

GraphQL

EFREI 2023-24

A propos de moi

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



Subscriptions

- Schéma :

```
type Subscription {  
  newAnimal: Animal!  
}
```

- Document :

```
subscription {  
  newAnimal {  
    name  
    length  
    wingsCount  
  }  
}
```

- Pas de protocole imposé dans les spécifications, plusieurs implémentations possibles :

- WebSocket :

- [deprecated] subscriptions-transport-ws

- ➔ <https://github.com/apollographql/subscriptions-transport-ws/blob/51270cc7dbaf09c7b9aa67368f1de58148c7d334/PROTOCOL.md>

- graphql-ws

- ➔ <https://github.com/enisdenjo/graphql-ws/blob/master/PROTOCOL.md>

- ...

- HTTP chunked multipart : plusieurs réponses HTTP envoyées successivement en réponse à une requête

- ➔ <https://github.com/graphql/graphql-over-http/blob/main/rfcs/IncrementalDelivery.md>

```
const resolvers = {  
  Subscription: {  
    newAnimal: {  
      subscribe: async function* () {  
        while (true) {  
          const newAnimal =  
            await waitForNewAnimal();  
          if (newAnimal === null) return;  
          yield newAnimal;  
        }  
      },  
    },  
  },  
  newHuman: {  
    subscribe: () =>  
      pubsub.asyncIterator(  
        ['HUMAN_CREATED']  
      ),  
  },  
},  
};
```

Custom scalars : côté serveur

```
# Schéma GraphQL
scalar Date
```

```
type Author {
  name: String!
  date: Date
}
```

```
# Configuration codegen pour TypeScript
generates:
```

```
./serverTypes.ts:
  config:
    scalars:
      Date: Date
```

```
import { GraphQLScalarType, Kind } from 'graphql';

const dateScalar = new GraphQLScalarType({
  name: 'Date',
  description: 'Date custom scalar type',
  serialize(value) { // How to serialize data from resolvers
    if (value instanceof Date) {
      return value.getTime();
    }
    throw Error('GraphQL Date Scalar serializer expected a `Date` object');
  },
  parseValue(value) { // How to parse values in received variables
    if (typeof value === 'number') {
      return new Date(value);
    }
    throw new Error('GraphQL Date Scalar parser expected a `number`');
  },
  parseLiteral(ast) { // How to parse values embedded in received documents
    if (ast.kind === Kind.INT) {
      return new Date(parseInt(ast.value, 10));
    }
    return null;
  },
});
```


Custom scalars : côté client

```
# Schéma GraphQL
type Query {
  nextDay(curDay: Date!): Date!
}

# Document GraphQL
query ($today: Date!) {
  withVariable: nextDay(curDay: $today)
  embedded: nextDay(curDay: 1713691884199)
}

# Configuration codegen pour TypeScript
generates:
  ./clientTypes.ts:
    config:
      scalars:
        Date: Date
```

```
new InMemoryCache({
  typePolicies: {
    Query: {
      fields: {
        nextDay: {
          merge: (existing, incoming) => {
            if (incoming === null || incoming === undefined) {
              return incoming;
            } else if (typeof incoming !== 'number') {
              throw new Error(`Expected number but got: ${typeof incoming}`);
            } else {
              return existing instanceof Date && existing.valueOf() === incoming
                ? existing
                : new Date(incoming);
            }
          },
        },
      },
    },
  },
});
```

- Séri­a­li­sa­tion des variables :

- Apollo Client : .toJSON()
→ dates en ISO "2024-04-21T08:06:55.787Z"
- Autre librairie ?

- Recommandations de types à utiliser :

- Ajouter un type Long : type TS back number, type TS front number, type JSON number, type AST Kind.Int
- Ajouter un type Date : type TS back Date, type TS front number, type JSON number, type AST Kind.Int

Directives

- Permet de documenter le code GraphQL (schéma et documents)
- Permet de modifier les sorties de codegen (ex: @defer)
- Permet d'indiquer des traitements côté serveur (ex: @skipIfUnauthorized)

```
# Schéma GraphQL
type ExampleType {
  oldField: String @deprecated(reason: "Use `newField`.")
  newField: String
}
```

```
# Document GraphQL
query ($isTest: Boolean!) {
  experimentalField @include(if: $isTest)
  nonExperimentalField @skip(if: $isTest)
}
```

