


TP6 Application mobile - HTTP

M. Tellene

Attention, ce TP est complexe et sert de point d'entrée pour voir projet de fin d'année. Le comprendre est primordial pour la réussite de ce dernier. Il y a à la fin du TP un point cours pour donner d'avantage d'explication sur les manipulations faites et les éléments mis en place lors du TP.

Lorsque vous verrez l'icône  c'est qu'il faudra faire des manipulations sur votre machine.






I/ Éléments nécessaires

Pour ce TP, vous allez avoir besoin des éléments suivants :

- Une machine virtuelle avec **Wamp** installé
- La base de données des Jeux Olympiques¹ sur **Wamp**
- Node.js installé sur votre machine
- Un téléphone pour émuler votre application

II/ Paramétrage du serveur Wamp

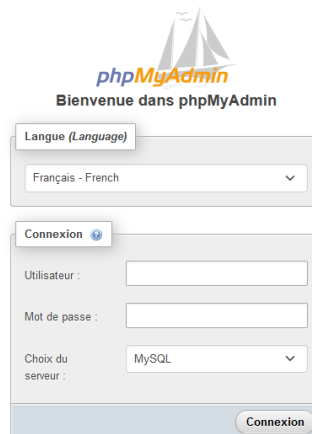
Pour commencer, il va falloir s'assurer que votre serveur **Wamp** est accessible depuis votre machine physique.

1.  Démarrer (si ce n'est pas fait) votre machine virtuelle
2.  Démarrer si ce n'est pas fait **Wamp**
3.  Récupérer l'adresse IP de votre machine virtuelle
4.  Sur votre machine physique, ouvrir un navigateur web
5.  Entrer l'URL suivante :

`http://XXX.XXX.XXX.XXX/phpmyadmin`

En remplaçant les XXX.XXX.XXX.XXX par l'adresse IP de votre machine virtuelle. Vous devriez arriver sur cette page :

1. Si vous ne l'avez pas elle se trouve sur moodle



6. Se connecter à *phpmyadmin*
7. Si ce n'a pas été fait! Créer un compte *phpmyadmin* avec les mêmes privilèges que le compte root (point cours : VII/1)

III/ Création d'un API REST et de l'application

Pour rappel, une API REST est une architecture logicielle qui définit un ensemble de contraintes pour la création de services Web. Ces services permettent de manipuler des ressources à l'aide d'opérations standardisées (GET, POST, PUT et DELETE).

Un serveur HTTP, quant à lui, est un logiciel qui écoute les requêtes HTTP et y répond en fonction de la configuration et de la logique applicative qu'il a en place. Il joue donc un rôle central dans la mise en oeuvre d'une API REST.

L'objectif de cette partie est de mettre en place un serveur HTTP (en utilisant le framework *Express.js*), que l'on va connecter avec la base de données Jeux Olympiques (plus tard, la base de données sera celle de votre projet). (point cours : VII/2)

⇒ Créer une nouvelle application Flutter

IV/ Mise en place du serveur HTTP

1. Dans le répertoire projet, dans un terminal, faire les commandes suivantes :



```
npm init -y //créer package.json qui stockera les dépendances
npm install express // installer Express.js
npm install mysql2 body-parser
```

2. Créer un fichier **index.js** à la racine de votre projet et y mettre le code du fichier **configServeur** (voir moodle).
3. Modifier le fichier **index.js** pour mettre les informations appropriées à votre configuration
4. Dans le terminal, taper la commande

```
node index.js
```

Si tout c'est bien passé, vous devriez avoir le message suivant :


```
Serveur API en écoute sur http://localhost:3000  
Connecté à la base de données MySQL
```

5.  Sur votre machine physique, ouvrir un navigateur web
6.  Taper dans la barre de recherche l'URL suivante :

```
http://localhost:3000/nation
```

Si tout c'est bien passé, vous devriez avoir les nations de la base de données sous format JSON. Vous venez de créer une API REST avec un serveur HTTP.

