Flutter

Accéder à une API Web publique

Web API

- De nombreuses applications, surtout mobiles, accèdent à des API Web
- On va voir ici comment faire des requêtes HTTP en Flutter sur une API Web et à gérer correctement les réponses
  - on va utiliser une API Web publique et gratuite
  - vous apprendrez plus tard à créer votre propre API Web

Au menu

- Récupération de donnés (*fetching*)
- Classes « *Model* » pour stocker les données
- *Parsing JSON* : conversion de données JSON en objets Dart
- *FutureBuilder* pour gérer les données reçues et mettre à jour l'UI
- Gestion basique des erreurs (erreurs réseau, réponses invalides…)

L'API Web Zenquotes

- https://zenquotes.io
- API publique qui fournit des citations aléatoires
  - https://zenquotes.io/api/random : exemple de réponse JSON
- Trois champs : *q* (citation), *a* (auteur), *h* (citation en HTML)

Exemple JSON

```
[
  {
    "q": "Often in the real world, it's not the smart that get ahead, but the bold.",
    "a": "Robert Kiyosaki",
    "h": "<blockquote>&ldquo;Often in the real world, it's not the smart that get
ahead, but the bold.&rdquo; &mdash; <footer>Robert Kiyosaki</footer></blockquote>"
  }
]
```

Rappels JSON

- Objets entre {accolades}
  - chaque objet contient des paires *propriété*: *valeur* séparées par des virgules
  - les valeurs peuvent être des chaînes, des nombres, des booléens, des tableaux ou des objets
  - ⇒ les objets peuvent donc être imbriqués
- Tableaux (*arrays*) entre [crochets] : liste d'objets
  - l'exemple précédent est donc un tableau d'un seul objet contenant trois propriétés dont les valeurs sont des strings

Avertissement sur l'utilisation de l'API

- **Zenquotes.io** limite le nombre de requêtes à 5 toutes les 30 secondes
- Si on accède tous à l'API depuis l'IP du lycée, on risque de dépasser cette limite fréquemment
- Il est donc probable que Zenquotes nous renvoie régulièrement le JSON suivant ; ce n'est pas bien grave, on aura quand même une citation et un auteur à afficher !

```
{
  "q": "Too many requests. Obtain an auth key for unlimited access.",
  "a": "zenquotes.io",
  "h": "Too many requests. Obtain an auth key for unlimited access @ zenquotes.io"
}
```

Le package HTTP

- Le package *http* fournit des fonctions simples pour faire des requêtes HTTP et gérer les réponses
- https://pub.dev/packages/http
- À installer

Écran *quote_screen.dart*

- Ce nouvel écran est dédié à l'affichage d'une citation aléatoire récupérée depuis l'API Web
- Dans un premier temps, on va juste se contenter de tester l'API
  - requête et récupération de la citation JSON
  - affichage du résultat **sur la console**
  - on s'occupera de l'UI plus tard

quote_screen.dart - Test API

```
import 'package:flutter/material.dart';
```

```dart
import 'package:http/http.dart' as http;

class QuoteScreen extends StatefulWidget {
  const QuoteScreen({super.key});

  @override
  State<QuoteScreen> createState() => _QuoteScreenState();
}

class _QuoteScreenState extends State<QuoteScreen> {
  @override
  void initState() {
    super.initState();
    fetchQuote();  // récupération de la citation
  }

  @override
  Widget build(BuildContext context) {
    return const Placeholder(); // pas d'UI pour l'instant
  }

  // Méthode pour la récupération de la citation
  // Premier jet qui affiche la citation sur la console
  Future<void> fetchQuote() async {
    final url = Uri.parse('https://zenquotes.io/api/random');
    final response = await http.get(url);
    if (response.statusCode == 200) { // 200 = OK
      final quote = response.body;
      print(quote);
    } else {
      print('Échec de récupération de la citation');
    }
  }
}
```

Opérations réseaux asynchrones

- À noter : **toutes les méthodes du package *http* sont asynchrones**
  - l'interaction avec le réseau est naturellement asynchrone
  - en effet, les opérations réseaux prennent un temps indéfini pour s'exécuter complètement
  - il est inadmissible de bloquer complètement l'UI à chaque fois qu'une requête réseau est effectuée
- Il faudra donc utiliser *async/await* et garder en tête que les méthodes retournent des *Future*

De l'API à l'UI

- Plusieurs étapes sont nécessaires pour finalement arriver à la présentation des données sur l'UI

- récupération en JSON via l'API (déjà fait)
- conversion du JSON en objet Dart (classes *Model*)
- utilisation des modèles pour affichage sur l'UI

