# INITIATION À GRAPHQL : L'API SUR DEMANDE.

# SOMMAIRE.

- Introduction
- Schéma
- GraphQL côté serveur
- Requêtes et mutations
- GraphQL côté client
- Serveur avancé

# LOGISTIQUE.

- Horaires
- Déjeuner & pauses
- Autres questions ?

# INTRODUCTION

# SOMMAIRE.

# QU'EST-CE QU'UNE API.

Application Programing Interface

- Logicielle
- *Service Web*

> un ensemble normalisé de classes, de méthodes, de fonctions et de constantes qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels

# QU'EST-CE QU'UNE *BONNE* API.

- Facile d’utilisation ✨
- Répond au besoin 👌
- Bien structurée 🏗️
- Utilisation intuitive 🧠
- Bonne documentation 📄

# FACILE D'UTILISATION

On ne doit jamais être surpris par une réponse. Celle-ci doit correspondre à ce qu'on attendait.

# RÉPOND AU BESOIN

Expose suffisamment de données pour répondre à la demande du métier. Ni trop, ni trop peu.

# UTILISATION INTUITIVE

l'API s'explique d'elle même : elle est "logique" dans sa construction. On peut parler d'une API Restful par exemple.

# BIEN STRUCTURÉE

l'API doit renvoyer des codes d'erreurs parlant, et, en cas d'erreur, nous orienté vers la correction de notre requête. Elle suit les standards.

# BONNE DOCUMENTATION

l'API doit être documenté pour qu'un nouvel utilisateur n'ait pas à deviner les ressources disponibles, l'authentification ou les paramètres. On peut ici parler de Swagger pour les API Rest.

# L'ARCHITECTURE LA PLUS COMMUNE : *REST.*

Representational state transfer

Créé par Roy Fielding en 1999.

- Style d’architecture
- Spécification (standard)
- Verbes d'actions : *GET / POST / PUT / DELETE / PATCH* (+ HEAD / OPTION / CONNECTION / TRACE)

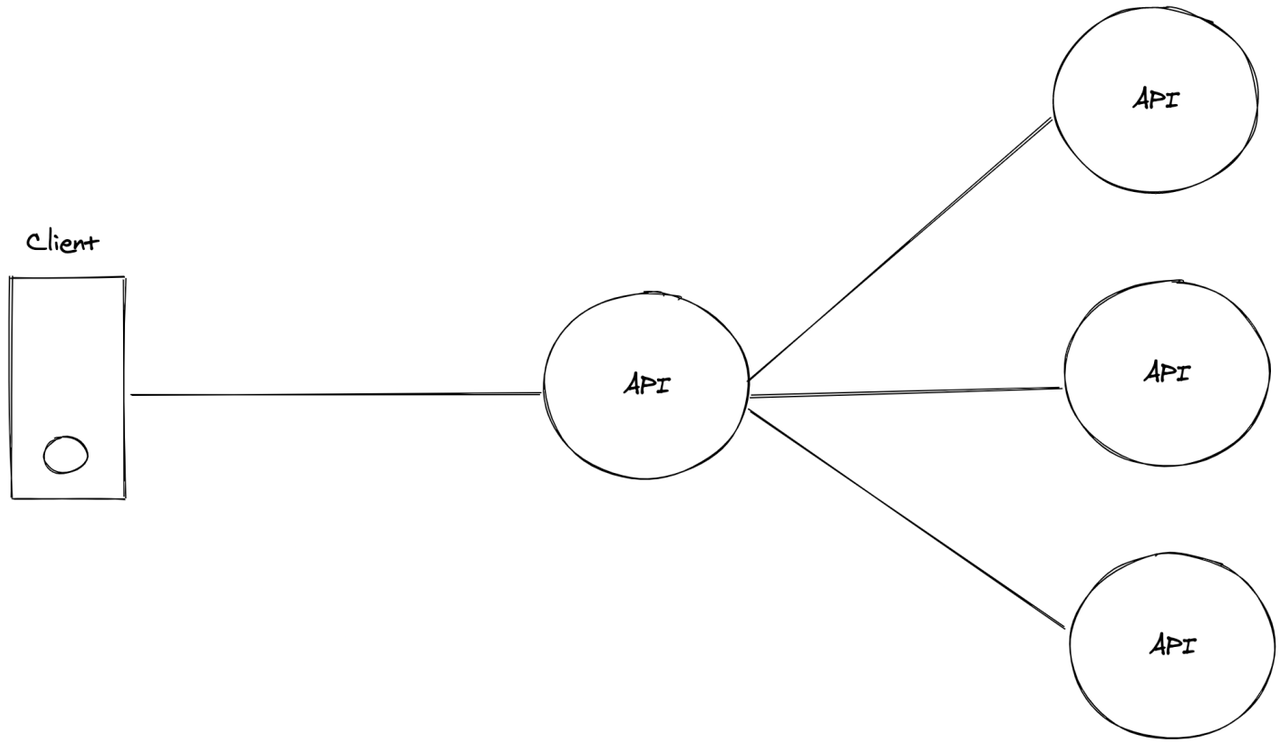
> *Implémenter un changement peut vite devenir pénible*

# AUTRES PROTOCOLES ET PATTERNS.

- Différents patterns et protocoles sont utilisés dans la réalisation d'API et peuvent être exploités dans un BFF : Back for front

- Exemples de techniques de communication entre API :
  - SOAP : style RPC basé sur XML avec définition d'un schéma
  - REST : notion de ressources gérées via les fonctions de HTTP et principalement représentées avec JSON
  - Protobuff : DSL pour décrire la structure des données à échanger
  - Pub/Sub : un consommateur reçoit les évènements d'un producteur

# BFF ET API : ARCHITECTURE COMMUNE.




- L'API gateway offre un point d'entrée unique simple à utiliser
- On parle de Backend For Frontend (BFF) lorsque la gateway offre un service dédié aux consommateurs

