

# GraphQL

EFREI 2023-24

# A propos de moi

- Jérôme Senot [js@stairwage.com](mailto:js@stairwage.com)
- Co-fondateur & CTO de Stairwage  
*Digitalisation des acomptes sur  
salaire & bien-être financier des  
salariés*  
[www.stairwage.com](http://www.stairwage.com)
- Plusieurs API GraphQL sur  
React/Node avec Apollo depuis 2017












# Batching des requêtes

- On peut envoyer en une seule « requête HTTP » plusieurs « requêtes GraphQL »
- Permet de mettre automatiquement en buffer les requêtes émises par les composants d'un front client, puis de les envoyer toutes en une fois au serveur
- Réduit la charge côté serveur, tout en conservant la récupération des données de chaque composant front au niveau du composant

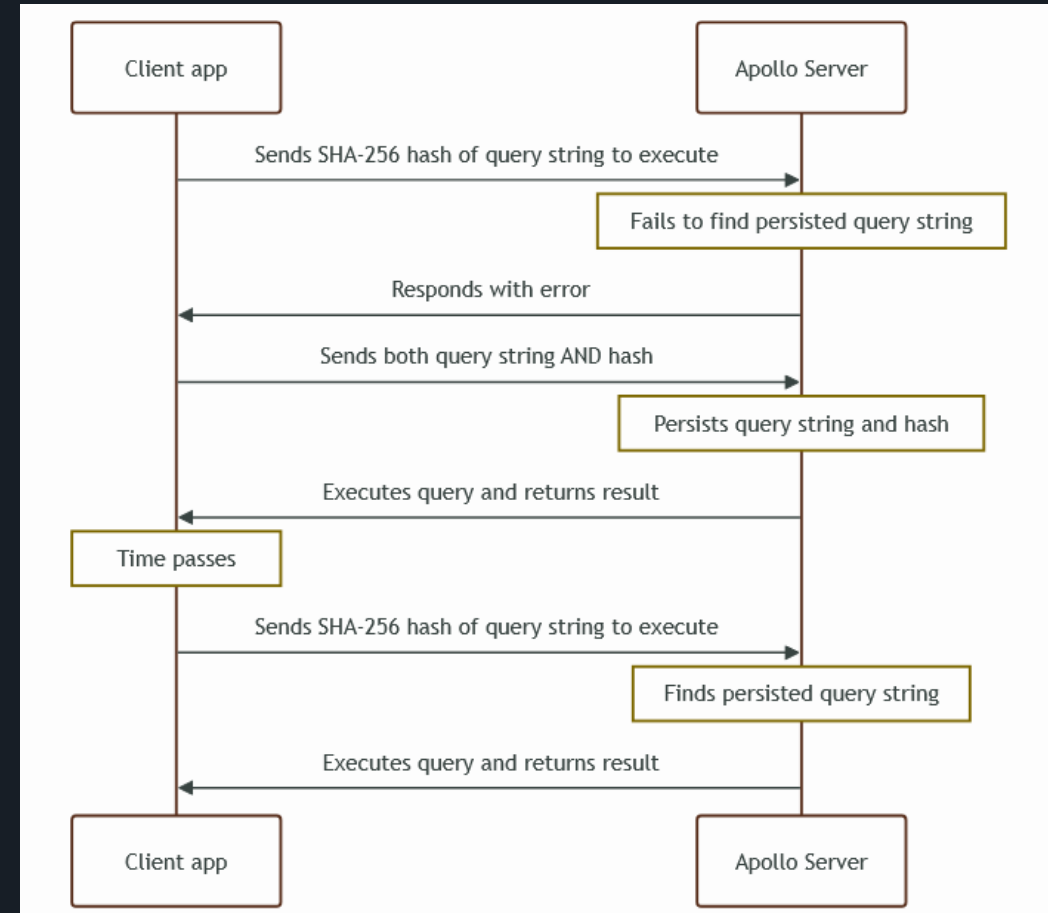
```
query first {  
  hello  
}  
query second {  
  world  
}
```



```
[  
  {  
    query: QUERY_1,  
    variables: {},  
  }, {  
    query: QUERY_2,  
    variables: {},  
  },  
]
```

