
Développement mobile avancé, IoT et embarqué
Gestion du State

Rapport du TP2

Étudiante :
GARCIA Léa

1 Lien vers dépôt GitHub TODO bien écrire le read me

Lien vers dépôt GitHub <https://github.com/Ravenn2203/RenduTP2Flutter.git>

2 Utilisation des Providers à la place de la fonction setState()

Le projet associé à cet exercice s'appelle 'tp2-w-providers'. Vous le trouverez sur le github, vu que c'est le même exercice que le TP1 je n'ai pas remis de screenshot de l'interface puisqu'elle est totalement identique !

2.1 Structure du projet

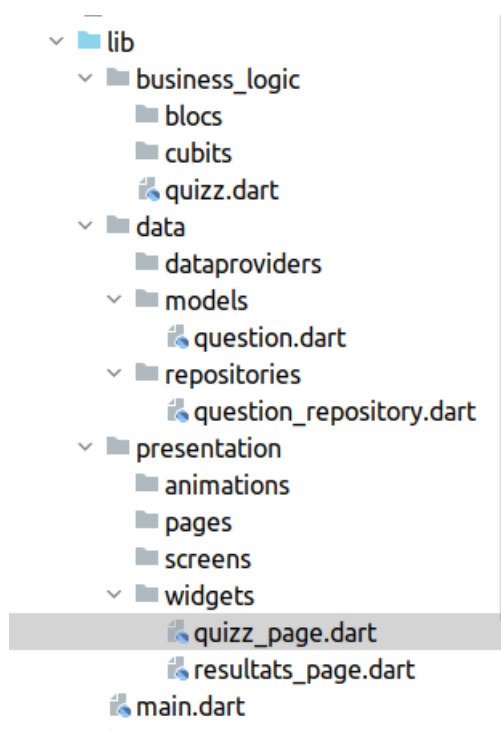


FIGURE 2.1 – Packages et classes permettant de structurer le projet par responsabilités

En suivant le cours j'ai divisé mon projet par package de responsabilités avec les packages : presentation, data et business-logic, contenant eux mêmes d'autres packages, contenant quant à eux des classes .dart.

2.2 Mise en place des providers

Afin d'utiliser les providers j'ai créé une classe Quizz, instance de ChangeNotifier. Dans cette classe j'ai déplacé toutes les variables que j'utilisais pour mettre à jour mon UI, pour contrôler le quizz, pour récupérer les prochaines questions, etc, dans le TP précédent.

Ensuite, j'ai créé deux fonctions pour remplacer les deux `setState()` différents que j'avais jusque là, une pour contrôler ce qu'il se passe lorsqu'on doit passer à une autre question et une autre pour contrôler la couleur des boutons lorsque l'utilisateur clique. Dans ces deux fonctions, après avoir fait les affectations je fais appel à la fonction `notifyListeners()` qui s'occupe d'informer les widgets qui l'écoutent, que le widget a besoin d'être reconstruit.

```

5  class Quizz extends ChangeNotifier{
6
7      bool aRepondu = false;
8      int nbQuestions = 7;
9      int nbQuestionsPosees = 6;
10     int nbBonnesReponses = 0;
11     Color couleurBouttonVrai = Colors.grey;
12     Color couleurBouttonFaux = Colors.grey;
13     Question question = QuestionRepository().getQuestion();
14
15     void changerQuestion(){
16         question = QuestionRepository().getNextQuestion(nbQuestionsPosees);
17         couleurBouttonVrai = Colors.grey;
18         couleurBouttonFaux = Colors.grey;
19         nbQuestionsPosees--;
20         aRepondu = false;
21         notifyListeners();
22     }
23
24     void affecterCouleursBouttons(Color couleurVrai, Color couleurFaux){
25         couleurBouttonVrai = couleurVrai;
26         couleurBouttonFaux = couleurFaux;
27         aRepondu = true;
28         notifyListeners();
29     }
30
31     void augmenterBonnesReponses() => nbBonnesReponses++;

```

FIGURE 2.2 – Classe Quizz qui étend la classe ChangeNotifier

Ensuite j'ai utilisé un widget `MultiProviders` pour créer et donner accès à mon instance de `ChangeNotifier` à tous les fils de mon arbre, ceci en mettant `MultiProviders` parent du widget contenant toute mon application : `MyApp()`.

