

Macaco, l'application de message journalier : Documentation technique

4LABO - Supinfo 2023-2024

Corentin HOAREAU - étudiant M.eng1

Sommaire

- Macaco, l'application de message journalier : Documentation technique
- Sommaire
- Introduction
- Lancement du projet
- L'application
 - Composantes principales
 - Calendrier
 - Message
 - Fonctionnalités de l'application
 - User stories
 - Fonctionnalités bonus
- Présentation technique
 - La stack technique
 - Backend
 - Schéma de données
 - Architecture backend
 - Fonctionnalités backend
 - Déploiement sur le cloud
 - Frontent
 - Premiers pas avec Dart et Flutter
 - Architecture du frontend
 - Gestion de l'état
 - Bloc VS Cubit
 - Fonctionnement de la librairie BLOC
 - Fonctionnement de la librairie BLOC : exemple
 - Quand ne pas utiliser BLOC
 - Le reste du projet
- Conclusion

Introduction

Ce document technique sert à présenter de manière approfondie l'application Macaco, réalisée dans le cadre du 4LABO à Supinfo International University.

Lancement du projet

Une APK Android est fournie dans l'archive de rendu du projet. Cette APK se connecte à la version Cloud du serveur. Si vous souhaitez lancer les versions de test du backend et de l'application mobile, les instructions sont fournies dans chacun des fichiers README.md des sous-dossiers respectifs.

L'application

Composantes principales

Macaco est une application de partage, vous permettant de faire découvrir à vos proches une nouvelle musique par jour. Pour comprendre son fonctionnement, il faut comprendre deux concepts : les calendriers, et les messages.

Calendrier

Un calendrier peut avoir des abonnés et des contributeurs. Il peut aussi être protégé, pour que seuls les abonnés aient accès aux messages ;

Tout utilisateur est libre de s'abonner à un calendrier public. Lorsqu'il s'agit d'un calendrier protégé, une requête est envoyé à ses contributeurs, ils choisissent alors d'accepter ou non ce nouvel utilisateur.

Un calendrier peut avoir un ou plusieurs contributeurs, les contributeurs d'un calendrier peuvent y ajouter des messages, et accepter les requêtes d'abonnement d'autres utilisateurs. Un contributeur peut aussi retirer des abonnés existants.

L'administrateur d'un calendrier est un contributeur avec des privilèges élevés, il est capable de modifier les paramètres du calendrier, et le supprimer. C'est l'administrateur qui choisit d'inviter de nouveaux contributeurs, il peut aussi retirer des contributeurs existants, et passer le rôle d'administrateur à un des contributeurs si il ne souhaite plus être administrateur du calendrier.

Message

Un calendrier contient des messages (ou messages journaliers). Un message représente une musique que les contributeurs du calendrier souhaitent partager avec les abonnés.

Le message est constitué d'un titre, et d'un lien (souvent Youtube) vers la chanson.

L'auteur peut ajouter un contenu au message pour décrire la musique, ou partager ses opinions.

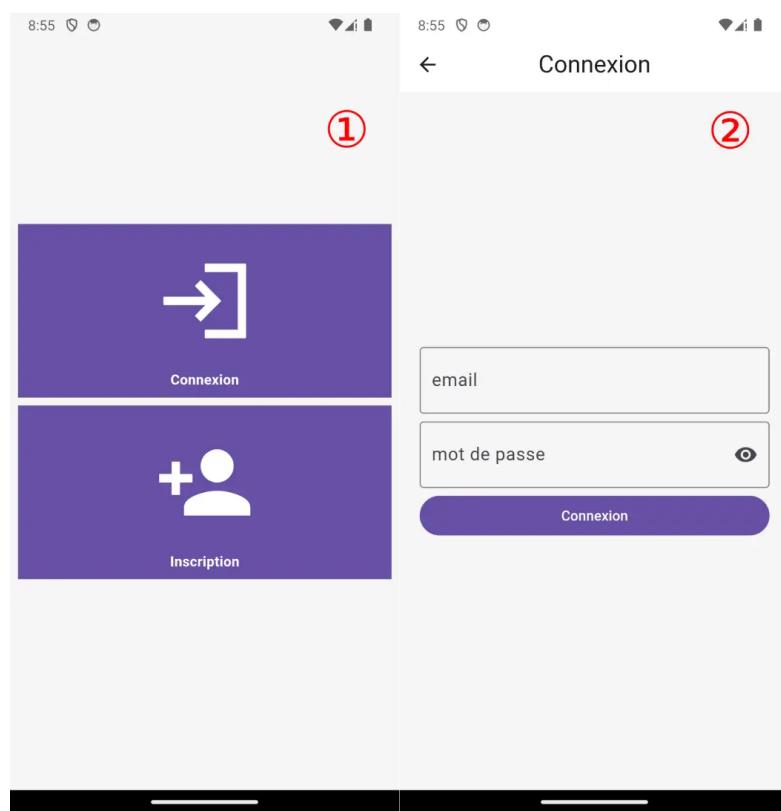
Fonctionnalités de l'application

Cette section a pour but de présenter l'intégralité des fonctionnalités de l'application.

User stories

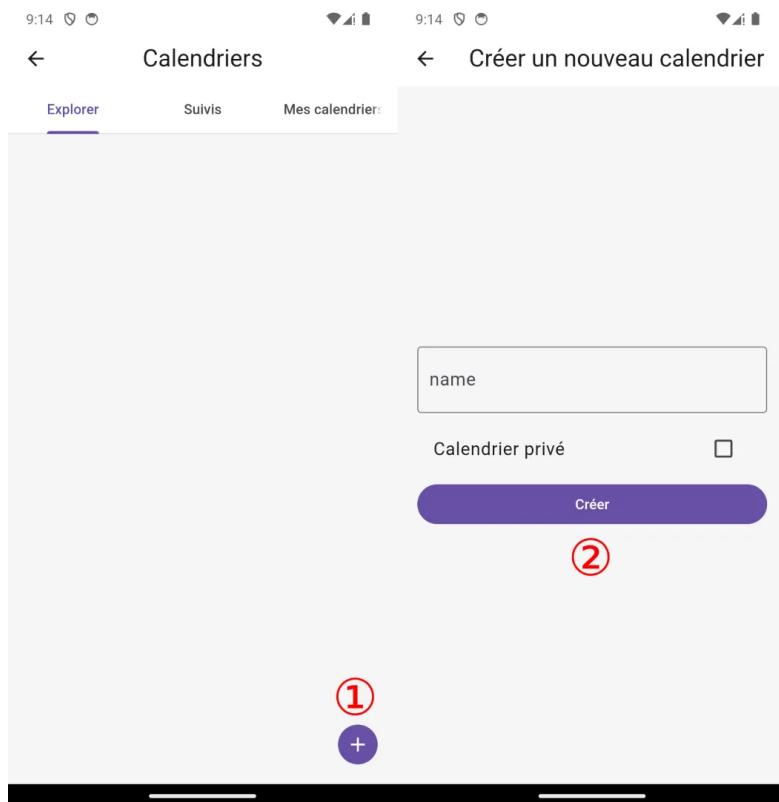
Les user stories suivantes ont été rédigées lors de la proposition de projet au Jury, elles sont rédigées plus en détail dans le fichier README du projet Flutter :

- Inscription et authentification
1. La page d'accueil contient deux boutons, connexion et inscription ;
 2. Chaque bouton mène au formulaire correspondant.