# Cache des requêtes : Automatic Persisted Queries (APQ)

- Le client doit préciser à chaque requête les données qu'il veut  
➔ beaucoup de répétitions
- Pour optimiser :
  - On peut envoyer seulement un hash du document graphql et les variables
  - Si le serveur ne connaît pas la query associée au hash, il la demande au client en renvoyant une erreur spécifique
  - Sinon il y répond directement





# Caching & batching des accès à la base de données : pistes d'optimisations

- Une requête peut demander ou nécessiter la récupération dans la BDD de :
  - Plusieurs objets du même type
  - Plusieurs fois le même objet
- Plusieurs optimisations sont possibles :
  - On peut regrouper les appels pour des objets du même type
  - On peut éviter de récupérer plusieurs fois les mêmes données

```
query allMoviesWithTheirProducer {  
  movies {  
    id  
    title  
    producer {  
      id  
      name  
      movies {  
        id  
        title  
      }  
    }  
  }  
}
```

```
const doc1 = await collection.findOne({ _id: id1 })  
const doc2 = await collection.findOne({ _id: id2 })  
const doc3 = await collection.findOne({ _id: id3 })
```

```
const docs = await collection.findOne({ _id: { $in: [id1,  
id2, id3] } }).toArray();  
const [doc1, doc2, doc3] = [id1, id2, id3].map(id =>  
docs.find((d) => d._id.equals(id)));
```

# Caching & batching des accès à la base de données : le module DataLoader

- On configure pour chaque type d'objet une fonction qui prend une liste d'ids et renvoie les objets
- On passe par le module DataLoader pour récupérer chaque objet
- Tant qu'il y a du code JS à exécuter, le DataLoader mettra les récupérations en attente, puis récupèrera tout d'un coup
- Les résultats sont gardés en mémoire en cas de seconde récupération d'un même objet
- ⚠ il faut invalider le cache lors de mutations et dans les subscriptions

```
const DataLoader = require('dataloader');

// On ajoute un loader au contexte de la requête
queryContext.producerLoader = new DataLoader((ids) => {
  const producers = await Db.getCollection('producers')
    .find({ _id: { $in: ids } }).toArray();
  ids.map((id) => producers.find((d) => d._id.equals(id)));
});

// On utilise le loader dans les resolvers
const resolvers = {
  Query: {
    getProducer: async (_, { input: { id } }, {
      producerLoader }) =>
      await producerLoader.load(new ObjectId(id)),
    //...
  },
  Movie: {
    producer: async ({ producerId }, { producerLoader }) =>
      await producerLoader.load(new ObjectId(producerId)),
  },
};
```



# Cache des resolvers

- Plutôt que d'appeler un resolver, on peut configurer le serveur pour utiliser une donnée précédemment calculée
- 2 configurations possibles :
  - Via le schéma
  - Lors de la 1<sup>ère</sup> exécution du resolver

```
type Post @cacheControl(maxAge: 240) {  
  id: ID!  
  title: String  
  author: Author  
  votes: Int @cacheControl(maxAge: 30)  
  comments: [Comment]  
  readByCurrentUser: Boolean! @cacheControl(maxAge:  
10, scope: PRIVATE)  
}
```

```
import { cacheControlFromInfo } from '@apollo/cache-control-types';  
const resolvers = {  
  Query: {  
    post: (_, { id }, _, info) => {  
      const cacheControl = cacheControlFromInfo(info)  
      cacheControl.setCacheHint({ maxAge: 60, scope: 'PRIVATE' });  
      return find(posts, { id });  
    },  
  },  
};
```

# Cache des réponses : Content Delivery Network (CDN)

- On peut éviter complètement le traitement de la requête par le serveur grâce à un CDN
- En fonction du cache configuré pour les resolvers, le serveur peut ajouter une entête « Cache-Control » à la réponse, qui sera utilisée par le CDN
-  les requêtes POST ne seront pas récupérées du cache :
  - Il faut configurer le client et le serveur pour utiliser des requêtes GET
  -  la taille des requêtes GET est souvent limitée par le navigateur  
→ utiliser le mécanisme APQ et configurer le client pour n'utiliser des requêtes GET que lorsque seul le hash est envoyé

