Table des matières

Remerciements

Michelle Henry et toute l'équipe de STE-Formations, qui ont su me convaincre d'entreprendre ces études.

Le corps professoral de l'Institut Saint Laurent pour ces quatre années d'enseignement.

Enfin, Isabelle pour son soutien inconditionnel.

1. Introduction

1.1. L’application

Dans le cadre de notre épreuve intégrée, nous avons choisi de développer un site de socialisation dont les thématiques concernent le monde de la cuisine.

Le contenu du site proviendra des actions de ses utilisateurs. Ces derniers pourront en effet créer ou évaluer des recettes, des restaurants, des ustensiles, etc. Ces pages pourront ainsi servir de référence aux visiteurs à la recherche d’informations.

Les membres du site, qui seront passés au préalable par une procédure d’inscription, auront accès à une page de profil dans laquelle ils pourront indiquer divers renseignements les concernant. Ces éléments seront ensuite visibles par les autres membres. Dans cet espace personnel, les membres pourront également retrouver l’ensemble de leurs contributions sur le site, utiliser le service de messagerie interne du site, gérer leur agenda, gérer leurs préférences.

Pour être en adéquation avec les besoins des internautes actuels, l’application devra être capable d’interagir avec certains réseaux sociaux. Le visiteur pourra s’inscrire à l’aide de son compte sur un de ces réseaux, ou encore partager certaines de ses actions.

Dans une optique d’ouverture sur le monde extérieur et d’interopérabilité des systèmes, nous proposerons également une API publique permettant de consulter nos données et d’interagir avec l’application.

1.2. Architecture de l'application

1.2.1. Modèle client-serveur

L'application développée est une application web, accessible au travers du protocole HTTP. Le principe de ce modèle repose sur les interactions entre un serveur et un client: le client fait une requête au serveur, le serveur sert une réponse en retour. Un serveur doit être capable de traiter une multitude de requêtes provenant d'une multitude de clients.

1.2.2. Architecture n-tier

Notre application sera développée selon un modèle d'architecture trois-tier, une extension du modèle client-serveur. Ce modèle aide le développement d'applications flexibles et réutilisables. Pour ce faire, il définit trois couches distinctes.

1.2.2.1. Couche présentation

C'est l'interface présenté à l'utilisateur. C'est avec elle qu'il interagit, et c'est elle qui affiche les informations renvoyées en retour. Elle ne doit contenir aucune logique qui ne soit directement liée à l'affichage.

1.2.2.2. Couche métier

C'est dans cette couche que sont effectués les traitements métiers, la logique de l'application. La couche prend en compte les actions de l'utilisateur et retourne éventuellement un résultat.

1.2.2.3. Couche d'accès aux données

Il s'agit de la couche qui s'occupe de la persistance des données. Elle est manipulée par la couche métier exclusivement, il n'existe pas de lien direct entre elle et la couche présentation.

Ce modèle respecte une approche de "separation of concerns" (SOC). En séparant le "quoi" du "comment", il permet de réduire le couplage entre le modèle et la vue.

En effet, la séparation des services en processus distincts permet de faire évoluer séparément les couches sans impacter toute l'application. Il devient également possible de changer de technologie de présentation sans modifier les autres couches. ou encore d'utiliser des clients de types différents. On pense par exemple à une base de données PostgreSQL qui peut être accédée par le logiciel pgAdmin ou par une application Java à l'aide du driver JDBC *ad hoc*, ou encore à un web service exécuté sur un serveur d'application Java et dont les données sont récupérées indifféremment par un client Javascript ou un client .Net.

Afin de respecter cette architecture à une échelle plus large, nous avons choisi de structurer ce document de la même manière. Après une introduction générale relative aux technologies utilisées, nous présenterons tout d'abord la couche de données, avant de continuer avec la couche métier, pour terminer par la couche de présentation.

1.3. Java

Le langage choisi pour notre application est le langage Java, dans sa version 1.7.0.45. Java est un langage de programmation orienté objet, qui existe depuis le milieu des années 90. Parmi ses caractéristiques, on peut noter la portabilité et l'orientation objet.

