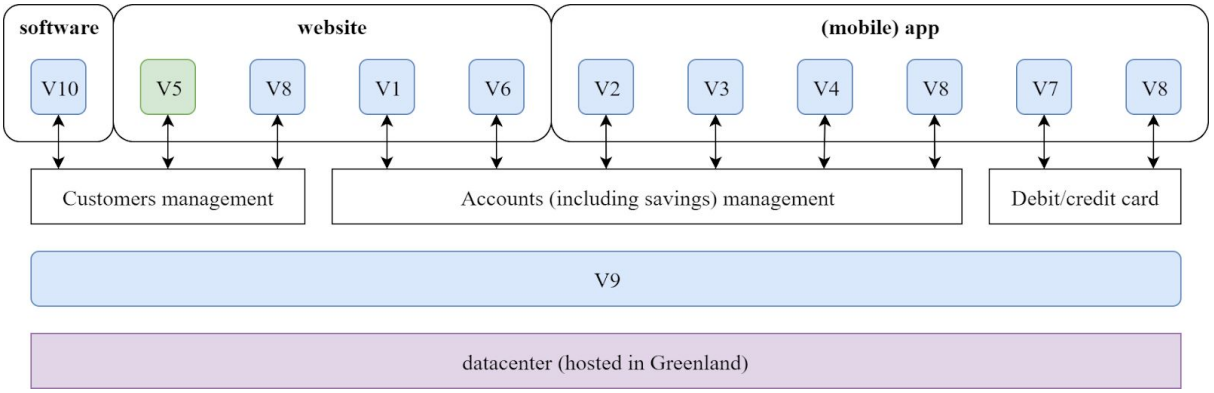
**V5: ARCHITECTURE**

# I *Semaine 40*

## I.1 Architecture de tout le système (avec la situation de chacune des variantes)




Réfléchir à l'architecture de tout le système, en nous efforçant de situer chacune des variantes, nous a permis d'avoir une vision plus nette du périmètre de notre projet.

Voici le scope choisi pour le projet sous forme d'user story, nous détaillerons le périmètre choisi pour la variante 5 ensuite :

- En tant que client, je souhaite créer un compte et renseigner mes détails. (US1)
- En tant que client, je souhaite pouvoir résilier mon compte. (US2)
- En tant que client, je souhaite pouvoir ajouter de l'argent sur mon compte. (US3)
- En tant que client, je souhaite pouvoir utiliser mon compte afin de payer. (US4)
- En tant que client, je souhaite avoir une carte bancaire. (US5)
- En tant que client, je souhaite pouvoir annuler ma carte bancaire. (US6)
- En tant que client, je souhaite pouvoir me connecter à mon compte. (US7)
- En tant que client, je souhaite pouvoir effectuer un virement. (US8)
- En tant que client, je souhaite avoir accès à l'ensemble des produits et services proposés par la banque. (US9)
- En tant que client, je souhaite pouvoir choisir un produit ou un service proposé par la banque. (US10)
- En tant que client, je souhaite pouvoir contacter mon conseiller bancaire. (US11)
- En tant que conseiller, je souhaite pouvoir créer un compte pour un client. (US12)
- En tant que conseiller, je souhaite pouvoir clôturer le compte d'un client. (US13)
- En tant que conseiller, je souhaite pouvoir ajouter et supprimer une carte bancaire au compte d'un client. (US14)
- En tant que conseiller, je souhaite pouvoir surveiller le compte d'un client. (US15)
- En tant que conseiller, je souhaite pouvoir bloquer un compte (par exemple retirer l'accès à la carte bancaire) d'un client. (US16)
- En tant que conseiller, je souhaite pouvoir accéder à la liste des clients dont je gère le compte. (US17)
- En tant que conseiller, je souhaite pouvoir contacter un client dont je gère le compte. (US18)

Concernant la création de compte, un utilisateur possédera un profil sur l'application, où il pourra avoir accès à un ou plusieurs comptes bancaires.

### I.2 Périmètre de notre variante

Notre variante consiste à intégrer un aspect webmarketing au projet à travers la conception, la mise en œuvre et la maintenance d'un système de recommandation de produits ou services de la banque, en se basant sur le profil des clients. Il peut être intéressant de nous tenir informés de ce que font les autres équipes, car elles proposeront peut-être des produits ou services pouvant être recommandés dans notre projet.

#### I.2.1 Nos utilisateurs

Le système sera utilisé par les personnes suivantes :

- un client de la banque (professionnel comme particulier)
- un conseiller bancaire

#### I.2.2 Ce que nous avons l'intention de couvrir d'un point de vue fonctionnel

Dans le cadre de notre variante (variante 5), nous voulons mettre en place les user story suivantes :

- En tant que client déménageant dans un autre pays, je souhaite qu'on me propose une solution adaptée au pays dans lequel je vais vivre. (USV1)
- En tant que client bientôt majeur, je souhaite recevoir une offre qui m'aide à m'engager. (USV2)
- En tant que client venant de perdre son emploi, je souhaite recevoir une proposition de formule adaptée à mes revenus diminués. (USV3)
- En tant que client achetant fréquemment sur Internet, je souhaiterais qu'on me recommande un produit qui facilite les achats en ligne. (USV4)
- En tant que client économe, je souhaite recevoir une notification lorsqu'une activité suspecte a lieu sur mon compte. (USV5)

*USV = user story pour notre variante

Quelle que soit la fonction, la recommandation doit être appropriée.

#### I.2.3 Ce qui est en dehors de notre périmètre

Les produits proposés à la suite d'une demande d'un client ou directement par le conseiller sont en dehors du périmètre. En effet, il faut toujours avoir à l'esprit le terme principal du sujet de notre variante, qui est la "recommandation", et qui se fera de façon automatique.

La gestion complète des produits n'entre pas dans le périmètre de notre projet car nous nous concentrerons sur la partie bancaire principalement. Nous implémenterons donc seulement une gestion simplifiée des produits afin de démontrer l'aspect principal de recommandation.

### I.3 Liste des types de produit et service auxquels nous avons déjà pensé

Chaque “USV” précédemment formulée appartient à l’un de ces types :

- produits recommandés en fonction du pays dans lequel le client est localisé
- produits recommandés en fonction de l’âge du client (livret jeune, livret A, compte d’épargne, PEL…)
- produits recommandés en fonction de la situation professionnelle du client (crédit, prêt, carte…)
- produits recommandés en fonction de certains comportements du client (service de facilitation des paiements en ligne…)
- services de notification (en cas d’opération suspecte sur un compte, en cas de comportement à risque…)

