

Formation d'ingénieur ISIS

ISIS Capitalist

Mini-projet dans le cadre du cours sur les architectures orientées services.

nicolas.singer@gmail.com



Contenu

1. Présentation du projet	3
Principes du jeu « Adventure Capitalist »	3
Gameplay.....	3
2. Conception du serveur de mondes	9
a. Spécifications de l'API du serveur	9
b. Mise en place de la partie serveur	12
c. Création du monde.....	13
d. Séparation des composants	16
3. Début du travail sur le client web	18
a. Serveur de test	18
b. Création du projet React	18
c. Description du travail attendu	18
d. Création des premiers composants et du service de communication avec le service web..	20
e. Réalisation du layout général	25
f. Eléments utiles de CSS.....	26
g. Quelques méthodes utilitaires	27
4. Actions du joueur	28
a. Démarrage de la production d'un produit	28
b. La boucle principale de calcul du score	29
c. L'achat de produit	31
5. Les managers	33
a. Interface pour lister les managers.....	33
b. Engagement d'un manager	35
c. Afficher un message éphémère pour l'utilisateur.....	36
d. Badger les boutons pour informer le joueur	36
6. Retour coté serveur.....	37
a. Modifier le service web pour qu'il récupère le nom du joueur	37
b. Prendre en compte les actions du joueur	38
c. Appel des interfaces par le client	40
7. Les <i>unlocks</i>	41
a. Affichage des <i>unlocks</i>	41

b.	Prise en compte des <i>unlocks</i> par le client	42
c.	Prise en compte des <i>unlocks</i> par le serveur	42
8.	Les <i>Cash upgrades</i>	43
a.	Affichage des <i>upgrades</i>	43
b.	Prise en compte des upgrades par le client.	44
c.	Transmission des upgrades du client au serveur.	44
d.	Prise en compte des upgrades par le serveur.	44
9.	Gestion des anges.....	45
a.	Gestion des anges coté client.....	46
b.	Gestion des anges coté serveur	46
c.	Prise en compte des anges dans les gains.....	47
10.	Gestion des <i>Angel Upgrades</i>	47
a.	Prise en compte des <i>angel upgrades</i> par le client.	48
b.	Prise en compte des <i>angel upgrades</i> par le serveur.	49
11.	Finalisation et branchement sur un autre monde	49

1. Présentation du projet

L'objectif de ce projet est de programmer un jeu vidéo calqué sur les mécanismes du jeu *free to play* « Adventure Capitalist » de la société Kongregate (Figure 1).



Figure 1 : Copie d'écran du jeu « Adventure Capitalist » de la société Kongregate.

Ce projet se composera d'une partie cliente (l'interface du jeu) et d'une partie serveur (la description du monde et sa persistance). Chaque étudiant devra développer ces deux parties en s'appuyant sur un référentiel commun qui vise à assurer l'interopérabilité de chaque client avec chaque serveur. Pour être plus précis, le jeu d'origine permet au joueur de naviguer dans plusieurs mondes. Ces mondes seront les serveurs développés par chaque étudiant qui devront donc être compatibles avec les clients de chacun.

Principes du jeu « Adventure Capitalist »

« Adventure Capitalist » est un jeu satirique où l'objectif est d'augmenter à l'infini ses revenus en investissant dans la production de produits. Son originalité est de ne proposer aucun challenge, aucune énigme, aucun défi à relever. Vos rares actions et le temps qui passe vous feront automatiquement passer d'un vendeur de limonade à un méga-multi-hyper milliardaire, le tout sur fond de croissance hypnotique de dollars accumulés, les nombres obtenus ne pouvant plus s'exprimer à la fin qu'en multiple de puissances de dix.

Gameplay

Investir dans des produits et lancer leur production

Chaque niveau, que nous appellerons « monde », se compose d'un certain nombre de types de produits qui correspondent aux investissements réalisables. Ces investissements se traduisent par l'achat par le joueur d'un certain nombre d'exemplaires de ces produits.

Chaque type de produit est caractérisé par son temps de production, par le revenu procuré par la production d'un de ses exemplaires, et par le coût d'achat d'un exemplaire supplémentaire. Ce coût d'achat augmente en fonction du nombre d'exemplaires déjà possédé, car il se calcule sous la forme d'un pourcentage d'augmentation par rapport au prix du dernier exemplaire acheté. Ce pourcentage d'augmentation est propre à chaque type de produit.

Par exemple dans le monde « Terre » du jeu d'origine, la production d'une bouteille de limonade se fait en une demi-seconde et rapporte un dollar. Le premier exemplaire d'une telle bouteille coûte 4 dollars et ce coût se voit appliquer une augmentation de 7% à chaque unité supplémentaire. Acheter une deuxième bouteille coûte donc 4,28 ($1.07 * 4$) dollars, une troisième 4,58 ($1.07 * 4,28$) dollars, etc...

Le joueur lance la production d'un type de produit en cliquant sur l'icône qui lui correspond. Quand le temps de production est écoulé, l'argent du joueur se voit augmenter du nombre d'exemplaires du produit détenu multiplié par le revenu de ce type de produit. Ainsi si le joueur possède 4 bouteilles de limonade et qu'il lance leur production, au bout d'une demi-seconde, il aura gagné 4 ($4 * 1$) dollars.

Quand il dispose de la somme suffisante, le joueur peut acheter des exemplaires de produit supplémentaires, en cliquant sur le bouton « Buy » attaché au produit (Figure 2).



Figure 2 : Ici, le joueur possède 25 bouteilles de limonade qui lui rapportent 84 dollars à chaque production. Acheter une bouteille de plus coûte 20,29 dollars.

Pour faciliter la vie du joueur, le jeu propose un bouton qui permet d'acheter plusieurs exemplaires d'un produit d'un coup, par dix, par cent, ou selon la quantité maximale achetable en fonction de l'argent actuel du joueur (Figure 3).

Au fur et à mesure de ses gains, le joueur peut débloquer des produits supplémentaires et ainsi lancer la production de plusieurs types de produit en même temps.

Les managers

Chaque produit dispose d'un manager qui permet d'automatiser sa production. Au début de la partie, ce manager est inactif et peut-être débloqué contre une certaine somme d'argent, ce qui dispense ensuite le joueur de cliquer pour lancer la production de ce produit. Ce qui est intéressant c'est que l'automatisation de la production persiste même quand le joueur n'est plus connecté. Ainsi le compte en banque du joueur augmente automatiquement en fonction du temps qui passe. L'achat d'un manager se fait en cliquant sur le bouton « manager » de l'interface (Figure 4).



Figure 3 : Quand le bouton de quantité d'achat est sur max, l'interface affiche combien le joueur peut acheter d'exemplaire d'un produit (ici 246, pour un coût total de 4,904 milliards).

Les seuils (unlocks)

Le revenu généré par un type de produit peut être boosté lorsque le joueur en obtient une quantité dépassant un certain seuil. Ce *boost* peut prendre la forme d'une augmentation de la vitesse de production ou d'une augmentation du revenu généré par le produit, sous la forme d'un ratio multiplicateur. La liste des seuils attachés à chaque produit est visualisable en cliquant sur le bouton « unlocks » de l'interface (Figure 5).

S'ajoutent à ces seuils par produit, des seuils globaux qui génèrent des bonus supplémentaires quand ils sont atteints par l'ensemble des produits. Par exemple sur la figure 5, on voit qu'atteindre une quantité de 25 pour chaque produit, double la production de chaque produit.

Les Cash Upgrades

Parallèlement aux bonus de seuil, le joueur a également la possibilité d'acheter des « upgrades » qui permettent eux aussi d'augmenter les revenus. Leur liste s'obtient par le bouton « upgrades ». Certains de ces bonus augmentent la production d'un seul produit alors que d'autres peuvent agir sur tous. Notons que certains *upgrades* peuvent également augmenter l'efficacité des anges (voir la prochaine section pour la signification de ces anges).

Les anges

Au fil du jeu le joueur va accumuler un certain nombre d' « Angel Investors ». Ces anges, quand ils sont actifs, offrent chacun un bonus de production de 2%. Par conséquent 50 anges actifs procurent un doublement des revenus ($2 * 50 = 100\%$). Le problème, et finalement le sel du *gameplay*, c'est que les anges obtenus ne deviennent actifs qu'après une remise à zéro (*reset*) de la partie. Ce mécanisme oblige le joueur à repartir de zéro s'il veut bénéficier du bonus apporté par les anges.

Plus précisément, dans le jeu d'origine et sur le monde « terre », le nombre d'anges obtenus est lié aux gains cumulés du joueur depuis qu'il a commencé à jouer, selon la formule :

$$\text{Nombre d'anges} = 150 * \sqrt{\frac{\text{gains cumulés}}{10^{15}}}$$



Figure 4 : Liste des managers avec leur nom, leur coût d'achat et le produit dont ils s'occupent.



Figure 5 : liste des prochains bonus associés à l'obtention d'une certaine quantité d'un type de produit.

A chaque reset de la partie, c'est ce nombre d'anges qui devient actif (moins ceux qui ont été dépensés dans les « Angel Upgrades », voir plus bas).

Cette formule traduit le fait qu'il faut de plus en plus d'argent pour accumuler des anges supplémentaires, mais aussi le fait que les anges persistent entre les *reset* puisque leur nombre dépend des gains accumulés par le joueur lors de chacune de ses parties.

Les Angel Upgrades

Les « Angel Upgrades » ont le même effet que les « Cash Upgrades » à ceci près que leur monnaie d'achat est l'ange au lieu d'être le dollar (Figure 6). Ainsi le joueur peut dépenser un certain nombre de ses anges actifs pour acheter des bonus de production (et parfois une certaine quantité de produits). Attention à bien réfléchir à ce type d'achat, car les anges dépensés sont perdus, et on ne profite plus de leur bonus de 2% de production supplémentaire.

Pour exemple, voici les valeurs de certains paramétrages du monde « terre » du jeu d'origine :

Produit	Revenu initial	Cout du premier exemplaire	Croissance du coût par exemplaire	Temps initial de production	Coût du manager
Limonade	\$1	\$4	7%	0,5 s	\$1 000
Journal	\$60	\$60	15%	3 s	\$15 000
Lavomatic	\$540	\$720	14%	6 s	\$100 000
Pizza	\$4 320	\$8 640	13%	12 s	\$500 000
Donut	\$51 840	\$103 680	12%	24 s	\$1 200 000
Crevette	\$622 080	\$1 244 160	11%	1 min 36 s	\$10 000 000
Hockey	\$7 464 000	\$14 929 920	10%	6 min 24 s	\$111 111 111
Film	\$89 579 000	\$179 159 040	9%	25 min 36 s	\$555 555 555



Figure 6 – Bonus possibles en dépensant des anges pour le monde « terre ».

2. Conception du serveur de mondes

a. Spécifications de l'API du serveur

Comme nous l'avons dit, le logiciel complet sera composé d'une partie serveur et d'une partie cliente. La partie serveur a pour responsabilité de définir le monde proposé à la partie cliente, sous la forme des produits dans lesquels le joueur pourra investir, de leurs caractéristiques (cout, revenu, etc.), de leurs managers, de leurs upgrades et de leurs seuils, etc.

Le serveur aura également pour rôle de conserver la persistance des parties de chaque joueur s'y connectant.

Pour que les serveurs soient compatibles les uns avec les autres (en fait compatibles avec les clients qui s'y connecteront), nous devons définir un référentiel d'interopérabilité qui spécifie les protocoles de communication entre le serveur et les clients, ainsi que le format des messages échangés.

En termes de **protocoles**, nous utiliserons une communication client-serveur basée sur un service web au format GraphQL. Les points d'accès de ce service seront :

getWorld : retourne au client une représentation complète de l'état actuel du monde sous la forme d'une entité de type « monde ».

acheterQtProduit : permet au client de communiquer au serveur qu'une certaine quantité d'un produit est achetée.

lancerProductionProduit : permet au client de communiquer au serveur qu'un produit vient d'être mis en production.

engagerManager : permet au client de communiquer au serveur l'achat du manager d'un produit.

acheterCachUpgrade : permet au client de communiquer au serveur l'achat d'un *Cash Upgrade*.

acheterAngelUpgrade : permet au client de communiquer au serveur l'achat d'un *Angel Upgrade*.

resetWorld : permet au client de demander le *reset* du monde.