1.3.1. Portabilité

L'objectif principal de Java a toujours été la plus grande portabilité possible. Cela signifie que le code généré par le compilateur n'est pas spécifique à un environnement (processeur, système d'exploitation). Ce code, appelé byte code, est destiné à être exécuté par une machine Java virtuelle (la JVM: Java Virtual Machine), qui, elle, est spécifique à une plate-forme.

Ainsi, du code peut être écrit sur une machine Linux, compilé sur une machine Windows, puis utilisé sur une machine sous Mac Os. On peut aussi mentionner Dalvik, qui est la machine virtuelle intégrée à Android, le système d'exploitation portable de Google. Ce modèle de compilation est appelé "write once, run everywhere".

1.3.2. Langage orienté objet

La pogrammation orientée objet (POO) est un paradigme de programmation qui, comme son nom l'indique, repose fondamentalement sur la notion d'objet. Un objet en POO est une structure de données qui cherche à représenter un objet du monde réel, ou un concept. Cette structure est composée de champs et de méthodes, que l'on appelle ses membres, et qui permettent de décrire ou de manipuler l'objet. On distingue une classe (le type) d'une instance (l'objet à proprement parler): une classe peut être instanciée de multiples fois, avec des caractéristiques différentes.

Nous avons par exemple modélisé dans notre application la classe User, qui représente un utilisateur du site. Cet objet contient des attributs comme le nom de l'utilisateur, son adresse email, etc. Il existera de multiples instances de l'objet User, dont les valeurs de ces attributs varieront en fonction de l'utilisateur représenté.

En tant que langage orienté objet, Java permet de respecter les trois principes fondamentaux de la POO: l'encapsulation, l'héritage et le polymorphisme.

1.3.2.1. Encapsulation

L'encapsulation est fortement liée au concept d'objet. En effet, il s'agit de réunir dans le même objet différentes données, tout en cachant au monde extérieur les détails de son fonctionnement internet, et en limitant l'accès à ses données si nécessaire.

Pour ce faire, les propriétés et les méthodes des objets Java possèdent un attribut qui permet de définir la manière dont ils peuvent être accédés: ils peuvent être déclarés public, private ou protected. Le plus souvent, toutes les propriétés d'un objet seront déclarées privées, ce qui signifie qu'elles ne sont pas accessibles depuis l'extérieur. Si on juge nécessaire de laisser un

accès à ces données, on munira alors la classe de méthodes publiques qui permettent de les manipuler tout en gardant le contrôle sur cet accès.

1.3.2.2. Héritage

Le concept d'héritage permet au développeur de faire hériter des classes d'une classe parente. La classe enfant, parfois appelée classe dérivée, hérite alors des attributs et des méthodes de la classe parente, mais peut également lui ajouter de nouvelles caractéristiques, dans un but de spécialisation. Par exemple, dans notre application, on peut trouver une classe Bean dont héritent toutes nos entités. Cette classe Bean contient un attribut id de type Integer, ainsi que deux méthodes: Integer getId() et setId(). Les classes qui héritent de Bean posséderont donc toutes cet attribut et ces méthodes, sans avoir besoin de les déclarer à nouveau, mais lui ajoutent de nombreuses propriétés qui les différencient de leur parent.
Contrairement à d'autres langages comme le C++, en Java il n'est possible de n'hériter que d'une classe à la fois.

1.3.2.3. Polymorphisme

Le mot polymorphisme signifie "qui peut prendre plusieurs formes". En POO, cela signifie que des fonctions qui portent le même nom peuvent agir de manière différente. On distingue deux formes de polymorphisme: la surcharge et la redéfinition.

1.3.2.3.1. Surcharge

La surcharge (overloading, en anglais) permet de définir une fonction plusieurs fois, en faisant varier le nombre et/ou le type de ses arguments, et d'adapter son comportement en fonction de la version qui est appelée. C'est le compilateur qui se chargera d'appeler la bonne fonction en fonction des arguments passés.

