

plate

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	4
	Principes du jeu « Adventure Capitalist ».....	4
	Gameplay.....	4
2.	Conception du serveur de mondes	11
a.	Spécifications de l'API du serveur	11
b.	Début du travail sur le serveur.....	14
c.	Création du monde.....	16
d.	Sérialisation et dé-sérialisation java du monde	18
e.	Création du service web servant le monde	18
f.	Autoriser le serveur à répondre au requêtes cross-domain	Erreur ! Signet non défini.
3.	Début du travail sur le client web	19
a.	Serveur de test	19
b.	Création du projet React.....	19
c.	Description du travail attendu	20
d.	Création des premiers composants et du service de communication avec le service web ..	22
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.	Permettre au joueur de spécifier son nom	37
b.	Modifier le service web pour qu'il récupère le nom du joueur	38
c.	Servir au joueur son propre monde	38

d.	Prendre en compte les actions du joueur	39
e.	Appel des interfaces par le client	42
7.	Les <i>unlocks</i>	42
a.	Affichage des <i>unlocks</i>	42
b.	Prise en compte des <i>unlocks</i> par le client.....	43
c.	Prise en compte des <i>unlocks</i> par le serveur.....	44
8.	Les <i>Cash upgrades</i>	45
a.	Affichage des <i>upgrades</i>	45
b.	Prise en compte des <i>upgrades</i> par le client.	45
a.	Transmission des upgrades du client au serveur.	46
b.	Prise en compte des <i>upgrades</i> par le serveur.....	46
9.	Gestion des anges	47
a.	Gestion des anges coté client	47
b.	Gestion des anges coté serveur	48
c.	Prise en compte des anges dans les gains	48
10.	Gestion des <i>Angel Upgrades</i>	48
a.	Prise en compte des <i>angel upgrades</i> par le client.....	49
b.	Prise en compte des <i>angel upgrades</i> par le serveur.	50
11.	Finalisation et branchement sur un autre monde.....	50
	Annexes.....	50
	Débuguer avec Visual Studio Code et Chrome	50

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 :

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

New!

\$ 5.12

5

0

0

Buy

X

Career

Managers

?

Managers make life easier!
Hire one to run your business for you, or to maximize efficiency, all for just one easy payment! Salary schmalary!

7 Angels

	Cabe Johnson Runs Lemonade Stands 1,000.00	Hire!
	Perry Black Runs Newspaper Deliveries 15,000.00	Hire!
	W.W. Heisenbird Runs Car Washes 100,000.00	Hire!
	Mama Sean Runs Pizza Deliveries 500,000.00	Hire!
	Jim Thorton Runs Donut Shops 1,200,000.00	Hire!
	Forest Trump Runs Shrimp Boats 10,000,000.00	Hire!
	Dawn Cheri Runs Hockey Teams 111,111,111.00	Hire!
	Stefani Speilburger Runs Movie Studios	Hire!

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 RESTful. Les points d'accès de ce service seront :

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

PUT /product : permet au client de communiquer au serveur une action sur l'entité de type « produit » passé en paramètre. Cette action sera soit l'achat d'une certaine quantité de ce produit, soit le lancement manuel de la production de ce produit.

PUT /manager : permet au client de communiquer au serveur l'achat du manager d'un produit, en passant le manager en question en paramètre sous la forme d'une entité de type « pallier ».

PUT /upgrade : permet au client de communiquer au serveur l'achat d'un *Cash Upgrade* en passant cet upgrade en paramètre sous la forme d'une entité de type « pallier ».

PUT /angelupgrade : permet au client de communiquer au serveur l'achat d'un *Angel Upgrade* en passant cet upgrade en paramètre sous la forme d'une entité de type « pallier ».

DELETE /world : permet au client de demander le *reset* du monde.

La **représentation** des entités échangées se fera soit au format XML, soit au format JSON, selon le format demandé par le client.

Pour ce qui est du monde Cette représentation est définie par une entité de type « monde » (worldType) telle que définie par le schéma XML donné Figure 7.

Ce schéma donne également la définition des entités « produit » (productType) et « pallier » (pallierType) qui sont utilisées par les requêtes de type PUT lorsque le client indique au serveur l'action réalisée par le joueur.

Quelques explications doivent être données pour éclaircir le fonctionnement de l'entité « pallier ». Celle-ci 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 (seuil ou encore *unlock*).

```

<xs:schema version="1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xs:element name="world">
    <xs:complexType>
      <xs:sequence>
        <!-- Titre du monde -->
        <xs:element name="name" type="xs:string"/>
        <!-- logo du monde -->
        <xs:element name="logo" type="xs:string"/>
        <!-- argent actuel du joueur -->
        <xs:element name="money" type="xs:double"/>
        <!-- argent cumulé par le joueur depuis le début -->
        <xs:element name="score" type="xs:double"/>
        <!-- total cumulé des anges gagnés par le joueur lors de chaque reset -->
        <xs:element name="totalangels" type="xs:double"/>
        <!-- nombre d'anges actifs (chiffre précédent moins les anges dépensés) -->
        <xs:element name="activeangels" type="xs:double"/>
        <!-- bonus par ange (2% par défaut) -->
        <xs:element name="angelbonus" type="xs:int"/>
        <!-- dernière mise à jour du monde par le serveur -->
        <xs:element name="lastupdate" type="xs:long"/>
        <!-- liste des types de produit -->
        <xs:element name="products" type="productsType"/>
        <!-- liste des seuils concernant tous les produits -->
        <xs:element name="allunlocks" type="palliersType"/>
        <!-- liste des cash upgrades -->
        <xs:element name="upgrades" type="palliersType"/>
        <!-- liste des angel upgrades -->
        <xs:element name="angelupgrades" type="palliersType"/>
        <!-- liste des managers -->
        <xs:element name="managers" type="palliersType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="productsType">
    <xs:sequence>
      <xs:element name="product" type="productType" minOccurs="2" maxOccurs="6"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="productType">
    <xs:sequence>
      <!-- identifiant unique du produit, à partir de 1 -->
      <xs:element name="id" type="xs:int"/>
      <!-- nom du produit -->
      <xs:element name="name" type="xs:string"/>
      <!-- chemin relatif vers une image du produit -->
      <xs:element name="logo" type="xs:string"/>
      <!-- cout d'achat d'un exemplaire supplémentaire du produit -->
      <xs:element name="cout" type="xs:double"/>
      <!-- multiplicateur d'augmentation du prix du produit par exemplaire,
        au format 1,xx, par exemple 1,07 indique une augmentation de 7%
        par exemplaire.
      -->
      <xs:element name="croissance" type="xs:double"/>
      <!-- revenu actuel du produit -->
      <xs:element name="revenu" type="xs:double"/>
      <!-- nombre de milli-secondes nécessaire pour créer le produit -->
      <xs:element name="vitesse" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