2.3 Utilisation de `Consumer`, `Provider.of`, `context.watch` et `context.read()`

2.3.1 Consumer

Afin d'être averti en cas de changement d'état, j'ai wrappé le widget `Image` d'un widget `Consumer<Quizz>` comme on peut le voir dans la figure 3.2. Lorsque dans la classe `Quizz` une des fonctions fait appel à `notifyListeners()`, le widget `Image` se reconstruit. J'ai également utilisé `Consumer` pour le widget `Card` contenant le texte des questions, ainsi ils se reconstruisent tous deux lors d'un changement. Ce changement a lieu lorsqu'on fait appel à la méthode qui récupère la question suivante dans la classe `QuestionRepository`.

2.3.2 Provider.of

J'ai choisi d'utiliser à certains endroits ceci : `'Provider.of<Quizz>(context, listen : false)'`,

```

return Container(
  decoration: BoxDecoration(
    border: Border.all(width: 5, color: Colors.black),
  ), // BoxDecoration
  child: Consumer<Quizz>(
    builder: (context, quizz, child) {
      return Image.asset(
        '${quizz.question.imagePath}',
        height: 180,
        width: 450,
        fit: BoxFit.fitWidth,
      ); // Image.asset
    },
  )); // Consumer, Container

```

FIGURE 2.3 – Utilisation du widget Consumer

afin de ne pas reconstruire le widget et de seulement récupérer la valeur dans la classe Quizz pour l'utiliser dans mon code. Comme on peut le voir par exemple dans la figure 3.3, j'ai utilisé Provider.of pour savoir si l'utilisateur a répondu. Cette valeur est actualisée par une fonction de Quizz, lorsque l'utilisateur appuie sur le bouton 'vrai' ou 'faux'. J'ai utilisé Provider a plusieurs reprises lorsque l'UI ne devait pas être actualisé.

```

onPressed: () {
  if (!Provider.of<Quizz>(context, listen: false).aRepondu) {
    if (_checkAnswer(true, context)) {
      context
        .read<Quizz>()
        .affecterCouleursBouttons(Colors.green, Colors.grey);
    }
  }
}

```

FIGURE 2.4 – Utilisation du widget Provider

2.3.3 Context.watch

Pour essayer les différentes possibilités, j'ai utilisé ceci : 'context.watch<Quizz>().couleurBouttonFaux' pour écouter les changements dans la classe Quizz. Comme on peut le voir à la figure 3.4 j'ai utilisé watch pour écouter un changement sur la variable donnant la couleur du bouton vrai, et j'ai fait de même pour le bouton false.

```

TextButton(
  child: Text('Vrai'),
  style: TextButton.styleFrom(
    foregroundColor: Colors.white,
    backgroundColor: context.watch<Quizz>().couleurBouttonVrai,
  ),
)

```

FIGURE 2.5 – Utilisation de la méthode watch

2.3.4 La fonction context.read()

Dans les fonctions associées à la propriété 'onPressed' j'ai remplacé les appels à la fonction 'setState()' par ceci : 'context.read<Quizz>().affecterCouleursBouttons(Colors.green, Colors.grey);'. Cela permet d'invoquer des méthodes de la classe Quizz, par exemple la méthode affecterCouleursBouttons() ici mais j'ai également utilisé ceci pour appeler la méthode changerQuestion().

```
onPressed: () {  
  if (!Provider.of<Quizz>(context, listen: false).aRepondu) {  
    if (_checkAnswer(false, context)) {  
      context  
        .read<Quizz>()  
        .affecterCouleursBouttons(Colors.grey, Colors.green);  
    }  
  }  
}
```

FIGURE 2.6 – Utilisation de la méthode read()