# LES PROBLÉMATIQUES LIÉES AU BFF.

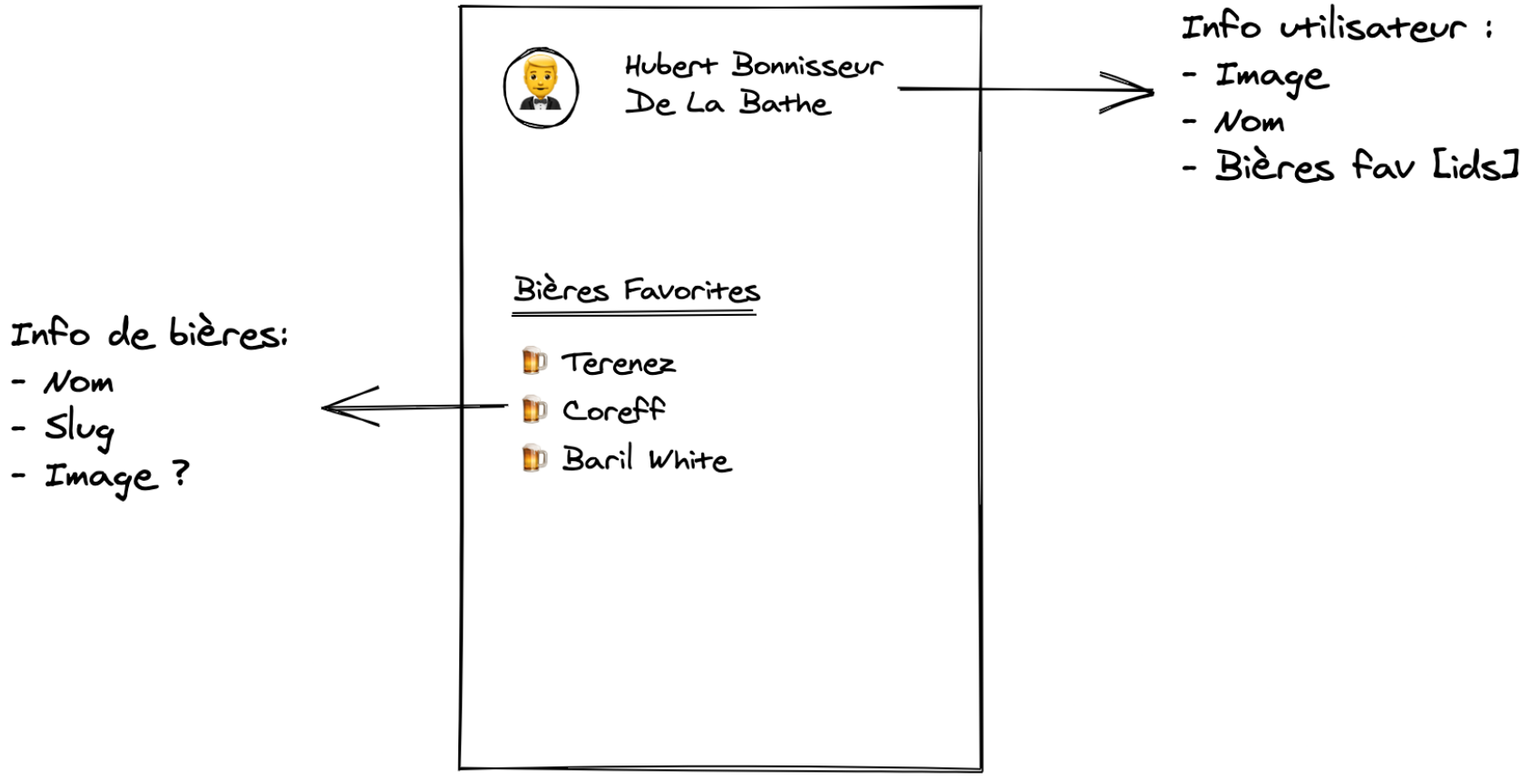
- Gérer des requêtes parallèles vers les API :
  - Comment optimiser les appels vers différentes API ?
- Gérer l'évolution des besoins :
  - Comment satisfaire le besoin en termes de changement de différentes IHM utilisant le même endpoint
- Gérer les erreurs :
  - Quelle API est en erreur ?
  - Quelle réponse envoyer au client si on n'a qu'une partie de la donnée ?
- Nature des paramètres :
  - Dans une API dédiée au frontend, comment identifier simplement qu'un paramètre d'appel correspond à un filtre ou une option d'affichage ?

# EXEMPLE.

Prenons la page de profil d'un amateur de bière.

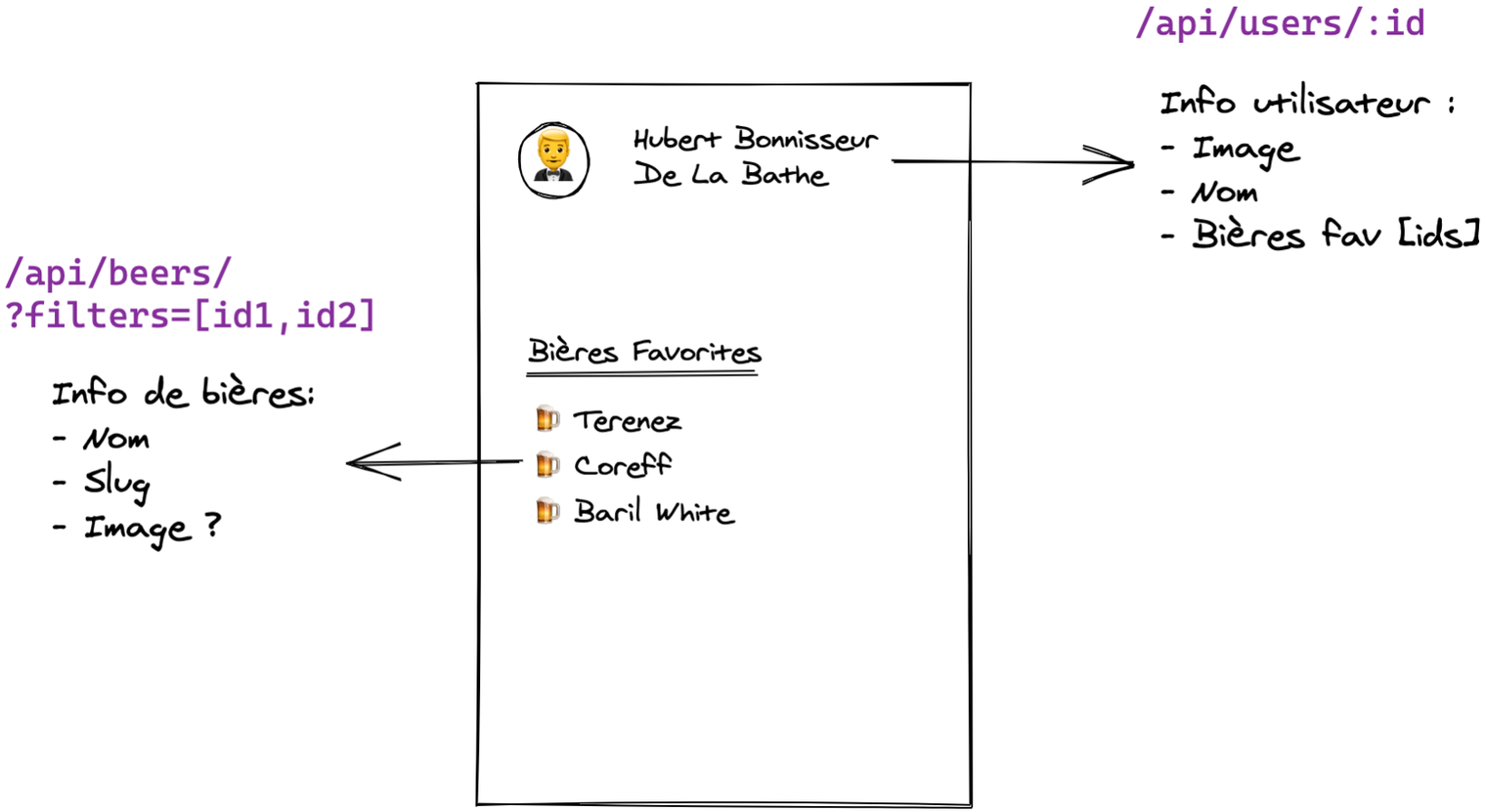
# EXEMPLE.

Et décryptons là :



# EXEMPLE.

Avec ses appels :




Quels inconvénients y voyez vous ?