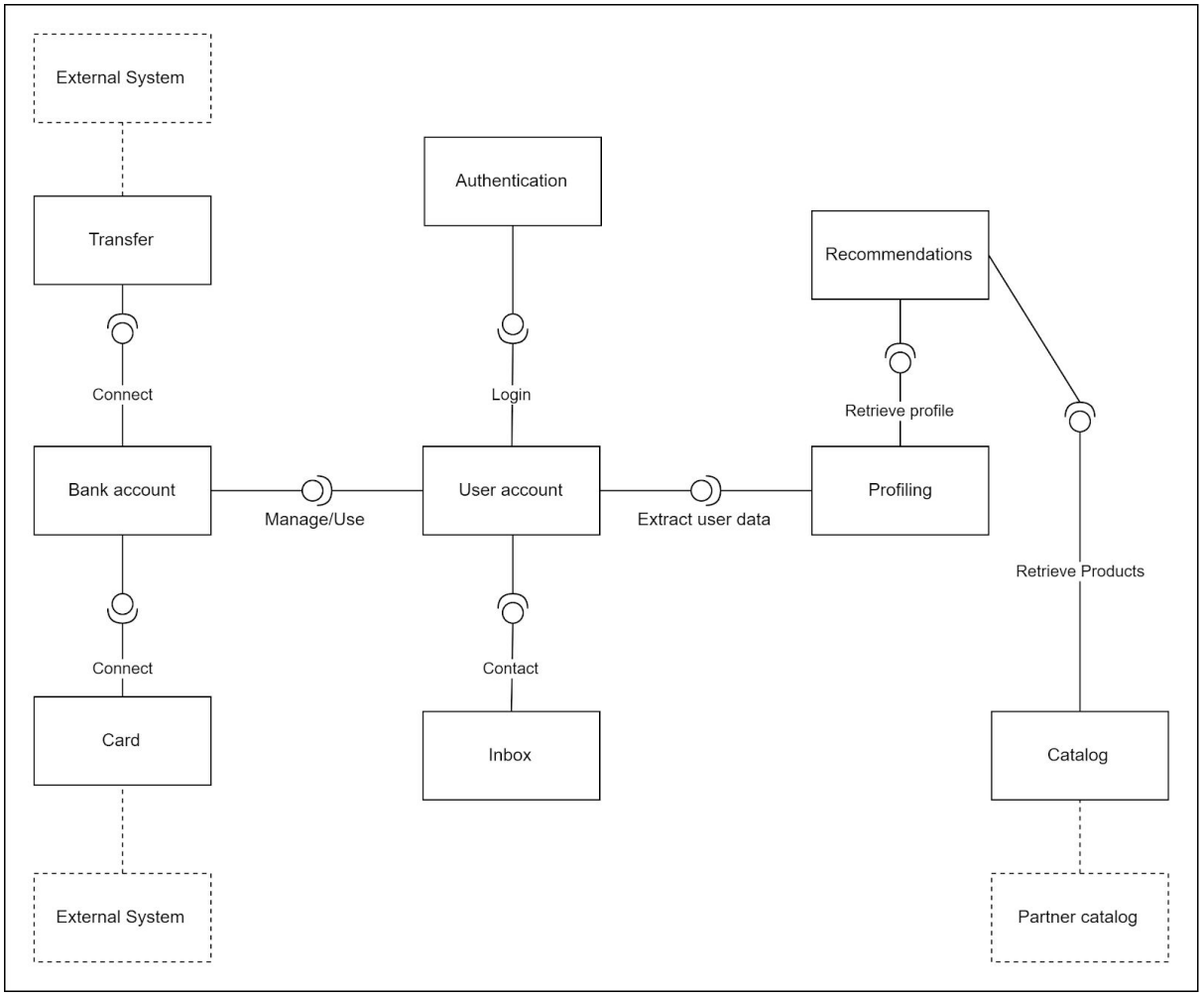
### I.4 Quelques pistes à propos des méthodes de profilage

Les algorithmes de profilage que nous mettrons en œuvre prendront en compte des données “statiques”, telles que les noms, âges ou encore les localisations, et des données “dynamiques”, telles que les historiques des clients (afin de dégager des habitudes de consommation, notamment).

Après quelques premières recherches sur les méthodes de profilage, nous avons trouvé certaines méthodes de *filtrage collaboratif* (“collaborative filtering”) intéressantes, telles que la méthode du “filtrage collaboratif *passif*”, par exemple, qui, consistant à analyser les comportements en “arrière-plan”, pourraient nous permettre de satisfaire à la “USV” 4 – au moins, partiellement.

# II *Semaine 41*

## II.1 Diagramme de composants




Voici la liste des différents composants prévus :

- Authentication : permet de créer un compte et de se connecter.
- User account : représente le compte “utilisateur” (que ce soit un client ou un conseiller bancaire). C’est ici que sont présentes les données sur un utilisateur.
- Profiler : récupère des données sur tous les comptes “utilisateur” afin de générer des profils.
- Catalog : contient tous les produits disponibles dans notre système.
- Recommendations : compare les profils générés par le Profiler avec les produits du Catalog, afin de proposer des produits aux clients ou aux conseillers.
- Inbox : représente le service de messagerie interne du système, entre les conseillers et les clients.
- Bank account : représente un compte bancaire d’un utilisateur.
- Transfer : représente le système des virements bancaires.
- Card : représente le système de gestion des cartes bancaires.

Nous avons décidé d'inclure d'éventuels services externes sur le diagramme de composants pour montrer où ils pourraient se greffer à notre système.

## II.2 Quelques scénarios

### II.2.1 Les personas

- Marcel, client de la banque, aura bientôt 18 ans.
- José, client de la banque, a 15 000 euros sur son compte.
- Killian, client de la banque, a 2 300 euros sur son compte.
- Noémie est conseillère bancaire.
- Patrick, client de la banque, est un homme de 45 ans.

#### II.2.2.1 Scénario 1

Marcel se connecte à son compte utilisateur ("**user account**") grâce au module d'authentification ("**authentication**"). Le module de recommandations ("**recommandations**") compare le profil généré par le profileur ("**profiling**") à partir des données recueillies sur Marcel avec les produits du catalogue ("**catalog**"). Il remarque que Marcel aura bientôt 18 ans et ne possède pas de compte "jeune". Il lui affiche alors une proposition pour créer un compte "jeune", adapté à sa situation.

#### II.2.2.2 Scénario 2

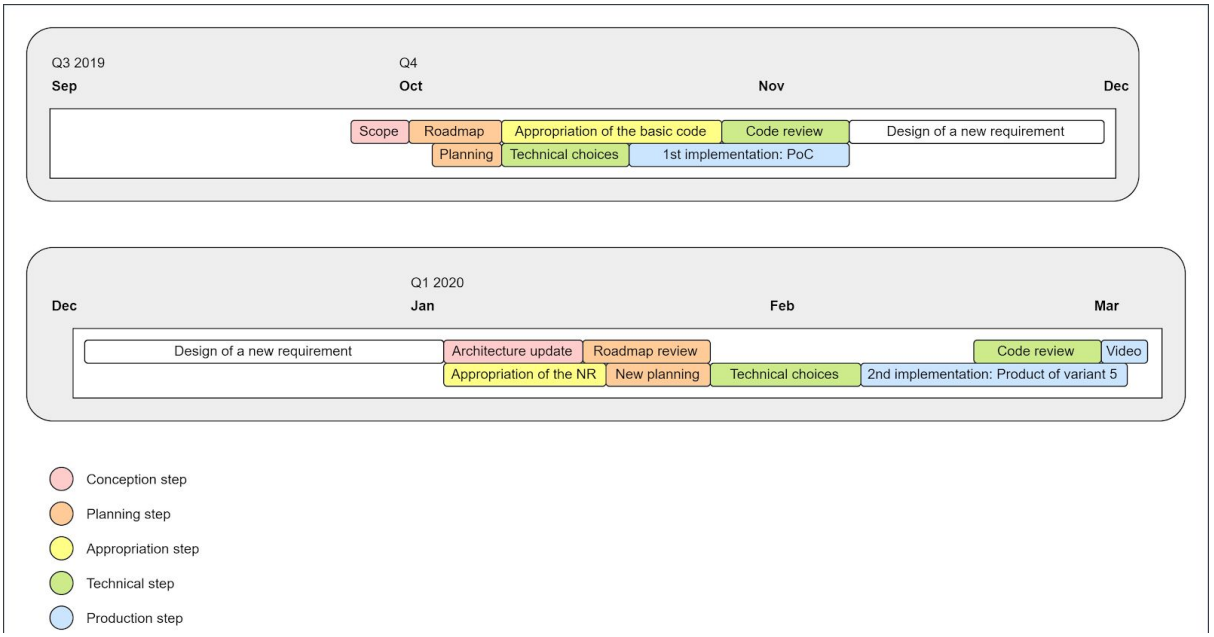
José et Killian possèdent tous deux un compte ("**bank account**") dans notre banque. José effectue un virement ("**transfer**") à destination de Killian, d'une valeur de 10 000 euros. Killian possède alors 12 300 euros sur son compte courant : dans les prochains jours, le système de recommandation ("**recommandations**") comparera le profil généré par le profileur ("**profiling**") pour Killian avec le catalogue des produits ("**catalog**"), et lui proposera de créer un compte ("**bank account**") avec un taux plus avantageux.

