

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.....	15
d. Sérialisation et dé-sérialisation java du monde	17
e. Création du service web servant le monde.....	17
f. Autoriser le serveur à répondre au requêtes cross-domain	18
3. Début du travail sur le client web	19
a. Création du projet Angular.....	19
b. Description du travail attendu	19
c. Création des premiers composants et du service de communication avec le service web ..	21
d. Réalisation du layout général	25
e. Eléments utiles de CSS.....	26
f. Les pipes d'Angular.....	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.....	32
a. Interface pour lister les managers.....	32
b. Engagement d'un manager	36
c. Afficher un message éphémère pour l'utilisateur.....	36
d. Badger les boutons pour informer le joueur	37
6. Retour coté serveur.....	38
a. Permettre au joueur de spécifier son nom	38
a. Transmettre le nom du joueur au serveur lors des requêtes	39
b. Modifier le service web pour qu'il récupère le nom du joueur	39
c. Servir au joueur son propre monde	40

d.	Prendre en compte les actions du joueur	40
e.	Appel des interfaces par le client	43
7.	Les <i>unlocks</i>	44
a.	Affichage des <i>unlocks</i>	44
b.	Prise en compte des <i>unlocks</i> par le client	44
c.	Prise en compte des <i>unlocks</i> par le serveur	47
8.	Les <i>Cash upgrades</i>	47
a.	Affichage des <i>upgrades</i>	47
b.	Prise en compte des <i>upgrades</i> par le client.	47
a.	Transmission des upgrades du client au serveur.	48
b.	Prise en compte des <i>upgrades</i> par le serveur.	48
9.	Gestion des anges.....	49
a.	Gestion des anges coté client.....	49
b.	Gestion des anges coté serveur	50
c.	Prise en compte des anges dans les gains.....	50
10.	Gestion des <i>Angel Upgrades</i>	50
a.	Prise en compte des <i>angel upgrades</i> par le client.	51
b.	Prise en compte des <i>angel upgrades</i> par le serveur.	52
11.	Finalisation et branchement sur un autre monde	52
Annexes		52
Débuguer avec Visual Studio Code et Chrome.....		52

1. Présentation du projet

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



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

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

Principes du jeu « Adventure Capitalist »

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

Gameplay

Investir dans des produits et lancer leur production

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

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

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

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

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



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

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

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

Les managers

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



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

Les seuils (unlocks)

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

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

Les Cash Upgrades

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

Les anges

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

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

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



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



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

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

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

Les Angel Upgrades

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

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

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



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

2. Conception du serveur de mondes

a. Spécifications de l'API du serveur

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

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

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

En termes de **protocoles**, nous utiliserons une communication client-serveur basée sur un service web au format 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

- Récupérez le schéma de la Figure 7 et sauvez-le dans un fichier.
 - 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 JAX-RS (pour le web service) et DevTools. Vous pouvez appeler le projet `com.isis.adventureIServer`.
- Ouvrez ce projet sous NetBeans et compilez-le pour ramener les dépendances nécessaires.

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. On peut y accéder par NetBeans, via le menu contextuel « New » de l'intitulé du projet, puis « XML/JAXB Binding ». L'interface qui s'ouvre

vous permet de désigner le fichier contenant le schéma que vous avez créé précédemment. Compilez le projet et si tout se passe bien les classes produites sont ajoutées à votre projet dans le package adéquat, ainsi qu'une copie du schéma xml. Il est important de noter que c'est cette copie (en principe située dans la partie `Other Sources`) qui devra désormais être modifiée en cas de changement dans le schéma. En effet, l'IDE régènerera les classes Java dès qu'il détectera une modification de ce fichier.

L'arborescence projet que vous devez obtenir suite à cette opération est donnée Figure 8.

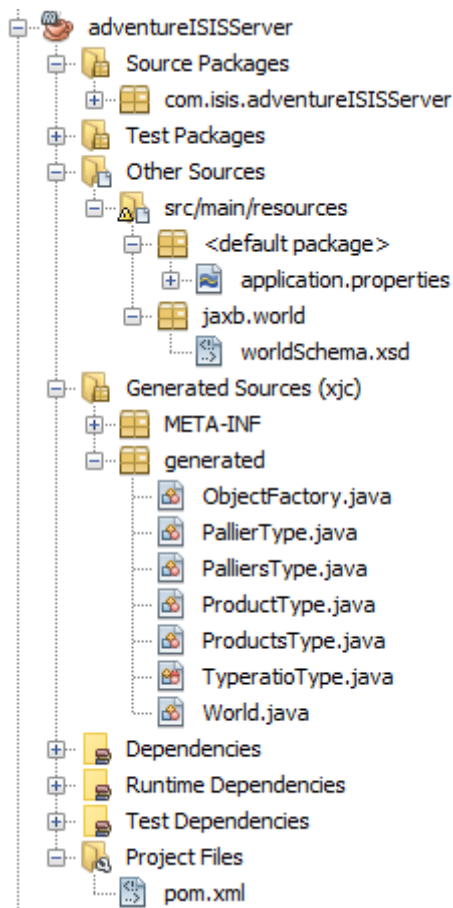


Figure 8 – Arborescence du projet après mapping schéma - java

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 dé-sérialisant 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 cela, passez par le menu `New File/XML/XML`

Document de Netbeans et spécifiez que le fichier est basé sur le schéma du monde. L'IDE va de la sorte pré-remplir le fichier XML avec les balises adéquates.