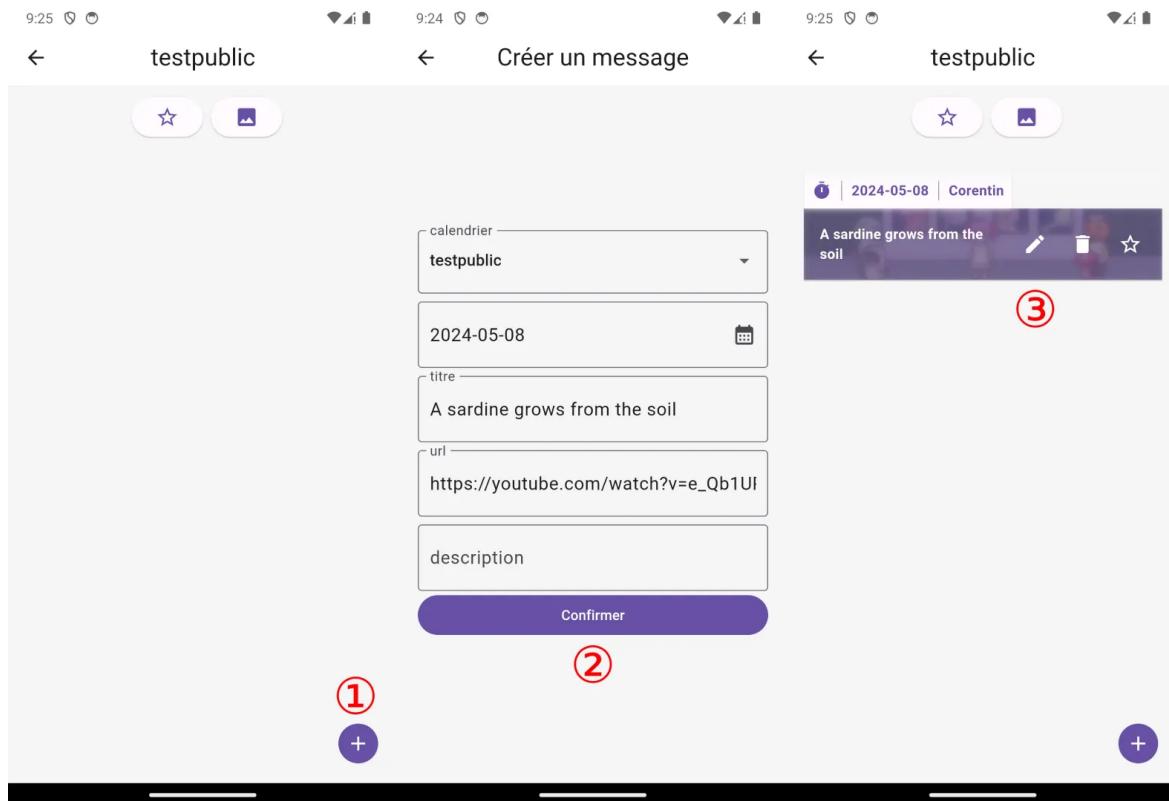
- **Création du calendrier**

1. La page de création d'un calendrier est disponible sur la page listant les différents calendriers ;
2. Le bouton "+" mène au formulaire de création.



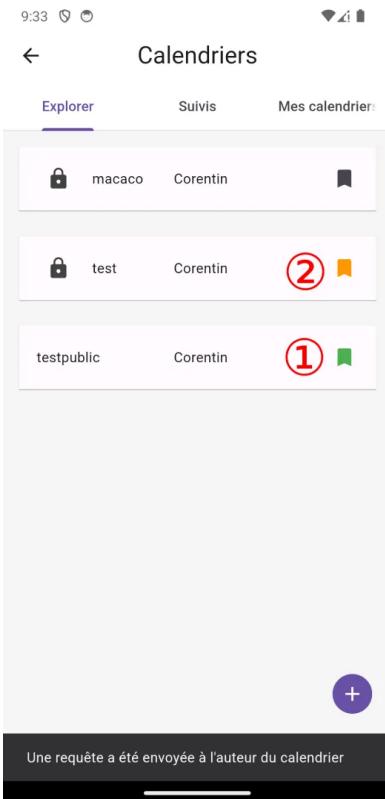
- CRUD des messages journaliers

1. La création de messages est disponible sur la page principale d'un calendrier ;
2. Le bouton "+" mène au formulaire de création ;
3. Un contributeur peut facilement modifier et supprimer les messages d'un calendrier.



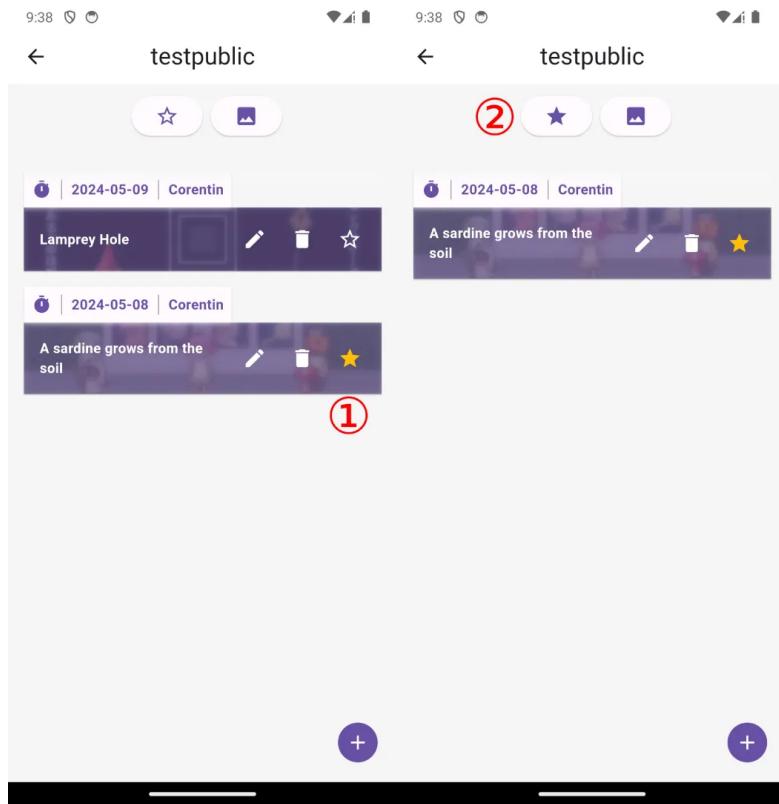
- Aboonnement à un calendrier

1. L'icône "signet" permet de s'abonner à un calendrier public ;
2. Dans le cadre des calendriers protégés, une couleur différente indique le status "en attente" de la demande d'abonnement.