# LES PROBLÈMES DE CETTE API.

- 2 appels réseaux nécessaires,
- 2eme appel dépendant du 1er (liste de bières)
  - => *Under fetching*
- Les payloads contiennent sûrement trop de chose
  - => *Over fetching*

# LE *POIDS* DES RÉPONSES : PARTIE 1 (USER).

La requête utilisateur `/api/users/:id` :


```
{
  "id": "cf1d58b4-157a-43a2-8149-3ae0d00aa26b",
  "displayname": "Jackson Ratke",
  "firstname": "Maybell",
  "lastname": "Kiehn",
  "username": "Otho1",
  "email": "Gabriel.Cormier55@gmail.com",
  "beers": [1, 2, 3, 4, 5, 6],
  "website": "http://leonor.org",
  "phone": "1-###-###-#### x###",
  "avatar": "https://s3.amazonaws.com/uifaces/faces/twitter/cemshid/128.jpg",
  "isOnBoarded": true,
  "description": "Rustic distributed interface Fresh Investment Account matrices Minnesota e-business Pizza Refined Frozen Ball Sudanese Pound neural-net magenta cross-platform Pa'anga Agent PCI context-sensitive Forward reboot open-source lime TCP alarm program Bedfordshire Operations Cloned Functionality hack National Practical programming Kansas Group"
}
```


Et potentiellement plus d'informations pas nécessairement utiles...

# LE *POIDS* DES RÉPONSES : PARTIE 2 (BEER).

La requête des bières `/api/beers/?filters=[id1,id2]` :


```
[
  {
    "id": 1,
    "name": "Buzz",
    "tagline": "A Real Bitter Experience.",
    "first_brewed": "09/2007",
    "description": "A light, crisp and bitter IPA brewed with English and American hops. A
small batch brewed only once.",
    "image_url": "https://images.punkapi.com/v2/keg.png",
    "abv": 4.5,
    "ibu": 60,
    "target_fg": 1010,
    "target_og": 1044,
    "ebc": 20,
    "srm": 10,
    "ph": 4.4,
    "attenuation_level": 75,
    "volume": {
      "value": 20,
      "unit": "litres"
    },
    "boil_volume": {
```

# LE *POIDS* DES RÉPONSES : CONCLUSION.

On observe ici les deux problématiques : trop de données récupérées et deux requêtes inter-dépendantes.
On imagine s'il fallait lister les bières favorites de tous les utilisateurs, il faudrait faire une requête de bière par utilisateurs...

# GRAPHQL À LA RESCOUSSE.

GraphQL nous permet de limiter les problématiques listées ci-dessus en nous permettant de choisir ce que l'on va récupérer.

Il permet de déporter les traitements *côté serveur*.

```
query {
  user(id: "...") {
    id
    name
    image
    beers {
      id
      name
    }
  }
}
```

- 1 seule requête
- Dépendances incluses

# GRAPHQL À LA RESCOUSSE : LA RÉPONSE.

```
{
  "user": {
    "id": "982a246d-4527-4960-8cf9-b097a1be89ac",
    "name": "Hubert Bonisseur de la Bathe",
    "image": "https://some-cdn.exemple.com",
    "beers": [
      {
        "id": "...",
        "name": "Terenez"
      },
      {
        "id": "...",
        "name": "Coreff"
      },
      {
        "id": "...",
        "name": "Baril White"
      }
    ]
  }
}
```

# MAIS QU'EST CE QUE GRAPHQL ?

> *A query language for your API*

GraphQL a été inventé par Facebook puis la spécification a été open sourcée en 2015.

- Langage de *requêtage*
- Système de *types*
- Environnement d'*exécution* (runtime)
- *Spécification* OpenSource

# COMMENT ÇA MARCHE ?

GraphQL est composé de plusieurs principes de bases, à savoir :

- Un schéma (système de types)
- Syntaxe de requêtage
- Validation
- Introspection

# EXEMPLE - SCHÉMA.

Schéma :

```
type Query {
  user(id: ID!): User!
}

type User {
  id: ID!
  name: String!
  age: Int
  image: String
  beers: [Beer!]!
}

type Beer {
  id: ID!
  name: String!
}
```

# EXEMPLE - REQUÊTE.

Une *query* GraphQL

```
query GetUser {
  user(id: "982a246d-4527-4960-8cf9-b097a1be89ac") {
    id
    name
    image
    beers {
      id
      name
    }
  }
}
```

# EXEMPLE - RÉPONSE.

La réponse ne contient *que* les données *demandées* et les entités liées sont déjà incluses

```
{
  "data": {
    "id": "someId",
    "name": "Hubert Bonisseur de la Bathe",
    "image": "https://some-cdn.exemple.com",
    "beers": [
      {
        "id": "anotherId",
        "name": "Baril White"
      }
    ]
  }
}
```

*C'est un graph !*

# SCHÉMA

# SOMMAIRE.

# LE SCHÉMA.

GraphQL repose sur un schéma qui décrit :

- Les *types* disponibles,
- Les *requêtes*, nommées *query*
- Les *mutations*,
- Les *souscriptions*,
- Des `Input` : *paramètres* pour une requêtes ou mutation
- Des *énumérations*

# TYPES SCALAIRES.

GraphQL prévoit un certain nombre de types scalaires par défaut :

- `Int` : nombre entier (32 bits)
- `Float` : nombre décimal (IEEE 754)
- `String` : une chaîne de caractères encodée en `UTF-8`
- `Boolean` : `true` ou `false`
- `ID` : utilisé pour les identifiants uniques, traités comme un `String` même s'il s'agit d'un id numérique (UUID)

> Il est possible de définir ses propres types scalaires comme un type `Date` par exemple

# OBJETS.

Un type objet se déclare en utilisant le mot clé `type`, les champs possibles sont déclarés à l'intérieur des accolades avec leurs types respectifs :

```
type User {
    id: ID
    name: String
    age: Int
}
```

Exemple :

```
{
    "id" : "123",
    "name": "John Doe",
    "age": 33
}
```

# LISTES.

Une liste se déclare en mettant des crochets autour du type. Ainsi pour déclarer une *liste* de rôle pour un utilisateur, on va écrire :

```
type User {
    id: ID
    name: String
    roles: [String]
}
```

roles est ici un champ de type liste de chaînes de caractères

Exemple :

```
{
    "id" : "123",
    "name": "John Doe",
    "roles" : [ "MAINTAINER", "REPORTER" ]
}
```

# LES TYPES NON NULLABLE.

Pour préciser qu'un type ne peut pas être `null`, on utilise le caractère `!`.

Exemple :

```
type User {
  id: ID! # L'id ne peut pas être nul
  roles: [String!]! # Tableau non nul de chaines non nulles
}
```

# TYPE QUERY.

Le type `Query` permet de définir les requêtes possibles :

```
type Query {
  emails: [String!]!
}
```

Ici on définit une seule requête pour récupérer un tableau d'emails

# ARGUMENTS.

Les arguments doivent être *nommés* et *typés* comme n'importe quel champ :

```
type Query {
    emails: [String!]!
    email(userId: ID!): String
}
```

Ici email prend en paramètre un `ID` d'utilisateur qui est *obligatoire*

# ARGUMENTS.

Plusieurs arguments peuvent être déclarés, séparés ou pas par des virgules :

```
type Query {
    price(id: ID!, currency: String): String
}
```

Ici l'argument `id` est obligatoire, mais l'argument `currency` est optionnel.