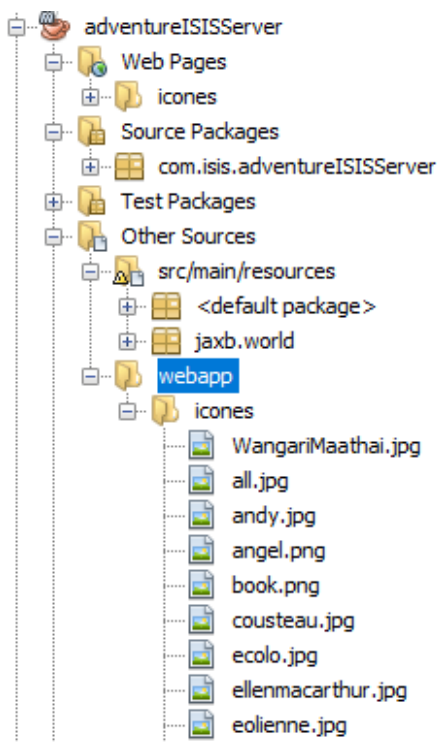
Le travail consiste ensuite à remplir le fichier XML de façon à spécifier :

- 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.

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). Le fait de les mettre à cet endroit, permettra aux clients d'y accéder au moyen de l'URL `http://adresse/icones`.

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 deux méthodes :

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

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`. La seconde doit réaliser l'opération symétrique. Vous utiliserez le framework JAXB pour réaliser ces opérations.

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 encapsulé dans un fichier `.war` qui est ensuite déployé et dézippé sur le serveur J2EE de diffusion, c'est-à-dire dans le dossier `target` de votre projet. 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).

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 :

```
Path("generic")  
public class Webservice {  
  
    Services services;  
  
    public Webservice() {  
        services = new Services();  
    }  
  
    @GET  
    @Path("world")  
    @Produces(MediaType.APPLICATION_XML)  
    public Response getWorld() {  
        return Response.ok(services.getWorld()).build();  
    }  
}
```

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`.

Pour que le web service se lance, il faut ajouter à votre projet une nouvelle classe java appelée JerseyConfig.java et contenant le code suivant :

```
@Component
@ApplicationPath("/adventureisis")
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {

        register(WebService.class);

    }
}
```

Lancez le projet et vérifiez que le service web fonctionne en pointant votre navigateur vers son adresse.

En modifiant l'annotation @Produces, on peut définir que la méthode getWorld retourne aussi le monde au format JSON. Pour cela il suffit de l'écrire :

```
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
```

Effectuez la modification, et pour tester ce point d'accès, 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 le console réseau , et n spécifier le header http *accept : application/json*. La présence de ce *header* provoquera l'appel automatique du point d'accès générant du JSON.

f. Autoriser le serveur à répondre aux requêtes cross-domain

Pour qu'un client autonome (comme le sera l'IHM du jeu) puisse se connecter au service web, celui-ci doit accepter les requêtes cross-domain, c'est-à-dire les requêtes provenant d'un autre domaine que le sien. Cela nécessite de configurer correctement le service web. Pour cela ajoutez à votre projet une classe java qui fera office de filtre pour toutes les requêtes et qui positionnera les entêtes requis pour que le cross-domain soit autorisé. Voici le code de cette classe :

```
@Provider
public class CORSResponseFilter implements ContainerResponseFilter {

    public void filter(ContainerRequestContext requestContext,
        ContainerResponseContext responseContext)
        throws IOException {

        MultivaluedMap<String, Object> headers =
            responseContext.getHeaders();

        headers.add("Access-Control-Allow-Origin", "*");
        headers.add("Access-Control-Allow-Methods", "GET, POST, DELETE,
            PUT, OPTIONS");
        headers.add("Access-Control-Allow-Headers", "X-Requested-With,
            Content-Type, X-Codingpedia, authorization,X-User");
    }
}
```

Déclarez ce filtre en ajoutant dans la classe JerseyConfig.java, la ligne :

```
register(CORSResponseFilter.class);
```

Dès lors des clients externes pourront se connecter chez vous.

3. Début du travail sur le client web

a. Création du projet Angular

Le client web va être construit en utilisant le framework Angular (<https://angular.io>). Pour mettre en place le projet, nous allons utiliser *angular-cli* (<https://github.com/angular/angular-cli>) qui est une suite d'outil permettant de gérer l'arborescence du projet et ses phases de compilation et d'exécution. Même si nous pourrions utiliser Netbeans comme IDE, on vous recommande ici d'utiliser Visual Studio Code (<https://code.visualstudio.com>), plus adapté au langage TypeScript qui est au cœur d'Angular.

L'installation d'Angular passe par l'installation de node.js (<https://nodejs.org>). On installe ensuite angular-cli en tapant la commande :

```
npm install -g @angular/cli
```

Créez ensuite un nouveau projet en utilisant la commande (et en choisissant un nom pour votre projet) :

```
ng new PROJECT-NAME
```

Cette commande a pour effet de créer un projet Angular minimal qui peut être lancé par les commandes :

```
cd PROJECT-NAME  
ng serve
```

La commande `ng serve` lance un serveur web (en l'occurrence node.js), y déploie le projet en général sur le port 4200. Il ne reste plus qu'à faire pointer un navigateur web sur l'adresse <http://localhost:4200> pour visualiser la page d'accueil. Vous devez laisser tourner en permanence cette commande lors de votre développement. Vous remarquerez que dès que vous modifierez un des composants du projet, celui-ci sera automatiquement redéployé et rechargé dans le navigateur.

Vous pouvez à présent lancer Visual Studio Code et ouvrir le dossier correspondant à votre projet pour commencer à travailler.

b. 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 9 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.