Directives personnalisées

```
# Schéma GraphQL
directive @forceCase(uppercaseOtherwiseLowercase: Boolean!) on FIELD_DEFINITION | OBJECT

type Query {
  hello: String @forceCase(uppercaseOtherwiseLowercase: true)
}
```

- Dans le schéma :

- SCHEMA
- SCALAR
- OBJECT
- FIELD_DEFINITION
- ARGUMENT_DEFINITION
- INTERFACE
- UNION
- ENUM
- ENUM_VALUE
- INPUT_OBJECT
- INPUT_FIELD_DEFINITION

- Dans les documents :

- QUERY
- MUTATION
- SUBSCRIPTION
- FIELD
- FRAGMENT_DEFINITION
- FRAGMENT_SPREAD
- INLINE_FRAGMENT

Implémentation avec Apollo Server : on parcourt le schéma et on modifie les resolvers concernés

➔ Exemple : <https://github.com/apollographql/docs-examples/blob/main/apollo-server/v4/custom-directives/upper-case-directive/src/index.ts>

Erreurs

- Peuvent survenir à différents niveaux :
 - Valeur de variable non sérialisable
 - ⚠ *conformité au schéma non vérifiée en runtime côté front, heureusement il y a TypeScript*
 - Erreur réseau lors de l'envoi au serveur
 - Document ou variables non compatibles avec le schéma
 - Erreur dans l'initialisation du contexte du traitement de la requête
 - Resolver non configuré ou qui renvoie undefined
 - Erreur lors de l'exécution d'un resolver
 - Valeur renvoyée par un resolver incompatible avec le schéma
 - Erreur réseau lors de la réception par le client
 - ⚠ *conformité de la réponse au schéma non vérifiée en runtime côté front, heureusement il y a TypeScript*

Erreurs GraphQL

- Erreurs côté client ou réseau à gérer côté front comme pour un fetch classique
- En plus ou à la place du champ data, présence d'un champ errors dans la réponse
- Une erreur doit contenir un message
- Possibilité de passer une extension à chaque erreur pour un traitement spécifique côté client :
 - Session expirée
 - Version du client trop ancienne
 - CGU à valider
 - Accès refusé
 - ...
- Possibilité de renvoyer en plus une erreur HTTP (40x, 50x ...)
- Avec Apollo Server :
 - Un resolver peut lever un erreur GraphQLError qui sera transmise au client.
 - On peut écouter et/ou modifier toutes les erreurs renvoyées par le serveur pour les formater et les ajouter aux logs

```
{
  "data": {
    "createProduct": null
  },
  "errors": [
    {
      "path": [
        "createProduct"
      ],
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "Product already exists",
      "extensions": {
        "code": "PRODUCT_ALREADY_EXISTS",
        "timestamp": "Fri Feb 9 14:33:09 UTC 2018",
        "anyOtherKey": "is permitted here"
      }
    }
  ]
}
```

Authentification

- Gérée au niveau HTTP comme avec une API REST
- Possibilité d'ajouter n'importe quelles entêtes aux requêtes HTTP :
 - Cookie (et Set-Cookie dans la réponse)
 - Authorization : ⚠ à privilégier car bien gérer dans les outils (autorisée, non divulguée...)
 - User/Password : « Basic bG9naW46cGFzcw== »
 - Token : « Bearer qdqfd5qs46df5d4sfq654df6 »
- Types d'authentification :
 - Clé API : la passer comme token dans l'entête Authorization
 - JSON Web Token (OAuth) : à passer comme token dans l'entête Authorization :
 - vérification de la signature et validité du JWT à chaque requête
 - application des autorisations qu'il contient
 - opérations cryptographiques, mais pas d'accès BDD nécessaire
 - permet de déléguer facilement l'authentification à un tiers
 - Token de session :
 - privilégier la transmission sous forme de cookie « server only » (pour éviter la récupération par un script injecté), mais ajouter une entête supplémentaire avec un autre secret pour éviter les attaques CSRF

Agrégation de plusieurs schémas

- Plusieurs niveaux :

1. « schema merging » :

- séparation de la définition du schéma entre plusieurs fichiers

2. « schema federation » :