```

```

    <!-- quantité actuelle du produit détenu par le joueur -->
    <xs:element name="quantite" type="xs:int"/>
    <!-- temps restant pour terminer la création du produit en millisecondes -->
    <xs:element name="timeleft" type="xs:long"/>
    <!-- booléen qui indique si le manager de ce produit est débloqué ou pas -->
    <xs:element name="managerUnlocked" type="xs:boolean"/>
    <!-- liste des seuils propres à ce produit -->
    <xs:element name="palliers" type="palliersType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="palliersType">
  <xs:sequence>
    <xs:element name="pallier" type="pallierType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="pallierType">
  <xs:sequence>
    <!-- identifiant du pallier -->
    <xs:element name="name" type="xs:string"/>
    <!-- chemin relatif menant à une image représentative du pallier -->
    <xs:element name="logo" type="xs:string"/>
    <!-- valeur du pallier à atteindre quand il s'agit d'un seuil, ou coût de l'upgrade sinon -->
    <xs:element name="seuil" type="xs:double"/>
    <!-- produit cible de l'upgrade, 0 s'il s'agit de tous les produits, -1 si c'est un effet sur un ange -->
    <xs:element name="idcible" type="xs:int"/>
    <!-- bonus obtenu sous la forme d'un multiplicateur de vitesse ou de revenu, ou de pourcentage d'ange -->
    <xs:element name="ratio" type="xs:double"/>
    <!-- type de bonus parmi VITESSE, GAIN, ou ANGE -->
    <xs:element name="typeratio" type="typeratioType"/>
    <!-- indicateur de déblocage de ce pallier -->
    <xs:element name="unlocked" type="xs:boolean"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="typeratioType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="vitesse"/>
    <xs:enumeration value="gain"/>
    <xs:enumeration value="ange"/>
  </xs:restriction>
</xs:simpleType>

</xs:schema>

```

Figure 7 – Schéma XML du monde

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 débloquer le manager ;
 - idcible : identifiant du produit géré par le manager ;
 - ratio, typeratio : non utilisés ;
 - unlocked : vrai ou faux selon que le manager est débloqué ;
- « 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. Début du travail sur le serveur

- Rendez-vous sur le site officiel de Spring Boot (<https://start.spring.io/>) pour télécharger un squelette de projet au format zip incluant les dépendances Spring Web (pour les web services) et DevTools (pour le *hot reload* du projet pendant le développement). Vous pouvez appeler le projet comme vous voulez (sur la capture écran qui suit, il s'appelle *ISISCapitalist*).
- Ouvrez ce projet sous votre éditeur Java préféré (Netbeans, IntelliJ ou Eclipse) et compilez-le pour ramener les dépendances nécessaires.

Project

☒ Maven Project
 ☐ Gradle Project

Language

☒ Java
 ☐ Kotlin
 ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT)
 ☐ 3.0.0 (M1)
 ☐ 2.7.0 (SNAPSHOT)
 ☐ 2.7.0 (M1)
 ☐ 2.6.4 (SNAPSHOT)
 ☒ 2.6.3
 ☐ 2.5.10 (SNAPSHOT)
 ☐ 2.5.9

Project Metadata

Group

com.example

Artifact

ISISCapitalist

Name

ISISCapitalist

Description

Demo project for Spring Boot

Package name

com.example.ISISCapitalist

Packaging

☒ Jar
 ☐ War

Java

☐ 17
 ☒ 11
 ☐ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Boot DevTools

DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

- Récupérez le schéma de la Figure 7 (disponible en téléchargement sur moodle) et sauvez-le dans un fichier que vous placerez dans le dossier `src/main/xsd` (il faudra créer le dossier `xsd` qui n'existe pas au départ) de votre projet.

La première chose à faire est de créer les classes Java qui correspondent au schéma XML que nous allons utiliser pour générer les entités qui seront échangées entre les clients et notre serveur. Nous avons vu que c'était le *framework* JAXB qui permettait cela.