Une valeur par défaut peut être spécifiée :

```
type Query {
    price(id: ID!, currency: String = "$"): String
}
```

# MUTATIONS.

Les mutations se déclarent dans le type `Mutation` et non Query. Elles permettent d'effectuer des changements, comme on peut le faire en REST avec POST, PUT, PATCH, DELETE. Ellse retournent toujours une donnée.

```
type Mutation {
    # Mutation nommée "updatePrice" qui retourne les données modifiées
    updatePrice(id: ID!, price: String!): Price!
}
```

# OBJETS D'INPUT.

Les arguments peuvent avoir n'importe quel type scalaire.

Pour les cas plus complexes, il est également possible de créer son propre type d'argument

```
type Query {
  books(search: String, sort: SortInput): Book
}

input SortInput {
  field: String
  order: String
}
```

# ENUMÉRATIONS.

Les énumérations se déclarent avec le mot clé enum :

```
enum Color {
    RED
    GREEN
    BLUE
}
```

Le schéma n'indique pas quel type scalaire sert à représenter les valeurs de l'énumération

Une énumération se manipule ensuite comme n'importe quel autre type :

```
type Product {
    name: String
    color: Color
}
```

# INTERFACES.

Les interfaces permettent d'introduire un niveau d'abstraction supplémentaire.

Plusieurs types peuvent implémenter la même interface : ils devront alors obligatoirement contenir *au moins tous les champs* de l'interface.

```
interface Person {
    name: String
}

type Teacher implements Person {
    name: String
    subject : String
 }

type Student implements Person {
    name: String
    rating: Int
}

type Query { find(name: String!): Person }
```

# UNIONS.

Les unions apportent le même niveau d'abstraction que les interfaces mais n'ont pas de champs en commun

Exemple :

```
type Book { title: String }

type Equipment { model : String }

type Furniture { height: Int length: Int }

union InventoryItem = Book | Equipment | Furniture

type Query {
   inventoryItem(id: ID!): InventoryItem
}
```

Il faudra obligatoirement utiliser des *fragments* pour requêter un champ avec un type `Union`

# DOCUMENTATION.

Les commentaires dans un schéma graphQL sont de la documentation. Et la documentation c'est bien 😍.

On défini un commentaire par le caractère `"`, ou `"""` pour un commentaire multi-lignes.

```
# Description du type User
type User {
  """
  l'identifiant unique de l'utilisateur,
  pour plus d'infos, voir [ici](https://fake.com)
  """
  id: ID!

  "Nom de l'utilisateur"
  name: String!
}
```

Un commentaire multi-lignes permet le texte enrichi par l'interprétation du *Markdown*

# GRAPHQL CÔTÉ SERVEUR

# SOMMAIRE.

# CÔTÉ SERVEUR : LES ENJEUX.

Offrir une API facilement exploitable pour les clients en :

- Définissant un schéma qui permet au client d'écrire des requêtes
- Implémentant les différentes fonctions qui permettront de résoudre différents types définis dans le schéma

Au delà des impératifs propres à la spécification GraphQL, vous devez être capable :

- D'appeler facilement des services tiers pour implémenter vos fonctions
- Rendre accessible vos fonctions via un serveur HTTP

Une des implémentation les plus utilisées est apollo server, elle permet d'écrire un serveur exposant une API GraphQL et s'exécutant sur NodeJS.

# DÉFINITION DU SCHÉMA.

On l'a vu, GraphQL expose un schéma. Vous pouvez opter pour une approche contract first où vous définissez un schéma en premier lieu.

Avec apollo-server, vous pouvez parser votre schéma comme ceci :

```
import { gql } from 'apollo-server'

const typeDefs = gql`
  type Query {
    beer(id: ID!): Beer!
    beers: [Beer!]!
  }

  type Beer {
    id: ID!
    name: String!
  }
`
```

La variable typeDefs pourra être alors utilisé dans l'initialisation d'un serveur HTTP avec apollo-server

# LES RESOLVERS.

Une fois le schéma écrit, il faut définir comment récupérer les données de chaque champ. Pour cela nous allons définir des resolvers

Il s'agit de fonction permettant de récupérer la valeur pour chacun des champs/type de notre schéma.

Ils sont automatiquement appelés par *Apollo* lorsqu'un champ est demandé par un client.

```
import beerService from './beerService'

const resolvers = {
  Query: {
    beer: (parent, { id }) => beerService.getById(id),
    beers: () => beerService.list(),
  },
}
```

# LES RESOLVERS.

Il est important de comprendre comment fonctionne le serveur GraphQL afin de comprendre ses limitations.

Chaque type de notre schéma doit définir son resolver afin de dire à notre serveur comment et ou récupérer les données de ce type en particulier.

Une query contenant des types imbriqués appellera *autant de resolvers que de types* dans la requête.

Il faudra écrire les resolvers de tout ce qui est dans notre schéma : nos types, mais aussi les query, les mutations etc...

# LANCEMENT DU SERVEUR.

En résumé, pour lancer notre serveur nous devons lui donner :

- Le schéma
- Tous nos resolvers

```
import { gql, ApolloServer } from 'apollo-server'

const typeDefs = gql`...`
const resolvers = { Query: { ... } }

const server = new ApolloServer({ typeDefs, resolvers })

server.listen().then(({ url }) => {
    console.log(`🚀  Server ready at ${url}`)
})
```

# CONSTRUCTION D'UN RESOLVER.

## 1ER ARGUMENT : ROOT

GraphQL possède une notion de hiérarchie entre resolvers.

Prenons par exemple le schéma suivant :