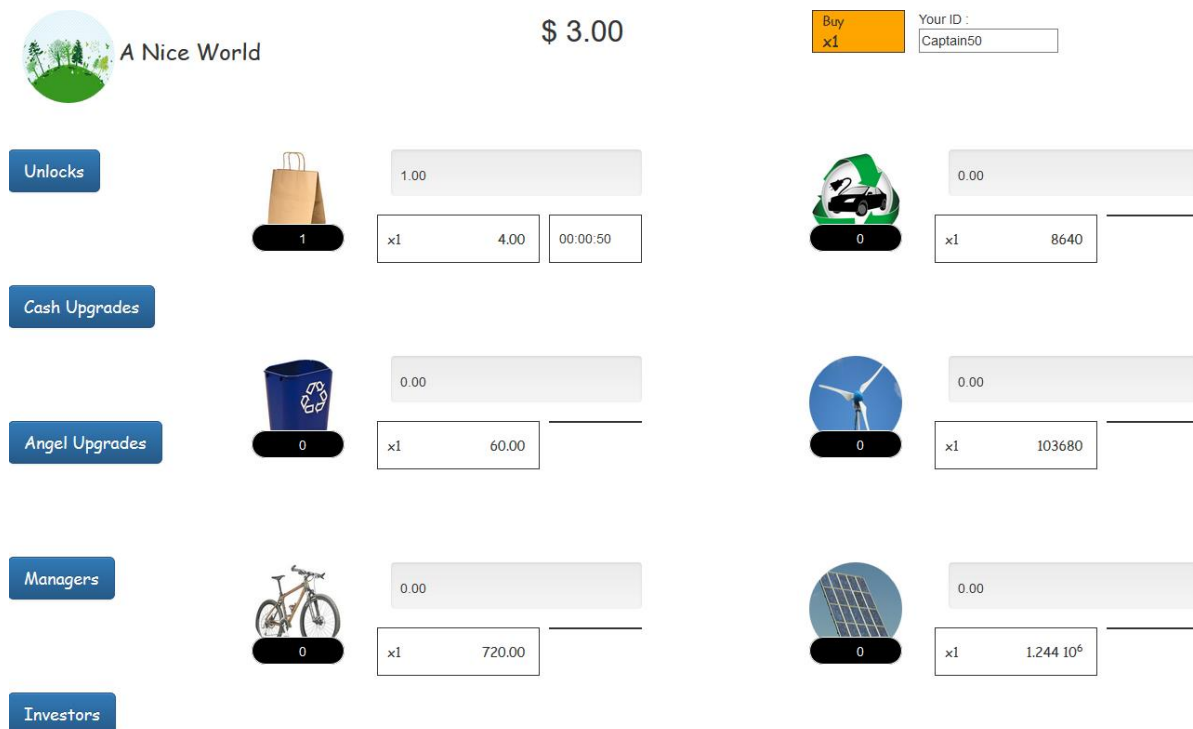


Figure 9 – Illustration de l'interface du jeu

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 réaliser cette mise en page, on peut utiliser le framework *bootstrap* (<http://getbootstrap.com/>) qui permet facilement de spécifier des mises en page adaptatives en colonnes. Son utilisation demande l'ajout de deux liens CSS dans le fichier `index.html` du projet. Voici ces deux lignes :

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css" >
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap-
theme.min.css" >
```

Le concept des mises en page réalisées avec bootstrap est de diviser la page en 12 colonnes et d'autoriser les éléments d'une ligne à occuper un certain nombre de ses douze colonnes. Consultez la documentation de ce système de grille pour comprendre comme il fonctionne (<http://getbootstrap.com/css/#grid>).

Pour donner un exemple, si l'on veut obtenir le *layout* ci-dessous :

Entete1	Entete2	Entete3
Partie Gauche	Partie droite	

On utilisera le code suivant :

```

</head>
<style>
  * {
    border: 1px solid red;
  }
</style>
</head>

<body>
  <div class="container-fluid">
    <div class="row">
      <div class="col-md-4">Entete1</div>
      <div class="col-md-4">Entete2</div>
      <div class="col-md-4">Entete3</div>
    </div>
    <div class="row">
      <div class="col-md-2">Partie Gauche</div>
      <div class="col-md-10">Partie droite</div>
    </div>
  </div>
</body>

```

Les balises `<row>` peuvent contenir d'autres balises `<row>` qui reprennent alors la numérotation des colonnes à zéro. Ce système permet à chaque cellule de pouvoir présenter une mise en page autonome.

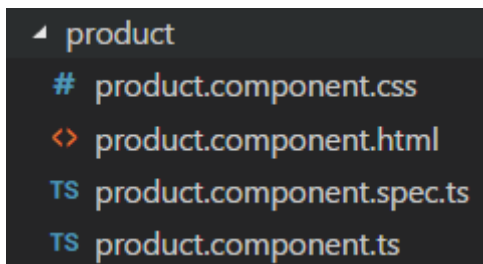
c. Création des premiers composants et du service de communication avec le service web

En ce qui concerne l'organisation des sources dans le projet Angular, nous utiliserons deux composants. Le premier, `app.component` (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.component`) le soin de gérer l'affichage des éléments d'un produit.

Pour créer un nouveau composant automatiquement, on peut utiliser la commande :


```
ng g component product
```

qui ajoute dans l'arborescence les fichiers nécessaires au composant product. Vous devez les voir apparaître dans Visual Studio Code.



Nous aurons également besoin d'un service qui se chargera des appels au service web. Là encore vous pouvez ajouter automatiquement ce service au projet en utilisant la commande :