Ajoutez dans le `pom.xml` les dépendances nécessaires pour utiliser JAXB :

```
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.3</version>
</dependency>

<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-core</artifactId>
  <version>2.3.0.1</version>
</dependency>
```

Toujours dans le `pom.xml`, ajoutez le plugin maven qui permettra de compiler le schéma en classes java :

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>2.5.0</version>
  <executions>
    <execution>
      <id>xjc</id>
      <goals>
        <goal>xjc</goal>
      </goals>
    </execution>
  </executions>
```

```
</execution>
</executions>
</plugin>
```

Activez le *goal* « jaxb2:xjc » du plugin pour compiler le schéma. Pour cela vous pouvez utiliser le menu relatif à Maven de votre IDE (vous trouverez le *goal* dans la partie plugin). Si vous ne trouvez pas ce menu, vous pouvez l'activer en ligne de commande en tapant dans le terminal du projet :

```
mvnw jaxb2:xjc1
```

Maintenant que nos classes Java sont en place, nous pourrions les utiliser pour créer notre monde en programmant leur instanciation sous la forme d'objets produits, paliers, managers, upgrades, etc. Cette solution serait fonctionnelle mais nous pouvons faire mieux en créant le monde sous la forme d'une instance XML du schéma, puis en désérialisant cette instance avec JAXB ce qui produira les instances Java automatiquement.

C'est ce que nous allons faire dans la section suivante.

c. Création du monde

Créez une représentation de votre monde dans un fichier XML conforme au schéma de la Figure 7 (vous pouvez l'appeler `world.xml` par exemple) et placez ce fichier dans le dossier `src/main/resources` de votre projet.

Pour aller plus vite vous pouvez générer un gabarit pour ce fichier à partir du schéma en utilisant un générateur en ligne (par exemple <http://xsd2xml.com/>) ou selon votre IDE les menus de création de fichiers XML. Par exemple sous NetBeans on peut passer par le menu New File/XML/XML Document de Netbeans et spécifier le schéma sur lequel le fichier se base. L'IDE va de la sorte pré-remplir le fichier XML avec les balises adéquates.

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

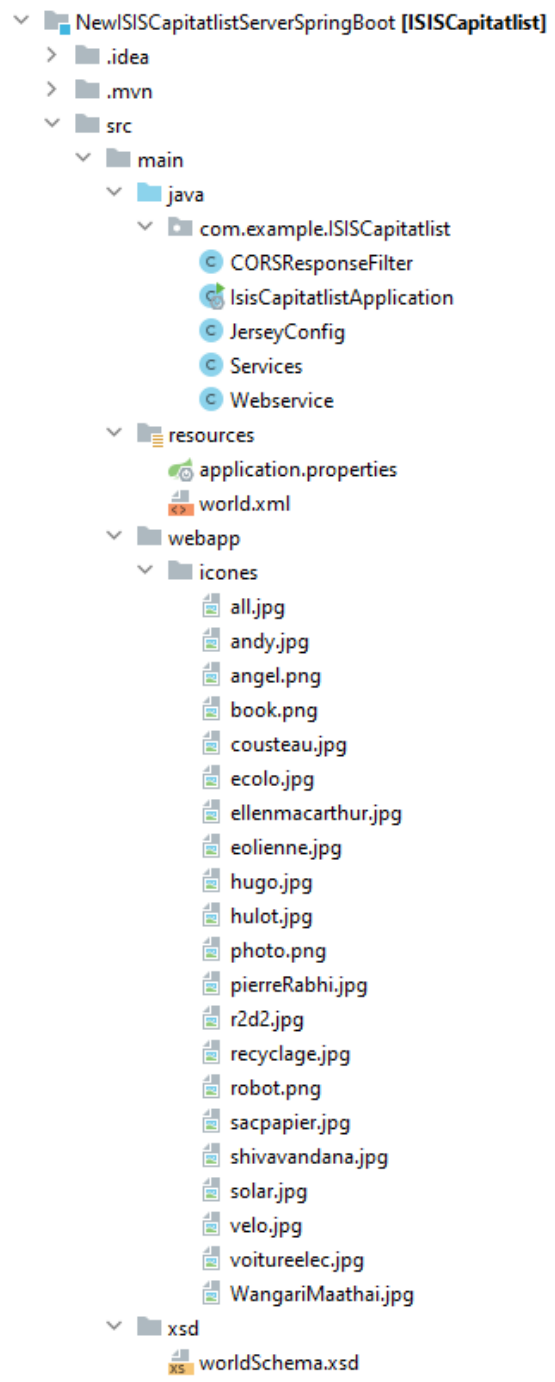
- Au moins deux produits qui composeront votre monde (il faudra en faire six au final mais arrêtez-vous à deux pour commencer le développement).
- Pour chacun de ces produits, définir ses caractéristiques de nom, icone, revenu, cout, croissance, etc.
- Pour chacun de ces produits, définir les différents 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.

¹ Si vous êtes sous Windows, il faudra plutôt taper la commande `mvnw.cmd` qui ne fonctionnera que si la variable `JAVA_HOME` pointe bien sur l'installation de votre jdk.

N'oubliez pas de réaliser une copie de sauvegarde de ce fichier au cas où (que vous placerez hors du projet).

Vous placerez les fichiers de type images correspondant aux produits, managers, upgrades... dans un dossier icones que vous créerez dans la partie « Web Pages » du projet (il s'agit du dossier `webapp` situé dans le dossier `src/main` du projet. Créez-le s'il n'existe pas déjà). Le fait de les mettre à cet endroit, permettra aux clients d'y accéder au moyen de l'URL `http://localhost:8080/icones/image.jpg`.

Voici par exemple l'arborescence permettant de placer les images dans le projet :



d. Sérialisation et dé-sérialisation java du monde

Dans votre projet créez une classe appelée `Services.java` que vous doterez de trois méthodes :

- `World readWorldFromXml()`
- `void saveWorldToXml(World world)`
- `World getWorld()`

La première doit lire le fichier XML conçu dans la section précédente et le retourner sous la forme d'un objet Java de type `World` (opération de *unmarshall*). La seconde doit réaliser l'opération symétrique (*marshall*). Vous utiliserez le framework JAXB pour réaliser ces opérations comme nous l'avons vu dans le TP JAXB.

Note importante : Le fichier `world.xml` contenant la représentation XML de votre monde se trouve initialement dans le dossier `resources` des sources de votre projet. Quand le projet est compilé et construit, l'ensemble de ses données est déployé sur le serveur de diffusion. Pour y accéder, java propose une méthode utilitaire qui retourne un `InputStream` pointant sur `world.xml`. Le code est le suivant :