Par exemple, la classe java.lang.String possède une méthode indexOf qui permet de retourner la position au sein d'une chaîne d'un argument spécifique. Cette méthode peut être appelée de différentes manières:

- int indexOf(int),
- int indexOf(int, int),
- int indexOf(String),
- int indexOf(String, int).

Ces méthodes portent le le même nom, ont le même type de retour, ont la même intention, mais ne traitent pas les mêmes données de la même manière.

1.3.2.3.2. Redéfinition

La redéfinition (overriding) est fortement liée à l'héritage. Cette notion spécifie qu'une classe qui hérite d'une classe de base peut redéfinir les méthodes dont il a hérité, de manière à se comporter d'une manière qui lui est propre.

Pour reprendre à nouveau un exemple du JDK (Java Development Kit), en Java, tous les objets héritent de la superclasse java.lang.Object. Cette classe hérite de la méthode toString, qui retourne une réprésentation de l'objet sous forme de chaîne de caractère. Grâce à l'héritage, il est donc possible d'appeler cette méthode sur n'importe quel objet.

Cependant, cette méthode en tant que telle ne fait que retourner le nom de la classe de notre objet, suivi de son adresse mémoire. On pourrait souhaiter obtenir un résultat plus adapté à notre classe, par exemple, dans le cas d'un User, afficher son id et son nom. Nous avons alors la possiblité de redéfinir cette méthode: il suffit de créer une méthode à la signature identique dans la classe dérivée et de retourner le résultat souhaité et c'est cette méthode qui sera appelée au lieu de la méthode parente. Dans le cas où l'on n'aurait pas redéfini toString, c'est toujours la version de la classe parente (Object, dans notre cas) qui sera appelée.

1.4. Java EE

Java EE[1] (Java Enterprise Edition, anciennement J2EE) est une plate-forme qui étend la la version standard de Java, JSE (Java Standard Edition). “Java EE est un ensemble de spécifications pour les applications d’entreprise ; il peut donc être considéré comme une extension de Java SE destinée à faciliter le développement d’applications distribuées, robustes, puissantes et à haute disponibilité.”[2]

Cette couche EE se compose d’un ensemble de technologies et d’APIs définis au moyen de spécifications. Ces technologies, qui répondent à de nombreuses problématiques, seront détaillées dans les sections relatives à la couche dans laquelle elles sont utilisées dans notre application. On citera notamment:

- les Entreprise JavaBeans,
- Java Persistence API,
- Java Server Faces,
- etc.

[1] On recontre parfois l’appellation JEE, mais l’appellation préconisée est bien Java EE
https://java.net/projects/javaee-spec/pages/JEE

[2] Antonio Goncalves, Java EE 6 et Glassfish, 2010, p. 6.

1.5. Maven

Nous avons choisi d'utiliser Maven dans le cadre de ce projet. Maven est un outil de build open-source, qui apporte une réponse à certaines problèmatiques rencontrées fréquemment:

- gestion du cycle de vie complet de l'application: compilation, test, packaging, déploiement, ...
- gestion des dépendances et des dépendances transitives,
- gestion de profils permettant de définir des paramètres utilisés à la compilation.
-

Au travers d'un (ou plusieurs) fichiers pom.xml, Maven permet de traiter tous ces aspects.

1.5.1. Cycle de vie de l'application

Maven définit plusieurs étapes précises du cycle de vie du projet, par exemple:

- compile,
- test,
- deploy.

A l'aide de goals associés à ces étapes, Maven exécute les tâches qu'il trouve dans le pom.xml au moyen de différents plugins. On peut ensuite exécuter ces goals en ligne de commande (ou à l'aide des plugins disponibles pour les différents IDE). Par exmple, une commande comme `mvn clean install` permet, entre autres, de supprimer les fichiers précédemment générés, de compiler le projet, de le tester, et enfin de l'installer dans le repository local.

