact** et **User** qui serves à faire le passage entre les données qui sorte de la DB et celles qui viennent de l'application. Il y a aussi 2 contrôleurs **UserController** et **ContactController**. Le premier est celui qui va permettre à l'utilisateur de s'identifier ou de créer un nouvel utilisateur, dans les 2 cas il va aussi permettre la création d'un token d'authentification. L'autre contrôleur va permettre d'utiliser les fonctionnalités CRUD sur les éléments contacts qui appartiennent bien sûr uniquement à l'utilisateur en question.

Toutes les routes qui mènent à l'utilisation des données des contacts ont besoin du token d'authentification ce qui semble logique car sans ce token impossible de savoir qui est connectez et donc de traiter uniquement ces données. Des informations supplémentaires sont présentes dans le prochain point *Système d'authentification*.

7.2. Système d'authentification

L'authentification est gérée par **Sanctum** dans Laravel, sont fonctionnement est simple il va examiner la requêtes HTTP qui arrive et vérifier dans les cookies s'il y un cookie d'authentification. Bien sûr dans le cas de l'application elle ne possède pas de cookies donc après la vérification de ceux-ci il va examiner le Header d'autorisation pour trouver le token API est donc identifié l'utilisateur.

Laravel Sanctum

Les Routes dans Laravel fonctionnent de 2 manières différentes dans le cas de cette application. D'abord il y a les Routes sans authententification qui vont permettre à l'utilisateur soit de ce créer un compte ou alors de s'identifier. Ces 2 routes vont créer par la suite un token qui permet de maintenir la session de l'utilisation ouverte.

```
Route::controller(UserController::class)->group(function() {
    Route::post('register', 'storeUser');
    Route::post('login', 'authUser');

});
```

Puis le 2eme type de routes, pour accedez à celle-ci l'utilisateur doit obligatoirement avoir créer un token via la connexion ou l'enregistrement de ces informations. Après ça il peut donc accéder aux fonctionnalités d'affichage de ces contacts, de suppressions, d'ajout et de modification de ceux-ci.

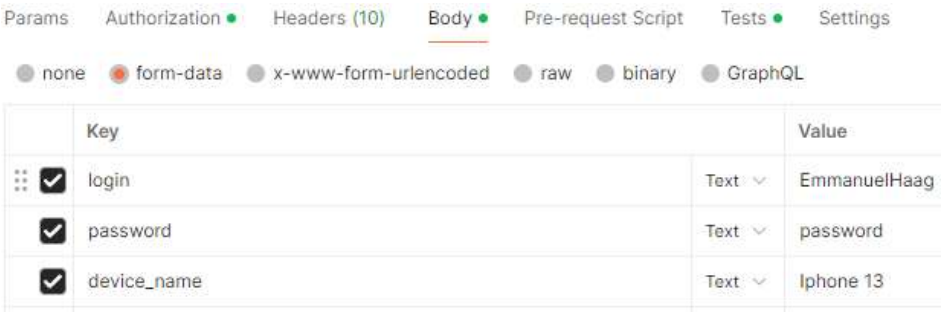
```
Route::middleware('auth:sanctum')->group(function () {
    Route::get('contact/{id}', [ContactController::class, 'index']);
    Route::get('search-contact/{id}/{search}', [ContactController::class, 'search']);
    Route::delete('delete-contact/{id}',[ContactController::class, 'destroy']);
    Route::post('add-contact/{id}',[ContactController::class, 'storeContact']);
    Route::put('edit-contact/{id}',[ContactController::class, 'update']);
    Route::delete('auth/token',[UserController::class, 'removeToken']);
});
```

7.3. Mise en place de l'API

Après avoir créé l'application, il l'essayer avant de l'implémenter a l'application Flutter. Pour cela l'utilisation de Postman semble parfaite, pour pouvoir effectuer les tests avec Postman il va falloir faire quelques ajustements de certains paramètres. De plus vous pouvez lancer la commande « php artisan db:seed » qui vous génère des données de tests.

Dans Postman, il faut commencer avec la requête login, dans une méthode **POST** puis entrez votre URL. Avant d'envoyer la requête, vous devez rajouter dans le **Headers** et rajouter les informations suivantes :

Et aussi ajouter les informations pour la connexion dans le **Body** :



Après ça vous pouvez envoyer votre requête, qui va en plus de créer un utilisateur va aussi créer un token d'authentification que vous devez récupérer.

Ensuite vous allez créer une requête de la méthode que vous avez choisi, par exemple si vous désirez récupérer tous les contacts liés à votre utilisateur, vous faites une requête **GET**. Il vous faut donc ajouter le token dans l'**Authorization** avec le type **Bearer Token.** Ajouter aussi l'ID de l'utilisateur dans l'URL et voula vous pouvez maintentant tester vos Routes.

8. Conclusion