VI/ Création d'une nouvelle route




 Pour prendre en main le mécanisme de données vous allez créer une nouvelle route. En vous inspirant de la manière de récupérer les données de la table `nation` (route `/nation`), créer une route `/sport` récupérant les données de la table `sport`.

Les données devront être accessibles à l'URL

```
http://localhost:3000/sport
```

VI/ Application Flutter

Passons maintenant à l'application. Normalement vous avez créé le projet, si ce n'est pas fait, faites le.

1.  Récupérer le fichier **main** (voir moodle) et le mettre à la place du **main** créé par défaut
2.  N'oublier pas le **gradle.properties**
3.  Lancer l'application sur un téléphone

VI/1 Analyse de main.dart

- L'écran d'accueil est constitué d'un bouton *Les nations*. Il amène l'utilisateur vers l'écran `NationsPage`. Ce bouton est défini de la ligne 49 à 55.
- L'écran `NationsPage` est semblable à l'écran du TP précédent : une zone de saisie. Lorsque l'utilisateur saisit quelque chose une liste de résultat apparaît. Lorsque la zone de texte est validée la méthode `_searchNation` est lancée.
- La méthode `_searchNation` se connecte à l'adresse `http://10.0.2.2:3000` (point cours : VII/3) et accède à la page `nation`.
- La classe `Nation` définit comment un objet `Nation` doit se définir.

VI/2 Retour à l'application

Si vous avez un peu manipuler l'application, vous avez dû vous apercevoir que peu importe ce qui est tapé dans la barre de recherche, le résultat est le même. Cela s'explique par le fait que la route utilisée : `http://10.0.2.2:3000/nation` récupère tous les enregistrements de la table `nation` (cf. lignes 29 à 37 de `index.js`).

Nous allons donc créer une nouvelle route. Cette route sera équivalent à la requête :

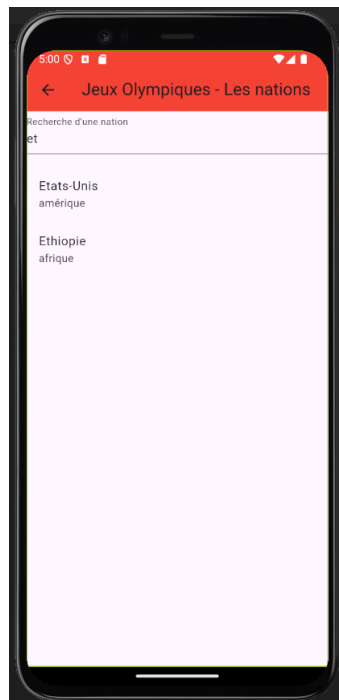
```
SELECT *  
FROM nation  
WHERE nom = ?;
```

Où le `?` sera ce qui est tapé par l'utilisateur.

📄 Dans le fichier `index.js`, ajouter le code suivant : (il faudra le mettre sous la déclaration de la route `/nation`)