1.5.2. Dépendances et dépendances transitives

Notre projet fait appel à plusieurs dépendances, comme par exemple le jar de Primefaces (primefaces-4.0.jar). Ce jar doit être ajouté au classpath de l'application pour pouvoir être utilisé; si plusieurs développeurs travaillent sur le même projet, il est alors nécessaire que chaque collaborateur l'ajoute manuellement, ou encore de l'héberger sur le logiciel de gestion de version afin de le récupérer lors d'un update. De plus, certaines dépendances font elles-même appel à des dépendances, c'est ce qu'on appelle les dépendances transitives. Par exemple, la librairie de test Junit (junit-4.11.jar) a besoin de hamcrest-core-1.3.jar pour fonctionner. Celui-ci pourrait à son tour lui aussi faire appel aux frameworks/librairies x, y, z… Il peut alors devenir fastidieux de récupérer les jars dans leurs bonnes version puis de les ajouter, et les héberger coûte un espace disque qui peut devenir important si on fait appel à de nombreuses librairies.

Maven propose une solution pour remédier à ce problème: en définissant ces dépendances dans le pom.xml du projet, le plugin de gestion de dépendances ira les chercher lors du build, soit dans un repository distant, soit dans le repository local (qui contient toutes les dépendances déjà téléchargées précédemment). De plus, Maven résoud aussi automatiquement les dépendances transitives, sans avoir besoin de les spécifier dans le pom.xml.

1.5.3. Environnements et paramètres

On souhaite parfois utiliser différents paramètres, effectuer différentes actions lors de la compilation. Maven propose à cet effet de créer des profils: à l'aide d'un paramètre spécifié en même temps que le goal que l'on souhaite exécuter, des paramètres ou des actions spéciales définis dans le pom.xml seront utilisés. Par exemple, nous avons défini un profil "deploy-to-local". Ces profil permet, grâce au plugin maven-glassfish-plugin. de déployer l'application sur notre serveur local à l'aide de propriétés (host, mot de passe, etc.) définies dans un pom.xml. Nous aurions ainsi pu créer un ou plusieurs autres profils contenant des propriétés différentes et qui nous auraient permis de déployer sur des serveurs différents.

1.5.4. Projet multi-module

Maven va également nous aider à respecter l'approche de separation of concerns dont nous avons parlé au chapître "Architecture". En effet, nous avons choisi de répartir notre code dans quatre modules distincts:

- le module domain,
- le module ejb,
- le module war.

On utilisera aussi un module ear qui servira à coordonner les différents module au sein d'un fichier ear (enterprise archive), destiné à être déployé sur le serveur.

1.5.4.1. Module domain

Dans ce module, on retrouve les entités de notre domaine, c'est-à-dire les classes qui représentent les tables de la base de données. C'est dans ce module que l'on retrouvera le fichier de configuration persistence.xml qui spécifie les paramètres de connexion à la base de données. Ce module représente bien entendu la couche de données; il est packagé sous forme de jar (java archive). Ce module est indépendant: il ne dépend ni du module ejb, ni du module war.

1.5.4.2. Module ejb

C'est ici que nous retrouverons les EJB, c'est-à-dire la couche métier de l'application. Comme ce module manipule les entités, il a bien entendu une dépendance avec le module domain. Au point de vue de Maven, il s'agit d'un module de type ejb, mais au niveau Java, c'est aussi un jar.

1.5.4.3. Module war

Comme son nom l'indique, c'est dans ce module que se trouve les resources nécessaires à l'application web, c'est-à-dire la couche présentation: pages html, Facelets, images, managed beans. Ce module dépend à la fois du module domain et du module ejb, et il est packagé sous forme de war (web archive).

1.5.4.4. Module ws

Ce module contient les classes relative au web service qui expose notre API. Ce web service REST étant accessible au travers du protocole HTTP, il est lui aussi packagé sous forme de war, et il dépend du module ejb ainsi que du module ejb étant donné qu'il manipule les données issues de notre base de données.

Chacun de ces modules possède un pom.xml qui lui est propre, ce qui permet de bien séparer les dépendances nécessaires à chaque module: ejb ne dépend pas de Primefaces, war ne dépend pas de javax.mail.