Voici le schéma GraphQL que vous devez utiliser :

```
enum RatioType {
  gain
  vitesse
  ange
}

type Palier {
  name: String!
  logo: String
  seuil: Float
  idcible: Int
  ratio: Int
  typeratio: RatioType
  unlocked: Boolean
}

type Product {
```

```

    id: Int!
    name: String
    logo: String
    cout: Float
    croissance: Float
    revenu: Float
    vitesse: Int
    quantite: Int
    timeleft: Int
    managerUnlocked: Boolean
    paliers: [Palier]
}

type World {
  name: String!
  logo: String
  money: Float
  score: Float
  totalangels: Int
  activeangels: Int
  angelbonus: Int
  lastupdate: String
  products: [Product]
  allunlocks: [Palier]
  upgrades: [Palier]
  angelupgrades: [Palier]
  managers: [Palier]
}

type Query {
  getWorld: World
}

type Mutation {
  acheterQtProduit(id: Int!, quantite: Int!): Product
  lancerProductionProduit(id: Int!): Product
  engagerManager(name: String!): Palier
  acheterCashUpgrade(name: String!): Palier
  acheterAngelUpgrade(name: String!): Palier
  resetWorld: World
}

```

Ce schéma est composé d'une entité *World* qui décrit le monde dans sa globalité avec ses propriétés, ses produits, les managers des produits, les upgrades achetables avec de l'argent ou des anges, et la liste des « allunlocks », cad les bonus débloqués une fois que **tous** les produits ont atteint une certaine quantité.

Il possède également une entité *Product* qui décrit les propriétés de chacun des produits, avec en particulier les liste des paliers de bonus débloqué une fois que **ce** produit a atteint une certaine quantité.

Cette entité « Palier » mérite quelques explications. Elle est en effet utilisée à la fois pour spécifier un *Manager*, un *Cash Upgrade*, un *Angel Upgrade*, et un bonus lié à la quantité d'un produit.

Voici comment on interprète sa valeur dans ces quatre situations :

- « pallier » dans la liste des managers :
 - name : nom du manager ;
 - logo : image du manager ;
 - seuil : somme nécessaire pour débloquent le manager ;
 - idcible : identifiant du produit géré par le manager ;
 - ratio, typeratio : non utilisés ;
 - unlocked : vrai ou faux selon que le manager est débloquent ;
- « pallier » dans la liste des *Cash Upgrades* ou des *Angel Upgrades* :
 - name : nom de l'upgrade ;
 - logo : image de l'upgrade, on peut utiliser l'icône du produit auquel l'upgrade est lié, ou une icône spécifique si tous les produits sont concernés, ou encore une icône d'ange s'il s'agit d'un *Angel Upgrade* ;
 - seuil : Argent ou nombre d'anges nécessaire pour acheter l'upgrade ;
 - idcible : identifiant du produit ciblé par l'upgrade ou 0 si tous les produits, ou -1 si le bonus augmente l'efficacité des anges ;
 - ratio :
 - Si typeratio est « vitesse », divise le temps de production par le ratio indiqué ;
 - Si typeratio est « gain », multiplie le revenu du produit par le ratio indiqué ;
 - Si typeratio est « ange », ajoute typeratio au bonus actuel des anges (par exemple si typeratio vaut 1, les anges gagnent 1% à leur bonus de production) ;
 - typeratio : « vitesse », « gain » ou « ange » selon la cible du bonus ;
 - unlocked : vrai ou faux selon que l'upgrade a été acheté ;
- « pallier » dans la liste des « unlocks » lié à la quantité de produits :
 - name : nom de l'unlock ;
 - logo : image de l'unlock, on peut utiliser l'icône du produit auquel l'unlock est lié, ou une icône spécifique si c'est un unlock global ;
 - seuil : Quantité de produit à atteindre pour débloquent le bonus ;
 - idcible : identifiant du produit ciblé par le bonus ou 0 si tous les produits ;
 - ratio :
 - Si typeratio est « vitesse », divise le temps de production par le ratio indiqué ;
 - Si typeratio est « gain », multiplie le revenu du produit par le ratio indiqué ;
 - Si typeratio est « ange », ajoute typeratio au bonus des anges ;
 - typeratio : « vitesse », « gain » ou « ange » selon la cible du bonus ;
 - unlocked : vrai ou faux selon que l'unlock est débloquent.

Seul concession faite par rapport au jeu d'origine, nous ne spécifierons pas d'upgrades de type « ajout d'une certaine quantité de produits ». Le type d'upgrade sera donc uniquement VITESSE, GAIN ou ANGE.

b. Mise en place de la partie serveur

On commence par la partie serveur, car c'est cette partie qui définit le monde et le client va donc en avoir besoin pour fonctionner. Si néanmoins vous vous êtes réparti les rôles et que vous souhaitez avancer la partie cliente en parallèle de la partie serveur, vous pouvez démarrer le développement du client en sautant à la section « Début du travail sur le client web » plus loin et en utilisant pour démarrer le serveur de test disponible à l'adresse :

<https://isiscapitalistgraphql.kk.kurasawa.fr/graphql>

Revenons au serveur. Dans cet énoncé on vous propose de réaliser la partie backend avec « node.js ». Suivez les étapes ci-dessous pour initialiser un projet node.js avec les dépendances nécessaires pour créer un service au format GraphQL.

- Créez un dossier portant le nom de votre projet et ouvrez-le avec votre éditeur javascript préféré (Webstorm, Visual Studio Code, ...).
- Dans un terminal, taper la commande `npm init -y` pour initialiser le projet. Cette commande crée un fichier « package.json » que nous laissons tel quel pour l'instant.
- Toujours avec la commande npm, ajoutez les dépendances dont nous avons besoin à savoir :

```
npm install apollo-server-express graphql
```

- Pour plus de confort pendant le développement ajoutez également la dépendance « nodemon » qui s'occupera de relancer automatiquement le serveur dès que nous modifierons le code :

```
npm install nodemon
```

et modifiez le fichier package.json pour dans la partie « scripts », ajouter une commande « start » qui lance le serveur avec nodemon :

```
"scripts": {  
  "start": "nodemon index.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

- Enfin, créez un fichier « index.js » qui contient le code suivant :

```
const express = require('express');  
const { ApolloServer, gql } = require('apollo-server-express');  
  
// Construct a schema, using GraphQL schema language  
const typeDefs = gql`  
  type Query {  
    hello: String  
  }  
`;  
  
// Provide resolver functions for your schema fields  
const resolvers = {  
  Query: {  
    hello: () => 'Hello world!',  
  },  
};
```

```

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

const app = express();
server.start().then( res => {
  server.applyMiddleware({app});

  app.listen({port: 4000}, () =>
    console.log(`🚀 Server ready at
http://localhost:4000${server.graphqlPath}`)
  );
})

```

Lancez le serveur en utilisant la commande `npm start` et ouvrez votre navigateur sur la page <http://localhost:4000/graphql>

Ce lien doit vous mener à l'interface de contrôle du serveur GraphQL qui vous permet d'en explorer le schéma et de tester les requêtes.

Mettez votre projet sous git mais auparavant ajoutez dans la racine un fichier appelé `.gitignore` avec le contenu suivant :

```

node_modules/
.idea/

```

Cela vous évitera de placer sous git des fichiers qui ne sont pas nécessaires.

c. Création du monde

Maintenant que tout est en place, créez une représentation de votre monde au format JSON dans un nouveau fichier que vous appellerez « `world.js` » conforme aux entités du schéma GraphQL.

Le squelette de ce fichier est le suivant :

```

module.exports = {
  "name": "Nom du monde",
  "logo": "icones/logomonde.jpg",
  "money": 0,
  "score": 0,
  "totalangels": 0,
  "activeangels": 0,
  "angelbonus": 2,
  "lastupdate": 0,
  "products": [
    {
      "id": 1,
      "name": "premier produit",
      "logo": "icones/premierproduit.jpg",
      "cout": 4,
      "croissance": 1.07,
      "revenu": 1,
      "vitesse": 500,
      "quantite": 1,
      "timeleft": 0,
      "managerUnlocked": false,
      "palliers": [
        {

```



```

        "name": "Nom du premier palier",
        "logo": "icones/premierpalier.jpg",
        "seuil": 20,
        "idcible": 1,
        "ratio": 2,
        "typeratio": "vitesse",
        "unlocked": "false"
    },
    {
        "name": "Nom deuxième palier",
        "logo": "icones/deuxiemepalier.jpg",
        "seuil": 75,
        "idcible": 1,
        "ratio": 2,
        "typeratio": "vitesse",
        "unlocked": "false"
    },
    ...
},
{
    "id": 2,
    "name": "Deuxième produit »,
    "logo": "icones/deuxiemeproduit.jpg",
    ...
}
],
"allunlocks": [
    {
        "name": "Nom du premier unlock général",
        "logo": "icones/premierunlock.jpg",
        "seuil": 30,
        "idcible": 0,
        "ratio": 2,
        "typeratio": "gain",
        "unlocked": "false"
    },
    ...
],
"upgrades": [
    {
        "name": "Nom du premier upgrade",
        "logo": "icones/premierupgrade.jpg",
        "seuil": 1e3,
        "idcible": 1,
        "ratio": 3,
        "typeratio": "gain",
        "unlocked": "false"
    },
    ...
],
"angelupgrades": [
    {
        "name": "Angel Sacrifice",
        "logo": "icones/angel.png",
        "seuil": 10,
        "idcible": 0,
        "ratio": 3,

```

```

        "typeratio": "gain",
        "unlocked": "false"
    },
    ...
  ],
  "managers": [
    {
      "name": "Wangari Maathai",
      "logo": "icones/WangariMaathai.jpg",
      "seuil": 10,
      "idcible": 1,
      "ratio": 0,
      "typeratio": "gain",
      "unlocked": "false"
    },
    ...
  ]
};

```

Le travail consiste ensuite à remplir le fichier de façon à spécifier :

- Au moins deux produits qui composeront votre monde (il faudra en faire six au final mais arrêtez-vous à deux pour commencer le développement).
- Pour chacun de ces produits, définir ses caractéristiques de nom, icone, revenu, cout, croissance, etc.
- Pour chacun de ces produits, définir les différents seuils qui octroient des bonus. Vous pouvez vous contenter de trois seuils par produit pour commencer et vous devez définir toutes leurs caractéristiques.
- Définir la liste des managers de produits avec leur cout, icone, nom...
- Définir une liste de Cash Upgrades. Vous pouvez vous contenter d'une liste d'une dizaine d'upgrades pour commencer (pour aller plus vite, vous pouvez utiliser la même icone pour l'upgrade et le produit auquel il offre un bonus).
- Définir une liste d'Angel Upgrades. Vous pouvez vous contenter de trois upgrades pour commencer.

Vous placerez les fichiers de type images correspondant aux produits, managers, upgrades... dans un dossier `public/icones` que vous créerez dans le dossier racine de votre projet. Pour que le serveur aille les chercher à cet endroit vous devez ajouter au fichier « `index.js` », juste après la ligne initialisant « `express` » :

```

const app = express();
app.use(express.static('public'));

```

Le fait de mettre les images à cet endroit, permettra aux clients d'y accéder au moyen de l'URL <http://localhost:4000/icones/image.jpg>.

Placez une image dans le dossier `icones`, et vérifiez que l'URL fonctionne bien.

d. Séparation des composants

Pour l'instant tous les composants sont dans le fichier « index.js » (à part « world.ts »), nous allons placer les principaux dans des fichiers distincts.

Un service GraphQL se définit avec deux composants : Le schéma, et les résolveurs. Créez un fichier pour contenir le premier (vous l'appellerez schema.js), et placez-y le schéma donné plus haut dans l'énoncé. Pour cela vous devez reprendre la syntaxe utilisée dans le fichier « index.js » à savoir :

```
const {gql} = require("apollo-server-express");
module.exports = gql`

  enum RatioType {
    GAIN
    VITESSE
    ANGE
  }

  type Product {
    id: Int!
    name: String
  }

  ...