```
ng g service restservice -m app.module
```

L'option `-m app.module` ajoute automatiquement le service au fichier `app.module.ts` ce qui nous évite de le faire manuellement.

Pour commencer, nous allons doter ce service du code nécessaire à faire un appel au service GET `/world` du service web. Ce code doit réaliser un appel asynchrone, et doit donc retourner une *promesse* de monde. Quel sera le type de ce monde ? Techniquement le service web retournera un objet au format JSON qui contiendra tous les éléments de votre fichier `world.xml`. Si nous étions en javascript standard, nous traiterions ce JSON de façon classique, en parcourant ses listes de propriétés-valeurs. Mais nous sommes ici non pas en Javascript mais en TypeScript qui est un langage qui demande un typage fort des valeurs. Nous devons donc définir une classe TypeScript qui corresponde au schéma XML du monde. Idéalement ce type devrait pouvoir se définir automatiquement, exactement comme nous le permet JAXB quand il définit nos classes Java. 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 Angular, au même niveau que le `app.module.ts` par exemple. 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;
  score: number;
  totalangels: number;
  activeangels: number;
  angelbonus: number;
  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;
    name : string;
    logo : string;
    cout : number;
    croissance: number;
    revenu: number;
    vitesse: number;
    quantite: number;
    timeleft: number;
    managerUnlocked: boolean;
    palliers : { "pallier" : Pallier[] };
}

export class Pallier {
    name: string;
    logo: string;
    seuil: number;
    idcible: number;
    ratio: number;
    typeratio: string;
    unlocked: boolean;
}

```

Une fois ce fichier importé dans le projet, revenez au fichier `restservice.service.ts` et importez dans le service les classes nécessaires, ainsi que le service http que nous allons utiliser :

```

import { HttpClient } from '@angular/common/http'

import { World, Pallier, Product } from './world';

```

Et on injecte le service http via le constructeur :

```
constructor(private http: Http) { }
```

Dotez ensuite cette classe de deux propriétés :

```
server = "http://localhost:8080/adventureISIS/"  
user = "";
```

La première propriété servira à stocker l'url menant au service web, la deuxième à mémoriser le nom du joueur (nous verrons plus tard comment on définit ce nom).

Ajoutez les setters et getters nécessaire pour accéder à la propriété user.

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 éventuellement lors de l'appel au service :

```
private handleError(error: any): Promise<any> {  
    console.error('An error occurred', error);  
    return Promise.reject(error.message || error);  
}  
  
getWorld(): Promise<World> {  
    return this.http.get(this.server + "webresources/generic/world")  
        .toPromise().catch(this.handleError);  
};
```

Il ne reste plus qu'à faire appel à cette méthode lors de l'initialisation du client, c'est-à-dire dans le constructeur du composant `app.component.ts` (vous n'oublierez pas d'importer les trois classes métiers `World`, `Pallier` et `Product`, ainsi que le service¹). Voici un extrait de code qui réalise cela :

```
import { RestserviceService } from './restservice.service';  
import { World, Product, Pallier } from './world';  
  
world: World = new World();  
server: string;  
  
constructor(private service: RestserviceService) {  
    this.server = service.getServer();  
    service.getWorld().then(  

```

¹ A partir de ce point, l'énoncé ne précisera plus systématiquement les imports ou les injections à réaliser. C'est à vous de les insérer en fonction des besoins.

```
world => {
  this.world = world;
});
}
```

Ok, votre composant principale dispose maintenant d'une propriété `world` qui contient une représentation du monde obtenu à partir du service web. Vous remarquerez aussi que nous l'avons doté d'une propriété `server` qui lui permet de connaître l'URL du service web. En effet les icônes du monde sont stockées sur ce serveur, et il faudra donc connaître cet URL pour les afficher.

Travaillons à présent sur le rendu visuel du monde.

d. Réalisation du layout général

Dotez les pages `app.component.html` et `app.component.css` 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 [attr.src]="server+world.logo" /> </span>
```

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 id="worldname"> {{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 au composant `product.component` 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) :

```
<app-product [prod]="world.products.product[0]"
class="product col-md-6"></app-product>
```

Dans le fichier `product.component.ts`, vous récupérez le produit passé par le composant parent en définissant une propriété annotée avec `@Input`. Voici à titre exemple comment on peut le faire, avec en plus ici l'usage d'un setter qui nous permettra d'intercepter les changements apportés à la liste des produits (vous verrez plus tard pourquoi c'est utile).

```
product: Product;
@Input()
```

```
set prod(value: Product) {  
    this.product = value;  
}  
}
```

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

Vous pouvez donc spécifier le code HTML du fichier `product.component.html` qui réalise l'affichage d'un produit.

A vous de jouer.

e. Eléments 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>

```

f. Les pipes d'Angular

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

Les *pipes* d'Angular fournissent une solution élégante à ce problème en permettant de définir un transformateur qui prend en paramètre une valeur (par exemple le temps en millisecondes) et qui produit une chaîne de caractères. A titre d'exemple définissons un pipe qui prend en paramètre un nombre flottant et qui produit son formatage en puissance de dix, avec quatre chiffres significatifs.