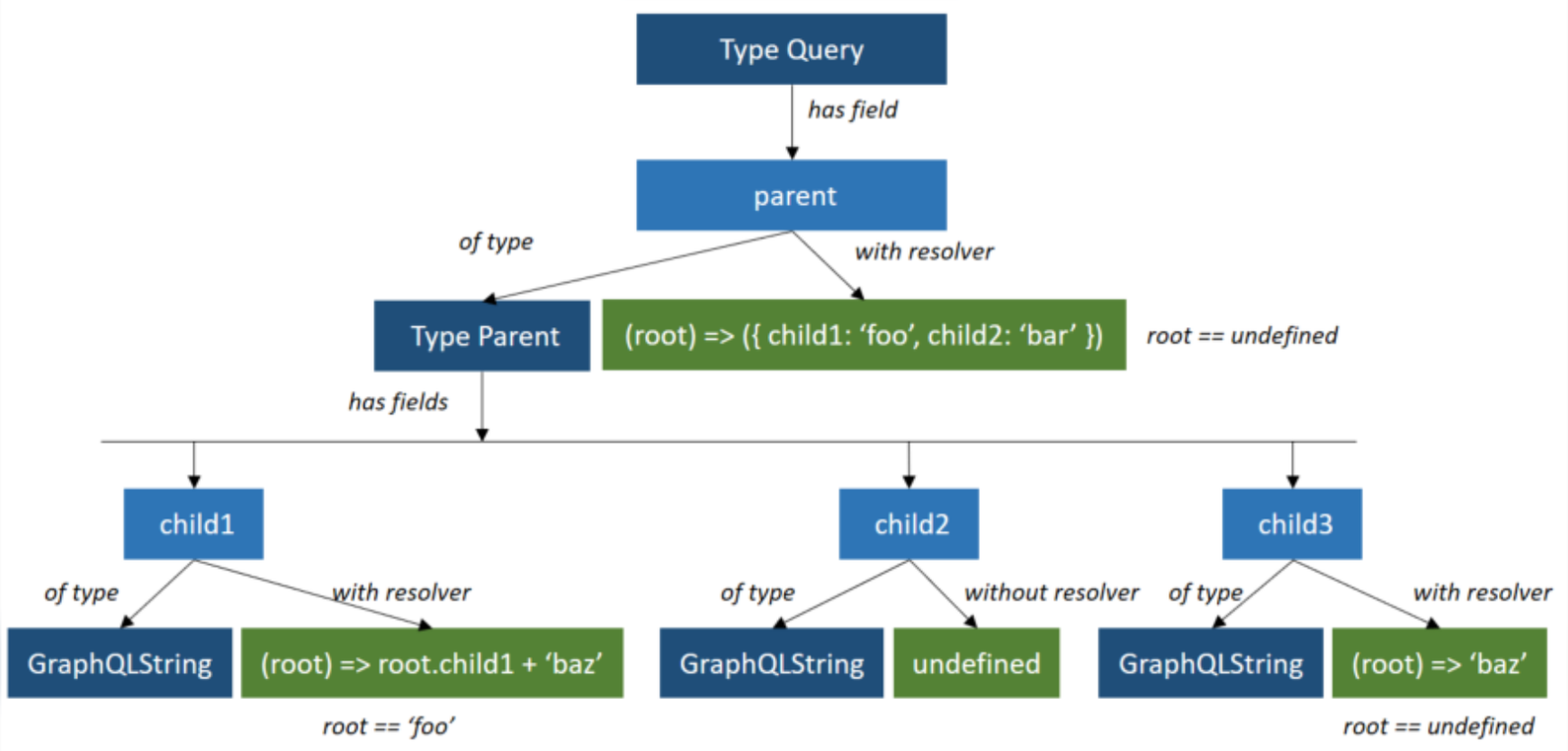
```
type Query {
    parent: Parent
}

type Parent {
    child1: String
    child2: String
    child3: String
}
```

et la requête suivante `{ parent { child1 child2 child3 } }`

# CONSTRUCTION D'UN RESOLVER.

## 1ER ARGUMENT : ROOT

Sa résolution se fera comme ci-dessous :



# CONSTRUCTION D'UN RESOLVER.

## 1ER ARGUMENT : ROOT

Quelques précisions sur les resolvers :

- La fonction resolve n'est pas obligatoire sur un type
- La valeur d'un type peut être spécifiée par le resolver du type hiérarchiquement parent
- Le premier paramètre du resolver est l'état de la valeur raçine au moment de l'exécution de la méthode (root)

# CONSTRUCTION D'UN RESOLVER.

## 1ER ARGUMENT : ROOT

Le code javascript peut donc correspondre à quelque chose comme cela :

```
const resolvers = {
    Query: {
        parent: (root) => { child1: "foo", child2: "bar" }, // Type: Parent
    },
    Parent: {
        child1: (root) => root.child1 + "baz",              // Type: String
        child3: (root) => "baz",                            // Type: String
    }
}
```

Et sa réponse serait :

```
{ "data" :
    { "parent" : { "child1" : "foobaz", "child2" : "bar", "child3" : "baz" } }
}
```

# CONSTRUCTION D'UN RESOLVER.

## 2ÈME ARGUMENT : ARGUMENTS

Le second paramètre contient tous les arguments du champ.

```
type Query {
    parent(parentArg: String): Parent
}

type Parent {
    child(childArg: String): String
}
```

```
const resolvers = {
    Query: {
        parent: (_, { parentArg }) => ...
    },
    Parent: {
        child: (_, { childArg }) => ...
    }
}
```

# CONSTRUCTION D'UN RESOLVER.

## 3ÈME ARGUMENT : CONTEXT

Le troisième paramètre est un contexte d'exécution qui contient par défaut la requête HTTP courante.

Il est configurable via les options de `ApolloServer`:

```
import { getUser } from './auth'

const server = ApolloServer({
    typeDefs,
    resolvers,
    context: ({ req }) => ({
        user: getUser(req.headers.authorization)
    })
}))
```

Il est souvent utilisé pour stocker les informations d'authentification de l'utilisateur et est modifiable afin d'échanger des informations entre resolver

# CONSTRUCTION D'UN RESOLVER.

## 4ÈME ARGUMENT : INFO

Le quatrième et dernier paramètre contient les informations liées à la requête exécutée. Il permet par exemple de construire une requête SQL. Par exemple cette requête combiné à ce resolver :

```
query {
    select { foo, bar }
}
```

```
const resolvers = {
    Query: {
      select: (_root, args, context, info) => {
          const toValue = selection => selection.name.value;
          const toValues = node => node.selectionSet.selections.map(toValue)
          const fields = info.fieldNodes.map(toValues).flat().join(", ")
          return sqlselect(` SELECT ${fields} FROM my_table`);
      }
    }
};
```

# CONSTRUCTION D'UN RESOLVER.

## 4ÈME ARGUMENT : INFO

Produira la requête SQL suivante :

```
SELECT foo, bar FROM my_table
```

# CONSTRUCTION D'UN RESOLVER.

En résumé :

- un resolver attend quatre paramètres :

```
resolve(root, params, context, info): Promise<Result> | Result
```

- Seuls les resolvers des champs demandés sont exécutés
- Les resolvers sont éxécutés en parallèle
- Les resolvers enfants sont éxécutés après que leur parent ait été résolu
- Un resolver peut être asynchrone

# REQUÊTES ET MUTATIONS

# SOMMAIRE.

# REQUÊTE DE BASE.

En GraphQL, une requête suit cette structure :

```
{
    field_1
    field_2
    field_n
}
```

Les champs peuvent être sur une même ligne :

```
{
    field_1 field_2 field_n
}
```

Et optionnellement séparés par des virgules :

```
{
    field_1, field_2, field_n
}
```

# MOT CLÉ QUERY.

Le mot clé query permet de nommer une requête, ce qui est une bonne pratique :

```
query myQuery {
    field_1  field_2  field_n
}
```

On peut omettre le nom de la requête

```
query {
    field_1  field_2  field_n
}
```

Ce qui équivaut à :

```
{
    field_1  field_2  field_n
}
```

# TYPES COMPLEXES.

Les exemples précédents montrent comment sélectionner des champs de type scalaire (chaîne de caractères, nombre...)

Nous pouvons aussi sélectionnés des champs dits complexes :

```
{
    field_1
    complexe_object {
        field_1
        field_2
    }
}
```

En revanche, il est interdit de sélectionner un champ de type complexe sans préciser de sous-champs.

```
{
    field_1
    complexe_object # Field "complexe_object" must have a selection of subfields
}
```

# ÉNUMÉRATIONS.

Les énumérations se distinguent simplement par le fait qu'elles définissent une liste finie de valeurs possibles. Elles permettent d'énumérer uniquement des types scalaires.