#### II.2.2.3 Scénario 3

Des opérations suspectes ont eu lieu sur le compte ("**bank account**") de Patrick, par exemple des sorties d'argent régulières pour un motif inhabituel. Le système de profilage ("**profiling**") indique cela en actualisant le profil de Patrick, et le système de recommandation ("**recommandations**") compare le nouveau profil avec le catalogue des produits ("**catalog**"). Il propose alors à Noémie, la conseillère de Patrick, la création d'un compte sécurisé ("**bank account**") pour son client, qui sera facturé à ce dernier chaque mois

mais qui le protègera des transactions suspectes. Noémie contacte alors Patrick grâce à la messagerie de l'application ("**inbox**") pour discuter de cette proposition.

## II.3 RoadMap




L'étape de "Design of a new requirement", qui est intermédiaire aux deux périodes, est une étape qui devrait être réalisée par les enseignants : au vu des PoC qui auront été livrés (vers le début du mois de novembre) par les différentes équipes, les enseignants nous proposeront, en effet, au début du mois de janvier sans doute (peut-être en décembre), une nouvelle exigence. ("Appropriation of the NR" désigne, à ce propos, l'étape d'appropriation de cette nouvelle exigence.)

Actuellement, bien que nous ayons une roadmap sur toute la durée du projet, il est trop tôt pour prévoir en détail ce qui est à faire ; voici, cependant, le planning prévisionnel des "release" des 3 ou 4 prochaines semaines, jusqu'à la livraison du PoC, c'est-à-dire jusqu'aux environs du (samedi) **9 novembre**.

## II.4 Planning

Etape 1 (**semaine 42**, du 14 au 20 octobre) : faire les choix technologiques, créer tous les composants, mettre en place une communication basique entre tous les composants pour montrer que tout fonctionne (walking skeleton).

Etape 2 (**semaine 43**, du 21 au 27 octobre) : mettre en place le scénario 1, donc mettre en place un système de recommandation et de profilage basique ainsi que la gestion sommaire

des comptes “utilisateur” (pouvoir renseigner son âge et son nom, pouvoir associer un compte bancaire à son compte “utilisateur”). En parallèle, implémenter les capacités basiques de notre application (créer un compte, mettre de l'argent dessus, vérifier qu'on peut en enlever) et faire quelques tests.

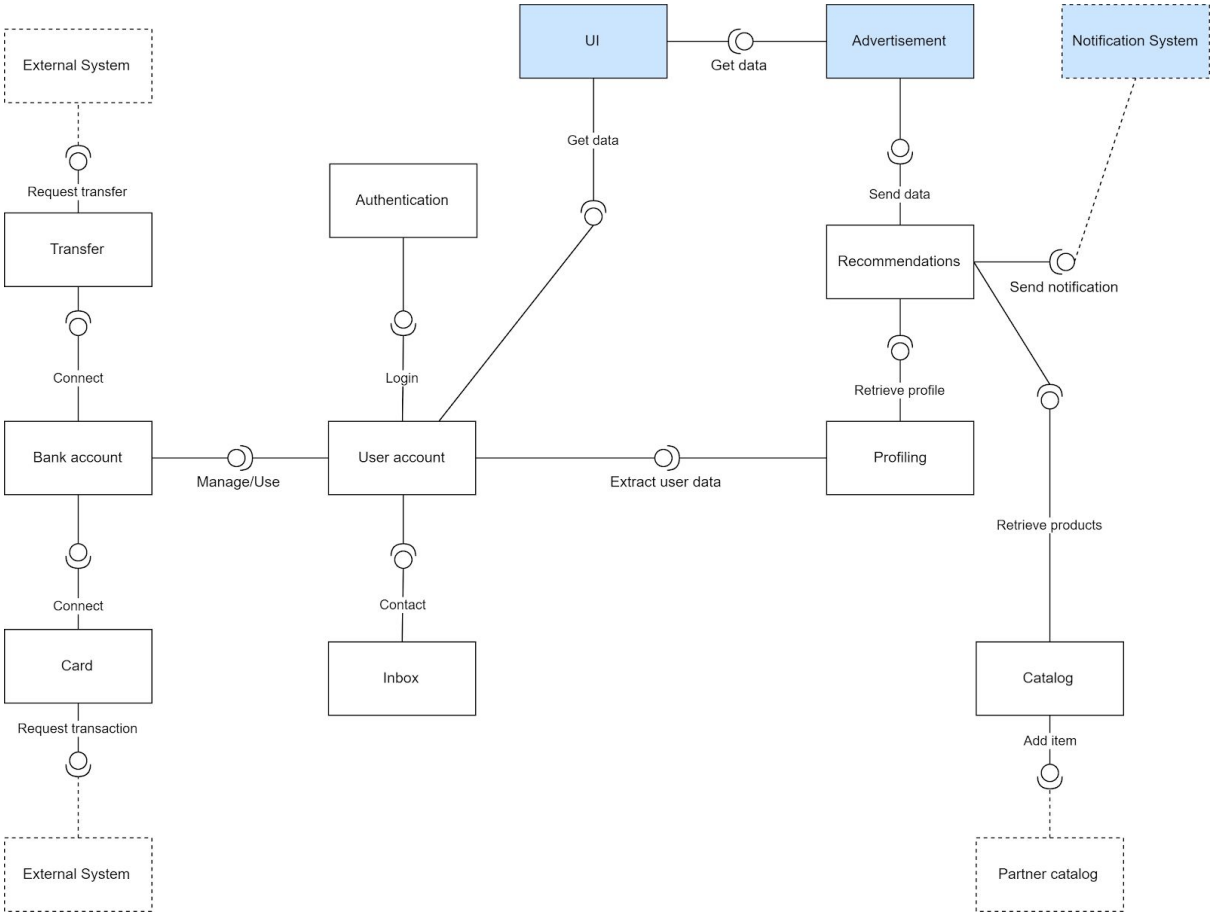
Etape 3 (**semaine 44**, du 28 octobre au 3 novembre) : mettre en place les scénarios 2 et 3, ajouter des tests d'intégration et compléter les tests unitaires, améliorer nos systèmes de profilage et de recommandation.

Ayant une approche “agile”, nous nous laissons la semaine 45 pour conserver une certaine souplesse de travail.