- maintien d'une cohérence entre plusieurs sous-schémas
- enrichissement de chaque sous-schéma avec des directives en vue d'une utilisation conjointe avec d'autres sous-schémas
- « router » pour traiter une requête en appelant les sous-API et en regroupant les résultats dans la réponse au client

3. « schema stitching » :

- une « gateway » est configurée pour faire le lien entre différents schémas qui n'ont pas forcément conscience d'être utilisés conjointement
- elle génère un nouveau schéma dont n'ont pas conscience les sous-schémas
- elle traite les demandes en utilisant sa propre configuration pour faire le lien entre les différents types d'objets de chaque sous-schéma

```
# Config codegen
schema: ./graphqlSchema/*.gql
generates:
  ...
```


Formatter

- Pourquoi ?
 - Gain de temps (activer le formatage à l'enregistrement)
 - Application des meilleures pratiques les plus répandues
 - Stabilité du code entre les différents commits
 - Code homogène plus facile à lire
- Principaux outils JS/TS :
 - Prettier : la référence historique
➔ <https://prettier.io/playground/>
 - Biome : en Rust, 35x plus rapide que Prettier
➔ <https://biomejs.dev/playground/>

```
{
  "$schema":
    "./node_modules/@biomejs/biome/configuration_schema.json",
  "formatter": {
    "indentWidth": 2,
    "indentStyle": "space",
    "lineWidth": 100,
    "enabled": true,
    "formatWithErrors": true
  },
  "javascript": {
    "formatter": {
      "trailingComma": "all",
      "quoteStyle": "single",
      "jsxQuoteStyle": "single",
      "quoteProperties": "asNeeded"
    }
  },
  "files": {
    "maxSize": 100000000000,
    "ignore": [
      "**/node_modules/**/*",
      "**/dist/**/*"
    ]
  }
}
```

Linters

- Pourquoi ?
 - Détection d'erreurs potentielles, par exemple :
 - variables non utilisées
 - dépendances cycliques
 - console.log(...)
 - Gain de temps avec les « fix » automatiques (ex: dépendances des hooks React)
 - Application des meilleures pratiques les plus répandues
 - Code homogène plus facile à comprendre
- Principaux outils JS/TS :
 - ESLint : la référence historique avec de nombreux plugins
→ <https://eslint.org/play/>
 - Biome : en Rust, plus rapide mais moins riche

```
{
  "$schema":
    "./node_modules/@biomejs/biome/configuration_schema.json",
  "linter": {
    "enabled": true,
    "rules": {
      "complexity": {
        "noUselessFragments": "error",
        "noForEach": "off"
      }
    },
    "suspicious": {
      "noShadowRestrictedNames": "off",
      "noExplicitAny": "off",
      "noConsoleLog": "off",
      "noRedundantUseStrict": "error",
      "noAssignInExpressions": "error",
      "noConfusingVoidType": "off",
      "noRedeclare": "off",
      "noApproximativeNumericConstant": "error",
      "noMisrefactoredShorthandAssign": "error"
    }
  },
  ...
},
...
"files": {
  "maxSize": 100000000000,
  "ignore": [
    "**/node_modules/**/*",
    "**/dist/**/*"
  ]
}
}
```

ESM vs CJS

- Historique :

1. pas d'import de code possible
2. arrivée de Node.js avec le CommonJS
3. arrivée d'un système d'imports (ES modules) dans les spéc. de JS
4. Node.js supporte maintenant les ESM

- Intérêt des ESM :

- import asynchrone possible (typiquement d'un fichier distant)
- « await » au plus haut niveau possible
- chemins statiques pour une analyse de code facilitée (ex: détection des modules utilisés à bundler)

- Problème avec Node :

- On ne peut pas importer un module ESM (avec un potentiel top-level await) depuis un CJS (avec un require(...)) synchrone, l'inverse est possible.

