

# **Initiation à GraphQL : l'API sur demande.**

## **Travaux Pratiques**



**zenika**

<animés par la passion>

# Pré-requis

---

## Installation

Afin de pouvoir réaliser les exercices, il va falloir d'abord préparer notre environnement.

Assurez vous d'avoir nodeJS installé. Nous allons ici utiliser la dernière version disponible ( `@latest` ).

Pour vérifier l'installation vous pouvez utiliser la commande suivante

```
node -v
# v14.17.0
```

```
npm -v
#7.13.0
```

Si vous n'avez pas nodeJS d'installé, vous pouvez procéder ainsi :

```
curl https://get.volta.sh | bash
```

Puis, relancez un nouveau terminal et lancez :

```
volta install node@latest
```

Vous pouvez utiliser un IDE comme VSCode ou VSCodium avec l'extension GraphQL afin d'avoir la colorisation syntaxique de votre schéma.

## TP1 - Mettre en place l'environnement

Afin d'aborder GraphQL, nous allons commencer par écrire notre schéma contenant tout ce que nous aurons besoin pour la suite.

## La base du schéma GraphQL

---

1. Commencez par créer un fichier `schema.graphql` dans votre dossier `tp1` . Il contiendra notre **contrat** de données et nos différents types.
2. Dans ce schéma nous allons définir deux types : `Users` et `Beers` . Ces deux types auront deux propriétés obligatoires : `id` et `name` , de type `ID` et `String` .
3. Créez ensuite des `Query` pour afficher
  - Un utilisateur par son id
  - Une bière par son id
  - La liste des utilisateurs
  - La liste des bières

# Pour aller plus loin

---

Nous avons maintenant deux types qui implémentent de manière identique deux propriétés.

1. Refactorisez ces deux types en les faisant implémenter une interface `Entity`
2. Ajoutez des commentaires afin de créer la documentation de votre schéma.
3. Ajoutez une propriété `likedBeers` au type `User` contenant un tableau de `Beer`.


## TP2 - Le serveur Apollo

Maintenant que nous avons un schema GraphQL relativement complet, nous allons lancer notre serveur GraphQL. Dans ce TP, nous allons aborder :

- La création du serveur
- Les `resolvers`
- Les `local resolvers`

## Création du serveur

---

 Pour les étapes suivantes, n'hésitez pas à vous référer à la [documentation Apollo](#)

1. Placez-vous dans le répertoire tp2 et lancez la commande `npm install` afin d'installer les dépendances.
  2. Ouvrez le fichier `index.js` à la racine du TP
  3. Importez la définition de votre schéma
    - Afin d'importer `schema.graphql` dans `index.js`, vous pouvez utiliser le code suivant
- ```
import { readFileSync } from "fs";

const typeDefs = readFileSync("./schema.graphql").toString("utf-8");
```
4. Définissez les [resolver](#) des `Query` définies dans votre **schéma**.
    - Pour les `query` utilisez les données importées depuis le fichier `data.js` présent dans le dossier `tp2`.
    - ⚠ Certains resolvers vont nécessiter de sélectionner le bon item dans l'array correspondant via les arguments.
  5. Fournissez les variables `typeDefs` et `resolvers` au `ApolloServer`
  6. Rendez vous dans le playground Apollo `http://localhost:4000/` et essayez les requêtes présentes sur l'interface.

## TP3 - Mutations et requêtes avancées

# Proxy de l'API *PunkAPI*

---

À présent, nous allons traiter notre serveur GraphQL comme un *BFF* : il va faire rebond sur une autre API. Et cette API, elle sert des bières 🍺

Nous allons utiliser [Punk API](#).

Nous allons remplacer notre resolver qui sert le contenu statique du fichier `data.js` par l'API sus-nommée.

1. Placez-vous dans le dossier correspondant et installez les dépendances avec `npm install`.
2. Nous sommes dans un environnement *serveur* et l'API fetch n'étant pas disponible, nous allons utiliser `node-fetch`.
  - Pour installer le paquet, lancez la commande `npm install node-fetch`.
3. Réécrivez le resolver du champ `beers` de `Query` en utilisant la fonction `fetch` précédemment importée.
  - Ce resolver sera asynchrone et interrogera l'url de l'api : `https://api.punkapi.com/v2/beers`.
  - La fonction `fetch` nécessite de transformer le `blob` de la réponse en json via `response.json()`. Voir la [documentation](#)
4. (Bonus) Procédez à la même réécriture pour le resolver du champ `beer` de `Query`.