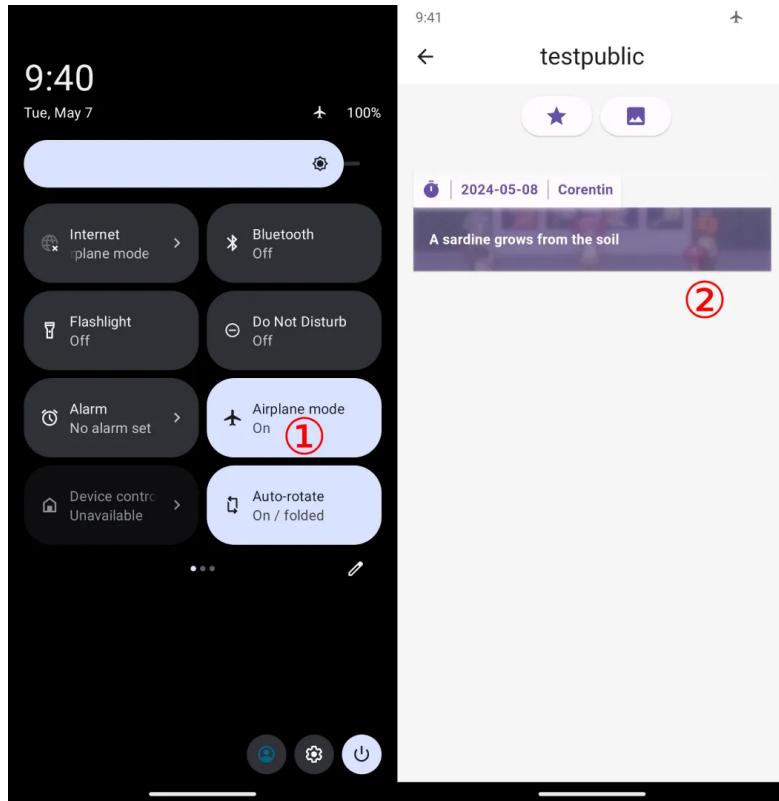
- Messages favoris

1. L'icône d'étoile permet l'ajout d'un message à la liste des favoris ;
2. Les favoris d'un calendrier peuvent être filtrés grâce au bouton en haut de l'écran.



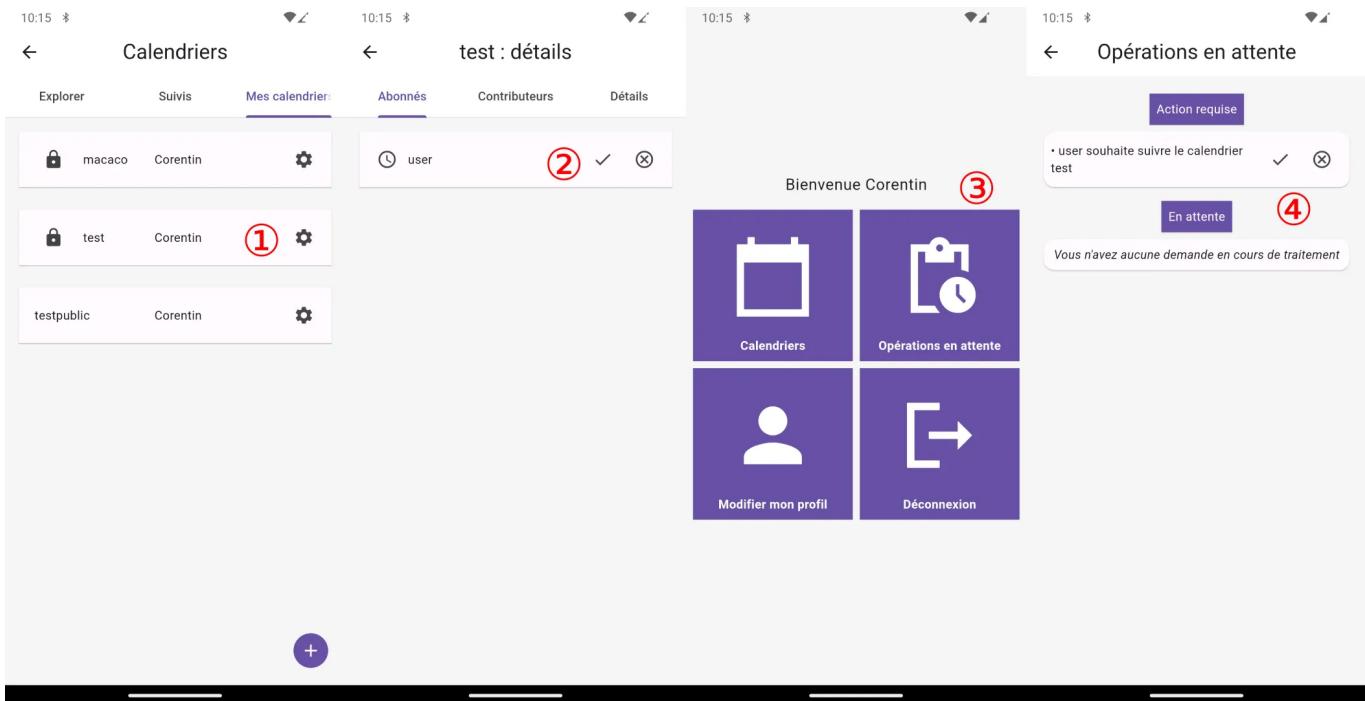
- Sauvegarde locale

1. Un base de donnée locale est mis en place pour mettre en place chaque calendrier et message journalier téléchargé ;
2. L'application en mode hors-ligne est épurée des fonctionnalités nécessitant une connexion internet, pour un affichage plus léger.