Maven permet de gérer la compilation de manière centralisée grâce à un pom parent qui définit l'ensemble de ces modules comme faisant partie du même projet. L'ordre dans lequel les différents modules sont compilés est défini de manière logique en fonction des dépendances des modules les uns avec les autres.

Par la suite et si le besoin se présentait, il serait aisé d'ajouter un ou plusieurs modules aux responsabilités distinctes et faisant appel aux modules qui lui sont strictement nécessaires.

1.6. Serveur applicatif

1.6.1. Rôle d'un serveur applicatif

Un serveur applicatif sert à exécuter le code contenu dans notre application. A l'instar du serveur web qui sert des ressources à ses clients à l'aide d'une URL (pages xhtml, images, etc.), le serveur applicatif renvoie une réponse (response) à une requête (request) d'un client, mais en analysant d'éventuel paramètres qui lui seraient transmis, et en adaptant sa réponse à ces paramètres après exécution ou non de code Java.

Plus exactement, un serveur Java est un conteneur qui implémente les spécifications de Java EE. Ce conteneur offre un environnement d'exécution, qui sera en charge de différentes responsabilités:

- gestion du cycle de vie des entités,
- gestion des transactions,
- injection des dépendances,
- etc.

Ces services fournis par le conteneur permettent de décharger le développeur de certains aspects techniques et facilitent le développement d'applications portables.

1.6.2. Glassfish

Nous avons choisi Glassfish comme serveur applicatif pour deux raisons principales.

Tout d'abord, il s'agit de l'implémentation de référence de Java EE, développée par Oracle. Il s'agit de plus d'un produit open-source.

Ensuite, nous avons souhaité développer notre application dans la version la plus récente de Java, la version 7, et à l'heure où nous écrivons ces lignes, il s'agit d'un des rares serveurs "Java 7 compliant", c'est-à-dire qui implémente Java EE 7[3].

[3]Avec Wildfly 8 et JEUS : http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

2. La couche de données

2.1. Base de données

2.1.1. Postgresql

Nous utilisons une base de données de type PostgreSQL, dans sa version 9.2. Il s'agit d'une base de données relationnelle open-source, qui respecte les standards de la norme SQL, ainsi que les principes ACID:

- atomicité: les transactions sont effectuées entièrement ou pas du tout,
- cohérence: des contraintes permettent de conserver à tout moment des données cohérentes,
- isolation: une transaction travaille dans un contexte qui lui est propre à un instant i, elle n'impacte pas d'autres transactions durant ses opérations,
- durabilité: les résultats d'une transaction doivent pouvoir survivre à une interruption, physique par exemple dans le cas d'une panne de courant.

2.1.2. Modélisation

Nous avons modélisé les tables de la base de données en se basant sur les objets du domaine qui ont été définis dans l'analyse; ces objets ont été enrichis au fur et à mesure du développement avec des concepts de la couche applicative. On trouvera par exemple une table menuitem qui définit l'ordre dans lequel sont affichés les onglets du menu utilisateur, et qui permet de modifier cet ordre dans l'interface d'administration.

2.1.3. Schéma

Voir annexes.

2.2. Java Persistence API

Java Persistence API (JPA) propose au développeur un ensemble d'outils qui facilitent la manipulation de données issues d'une base de données relationnelle. Ces outils permettent d'établir une correspondance entre les tables de la base de données et des objets Java; c'est ce qu'on appelle un ORM (object-relational mapping). Chaque table sera associée à un objet, et chaque attribut de cet objet à une colonne de la table. Chaque instance de l'objet représente une ligne de la table.

Ce mapping qui facilite la gestion de la persistance permet en outre de ne plus utiliser directement JDBC et donc beaucoup d'éviter tout le "boilerplate code", le code répétitif utilisé pour ouvrir et fermer les connection, exécuter les requêtes et gérer les erreurs à l'aide de plusieurs blocs try-catch.

2.2.1. Configuration par annotations

La correspondance entre les tables et les objets est établie dans notre application au moyen des annotations proposées par JPA (il aurait également été possible d'utiliser un fichier de configuration au format xml). Ces objets sont appelés des entités, qui ne sont que de simples POJOs (plain old java object) annotés.