`;
```

Autrement dit vous devez encadrer le schéma dans une chaîne de caractère qui commence par `gql` puis une apostrophe inversée, et qui finit par une apostrophe inversée. Comme on veut importer ce fichier dans d'autres composants, on place cette chaîne dans la propriété `exports` d'une variable spéciale de Node qui s'appelle `module`.

Ceci fait vous pouvez supprimer le schéma initial du fichier « index.js » et le remplacer par l'importation de celui contenu dans « schema.js » en insérant au début du fichier :

```
const typeDefs = require("../schema")
```

Faisons la même chose pour les résolveurs. Créez un fichier « resolvers.js » dont la structure est la suivante :

```
module.exports = {
  Query: {
    getWorld(parent, args, context) {
      return context.world
    },
  },
  Mutation: {
  },
};
```

Pour l'instant ce résolveur ne comporte qu'une seule « query », celle qui doit retourner le monde au client. On voit que la query prend trois paramètres. Le premier est l'objet parent de l'objet qu'on est en train de résoudre. Ce premier paramètre n'est utile que quand on résout des sous-propriétés, ce

qui n'est pas le cas de notre requête `getWorld` qui est une requête racine. Le deuxième (`args`) contient les paramètres éventuels de la requête (`getWorld` n'en a pas). Le troisième contient un contexte que nous utiliserons pour transmettre le monde au résolveur. Nous allons voir juste après comment on fait cela, mais pour l'instant on suppose que `context.world` contient le monde complet qui est donc retourné par le résolveur.

Retournez dans le fichier « `index.js` » et supprimez son résolveur. Remplacez-le par l'inclusion du résolveur que nous avons mis dans « `resolvers.js` » avec la ligne :

```
const resolvers = require("./resolvers")
```

Enfin, il ne nous reste plus qu'à inclure dans « `index.js` » le monde complet défini dans « `world.js` » et à le placer dans le contexte du résolveur. Pour le premier point ajoutez dans « `index.js` » la ligne suivante :

```
let world = require("./world")
```

Pour le deuxième, ajoutez un troisième paramètre quand vous créez le serveur apollo en modifiant la ligne actuelle pour qu'elle devienne :

```
const server = new ApolloServer({
  typeDefs, resolvers,
  context: async ({ req }) => ({
    world: world
  })
});
```

C'est ce troisième paramètre qui permet de définir l'objet `context` passé au résolveur. On y place une propriété `world` que l'on initialise à la valeur de notre variable `world`.

Testez tout ça en revenant dans votre navigateur sur l'adresse <http://localhost:4000/graphql> et en utilisant le sandbox studio pour appeler la requête `getWorld` :

```
query getWorld {
  getWorld {
    name
    money
  }
}
```

Si tout va bien la requête doit retourner le nom du monde complet et l'argent initial. Vous pouvez commencer le travail sur le client en continuant section suivante, ou continuer le travail sur le serveur en vous rendant page 37.

3. Début du travail sur le client web

a. Serveur de test

Si le développement du serveur n'est pas assez avancé, vous pouvez utiliser un serveur de test sur lequel connecter votre client pendant la phase de développement. L'URL de ce serveur est le suivant :

<https://isiscapitalist.kk.kurasawa.fr>

Attention ce serveur doit être remplacé par votre propre serveur, dès que ce dernier sera apte à répondre aux requêtes du client.

b. Création du projet React

Le client web va être construit en utilisant la bibliothèque React (<https://fr.reactjs.org/>). Pour mettre en place le projet.

On crée ensuite un projet React en tapant la commande (et en remplaçant « PROJECT_NAME » par le nom de votre projet) :

```
npx create-react-app PROJECT_NAME --template typescript
```

La commande crée un projet minimal qui peut être lancé par les commandes :

```
cd PROJECT-NAME
npm start
```

La commande `npm start` lance un serveur web (en l'occurrence node.js), y déploie le projet en général sur le port 3000. Il ne reste plus qu'à faire pointer un navigateur web sur l'adresse <http://localhost:3000> pour visualiser la page d'accueil. Vous devez laisser tourner en permanence cette commande lors de votre développement. Vous remarquerez que quand vous modifierez un des composants du projet, celui-ci sera automatiquement redéployé et rechargé dans le navigateur.

Vous pouvez à présent lancer votre IDE Web favori (Visual Studio, WebStorm, Atom, Sublime Text, etc.) et ouvrir le dossier correspondant à votre projet pour commencer à travailler.

c. Description du travail attendu

Dans un premier temps, nous allons construire un client autonome qui ne synchronisera pas les actions de l'utilisateur avec le serveur. La seule interaction de ce client avec le serveur consistera à obtenir la description du monde lors du premier chargement de la page web.

En fonction de cette description, le client doit bâtir une mise en page présentant les éléments de base du jeu à savoir les différents produits (icône, nom, quantité, barre de production) et les éléments d'interaction (bouton d'achat des produits, boutons cliquables permettant de spécifier les quantités d'achat et de faire apparaître les différents *upgrades*, *unlocks* et autres *managers*) La Figure 7 présente un exemple de l'interface à obtenir.

Essentiellement cette interface repose sur une mise en page divisant la page en trois parties :

- Un bandeau d'entête annonçant le monde (icône et nom), l'argent du joueur, le bouton de quantité d'achat et un champ texte permettant au joueur de saisir son nom ;

- Un bandeau gauche spécifiant les boutons cliquables permettant d'afficher des fenêtres supplémentaires décrivant des éléments supplémentaires du jeu ;
- Une partie centrale listant les produits et leurs éléments d'interaction.

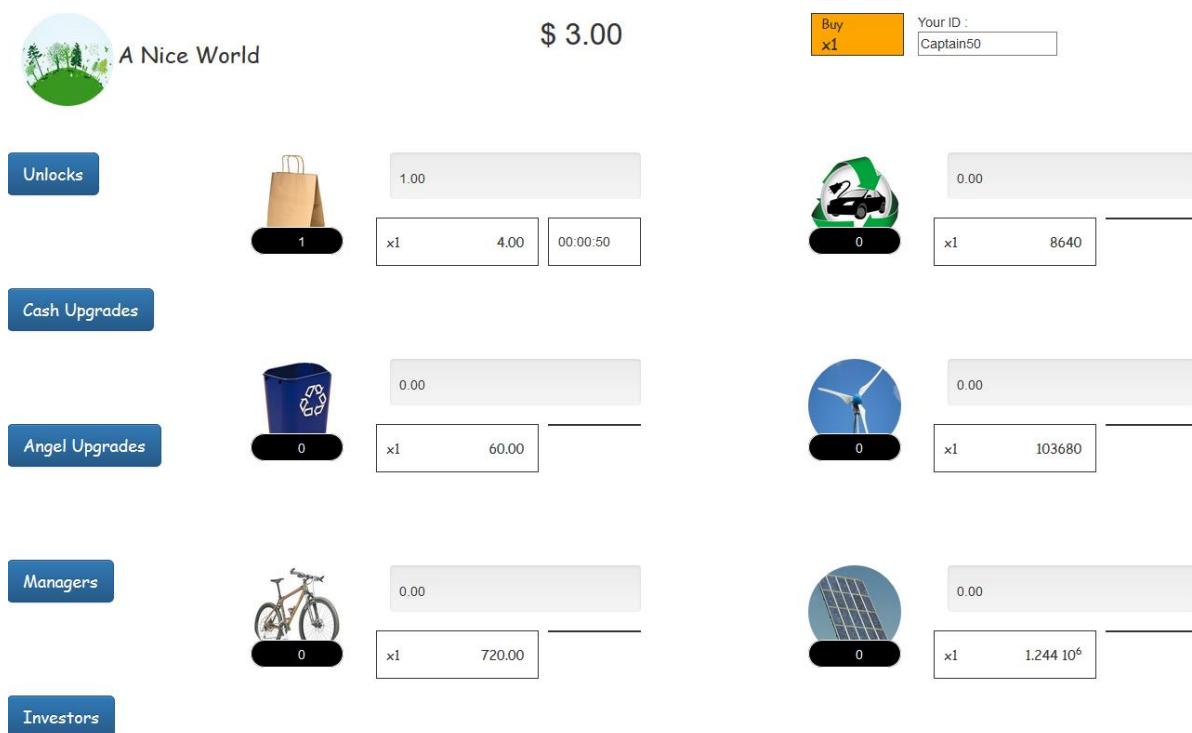


Figure 7 – Illustration de l'interface du jeu

Les produits sont eux-mêmes affichés sous la forme d'une mise en page en plusieurs parties :

- Une partie gauche présentant l'image du produit à laquelle se superpose sa quantité
- Une partie droite divisée en :
 - Une partie haute présentant la barre de progression de la production qui indique aussi le gain qui sera généré. Ne vous occupez pas pour l'instant du rendu de cette barre de progression, vous réserverez simplement l'espace qui lui sera nécessaire.
 - Une partie basse qui permet d'acheter plus de ce produit et qui indique également la quantité qui sera achetée et le coût associé, avec à côté le temps qui reste à s'écouler pour que la production du produit soit complète.

Pour styler l'affichage, on peut utiliser « MUI Core » qui est un ensemble de styles CSS et de widget. Pour installer ce composant, tapez la commande suivante à la racine de votre projet :

```
npm install @mui/material @emotion/react @emotion/styled
```

Pour réaliser une mise en forme sous forme de grille, on peut utiliser le CSS3 Grid Layout qui permet facilement de spécifier des mises en page adaptatives en colonnes (voir par exemple https://developer.mozilla.org/fr/docs/Web/CSS/CSS_Grid_Layout).

Voici par exemple comment on peut définir un affichage avec un entête constitué de quatre colonnes, et une partie principale composée de deux colonnes, la deuxième colonne étant elle-même composée de cellules réparties sur deux colonnes


```

<div class="header">
  <div> logo monde </div>
  <div> argent </div>
  <div> multiplicateur </div>
  <div> ID du joueur </div>
</div>

<div class="main">
  <div> liste des boutons de menu </div>
  <div class="product">
    <div> premier produit </div>
    <div> second produit </div>
    <div> troisième produit </div>
    <div> quatrième produit </div>
    <div> cinquième produit </div>
    <div> sixième produit </div>
  </div>
</div>

```

Avec le style CSS suivant :

```

.header {
  display: grid;
  grid-template-columns: 25% 25% 30% 20%;
}

.main {
  display: grid;
  grid-template-columns: 20% 80%;
}

.product {
  display: grid;
  grid-template-columns: 1fr 1fr;
}

```

Vous adapterez cet exemple pour réaliser la mise en page que vous souhaitez pour votre jeu.

d. Création des premiers composants et du service de communication avec le service web

En ce qui concerne l'organisation des sources, nous utiliserons plusieurs composants React. Le premier, `App.tsx` (déjà créé lors de la phase d'initialisation du projet) s'occupera d'aller chercher le monde, et de définir un champ permettant de saisir le nom du joueur. Il délèguera à un autre composant (`Main.tsx`) le soin de définir les éléments globaux de l'interface (icône et nom du monde, argent, multiplicateur d'achat, bandeau de droite avec les upgrades, etc.), qui lui-même délèguera à un troisième composant (`Product.tsx`) le responsabilité de gérer l'affichage des éléments d'un produit.

Nous aurons également besoin de faire des appels au serveur au format GraphQL. Pour cela nous allons utiliser la bibliothèque client Apollo. Installez sa dépendance en tapant :