- Solutions :

- Ne pas utiliser les modules ESM, mais de plus en plus de librairies sont 100% ESM
- Utiliser la fonction async « await import(...) », mais impossible au plus haut niveau ni dans une fonction synchrone
- Passer tout son code en ESM et utiliser l'import par défaut quand on import du CJS


⚠ Attention aux lib CJS qui n'exportent que dans un « exports.default = ... » car, depuis un ESM, Node écrase l'import « default » par « exports » :

```
import nodeDefImport from "cjsLib"; const defImport = nodeDefImport.default;
```

```
// CJS
exports.twoPi = void 0;
const constants_1 = require("./constants");
exports.twoPi = constants_1.valueOfPi * 2;
```

```
// ESM
import { valueOfPi } from "./constants";
export const twoPi = valueOfPi * 2;
```

TypeScript : présentation

- Pourquoi ?
 - Outils de développement : autocomplétion, refactoring, documentation...
 - Détection d'erreurs en codant : mauvais types, cas particuliers non gérés, faute de frappe, évolution d'un module externe...
- Le code TS n'est pas exécutable directement, il doit être « transpilé » en JS pour enlever le surplus TS
 - très rapide, nécessite juste un parsing des fichiers séparément
 -  TS n'a aucun impact sur le runtime !
- Le code TS peut être vérifié dans l'éditeur et éventuellement à la transpilation
 - analyse poussée du code à la recherche d'incompatibilités entre les types
- Les types peuvent être extraits dans des « .d.ts »
 - utilisé par les librairies pour mettre à disposition leurs types TS en plus de leur code transpilé en JS
- On peut transpiler à la volée avant de l'exécuter du code TS avec des outils comme tsx ou ts-node

TypeScript : les bases

```
// Typage des éléments JS
const a: number = 1;
const b = 'yes'; // b est de type string
function f(b: string): boolean {
  return b !== '';
}
const g: (b: string) => boolean = f;
```

```
// Définition de types
type T =
  | undefined
  | null
  | number
  | string[]
  | 'specific string'
  | `template string ${'with nested' | 'string'}`;
type S = { p: Date; q?: object };
type U = S & { readonly r: string };
interface I {
  readonly p: I;
}
interface J extends I {
  q: string;
}
```

```
// Types génériques
type Pair<T1, T2 extends string, T3 = 'default'> = {
  first: T1;
  second: T2 | T3;
};
function h<Type>(a: Type): Type {
  return a;
}
type H = <Type>(a: Type) => Type;
```

```
// Types spéciaux
const ts1: any = 1; // Tout peut être assigné à any
const ts11: string = ts1; // any peut être assigné à tout
const ts2: unknown = null; // Tout peut être assigné à unknown
const ts22: null = ts2; // Erreur: unknown ne peut être assigné à rien
const ts3: never = undefined; // Erreur: rien ne peut être assigné à never
```

```
// Mots clés
var v = 'hello' as const; // v est de type 'hello' et non pas string
const w = 1 as 1 | 2 | 3; // ⚠ w est de type 1 | 2 | 3
const x = 1 as unknown as string; // ⚠ x est de type string
const z = 1 satisfies string; // Erreur: 1 n'est pas de type string
```

```
// Types utilitaires
type S1 = Pick<S, 'p' | 'q'>;
type I1 = Omit<J, 'q'>;
type T1 = Extract<T, string[]>;
type T2 = Exclude<T, string>;
type T3 = NonNullable<T>;
type P = Parameters<H>;
type R = ReturnType<H>;
type P1 = J['q'];
```

TypeScript : configuration

```
{
  "compilerOptions": {
    "lib": ["es2023"], ➔ version des types de base JS (ex: Array.flatMap)
    "target": "es2022", ➔ TS peut convertir du code utilisant les dernières syntaxes JS en du code plus « vieux » (ex: arrow func)
    "module": "NodeNext", ➔ système de modules à utiliser en sortie : commonjs, esnext, nodenext, preserve...
    "moduleResolution": "NodeNext", ➔ où aller chercher un fichier importé ? en particulier pour une lib externe
    "esModuleInterop": false, ➔ comment convertir des « import ... from ... » en « = require(...) »
    "types": ["node"], ➔ types supplémentaires disponibles dépendant du contexte : node, jest, react-native
    "noEmit": true, ➔ lors de l'utilisation de la commande tsc, faut-il juste vérifier les types ou transpiler en plus
    "strict": true, ➔ vérification stricte des erreurs de types
    "allowImportingTsExtensions": true ➔ normalement l'extension dans les imports est utilisée en runtime et doit donc être en .js mais si on utilise un bundler ou si on transpile à la volée, il est moins perturbant d'utiliser des .ts
  }
}
```