2.3.5 La récupération de valeurs depuis le widget de la page de résultats

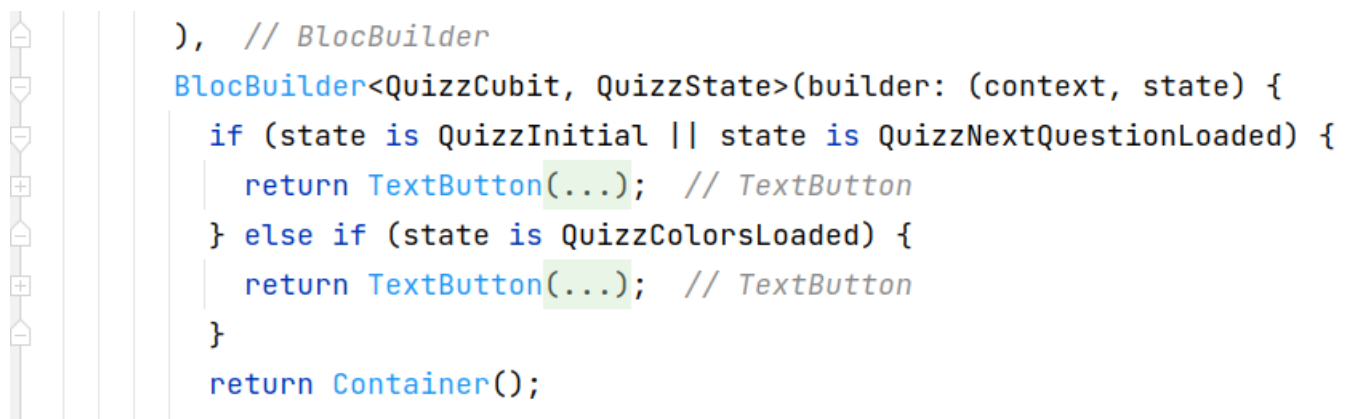
Grâce à l'utilisation des provider, la récupération du nombre de questions et du nombre de bonnes réponses depuis la classe ResultatsPage est grande facilité. En effet, la classe ChangeNotifierProvider est accessible depuis tout l'arbre et donc depuis le dernier widget ! On récupère.

3 Utilisation des BLoCs à la place de la fonction `setState()`

Le projet associé à cet exercice s'appelle 'tp2-w-blocs-cubits'. Vous le trouverez sur le github, vu que c'est le même exercice que le TP1 je n'ai pas remis de screenshot de l'interface puisqu'elle est totalement identique !

Afin d'utiliser cette fois les cubits à la place des fonctions '`setState()`' ou des providers, j'ai créé un cubit appelé 'Quizz'. Nous verrons plus en détails les cubits dans la partie suivante sur la météo. J'ai défini des états pour mon cubit dont principalement : 'QuizzInitial', 'QuizzColorsLoaded', 'QuizzNextQuestionLoaded'. J'ai utilisé un BlocProvider à la racine de l'arbre pour fournir mon Cubit à tout l'arbre. J'ai ensuite utilisé des BlocBuilder pour faire reconstruire mon UI en fonction des états du cubit.

Voici ci dessous un exemple d'utilisation du BlocBuilder avec la gestion des états. J'ai utilisé ceci pour récupérer l'image, le texte de la question, la couleur des boutons 'vrai' et 'faux'.



```
), // BlocBuilder\n\nBlocBuilder<QuizzCubit, QuizzState>(builder: (context, state) {\n  if (state is QuizzInitial || state is QuizzNextQuestionLoaded) {\n    return TextButton(...); // TextButton\n  } else if (state is QuizzColorsLoaded) {\n    return TextButton(...); // TextButton\n  }\n  return Container();\n});
```

FIGURE 3.1 – Utilisation d'un BlocBuilder pour reconstruire un TextBouton.