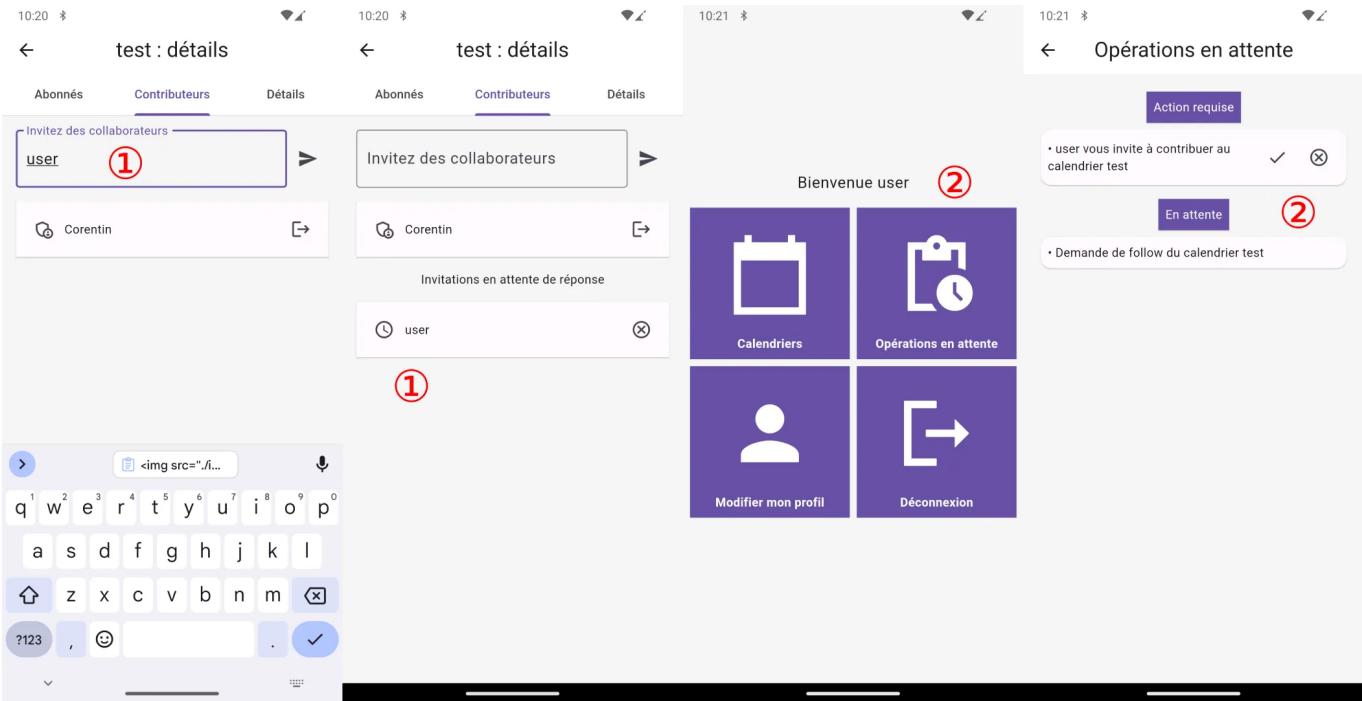
- Protection des calendriers

1. Un contributeur a accès à la page de gestion d'un calendrier
2. Sur cette page accès aux demandes d'abonnement et peut les accepter/refuser ;
3. L'écran principal de l'application contient aussi une page "opérations en attente" ;
4. Cette page liste les demandes d'abonnement sur chaque calendrier où l'utilisateur est un contributeur.



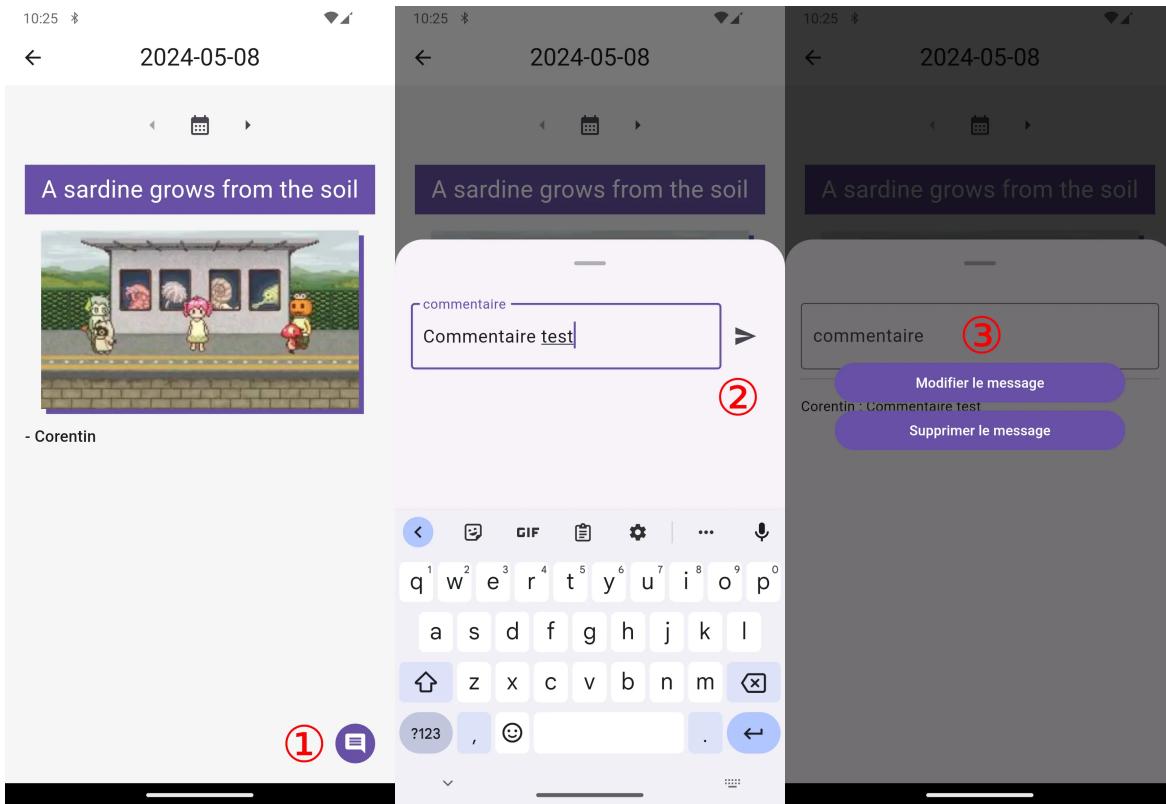
- Invitation de contributeurs

1. L'administrateur du calendrier peut envoyer des invitations à des utilisateurs ;
2. Sur la page "opérations en attente", un utilisateur peut voir et traiter les invitations reçues.