Les champs de types énumérés se requêtent comme des types scalaires :

```
{ name color }
```

- name est de type scalaire (une chaîne de caractère libre) et color est une énumération de valeurs de type scalaire (black et white par exemple)
- Aucune différence n'est faite dans la requête. Résultat possible :

```
{ "data": { "name": "chair", "color": "white" } }
```

# LISTES.

Un champ peut être une liste d'objets d'un certain type. Dans ce cas, la réponse pour un tel champ sera un tableau :

```
{
    cities
}
```

Réponse possible dans le cas où cities est une liste de chaînes de caractères :

```
{
    "data": {
       "cities": [
           "Paris", "Lyon", "Bordeaux", "Nantes", "Rennes"
       ]
    }
}
```

# RÉPONSE.

Elle est souvent en JSON, mais peut être dans d'autres format en fonction de l'implémentation voulue. Elle contient généralement la valeur des différents champs sélectionnés dans la requête dans une propriété data.

```
query {
    ville
    stats { superficie, population }
}
```

Réponse :

```
{
    "data": {
        "ville": "Lille",
        "stats": {
            "superficie" : 34.51,
            "population" : 1182127
        }
    }
}
```

# ERREUR.

En GraphQL, il n'y a pas de code d'erreur HTTP. Une requête qui échoue reste en HTTP 200 OK. L'erreur est en revanche spécifié dans la réponse de la requête dans un champ errors.

```
query {
    ville, stats { superficie, population }
}
```

Une réponse indiquant une erreur pour stats peut être retournée :

```
{
    "data":  { "ville" : "Lille", "stats" : null },
    "errors": [{
        "message": "Stats are unavailable",
        "locations": [{ "line" : 2, "column" : 5 }],
        "path": ["stats"]
    }]
}
```

# ARGUMENTS.

Il est possible de fournir des arguments pour un champ donné en utilisant les parenthèses.

```
{ champAvecParam(param1: "foo", param2: "bar") }
```

Comme pour la séléction des champs, les virgules pour séparer les arguments sont optionnelles.

```
{ champAvecParam(param1: "foo" param2: "bar") }
```

Les paramètres sont typés et peuvent être des types scalaires, complexes et énumérés. Le résultat d'une résolution peut dépendre d'un paramètre :

```
{ beers(country: "belgium") { name } }
```

```
{ beers(country: "france") { name } }
```

Ces deux requêtes afficheront un name différent

# ARGUMENTS.

La requête peut elle-même déclarer des arguments utilisables dans les champs :

```
query($param: String) { champAvecParam(param1: $param) }
```

- La variable param doit être déclarée avec un type correspondant à celui attendu pour param1
- Il faut fournir au serveur un objet contenant les valeurs de chacune des variables déclarées dans la requête associée

Dans le cadre d'un serveur HTTP, on peut spécifier un Content-Type: application/json et encapsuler les 2 informations dans un JSON :

```
{
    "query": "query($param: String) { champAvecParam(param1: $param) }",
    "variables" : { "param" : "foo" }
}
```

# ARGUMENTS.

Des arguments peuvent être optionnels. Si on ne veut pas préciser de valeur, il suffit d'omettre l'argument.

```
query {
    champAvecParam
}
```

Revient à écrire :

```
query {
    champAvecParam(param1: null)
}
```

Si on ne précise aucun argument, il ne faut pas mettre les parenthèses

# ALIAS.

Dans la réponse, les noms des champs sont ceux utilisés dans la requête. Ce qui implique que vous ne pouvez pas sélectionner plusieurs fois le même champ dans une même query. Il est nécessaire de lui mettre un `Alias`

```
query {
    # 👇 Alias "belgium" ici
    belgium: beers(country: "belgium") { name }
    # 👇 Alias "france" ici
    france: beers(country: "france") { name }
}
```

Va renvoyer :

```
{
    "data" : {
      "belgium" : [{ "name" : "Belgian beer"}, ...],
      "france" : [{ "name" : "French beer"}, ...]
    }
}
```

# LES FRAGMENTS.

À contrario, dans une requête, vous pouvez sélectionner les mêmes champs à plusieurs endroits différents, par exemple :

```
query {
    belgium: beers(country: "belgium") {
        name ibu beertype ingredients { name }
    }

    france: beers(country: "france") {
        name ibu beertype ingredients { name }
    }
}
```

# LES FRAGMENTS.

Cette répétition peut être refactorisée grâce aux fragments, de la même façon que la syntaxe de décomposition en javascript :

```
query {
    belgium: beers(country: "belgium") { ...beerFields }
    france: beers(country: "france") { ...beerFields }
}

fragment beerFields on Beer {
    name ibu beertype ingredients { name }
}
```

# LES *INLINE* FRAGMENTS.

GraphQL permet de définir des types à partir d'union ou d'interfaces. Cela signifie que les champs peuvent différer selon le type retourné.

Afin de sélectionner des champs particuliers selon le type, vous pouvez utiliser des inline fragments.

```
query {
    shape(id: 1) {
        area
        ... on Square { length }
        ... on Circle { diameter }
    }
}
```

```
{
    "data" : {
        "shape" : {
            "area" : "16",
            "length" : "4"
        }
    }
}
```

# INTROSPECTION.

L'introspection permet d'obtenir des meta données : des informations complémentaires concernant notre serveur graphQL ou notre schéma.

Par exemple, si on prend l'exemple précédent, on peut utiliser la meta donnée __typeName afin de déterminer facilement quel type et donc quels champs seront présents dans la réponse :

```
query {
    shape(id: 1) {
        __typeName
        ... on Square { length }
        ... on Circle { diameter }
    }
}
```

```
{
    "data" : {
        "shape" : {
            "__typeName" : "Square",
            "length" : "4"
        }
```

# DIRECTIVES.

Les directives permettent de conditionner la présence de certains champs dans une réponse.

Deux possibilités:

- @include(if: Boolean) : permet d'inclure le champ si le booléen est vrai
- @skip(if: Boolean) : permet d'exclure le champ si le booléen est vrai

Il est possible de passer en paramètre d'une directive une variable, ce qui permet de modifier la requête très facilement :

```
query Person($withFather: Boolean!, $withoutChildren) {
    characters {
        name
        father @include(if: $withFather) { name }
        children @skip(if: $withoutChildren) { name }
    }
}
```

# DIRECTIVES.

Avec withFather = true et withoutChildren = true

```
{
    "data" :  {
        "characters" : [{ "name": "Eddard", "parent" : { "name" : "Rickard" } }]
    }
}
```

Avec withFather = false et withoutChildren = false

```
{
  "data" : {
      "characters" : [{
          "name": "Eddard",
          "children" : [
              { "name" : "Rob" },
              { "name" : "Sansa" },
          ]
      }]
  }
}
```

# MUTATIONS.

Bien que GraphQL soit très utilisé pour la lecture - avec query - il ne peut pas être complet sans écriture. Les mutations, comme leur nom l'indique, sont là pour muter les données.

Une mutation retourne un objet sur lequel on peut sélectionner des champs exactement comme pour une query. Cela permet par exemple de récupérer une donnée à jour après écriture.

```
mutation($article: Article, $comment: Comment) {
  addComment(article: $article, comment: $comment) {
      comments { author content }
  }
}
```

Comme une query une mutation peut contenir plusieurs opérations. En revanche, contrairement aux query qui sont exécutées en *parallèle*, les mutations sont exécutées *en série*.

# MUTATIONS.

Il est recommandé de nommer une mutation au même titre qu'une query :

```
mutation addCommentAndGetList($article: Article, $comment: Comment) {
  addComment(article: $article, comment: $comment) {
      comments { author content }
  }
}
```

Les `mutations` peuvent recevoir des données via les `arguments`, comme vu plus haut.