Classes *Model*

Principe : JSON ⇒ objet Dart

Classes *Model*

- Le format JSON est très souple et pratique pour le stockage et l'échange de données
  - en revanche, il n'est pas pratique pour manipuler ces données dans le code
- les classes *Model* sont des classes qui représentent les données (ici récupérées depuis l'API)
  - un objet modèle va stocker les données (par exemple ici, d'une citation) de manière structurée et les rendre facilement et directement accessibles depuis le code
- On va créer une classe *Quote* pour stocker les citations
  - ⇒ chaque objet de type *Quote* représentera donc une citation

Classe *Quote*

- Créons la classe *Quote* dans un nouveau répertoire *models*
  - citation : propriété *quote* (`q` dans le JSON)
  - auteur : propriété *author* (`a` dans le JSON)
- Il est fréquent de ne pas forcément utiliser toutes les données fournies par une API publique
  - ici, on n'aura pas l'usage de la propriété JSON *h* (citation au format HTML)

```
class Quote {
  final String text;
  final String author;

  Quote(this.text, this.author);
}
```

Conversion JSON ⇒ objet Dart

- Maintenant on a de quoi stocker les données JSON en mémoire
- Mais comment convertir les données JSON en objets Dart ?
- ⇒ Bibliothèque *dart:convert* (fournie avec Dart)
  - `import 'dart:convert';`
  - méthode *decode* : JSON ⇒ Dart

    - méthode *encode* : Dart ⇒ JSON

Nouveau constructeur *Quote.fromJSON*

- Ajoutons un constructeur nommé *fromJSON* à la classe *Quote*
    - capable de prendre un objet JSON provenant de l'API
    - de le convertir en objet *Quote*
- On se souvient que l'API renvoie un tableau d'un seul objet
    - ⇒ il faudra extraire cet objet unique avant de le traiter

```
Quote.fromJSON(String quoteJSON) {
  final List jsonList = json.decode(quoteJSON); // conversion JSON => Dart
  final quoteMap = jsonList.first; // unique objet de la liste
  text = quoteMap['q'];   // récup du texte par la clé 'q'
  author = quoteMap['a']; // récup de l'auteur par la clé 'a'
}
```

fetchQuote - utiliser le constructeur *fromJSON*

- On va maintenant mettre à jour la méthode *fetchQuote* pour convertir le JSON en objet *Quote* en utilisant ce nouveau constructeur

```
Future fetchQuote() async {
  final url = Uri.parse('https://zenquotes.io/api/random');
  final response = await http.get(url);
  if (response.statusCode == 200) {  // 200 = OK
    final quoteJSON = response.body; // récup du body de la réponse
    final quote = Quote.fromJSON(quoteJSON); //
    print('Citation : ${quote.text} ; Auteur : ${quote.author}');
  } else {
    print('Échec de récupération de la citation');
  }
}
```

fetchQuote - renvoyer le résultat

- Maintenant qu'on a correctement récupéré un objt modèle *Quote*, on va renvoyer ce résultat pour pouvoir l'utiliser plus tard
    - on se débarrasse du *print* de débogage
    - on renvoie l'objet *Quote* à la place
    - on n'oublie pas de changer le type de retour

```
Future<Quote> fetchQuote() async {
  final url = Uri.parse('https://zenquotes.io/api/random');
  final response = await http.get(url);
  if (response.statusCode == 200) {
    final quoteJSON = response.body;
    Quote quote = Quote.fromJSON(quoteJSON);
    return quote;  // renvoi de la citation
  } else {
    // En cas d'erreur, on utilise le constructeur non-nommé
    // pour renvoyer l'erreur dans le texte de la citation
    return Quote('Erreur de récupération', '');
  }
}
```

Écran de la citation

- On va enfin s'occuper d'afficher la citation sur l'écran dédié
- Citation centrée sur l'écran (avec l'auteur juste en dessous), horizontalement et verticalement
  - citation en italique
  - auteur en gras (*bold*)
  - léger espace entre les deux
- Padding sur l'élément englobant pour éviter les bords de l'écran

Gestion de la citation

- Ne pas oublier de récupérer la *Quote* depuis *initState*
  - et de la stocker dans une variable d'instance
  - pour pouvoir utiliser ses 2 propriétés dans *build*