- Gestion des commentaires

1. Chaque message journalier a un espace "commentaire" ;
2. La modale de l'espace commentaire permet de voir et écrire des commentaires ;
3. Un appui long permet d'éditer/supprimer un commentaire

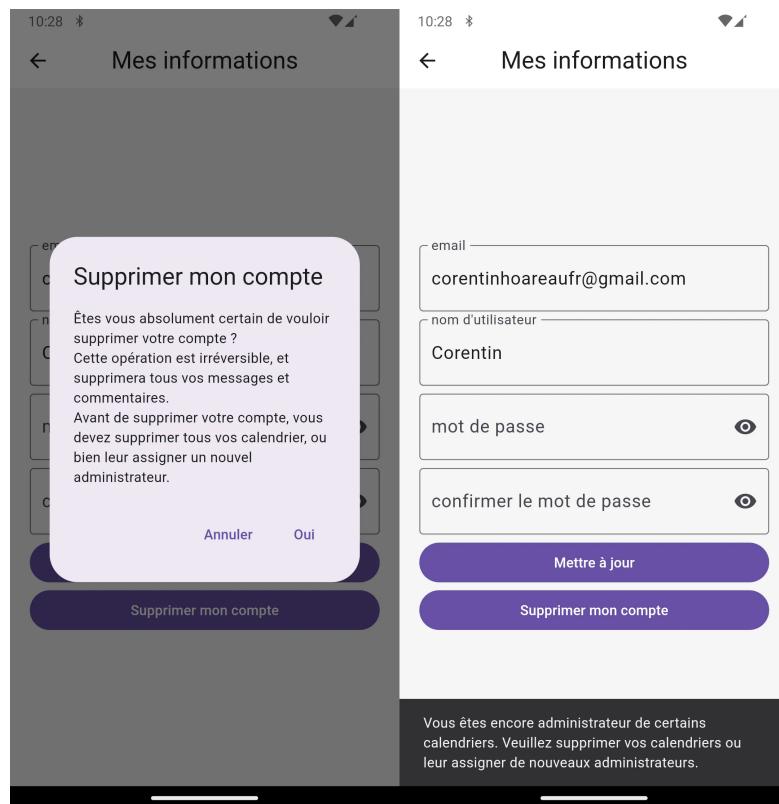


Fonctionnalités bonus

Ces fonctionnalités ont été réalisées en plus de celles originalement planifiées lors de la proposition de projet. Il peut s'agir de fonctionnalités additionnelles, ou bien d'amélioration de la qualité de vie pour l'existant

- Protection/validation

1. Chaque opération de suppression passe par un message de confirmation pour éviter les manipulations accidentelles ;
2. Un utilisateur ne peut pas supprimer son compte si il est administrateur d'au moins un calendrier, il est averti des tenants et aboutissants de la suppression de son compte.



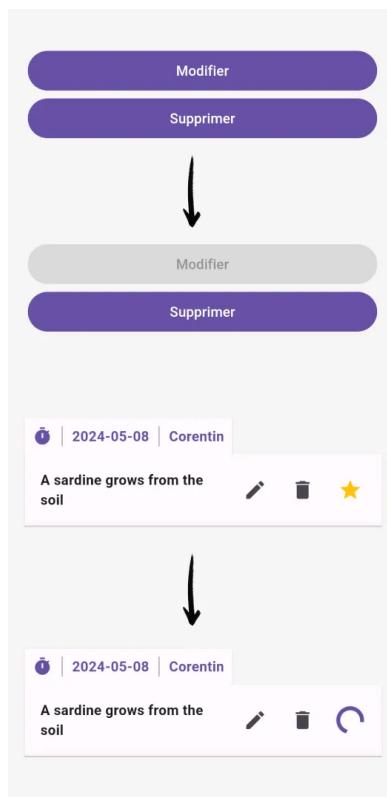
- Sauvegarde des préférences

1. L'onglet choisi dans la navigation des calendriers, ainsi que d'autres préférences, sont conservées même après la fermeture de l'application, évitant alors les opérations superflues pour les utilisateurs ayant les mêmes habitudes de navigation.

```
I/flutter (14527): [  #0  SecureStorageProvider.setPreference (package:daily_message_app/common/providers/local_storage_provider.dart:57:12)
I/flutter (14527): | #1  GlobalCubit.setOnlineCalendarListTab (package:daily_message_app/common/cubit/global_cubit.dart:35:28)
I/flutter (14527): |
I/flutter (14527): | 📁 Saving preference calendarListTab: 1
I/flutter (14527): |
D/EGL_emulation(14527): app_time_stats: avg=16.93ms min=6.31ms max=30.17ms count=59
I/flutter (14527): |
I/flutter (14527): | #0  SecureStorageProvider.setPreference (package:daily_message_app/common/providers/local_storage_provider.dart:57:12)
I/flutter (14527): | #1  GlobalCubit.setOnlineCalendarListTab (package:daily_message_app/common/cubit/global_cubit.dart:35:28)
I/flutter (14527): |
I/flutter (14527): | 📁 Saving preference calendarListTab: 2
I/flutter (14527): |
I/flutter (14527): |
```

- Réactivité

1. L'utilisateur est correctement averti lors d'un chargement de données ;
2. Les requêtes asynchrones déclenchées par un bouton offrent un retour visuel direct à l'utilisateur.

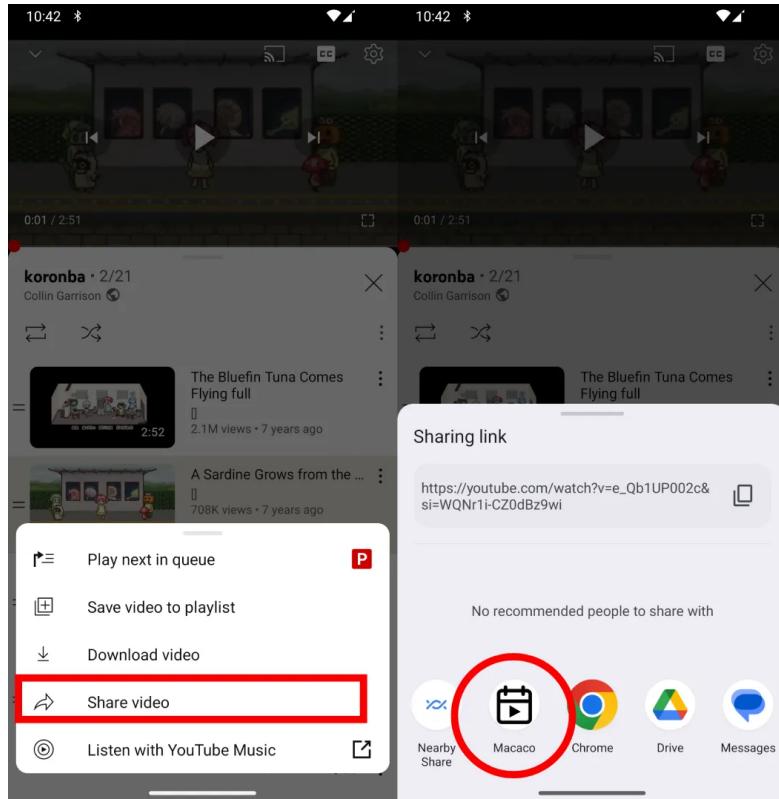


- Rafraîchissement et polling

1. Les pages contenant de la donnée peuvent être rafraîchies manuellement par l'utilisateur ;
2. La page "opérations en attente" actualise périodiquement ses données.