Pour créer le *pipe*, on peut utiliser la commande :

```
ng generate pipe bigvalue -m app.module
```

La classe produite (`bigvalue.pipe.ts`) possède une méthode `transform` qu'il faut coder pour réaliser la transformation. Elle prend en paramètre la valeur à transformer et doit retourner la chaîne de caractère produite. Une implémentation un peu naïve de cette méthode pourrait être :

```

transform(valeur: number, args?: any): 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.

Pour ensuite utiliser le *pipe* dans le rendu HTML, on peut par exemple écrire :

```
<span [innerHTML]="world.money | bigvalue"> </span>
```

Notez l'utilisation de l'attribut `[innerHTML]` qui permet au rendu d'interpréter les balises HTML produites par le *pipe*. Si vous utilisez la notation habituelle `{{ world.money | bigvalue }}`, le nom des balises apparaîtra dans le rendu, ce qui n'est pas l'effet recherché.

4. Actions du joueur

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

a. Démarrage de la production d'un produit

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

La méthode `startFabrication()` doit lancer une barre de progression dans l'espace prévu à cet effet. Pour le rendu de cette barre nous allons utiliser la bibliothèque *progressbar* disponible à l'adresse : <http://kimmobrunfeldt.github.io/progressbar.js/>

Cette bibliothèque n'est pas une bibliothèque native Angular, il faut donc l'ajouter de la façon suivante à votre projet :

1. Utiliser la commande `npm install progressbar.js --save` pour ajouter la bibliothèque à la liste de vos modules NodeJs.
2. Ajouter dans le tableau `scripts` du fichier `angular-cli.json` (ou `angular.json` si votre version d'Angular est postérieure à la 6) la ligne :

```
"scripts": ["node_modules/progressbar.js/dist/progressbar.js"],
```

3. Ajouter au début du fichier `product.component.ts` les lignes suivantes :

```
declare var require;  
const ProgressBar = require("progressbar.js");
```

4. Ajouter une propriété à la classe pour y stocker la barre de progression :

```
progressbar: any;
```

Ensuite, dans le fichier `product.component.html`, vous devez identifier la partie qui contiendra la barre de progression. En principe vous devez avoir une balise `<div>` qui joue ce rôle, identifiez là avec l'identifiant `#bar` comme illustré ci-dessous :

```
<div class="progress" #bar> </div>
```

Enfin pour faire référence à cette partie à partir du fichier `product.component.ts`, ajoutez au début de celui-ci une annotation `@ViewChild` comme ci-dessous :

```
@ViewChild('bar') progressBarItem;
```

Cette annotation permet de déclarer une propriété `progressBarItem` qui fait référence à la partie identifiée `#bar` de votre html.

Vous pouvez consulter la documentation complète de la bibliothèque progressbar.js sur son site officiel. Pour la résumer, et en admettant que la variable `this.progressBarItem` désigne la partie où la barre doit apparaître, on peut créer une barre de progression avec la syntaxe suivante :

```
this.progressBar = new  
ProgressBar.Line(this.progressBarItem.nativeElement, { strokeWidth: 50, color:  
'#00ff00' });
```

Cette création doit avoir lieu lors de l'initialisation du composant produit. On peut donc le faire par exemple dans la méthode `ngOnInit()` de `product.component.ts`.

Pour démarrer la barre, on doit indiquer le temps que doit prendre la production (exprimé en millisecondes). Cela se fait avec la ligne suivante :

```
this.progressBar.animate(1, { duration: this.product.vitesse });
```

Cette ligne démarre donc le remplissage de la barre qui mettra autant de temps que la vitesse du produit à aller au bout.

On peut également fixer manuellement le taux de progression d'une barre avec la commande :

```
this.progressBar.set(progress);
```

La variable `progress` devant avoir une valeur comprise entre 0 et 1. Attention, cette ligne stoppe l'animation qu'il faut donc relancer ensuite.

Vous avez tous les éléments techniques pour faire en sorte qu'un *click* sur l'icône du produit lance visuellement sa production. A vous de jouer et pensez à ne pas lancer la production si la quantité de ce produit n'est pas d'au moins 1.



Pour l'instant la barre de production reste pleine une fois la production terminée, nous allons arranger cela dans la prochaine section.

b. La boucle principale de calcul du score

Nous savons afficher la barre de production d'un produit mais pour l'instant notre score (l'argent gagné) n'évolue pas (du reste la barre une fois remplie reste remplie au lieu de se remettre à zéro).

Nous pourrions implémenter un callback qui serait appelé lorsque la barre de production a rempli sa jauge. Nous allons néanmoins préférer une autre approche.

L'idée est de créer une fonction de mise à jour du score qui décrémente le temps restant pour terminer la production d'un produit, et qui, si ce temps est devenu nul (ou négatif), augmente l'argent du joueur de ce que rapporte la production du produit. Cette fonction sera appelée à intervalle régulier, mettons toutes les dixièmes de secondes si on veut une mise à jour du score aussi rapide que dans le jeu original.

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 (donc toujours dans la méthode `ngOnInit()`), l'appeler tous les dixièmes de seconde en écrivant :

```
setInterval(() => { this.calcScore(); }, 100);
```

Il ne reste plus qu'à implémenter une méthode `calcScore()` qui pour chaque 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.progressbar.set(0)`).

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 car celui-ci est géré par le composant parent (`app.component`). 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. Angular propose le mécanisme des événements pour faire cela. Lors de la production d'un produit, le composant produit génère un événement de production qui sera transmis au parent. Le parent récupérera cet événement et fera la mise à jour du score. Pour réaliser cela :

Dans la classe `product.component.ts`, déclarez un événement de sortie, que nous appellerons par exemple `notifyProduction` qui prend en paramètre en élément de type `Product` :

```
@Output() notifyProduction: EventEmitter<Product> = new  
EventEmitter<Product>();
```

Dans la méthode `calcScore()`, quand la production d'un produit est terminée, générez cet événement en écrivant :