Citation du jour
Mistakes are always forgivable, if one has the courage to admit them.
Bruce Lee

```
class _QuoteScreenState extends State<QuoteScreen> {
  Quote quote = Quote('', '');

  @override
  void initState() {
    super.initState();
    // utilisation de then
    // permet d'exécuter une action dès que la Quote est récupérée
    fetchQuote().then((value) {
      // on n'oublie pas ssetState car on modifie l'UI
      setState(() {
        quote = value;
      });
    });
  }

  // ...

}
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Citation du jour')),
    body: Padding(
      padding: const EdgeInsets.all(20.0),
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              quote.text,
              style: const TextStyle(
                fontStyle: FontStyle.italic,
                fontSize: 24,
              ),
              textAlign: TextAlign.center,
            ),
            const SizedBox(height: 16),
            Text(
              quote.author,
              style: const TextStyle(
                fontWeight: FontWeight.bold,
                fontSize: 18,
              ),
              textAlign: TextAlign.center,
            ),
          ],
        ),
```

```
      ),
    ),
  );
}
```

mainAxisAlignment

- Ici un *widget Center* ne suffirait pas à centrer la *Column* verticalement
  - en fait la *Column* est bien centrée (car elle prend de toute façon tout l'espace disponible)
  - mais ses enfants en le sont pas (ils sont relatifs à la *Column*, et pas au *Center*)
- On utilise la propriété *mainAxisAlignment* de *Column* (aussi sur *Row*) :
  - elle permet de définir comment les enfants de la *Column* vont être alignés verticalement
  - ici, *MainAxisAlignment.center* centre correctement les enfants *à l'intérieur* de la *Column*

Bug - Paramètres manquants

- Notre application a actuellement un bug qui ne concerne que son tout premier lancement après installation (ou après suppression des données de l'application)
- Lorsque l'app va chercher les paramètres pour charger l'écran d'accueil, elle ne trouvera rien dans les *SharedPreferences*
- Ce bug n'a normalement pas été rencontré dans le développement
  - en effet, on a d'abord dev l'écran de paramètres, et donc enregistré des paramètres, avant de les utiliser !
  - on n'a jamais eu à expérimenter l'app, **avec les fonctionnalités de paramètres**, lors d'un premier lancement
- Cela montre l'importance de tester son app dans toutes les situations possibles


Accueil
Bienvenue, !
Paramètres

Correction du bug

- Un moyen simple est de s'assurer, au moment de l'utilisation des variables lors de l'affichage (*build*), que les variables ne sont pas vides
  - on peut ici utiliser l'opérateur ternaire pour afficher un texte par défaut si la variable est vide
- L'implémentation ci-après utilise de nouveau l'interpolation avec accolades (`${⋯}`) pour effectuer un traitement légèrement plus complexe pour calculer la valeur à afficher à cet endroit

```
// ...
child: Image.asset(
  backgroundImage.isEmpty        // test□: string vide□?
      ? 'assets/images/sea.jpg'  // valeur par défaut
      : backgroundImage,         // valeur normale
  fit: BoxFit.cover,
),

// ...

child: Text(
  'Bienvenue, ${username.isEmpty ? 'utilisateur' : username} !',
  ),
// ...
```

Réimplémenter l'accès aux paramètres

- Actuellement, on a utilisé le bouton d'accès aux paramètres pour tester notre *QuoteScreen*
- On pourrait ajouter un second bouton pour pouvoir naviguer sur les deux écrans
- Mais on va placer les paramètres dans la *appBar* (pattern classique dans les apps mobiles)
  - *appBar* possède une propriété *actions* sur laquelle on peut placer des *IconButton*
  - ceux-ci apparaîtront à droite de la *appBar*
  - utilisé traditionnellement pour la navigation, la recherche…

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Accueil'),
      actions: [
        IconButton(
          onPressed: () async {
            goToSettings(context);
          },
```

```
          icon: const Icon(Icons.settings),
        ),
      ],
    ),
    // ...
  )
}

Future<void> goToSettings(BuildContext context) async {
  await Navigator.of(context).push(
    MaterialPageRoute(builder: (context) => const SettingsScreen()),
  );
  loadSettings(); // maj de l'UI après retour
}
```

Extraction de méthode pour la navigation

- On a extrait la méthode *goToSettings* pour la navigation vers l'écran des paramètres
- De la même manière, on va extraire la méthode *goToQuote* pour naviguer vers l'écran de citation
- Pour extraire une méthode dans VS Code :
  - sélectionner le code à extraire
  - clic droit > *Refactor* > *Extract Method...* (ou `Ctrl+.` > *Extract Method...*)

Extraction de *goToQuote*

```
// ...
child: ElevatedButton(
  onPressed: () {
    goToQuote(context);
  },
  child: const Text('Citation du jour'),
),

// ...