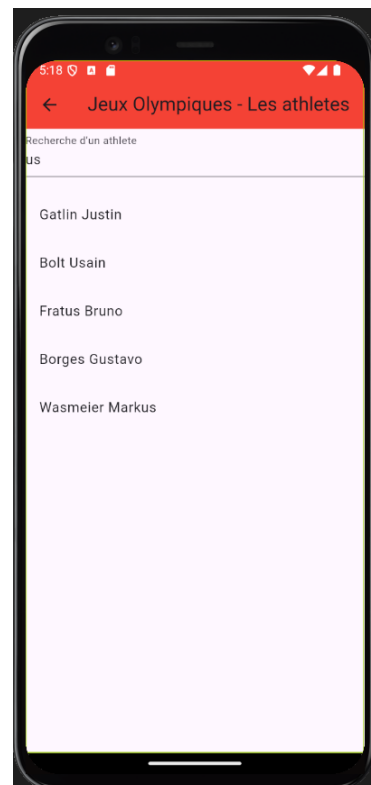
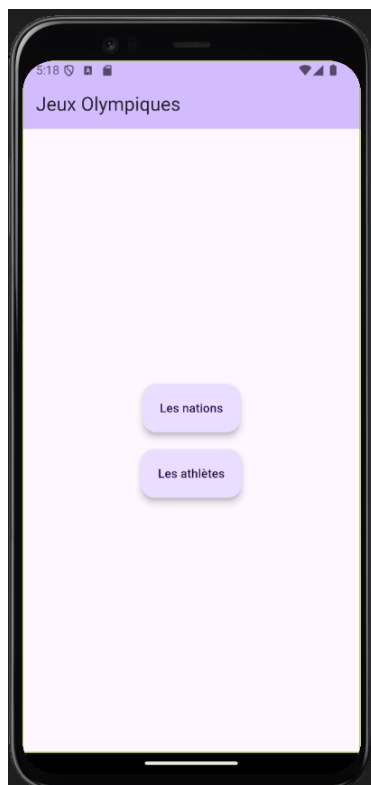
```
app.get('/nation/:nom', (req, res) => {  
  // Capture le paramètre 'nom' de l'URL  
  const nom = '%' + req.params.nom + '%';  
  
  // Crée la requête SQL avec un paramètre pour le nom  
  const sql = 'SELECT * FROM nation WHERE nom LIKE ?';  
  
  db.query(sql, [nom], (err, results) => {  
    if (err) {  
      return res.status(500).send(err);  
    }  
  
    if (results.length === 0) {  
      // Si aucune nation n'est trouvée, renvoyer une erreur 404  
      return res.status(404).json({ message: 'Nation not found' });  
    }  
  
    // Si des résultats sont trouvés, renvoyer les données  
    res.json(results);  
  });  
});
```

📄 Maintenant modifier la méthode `_searchNation` pour prendre en compte ce qui est tapé par l'utilisateur. Voici un exemple de résultat :




VI/3 Amélioration de l'application

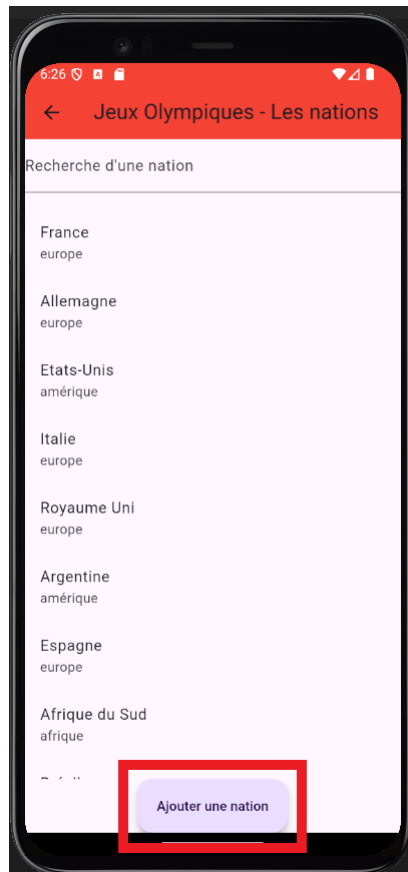
🖥 Maintenant votre but est d'améliorer l'application en permettant la recherche d'athlètes. L'idée sera de modifier l'application comme ceci :




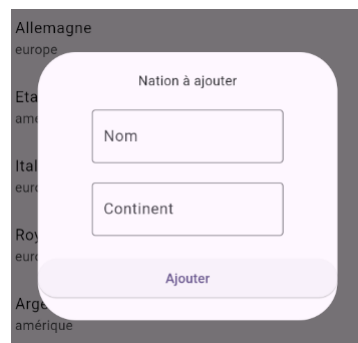
VI/4 Ajouter des données à la base


Retournons sur la page de recherche des nations. Nous allons permettre à l'utilisateur d'ajouter des données à la base. Voici la marche à suivre :

1.  Créer, sur l'écran des nations, un bouton (conseil utiliser un bouton `FloatingActionButton.extended`)



2.  Lorsque l'on appuie (`onPressed`) sur ce bouton, un fenêtre pop-up doit s'ouvrir (élément `showDialog`). La pop-up doit contenir deux zones de texte : une pour le nom et une pour le continent de la nation ajoutée.