# GRAPHQL CÔTÉ CLIENT

# SOMMAIRE.

# GRAPHQL VIA HTTP.

Les principales implémentations de GraphQL se basent sur HTTP, bien que la spécification soit agnostique de ce protocole.

Côté client, on émet une requête GraphQL en l'envoyant via une requête HTTP au serveur.

La requête HTTP à envoyer dépend de l'implémentation côté serveur

Les queries et les mutations doivent respecter le schéma exposé par le serveur.

La réponse est généralement en JSON, il revient au client de l'interpréter en accord avec la spécification.

# EXEMPLE AVEC EXPRESS-GRAPHQL.

Par exemple, avec express-graphql, on envoie une requête HTTP de type POST avec la requête GraphQL dans le corps de la requête.

```
const headers = new Headers();
headers.append("Content-Type", "application/graphql");

await fetch('http://localhost:4000', {
  method: "POST",
  body: "{ products { description price } }",
  headers
})
```

# EXEMPLE AVEC EXPRESS-GRAPHQL.

Il est également possible d'encapsuler la requête dans du JSON pour utiliser des variables.

```
const headers = new Headers();
headers.append("Content-Type", "application/json");

await fetch('http://localhost:4000', {
  method: "POST",
  body: {
    query: "query q($type: String!) { products(type: $type) { description } }",
    variables : { "type" : "car" }  },
  headers
})
```

Il est également possible de tout passer en query string via un GET :

```
/?query=query+q($t:String!){products(type:$t){description}}&variables={"t":"car"}
```

A noter que dans le cas d'un POST on peut également utiliser un Content-Type: application/x-www-form-urlencoded

# EXÉCUTER SES REQUÊTES GRAPHQL.

Communiquer avec un serveur GraphQL en HTTP pur est en général trop bas niveau. Voyez plutôt :

```
async function fetchZipCode {
  const response = await fetch('http://localhost:4000/graphql', {
    method: 'POST',
    body: `
      query {
        search(query: "Nantes") { zipCode }
      }
    `
  })

  if(!response.ok) throw new Error('Fetch Error')
  const {data, errors} = await response.json()
  if(errors.length) throw new Error('GraphQL Error')

  return data.search.zipCode
}

console.log(await fetchZipCode())
```

# EXÉCUTER SES REQUÊTES GRAPHQL.

On aimerait bien s'appuyer sur des composants plus haut niveau pour avoir

- Une gestion du cache automatique
- Gestion de l'authentification
- Gestion du batch de query
- Gestion des erreurs et du retry en cas d'erreur

Exemple :

```
function fetchZipCode(search) {
  const { data, errors } = await client.query({
    query: gql` query($search: String) { search(query: $search) { zipCode } } `,
    variables: { search },
  })

  if(errors) throw new Error('Graphql Error')
  return data.search.zipCode
}
```

# APOLLO CLIENT.

Apollo (www.apollographql.com) est un ensemble de projets visant à faciliter l'écriture des clients GraphQL dans différents langages

- Technologies : React, Angular, Vue.js, iOS, Android
- Languages : Java, Kotlin, Scala, Swift, Javascript, Typescript
- Le projet se structure en deux familles de composants :
  - ceux qui permettent d'exécuter des requêtes GraphQL sur un serveur HTTP dans un langage donné
  - ceux qui permettant de générer des classes correspondant aux requêtes et résultats attendus
- Dans nos applications, nous pouvons donc :
  - Importer Apollo pour exécuter des requêtes au runtime
  - Utiliser l'outillage pour générer des classes à partir de nos requêtes et du schéma au build time (pour les langages compilés)

# EXEMPLE AVEC APOLLO CLIENT.

Nous utilisons le client Apollo pour faciliter l'exécution de requête au runtime :

```
import { ApolloClient, InMemoryCache } from '@apollo/client';

const client = new ApolloClient({
    uri: '<YOUR_GRAPHQL_ENDPOINT>',
    cache: new InMemoryCache()
});

async function main() {
  const { data } = await client.query({
    variables: { query: "Nantes" },
    query: gql`
      query search($query: String!) { zipCode }
    `
  })

  console.log(data.search.zipCode)
}
```

# EXEMPLE EN JAVA AVEC APOLLO.

On peut utiliser d'autres langages pour requêter notre serveur

```
final ApolloClient apolloClient = ApolloClient.builder()
        .serverUrl("http://localhost:4000/graphql").build();

apolloClient.query(SearchQuery.builder().query("LILLE").build()).enqueue(
    new ApolloCall.Callback<SearchQuery.Data>() {

   public void onResponse(Response<SearchQuery.Data> response) {
       final SearchQuery.Data data = response.data();
       System.out.println(data.search().zipCode());
   }

   public void onFailure(ApolloException e) { e.printStackTrace(); }
});
```

- ApolloClient.query() prend en paramètre une instance d'une classe qui doit implémenter l'interface Query

# LES LINKS APOLLO.

Apollo propose différentes implémentations permettant d'enrichir le comportement du client dans ses interactions avec le serveur.

Ce sont les *Links*. Par exemple :

- `@apollo/client/link/error` : inspecter les erreurs GraphQL
- `@apollo/client/link/retry` : réessayer un appel en cas d'erreur
- `@apollo/client/link/batch-http` : grouper plusieurs opérations

Il est tout à fait possible d'en combiner plusieurs et d'écrire ses propres *Link* :

```
import { from, ApolloLink, HttpLink } from '@apollo/client';
import { RetryLink } from '@apollo/client/link/retry';
import { AuthLink } from './auth-link' // Custom link

const link = from([
  new RetryLink(),
  new AuthLink(),
```

# GÉNÉRATION AUTOMATIQUE.

Pour générer les implémentations de manières automatiques par rapport à nos requêtes brutes on peut s'appuyer sur des outils comme apollo-codegen et apollo-compiler.

Les implémentations sont disponible dans pas mal de langages comme Java, Kotlin et même Typescript.

# APPARTÉ SUR LES TESTS.

Pour faciliter les développements et les tests unitaires, Apollo permet de créer des serveurs avec des mocks automatiques.

Grâce à la force du Schéma qui lie nos intervenants on peut facilement générer des données aléatoire basée sur les types des champs.

C'est ce que fait par exemple le serveur de mock d'Apollo.

# SERVEUR AVANCÉ

# SOMMAIRE.

# SOUSCRIPTIONS.

La spécification définie Query (pour la lecture) et Mutation (pour la modification), mais il y a également le type Subscription (pour l'abonnement aux changements).

L'implémentation de ce concept dans le monde du WEB s'appuie principalement sur les websockets.

L'API GraphQL liste toutes les souscriptions disponibles, chacune correspondant à un évènement spécifique envoyé au client qui y souscrit.