## III *Semaine 42*

### III.1 Diagramme de composants complet

Cette semaine, notre objectif est de parvenir à un “walking skeleton” ; or, afin de le remplir, il nous fallait définir les interfaces entre les composants et, pour définir les interfaces entre les composants, il nous fallait nous assurer que notre diagramme de composants était complet : en fin de compte, nous nous sommes rendus compte que plusieurs composants manquaient (coloriés en bleu dans le diagramme).




### III.2 Les composants

Avant de définir les interfaces (entre les composants) nous avons divisé notre architecture en blocs de composants ; nous avons alors distingué trois blocs importants :

- le bloc “métier”, formé des composants “Transfer”, “Bank account” et “Card” ;
- le bloc de communication, formé des composants “Authentication”, “User account” et “Inbox” ;
- le bloc du système de recommandation, formé des composants “Advertisement”, “Recommandations”, “Profiling” et “Catalog”.

Aux composants cités de chacun de ces blocs viennent évidemment s'ajouter les composants externes mentionnés dans notre diagramme ; l'UI est un composant à part, dont nous ne nous occuperons pas (au moins, dans un premier temps).

### III.2.1 Le bloc "métier"

Le langage que nous avons choisi d'utiliser pour implémenter les composants du bloc "métier" est **Java**, avec Spring.

Le choix de Spring a été motivé par le fait que celui-ci est un "framework" offrant de nombreux outils, notamment pour le REST (ce qui nous permettra de communiquer facilement avec la partie en Python, c'est-à-dire le bloc du système de recommandation) – et qu'il est orienté "composant". De plus, certains d'entre nous ont déjà de l'expérience avec Spring, ce qui nous permettra de gagner du temps, le projet se déroulant sur une période assez courte.

### III.2.2 Le bloc de communication

Nous avons choisi d'implémenter les composants du bloc de communication en **Java**, avec Spring.

### III.2.3 Le bloc du système de recommandation

**Python** a été choisi pour implémenter les composants de ce bloc pour ses "modules" (bibliothèques), nombreux, pour le machine learning et pour la clarté de son écriture.

## III.3 Les interfaces

### III.3.1 Le bloc "métier"

L'interface "**Request transfer**" (fournie par le composant "*Transfer*", à *un composant d'un système externe*) demandera d'implémenter la méthode dont l'en-tête est le suivant :

```
boolean requestTransfer(String senderAccountId, String receiverAccountId, double amount)
```

L'interface "**Manage/Use**" (fournie par le composant "*Bank account*", au composant "*User account*") demandera d'implémenter les méthodes dont les en-têtes suivent :

```
boolean createAccount(String userId)
Infos getInfos(String accountId)
```

L'interface "**Connect**" (fournie par le composant "*Bank account*", au composant "*Transfer*") demandera d'implémenter les méthodes dont les en-têtes suivent :

```
Infos getInfos(String accountId)
boolean canPayTransfer(String accountId, double amount)
```

L'interface "**Connect**" (fournie par le composant "*Bank account*", au composant "*Card*") demandera d'implémenter les méthodes dont les en-têtes suivent :

```
Infos getInfos(String accountId)
boolean canPayCard(String accountId, double amount)
```

L'interface "**Request transaction**" (fournie par le composant "*Card*", à *un composant d'un système externe*) demandera d'implémenter la méthode dont l'en-tête suit :

```
boolean requestTransaction(String senderAccountId, String receiverAccountId, double amount)
```

### III.3.2 Le bloc de communication

L'interface "**Login**" (fournie par le composant "*Authentication*", au composant "*User account*") demandera d'implémenter la méthode dont l'en-tête est le suivant :

```
boolean login(String id, String password)
```

L'interface "**Extract user data**" (fournie par le composant "*User account*", à *l'UI*) demandera d'implémenter la méthode dont l'en-tête suit :

```
Object extractUserData()
```

L'interface "**Contact**" (fournie par le composant "*Inbox*", au composant "*User account*") demandera d'implémenter la méthode dont l'en-tête devrait être :

```
boolean contact(String id, String message)
```

### III.3.3 Le bloc du système de recommandation

L'interface "**Retrieve profile**" (fournie par le composant "*Profiling*", au composant "*Recommandations*") demandera d'implémenter la méthode dont l'en-tête est le suivant :

```
Profile retrieveProfile(String id)
```

Le composant "Recommandations" ne fournit aucune interface.

L'interface "**Send data**" (fournie par le composant "*Advertisement*", au composant "*Recommandations*") demandera d'implémenter la méthode dont l'en-tête suit :

```
Object sendData()
```

Enfin, le composant "*Catalog*", qui est en dehors des (trois) blocs, fournira, d'une part, l'interface "**Retrieve products**" (au composant "*Recommandations*") – qui demandera d'implémenter la méthode "`List<Product> retrieveProducts()`" et, d'autre part, l'interface "**Add item**" (à *un composant d'un "catalogue" partenaire*) – qui demandera d'implémenter la méthode "`boolean addItem(Product product)`".

# IV *Semaine 44*

## IV.1 Choix de conception

Lors de la mise en place de notre architecture, nous avons considéré les hypothèses suivantes :

- Les utilisateurs doivent être connectés pour recevoir des recommandations de l'application.

- Un utilisateur enregistré possède un unique compte utilisateur sur l'application, auxquels peuvent être liés plusieurs comptes bancaires.

- Un conseiller bancaire s'enregistre et se connecte de la même façon qu'un utilisateur à l'application, il possède simplement un rôle différent lui accordant d'autres droits qu'un utilisateur lambda.

- Les recommandations de produit seront affichées sur l'application à la manière d'une publicité. Dans une version sans UI, cela correspondra simplement à un envoi de message dans la console.

Nous avons également décidé les choses suivantes :

- Nous ne mettrons pas en place d'UI : nous avons renseigné les endroits où celle-ci prendrait place dans notre diagramme d'architecture, mais ceci n'est qu'à titre indicatif.

- Le comportement d'aucun composant ne sera mocké, mais les comportements resteront simples. Par exemple, le système de recommandation ne reposera pour le moment pas sur des techniques de machine learning avancées, mais seulement sur des opérateurs conditionnels simples.

- Notre projet est conçu dans l'idée de pouvoir y brancher des systèmes externes (virements, carte bancaire, catalogue partenaire). Cependant, nous ne mettrons pas de réel système externe en place, au moins pendant la première partie du projet.

### IV.2 Justifications

Nous avons décidé de découper notre architecture en plusieurs blocs de composants. Les blocs ont été découpés selon la logique métier qu'ils mettent en place. Ainsi, nous possédons un bloc de composants permettant de mettre en place la logique principale de l'application (création de compte, login, informations de l'utilisateur, message interne), un autre représentant les comptes bancaires et les transactions correspondantes (virements bancaires, cartes bancaires) et enfin un bloc permettant de proposer des produits au client (profilage, recommandation et catalogue de produits).

Cela présente plusieurs avantages :

- Le découpage en blocs logiques permet de limiter les communications vers l'extérieur : dans l'architecture actuelle, les seules communications vers l'extérieur se font entre le compte utilisateur et les comptes bancaires, ainsi qu'entre le compte utilisateur et le système de profilage. Ainsi, nous avons besoin d'exposer une moins grande quantité de données que si notre système était plus découpé (par exemple dans le cas d'une architecture micro services).

- Le découpage en blocs logiques permet également d'augmenter la clarté de l'architecture en séparant plus clairement les responsabilités de chacune des parties du code, ce qui facilite grandement le développement.