3.  Lorsque l'on appuie (`onPressed`) sur le *Ajouter*, la méthode `ajouterNation` doit être exécutée. Une partie de la méthode est donnée en dessous (il va falloir la compléter) et doit être placée dans la classe `_NationSPageState`

```
Future<void> ajouterNation() async {
  //si les champs textes ne sont pas vides alors
  final url = Uri.parse('http://10.0.2.2:3000/nation');
  final headers = {'Content-Type': 'application/json'};

  final body = json.encode({
    'nom': ..... //récupérer la valeur du champs texte
    'continent': ..... //récupérer la valeur du champs texte
  });

  try {
    final response = await http.post(
      url,
      headers: headers,
      body: body
    );

    if (response.statusCode == 200) {
      print('Nation ajoutée !');
      print('Réponse: ${response.body}');
    } else {
      print('Échec de l\'ajout, erreur : ${response.statusCode}');
    }
  } catch (e) {
    print('Error: $e');
  }
}
```

VII/ Points cours

VII/1 Pourquoi créer un autre compte avec les droits root

Utiliser le compte root pour des services web n'est pas une bonne pratique. Par exemple PHP PDO bloque l'utilisation du compte root et sans mot de passe est généralement bloquée par les configurations modernes pour des raisons de sécurité. Voici une explication de ce phénomène et ce qui est recommandé à la place.

- **Meilleures pratiques en matière de sécurité :** l'utilisation de l'utilisateur root sans mot de passe constitue un risque important pour la sécurité. Elle permet un accès illimité à la base de données, ce qui la rend vulnérable aux scripts malveillants, aux attaques ou même à une mauvaise gestion accidentelle.
- **Configuration par défaut :** de nombreuses installations de *MySQL*, en particulier sur les serveurs de production, configurent le compte root de manière à ce qu'un mot de passe soit nécessaire pour l'authentification. Cette configuration s'aligne sur les pratiques sécurisées par défaut visant à protéger la base de données contre les accès non autorisés.
- **Principe du moindre privilège :** les applications ne doivent pas utiliser l'utilisateur root pour les opérations normales de la base de données. Elles devraient plutôt utiliser un compte d'utilisateur dédié disposant d'autorisations spécifiques adaptées aux besoins de l'application. Cela minimise l'impact des vulnérabilités potentielles.

VII/2 Pourquoi créer une API REST et un serveur HTTP

Se connecter directement à une base de données depuis une application Flutter (ou toute autre application côté client) est généralement considéré comme une mauvaise pratique pour plusieurs raisons importantes. Voici pourquoi l'utilisation d'une API REST en tant qu'intermédiaire est une bien meilleure approche :

1. Risques de sécurité :

- Exposition des informations d'identification de la base de données : si vous vous connectez directement à la base de données depuis votre application Flutter, vous risquez d'exposer des informations sensibles (comme le nom d'utilisateur et le mot de passe de la base de données) dans le code source de votre application. Avec de la rétro-ingénierie il est facile de récupérer ces informations.
- Accès non autorisé : l'accès direct à la base de données signifie que n'importe quel client (même malveillant) peut tenter d'interagir avec elle, ce qui peut entraîner une corruption des données, leur récupération ou leur suppression sans autorisation.
- Absence d'authentification et d'autorisation : avec une API REST, vous pouvez mettre en oeuvre l'authentification et l'autorisation des utilisateurs pour contrôler l'accès à la base de données. Cela garantit que seuls les utilisateurs ou les demandes autorisés peuvent interagir avec la base de données (OAuth ou JWT).

2. Exposition de la base de données :

- Ouverture des ports : se connecter directement à une base de données nécessiterait d'exposer votre serveur de base de données à l'Internet public (si l'application Flutter n'est pas sur le même réseau que la base de données). Cela rend votre base de données vulnérable aux attaques telles que l'injection SQL, les tentatives de connexion par force brute

et les attaques DDoS. Une API REST peut agir comme une couche sécurisée, cachant la base de données derrière elle, empêchant ainsi l'exposition directe au public.