```
// on prévient le composant parent que ce produit a généré son revenu.  
this.notifyProduction.emit(this.product);
```

Du côté du parent à présent, modifiez le fichier `app.component.html`, pour déclarer que la balise `<app-product>` peut générer un évènement `notifyProduction` et définir la méthode à appeler. On y ajoutera donc l'attribut :

```
(notifyProduction)="onProductionDone($event)"
```

A vous à présent d'implémenter la méthode `onProductionDone(Product p)` de la classe `app.component.ts` pour qu'elle augmente l'argent (et le score) du joueur en fonction de ce que rapporte la production du produit.

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 `app.component` 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 `app.component.ts`, on utilisera une propriété (appelons là `qtmulti`) pour stocker l'état actuel du commutateur d'achat général.
- On transmettra la valeur de cette propriété aux composants produit, par exemple en ajoutant à la balise `<app-product>` l'attribut :

```
[qtmulti]="qtmulti"
```

Et en modifiant le composant `product.component.ts` pour qu'il puisse réceptionner cette valeur via un setter (pour pouvoir agir quand la valeur change) :

```
_qtmulti: string;
@Input()
set qtmulti(value: string) {
  this._qtmulti = value;
  if (this._qtmulti && this.product) this.calcMaxCanBuy();
}
```

- 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 `product.component.ts`, 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. Implémentez donc un événement `onBuy<Number>` qui prendra en paramètre le coût total d'achat et qui sera utilisé pour que le parent soit averti de la somme devant être soustraite de l'argent possédé par le joueur (et de son score).

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.

Une façon simple de gérer des fenêtres superposées est d'utiliser le widget modal de bootstrap documenté à l'adresse <https://getbootstrap.com/javascript/#modals>

Pour adapter ce widget au monde Angular, il faut définir dans votre projet un nouveau composant qui gèrera l'ouverture et la fermeture de la modale. Utilisez la commande :

```
ng g component modal
```

pour créer ce nouveau composant et peuplez le fichier `modal.component.ts` du code suivant (disponible également sur moodle) :

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-modal',
  template: `
    <div (click)="onContainerClicked($event)" class="modal fade" tabindex="-1"
[ngClass]="{'in': visibleAnimate}"
    [ngStyle]="{'display': visible ? 'block' : 'none', 'opacity':
visibleAnimate ? 1 : 0}">
      <div class="modal-dialog">
        <div class="modal-content">
          <div class="modal-header">
            <ng-content select=".app-modal-header"></ng-content>
          </div>
          <div class="modal-body">
            <ng-content select=".app-modal-body"></ng-content>
          </div>
          <div class="modal-footer">
            <ng-content select=".app-modal-footer"></ng-content>
          </div>
        </div>
      </div>
    </div>
  `
})
export class ModalComponent {

  public visible = false;
  public visibleAnimate = false;

  public show(): void {
    this.visible = true;
    setTimeout(() => this.visibleAnimate = true, 100);
  }

  public hide(): void {
    this.visibleAnimate = false;
    setTimeout(() => this.visible = false, 300);
  }
}
```

```

}

public onContainerClicked(event: MouseEvent): void {
    if ((<HTMLElement>event.target).classList.contains('modal')) {
        this.hide();
    }
}
}
}

```

Une fois cela fait, on peut définir le contenu d'une fenêtre de type modal (superposée) de la façon suivante :

```

<app-modal #managersModal>
  <div class="app-modal-header">
    <h4 class="modal-title">Managers make you feel better !</h4>
  </div>
  <div class="app-modal-body">
    <div *ngFor="let manager of world.managers.pallier">
      // a completer pour réaliser le rendu d'une ligne de manager
    </div>
  </div>
  <div class="app-modal-footer">
    <button type="button" class="btn btn-default"
(click)="managersModal.hide()">Close</button>
  </div>
</app-modal>

```

Et faire en sorte que la fenêtre s'ouvre lors d'un clic sur un bouton défini de la façon suivante :

```

<div class="buttonleft btn btn-primary btn-lg"
  data-toggle="modal" (click)="managersModal.show()" >
  Managers
</div>

```

Utilisez cette façon de faire pour compléter le fichier app.component.html pour qu'il remplisse la fenêtre des managers en fonction de ceux définis dans votre monde selon le modèle proposé Figure 10 (mais vous pouvez choisir un autre design si vous préférez).

Managers make you feel better !



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 10 – 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 lancé 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 la bibliothèque *toastr.js* dont le github se trouve à l'adresse <https://github.com/Stabzs/Angular2-Toaster>.

Ajoutez la bibliothèque à votre code en tapant la ligne suivante :

```
npm install angular2-toaster --save
```

et en ajoutant dans la partie *styles* du fichier *.angular-cli.json* la ligne suivante :

```
"styles": [  
  "styles.css", "node_modules/angular2-toaster/toaster.css"  
],
```

Il faut également importer la bibliothèque dans le *app.module.ts* en ajoutant :

```
import { ToastrModule } from 'angular2-toaster';
```

et ne pas l'oublier non plus dans le tableau des imports globaux :

```
imports: [  
  BrowserModule, HttpClientModule, BrowserAnimationsModule, ToastrModule  
],
```

Il reste à définir un emplacement pour le toaster dans le fichier `app.component.html`. Vous pouvez par exemple le mettre à la fin de la page en ajoutant juste avant la fermeture du `<div class='container-fluid'>`:

```
<toaster-container></toaster-container>
```

Ceci fait il est très simple d'envoyer des messages éphémères à l'utilisateur. Voici comment par exemple on génère un message d'information contenant le nom du manager engagé (la propriété `toasterService` doit être injecté via le constructeur de `app.component.ts`):

```
this.toasterService.pop('success', 'Manager hired ! ', manager.name);
```