```
InputStream input =  
getClass().getClassLoader().getResourceAsStream("world.xml");
```

Pour créer un `OutputStream` qui sera utilisé par `saveWorldToXml(World world)`, c'est plus facile car il suffira d'utiliser l'instruction :

```
OutputStream output = new FileOutputStream(file);
```

et le serveur créera le fichier à l'endroit où cela lui convient le mieux (avec Spring Boot, ce sera dans le dossier racine de votre projet).

Enfin faites en sorte que la méthode `getWorld()` se contente pour l'instant d'appeler et de retourner la méthode `readWorldFromXml()`. Vous verrez plus tard quel est l'intérêt de cette méthode intermédiaire.

e. Création du service web servant le monde

Nous voulons que l'appel `GET /world` retourne au client une représentation complète de l'état du monde au format XML.

Créez une nouvelle classe java du nom de `Webservices.java` et dotez là du squelette suivant :

```
@RestController  
@RequestMapping("adventureisis/generic")  
@CrossOrigin  
public class Webservice {  
  
    Services services;  
  
    public Webservice() {  
        services = new Services();  
    }  
  
    @GetMapping(value = "world", produces = {"application/xml",  
        "application/json"})
```

```

    public ResponseEntity<World> getWorld() {
        World world = services.getWorld();
        return ResponseEntity.ok(world);
    }
}

```

Ce point d'accès web répond à une requête de type GET sans paramètres et produit le monde au format XML tel que produit par un appel à `readWorldFromXml()` de `Services.Java`.

Lancez le projet et vérifiez que le service web fonctionne en pointant votre navigateur vers son adresse. Si vous avez spécifié les mêmes annotations de chemin que sur l'énoncé, l'URL doit être :

`http://localhost:8080/adventureisis/generic/world`

Pour tester le retour en JSON il faut donc faire une requête GET vers l'URL `/world` mais en demandant une réponse au format JSON. Vous pouvez pour cela utiliser un plugin de votre navigateur (RESTED pour firefox par exemple) ou directement la console réseau, et spécifier le header `http accept : application/json`. La présence de ce *header* provoquera l'appel automatique du point d'accès générant du JSON.

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>

Vous pouvez vérifier qu'il fonctionne en allant chercher le monde en utilisant l'URL suivant et en faisant un GET dessus :

<https://isiscapitalist.kk.kurasawa.fr/adventureisis/generic/world>

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.

L'installation d'un projet React est facilité par l'installation de node.js (<https://nodejs.org>). 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 8 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.

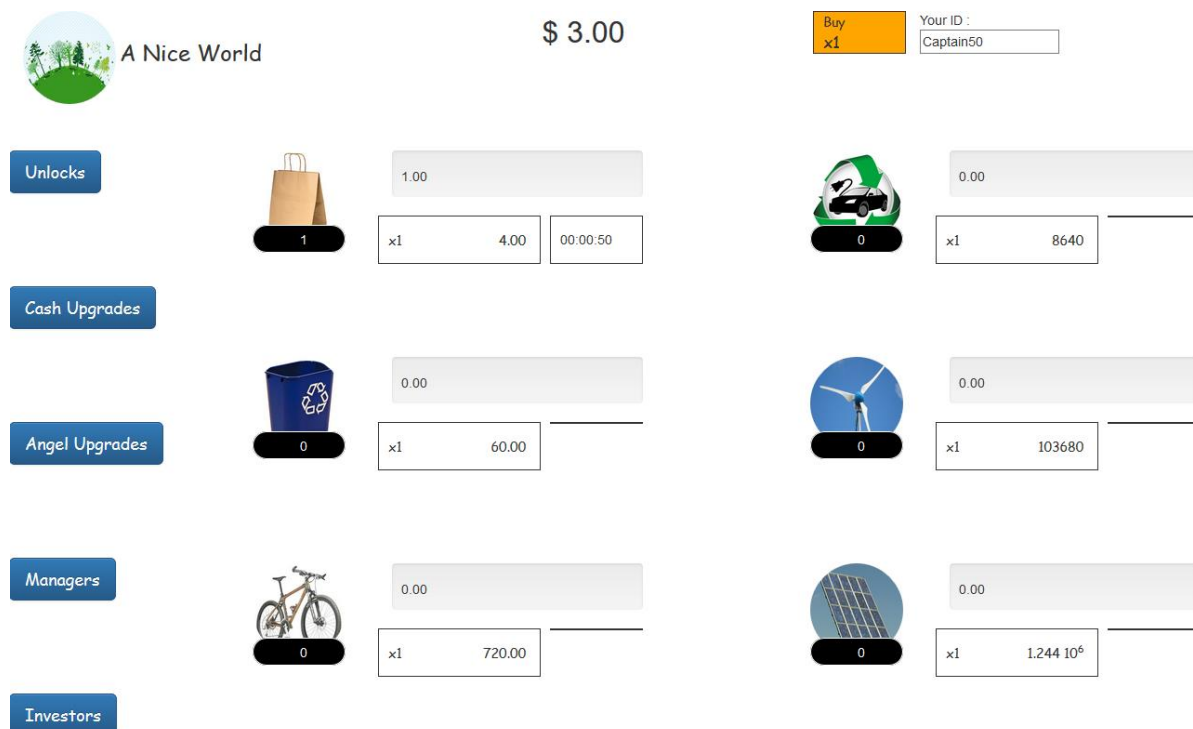


Figure 8 – Illustration de l'interface du jeu

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 de définir tous les éléments globaux de l'IHM (icône et nom du monde, argent, multiplicateur d'achat, bandeau de droite avec les upgrades, etc.), mais délèguera à un second (`Product.tsx`) que nous créerons plus tard le soin de gérer l'affichage des éléments d'un produit.

Nous aurons également besoin de faire des appels au serveur au format REST. Pour cela on peut utiliser une bibliothèque javascript du nom d'Axios. Installez sa dépendance en tapant :