Lorsque l'application se lance, le cubit est dans l'état initial donc je mets les boutons en gris et c'est aussi le cas lorsque l'on change de question donc j'ai associé les deux états au même corps de la condition if (voir figure 3.1). Lorsque l'utilisateur répond à une question, depuis l'un de ces deux états, on fait appel à une méthode du cubit : 'BlocProvider.of<QuizzCubit>(context).changeButtonsColor()' à laquelle on fournit en argument les couleurs des boutons en fonction de la réponse de l'utilisateur et de la vraie réponse. Une fois appelée, cette méthode a presque le même corps que lors des deux autres exercices sauf que là elle émet un nouveau 'state' pour notre cubit, il s'agit du 'QuizzColorsLoaded' state.

Une fois que cet état est émis, le BlocBuilder va reconstruire le TextButton associé, celui qui est dans la condition 'else if(state is QuizzColorsLoaded)' et lui associer la bonne couleur.

Après avoir fait l'exercice suivant sur la météo, je me rends compte que les états n'avait pas besoin d'être si découpés ni d'être autant utilisé dans les conditions car dans cet exercice on ne met pas d'écran de chargement, d'écran d'erreur... mais cela m'a permis de comprendre le fonctionnement avant d'attaquer l'exercice le plus dur de ce tp.

4 Création d'une application de consultation de météo

Le projet associé à cet exercice s'appelle 'tp2-appMeteo'. Vous le trouverez sur le github donné avec une vidéo courte nommée 'tp2-demo.mp4' qui montre le résultat de mon application dans l'émulateur.

4.1 Structure du projet

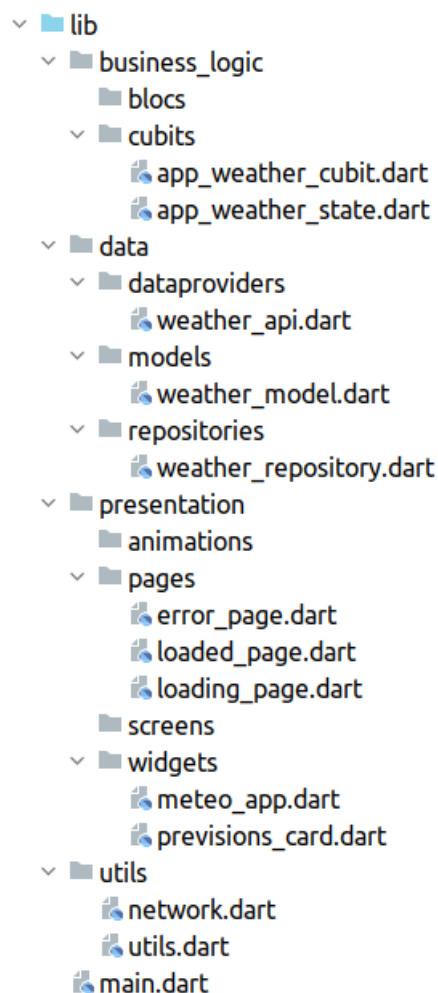


FIGURE 4.1 – Packages et classes permettant de structurer le projet par responsabilités

Voici la structure de ce projet, j'ai pu comprendre l'utilité de chacun des dossiers étant donné que j'ai créé différentes classes avec différentes utilités comme nous allons le voir.

4.2 Cubit et States

Afin de gérer les différents états de l'application j'ai créé un cubit 'AppWeatherCubit' et une fonction `getWeather()` qui en fonction de la ville donnée dans la TextField proposé à l'utilisateur, utilise le `WeatherRepository` pour émettre une requête vers l'API. Nous verrons ceci en détails dans la prochaine section.

```
7
8 class AppWeatherCubit extends Cubit<AppWeatherState> {
9     final WeatherRepository weatherRepository = WeatherRepository();
10     Future<WeatherForecastModel>? weatherModel;
11
12     AppWeatherCubit() : super(AppWeatherInitial());
13
14     void getWeather(String cityName) async {
15
16         try {
17             weatherModel = weatherRepository.getWeatherForecast(cityName: cityName);
18             emit(AppWeatherLoaded());
19             print('Ville trouvée.');
```