```
npm install @apollo/client graphql
```

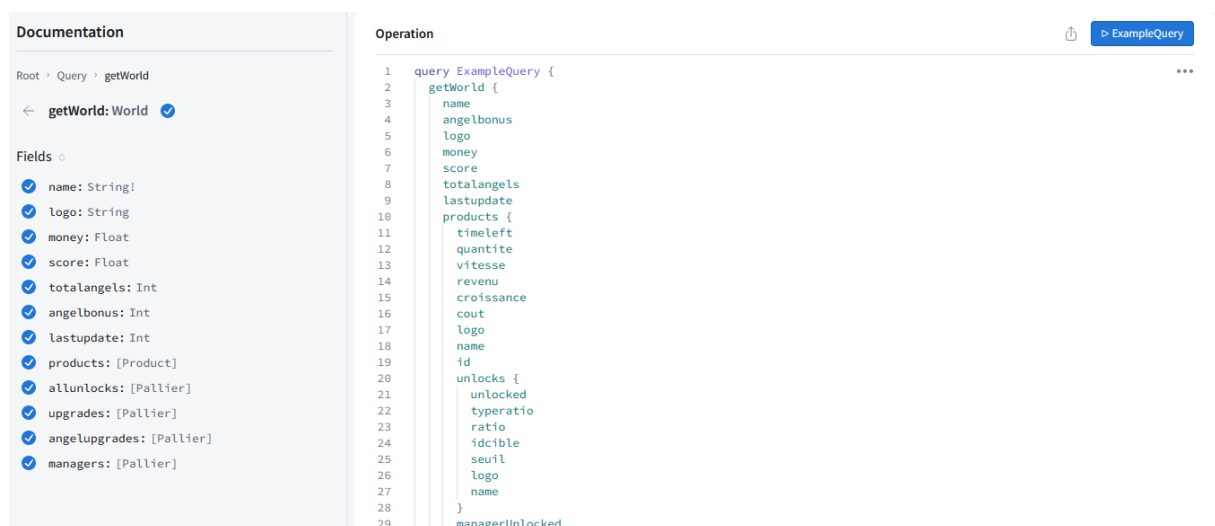
Dans le fichier « index.tsx » ajoutez les lignes pour initialiser le client. Vous devez transformer le fichier pour qu'il ressemble à cela :

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

const client = new ApolloClient({
  uri: 'http://localhost:4000/graphql',
  cache: new InMemoryCache()
});

ReactDOM.render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

Construisons à présent la requête GraphQL qui va aller chercher le monde. On désire obtenir le monde avec toutes ses propriétés. La requête va donc être assez longue. Aidez-vous du sandbox studio du serveur pour la construire. Pour cela cliquez sur la requête « getWorld » puis sur toutes les sous propriétés de cette requête de façon à obtenir la requête complète (voir copie écran ci-dessous).

The screenshot shows the GraphQL Playground interface. On the left, the 'Documentation' panel displays the 'getWorld' query with its fields: name, logo, money, score, totalangels, angelbonus, lastupdate, products, allunlocks, upgrades, angelupgrades, and managers. On the right, the 'Operation' panel shows the query text:

```
1 query ExampleQuery {
2   getWorld {
3     name
4     angelbonus
5     logo
6     money
7     score
8     totalangels
9     lastupdate
10    products {
11      timeleft
12      quantite
13      vitesse
14      revenu
15      croissance
16      cout
17      logo
18      name
19      id
20      unlocks {
21        unlocked
22        typeratio
23        ratio
24        idcible
25        seuil
26        logo
27        name
28      }
29    }
30  }
31 }
```

Copiez la requête obtenue et collez là dans le fichier App.tsx en la mettant dans une variable que vous pouvez appeler « GET_WORLD » :

```
const GET_WORLD = gql`
  query getWorld {
    getWorld {
      name
    }
  }
`
```

```

        angelbonus
        logo
        money
        score
        totalangels
        lastupdate
        products {
            timeleft
            quantite
            vitesse
            revenu
            croissance
            cout
            logo
            name
            id
        }
    }
}

```

A présent que nous avons notre requête, nous allons pouvoir l'appeler, mais auparavant, comme nous sommes en TypeScript, il va falloir que nous définissions les classes d'objet que cette requête va nous renvoyer. Ce travail un peu fastidieux vous est fourni sous la forme d'un fichier `world.ts`, disponible sur moodle que vous devez importer dans la racine de votre projet. Vous remarquerez qu'il définit trois classes : `World`, `Product`, et `Pallier` qui correspondent aux trois types définis par le schéma GraphQL du monde :

```

export class World {
    name : string = ""
    logo : string = ""
    money: number = 0
    score: number = 0
    totalangels: number = 0
    activeangels: number = 0
    angelbonus: number = 0
    lastupdate: string = ""
    products : Product[]
    allunlocks: Pallier[]
    upgrades: Pallier[]
    angelupgrades: Pallier[]
    managers: Pallier[]

    constructor() {
        this.products = [ ]
        this.managers = [ ]
        this.upgrades = [ ]
        this.angelupgrades = [ ]
        this.allunlocks = [ ]
    }
}

export class Product {
    id : number = 0
    name : string = ""
    logo : string = ""
    cout : number = 0
    croissance: number = 0
    revenu: number = 0
    vitesse: number = 0
    quantite: number = 0
}

```

```

    timeleft: number = 0
    lastupdate: number = 0
    managerUnlocked: boolean = false
    palliers : Pallier[]

    constructor() {
        this.palliers = []
    }
}

export class Pallier {
    name: string = ""
    logo: string = ""
    seuil: number = 0
    idcible: number = 0
    ratio: number = 0
    typeratio: string = ""
    unlocked: boolean = false
}

```

Une fois ce fichier importé dans le projet, modifiez la page « App.tsx » pour qu'elle :

- Affiche un champ de formulaire pour que le joueur puisse saisir son nom.
- Réalise la requête GraphQL `getWorld` en passant dans l'entête le nom du joueur.

Pour le premier point, l'interface doit proposer un état local pour représenter le nom du joueur et un champ texte (en haut à droite sur la Figure 7) permettant à l'utilisateur de le spécifier :

```
const [username, setUsername] = useState("")
```

et dans le JSX :

```
<input type="text" value={username} onChange={onUserNameChanged}/>
```

A vous d'implémenter la méthode `onUserNameChanged` pour mettre à jour l'état quand l'utilisateur saisi un nouveau pseudo.

Ce pseudo va devoir être stocké côté client (pour que lors du rechargement de la page, le joueur n'ait pas à ressaisir son nom). Pour cela on peut utiliser le **localStorage** du navigateur qui permet de sauvegarder des valeurs localement sous la forme de paires clés-valeur. Au chargement de la page on ira voir si cet espace contient par exemple la clé `username` et on utilisera sa valeur en tant que pseudo de l'utilisateur :

```
let username = localStorage.getItem("username");
```

Si initialement ce pseudo est vide, on peut en générer un aléatoirement. On peut par exemple tirer un nombre au sort entre 0 et 10000 avec l'expression `Math.floor(Math.random() * 10000)` et concaténer ce nombre à la fin d'un pseudo (genre Captain1208).

Une fois le pseudo initialisé, mais aussi à chaque fois qu'il est mis à jour, on mettra à jour sa valeur dans le **localStorage** avec l'expression :

```
localStorage.setItem("username", username);
```

-

Pour le deuxième point, le client Apollo propose un Hook React, qui permet de faire une requête GraphQL est de récupérer un quator de variable qui permet respectivement de connaître l'état de chargement de la requête (en chargement ou prête), l'erreur éventuelle, la donnée retournée, et une fonction qu'on peut appeler pour refaire la requête. On s'en sert de la façon suivante :

```
const {loading, error, data, refetch } = useQuery(GET_WORLD, {
  context: { headers: { "x-user": username } }
});
```

Vous remarquerez qu'on peut passer en paramètre les données que l'on souhaite ajouter à l'entête de la requête. Ici comme demandé, on passe le nom du joueur dans un entête appelé « x-user ». Cet entête sera ensuite lu par le serveur pour que le monde retourné soit celui du joueur (on verra cela quand on reviendra sur la partie serveur).

Après avoir exécuté la requête, il faut tester la valeur de la variable `loading` pour savoir si la requête a fini de s'exécuter (si `loading` est vrai c'est que la requête est encore en cours de chargement), puis tester la variable `error` pour savoir s'il y a eu une erreur, et si ce n'est pas le cas on accède au données du monde dans `data.getWorld`.

Au final on écrira :

```
let corps = undefined
if (loading) corps = <div> Loading... </div>
else if (error) corps = <div> Erreur de chargement du monde ! </div>
else corps = <div> { data.getWorld.name } </div>
return (
  <div>
    <div> Your ID :</div>
    <input type="text" value={username} onChange={onUserNameChanged}/>
    { corps }
  </div>
);
```

Testez le fonctionnement, en principe la page doit afficher le nom de votre monde.

Passant à présent au composant « Main.tsx » qui doit afficher l'interface principale. Créez ce nouveau composant. Il doit prendre en paramètre le monde obtenu par son parent, ainsi que le nom de l'utilisateur. Il sera donc appelé dans « App.tsx » comme ceci :

```
<Main loadworld={data.getWorld} username={username} />
```

Par conséquent, vous devez dans « Main.tsx » définir les propriétés en paramètres comme ceci :

```
type MainProps = {
  loadworld: World
  username: string
}
```

```
export default function Main({ loadworld, username } : MainProps) {
...
}
```

Le composant Main va donc récupérer dans la variable loadworld, l'état complet du monde. Ce monde est en lecture seule, et nous allons en faire une copie pour le faire évoluer en fonction des actions du joueur, et en faire un état local avec le hook d'état. Ajoutez donc au début de la fonction principale la ligne suivante :

```
const [world, setWorld] = useState(JSON.parse(JSON.stringify(loadworld)) as World)
```

Travaillons à présent sur le rendu visuel du monde.

e. Réalisation du layout général

Dotez la page Main.tsx du code HTML nécessaire pour spécifier les grandes lignes de votre mise en pages, en insérant aux endroits adéquats les valeurs associés aux propriétés de votre monde.

Par exemple, à l'endroit où doit s'afficher l'icône du monde, on pourra trouver le code html suivant :

```

```

Ce code insère en effet une balise dont l'attribut src est calculé à partir de l'adresse du serveur web et du nom du logo du monde.

Autre exemple, voici comment on peut insérer le nom du monde :

```
<span> {world.name} </span>
```

Bien entendu, ces tags html devront être stylés pour réaliser un design un peu sympa (voir plus bas pour quelques éléments et astuces CSS).

Procédez ainsi pour tous les éléments généraux du monde (sauf pour le détail des produits).

Déléguez à un composant Product le soin de réaliser le design d'un produit. Pour cela, à l'endroit où doit s'afficher par exemple le premier produit, vous insèrerez le code html suivant (les classes CSS sont à adapter en fonction du rendu que vous souhaitez obtenir) :

```
<Product prod={ world.products.product[0] } />
```

Vous remarquerez qu'on passe au composant Product, le produit dont il doit s'occuper via la propriété prod, et le service d'appels dont il aura aussi besoin via la propriété services.

Créez donc un nouveau fichier Product.tsx pour le composant produit suit :

```
type ProductProps = {
  prod: Product
}
```



```
export default function ProductComponent({ prod, services } : ProductProps)
{
  return (
    <div> ... </div>
  )
}
```

Notez que nous avons typé les propriétés passées à ce composant avec le mot clé `type` de `typescript`.

A partir de ce point, vous disposez dans le composant `Product` d'une propriété `prod` qui correspond au produit à afficher.

Vous pouvez donc spécifier le code HTML qui réalise l'affichage d'un produit. A vous de jouer.

f. Éléments utiles de CSS

Voici quelques connaissances CSS qui vous permettront d'optimiser votre rendu graphique. Ces styles sont à placer soit dans le fichier `styles.css` si vous voulez qu'ils soient globaux à toutes les pages html (méthode conseillée ici), soit dans les fichiers `component.css` si vous souhaitez qu'ils ne s'appliquent qu'au composant en question.

Images arrondies :

Pour spécifier les images de vos produits (et plus tard de vos managers), vous pouvez d'une part spécifier leur taille et d'autre part les insérer dans des cercles en leur appliquant le style CSS suivant :

```
.round {
  width: 100px;
  height: 100px;
  border-radius: 50%
}
```

Superposition de deux éléments :

Pour superposer deux éléments, il faut les encapsuler ensemble dans une même balise `<div>` à laquelle on applique un positionnement relatif, et spécifier l'un des deux éléments (ou les deux) en positionnement absolu.

Ainsi si au lieu de mettre l'un sous l'autre les deux éléments « contenu 1 » et « contenu 2 », on veut légèrement les superposer, on pourra écrire :