Et pour un message d'erreur :

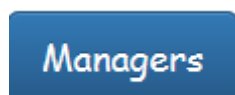
```
this.toasterService.pop('error', 'Reset failed ! ', reason.status)
```

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

d. Badger les boutons pour informer le joueur

Bootstrap 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 texte *New* :



Pour « badger » un bouton, on ajoute à son contenu une partie `` de classe `badge` de la façon suivante :

```
<div class="buttonleft btn btn-primary btn-lg" data-toggle="modal" ...>
  <span class="badge"></span> Managers
</div>
```

Pour faire apparaître le badge, il ne reste plus qu'à remplir la partie `span` comme par exemple :

```
<span class="badge">New</span> Managers
```

Modifiez donc la fonction le composant `app.component.ts` pour qu'à chaque évolution de l'argent du joueur, la possibilité d'acheter un nouveau manager soit vérifiée. Si c'est le cas, on doit positionner (ou enlever, quand l'argent a diminué) le badge « new » 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 champ texte (en haut à droite sur la Figure 9) permettant à l'utilisateur de spécifier un pseudo :

```
<input type="text" [(ngModel)]="username" change)="onUsernameChanged()"/>
```

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 :

```
this.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", this.username);
```

Ce pseudo sera plus tard transmis au serveur à chaque requête vers le service web. C'est pourquoi il faut doter aussi le service `restservice.service.ts` d'une propriété `user`, avec les getter et setter associés, et mettre à jour cette propriété, dès que le `username` de `app.component.ts` change.

a. Transmettre le nom du joueur au serveur lors des requêtes

Nous pourrions faire cela en ajoutant à toutes les requêtes un paramètre supplémentaire contenant le pseudo. Nous proposons néanmoins une façon de faire plus élégante qui consiste à ne pas toucher aux requêtes et à transmettre le nom de l'utilisateur dans l'entête http avec un header personnalisé que nous nommerons X-user.

Pour construire ce header, ajoutez la méthode suivante dans le fichier `restservice.service.ts` :

```
private setHeaders(user : string) : Headers {  
    return { "X-User" : user };  
}
```

et modifiez la façon dont vous utilisez la méthode `get` de l'objet `http`, pour lui ajouter un dernier paramètre correspondant à ce header. Voici donc la nouvelle version de la méthode `getWorld` qui passe le nom du joueur au serveur :

```
getWorld(): Promise<World> {  
    return this.http.get(this.server + "webresources/generic/world", {  
headers: this.setHeaders(this.user)})  
        .toPromise().catch(this.handleError);  
};
```

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 :

```
@GET  
@Path("world")  
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})  
public Response getWorld() {  
    return Response.ok(services.getWorld()).build();  
}
```

Pour récupérer dans ces méthodes l'entête http, vous devez y injecter un objet qui représentera la requête du client. Cela se fait en les modifiant comme ci-dessous :

```
@GET  
@Path("world")  
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})  
public Response getXml(@Context HttpServletRequest request) {  
    String username = request.getHeader("X-user");  
    return ...;  
}
```

Au final 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

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

- Lancement de la production d'un produit
- Achat d'une certaine quantité de produit
- Achat d'un manager

Nous allons 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 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 au format json et ont donc un paramètre au format `String` qui sera de-sérialisé pour la première en un objet de type `ProductType`, pour la deuxième en un objet de type `PallierType`.

Pour passer d'une représentation au format json contenue dans un objet de type `String`, à un objet Java, on utilise la bibliothèque `gson` et sa méthode `fromJson(data, classe)` ;

Par exemple pour obtenir un objet de type `ProductType` à partir d'une chaîne de caractère appelée `data` et contenant le json d'un produit (tel que passé en paramètre par le client) on écrira :

```
ProductType product = new Gson().fromJson(data, ProductType.class);
```

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 11 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 12.

```
// 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 11 – 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;
    }

    // débloquent 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 12 - 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 toute 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): Promise<Response> {  
  return this.http.put(this.server + "webresources/generic/manager", manager,  
  { headers: this.setHeaders(this.user)} )  
    .toPromise();  
}
```

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 ou 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. On parle ici de rechargement de la page, mais en fait il faut repositionner les barres de progression dès que la liste des produits change. C'est pour cela que dans `product.component.ts`, nous avons choisi d'utiliser un setter lors du `@Input` de la propriété `prod`.

En fonction de ce que nous venons d'expliquer, voici comment le setter de l'`@Input` de `prod`, doit être modifié pour que la barre de progression commence là où en est la production du produit au démarrage du client.

```
product: Product;
@Input()
set prod(value: Product) {
  this.product = value;
  if (this.product && this.product.timeleft > 0) {
    this.lastupdate = Date.now();
    let progress = (this.product.vitesse - this.product.timeleft) /
this.product.vitesse;
    this.progressbar.set(progress);
    this.progressbar.animate(1, { duration: this.product.timeleft });
  }
}
```

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 13. 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 là doivent être vérifié par le composant `product.component.ts` à chaque augmentation de la quantité du produit.
- Ceux qui se déclenchent quand **tous** les produits ont atteint une certaine quantité (les *allunlocks*). Ceux là doivent être vérifié par le composant `app.component.ts` dès qu'un des produits voit sa quantité augmenté. La difficulté c'est que la prise en compte de ces *allunlocks* doit se faire au niveau de tous les produits et immédiatement. Il va donc falloir trouver un moyen pour le parent puisse parcourir la liste de ses enfants pour leur faire prendre en compte les *unlocks* généraux venant de se débloquent (voir ci-après).