- Config pour chaque version de Node : <https://github.com/microsoft/TypeScript/wiki/Node-Target-Mapping>
- Extension VSCode « Pretty TypeScript Errors » pour des erreurs mieux formatées

Amélioration des scripts

- Dossier dédié aux scripts avec propre tsconfig avec :
 - "types": ["node"],
 - la configuration pour Node.js@20 (pour `.replaceAll(...)`)
- Ajouter un script pour lancer le serveur mongo (plutôt qu'un lancement automatique au démarrage du serveur) :
 - Gain de temps au rechargement du serveur
 - Port constant (à choisir: 27017) pour accès facile via Compass
 - Pas de plantage si relance trop rapide
- Mettre le plugin codegen dedans
 - En profiter pour sécuriser l'injection du schéma `String(res).replaceAll('\'', '\\\'').replaceAll(``, '\\`')`
 - Pour éviter que ts-node trouve le bon tsconfig utiliser `tsx/preflight` :
 - Dans la config codegen remplacer l'import de ts-node/loader et remplacer le chemin du plugin par `"./scripts/exportSchemaCodegenPlugin.ts"`
 - tsx gère le require d'un ESM (!), on peut passer le script en ESM (d'où l'extension en .ts) :

```
import type { CodegenPlugin } from '@graphql-codegen/plugin-helpers';
import { plugin as schemaAstPlugin } from '@graphql-codegen/schema-ast';

export const plugin: CodegenPlugin['plugin'] = async (schema, documents, config) => {
  const res = await schemaAstPlugin(schema, documents, config);
  return `export const typeDefs = `#graphql\n` + String(res).replaceAll('\'', '\\\'').replaceAll(``, '\\`') + `;`;
};
```


Amélioration config codegen des resolvers

```
schema: "./graphql/schema/schema.gql"
emitLegacyCommonJSImports: false
generates:
  ./graphql/src/resolvers.ts:
    plugins:
      - add:
          content: |
            export type DeepPartialGraphQLType<T> = T extends unknown
              ? T extends Function | Date | Promise<unknown> | (() => any) | undefined
                ? T
                : T extends Array<infer U>
                  ? Array<undefined | DeepPartialGraphQLType<U>>
                  : T extends ReadonlyArray<infer V>
                    ? ReadonlyArray<undefined | DeepPartialGraphQLType<V>>
                    : T extends object
                      ? { [P in keyof T]?: undefined | DeepPartialGraphQLType<T[P]> }
                      : T
              : never;
      - typescript:
          avoidOptionals:
            field: true
            inputValue: false
            object: true
            defaultValue: false
      - typescript-resolvers:
          defaultMapper: DeepPartialGraphQLType<{T}>
    config:
      useTypeImports: true
      useIndexSignature: false
      immutableTypes: true
      nonOptionalTypename: true
      defaultMapper: Partial<{T}>
  ./graphql/src/schema.ts:
    plugins:
      - "./scripts/exportSchemaCodegenPlugin.cjs"
```

Ecriture du schéma

- Séparation possible en plusieurs fichiers pour schéma et documents
- Outils VSCode :
 - Coloration syntaxique : GraphQL: Syntax Highlighting *by GraphQL Foundation*
 - LSP : GraphQL: Language Feature Support *by GraphQL Foundation*
 - Fichier de configuration à la base du repository :

```
# graphql.config.yml
schema: 'graphql/schema/*.gql'
documents: 'graphql/documents/*.gql'
```
 - Formatter possible : Prettier - Code formatter *by Prettier*
- Codegen auto en mode watch :
 - Installer @parcel/watcher
 - Dans package.json : "generate": "graphql-codegen --config codegen.yml --watch",

MongoDB : principales opérations

```
await collection.insert({
  name: 'John Doe',
  age: 25,
  email: 'john@mail.com',
  friends: [],
});

await collection.find({
  name: 'John Doe',
}).toArray();

await collection.updateMany(
  { name: 'John Doe' },
  {
    $set: { email:
'johnny@mail.com' },
    $increment: { age: 1 },
  },
);

await
collection.findOneAndUpdate(
  { name: 'John Doe' },
  { $push: { friends: 'Billy' } }
);
```

```
await collection.aggregate([
  { $match: { name: 'John Doe' } },
  {
    $lookup: {
      as: 'friendPersons',
      from: 'persons',
      localField: 'friends',
      foreignField: 'name',
      pipeline: [
        { $match: { age: { $gt: 25 } } },
        { $sort: { age: -1 } },
        { $limit: 3 },
        { $project: {
          _id: false,
          email: true,
        } },
      ],
    },
  },
]).toArray();

await collection.updateOne(
  { $expr: { $eq: [
    { $substr: ['$name', 0, 3] },
    'Joh'
  ] } },
  [{ $set: {
    name: { $toLower: '$name' } } },
  ]
);
```