```
.lesdeux {
  position: relative;
}

.lesecond {
  position: absolute;
  // on met cet élément à 10 pixels du bas de son conteneur
  // du coup il se superpose à l'autre qui est dans le flux normal
  bottom: -10px;
}
```

```
<div class="lesdeux">
  <div class="lepremier">Contenu 1</div>
  <div class="lesecond">Contenu 2</div>
</div>
```

g. Quelques méthodes utilitaires

A plusieurs reprises vous serez amené à formater différentes valeurs dans votre interface. C'est par exemple le cas des grands nombres comme l'argent possédé par le joueur (qui doit s'afficher sous la forme de puissances de 10 quand ce nombre est important) ou le temps restant pour la production d'un produit qui doit s'afficher sous la forme *heure : minutes : secondes : dixièmes de secondes*.

Nous aurons donc besoin de fonctions utilitaires qui formattent ces données, en prenant en paramètre la valeur à formater, et en retournant la chaîne de caractère à afficher. A titre d'exemple définissons une fonction qui prend en paramètre un nombre flottant et qui produit son formatage en puissance de dix, avec quatre chiffres significatifs.

Une implémentation un peu naïve de cette méthode pourrait être :

```
export function transform(valeur: number): string {
  let res : string = "";
  if (valeur < 1000)
    res = valeur.toFixed(2);
  else if (valeur < 1000000)
    res = valeur.toFixed(0);
  else if (valeur >= 1000000) {
    res = valeur.toPrecision(4);
    res = res.replace(/e\+(.*)/, " 10<sup>$1</sup>");
  }
  return res;
}
```

Le code ci-dessus restreint la précision des grands nombres flottant à quatre chiffres, puis transforme l'exposant en un formatage adéquat pour un rendu HTML faisant apparaître un 10ⁿ. Cette implémentation peut éventuellement être améliorée en fonction de vos propres désirs d'affichage et de design.

Placez cette méthode dans un nouveau fichier que vous pouvez appeler par exemple `utils.ts`.

Le fait d'avoir utilisé le mot clé `export` devant la définition de la fonction vous permet de l'importer et d'en faire usage dans les fichiers où vous en avez besoin. Par exemple pour afficher l'argent du monde correctement formaté on pourra importer la fonction comme ceci (votre IDE devrait vous le proposer automatiquement en principe).

```
import {transform} from "../utils";
```

et ensuite pour afficher l'argent :

```
<span dangerouslySetInnerHTML={{ __html: transform(world.money) }} /></span>
```

Notez l'utilisation de l'attribut `dangerouslySetInnerHTML` qui permet au rendu d'interpréter les balises HTML produites par le *pipe*. Si vous utilisez simplement la notation habituelle `{transform(world.money)}`, le nom des balises `<sup>` apparaîtra dans le rendu, ce qui n'est pas l'effet recherché.

4. Actions du joueur

Le *layout* étant en place, nous allons commencer à implémenter le traitement des actions du joueur. La première d'entre elles consiste à cliquer sur l'icône d'un produit pour en lancer la production.

a. Démarrage de la production d'un produit

Dans le fichier `Product.tsx`, ajoutez un gestionnaire d'évènement (*click*) sur les icônes de produits qui mène à une méthode `startFabrication()` que vous allez implémenter.

La méthode `startFabrication()` doit lancer une barre de progression dans l'espace prévu à cet effet. Vous êtes libres d'utiliser d'implémenter cette barre comme vous voulez, mais l'énoncé vous en a fourni une sous la forme d'un composant `MyProgressbar.tsx` que vous trouverez sur moodle.

On ajoute ensuite ce composant dans le html avec :

```
<MyProgressbar className="barstyle" vitesse={product.vitesse}
  initialValue={product.vitesse - product.timeleft}
  run={run} frontcolor="#ff8800" backcolor="#ffffff"
  auto={product.managerUnlocked} orientation={Orientation.horizontal}
  onCompleted={onProgressbarCompleted}/>
```

La signification des attributs est la suivante :

- `vitesse` correspond au temps en millisecondes pour que la barre se remplisse complètement. On la règle donc sur la vitesse de production du produit.
- `initialvalue` correspond au temps en millisecondes déjà écoulé dans la production du produit lors du rendu de la barre. C'est donc la différence entre la vitesse du produit et son `timeleft`.
- `run` est un booléen qui quand il est vrai anime la production de la barre, et quand il est faux la remet à zéro. Il faut donc le mettre en vrai quand on lance la production du produit, et le remettre à faux quand la production est terminée. En React, cette variable correspond donc à un état que vous devrez déclarer avec un *hook* `useState`.
- `frontcolor` et `backcolor` représentent respectivement la couleur de la barre et de son fond.
- `auto` est un booléen qui rend automatique le redémarrage de la barre de la progression. Il faut donc le mettre en vrai quand le manager du produit est débloqué. Une fois que `auto` est vrai, la barre fonctionne toute seule et il ne faut plus changer la valeur de l'attribut `run`.
- `orientation` permet d'avoir une barre de progression horizontale ou verticale. Par défaut elle est horizontale.
- `onCompleted` permet de spécifier une fonction qui sera appelée à chaque fois que la barre de progression arrive au bout. Quand le manager du produit n'est pas encore débloqué, vous pouvez par exemple vous en servir pour passer l'état `run` à faux, afin de remettre la barre à zéro.

Faites maintenant en sorte qu'un *clic* sur l'icône du produit lance sa production. Pour l'instant laissez vide la méthode appelée (voir page suivante pour savoir quoi mettre dedans).

b. La boucle principale de calcul du score

Nous devons à présent définir la boucle principale de notre produit, une méthode qui sera appelée à intervalle régulier (par exemple tous les dixièmes de seconde) pour mettre à jour l'interface en fonction du temps qui passe et calculer l'évolution de l'argent gagné.

En javascript, c'est la méthode `setInterval(...)` qui permet d'exécuter du code toutes les *ms* millisecondes. Ainsi si on appelle `calcScore()` notre fonction de calcul du score, on peut, lors du démarrage du composant, l'appeler tous les dixièmes de seconde en écrivant :

```
setInterval(() => scalcScore(), 100)
```

Pour déclencher cette programmation au démarrage, le *hook* `useEffect()` est tout indiqué, d'autant qu'il peut retourner une fonction qui doit être appelée quand le composant est détruit, et à qui nous ferons ici annuler la programmation du `setInterval()`. Malheureusement du fait de la fermeture lexicale de Javascript, si vous l'utilisez comme ci-dessous, cela ne fonctionnera pas :

```
useEffect(() => {
  let timer = setInterval(() => sacalcScore(), 100)
  return function cleanup() {
    if (timer) clearInterval(timer)
  }
}, [])
```

La raison de ce non-fonctionnement est un peu compliquée à expliquer ici, mais elle est liée au fonctionnement des méthodes en Javascript et au fait que les *hooks* de React sont à la base des hacks qui permettent d'utiliser des fonctions à la place des classes.

Pour que cela fonctionne, il faut donc utiliser en plus un autre *hook*, qui permet d'obtenir une référence à une fonction, et c'est cette référence qui doit être passée à `useEffect()`. La version correcte est donc la suivante :

```
const savedCallback = useRef(calcScore)

useEffect(() => savedCallback.current = calcScore)

useEffect(() => {
  let timer = setInterval(() => savedCallback.current(), 100)
  return function cleanup() {
    if (timer) clearInterval(timer)
  }
}, [])
```

Si vous ne comprenez pas ce code, ce n'est pas très important, contenez-vous de le recopier. On touche à aux limites des *hooks* React en termes d'élégance d'utilisation (nous n'aurions pas eu ce problème si nous avions utilisé une classe et sa méthode `ComponentDidMount`).

Il ne reste plus qu'à implémenter une méthode `calcScore()` qui pour un produit et en fonction du temps écoulé depuis la dernière fois, décrémente le temps restant de production du produit, et si ce temps devient négatif ou nul, ajoute l'argent généré au score et efface la barre de production.

Quelques indications pour faire cela :

- Lors de la mise en production d'un produit (donc lors du *clic* sur son icône), initialisez sa propriété `timeleft` avec la valeur de sa propriété `vitesse`. Ainsi s'il faut 2000ms pour créer un produit, lors du lancement, il reste bien (`timeleft`) 2000ms pour qu'il soit créé.
- Toujours lors du lancement de la production d'un produit, utilisez une variable globale `lastupdate` initialisée à `Date.now()` pour stocker l'instant de démarrage de cette production.
- Dans la méthode `calcScore()`, testez :
 - Si sa propriété `timeleft` vaut zéro ne faites rien. Cela signifie que le produit n'est pas en cours de production.
 - Sinon calculez le temps écoulé depuis sa dernière mise à jour (`Date.now() - lastupdate`), et soustrayez ce temps au temps qui lui reste avant de finir sa production pour obtenir la nouvelle valeur de la propriété `timeleft`.
 - Si `timeleft` est devenu nul ou négatif (s'il est négatif remettez-le à zéro), notez ce que rapporte la production à l'argent du joueur (voir plus bas), et remettez la barre de production à zéro.
 - Sinon `timeleft` est strictement positif. Calculez la nouvelle valeur de la barre de progression qui est `progress = Math.round(((product.vitesse - product.timeleft) / product.vitesse) * 100)`
 - Dans tous les cas, n'oubliez pas de remettre à jour la variable `lastupdate` pour qu'elle représente toujours la date de dernière mise à jour.

Au final, chaque clic sur un produit lance sa production, et à la fin vous devez augmenter le score du gain obtenu. Problème, le composant produit n'a pas accès au score global car celui-ci est géré par le composant parent (`App.tsx`). Il faut donc que le composant produit communique avec son parent et lui indique, à chaque fin de production d'un produit, qu'il faut augmenter l'argent possédé par le joueur.

En React, la communication dans le sens enfant vers parent, passe par le passage d'une méthode du parent vers l'enfant, méthode qui est ensuite appelée par l'enfant.

Dans la classe `Product.tsx`, déclarez une *prop* supplémentaire comme ceci :

```
type ProductProps = {
  prod: Product
  onProductionDone: (product: Product) => void,
  services: Services
}

export default function ProductComponent({ prod, onProductionDone, services
} : ProductProps) {

...

}
```

Le type de cette *prop* que nous avons appelée `onProductionDone`, est une méthode qui prend en paramètre un produit, et qui ne renvoie rien.

Modifiez aussi la méthode `calcScore()` pour que quand elle détecte qu'un produit a terminé sa production, elle appelle la méthode passée par le parent :

```
function calcScore() {
  if (...) {
    // quand la production est terminée, on prévient le composant parent
    onProductionDone(product)
  }
}
```

Du côté du parent (`App.tsx`), on définit cette méthode et on la passe à l'enfant comme ceci :

```
function onProductionDone(p: Product): void {
  // calcul de la somme obtenue par la production du produit
  let gain = ...
  // ajout de la somme à l'argent possédé
  addToScore(gain)
}
```

Et dans le JSX :

```
<ProductComponent onProductionDone={onProductionDone}
  prod={p} />
```

c. L'achat de produit

Le joueur doit avoir la possibilité d'acheter une certaine quantité de produit en cliquant sur le bouton prévu à cet effet. Le nombre de produits achetable dépend d'une part de l'argent qu'il possède, d'autre part du commutateur général `x1`, `x10`, `x100`, ou `xMax` situé en haut à droite de l'interface.

Commencez-donc par implémenter ce bouton commutateur qui est géré par le composant principal et qui doit fonctionner de la façon suivante :

- Chaque clic sur le bouton doit lui faire changer de position selon le cycle `x1` -> `x10` -> `x100` -> `Max` -> `x1`, etc.
- A chaque changement de position, le composant doit prévenir chaque produit de la nouvelle position ;
- Les produits doivent modifier le marqueur de quantité d'achat selon la nouvelle position. Quand il s'agit des positions `x1`, `x10` ou `x100`, on prendra soin de rendre le bouton d'achat cliquable uniquement si le joueur est en capacité financière d'acheter la quantité spécifiée. Cependant, quand le bouton commutateur est sur la position `Max`, il s'agit de calculer la quantité maximale achetable par le joueur de ce produit, et d'inscrire cette quantité dans le bouton d'achat.

Quelques indications pour réaliser cela :

- Dans le composant principal, on utilisera un état local `qtmulti` pour stocker l'état actuel du commutateur d'achat général (vous devez donc ajouter un état supplémentaire avec le *hook* `useState`).
- On transmettra la valeur de cette propriété aux composants produit par exemple :