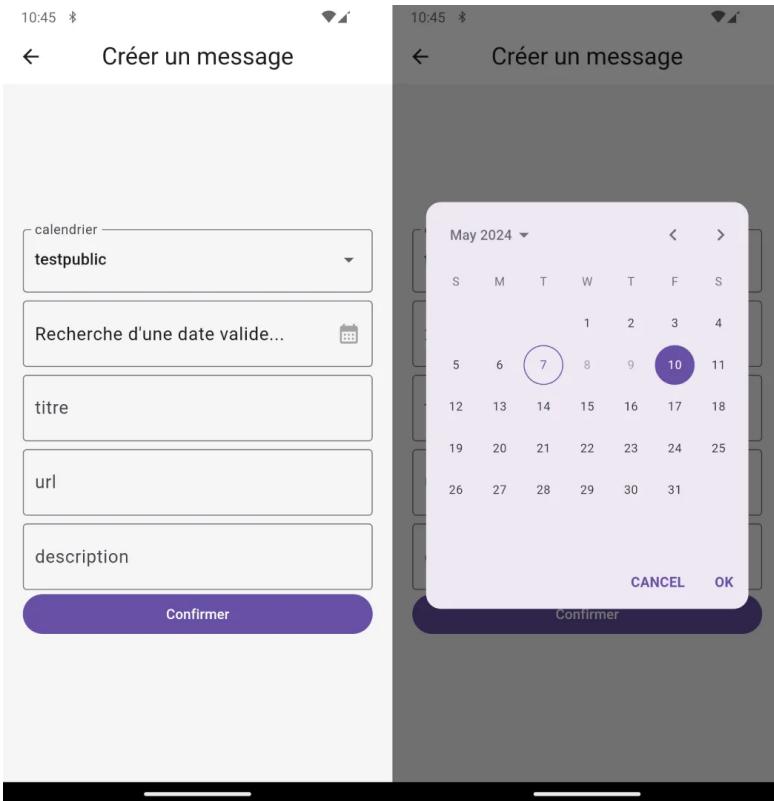
- Intégration Android : partage

1. L'application s'intègre au menu "partager" d'Android ;
2. Le lien est alors directement intégré au message, plus qu'à choisir le calendrier.



- Gestion des conflits de date

1. Lorsqu'on crée un message, la prochaine date disponible la plus proche est automatiquement choisie ;
2. Un calendrier interactif permet de changer la date du message, et désactive automatiquement toutes les dates déjà prises pour éviter les conflits.



- Navigation avancée

1. Un mode "liste" permet d'afficher de manière succincte tous les messages d'un calendrier ;
2. Lorsque l'on affiche les détails d'un calendrier, il est possible de naviguer aux dates adjacentes grâce au volet de navigation ;
3. Le sélecteur de date interactif permet aussi de naviguer à un message précis ;
4. Les dates sans messages sont automatiquement désactivées.

Présentation technique

Cette partie relatera les différents choix de conception ainsi que mes retours d'expérience quant à la création de l'application Macaco.

La stack technique

L'application se compose d'un backend et d'un frontend.

Le backend est écrit en Javascript avec le framework NestJS. Les données sont stockées dans une base de données, avec laquelle le backend interagit par l'intermédiaire de l'ORM Prisma.

Le frontend est une application mobile basée sur le framework multi-plateforme Flutter, s'appuyant sur le langage de programmation Dart. Deux technologies créées par Google dans le but d'offrir une alternative à Javascript.

Backend

Schéma de données

Une approche "schema first" est utilisée pour ce projet.

Le schéma de données est décrit par le schéma Prisma, puis synchronisé à la base de données. Le client Prisma permet un typage fort dans le code sans besoin de réécrire les modèles de chaque entité.

Le mappage des noms permet de respecter les conventions de nommage en passant du snake-case au pascalCase avec un effort minimal :

`is_private (PostgreSQL) => isPrivate (Javascript).`

Le schéma de données est le suivant :

Enums

ERole
USER
ADMIN

Tables principales

User	DailyMessage	Calendar
id: Int username: String email: String role: ERole password: String createdAt: DateTime	id: Int title: String calendarCode: String userId: Int day: String content: String media: String createdAt: DateTime	code: String name: String isPrivate: Boolean createdAt: DateTime
MessageComment		
	id: Int messageId: Int userId: Int content: String createdAt: DateTime	

Tables de liaison

UserContributor	UserCalendarFollowing	MessageFavorite
calendarCode: String userId: Int isCreator: Boolean createdAt: DateTime	calendarCode: String userId: Int createdAt: DateTime	messageId: Int userId: Int createdAt: DateTime
CalendarCollaboratorInvitation	CalendarFollowRequest	
calendarCode: String userId: Int message: String? createdAt: DateTime	calendarCode: String userId: Int message: String? createdAt: DateTime	

Architecture backend

NestJS fonctionne avec des modèles et des contrôleurs. La racine du code métier présente différents sous dossiers (user ; calendar ; message ; etc...), ceux-ci contenant modules, services, et contrôleurs. Il s'agit donc d'une architecture découpée par fonctionnalités (Feature-Driven architecture) et non par couches (Layer-Driven).

NestJS fait forte utilisation de l'injection de dépendance, un module exporte des services et des providers, et inversement, importe d'autres modules, c'est de cette manière que le code peut être partagé.

(Note : Le client Prisma permet d'outrepasser, d'une certaine manière, cette dichotomie, car il s'agit d'un unique client partagé, contenant à lui seul les fonctionnalités CRUD de chaque table en base de données. La manière dont Prisma couple fortement les relations est pratique, mais peu recommandable si l'on souhaite migrer à une infrastructure par micro-services, il s'agit donc d'une fonctionnalité à utiliser de manière responsable).

Fonctionnalités backend

NestJS est la technologie que j'ai principalement utilisée ces deux dernières années dans le cadre de mon alternance. L'expérience, les connaissances, et les bonnes pratiques accumulées avec ce framework ont beaucoup simplifié la réalisation du backend.

Cette section décrira les différentes solutions techniques en place sur le serveur backend de Macaco.

- **Gestion de l'environnement**

Une combinaison de NestJS, Joi, et typescript permet la validation et le typage fort des variables d'environnement.

```
// stc/common/environment/index.ts
export const environmentValues = () => ({
  env: {
    port: process.env.APP_PORT,
    jwtSecret: process.env.JWT_SECRET,
    nodeEnv: process.env.NODE_ENV || 'production',
  },
});
export const environmentValidationSchema = Joi.object({
  APP_PORT: Joi.required(),
  JWT_SECRET: Joi.required(),
});
export type TEnvironmentValues = ReturnType<typeof environmentValues>['env'];
```

Les variables sont ensuite configurées dans le module principal de l'application.

```
// src/app.module.ts
// ...
ConfigModule.forRoot({
  load: [environmentValues],
  validationSchema: environmentValidationSchema,
  ignoreEnvFile: true,
  isGlobal: true,
}),
// ...
```

Enfin, l'utilisation des variables est rendue triviale grâce au typage fort défini précédemment.