# Attaque Denial-of-Service (DoS) : exemples

- Le client peut demander ce qu'il veut, il peut demander BEAUCOUP de choses en une seule requête et ainsi saturer le serveur

```
query dos {  
  m1: movies {  
    id  
  }  
  m2: movies {  
    id  
  }  
  # ...  
}
```

```
[  
  {  
    "query": "...",  
    variables: {}  
  },  
  {  
    "query": "...",  
    variables: {}  
  },  
  ...  
]
```

```
query dos { # 5 movies by 1 producer  
  movies { # 5  
    producer {  
      movies { # 5 x 5 = 25  
        producer {  
          movies { # 5 x 25 = 125  
            id  
          }  
        }  
      }  
    }  
  }  
}
```

# Attaque Denial-of-Service (DoS) : protection par limite de complexité

- Interdire ou limiter le nombre de requêtes par batch
- Limiter la complexité de chaque requête

```
import { getComplexity, simpleEstimator } from 'graphql-query-complexity';

// Avant de traiter la requête (cf didResolveOperation), on vérifie sa complexité
const complexity = getComplexity({
  schema,
  query,
  variables,
  estimators: [simpleEstimator({ defaultComplexity: 1 })],
});
if (complexity >= MAXIMUM_COMPLEXITY) {
  throw new GraphQLError('Query too complex');
}
```

# Attaque Denial-of-Service (DoS) : protection par liste de requêtes autorisées (persisted queries)

- N'autoriser que certaines requêtes en configurant une liste de leur « hash »
  1. Générer la liste de toutes les requêtes utilisées par tous les clients possibles  
➔ API fermée uniquement
  2. Calculer le hash de chaque requête
  3. Configurer dans le serveur la liste des requêtes avec document et hash  
⚠ garder dans la liste les requêtes des versions précédentes des clients
  4. Le client ne fournit que le hash de ses requêtes (on évite le 1<sup>er</sup> aller-retour de l'APQ)
  5. Si le hash est inconnu du serveur, il refusera la requête

# Attaque par mutations nombreuses

- Le client peut demander ce qu'il veut, il peut déclencher BEAUCOUP de mutations en une seule requête et ainsi tester des combinaisons
- Même protection que pour DoS

```
query {  
  m1: login(email: "a@b.c", password: "123") {  
    userId  
  }  
  m2: login(email: "a@b.c", password: "456") {  
    userId  
  }  
  # ...  
}
```

```
[  
  "query":  
    "query{\nlogin(email:\n\"a@b.c\n\",password\n:\n\"123\n\") { \nuserId\n}\n}",  
  "query":  
    "query{\nlogin(email:\n\"a@b.c\n\",password\n:\n\"456\n\") { \nuserId\n}\n}",  
  ...  
]
```



# Attaque par mutations simultanées

- Le client peut demander ce qu'il veut, il peut demander plusieurs choses exactement au même instant
- Protection : être particulièrement vigilant dans la gestion des requêtes simultanées

```
[
  {
    "query":
      "mutation{\nvalidateTOTP(authId:\"f45q34df3q4dsf4f35\",smsCode:
        \"1234\"){\n...onValidTOTPPayload{\nauthToken\n}\n}\n}"
    },
    {
      "query":
        "mutation{\nvalidateTOTP(authId:\"f45q34df3q4dsf4f35\",smsCode:
          \"5678\"){\n...onValidTOTPPayload{\nauthToken\n}\n}\n}"
    },
    ...
  ]
  [
    { "data": { "validateTOTP": {} } },
    {
      "data": {
        "validateTOTP": {
          "authToken": "ze5g4qegqg46r4g"
        }
      }
    },
    ...
  ]
]
```



```
export const resolvers = {
  Mutation: {
    validateTOTP: async (_, { authId, smsCode }) => {
      const auth = await db.auths
        .findOne({
          _id: new ObjectId(authId),
          remainingAttempts: { $gt: 0 },
        });
      if (auth.smsCode !== smsCode) {
        await db.auths
          .updateOne(
            { _id: new ObjectId(authId) },
            { $inc: { remainingAttempts: -1 } },
          );
        return { __typename: 'InvalidTOTPPayload' };
      }
      return {
        __typename: 'ValidTOTPPayload',
        authToken: auth.authToken,
      };
    },
  },
};
```