```
<ProductComponent
  onProductionDone={onProductionDone}
  qtmulti={qtmulti}
  prod={p}
  services={ services } />
```

- Le composant produit va avoir besoin de connaître l'argent possédé par le joueur. Pour l'instant ce n'est pas le cas puisque qu'il ne connaît que les données d'un produit, et la valeur du commutateur d'achat `qtmulti`. Comme nous l'avons fait pour `qtmulti`, faites en sorte que le composant parent passe au composant produit, la valeur de `world.money`.
- Dans le composant produit, on implémentera une fonction `calcMaxCanBuy()` qui calcule la quantité supplémentaire maximale achetable par le joueur de ce produit. Notez que chaque achat d'un produit supplémentaire coûte le prix actuel du produit multiplié par son pourcentage de croissance. Ainsi si x est le coût actuel d'un exemplaire de produit, et si c est la croissance de ce coût, alors acheter un produit de plus coûtera $x * c$, acheter deux produits coûtera $x * c + x * c * c$, trois produits $x * c + x * c * c + x * c * c * c$, etc. Pour généraliser, acheter n produits demandera $x * (1 + c + c^2 + c^3 + \dots + c^n)$ argent. A vous d'être capable à partir d'une certaine somme possédée par le joueur, de trouver n (vous devriez chercher du côté des suites géométriques...).
- On fera en sorte que la valeur calculée par `calcMaxCanBuy()` soit utilisée pour afficher le bon nombre de produit achetable dans l'interface du produit, qui le rend sélectionnable ou pas en fonction de l'argent du joueur.
- Notez que la méthode `calcMaxCanBuy()` doit être appelé non seulement quand la valeur du commutateur général change, mais aussi quand l'argent possédé par le joueur évolue.
- Pensez bien à mettre jour la nouvelle quantité de produit possédé une fois l'achat effectué.
- Afin, acheter un certain nombre de produits doit décrémenter l'argent détenu par le joueur du coût des produits. Comme nous l'avons fait pour la production, le composant produit doit donc prévenir son parent qu'un achat vient d'être effectué. Coté parent implémentez donc une méthode `onProductBuy(qt: number, product: Product)` qui prend donc en paramètre le produit acheté et la quantité achetée. Passez cette méthode au composant produit comme nous l'avons fait avec la méthode `OnProductionDone`, et faites en sorte que l'enfant appelle cette méthode quand une certaine quantité de produit est achetée.

5. Les managers

a. Interface pour lister les managers

Dans cette partie nous allons implémenter la partie cliente de l'achat de managers qui permettront d'automatiser la production de produits.

La liste des managers doit apparaître lorsque le joueur clique sur le bouton « *managers* » situé dans la colonne gauche de l'interface. Cette liste doit venir se superposer à l'interface en cours et elle devra se fermer quand le joueur cliquera sur son bouton de fermeture.

Vous êtes libre de choisir votre propre façon de faire cela (par exemple vous pouvez créer un composant dédié à cela, ou vous pouvez utiliser les modales présentes dans MUI React), mais voici une façon de le faire :

On peut de façon simple gérer des fenêtres superposées en utilisant un style css qui positionne la section en fixe. Par exemple :

```
.modal {  
  position: fixed; /* reste en place */  
  z-index: 2; /* au dessus du reste */  
  width: 800px; /* largeur */  
  height: 800px; /* hauteur */  
  overflow: auto; /* barre de scroll si besoin */  
  background-color: white; /* opaque */  
  border: solid;  
  top:10%; /* a 10% du haut */  
  left:20%; /* a 20% de la gauche */  
}
```

Une fois cela fait, on peut ajouter dans le composant, une partie destinée à être affichée sur demande, avec une condition qui vérifie s'il faut l'afficher ou pas. Voici un exemple de code qui réalise l'affichage des managers (cette partie suppose que vous avez un état local de type booléen qui détermine si la fenêtre des managers doit être affichée ou pas) :







```
<div> { showManagers &&  
<div class="modal">  
  <div>  
    <h1 class="title">Managers make you feel better !</h1>  
  </div>  
  <div>  
    world.managers.pallier.filter( manager => !manager.unlocked).map(  
manager =>  
    <div key={manager.idcible} className="managergrid">  
      <div>  
        <div className="logo">  
          <img alt="manager logo" className="round" src= {  
this.props.services.server + manager.logo} />  
        </div>  
      </div>  
      <div className="infosmanager">  
        <div className="managername"> { manager.name} </div>  
        <div className="managercible"> {
```

```

    this.props.world.products.product[manager.idcible-1].name } </div>
      <div className="managercost"> { manager.seuil} </div>
    </div>
    <div onClick={() => this.hireManager(manager)}>
      <Button disabled={this.props.world.money < manager.seuil}>
Hire !</Button>
    </div>
  </div>
)
    <button class="closebutton" (click)="showManagers =
!showManagers">Close</button>
  </div>
</div>
} </div>

```

Complétez le composant principal pour qu'il affiche la fenêtre des managers en fonction de ceux définis dans votre monde selon le modèle proposé Figure 8 (mais vous pouvez choisir un autre design si vous préférez)

	Wangari Maathai Paper Bags 1000	Hire !
	Ellen MacArthur Recycle Bins 15000	Hire !
	Pierre Rabhi Bicycles 100000	Hire !
	Nicolas Hulot Electrical Cars 500000	Hire !
	Jean-Yves Cousteau Wind Turbines 1200000	Hire !
	Shiva Vandana Solar Energy 10000000	Hire !

[Close](#)

Figure 8 – Liste des managers

Arrangez-vous également pour que le bouton d'engagement du manager (*Hire*) soit cliquable si l'argent possédé par le joueur est en quantité suffisante.

Enfin faites en sorte que si un manager est débloqué (propriété *unlocked* sur *true*), il n'apparaisse pas dans la liste.

b. Engagement d'un manager

Implémentez le gestionnaire d'évènement associé au *clic* sur le bouton d'engagement d'un manager. Ce gestionnaire doit :

- Vérifier que l'argent du joueur est suffisant pour acheter le manager en question.

- Retirer le coût du manager de l'argent possédé par le joueur.
- Positionner la propriété *unlocked* du manager à vrai. Pareillement pour la propriété *managerUnlocked* du produit lié à ce manager.
- Modifiez la fonction `calcScore()` du composant produit qui calcule le score toutes les dixièmes de seconde. Cette fonction doit en effet désormais relancer automatiquement la mise en production d'un produit dont le manager est débloqué. Elle doit aussi immédiatement lancer la production d'un produit dont le manager est débloqué, même s'il n'était pas déjà en production.

c. Afficher un message éphémère pour l'utilisateur

Nous voulons qu'un message d'information soit affiché au joueur lorsque qu'il vient d'engager un nouveau manager. Nous aurons aussi besoin de ce genre de feedback lorsque que le joueur débloquent des *unlocks* ou des *upgrades*, ou encore pour prévenir le joueur d'une mauvaise transmission d'informations entre le client et le serveur.

Vous pouvez utiliser pour cela le composant `SnackBar` de MUI dont vous trouverez la documentation ici :

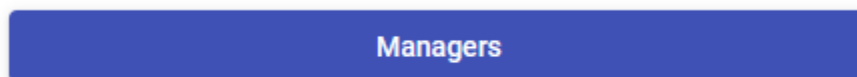
<https://mui.com/components/snackbars/>

Générez donc un tel message lors de l'embauche d'un nouveau manager.

d. Badger les boutons pour informer le joueur

MUI propose aussi un composant qui permet de « badger » certains éléments de l'interface de façon à modifier légèrement leur apparence. Quand il s'agit d'un bouton cliquable, ce visuel permet d'avertir le joueur qu'une action est disponible en cliquant sur ce bouton.

Par exemple voici le bouton d'accès à la liste des managers sans badge :



Et voici le même bouton badgé avec le nombre de managers achetable :



L'utilisation de ce composant est documentée sur cette page :

<https://mui.com/components/badges/>

Modifiez donc le composant principale (ou le composant supplémentaire que vous avez créé pour gérer les managers si c'est le cas) pour qu'à chaque évolution de l'argent du joueur, la possibilité d'acheter un nouveau manager soit vérifiée. Si c'est le cas, activez le badge donnant le nombre de managers accessibles sur le bouton des managers.

6. Retour coté serveur

Jusqu'à présent notre application est relativement autonome, la seule communication avec le serveur ayant lieu lors du chargement initial et consistant à obtenir le monde.

Dans cette partie nous allons nous attacher à transmettre au serveur les actions de l'utilisateur modifiant le monde de façon que ce dernier en gère la persistance (ainsi que l'évolution des gains du joueur en fonction du passage du temps).

a. Modifier le service web pour qu'il récupère le nom du joueur

Retournez dans la partie serveur du projet et dans son fichier `index.js`. Modifiez la création du serveur Apollo pour qu'il aille prélever l'entête `x-user` dans la requête http et qu'il injecte sa valeur dans le contexte. Voici la nouvelle version qu'il faut obtenir :

```
const server = new ApolloServer({
  typeDefs, resolvers,
  context: async ({ req }) => ({
    world: world,
    user: req.headers["x-user"]
  })
});
```

Cela va permettre aux résolveurs de récupérer le nom du joueur dans le contexte.

Nous allons justement modifier le résolveur de la *query* `getWorld` pour qu'avant de renvoyer le monde, elle en fasse une copie spécifique au joueur. Nous mettrons cette copie dans un dossier appelé `userworlds` de votre projet (créez ce dossier), et nous appellerons le fichier avec le nom du joueur suivi de « `-world.json` »

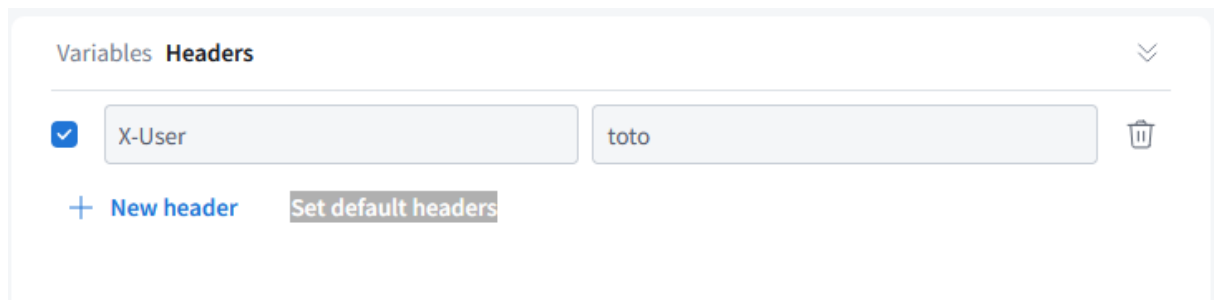
Ajoutez au début du fichier `resolvers.js`, le code qui réalise cette sauvegarde :

```
function saveWorld(context) {
  fs.writeFile("userworlds/" + context.user + "-world.json",
    JSON.stringify(context.world), err => {
      if (err) {
        console.error(err)
        throw new Error(
          `Erreur d'écriture du monde coté serveur`)
      }
    })
}
```

Et appelez ce code avant de retourner le monde :

```
getWorld(parent, args, context, info) {
  saveWorld(context)
  return context.world
}
```

Faites une requête avec l'interface de test, en utilisant cette interface pour ajouter le bon entête de la requête comme dans l'exemple ci-dessous (ici l'utilisateur s'appelle « toto »).



Exécutez la requête et vérifiez bien qu'un nouveau fichier apparaît dans le dossier `userworlds` de votre projet et qu'il contient bien une copie du monde au format json.

Ce que nous voulons à présent, c'est que la prochaine requête d'un même utilisateur, lui permette d'obtenir son monde à lui, et non plus celui d'origine. Il va donc falloir faire en sorte que le world qu'on injecte dans le contexte soit celui du joueur, et non pas celui contenu dans `world.js`.

Pour cela retournez dans le fichier `index.js`, et ajoutez au début une fonction qui tente de lire le monde associé au joueur. Si ce monde n'existe pas, on retourne le monde initial, sinon on retourne son monde à lui.

Voici cette fonction :