FIGURE 4.2 – Cubit AppWeatherCubit

J'ai défini les 'states' suivants qui sont associés chacun à une page, représentée par une classe/widget

```
7
8 class AppWeatherInitial extends AppWeatherState {}
9
10 class AppWeatherLoading extends AppWeatherState {}
11
12 class AppWeatherLoaded extends AppWeatherState {}
13
14 class AppWeatherError extends AppWeatherState {}
```

FIGURE 4.3 – Les différents state possible du cubit 'AppWeatherCubit'

4.3 DataProvider, Model et Repositories

Afin de mieux diviser et mieux comprendre mon application j'ai utilisé la classe `Network` et je l'ai découpé en deux classes : `WeatherAPI` (un `dataproducer`) et `WeatherRepository` (un `repository`). Dans la classe `WeatherAPI` je récupère avec une requête `http`, les données exposées par l'API puis je retourne ces données décodées avec la méthode `jsonDecode`. Dans la classe `WeatherRepository` je fais appel à la méthode de l'API retournant un '`Future<WeatherForecastModel>`' et avec un `await` je propage encore un `Future<WeatherForecastModel>` permettant d'attendre que les données soient récupérées et

elles le seront forcément (données ou erreur). Enfin grâce à une conversion d'une réponse de JSON de l'API vers du Dart j'ai pu récupérer et créer une classe 'WeatherForecastModel' qui représente le modèle des données reçues afin de correctement les représenter et les utiliser par la suite.

4.4 Le widget principal : MeteoApp

Le widget principal est celui-ci, il s'appelle 'MeteoApp' et contient dans son body, une Column contenant un TextField et un BlocBuilder comment on peut le voir sur la figure suivante :

```
18 class _MeteoAppState extends State<MeteoApp> {
19   @override
20   Widget build(BuildContext context) {
21     return Scaffold(
22       backgroundColor: Colors.white,
23       appBar: AppBar(
24         title: Text('Météo App'),
25         centerTitle: true,
26         backgroundColor: Colors.grey[900],
27       ), // AppBar
28       body: ListView(
29         children: [
30           _getTextField(),
31           _getBlocBuilder(),
32         ],
33       ), // ListView
34     ); // Scaffold
```

FIGURE 4.4 – Widget StatefulWidget MeteoApp

Le BlocBuilder permet de gérer les différents états de notre cubit. Une première condition teste s'il on est dans l'état initial ou si les données ont déjà été récupérées après une recherche. Tout d'abord vous pouvez voir ce à quoi ressemble l'écran lors du lancement de l'application et donc de l'état initial du cubit sur la figure 4.5.