- Chaque membre de l'équipe peut facilement faire progresser un bloc de fonctionnalité en parallèle, il suffit à l'équipe de se mettre d'accord sur les interfaces et le type de données à envoyer. Cela permet au projet de progresser plus rapidement, tout en limitant la difficulté de maintenabilité que l'on rencontrerait dans une architecture monolithique (nombreuses dépendances internes) ou micro services (nombreux composants) à long terme.

Cependant, nous avons également remarqué plusieurs inconvénients à cette architecture :

- La présence de services externes, ainsi que l'augmentation en complexité de chacun des blocs peut à terme mener à des blocs de taille bien trop importante. Il s'agira dans ce cas de découper à nouveau les blocs devenus trop grands, pour limiter la complexité à maintenir le code.

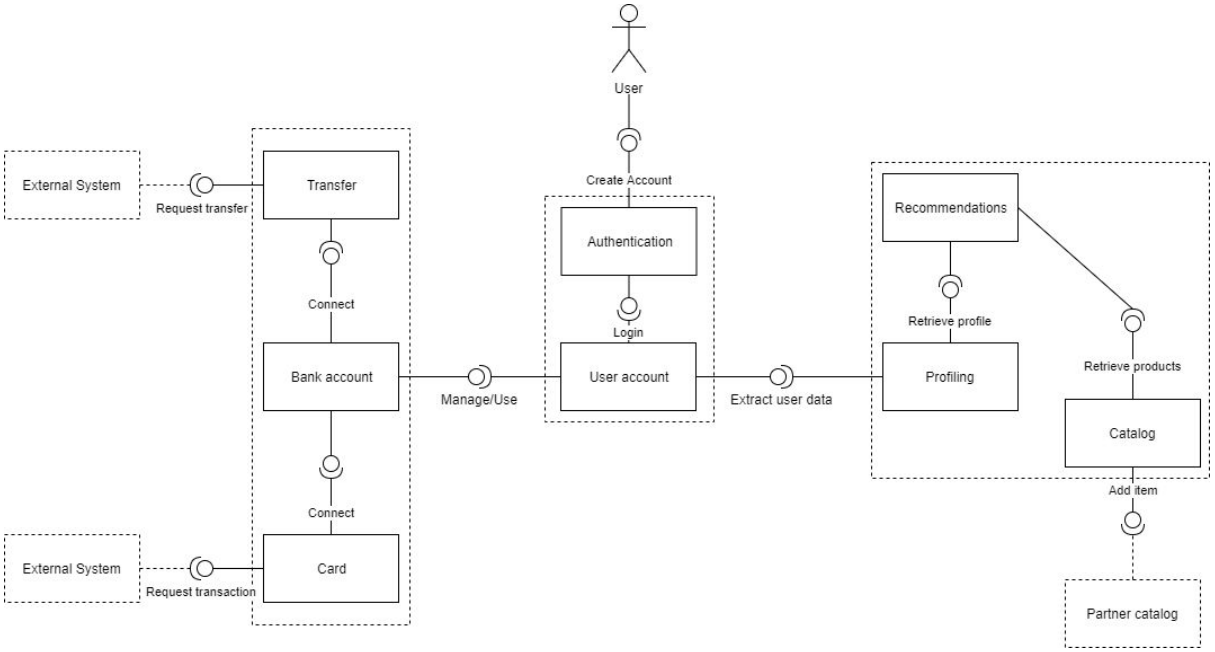
- Au bout d'un certain temps, certains blocs logiques, par exemple le compte bancaire, ne nécessiteront plus d'ajouter des fonctionnalités. L'équipe complète devra alors travailler sur un ou deux blocs de fonctionnalités, ce qui complexifiera le développement de l'application, ainsi que sa maintenance.

- Actuellement, le compte utilisateur représente un SPOF (single point of failure) de notre architecture : si le composant n'est plus accessible, nous ne pouvons plus nous connecter ni accéder aux comptes bancaires, et le système de profilage ne peut plus

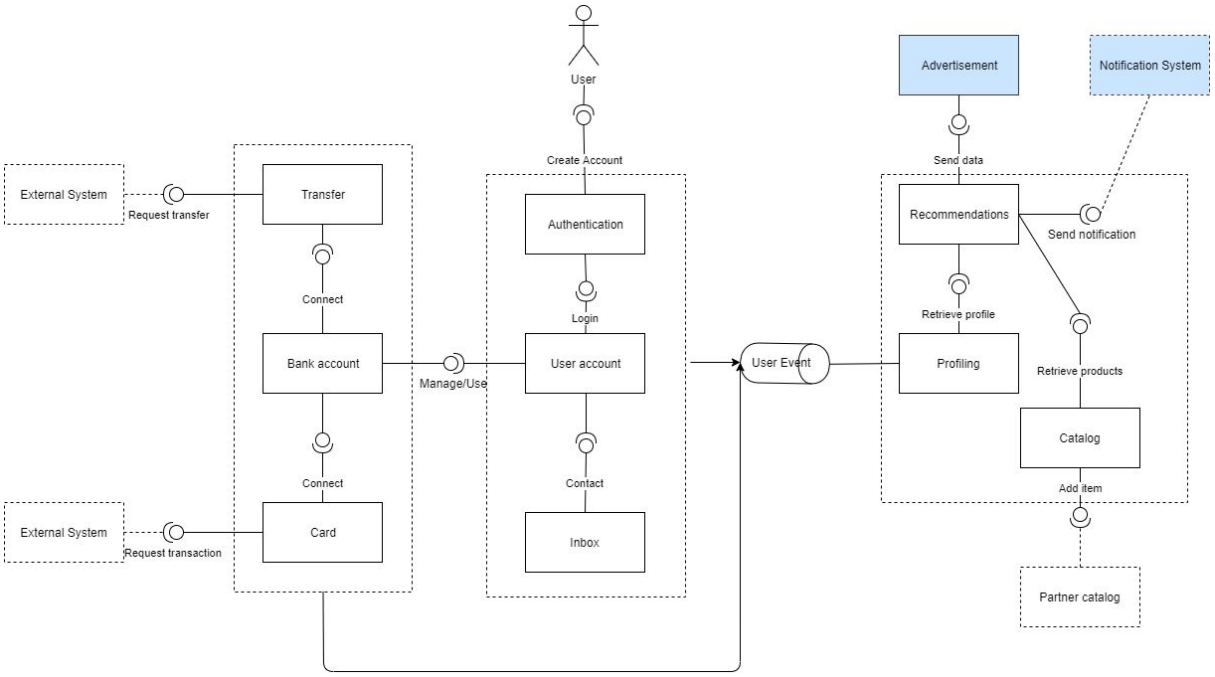
récupérer de données, donc le système de recommandation ne peut plus proposer de produits. Cela représente un gros point négatif de notre architecture, puisqu'il suffit d'un seul problème technique sur un composant pour que l'application perde toutes ses fonctionnalités. Cela est dû au fait que le compte utilisateur est trop central dans notre architecture, et relié à un trop grand nombre de composants et autres blocs de fonctionnalités.

# V *Semaine 46*

## V.1 Diagramme d'architecture actuel




## V.2 Diagramme d'architecture prévu pour l'implémentation du scénario 3




Au lieu que le système de Profiling récupère les informations de façon régulière auprès des utilisateurs, nous envoyons les informations des utilisateurs et des comptes bancaires dans un topic kafka, qui sera consommé au besoin par le système de profilage. Cela permet de simplifier le transfert des données, et de limiter le nombre de communications synchrones.