```
async function readUserWorld(user) {
  try {
    const data = await fs.readFile("userworlds/" + user + "-world.json");
    return JSON.parse(data);
  }
  catch(error) {
    return world
  }
}
```

Il ne reste plus qu'à l'appeler lors de la création du serveur Apollo :

```
const server = new ApolloServer({
  typeDefs, resolvers,
  context: async ({ req }) => ({
    world: await readUserWorld(req.headers["x-user"]),
    user: req.headers["x-user"]
  })
});
```

Et voilà, chaque joueur obtient le monde qui lui correspond, sauf la première fois qu'il se connecte auquel cas il obtient le monde initial. Vous pouvez tester qu'il n'y a pas d'erreur, mais vous ne pouvez toujours pas différencier le monde du joueur du monde initial puisque le monde n'évolue pas encore. C'est l'objet de la section suivante.

b. Prendre en compte les actions du joueur

Côté client nous avons jusqu'à présent codé trois actions du joueur :

- Lancement de la production d'un produit

- Achat d'une certaine quantité de produit
- Achat d'un manager

Nous allons implémenter les mutations GraphQL qui permettront au client de signaler ces actions au serveur. Comme nous l'avons précisé lors des spécifications page 9, les trois mutations en mettre en place sont :

```
acheterQtProduit(id: Int!, quantite: Int!): Product
lancerProductionProduit(id: Int!): Product
engagerManager(name: String!): Manager
```

Il faut donc programmer ces trois mutations dans le fichier `resolvers.js`.

Commençons par *acheterQtProduit*. Le résolveur doit aller chercher deux paramètres dans les arguments : `args.id` et `args.quantite` ; Il doit ensuite aller chercher le produit comportant cet identifiant dans le monde, sachant que s'il n'existe pas il faut générer une erreur sous la forme d'une exception :

```
throw new Error(
  `Le produit avec l'id ${args.id} n'existe pas`)
```

Une fois le produit trouvé, il faut augmenter sa quantité en ajoutant à la quantité actuellement possédée, la valeur de `args.quantite`. Mais cela ayant un coût, il faut déduire de l'argent du monde, le coût de l'achat. Ensuite il faut mettre à jour le coût d'achat du produit, puisque ce coût doit toujours représenter le coût d'achat du prochain produit. Enfin, il faut sauver le monde pour mémoriser les changements opérés.

Pour ce qui est de *lancerProductionProduit*, c'est plus simple. Il faut comme précédemment aller chercher le produit du monde possédant l'identifiant passé en paramètre, puis affecter sa propriété `vitesse` à sa propriété `timeleft`. Autrement dit quand le serveur apprend que le client a lancé la production d'un produit, il se contente de noter le temps qu'il reste au produit pour terminer sa production. On verra plus tard comment le serveur sait que la production est terminée.

Enfin pour *engagerManager*, il faut aller chercher dans le monde le manager de nom passé en paramètre, repérer de quel produit il est un manager, et aller chercher le produit du monde correspondant. Ensuite, on se borne à débloquer le manager en passant à vrai la propriété `managerUnlock` du produit, et la propriété `unlocked` du manager.

Il manque encore quelque chose au code du serveur. Certes celui-ci traite les actions de l'utilisateur mais il ne met pas encore à jour le score (l'argent gagné) du joueur. Contrairement au client qui met à jour le score tous les dixièmes de seconde pour des besoins d'affichage, le serveur n'a pas besoin de le faire aussi fréquemment.

Il lui suffit de le faire juste avant de traiter la prochaine action du joueur.

Ainsi, si par exemple le joueur annonce au serveur qu'il lance la production d'un produit, le serveur se contente de noter cette action (en réglant `timeleft` sur `vitesse`) et à quel moment elle survient (en réglant le `lastupdate` du monde sur le temps courant que l'on obtient avec l'expression javascript `Date.now()`).

Si un peu plus tard, le joueur annonce qu'il achète un exemplaire supplémentaire de ce produit, le serveur va évaluer le temps écoulé depuis. Si ce temps est supérieur ou égal au temps de production, il met à jour le score du joueur en ajoutant les gains. Ce n'est qu'ensuite qu'il traitera l'achat du joueur.

En résumé, à chaque fois que le serveur doit traiter une action du joueur (y compris le chargement initial du monde avec `getWorld`), il doit d'abord mettre à jour le score du joueur.

Il vous reste donc à écrire la méthode qui met à jour le score du joueur en fonction du temps qui s'est écoulé depuis la dernière mise à jour. Pour l'essentiel cette méthode doit parcourir chaque produit et pour chacun calculer combien d'exemplaires de ce produit a été créé depuis la dernière mise à jour.

Si ce produit n'a pas de manager, il suffit de vérifier que `timeleft` n'est pas nul, et qu'il est inférieur au temps écoulé. Si c'est le cas, un produit a été créé (et donc on ajoute les gains au score), sinon on met à jour `timeleft` en soustrayant le temps écoulé.

Si ce produit a un manager, c'est plus compliqué car il faut calculer combien de fois sa production complète a pu se produire depuis la dernière mise à jour, et mettre à jour le `timeleft` du produit en conséquence.

Le code final de cette méthode n'est pas très long mais vous demandera sans doute un peu de réflexion pour obtenir quelque chose qui fonctionne proprement. N'oubliez pas à chaque mis à jour de repositionner le `lastupdate` du monde sur l'instant courant. A ce propos, vous noterez que dans le schéma GraphQL `lastupdate` est un string et pas un entier. La raison c'est que `Date.now()` renvoie un entier codé sur 64bits et que les entiers GraphQL sont limités à 32 bits. Il faut donc convertir la valeur renvoyé par `Date.now()` en chaîne de caractère avant de la stocker dans `lastupdate`. Inversement, il faudra convertir en entier la valeur de `lastupdate`, avant de s'en servir coté serveur.

c. Appel des interfaces par le client

Il ne reste plus qu'à modifier le client pour qu'à chaque action du joueur, il appelle les interfaces de services que nous venons d'implémenter.

Commençons par la mise en production d'un produit. Cette mise en production intervient dans le fichier « `product.tsx` ». Dans ce fichier, définissez la mutation GraphQL que nous allons utiliser. La voici :

```
const LANCER_PRODUCTION = gql`
  mutation lancerProductionProduit($id: Int!) {
    lancerProductionProduit(id: $id) {
      id
    }
  }
`
```

Avec le client Apollo pour React, on appelle cette requête en utilisant le *hook* `useMutation`. Sa forme est la suivante :

```
const [lancerProduction] = useMutation(LANCER_PRODUCTION,
  { context: { headers: { "x-user": username } },
    onError: (error): void => {
      // actions en cas d'erreur
    }
  }
)
```

Cette forme ressemble beaucoup au *hook* `useQuery` que nous utilisons pour récupérer le monde. La grande différence est l'apparition d'une variable entre crochets qui sert à définir le nom de la fonction à appeler pour lancer la mutation. Ici nous avons choisi d'appeler cette variable `lancerProduction`. Remarquez que vous avez besoin de connaître le nom du joueur (`username`), faites donc en sorte que le composant `Main.tsx` le passe au composant produit au moyen d'une propriété (props). Attention contrairement au *hook* `useQuery`, la mutation n'est pas lancée, elle est simplement déclarée. Enfin remarquez qu'on peut passer à cette déclaration un callback `onError` qui permet de définir ce que l'on doit faire en cas d'erreur d'exécution.

Il ne reste plus qu'à exécuter la mutation au bon endroit dans le composant `Produit` en écrivant :

```
lancerProduction({ variables: { id: product.id } });
```

On voit que c'est quand on exécute la mutation que l'on précise la valeur de ses paramètres. Ici conformément au schéma GraphQL il n'y en a qu'un seul, l'id du produit ;

Voilà vous savez utiliser les mutations. La mutation `lancerProduction` est la seule qui se fait dans le composant `Produit`. Les autres se font dans le composant `Main.tsx` puisque c'est lui qui centralise l'achat d'une certaine quantité d'un produit, ainsi que l'engagement des managers. A vous de jouer pour rajouter l'exécution de ces deux mutations dans `Main.tsx` pour que le client prévienne le serveur lorsqu'un produit est acheté et un manager engagé.

A cet instant de l'énoncé, vous devez avoir un programme où client et serveur sont synchronisés. En particulier tout rechargement de la page web doit afficher le monde dans l'état où il était juste avant le reload. Si vous constatez un décalage entre le score donné par le serveur et celui calculé par le client, c'est que quelque chose ne va pas.

Vous aurez aussi probablement un problème à régler au niveau des barres de progression des productions de produits lors du *reload*. En effet (sauf si vous l'avez déjà prévu dès le départ), lors du démarrage du client (donc lors du chargement de la page), la progression des barres de production doit être fixée en fonction de la propriété `timeleft` des produits. Pour être plus précis, cette progression s'exprime en un pourcentage ramené entre zéro et un, et est donc égale à $(product.vitesse - product.timeleft) / product.vitesse$. Ainsi lors du chargement de la page, et pour tous les produits pour lesquels `timeleft` n'est pas nul, il faut positionner la barre de progression au bon endroit, et lancer l'animation pour qu'elle aille au bout (souvenez-vous qu'on peut fixer l'état initial du *hook* `useState`)

7. Les *unlocks*

a. Affichage des *unlocks*

Coté client ajoutez le code nécessaire à l'affichage des seuils attachés à chaque produit. La réalisation de cette partie ressemble énormément à la gestion des managers. La seule différence réside dans le

fait que les *unlocks* ne s'achètent pas, ils sont automatiquement débloqués quand la quantité de produits atteint le seuil qui leur est attaché. Le bonus associé aux *unlocks* peut être de type vitesse ou gain et l'affichage doit clairement faire apparaître le type et la quantité de bonus obtenu.

Tout comme pour les managers, les *unlocks* déjà débloqués ne doivent pas être affichés.

Concevez-donc une fenêtre de type *modal* qui sera ouverte lors d'un clic sur le bouton *unlock*. La fenêtre doit ressembler à celle affichée Figure 9. Si les *unlocks* sont trop nombreux, vous pouvez choisir de n'afficher que les *n* premiers, ou de n'afficher que le prochain seuil associé à chaque produit.

b. Prise en compte des *unlocks* par le client

Adaptez le code du client pour prendre en compte les seuils atteints. Pour réaliser cela, vous devez vérifier, à chaque nouvelle quantité de produit acheté, si un seuil spécifié par un *unlock* a été atteint.

Or il y a deux sortes d'*unlocks* :

- Ceux qui sont spécifiques à un produit et qui se déclenchent quand ce produit a atteint une certaine quantité.
- Ceux qui se déclenchent quand **tous** les produits ont atteint une certaine quantité (les *allunlocks*).

Dans tous les cas, l'application d'un *unlock* à un produit consiste :

- S'il s'agit d'un *boost* de revenu, il suffit de multiplier le revenu du produit par le bonus obtenu.
- S'il s'agit d'un *boost* de vitesse, il s'agit de diviser sa vitesse de production par le bonus.

Enfin quand un bonus de seuil est obtenu, prévenez le joueur en envoyant un message éphémère sur l'interface comme vous l'avez fait en cas d'achat d'un nouveau manager :



c. Prise en compte des *unlocks* par le serveur

Tout comme le client, le serveur doit vérifier où en sont les seuils à chaque nouvelle quantité de produit acheté par le joueur et les débloquer une fois le seuil atteint, en appliquant l'effet du seuil sur les produits. Les algorithmes en mettre en œuvre sont les mêmes que coté client, l'animation de la barre de progression en moins.

Dans le fichier « *resolvers.js* », modifiez donc la mutation `acheterQtProduit` pour qu'elle vérifie et applique les *unlocks* à chaque quantité de produit acheté.

Vérifiez que le score continue bien d'évoluer de la même façon coté client que coté serveur et que les *unlocks* sont bien pris en compte.

Want to maximize profits ? Get your investments to these quotas ! ✕



Don't forget your paper bag !

75

Paper Bags VITESSE x2



Give me some good bins !

20

Recycle Bins VITESSE x2



More Bicycles !

20

Bicycles VITESSE x2



These cars are wizzzzzz !

20

Electrical Cars VITESSE
x2



I feel like the wind !

20

Wind Turbines VITESSE
x2

Figure 9 – Liste des *unlocks*

8. Les *Cash upgrades*

a. Affichage des *upgrades*

Tout comme vous l'avez fait pour les managers, concevez une fenêtre modale présentant les prochains *upgrades* disponibles en cas de clic sur le bouton « Cash Upgrades ».

Nous ne donnerons pas plus d'explications ici, le travail à faire étant très proche de celui réalisé pour l'affichage des managers et des *unlocks*. Notez simplement que les *upgrades* doivent pouvoir être achetés par le joueur, il faudra donc prévoir un bouton prévu à cet effet comme l'illustre la Figure 10. Là encore, n'hésitez pas à n'afficher que les n premiers *upgrades* si la liste est trop longue.

Prévoyez également un badge qui viendra apparaître sur le bouton « upgrades » quand le joueur possèdera la somme nécessaire pour acheter au moins un des upgrades non encore débloqués.

Cash Upgrades 6

b. Prise en compte des upgrades par le client.

Modifiez le code du client pour appliquer aux produits concernés le bonus obtenu en cas d'achat d'un *upgrade*. Comme pour les *unlocks*, le bonus peut être de type *gain* ou *vitesse* (il peut sans doute être aussi de type *ange* mais nous n'avons pas encore implémenté les anges à cet endroit de l'énoncé).

L'application des bonus dus aux upgrades doit être en tout point identique à l'application des bonus dus aux *unlocks*. Une grande partie du code doit donc être repris de la partie précédente. Notez que tout comme pour les *allunlocks*, certains upgrades s'appliquent à tous les produits.

c. Transmission des upgrades du client au serveur.

Quand le joueur achète un upgrade, il faut transmettre cette action au serveur. D'après la spécification, cette action doit être transmise par la mutation :