```
npm install axios
```

Pour gérer les appels REST vers notre serveur, nous allons définir une classe qui contiendra toutes les méthodes d'appels. Ce service va donc réceptionner et mettre à jour les différents éléments du monde. Pour représenter ce monde et ses objets, il nous faut l'équivalent typescript des classes Java qui ont été générées à partir du schéma XML. Malheureusement, il n'existe pas encore d'outils TypeScript permettant de réaliser cela (à l'heure de l'écriture de cet énoncé en tout cas). Il faut donc définir manuellement une classe `World` en TypeScript qui correspond à chaque élément du schéma du monde. Ce travail un peu fastidieux vous est fourni sous la forme d'un fichier `world.ts`, disponible sur moodle, et 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 xsd 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": Product[] };
  allunlocks: { "pallier": Pallier[] };
  upgrades: { "pallier": Pallier[] };
  angelupgrades: { "pallier": Pallier[] };
  managers: { "pallier": Pallier[] };
}

```

```

        constructor() {
            this.products = { "product":[ ] } ;
            this.managers = { "pallier":[ ] };
            this.upgrades = { "pallier":[ ] };
            this.angelupgrades = { "pallier":[ ] };
            this.allunlocks = { "pallier":[ ] };
        }
    }

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
    managerUnlocked: boolean = false
    palliers : { "pallier" : Pallier[] };

    constructor() {
        this.palliers = { "pallier": [ ] }
    }
}

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, créez un fichier Services.ts qui définira toutes les méthodes permettant d'appeler notre serveur REST. Nous verrons plus tard que les appels devront comporter un nom d'utilisateur (pour distinguer le joueur qui s'adresse au serveur), c'est pourquoi le constructeur de cette classe prend en paramètre ce nom.

```

export class Services {

    server = "http://localhost:8080/"
    api = this.server + "adventureisis/generic";
    user = "";

    constructor(user: string) {
        this.user = user
    }
}

```

Vous remarquerez que nous avons inséré dans cette classe, deux propriétés qui représentent respectivement l'URL racine du serveur, et l'URL menant à l'API REST (vous ajusterez au besoin ces

valeurs si vous avez paramétré différemment votre serveur ou si elle hébergé ailleurs que sur votre machine locale)

Ajoutez à présent une méthode `getWorld()` qui réalisera l'appel GET `/world` au service web. Voici à quoi peut ressembler le code de cette méthode. Vous remarquerez que nous avons ajouté une méthode à part pour traiter une erreur éventuelle lors de l'appel au service, ainsi qu'une partie permettant de préciser le nom du joueur dans l'entête des requêtes :

```
export class Services {

    server = "http://localhost:8080/"
    api = this.server + "adventureisis/generic";
    user = "";

    constructor(user: string) {
        this.user = user
    }

    private static handleError(error: AxiosError): AxiosPromise<any> {
        console.error('An error occurred', error.toJSON());
        return Promise.reject(error.message || error);
    }

    private static setHeaders(user : string)    {
        return {
            "X-User": user
        }
    }

    getWorld(): AxiosPromise<World> {
        return axios({
            method: 'get',
            url: this.api + '/world',
            headers: Services.setHeaders(this.user)
        }).catch(Services.handleError)
    }
}
```

Pour permettre à notre composant React principal (`App.jsx`) d'utiliser cette classe, nous allons en faire un état du composant. Placez vous dans le fichier `App.jsx` et ajoutez un *hook* d'état pour stocker une instance de cette classe comme suit :

```
const [services, setServices] = useState(new Services(""))
```

Vous remarquerez que par défaut le service est créé avec un nom de joueur vide.

Un autre état donc nous aurons évidemment besoin, c'est l'état du monde. Ajoutez donc un autre état comme ceci :

```
const [world, setWorld] = useState(new World())
```

Ce monde initial est vide, puisque toutes ses propriétés sont initialisées avec des listes vides, des zéros, ou des chaînes de caractères vides (voir le fichier `world.ts`)

Pour démarrer avec un monde initialisé avec la version détenue par le serveur, nous devons appeler la méthode `getWorld()` de notre classe `services`.

Nous avons vu qu'en React, pour exécuter du code qui s'exécute avant le premier rendu du composant, on pouvait utiliser le *hook* `useEffect()`. Voici comment on peut l'écrire

```
:  
  
useEffect(() => {  
  
    let services = new Services(username)  
    setServices(services)  
    services.getWorld().then(response => {  
        setWorld(response.data)  
    })  
}, [])
```

Ce *hook* crée un nouvel objet de type `Service`, l'assigne à l'état `services` du composant avec `setServices(services)`, l'utilise pour appeler la méthode `getWorld()`, et quand la réponse est prête, modifie l'état du monde pour faire de la réponse le nouvel état.

Ok, votre composant principal dispose maintenant d'un état **world** qui contient une représentation du monde obtenu à partir du service web, et d'un état **services** qui représente le service d'appels au serveur (qui lui-même possède une propriété `server` qui lui permet de connaître l'url du service web et donc par exemple l'URL racine menant aux icônes du monde)r.

Travaillons à présent sur le rendu visuel du monde.

e. Réalisation du layout général

Dotez la page `App.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 :

```
<img src={services.server + world.logo}/>
```

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] } services={ services }/>
```

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
  services: Services
}

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 : dixièmes de secondes*.

Nous aurons donc besoin de fonctions utilitaires qui formatent 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 du grand nombres flottant à quatre chiffres, puis transforme l'exposant en un formatage adéquat pour un rendu HTML faisant apparaître un 10^n . 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.js`.

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 utiliser 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 elle 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 fourni une sous la forme d’un composant `ProgressBar.tsx` que vous trouverez sur moodle.