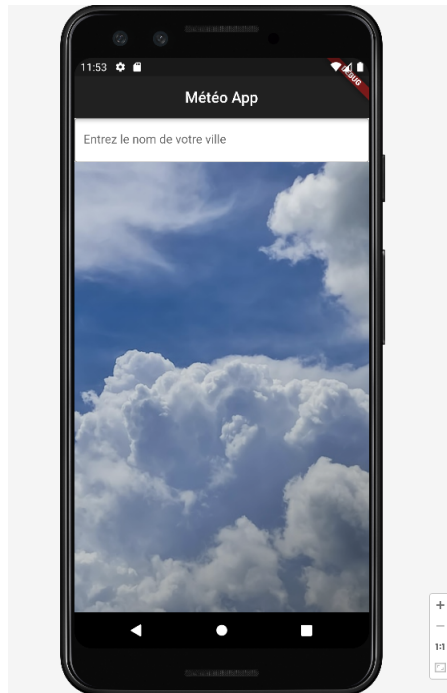


FIGURE 4.5 – Etat Initial de l'application

Si on tape une recherche dans la barre et que l'on fait entrée alors on appelle la fonction 'BlocProvider.of<AppWeatherCubit>(context).getWeather(value);' qui va émettre un état 'AppWeatherLoaded' et le traitement va commencer, voici le code associé :

```

7
9 BlocBuilder<AppWeatherCubit, AppWeatherState> _getBlocBuilder() {
1   return BlocBuilder<AppWeatherCubit, AppWeatherState>(
2     builder: (context, state) {
3       if (state is AppWeatherLoaded) {
4         return FutureBuilder(
5           future: BlocProvider.of<AppWeatherCubit>(context).weatherModel,
6           builder: (context, snapshot) {
7             if (snapshot.connectionState == ConnectionState.done) {
8               if (snapshot.hasError) {
9                 return ErrorPage(errorMessage: snapshot.error.toString());
10              }
11              return LoadedPage(model: snapshot.data);
12            } else if (snapshot.connectionState == ConnectionState.waiting) {
13              return LoadingPage();
14            }
15            return Text('');
16          },
17        ); // FutureBuilder

```

FIGURE 4.6 – Code du BlocBuilder quand une recherche a été effectuée

Dans cette condition du if j'ai utilisé un FutureBuilder qui reconstruira les données lorsqu'ils les aura obtenues. Je réalise quelques tests pour lancer la page `ErrorPage` si jamais l'API retourne une erreur, nous verrons à quoi cela ressemble dans la dernière section. Si la connexion à l'API est en mode 'waiting' je lance une page `Loaded`. Je me rends compte ici que je n'ai pas utilisé les différents états du cubit que j'ai créés pour faire cette gestion d'états mais j'aurai dû...

Cependant, si la connexion est bonne on retourne une page `LoadedPage()` à laquelle on fournit le modèle récupéré en tant que future par l'API. Nous allons voir ce widget en détails.

4.5 LoadedPage() et Previsions()

Lorsque la recherche a été effectuée, qu'aucune erreur ne s'est produite et que le chargement des données est terminé, le BlocBuilder, va reconstruire l'ensemble des widgets enfants. Le widget appelé est LoadedPage(), il est divisé en trois parties, une partie sur la météo courante, une partie texte et une partie sur les prévisions pour 3 jours (selon l'heure où on exécute l'application, la première prévision est le lendemain ou le jour même). Voici le rendu finale pour la ville de 'Paris'



FIGURE 4.7 – Résultats de la recherche 'Paris'



FIGURE 4.8 – Résultats de la recherche 'Paris' scroll des prévisions

Pour construire cette interface graphique j'ai utilisé plusieurs widgets, déjà détaillés dans le TP1 et d'autres nouveaux comme le widget 'SingleChildScrollView' qui permet de faire en sorte que la partie des prévisions soit scrollable horizontalement. Afin de remplir tous les champs en fonction de la ville demandée, j'ai créé des fonctions génériques utilisables par le widget de la météo actuelle et les widgets des prévisions. Chacune prend en paramètre un entier qui correspond à l'élément que l'on veut récupérer dans le JSON renvoyé par l'API. Pour la météo actuelle j'utilise l'élément 0 et comme c'est découpé par tranches de 3h, j'utilise le 8e, 16e, 24e éléments pour avoir les prévisions pour les prochains jours. La figure 4.9 montre un code type qui permet de récupérer dans l'instance du WeatherForecastModel récupéré via l'API, les valeurs dont on a besoin.

```

String _getHumidity(int jour) {
    //print(modele?.liste?.elementAt(0).dtTxt);
    int humidity = modele?.liste?.elementAt(jour).main?.humidity ?? 0;
    String humidityy = humidity.toStringAsFixed(1);
    return humidityy + "%";
}

```

FIGURE 4.9 – Code qui permet de récupérer dans le modèle l'humidité pour un certain jour, donné en argument de la fonction

Enfin pour créer la partie scrollable des prévisions j'ai utilisé le widget SingleChildScrollView dans lequel j'ai instancié 4 fois le widget Previsions auquel je donne l'instance du modèle et le jour à récupérer dans le JSON

```

l63 SingleChildScrollView _previsionCards() {
l64     return SingleChildScrollView(
l65         scrollDirection: Axis.horizontal,
l66         child: Row(
l67             children: [
l68                 Previsions(journee: 0, modele: modele),
l69                 Previsions(journee: 8, modele: modele),
l70                 Previsions(journee: 16, modele: modele),
l71                 Previsions(journee: 24, modele: modele),
l72             ],
l73         ), // Row
l74     ); // SingleChildScrollView
l75 }
l76