Voici un exemple d'utilisation.

```
// src/auth/auth.guard.ts
// ...
const env = this.configService.get<TEnvironmentValues>('env');
const payload = await this.jwtService.verifyAsync(token, {
  secret: env.jwtSecret,
});
// ...
```

Cette technique a été mise en place lors d'un de mes projets en alternance, sur un système avec un environnement plus complexe.

- Sécurité par défaut et évolutivité

Un guard personnalisé est utilisé pour l'authentification utilisateur, cela permet d'envoyer une requête au service utilisateur pour récupérer des informations à jour. Dans des systèmes plus complexes avec une gestion de rôle, la mise en place de RBAC (Role-Based Access Control) est facilitée.

L'architecture opère sur un principe de sécurité par défaut. C'est à dire que, au lieu de définir quels endpoints sont sécurisés par JWT, nous définissons plutôt le guard d'authentification sur l'intégralité de l'application, puis le désactivons au besoin grâce à un décorateur @Public.

Un autre concept de sécurité par défaut implémenté dans le backend est l'idée d'inférer toutes les requêtes de modification d'utilisateur depuis le JWT, au lieu de passer l'id d'utilisateur en paramètre d'URL. Si le backend venait à évoluer de sorte à ce qu'un rôle Admin permettrait de faire des opérations de modification/suppression sur des utilisateurs, alors il s'agirait de tout nouveaux endpoints disponibles exclusivement pour ledit rôle.

Bien que les fonctionnalités aient été définies lors de la proposition de projet, le backend a tout de même été pensé pour permettre une certaine évolutivité au-delà de ces limites (un découpage en une architecture par micro-services demanderait plusieurs refactors).

Déploiement sur le cloud

Ce projet étant pensé pour un usage familial, le backend doit pouvoir être déployé en dehors d'un environnement de développement local. Mon choix s'est tourné vers le cloud Azure, comme il s'agit de la solution cloud sur laquelle Supinfo m'a le plus formé.

L'architecture cloud est aussi simple que celle de développement local, soit une instance [Azure App Services](#), et une instance [Azure Database for PostgreSQL](#).

La connexion à PostgreSQL ainsi que les migrations se sont déroulés sans réel problème. Le déploiement du serveur backend s'est avéré plus complexe, quelques problèmes ont été rencontrés quant aux variables d'environnement, et au port de déploiement, qui a dû être paramétrisé (Azure App Service utilisant le port 8080 pour effectuer les vérifications d'état du serveur).

Frontent

Premiers pas avec Dart et Flutter

Ce projet a été ma première expérience avec la langage Flutter, j'ai effectué mon apprentissage des fondamentaux avec la documentation et des vidéos en lignes. Il en vient de même pour Dart de manière générale.

Dart est assez facile à assimiler pour un développeur Javascript. Se présentant comme une alternative audit langage, le langage s'utilise de manière assez similaire, mais avec un typage plus rigide (et ainsi plus robuste). Dart tente d'éviter de nombreux fonctionnements "implicites" de Javascript, tels que la coercion de type, par conséquent, il en devient plus verbose.

Mon expérience de développement avec Flutter a été facilitée par mon expérience sur React grâce à un point fondamental, le vocabulaire acquis :

Flutter est un framework avec une très bonne documentation officielle, mais une communauté réduite, il est donc difficile de trouver ce que l'on cherche en n'expliquant vaguement le problème. Il faut savoir le nom de ce que l'on cherche afin de pouvoir trouver la solution. En somme, il est plus facile de répondre au "comment" qu'au "quoi".

Architecture du frontend

Le frontend est aussi organisé sur un découpage par fonctionnalités, celui-ci se veut plus complexe à mettre en place, mais est plus facilement maintenable au moyen terme.

Gestion de l'état

L'état (state) de l'application correspond aux données stockées en mémoire. Plusieurs librairies proposent de faciliter la gestion de l'état, mon choix s'est porté sur la librairie Bloc.

Bloc découpe la gestion de l'état en trois couches :

- Data : les interactions directes avec l'API ou la source de données ;
- Business : le traitement des données reçues, la logique métier à effectuer, celle-ci est effectuée dans les Blocs/Cubits ;
- Presentation : les widgets et écrans utilisant les constructeurs/sélecteurs de Bloc.

L'apprentissage de cette librairie a été complexe et a requis plusieurs relectures de la documentation de la librairie pour plusieurs raisons :

- La définition de l'état d'un cubit est assez verbeuse ;
- L'apprentissage de la librairie s'est fait en parallèle de l'apprentissage de Flutter, et ainsi de la gestion d'état de Flutter (StatefulWidgets). Comme la librairie Bloc repose sur des concepts de gestion d'état natifs à Dart, il est plus difficile de l'appréhender lorsque l'on débute sur le langage.

Bloc VS Cubit

Bloc et Cubit permettent la gestion et modification de l'état, un Bloc est plus verbeux et la communication avec la couche de présentation se fait avec des évènements (ces évènements ont eux-même leur propre typage) et son implémentation est plus verbeuse.

Un Cubit est plus simple à comprendre car communiquant avec l'interface utilisateur en exposant des fonctions.

L'utilisation du Cubit est plus adaptée à des cycles de développement rapides, l'utilisation d'un Bloc est préférable si l'on souhaite un fort niveau d'observabilité quant aux évènements appelés.

Dans ce projet, seuls des Cubits sont mis en place.

Fonctionnement de la librairie BLOC

Voici comment un Bloc/Cubit est implémenté dans une application flutter :

1. Choix de l'écran ou de la fonctionnalité pour laquelle on veut créer un nouveau Bloc ;
2. Création de l'état du bloc, les `data class` n'existant pas dans Dart, il faut manuellement programmer :
 1. Les mises à jour partielles de l'état avec une fonction `CopyWith` ;
 2. Les comparaisons profondes d'objets avec la classe parent `Equatable` ;
 3. Sans ces deux implémentations, Bloc ne sera pas capable de détecter ou appliquer les changements d'état ;
3. Création du Bloc, le Bloc contient l'état de composant Flutter, ainsi que les fonctions que le composant peut appeler pour altérer cet état ;
 1. Pour envoyer/recevoir de la donnée, on injecte dans le Bloc un `repository`, le `repository` possède un ou plusieurs `providers`, dont le rôle est d'effectuer les requêtes à une source de données ;
4. Dans la couche de présentation :
 1. Créer le `repository` à l'aide d'un `RepositoryProvider` ;
 2. Dans les propriétés enfant du `RepositoryProvider`, créer le Bloc et y injecter le `repository` avec un `BlocProvider` ;
 3. Tous les composants enfant du `BlocProvider` peuvent maintenant accéder au Bloc, son état, et ses fonctions grâce au `contexte`.