## Le resolver pour `likedBeers`

---

Nous allons maintenant implémenter le resolver permettant de connaître les bières préférées d'un utilisateur.

1. Implémentez le resolver du champ `likedBeers` sur le type `User`, qui pour le moment retourne un tableau vide.
  - Ce `resolver` va se baser sur la valeur du champ `likedBeersIds` de l'objet utilisateur parent (premier paramètre de la fonction resolver), il va devoir requêter l'API pour récupérer les bières correspondantes.
  - Vous pouvez utiliser le paramètre d'url `?ids=ID|ID` de la *PunkAPI* pour récupérer plusieurs bières spécifiques en une seule requête.

Qu'observez-vous avec ce procédé ? Que pensez-vous de cette façon de faire ?

## Les mutations

---

Maintenant que nous avons implémenté la récupération de nos bières préférées, il serait intéressant d'implémenter la *mutation* pour en ajouter de nouvelles et pour en supprimer.

1. Ajoutez dans le schéma un type Mutation
2. Ajoutez au type Mutation un champ `toggleLike` qui prend en argument deux id de type `ID`, celui du user et celui de la bière, et renvoie l'utilisateur modifié.
3. Implémentez le resolver du champ `toggleLike` de `Mutation`.

- Ce resolver doit ajouter la bière au tableau si elle n'y est pas déjà, sinon il la supprime du tableau.

4. Appelez la mutation correspondante dans le playground et vérifiez que le comportement est celui attendu.

## Les fragments

Nous aimerions une requête qui recherche dans toutes les entités connues, indépendamment de leur type. Nous avons donc besoin dans notre API d'implémenter cette `Query`.

1. Ajoutez dans le schéma la `Query` `search` qui retournera un type `Entity`.
2. Ajoutez dans le schéma la propriété `tagline` sur le type `Beer`.
3. Elle prendra en paramètre un `queryString` qui recherchera dans les champs `name`.
4. Implémentez le `resolver` permettant de rechercher une entité.
  - ⚠ Pour pouvoir requêter notre interface `Entity` - qui peut être de plusieurs type - il est nécessaire d'ajouter un resolver pour cette interface. Celui-ci permet de dire à notre serveur dans les résultats que c'est de type X ou Y.

```
const resolvers = {
  Entity: {
    __resolveType: (root) => {
      return root.tagline ? "Beer" : "User";
    },
  },
},
```

5. En utilisant les `inline fragments` écrivez une `query` dans *Apollo studio* qui affiche la tagline si c'est une bière et le nom si c'est un user.

## TP4 - Apollo client

### Lancement des briques

Vous allez avoir besoin de deux terminaux. Nous utiliserons le serveur du `tp3` comme serveur GraphQL.

1. Placez-vous dans le répertoire `tp3` et lancez votre serveur avec `npm run start`.
2. Placez-vous maintenant de le répertoire `tp4` et lancez l'installation des dépendances avec `npm install`.
3. Lancez l'application avec `npm run start`.
4. Visitez la page `http://localhost:1234`.

Vous devriez voir un formulaire de recherche s'afficher. Pour le moment celui-ci ne fonctionne pas, nous allons le compléter ensemble.

La micro application que nous venons de lancer est généré avec ParcelJS@2. Ce bundler permet d'avoir vite une compilation des assets et sans code. Nous évitons de ce fait l'utilisation d'un framework comme Vue ou React.

# Instanciation d'Apollo client

---

L'instanciation de notre client Apollo a déjà été initialisée dans le fichier `app.js`.

1. Complétez la configuration du client avec l'URL de notre API.
2. Quelles sont les autres variables de configuration indispensable pour instancier un client Apollo ?

## Lancement de la recherche

---

L'implémentation du lancement de la recherche a été initialisée dans l'event listener présent dans le fichier `app.js`.

1. Compléter l'appel à `client.query()` pour faire fonctionner la recherche