```
acheterCashUpgrade(name: String!): Palier
```

Implémentez donc ce point d'accès du côté du serveur, et faites en sorte que le client l'appelle quand le joueur achète un upgrade.

d. Prise en compte des upgrades par le serveur.

Quand le serveur récupère une action de type *upgrade* via son interface de service web, il doit appliquer l'upgrade sur le ou les produits concernés. Le code réalisant cela est en grande partie identique à celui consistant à appliquer l'effet d'un *unlock*. Cette partie ne devrait donc pas présenter de problèmes particuliers.

A la fin, vérifiez que l'évolution du score continue à être synchronisée entre le serveur et le client avec prise en compte des upgrades.






	<p>A nice bicycle</p> <p>15000 \$</p> <p>Bicycles Profits x3</p>	Buy !
	<p>I want this car !</p> <p>100000 \$</p> <p>Electrical Cars Profits x3</p>	Buy !
	<p>Don't laugh ! Just buy !</p> <p>200000 \$</p> <p>Wind Turbines Profits x3</p>	Buy !
	<p>A big advance</p> <p>3.000 10⁶ \$</p> <p>Solar Energy Profits x3</p>	Buy !
	<p>I want it all !</p> <p>3.500 10⁶ \$</p> <p>All Products Profits x3</p>	Buy !

Figure 10 – Liste des *upgrades*.

9. Gestion des anges

Comme nous l'avons précisé lors de la description du *gameplay*, le joueur a la possibilité d'accumuler des anges pendant la partie. Le nombre d'ange gagné dépend de l'argent accumulé par le joueur. On parle ici de ses revenus totaux, pas de son argent actuel. C'est la raison pour laquelle le monde possède une propriété `score` qui représente l'argent total gagné par le joueur depuis le début de la partie. Le nombre d'ange gagné dépend donc directement de ce score.

Les anges n'ont aucun effet tant que le joueur ne remet pas la partie à zéro. Quand cela se produit, les anges gagnés commencent alors à procurer un bonus de 2% par ange aux revenus. Certains de ces anges peuvent également être dépensés en Angel Upgrade pour procurer des bonus supplémentaires **en remplacement** du bonus de 2% qu'ils apportaient.

C'est pourquoi le monde possède également une propriété `totalangels` qui représente les anges accumulés depuis le début de la partie, mais également une propriété `activeangels` qui représente les anges actuellement actifs. La différence entre ses deux propriétés représente les anges dépensés en *angel upgrades*.

Ainsi le nombre d'anges **supplémentaires** gagnés par la partie en cours est donc égal à :

$$\text{Nombre d'anges} = 150 * \sqrt{\frac{\text{score}}{10^{15}}} - \text{totalangels}$$

a. Gestion des anges coté client

Implémentez le clic sur le bouton « Investors ». Ce bouton doit afficher une fenêtre donnant le nombre d'anges actuellement actifs, ainsi que le nombre d'anges supplémentaires accumulés par la partie en cours. Cette fenêtre doit également proposer un bouton permettant de remettre à zéro la partie et donc de récupérer les anges supplémentaires. La Figure 11 illustre à quoi doit ressembler cette fenêtre.

En cas de click sur le bouton de « reset », le client doit prévenir le serveur en utilisant la mutation suivante :

```
resetWorld: World
```

Suite à cette requête, le client doit remettre le monde dans l'état initial. Le plus simple pour réaliser cela est de demander un *reload* de la page et de laisser le serveur resservir un monde vierge.

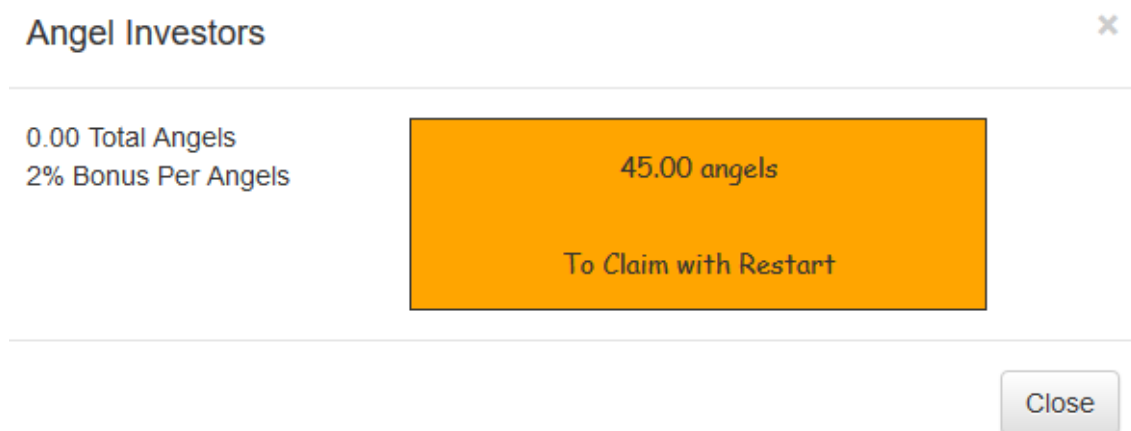


Figure 11 – Interface de gestion des anges

b. Gestion des anges coté serveur

Implémentez la mutation prévue pour prendre en compte la demande de reset du monde. Cette action doit essentiellement avoir pour effet :

- D'accumuler les anges supplémentaires gagnés lors de la partie en cours.
- De remettre le monde dans son état initial en conservant néanmoins son score, et ses deux propriétés relatives aux anges.

Pour le premier point on ajoutera simplement les anges gagnés aux propriétés du monde `totalangels` et `activeangels`.

Pour le deuxième point, le plus simple pour remettre à zéro est de repartir du monde initial (celui servi à un nouveau joueur) et d'initialiser ses propriétés `score`, `totalangels` et `activeangels` aux mêmes valeurs que le monde en cours de reset.

c. Prise en compte des anges dans les gains

Enfin il reste à modifier les fonctions de calcul du score, aussi bien chez le serveur que chez le client, pour qu'elles appliquent le bonus de revenu par ange actif. Ce bonus pouvant évoluer, on le trouvera dans la propriété `angelbonus` du monde (initialement il est fixé à 2%).

Le revenu gagné par la production d'un produit est alors de :

$$\text{product.quantite} * \text{product.revenu} * (1 + \text{world.activeangels} * \text{world.angelbonus} / 100)$$





10. Gestion des *Angel Upgrades*

Il ne reste plus qu'à mettre en place l'interface d'achat des *Angel Upgrades* comme illustré par la Figure 12.

Le badge « new » du bouton « Angel Upgrades » sera activé quand le joueur aura accumulé le nombre d'anges nécessaires à l'achat d'au moins 1 *upgrade*.

Spend your Angels Wisely !

×

	Angel Sacrifice 10.00 angels All Products Profits x3	Buy !
	Angelic Mutiny 100000 angels Angel Effectiveness + 2%	Buy !
	Angelic Rebellion 1.000 10⁸ angels Angel Effectiveness + 2%	Buy !
	Angelic Selection 1.000 10⁹ angels All Products Profits x5	Buy !

Close

Figure 12 - Liste des Angel Upgrades

a. Prise en compte des *angel upgrades* par le client.

Lors de l'achat d'un *angel upgrade*, le client doit appliquer l'upgrade sur les anges ou les produits concernés, selon le type de cet upgrade.

S'il s'agit d'un upgrade de type ANGE, alors on augmentera le bonus de production apporté par les anges selon la quantité de bonus de l'upgrade.

Sinon on réutilisera le code déjà en place pour appliquer soit un bonus de vitesse, soit un bonus de revenu.

Dans tous les cas, on décrémentera le nombre d'anges actifs du coût de l'upgrade.

On n'oubliera pas également d'appeler le point d'accès serveur qui correspond à la méthode PUT /angelupgrade pour communiquer au serveur l'achat d'un *Angel Upgrade* en passant cet upgrade en paramètre sous la forme d'une entité de type « palier ».

b. Prise en compte des *angel upgrades* par le serveur.

Implémentez la mutation qui permet au serveur de réceptionner un *angel upgrade*, à savoir :

```
acheterAngelUpgrade(name: String!): Palier
```

Le serveur doit réaliser les mêmes opérations que le client, à savoir décrémenter le nombre d'anges actifs du coût de l'upgrade, et appliquer le bonus apporté par ce dernier.

Comme d'habitude vérifiez que tout fonctionne, que les upgrades sont bien appliqués, et que serveur et client continuent de faire évoluer le score de la même façon.

11. Finalisation et branchement sur un autre monde

Finaliser votre monde en donnant les spécifications complète de six produits et en testant que le jeu est intéressant du point de vue de la croissance des revenus (qui ne doit être ni trop rapide, ni trop lente).

Essayez également de vous brancher sur un autre monde, par exemple un monde conçu par un de vos camarades.

Il vous faut pour cela obtenir l'adresse du serveur hébergeant ce monde, et de modifier l'adresse du serveur dans vos fichiers clients. Pour que cela soit plus facile, vous avez sûrement intérêt à stocker une seule fois cette adresse, par exemple dans le fichier `utils.js` :

```
export const server = "http://localhost:4000/"
```

puis à importer ce fichier là où vous en avez besoin, pour pouvoir utiliser la variable `server`.

Si votre partie cliente est compatible avec la partie serveur de l'autre groupe, les communications devraient fonctionner correctement.

Si de plus les codes sont exempts de bugs, le score calculé par ce serveur, et le score calculé par le client devrait être *relativement* identique, le *relativement* venant du fait qu'il est normal qu'une certaine dérive puisse apparaître avec le temps, du fait de micro-différences temporelles entre les calculs du client et du serveur. Au final c'est le serveur qui fait foi puisque c'est sur lui que se recalcule le client à chaque rechargement du jeu.

Vous êtes allés au bout du projet, félicitation !

Vous avez fait preuve de solides compétences en développement en ce qui concerne les langages Java, JavaScript (et TypeScript), HTML et CSS. Vous avez créé des services web au format GraphQL et utilisé le framework React.

Vous avez de plus exercé vos compétences en calculs mathématiques et avez travaillé sur des problématiques de synchronisation temporelle entre un serveur et ses clients.