On ajoute ensuite ce composant dans le html avec :

```
<Box sx={{width: '100%'}}>
  <ProgressBar transitionDuration={"0.1s"} customLabel={" "}
  completed={progress}/>
</Box>
```

La valeur de la propriété `completed` de `ProgressBar` représente le taux de remplissage de la barre. Dans votre composant `Product`, vous devez donc gérer un état `progress` dont la valeur sera tenue à jour pour refléter le pourcentage de production du produit entre 0 et 100. Rappelons qu’avec les *hooks* on définit un état local comme ceci :

```
const [progress, setProgress] = useState(0)
```


Faites maintenant en sorte qu'un *click* 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(() => calcScore(), 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(() => calcScore(), 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 ce la 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, contentez-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 utiliser 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 nouvelle propriété du composant produit que vous appellerez `lastupdate` pour stocker l'instant de démarrage de cette production. En javascript on obtient l'instant courant avec `Date.now()`.
- 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() - this.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 (`this.progressbarvalue = 0`).
 - Sinon (`timeleft` est strictement positif) calculez la nouvelle valeur de la barre de progression qui est $\text{this.progressbarvalue} = ((\text{this.product.vitesse} - \text{this.product.timeleft}) / \text{this.product.vitesse}) * 100$

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}
  services={ services }/>
```

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 9 (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 9 – 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 :

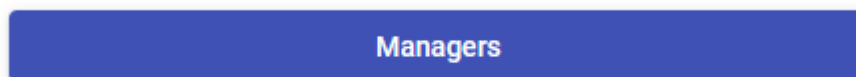
<https://mui.com/components/snackbars/>

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).

Auparavant nous devons néanmoins gérer l'identification du joueur auprès du serveur. En effet le serveur doit s'attendre à être accédé par plusieurs joueurs simultanément, il convient donc que chaque joueur quand il accède au serveur s'identifie. Dans ce projet le serveur fera confiance au joueur en ce qui concerne son identité et ne cherchera pas à la vérifier (il n'y aura donc pas d'authentification du joueur).

a. Permettre au joueur de spécifier son nom

Le premier travail à effectuer se situe coté client. Votre interface doit proposer un état local pour représenter le nom du joueur et un champ texte (en haut à droite sur la Figure 8) 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 coté client. 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);
```

Ce pseudo sera plus tard transmis au serveur à chaque requête vers le service web. C'est pourquoi il faut doter aussi notre composant principal d'un état pour stocker ce nom :

et faire en sorte que le service soit recréé quand ce nom change. On y parvient avec un nouveau *hook* `useEffect` qui s'occupe d'initialiser le nom au démarrage de l'application, et en rendant le *hook* `useEffect` du service, dépendant de ce nouveau *hook*. Au final voici la modification nécessaire :

```
useEffect(() => {
  if (username !== "") {
    let services = new Services(username)
    setServices(services)
    services.getWorld().then(response => {
      let liste = compute_unlocks_list(response.data)
      setWorld(response.data)
      setUnlockList(liste)
    })
  }
}, [username])

useEffect(() => {
  let username = localStorage.getItem("username");
  // si pas de username, on génère un username aléatoire
  if (!username || username === "") {
    username = "Captain" + Math.floor(Math.random() * 10000);
  }
  localStorage.setItem("username", username);
  setUsername(username)
}, [])
```

Remarquez bien la notation `[username]` à la fin du premier `useEffect`, c'est elle qui fait que le *hook* est rappelé quand `username` est modifié.

Le service client est déjà écrit pour transmettre le nom du joueur dans l'entête de la requête http, il ne reste plus qu'à récupérer ce paramètre coté serveur.

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

Retournez dans la partie serveur du projet et dans sa classe `WebService.java` qui implémente l'interface du service web. Actuellement la méthode retournant le monde doit ressembler au code ci-dessous :

```
@GetMapping(value = "world", produces = {"application/xml",
"application/json"})
public ResponseEntity<World> getWorld() {
    World world = services.getWorld();
    return ResponseEntity.ok(world);
}
```

Pour récupérer dans ces méthodes l'entête http, vous devez injecter dans les paramètres un objet qui représentera l'entête passé par le client. Cela se fait en les modifiant comme ci-dessous :

```
@GetMapping(value = "world", produces = {"application/xml",
"application/json"})
public ResponseEntity<World> getWorld(@RequestHeader(value = "X-User",
required = false) String username) {
    World world = services.getWorld(username);
}
```

```
        return ResponseEntity.ok(world);  
    }
```

La variable `username` contient le pseudo passé par le client.

c. Servir au joueur son propre monde

Enfin, il faut associer à chaque joueur sa propre copie du monde. Lors de la première requête faite par un joueur, le serveur doit retourner le monde initial et en faire une copie au nom de ce joueur. Par la suite c'est cette copie qui sera retournée au joueur et qui sera modifiée en fonction de ces actions.

Vous devez donc modifier les méthodes `readWorldFromXml()` et `saveWorldToXml()` de la classe `Services.java` pour qu'elles prennent en paramètre le nom du joueur et qu'elles sauvent et retournent le monde qui lui correspond.

Plus précisément, soit `world.xml` le nom du fichier contenant le monde initial. On pourra décider que le monde d'un joueur nommé *pseudo* se nommera *pseudo-world.xml*. Quand on appellera `readWorldFromXml(pseudo)`, cette dernière devra retourner le fichier *pseudo-world.xml* s'il existe et `world.xml` sinon, ce dernier cas correspondant à l'arrivée d'un nouveau joueur. De même quand on appellera `saveWorldToXml(pseudo)`, le monde devra être sauvegardé dans le fichier *pseudo-world.xml*.

d. Prendre en compte les actions du joueur

Coté 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 mettre en place l'interface du web service qui permettra au client de signaler ces actions au serveur. Comme nous l'avons précisé lors des spécifications page 11, les deux interfaces en mettre en place sont :

`PUT /product` : permet au client de communiquer au serveur une action sur l'entité de type « produit » passé en paramètre. Cette action sera soit un achat dans une certaine quantité de ce type de produit, soit le lancement manuel de la production de ce produit.