En effet, dans la version actuelle de l'architecture, le système de profilage ne peut récupérer des données que sur les comptes utilisateurs, et doit le faire à travers des requêtes REST : si un des services n'est pas disponible, rien ne fonctionnera comme voulu, et ce n'est pas le comportement souhaité.

Nous avons également remis dans le diagramme le composant s'occupant de la messagerie interne de notre application, qui n'était pas utilisé dans les deux premiers scénarios, mais sera nécessaire pour accomplir le troisième scénario.

# VI *Semaine 3*

## VI.1 Nouvelles exigences du sujet

Afin de respecter les nouvelles exigences du sujet "Simulation. Noémie veut ajouter un nouveau produit bancaire. Elle peut exécuter une simulation sur l'ensemble des clients, qui rejoue les recommandations, et elle peut observer ce qui a été recommandé à quelle population.", nous avons défini un nouveau scénario :

**Scénario "Simulation" :**

Noémie, la gestionnaire bancaire, ajoute un produit bancaire au catalogue à l'aide du module "Catalog". Elle peut vérifier la liste des produits bancaires proposés par la banque : le nouveau produit ajouté y apparaît correctement.

Ensuite, elle décide de lancer une simulation à l'aide du module “Recommendations”. Elle obtient ainsi la liste des produits recommandés pour chacun des clients de la banque. Elle remarque alors que le produit qu'elle vient d'ajouter est correctement proposé à certains clients. De plus, elle peut étudier plus en détail le résultat obtenu afin de vérifier que tout est cohérent. Enfin, en accédant à l'historique des recommandations, elle peut comparer la simulation avec les dernières recommandations faites.

Pour cela, nous avons décidé d'apporter les changements suivants :

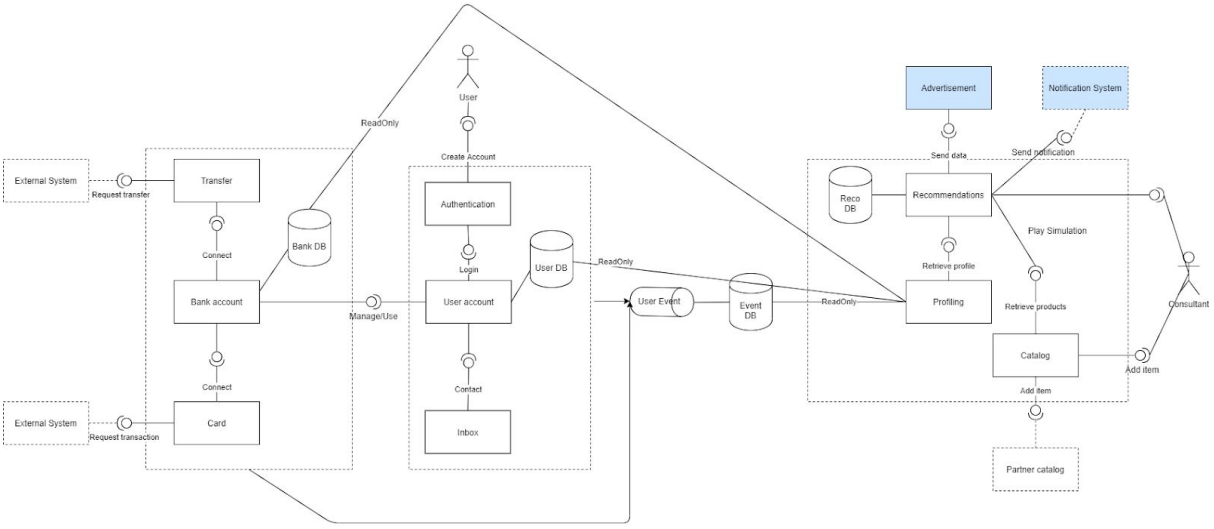
- création d'une base de données pour stocker l'ensemble des événements utilisateurs envoyés (chaque fois que le profil d'un utilisateur est modifié, c'est à dire à chaque transfert d'argent, modification de ses informations etc)

- création d'une base de données pour stocker la recommandation la plus récente faite aux clients, ainsi que la précédente. Cela permettra de constater aisément l'évolution des recommandations.

- permettre l'ajout d'un produit grâce à une interface sur le catalogue (voir même sous la forme d'un DSL). Par la même occasion, compléter le code du catalogue, puisque la liste des produits est actuellement hardcodée.

- permettre le lancement d'une simulation pour observer les différentes recommandations faites aux clients.

### VI.2 Diagramme d'architecture mis à jour



Les événements utilisateurs seront envoyés dans le bus Kafka “UserEvent”, qui se contentera de les stocker dans la base de données “Event BD”. Tous les jours à minuit (en théorie, mais notre système n’est pas actif la nuit), le système de profilage récupère tous les événements dans la base Event BD ainsi que toutes les données dont il a besoin dans les bases User BD et Bank BD (qui contiennent respectivement les informations des comptes utilisateurs et les informations des comptes bancaires) afin d’établir à nouveau le profil de chacun des clients, pour pouvoir effectuer des recommandations.

Par rapport à notre architecture précédente on note plusieurs modifications :

- Dans la version précédente, nous n’avions pas de traçabilité dans notre système. Cela pose notamment des inconvénients pour la mise en place de notre système de recommandations. En effet, nous voulons pouvoir effectuer des recommandations non seulement sur l’état actuel du système (compte en banque d’une personne, âge de la personne, ...) mais aussi sur les événements qui ont eu lieu pour que le système en arrive là (nombre de transactions effectuées, si une transaction dépasse un certains montant etc). Ainsi, dans cette nouvelle version nous allons mettre en place de l’event logging.

- Dans la version précédente nous partions du principe que notre système de recommandation s'occuperait aussi de gérer les évènements de sécurité. Ce n’est plus le cas maintenant. Vu que le bus ‘UserEvent’ reçoit tous les évènements de logs, si l’on veut gérer la sécurité il suffira de rajouter un consommateur sur le topic.

L’utilisation d’un bus n’est pas forcément nécessaire. Cependant, comme explicité précédemment cela permet de rajouter facilement de l’analyse et du traitement sur les événements. Ce bus permet aussi d’encaisser la charge à la place de la BD Event.

Comme explicité précédemment nous allons faire de l’event logging. Nous pensions dans un premier temps faire de l’event sourcing mais nous nous sommes rendu compte que ce n’était pas adapté à notre système actuel :

- D’une part car cela n’a que très peu de sens de remplacer le CRUD dans la partie UserAccount par de l’event sourcing.

- D’autre part pour la partie BankAccount / paiement, utiliser de l’event sourcing a de l’intérêt. Cela nous permettrait en l’alliant avec du CQRS d’avoir des meilleurs performances, une meilleure disponibilité et de faciliter la maintenance. Cependant, cela nous poserait des inconvénients au niveau de la persistance (comment être sûr qu’un utilisateur a assez d’argent pour payer si tous les events le concernant ne sont pas encore traités ?). Pour palier à cela il faudrait utiliser un ‘aggregate’ pour la gestion du dépôt d’un compte. Ainsi, cela compliquerait énormément notre architecture à ce niveau là alors que ce n’est pas le point d'intérêt de notre projet. Nous préférons nous concentrer sur le système de recommandation.

En conclusion, notre architecture comporte des défauts assumés car nous avons choisis de nous concentrer sur le système de recommandation.