Le `contexte` est un élément crucial de Flutter, il est essentiel au fonctionnement des Blocs.

Dans flutter, chaque Widget possède un contexte lui permettant de savoir où est-ce qu'il se trouve par rapport à l'arborescence des Widgets. Si un Widget parent utilise un `Provider` pour fournir une ressource, alors tous les widgets enfant peuvent utiliser leur contexte pour récupérer l'unique instance de la ressource fournie.

Il y a plusieurs manières d'utiliser les widgets fournis par BLOC dans flutter :

1. `BlocBuilder` récupère l'état complet du bloc et reconstruit tous les enfants dès que l'état change ;
2. `BlocSelector` récupère une propriété ciblée de l'état du bloc pour éviter de reconstruire inutilement l'arborescence ;
3. `BlocListener` permet d'effectuer de la logique n'impactant pas l'interface utilisateur lors d'un changement d'état.

Fonctionnement de la librairie BLOC : exemple

Prenons comme exemple le Cubit de l'écran de login.

L'état du Cubit contient les propriétés du formulaire de connexion.

```
sealed class LoginFormData extends Equatable {
    final AutovalidateMode autovalidateMode;
    final String email;
    final String password;
    final bool obscureText;

    const LoginFormData({
        /* OMIS : toutes les propriétés ci-dessus */
    });

    LoginFormData copyWith({
        /* OMIS : toutes les propriétés ci-dessus */
    });

    @override
    List<Object?> get props => [
        /* OMIS : toutes les propriétés ci-dessus */
    ];
}

class LoginFormUpdate extends LoginFormData {
    const LoginFormUpdate({
        /* OMIS : toutes les propriétés ci-dessus */
    });

    @override
    LoginFormUpdate copyWith({
        /* OMIS : toutes les propriétés ci-dessus */
    }) {
        /* OMIS : implémentation de copyWith */
    }
}
```

On peut constater la nature très verbeuse de la définition de l'état d'un Cubit, `get props` est une implémentation de la classe parent `equatable`, et `copyWith` permet de mettre à jour seulement une propriété de la classe.

L'implémentation du Cubit est comme suit :

```
class LoginFormCubit extends Cubit<LoginFormState> {
    LoginFormCubit(this._authRepository) : super(const LoginFormUpdate());

    final AuthRepository _authRepository;

    void initForm({
        String email = '',
        String password = '',
    }) {
        emit(state.copyWith(
            email: email,
            password: password,
        ));
    }

    void resetForm() {
        emit(LoginFormUpdate());
    }

    void updateAutoValidateMode(AutovalidateMode? autovalidateMode) {
        emit(state.copyWith(autovalidateMode: autovalidateMode));
    }

    /* OMIS : implémentation similaire de updateXXX pour les autres
    propriétés de l'état */

    Future<LoginModel> submit() async {
        return _authRepository.login(state.email, state.password);
    }
}
```

Le cubit contient les méthodes pour changer les différentes entrées du formulaire de connexion, ainsi qu'une méthode pour envoyer la requête de connexion, cette méthode utilise le `AuthRepository`. `AuthRepository` ne fait qu'implémenter `AuthProvider`, qui lui-même ne fait qu'envoyer les requêtes http. Dans d'autres cas, un `Repository` peut contenir deux providers, un pour les appels http, et un autre pour les appels à une base de données locale SQLite permettant la mise en cache des données, c'est le cas des calendriers et des messages.

Une fois Le cubit créé, il est implémenté dans la couche de présentation :

```
/* OMIS : Arborescence flutter classique */
body: RepositoryProvider(
  create: (context) => AuthRepository(),
  child: BlocProvider(
    create: (context) =>
      LoginFormCubit(context.read<AuthRepository>()..initForm(),
/* OMIS : Arborescence du formulaire */
    child: TextFormField(
      initialValue:
        context.read<LoginFormCubit>().state.email,
      validator: validateEmail,
      keyboardType: TextInputType.emailAddress,
      onChanged:
        context.read<LoginFormCubit>().updateEmail,
      decoration: InputDecoration(
        labelText: 'email',
        border: OutlineInputBorder(),
      ),
    )
/* OMIS : Reste de l'arborescence */
))
```

Le champ `email` utilise le Cubit pour lire et mettre à jour la propriété `email` de l'état du Cubit. L'accès au cubit est possible grâce aux Providers plus haut dans l'arborescence.

Quand ne pas utiliser BLOC

BLOC n'est pas considéré dans ce projet comme un "remplacement" des StatefulWidget. Lorsqu'une fonctionnalité nécessite des changements d'état, mais est purement esthétique, il est possible de préférer l'utilisation de la solution native de Flutter.

```
class _CalendarDetailsScreenState extends State<CalendarDetailsScreen> {
    bool isLoading = false;

    void fetchCalendarDetails(BuildContext context) {
        setState(() {
            isLoading = true;
        });
        context
            .read<CalendarDetailsCubit }()
            .fetchCalendarDetails(widget.calendarCode)
            .then((value) {
        setState(() {
            isLoading = false;
        });
    });
}
/* OMIS : Reste du StatelessWidget */
}
```

Si l'on voulait implémenter la variable `isLoading` dans l'état du Cubit, il aurait fallu l'écrire à sept endroits différents (le constructeur de la classe abstraite, `copyWith`, `get props`, etc...).

En somme, BLOC est une librairie utile pour la gestion d'état, mais il faut aussi savoir quand se rabattre sur la gestion d'état traditionnelle pour éviter un mauvais retour sur investissement.

Le reste du projet

Le reste du frontend s'appuie sur une connaissance des différents Widgets mis à disposition par Flutter, ainsi que d'une compréhension des concepts de gestion d'état précisé précédemment. Il a fallu se familiariser avec les outils du Framework, et savoir quand prendre des raccourcis pour pouvoir délivrer le projet dans les temps, tout en ne sabotageant pas les futures possibilités d'évolution/refactor.

Conclusion

En conclusion, ce projet a été une excellente épreuve pour mettre en pratique les compétences full-stack acquises au sein de Supinfo.

Le frontend s'est avéré être la partie la plus complexe du projet, car demandant dans de cours délais l'apprentissage simultané et autodidacte de :

- Dart ;
- Flutter, le framework utilisant dart ;
- Bloc, la librairie de gestion d'état utilisant Dart et Flutter.

Ce challenge aura toutefois été un succès, ayant augmenté mon niveau de confiance quant à la création d'application mobile, il s'agit aussi du tout premier serveur que j'ai déployé sur le cloud dans un cadre externe aux projets guidés de Supinfo.

J'ai beaucoup apprécié travailler sur ce projet, et j'espère que cela pourra se ressentir à travers le code et l'expérience utilisateur dans l'application. Merci pour votre lecture.