`PUT /manager` : permet au client de communiquer au serveur l'achat du manager d'un produit, en passant le manager en question en paramètre sous la forme d'une entité de type « pallier ».

Programmer donc deux méthodes de services web supplémentaire dans la classe `WebService.java`. Les deux consomment des objets respectivement de type `ProductType` et `PallierType`.

Ce qui se passe lorsque ces méthodes sont appelées doit être implémenté dans la classe `Service.java`. Vous doterez donc cette dernière d'une méthode `updateProduct(ProductType product)` et d'une méthode `updateManager(Pallier manager)`.

Commençons par `updateProduct()` qui est la plus élaborée. Le squelette du code de cette classe est donné Figure 10 et est commenté pour insister sur les tâches à coder.

Le squelette du code de `updateManager(Pallier manager)` est plus simple et est donné Figure 11.

```
// prend en paramètre le pseudo du joueur et le produit
// sur lequel une action a eu lieu (lancement manuel de production ou
// achat d'une certaine quantité de produit)

// renvoie false si l'action n'a pas pu être traitée
public Boolean updateProduct(String username, ProductType newproduct) {

    // aller chercher le monde qui correspond au joueur
    World world = getWorld(username);

    // trouver dans ce monde, le produit équivalent à celui passé
    // en paramètre
    ProductType product = findProductById(world, newproduct.getId());
    if (product == null) { return false;}

    // calculer la variation de quantité. Si elle est positive c'est
    // que le joueur a acheté une certaine quantité de ce produit
    // sinon c'est qu'il s'agit d'un lancement de production.
    int qtchange = newproduct.getQuantite() - product.getQuantite();
    if (qtchange > 0) {
        // soustraire de l'argent du joueur le cout de la quantité
        // achetée et mettre à jour la quantité de product

    } else {
        // initialiser product.timeleft à product.vitesse
        // pour lancer la production

    }

    // sauvegarder les changements du monde
    saveWorldToXml(username, world);
    return true;
}
```

Figure 10 – la méthode `updateProduct()` de la classe `Service.java`

```
// prend en paramètre le pseudo du joueur et le manager acheté.
// renvoie false si l'action n'a pas pu être traitée
public Boolean updateManager(String username, PallierType newmanager) {
    // aller chercher le monde qui correspond au joueur
    World world = getWorld(username);

    // trouver dans ce monde, le manager équivalent à celui passé
    // en paramètre
    PallierType manager = findManagerByName(world, newmanager.getName());
    if (manager == null) {
        return false;
    }

    // débloquer ce manager
}
```

```

// trouver le produit correspondant au manager
ProductType product = findProductById(world, manager.getIdcible());
if (product == null) {
    return false;
}
// débloquent le manager de ce produit

// soustraire de l'argent du joueur le cout du manager

// sauvegarder les changements au monde
saveWorldToXml(username, world);
return true;
}

```

Figure 11 - la méthode `updateManager()` de la classe `Service.java`

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 `java System.currentTimeMillis()`).

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.

Une autre façon de voir les choses et de considérer qu'à chaque fois que le serveur lit le monde (pour le transformer en objets java), il doit mettre à jour le score du joueur, et immédiatement sauvegarder cette mise à jour. C'est pourquoi nous utilisons dans les squelettes des méthodes ci-dessus une méthode `getWorld(username)` qui encapsule la méthode `readWorldFromXml()` en la faisant suivre d'une mise à jour du score, puis immédiatement d'un `saveWorldToXml()`. En procédant de la sorte, on est certain que la modification du monde a lieu sur une version dont le score a été mis à jour.

Il vous reste 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.

e. 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.

Le client va faire cela via une requête Ajax de type PUT tantôt sur l'interface `/manager`, tantôt sur l'interface `/product` et en passant tantôt un pallier, tantôt un product.

Voici par exemple la méthode permettant de réaliser un appel d'engagement d'un nouveau manager :

```
putManager(manager : Pallier): AxiosPromise<Response> {
  return axios({
    method: 'put',
    url: this.api + '/manager',
    data: manager,
    headers: Services.setHeaders(this.user)
  }).catch(Services.handleError)
}
```

De la même façon, vous pouvez implémenter les autres appels clients correspondant aux restes des interfaces du service web.

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.

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 12. 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, c'est un petit peu plus subtil pour la raison suivante expliqué ci-dessous.

Mettons qu'un produit soit en production et que sa vitesse de production totale soit de 2mn. Mettons également qu'il reste à ce produit 30s avant que sa production ne soit complète (sa barre de production est donc remplie au trois quarts). Si à ce moment le joueur achète 10 exemplaires supplémentaire de ce produit ce qui lui fait franchir le seuil des 20 exemplaires nécessaire à obtenir un bonus de x2 à la vitesse de production, l'effet du bonus est double :

- Il réduit la vitesse de production du produit de moitié (qui devient donc égale à 1mn). Cette réduction concernera le prochain lancement de production du produit.
- Il réduit également de moitié le temps restant avant la fin de la production courante du produit (qui passe donc de 30s à 15s).
- Autrement dit, quand on obtient un bonus de vitesse de production, il faut faire attention aux produits en cours de production. Non seulement il faut réduire le temps total de production du produit, mais il faut penser à mettre à jour le temps restant de la production en cours. Cette prise en compte doit visuellement conduire à une accélération de la barre de production.

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 12 – Liste des *unlocks*

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 :

✓ Paper Bags Speed x2

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 débloquent 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.