# Précisions & corrections

- Formatter vs linter :
  - eslint abandonne ses règles de mise en forme
  - Cf <https://eslint.org/blog/2023/10/deprecating-formatting-rules/>
- Configuration TypeScript : activez l'option "strict": true
- Plugin codegen et plus généralement pour utiliser TS dans des fichiers de configuration :
  - Après avoir importé 'tsx/preflight' il est possible d'importer des modules TS et même des ESM depuis des CJS (supporté par tsx)
  - On peut donc créer un fichier JS qui import tsx/preflight et ensuite réexporter le contenu d'un fichier TS
- Node 22 supporte de manière expérimentale le require d'ESM :
  - Il faut utiliser le flag --experimental-require-module
  - Uniquement pour les modules sans top-level await
  - Cf <https://nodejs.org/api/modules.html#loading-ecmascript-modules-using-require>
- Pour faire tourner un projet create-react-app dans un sous-dossier il y a pas mal de modification à faire dans la configuration générée  
→ n'hésitez pas à utiliser un outil de monorepo comme nx cf <https://nx.dev/nx-api/react>

# Librairies complémentaires/alternatives

- Côté serveur :

- NestJS

- Permet de structurer le code du serveur : controllers / modules / components / services / tests / ...
    - S'appuie largement sur des décorateurs JS pour générer automatiquement le schéma GraphQL à partir du code JS ➔ « code first » (vs « schema/DSL first »)

- Yoga

- Alternative à @apollo/server

- Prisma

- Sorte d'ORM (Object-Relational Mapping) pour accéder à une BDD quelconque (SQL, MongoDB ...) via des méthode JS générées spécifiquement
    - Propre langages de modélisation des données (*schema.prisma*) et de requêtes :  
`await prisma.user.findMany({ include: { posts: true } })`
    - Peut être utilisé comme n'importe quel autre ORM (Sequelize, Mongoose...) dans les resolvers pour interagir avec la BDD

- Côté client :

- URQL

- Alternative à @apollo/client

# Exemples de « code first »

```
import {
  ObjectType,
  Field,
  ID,
  buildSchemaSync,
} from 'type-graphql';

@ObjectType()
export class User {
  @Field()
  email: string

  @Field((type) => String, { nullable: true })
  name?: string | null
}

@Resolver(User)
export class UserResolver {
  @Query((returns) => [User], { nullable: true })
  async allUsers() {
    return prisma.user.findMany()
  }
}

const schema = buildSchemaSync({
  resolvers: [PostResolver, UserResolver],
});
```

```
import {
  queryType,
  objectType,
  makeSchema
} from '@nexus/schema';

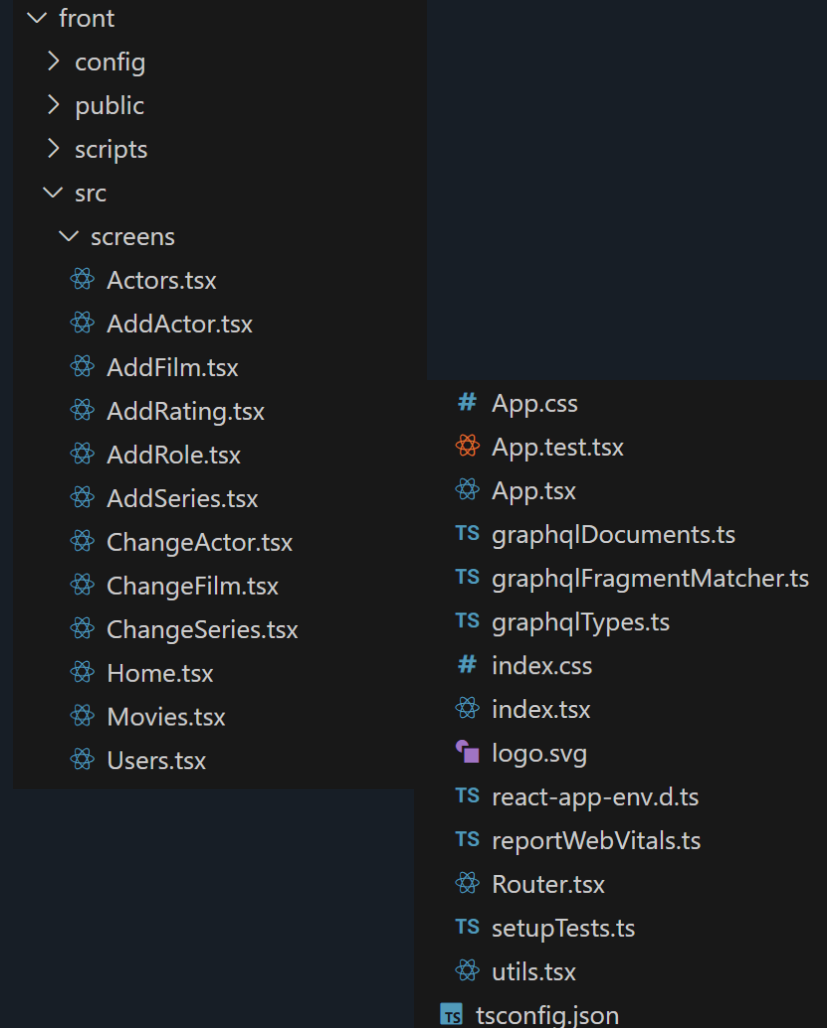
const User = objectType({
  name: 'User',
  definition(t) {
    t.string('email');
    t.string('name', { nullable: true });
  }
});

const Query = queryType({
  definition(t) {
    t.list.field('allUsers', {
      type: 'User',
      resolve: () => prisma.user.findMany()
    });
  }
});

const schema = makeSchema({
  types: [User, Query]
});
```