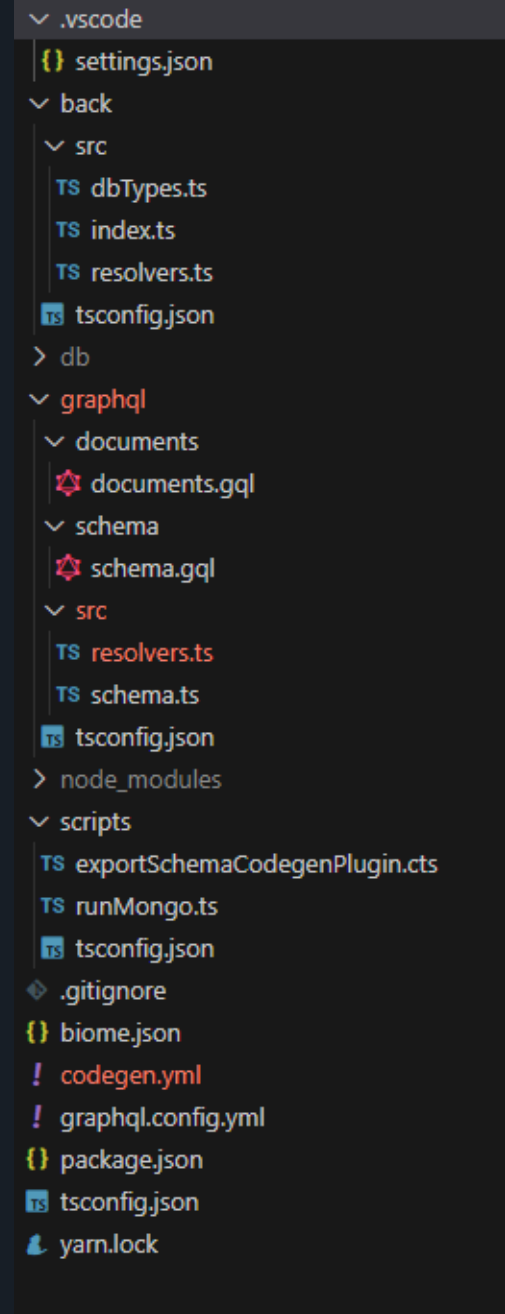
```
const session = client.startSession();
await session.withTransaction(async (session)
=> {
  const john = await collection.findOne(
    { name: 'John Doe' }
  );

  await collection.updateMany(
    { _id: john._id },
    {
      $set: { email: 'johnny@mail.com' },
      $increment: { age: 1 },
    },
    { session }
  );

  if(Date.now() > maxDate)
    throw new Error('Too late');

  await collection.updateOne(
    { _id: john._id },
    { $push: { friends: 'Billy' } }
    { session }
  );
});
```

Revue du code du serveur



React : présentation

- Framework frontend développé par Facebook (Meta) en 2011 et open sourcé en 2013
- Le plus populaire depuis plusieurs années
 - 6x plus utilisé que Vue ou Angular
- Basé sur des composants (« React component ») qui reçoivent des « props » et renvoient un élément (« React element ») :
- JSX/TSX transpilé en `React.createElement` pour un code plus clair :
- Un composant « Root » est rattaché à un `<div>` existant
- Tous les composants sont récursivement convertis (« render ») en arbre d'éléments jusqu'à ce qu'il ne reste que des éléments HTML de base (props correspondant aux attributs) puis le DOM est synchronisé avec.

```
const Intro = ({ msg = 'Hello!' }) =>  
  React.createElement(  
    'div',  
    { id: 'divId' },  
    React.createElement('p', {}, msg),  
  );
```

```
const IntroUsingJSX =  
  ({ msg = 'Hello!' }) => (  
    <div id='divId'>  
      <p>{msg}</p>  
    </div>  
  );
```