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. Pour obtenir cet effet visuel il faut relancer un *animate* sur la barre en lui donnant le nouveau temps nécessaire à ce qu'elle soit totalement remplie. Les lignes ci-dessous réalisent cela :

```
this.product.timeleft = this.product.timeleft / this.bonusvitesse;  
this.progressbar.animate(1, { duration: this.product.timeleft });
```

A vous de voir où et quand les appeler dans la méthode *calcScore()* d'un produit.

Voyons maintenant comment le parent peut prévenir chacun de ces produits qu'un *unlock* général (*allunlock*) vient de débloquent. Mettons que le composant *product.component.ts* dispose d'une méthode *calcUpgrade(Pallier pallier)* qui calcule l'effet sur le produit du *unlock* passé en paramètre. Il faudrait que le composant *app.component.ts* puisse appeler cette méthode pour tous les produits enfants. Cela est possible en ajoutant à la classe *app.component.ts*, la propriété suivante :

```
@ViewChildren(ProductComponent) productsComponent: QueryList<ProductComponent>;
```

L'annotation *@ViewChildren* donne à accès à tous les composants enfants de type *ProductComponent*.

Ainsi on peut appeler la méthode *calcUpgrade(Pallier pallier)* de chaque composant de type produit avec la boucle suivante :

```
this.productsComponent.forEach(p => p.calcUpgrade(tu));
```

Vous voici donc doté d'un moyen simple permettant à un composant parent d'appeler n'importe quelle méthode de ses composants enfants, ce qui devrait vous permettre de gérer correctement les *allunlocks*.

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 13 – 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 :



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 11 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 14. Là encore, n'hésitez pas à n'afficher que les n premiers *upgrades* si la liste est trop longue.

Prévoyez également un badge « New » 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 14 – 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 15 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.

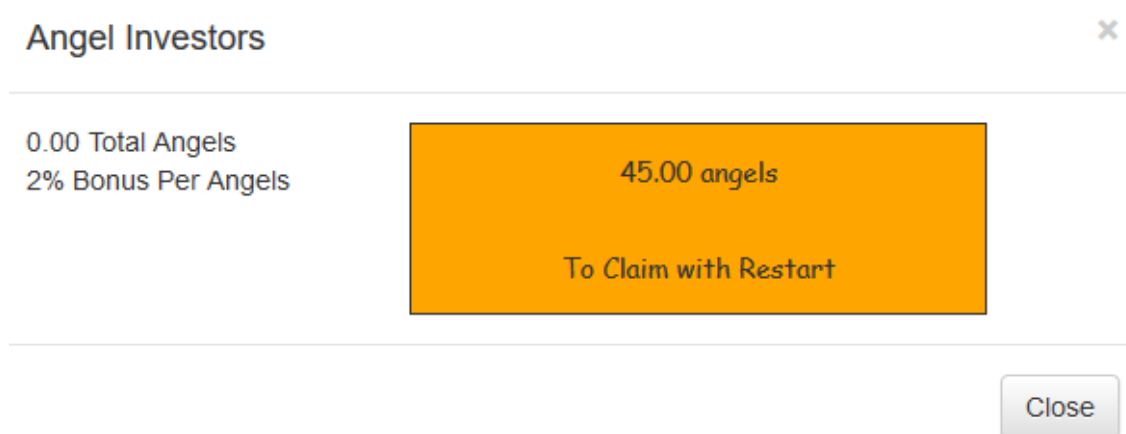


Figure 15 – Interface de gestion des anges

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 16.

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

Spend your Angels Wisely !



Angel Sacrifice

10.00 angels

All Products Profits x3

Buy !



Angelic Mutiny

100000 angels

Angel Effectiveness + 2%

Buy !



Angelic Rebellion

1.000 10^8 angels

Angel Effectiveness + 2%

Buy !



Angelic Selection

1.000 10^9 angels

All Products Profits x5

Buy !

Close

Figure 16 - 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 `restservice.service.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 et manipulant des représentations au format XML et JSON. Vous avez manipulé la plateforme Java EE et les *frameworks* web Angular et Bootstrap.

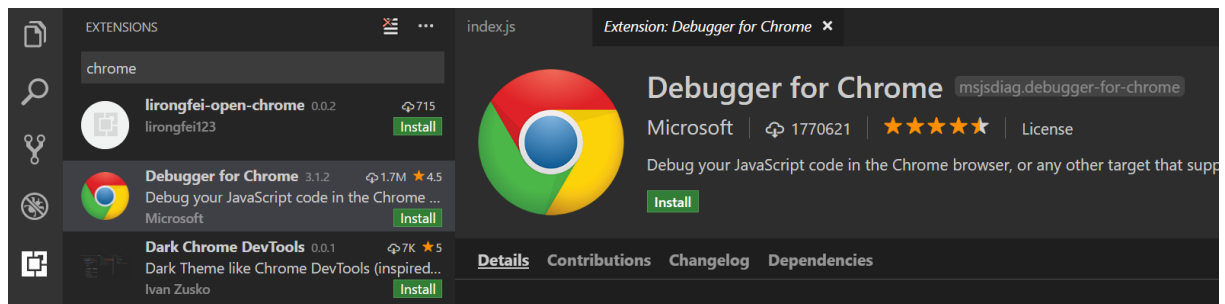
Vous avez de plus exercé vos compétences en calculs mathématiques et avez travaillé sur des problématiques de synchronisation temporelle entre un serveur et ses clients.

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 4200 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:4200",
      "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.