# Revue du code du client

- Générations avec codegen des :
  - Documents
  - Types
  - « possible types » via le plugin « fragment matcher »
- Configuration d'Apollo client
- Système de navigation basique
- Ecran type avec useQuery
- Mutation dans un évènement
- Mise à jour de l'interface après une mutation
- Apollo Client Devtools




# Authentification & mise à jour instantanée des notes

## 1. Authentification :

- Ajouter aux utilisateurs un champ mot de passe
- Sessions : id du user, token, fin de validité, date de dernière utilisation
- Formulaire de connexion : vérification du mot de passe, création d'une session, ajout d'un cookie
- Authentification de chaque requête : vérification validité et utilisée depuis moins de 1h
- Erreur GraphQL en cas de session expirée et navigation vers le formulaire de connexion

## 2. Mise à jour instantanée des notes :

- Ajouter une subscription au schéma pour la modification des notes d'un film
- Ajouter à Apollo server et Apollo client la configuration nécessaire cf <https://www.apollographql.com/docs/react/data/subscriptions#websocket-setup>
- Implémenter le resolver de la subscription en utilisant `db.collection.watch()`  
 on ne peut pas récupérer le contenu d'un document supprimé → si besoin, changer un champ contenant les valeurs qui nous intéressent avant, et garder ce changement en mémoire dans la subscription
- Ajouter un nouveau token aux sessions à récupérer lors de la connexion et à utiliser pour l'authentification avec le websocket
- Utiliser `useSubscription` dans l'écran de la liste des films pour mettre à jour automatiquement le cache et donc les données affichées

# Protection DoS & DataLoader

## 3. Protection DoS :

- Installer `graphql-query-complexity`
- Dans la configuration du server ajouter un plugin avec `didResolveOperation`
- Y calculer la complexité de la requête et lever une erreur si besoin
- Désactiver le batching de requêtes

## 4. DataLoader :

- Installer `dataloader`
- Ajouter au contexte un objet avec un dataloader pour chaque collection
- Utiliser les dataloaders du contexte dans les resolvers plutôt que des `.find` ou `.findOne` par sur le champ `_id`
- Ajouter au contexte une fonction pour vider le cache de tous les dataloaders :
  - Wrapper les resolvers de mutations pour l'appeler juste avant de renvoyer leur résultat
  - L'appeler aussi avant de renvoyer une valeur dans les subscriptions