## VII *Semaine 4*

Nous avons décidé de mettre en place un DSL pour permettre à notre conseillère bancaire d'ajouter un produit au catalogue. Cela lui permettra d'utiliser un langage plus adapté à son niveau en informatique, puisqu'elle n'a aucune notion en développement.

Nous avons également décidé d'utiliser les fonctions Lambda AWS pour la partie recommandation et la simulation. Cela nous permettra de supporter le grand nombre de calculs à effectuer à chaque recommandation. Ce choix est majoritairement justifié par le fait que les recommandations et simulations ne sont lancées que rarement (les recommandations une fois par jour à minuit, les simulations manuellement, assez peu fréquemment) et constituent donc un énorme pic de charge : en dehors de cela, le service ne sera pas utilisé et il est donc inutile de le maintenir constamment en ligne. Les fonctions Lambda AWS semblent donc parfaitement adaptées à ce cas de figure.

## VIII *Semaine 6*

Nous avons terminé le DSL pour ajouter un produit. Si nous avions mis en place une vraie application bancaire avec un front end, l'ajout de produit se ferait sous la forme d'un formulaire, où le conseiller aurait simplement des cases à remplir.

Afin de rester dans cette optique, nous avons finalement décidé de présenter notre DSL sous une forme simple : il consiste en un appel REST avec un String écrit sous la forme suivante (sous forme BNF) :

create product <productName> with condition <importantField> <operator> <amount>
…
Par exemple, on peut avoir la déclaration de produit suivante :
create product produit1 with condition money > 500

Cette déclaration permet de créer le produit avec le nom "produit1", dont la condition de recommandation est que l'argent total sur les comptes d'un utilisateur est supérieure à 500.

Dans l'état actuel, nos produits n'ont qu'un seul champ utilisé pour la recommandation, mais à terme, il sera facile d'étendre le DSL :

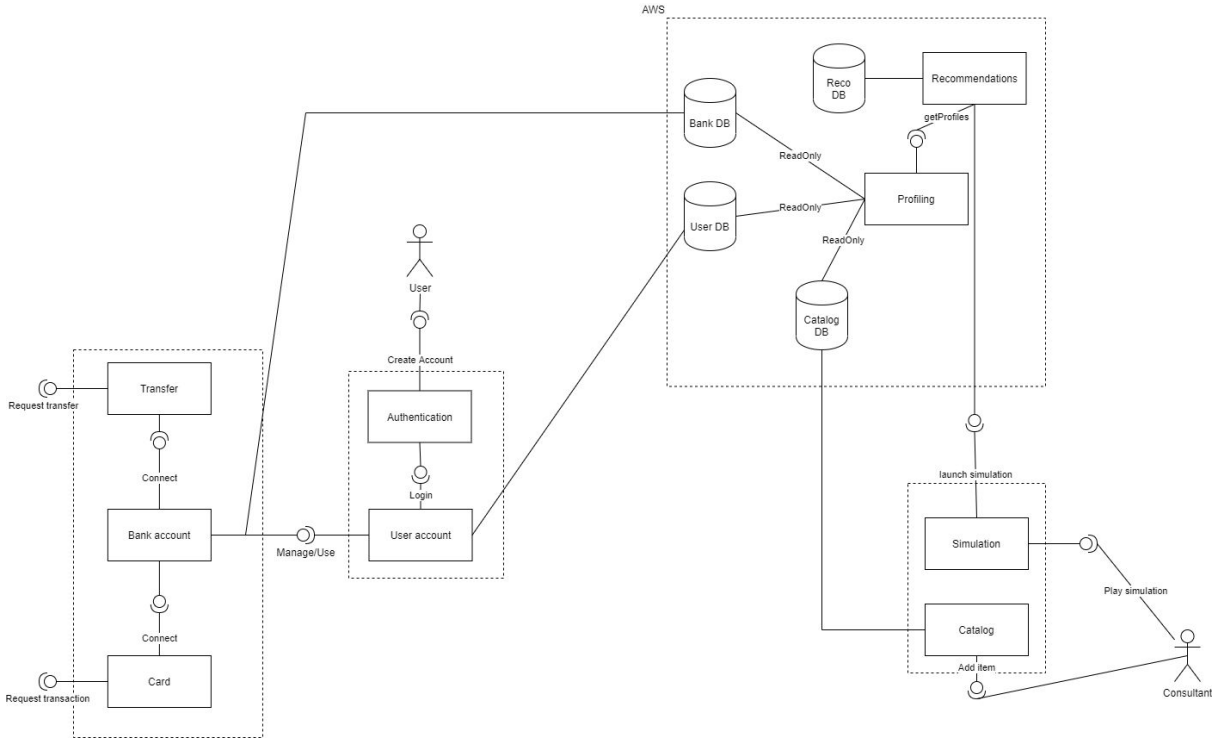
create product <productName> with condition <importantField> <operator> <amount> [and <importantField> <operator> <amount>]

A chaque requête, on peut ajouter plusieurs produits, en écrivant plusieurs fois cette phrase, en les séparant d'un saut de ligne.

## IX *Semaine 9*

### IX.1 Architecture actuelle

Voici notre architecture actuelle :




Nous n'avons pas mis en place le bus d'événements présent sur le précédent diagramme d'architecture, puisque nous avons rencontré plusieurs problèmes avec AWS.

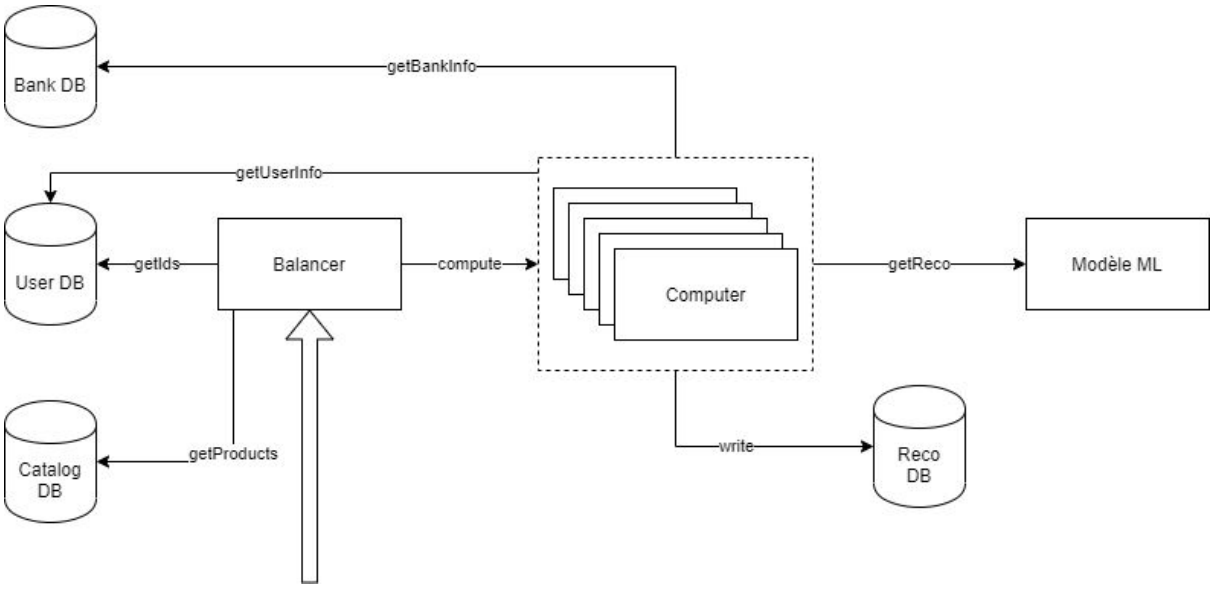
En effet, nous utilisons une version étudiante d'AWS, qui limite grandement la gamme de machines que nous pouvons utiliser : il nous est donc impossible d'entraîner par nous-même un modèle de machine learning, car aucune des machines proposées n'offre cette possibilité.