Modifiez donc la méthode `updateProduct()` de la Figure 10 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.

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 13. 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.



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.











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

Figure 13 – Liste des *upgrades*.

a. 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 méthode :

PUT /upgrade : permet au client de communiquer au serveur l'achat d'un *Cash Upgrade* en passant cet *upgrade* en paramètre sous la forme d'une entité de type « *pallier* ».

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*.

b. 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.

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 14 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 méthode suivante :

DELETE /world : permet au client de demander le *reset* du monde.

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.

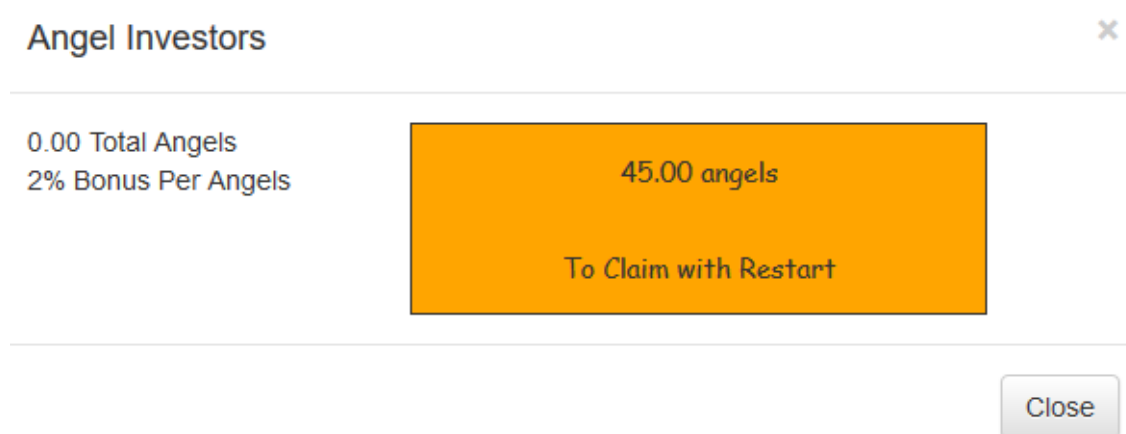


Figure 14 – Interface de gestion des anges

b. Gestion des anges coté serveur

Implémentez l'interface de web service 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 recharger le monde original (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 15.

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 !



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

Close

Figure 15 - 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 « pallier ».

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

Implémentez l'interface web service qui permet au serveur de réceptionner un *angel upgrade*.

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 le fichier `Services.ts` pour que votre client s'y connecte. Si les deux parties sont compatibles, 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 recale 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 REST en manipulant des représentations au format XML et JSON. Vous avez manipulé la plateforme Spring boot coté serveur et ReactJS coté client.

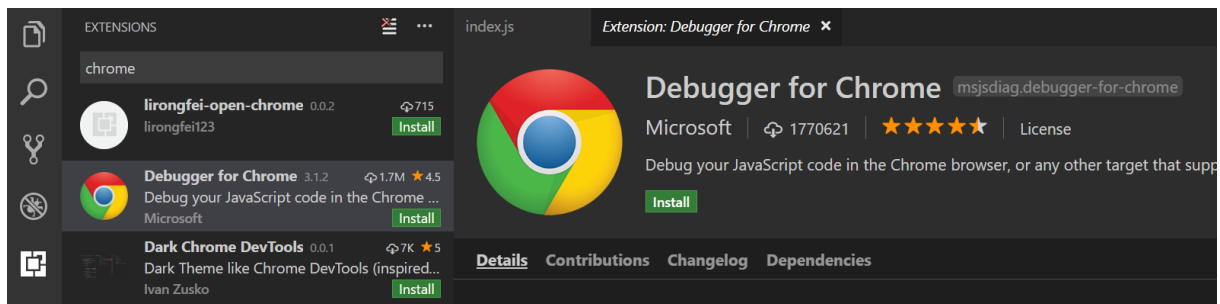
Vous avez de plus exercé vos compétences en calculs mathématiques et avez travaillé sur des problématiques de synchronisation temporelle entre un serveur et ses clients.

Annexes

Débugger avec Visual Studio Code et Chrome

Il est parfois utile d'utiliser un débogueur pour analyser le déroulement de l'exécution du code ou effectuer un suivi de l'évolution de certaines variables et propriétés. Voici comment en configurer un pour une utilisation avec Visual Studio Code et le navigateur Chrome :

Dans Visual Studio Code, installez l'extension « Chrome Debugger » en passant par le menu « afficher/extension » puis en tapant « chrome » dans le champ de recherche.



Relancez Visual Studio Code et cliquez dans la vue de débogage (icône en forme d'insecte barré sur la gauche). Ajoutez une configuration de débogage en cliquant sur l'icône en forme d'engrenage. Cela crée un fichier `launch.json` auquel vous allez ajouter du contenu en choisissant dans le menu déroulant à gauche de l'engrenage « *ajouter une configuration* » puis « *chrome : launch* ». Au final le fichier `launch.json` doit contenir (pensez bien à corriger le numéro de port du lien http pour y mettre la valeur 3000 qui est celle sur laquelle écoute par défaut le serveur) :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Chrome",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

Voilà la configuration est terminée.

Vous pouvez à présent utiliser le débogueur en mettant en place des points d'arrêts dans le programme (en cliquant dans la gouttière à gauche des numéros de lignes). Une fois un point d'arrêt atteint, vous pouvez regarder la valeur des variables et propriétés en les survolant avec la souris. Vous pouvez ensuite exécuter le programme instruction par instruction (pas à pas principal) ou le faire continuer jusqu'au prochain point d'arrêt (continuer).

Vous pouvez également insérer des variables dont vous voulez suivre en permanence la valeur dans la catégorie « espions », insérez des points d'arrêt conditionnels, etc.