Future<void> goToQuote(BuildContext context) async {
  await Navigator.of(context).push(
    MaterialPageRoute(builder: (context) => const QuoteScreen()),
  );
}
```

Refresh de la citation

- Actuellement, pour avoir une nouvelle citation, on doit sortir de l'écran *QuoteScreen* et y

revenir pour forcer une nouvelle requête à l'API

- On souhaite faire en sorte de pouvoir récupérer une nouvelle citation tout en restant dans l'écran *QuoteScreen*
- Solution simple et pratique : un bouton permettant de rafraîchir l'UI
  - de nouveau ici, on va choisir de placer cette fonctionnalité dans la *appBar*

```
// appBar de QuoteScreen
appBar: AppBar(
  title: const Text('Citation du jour'),
  actions: [
    IconButton(
      onPressed: () {
        // on vide la citation (force l'affichage du spinner
        //     en attendant la nouvelle citation)
        setState(() {
          quote.text = '';
          quote.author = '';
        });
        fetchQuote().then((value) {
          setState(() {
            quote = value;
          });
        });
      },
      icon: const Icon(Icons.refresh),
    ),
  ],
),
```

FutureBuilder - Principe

- Actuellement, dans le *initState* de *QuoteScreen* :
  - on appelle *fetchQuote*
  - qui fait une opération asynchrone (requête HTTP)
  - on récupère un *Future*
  - on appelle *setState* pour mettre à jour l'UI en fonction de ce résultat
- Ça fonctionne, mais c'est un *pattern* tellement utilisé que Flutter propose un *widget* dédié : *FutureBuilder*
  - cela facilite également la gestion du *feedback* utilisateur pendant le traitement (actuellement on fait ça « à la main »)

FutureBuilder - Fonctionnement

- supprimer le code de *initState*

- englober le *widget* principal du *body* du *Scaffold* dans un *FutureBuilder* (utiliser les *code actions*)
- ajouter la propriété *future* et y appeler *fetchQuote*
- ajouter le *snapshot* comme 2ème paramètre du *builder*
  - le *snapshot* contient les données renvoyées par le *Future* une fois terminé et ainsi que l'état du *Future* (propriété *connectionState*)

FutureBuilder - *Snapshot*

- L'intérêt du *snapshot* est de pouvoir gérer l'UI en fonction de l'état du *Future*
- Dans le code, on va tester la propriété *connectionState* du *snapshot* (avec des `if`, tout simplement)
- en fonction de l'état du *Future*, on va retourner un *widget* différent
  - *spinner* pour montrer qu'on est en attente de la réponse
  - interface complète une fois la réponse reçue
  - interface d'erreur si la requête a échoué

FutureBuilder - les différents états

- Le *snapshot* peut être dans 4 états différents :
  - *ConnectionState.none* : pas encore de *Future* associé
  - *ConnectionState.waitin* : le *Future* est en cours d'exécution, mais n'est pas encore terminé (l'UX voudrait qu'on indique à l'utilisateur que le traitement est en cours)
  - (*ConnectionState.active* : peu utilisé avec *FutureBuilder*)
  - *ConnectionState.done* : le *Future* a terminé son exécution (succès ou erreur)
    - on peut utiliser les données contenues dans *snapshot.data* (ou gérer l'erreur)
- Gestion des erreurs :
  - propriété *snapshot.hasError* à *true* si une erreur est survenue
  - propriété *snapshot.error* contient l'erreur

FutureBuilder - Implémentation

- On n'a plus besoin du *initState* : le *FutureBuilder* va appeler lui-même *fetchQuote* via sa propriété *future*
- Dans le *builder*, on utilise une suite de `if/else` pour renvoyer l'UI correspondante à l'état actuel du *Future*
- Le bouton de rafraichissement n'a plus qu'à appeler *fetchQuote* pour rafraîchir la citation

```
class _QuoteScreenState extends State<QuoteScreen> {
  @override
  Widget build(BuildContext context) {
```

```
    return Scaffold(
      appBar: AppBar(
        title: const Text('Citation du jour'),
        actions: [
          IconButton(
            onPressed: () {
              setState(() {
                fetchQuote(); // cet appel suffit maintenant
              });
            },
            icon: const Icon(Icons.refresh),
          ),
        ],
      ),
      body: FutureBuilder(
        future: fetchQuote(),  // l'action gérée par le FutureBuilder
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return const Center(
              child: CircularProgressIndicator(),
            );
          } else if (snapshot.hasError) {
            return Center(
              child: Text('Erreur : ${snapshot.error}'),
            );
          } else {  // ici on sait que le Future est terminé
            Quote quote = snapshot.data as Quote;
            return Padding(
              // UI exactement identique à auparavant
            );
          }
        },
      ),
    );
  }
  // ...
}
```

En résumé (1)

- API Web et format JSON
- Package *http* pour les requêtes HTTP
  - `http.get` pour de simples requêtes GET
  - *status code* des réponses
- Classes *Model* pour stocker les données en Dart
- Conversion JSON ⇒ Dart

En résumé (2)

- *FutureBuilder* pour gérer les traitement asynchrones avec l'UI
- Gestion diverse des erreurs
- *CircularProgressIndicator* pour donner un *feedback* de traitement en cours à l'utilisateur
- *actions* dans la *appBar*