Suite à cela, nous avons pensé à utiliser un modèle déjà entraîné afin de mettre en place une communication correcte avec un réel modèle de machine learning, ce qui nous aurait permis de soulever les difficultés de connexion sous jacentes. Cependant, cette solution ne s'est pas avérée viable car les algorithmes utilisés par AWS ne sont pas exportables (on ne peut pas utiliser leur algo sur notre propre machine) et AWS n'accepte que ses propres modèles comme entrée (vendor lock in très important à ce niveau là).

Ainsi, nous avons décidé de créer une fonction Lambda AWS qui mocke les recommandations, et nous renvoie ainsi une liste de produits au hasard de notre catalogue pour chacun des utilisateurs, après un temps d'attente 'forcé'. Cela nous permet de simuler la durée que prendrait un vrai modèle à nous renvoyer une réponse, et nous pouvons donc l'utiliser pour vérifier que notre système résiste correctement à la charge, lorsque nous lançons une recommandation pour un grand nombre d'utilisateurs.

### IX.2 Fonctions Lambda AWS

Le schéma suivant permet d'expliquer le fonctionnement de nos fonctions Lambda AWS.




Le Balancer récupère la liste des produits disponibles dans notre catalogue, ainsi que la liste des ID des utilisateurs de notre système. Ensuite, pour chaque utilisateur une requête va être envoyée vers la fonction Computer. On aura alors autant d'instances de la fonction qui vont se lever que l'on a d'utilisateurs dans la BD.

La lambda fonction Computer prend ainsi en entrée l'id d'un utilisateur ainsi que la liste des recommandations. Elle va alors récupérer les informations bancaires et les données du compte de ce client afin de générer son profil et d'appeler le modèle de machine learning pour obtenir une liste de produits recommandés. Dès que le modèle de machine learning a renvoyé sa réponse au Computer, ce dernier se charge d'écrire les recommandations dans la base de données correspondante (Reco DB).

Les fonctions sont écrites en Node.js. Comme un des gros inconvénients du serverless est le temps d'instanciation des fonctions, il nous a paru judicieux d'utiliser un langage qui minimise ce temps d'instanciation.

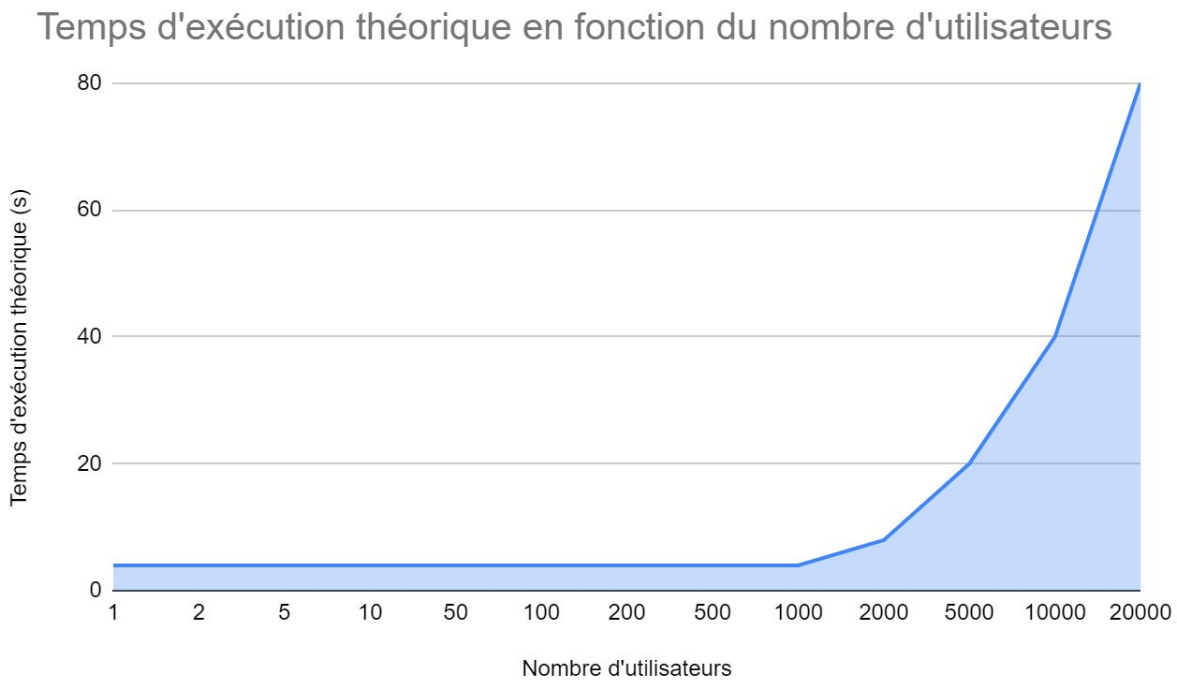
Avec du recul il aurait été plus judicieux de traiter d'envoyer les utilisateurs vers le computer non pas un par un mais par lot afin de profiter des optimisations batch quand on va lire dans les différentes bases de données.

Nous n'avons pas expliqué jusqu'à présent pourquoi nous avons choisi ce format (serverless) de cloud computing. On fait du cloud computing pour s'adapter à la charge, mais pourquoi du serverless et pas une solution Iaas avec de l'autoscaling ? La réponse est simple :

- D'une part le serverless est la solution qui nous permet de déléguer un maximum la gestion de l'élasticité. Nous ne perdons pas de temps à établir des règles d'autoscaling.

- D’autre part, le serverless est le système avec le vendor lock in le plus important. Cela engendre des inconvénients évident si l’on veut changer de solution. Cependant, comme le framework est très cadré et très restrictif, AWS peut optimiser la solution bien plus qu’en Iaas ou en Paas.

### IX.3 Montée en charge




Nous avons rencontré plusieurs problèmes limitant le scaling de la solution. Tout d’abord, MongoDB ne supporte pas plus de 80 connexions simultanées avec la version étudiante d’AWS. Cela limite grandement le nombre de recommandations que nous pouvons traiter à la fois, puisque chaque Computer se connecte à MongoDB.

Si nous n’étions pas limités par MongoDB, nous aurions rencontré le bottleNeck théorique visible sur le graphique ci-dessus. On ne peut avoir que 1000 fonctions Lambda en parallèle. Cela signifie que si on a plus de 1000 utilisateurs dans la BD, tous les utilisateurs ne seront pas traités en même temps car AWS refusera de lever des instances supplémentaires.

Cette restriction engendre un autre problème : si le balancer a plus de 1000 utilisateurs à traiter, avec notre système actuel il les envoie tous au computer en même temps. On risque ainsi de perdre des utilisateurs en chemin car le computer va refuser les appels quand 1000 instances tournent déjà. Pour remédier à cela, il faudrait découpler le Balancer et le Computer en mettant une queue entre les deux. Le Balancer envoie les requêtes dans la queue tandis que le Computer se contente de consommer les éléments dans la queue. On

est toujours limités par le nombre de fonctions lambdas qu'on peut lancer simultanément mais au moins on ne perd plus de requêtes en chemin.