- Pas de protection contre les attaques DDoS et les injections SQL : alors qu'une API REST peut inclure des mesures telles que la limitation du débit, la vérification des entrées et la validation, l'accès direct à la base de données ne bénéficierait pas de telles protections. Sans couche pour filtrer le trafic malveillant, votre base de données est plus exposée.

3. **Séparation des préoccupations :**

- Validation des données et logique d'entreprise : cela permet de s'assurer que les données envoyées à votre base de données sont correctes et dans le format attendu. Sans API, votre application Flutter devrait gérer ces processus, ce qui pourrait entraîner une duplication de la logique et des incohérences potentielles.
- Facilité de maintenance : avec une API en place, vous pouvez modifier votre schéma de base de données, votre backend de stockage de données sans affecter directement votre application. L'application n'interagit qu'avec l'API, ce qui rend votre système plus facile à maintenir et à adapter à l'avenir.

4. **Évolutivité :**

- Séparation entre le client et le serveur : une API REST permet à votre client (application Flutter) d'interagir avec votre serveur de manière uniforme, quel que soit le backend ou la base de données que vous utilisez. Si vous devez faire évoluer votre application (par exemple, changer de base de données, migrer vers une autre infrastructure ou prendre en charge plusieurs types de clients), une API REST vous permet d'effectuer ces changements sans affecter le code du client.
- Équilibrage de la charge : les API peuvent être équilibrées en charge, ce qui vous permet d'évoluer horizontalement (en ajoutant des serveurs) en fonction des besoins, sans faire évoluer directement le client ou la base de données.

5. **Mise en cache et optimisation :**

- Couche de mise en cache : une API REST vous permet de mettre en oeuvre des mécanismes de mise en cache pour les données fréquemment demandées, ce qui réduit la charge sur la base de données et améliore les performances de votre application.
- Requêtes efficaces : l'API peut optimiser les requêtes en ne récupérant que les données nécessaires, tandis que le client n'a pas besoin de gérer lui-même des requêtes complexes ou de filtrer des ensembles de données volumineux.

6. **Cohérence :**

- Logique commerciale centralisée : avec une API REST, toute votre logique et vos règles commerciales peuvent être centralisées. Cela garantit que tout client (web, mobile, etc.) interagissant avec votre backend suivra la même logique et les mêmes règles. Sans API, vous risquez d'avoir une logique incohérente entre les différents clients.

7. **Gestion des erreurs :**

-
- Réponses normalisées : une API REST fournit une manière standardisée de gérer les erreurs et les réponses (codes d'état HTTP, messages, etc.). Il est ainsi plus facile pour l'application Flutter d'interpréter les erreurs et d'y réagir.

8. Compatibilité multiplateforme :

- Intégration flexible : avec une API REST, votre backend peut servir plusieurs types de clients (par exemple, des applications mobiles, des applications web ou même des applications tierces) à l'aide d'une interface cohérente. Les connexions directes à la base de données lieraient votre application Flutter directement à votre base de données, ce qui rendrait plus difficile l'adaptation à d'autres plateformes à l'avenir.

VII/3 Pourquoi l'utilisation de l'adresse 10.0.2.2 ?

Les émulateurs Android utilisent un réseau virtuel isolé. Ils ne comprennent pas localhost comme étant votre machine hôte (l'ordinateur qui exécute Flutter et votre serveur API). Android Studio configure automatiquement 10.0.2.2 comme une passerelle spéciale qui pointe vers localhost de votre ordinateur hôte.

Quand utiliser 10.0.2.2 ? Utilisez cette adresse uniquement lorsque vous travaillez avec un émulateur Android. Pour un appareil physique (téléphone connecté), vous devrez toujours utiliser l'adresse IP locale comme 192.168.x.x.

Exemple avec /nation :

Appareil/Configuration	Adresse API à utiliser
Émulateur Android	http://10.0.2.2:3000/nation
Appareil physique (mobile)	http://192.168.x.x:3000/nation
Ordinateur (Postman/Browser)	http://localhost:3000/nation http://127.0.0.1:3000/nation