```

FIGURE 4.10 – Partie scrollable contenant 4 widgets Previsions

4.6 ErrorPage() et LoadingPage()

Afin de respecter les bonnes pratiques j'ai défini deux pages (widgets) différents, un premier pour afficher sur l'écran le chargement de la récupération des données et un deuxième pour afficher une erreur si jamais il y en a une. Voici ce qu'il s'affiche dans ces cas là :

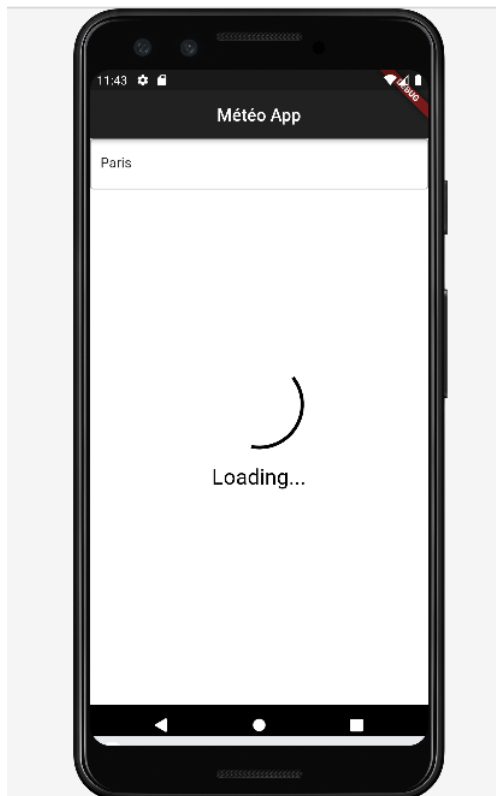


FIGURE 4.11 – Ecran de chargement (visible moins d'une seconde) qui montre que l'on essaie d'accéder aux données

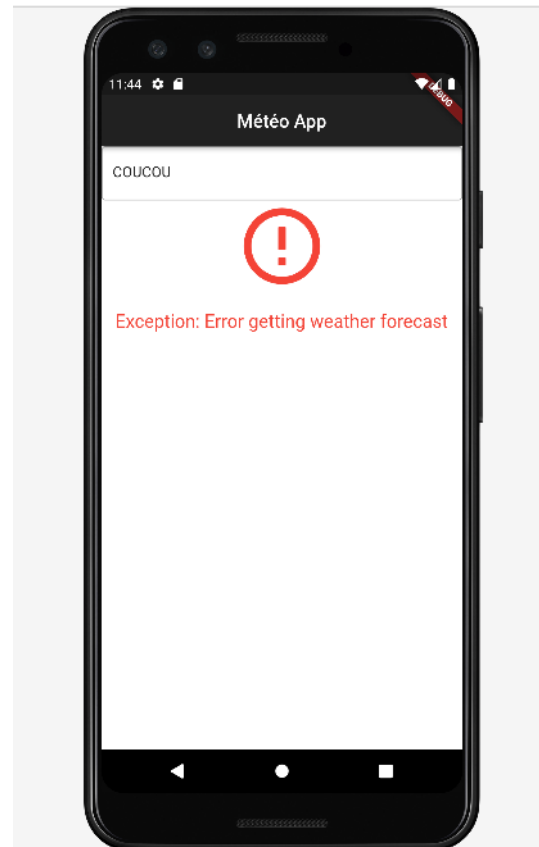


FIGURE 4.12 – Ecran d'erreur lorsque l'API ne trouve pas une ville

Petit détail, j'ai en fonction des jours, une erreur "int is not a subtype of double?" ou l'inverse qui s'affiche pour certaines villes, hier c'était Paris aujourd'hui c'est Montpellier. J'ai essayé de modifier le modèle mais l'API ne semble par toujours renvoyer le même type donc il se peut que cette erreur s'affiche lors de la recherche d'une certaine ville.