React : hooks

- Les composants sont montés (« mount ») lors de leur première apparition dans l'arbre puis démontés lors de leur disparition.
- Des « hooks » permettent d'exploiter cette persistance des composants

```
const [color, setColor] = React.useState('red');  
const onClickHandler = () => setColor('green');
```

```
const colorRef = React.useRef('red');  
if (colorRef.color === 'green') {  
  colorRef.color = 'blue';  
}
```

```
React.useEffect(() => {  
  const timeout = setTimeout(  
    () => setColor('blue'),  
    1000  
  );  
  return () => clearTimeout(timeout);  
}, []);
```

```
const stableComputationResult = React.useMemo(() => {  
  return complexComputation(props.inputValue);  
}, [props.inputValue]);
```

```
const stableHandler = React.useCallback(() => {  
  return console.log("Clicked!");  
}, []);
```

React : performance

- Lorsqu'un composant est rerendu, tous ses enfants le sont aussi, et donc récursivement tous ses descendants
- Un composant est donc rerendu quand :
 - un de ses « state » change
 - son parent est rerendu
- On peut bloquer son rerender en l'encapsulant dans un `React.memo`

⚠ le render ne sera évité que si toutes les props sont inchangées (« === ») y compris ses enfants (prop « children »), on perd la simplicité du JSX...

```
const Comp = React.memo(
  ({ title = 'Hello!', children }) => (
    <div id='divId'>
      <p>{msg}</p>
      {children}
    </div>
  )
);
```

```
const App = () => (
  <Comp title='Hello!'>
    <p>This is NOT a super app!</p>
  </Comp>
);
```

```
const App = () => {
  const details = React.useMemo(
    () => <p>This is a super app!</p>,
    []
  );
  return (
    <Comp title='Hello!'>{details}</Comp>
  );
};
```


Navigation dans une app React

- Le plus simple :
 - Dans un composant à la racine de l'app, utiliser un state du type :
 - Dans son render, choisir l'écran à afficher en fonction de ce state
 - Passer à chaque composant de type « écran » une fonction pour changer ce state
- On peut aussi utiliser un context React, un observable Mobx...
- L'idéal est une librairie comme React Router qui va gérer l'URL affichée par le navigateur et les boutons back/forward

```
type NavigationState =  
  | { route: 'LIST_FILMS' }  
  | { route: 'CHANGE_FILM'; filmId: string }  
  | { route: 'ADD_FILM' }  
  | { route: 'VIEW_FILM'; filmId: string }  
  | { route: 'VIEW_ACTOR'; actorId: string }  
  ...  
;
```

A vos claviers !

1. Finir l'implémentation du serveur
2. Ajouter une application React avec <https://create-react-app.dev>
 1. `npx create-react-app front --template typescript` → `cd front` → `npm run eject`
 2. Remonter d'un niveau :
 1. Mettre le contenu intéressant de `front/package.json` dans celui à la racine → `npm install`
 2. Y ajouter `/front` dans le chemin des scripts et ajouter ce `package.json` dans `/front/config` et `/front/scripts`

```
{ "type": "commonjs" }
```
 3. Ajouter les extensions `.tsx` aux chemins des imports dans les fichiers de `/front/src`
 4. Ajouter dans `/front/tsconfig.json` `"allowImportingTsExtensions": true,`
 5. Dans `/front/config/webpack.config.js` ajouter dans la config `ForkTsCheckerWebpackPlugin` le chemin du `tsconfig` :

```
typescript: { configFile: paths.appTsConfig,
```
 6. Dans `/front/config/paths.js`, corriger les chemins :

```
const appDirectory = path.join(__dirname, '..');  
../package.json  
../node_modules
```
3. Suivre la documentation d'Apollo Client (React) <https://www.apollographql.com/docs/react/get-started>
4. Coder le système de navigation
5. Générer les documents et types (variables et résultat des requêtes) avec codegen
→ <https://the-guild.dev/graphql/codegen/plugins/typescript/typescript-operations>
6. Afficher la liste des films comme écran par défaut en récupérant les données avec un `useQuery`
7. Implémenter l'ajout d'un film avec un `useMutation`
8. Implémenter les autres écrans