Les souscriptions définissent un usage spécifique et apporte leurs lots de complexité de mise en oeuvre. Il faut donc bien y réfléchir et s'assurer de son cas d'usage. Voir https://www.apollographql.com/docs/react/data/subscriptions/#when-to-use-subscriptions

# SYNTAXE DES SOUSCRIPTIONS.

## CÔTÉ SCHÉMA

Dans notre Schéma cela prend la forme d'un nouveau type Subscription qui listera toutes nos souscriptions.

```
type Subscription {
  commentAdded: String!
}

type Query {
  comments: [String!]!
}

type Mutation {
  addComment: String!
}
```

# SYNTAXE DES SOUSCRIPTIONS.

## CÔTÉ APOLLO-SERVER

> Apollo dans sa version 3 ne prend plus en charge les souscription *par défaut*. Le système n'était de toute façon pas stable pour une utilisation en production. Il est possible de le rendre compatible, ce qui est fait dans le TP5 pour tester cette fonctionnalité.

La souscription se compose de deux parties : l'*écouteur* et l'*écouté*.

1. L'écouteur va se trouver des les resolvers de la Subscription correspondante et être une fonction retournant un `AsyncIterator`
2. L'écouté va être déclenché depuis une *mutation* par exemple, en passant le payload en paramètre du nom de la souscription.

# SYNTAXE DES SOUSCRIPTIONS.

## CÔTÉ APOLLO-SERVER

```
import { PubSub } from "graphql-subscriptions";
const pubsub = new PubSub()

const resolvers = {
  Subscription: { commentAdded: {
    subscribe: () => pubsub.asyncIterator(['COMMENT_ADDED'])
  } }
};
```

```
// in one of our mutation, pubsub.publish()

const resolvers = {
  Mutation: { addComment: () => {
    pubsub.publish("COMMENT_ADDED", { commentAdded: '<COMMENT>' });
  }
};
```

# SYNTAXE DES SOUSCRIPTIONS.

## CÔTÉ APOLLO-CLIENT

Côté client, il faut mettre en place un `link` spécifique pour supporter les websockets :

```
import { split, HttpLink, InMemoryCache } from "@apollo/client";
import { getMainDefinition } from "@apollo/client/utilities";
import { WebSocketLink } from "@apollo/client/link/ws";

const httpLink = new HttpLink({ uri: "http://localhost:4000/" });
const wsLink = new WebSocketLink({ uri: `ws://localhost:4000/` });

const isSubscription = ({ query }) => {
  const definition = getMainDefinition(query);
  return (
    definition.kind === "OperationDefinition" &&
    definition.operation === "subscription"
  );
};

const link = split(isSubscription, wsLink, httpLink);

const client = new ApolloClient({ link, cache: new InMemoryCache() });
```

# DIRECTIVES DE SCHÉMA.

Il est possible d'utiliser des directives au niveau du schéma. GraphQL propose par défaut la directive `@deprecated` :

```
enum MyEnum {
    VALUE
    OLD_VALUE @deprecated
    OTHER_VALUE @deprecated(reason: "Terrible reasons")
}

type Query {
    field1: String @deprecated
    field2: Int @deprecated(reason: "Because I said so")
    enum: MyEnum
}
```

Passer un champ en `@deprecated` avant de le supprimer de votre schéma est recommandé car cela donne le temps aux clients de migrer

# DIRECTIVES PERSONNALISÉES.

Les directives par défaut sont `@skip`, `@include` et `@deprecated`, et il est possible de définir ses propres directives. Pour cela il est recommandé d'utiliser la librairie `@graphql-tools` disponible sur npm.

C'est un ensemble de paquet que l'on peut installer pour s'aider et étendre les fonctionnalités de notre serveur graphQL.

- Voir la doc officielle sur @graphql-tools
- Exemple avec Apollo

# DIRECTIVES PERSONNALISÉES.

Une directive énumère les endroits du schéma où elle peut être utilisée. Par exemple sur une requête :

- Sur le type Query
- Sur le type Mutation
- Sur un champ
- Sur un fragment

Sur un schéma :

- Sur un champ ou sur son argument
- Sur une interface
- Sur une union
- Sur une énumération

# LES DATASOURCES.

Apollo propose une solution clef en main pour gérer les appels à des service externes (Rest, base de données) dans nos resolvers.

Les `DataSource` gèrent automatiquement :

- Le cache
- La déduplication des requetes
- La gestion des erreurs

Ce sont des classes de service qui seront mis à disposition des resolvers dans le contexte d'éxécution.

# LES DATASOURCES.

## CAS D'UNE API REST

Si on veut pouvoir faire proxy dans notre API pour une certaine partie d'un certain type, il est possible d'utiliser un datasource rest. Exemple :

```
import { RESTDataSource } from "apollo-datasource-rest";

class MoviesAPI extends RESTDataSource {
  constructor() {
    super();
    this.baseURL = "https://movies-api.example.com/";
  }

  async getMovie(id) {
    return this.get(`movies/${id}`);
  }

  async getMostViewedMovies(limit = 10) {
    const data = await this.get("movies", { per_page: limit, order_by: "most_viewed" });
    return data.results;
  }
}
```

# LES DATASOURCES.

## CÔTÉ RESOLVERS

L'utilisation d'un datasource peut faciliter grandement la récupération de données externes.

```
const resolvers = {
  Query: {
    movie: async (_, { id }, { dataSources }) => dataSources.moviesAPI.getMovie(id),
    mostViewedMovies: async (_, __, { dataSources }) =>
dataSources.moviesAPI.getMostViewedMovies()
  },
};

const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources: () => {
    return {
      moviesAPI: new MoviesAPI(),
    };
  },
});
```

# LES DATALOADER.

- Les appels à différents services réalisés par GraphQL afin d'honorer une requête ne doivent pas être trop nombreux
- Un appel unitaire par objet peut dégrader les performances

Prenons l'exemple d'une liste d'adresses avec des coordonnées GPS :

```
type Address { name : String coordinates : GPS }
type GPS { lon : Float lat : Float }
type Query { addresses : [Address] }
```

Deux APIs sont exploitées dans les resolvers :

- Une pour récupérer une liste d'adresses appelée à la résolution du champ adresses
- Une pour récupérer les coordonnées GPS d'une adresse appellée à la résolution du champ coordinates

Le nombre d'appels générés pour une requête retournant n villes est donc de n + 1, ce qui n'est pas performant.

# LES DATALOADER.

Les DataLoaders s'utilisent conjointement aux DataSources. Il faut cependant que les fournisseurs des APIs offrent des services de type bulk pour que ça fonctionne.

> Attention, les dataloader sont peu efficace avec REST car fonctionnent très mal avec le cache HTTP.
>
> À réserver aux requêtes HTTP non cachable (POST, PUT, DELETE) ou aux DataSource de base de données.

# LES DATALOADER.

Exemple d'implémentation dans un DataSource

```
class GpsDataSource extends RESTDataSource {
  coordinatesLoader = new DataLoader(async (names) => {
    return this.get("coordinates", {
      names: names.join(","),
    });
  });

  async find(name) {
    return this.coordinatesLoader.load(name);
  }
}
```

# COMPOSITION DE SCHÉMA.

GraphQL permet de facilement construire une API par dessus d'autres. L'approche classique se résume dans l'ajout d'une dépendance à :

- l'enrichissement du schéma
- l'ajout d'un resolver

On peut rapidement avoir une API qui devient beaucoup trop grosse. Apollo permet de déclarer des schémas distant et de les fusionner !

Pour cela il est possible d'utiliser toujours `@graphql-tools` et ses différentes fonctions utilitaires mais on peut aussi utiliser *Apollo Fédération* qui offre cette fonctionnalité.