Néanmoins, pour qu'un objet Java soit reconnu par JPA comme une entité, il doit se conformer à certaines obligations. On notera entre autres qu'il doit être annoté avec @javax.persistence.Entity, et qu'il doit posséder un constructeur public ou protected, sans argument. Il doit également posséder un attribut qui correspond à sa clé primaire, et qui sera annoté avec @javax.persistence.Id.

Les attributs de l'objet, qui correspondent à des colonnes dans la table, peuvent être annotés avec @Column, mais par défaut, tout attribut non annoté avec @Transient correspond à une colonne. Ces attributs doivent être privés et accompagnés d'un couple getter/setter qui permet de les manipuler. Ce couple doit être nommé de manière à respecter les conventions JavaBeans[4].

Si on reprend notre classe be.isl.desamouryv.sociall.domain.User, nous remarquons qu'elle correspond à ces exigences: elle possède l'annotation @Entity ainsi qu'un constructeur sans argument. Chaque propriété persistente est privée, et possède un getter et un setter. Il en va de même pour chacune de nos entités (module domain, package be.isl.desamouryv.sociall.domain).

```
@Entity
class User {
```

[4] Code Conventions for the Java ™ Programming Language, http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

```
        @Id
        private Integer id;
        private String firstName;

        public User(){}

        public String getFirstName(){
                return this.firstName;
        }
        public void setFirstName(String firstName){
                this.firstName = firstName;
        }
        // ...
}
```

2.2.2. Relations et cardinalités

JPA, en tant qu'ORM, permet également de définir les relations qui existent entre les différentes tables. Dans le monde des bases de données, ces relations sont définies à l'aide de clés étrangères (foreign keys). JPA permet de définir différentes cardinalités entre les relations, toujours à l'aide d'annotations:

- @OneToOne: relation de un à un,
- @OneToMany: relation un à n,
- @ManyToOne: relation n à un,
- @ManyToMany: relation n à n.

Ces relations peuvent être unidirectionnelles ou bidirectionnelles.

Les annotations @OneToMany et @ManyToMany doivent être placées sur un attribut qui implémente une des interfaces de l'API Collection: java.util.Collection, java.util.Set, java.util.List ou java.util.Map. Le type des éléments de la collection sera le type qui correspond à la table référencée dans la clé étrangère.

Quant aux annotations @OneToOne et @ManyToOne, elles doivent être placée sur un attribut du type qui correspond de la même manière à la table référencée.

Ainsi, la classe be.isl.desamouryv.sociall.domain.User possède différents attributs qui correspondent aux relations que l'on retrouve dans la base de données:

- une et une seule image de profil: @OneToOne private Image profilePic
- un ou plusieurs roles avec une cardinalité n-n: @ManyToMany private Set<Role> roles,
- un ou plusieurs avis avec une cardinalité n-1: @OneToMany private List<Review> reviews (l'entité Review comprend la relation inverse @ManyToOne private User author).

2.2.3. Gestion de l'héritage

Comme nous l'avons vu dans la section relative au langage Java, il s'agit d'un langage orienté objet, or la base de données, quant à elle, est conçue selon un modèle relationnel. Ces deux conceptions sont parfois difficiles à concilier, notamment en ce qui concerne la représentation de l'héritage[5], or la hiérarchie de nos classes implique une relation d'héritage entre la classe Artifact et les classes Event, Place, Product et Recipe.

JPA propose plusieurs stratégies pour répondre à ce problème, que l'on spécifie au niveau de l'entité à l'aide de l'annotation @Inheritance et d'un attribut qui indique la stratégie.

Tout d'abord, InheritanceType.SINGLE_TABLE propose l'utilisation d'une seule table par hiérarchie de classe, avec une colonne discriminante qui permet de distinguer le type de classe. Cette solution a l'avantage d'être simple à comprendre, néanmoins elle impose que les colonnes propres aux classes dérivées soient toutes nullables pour permettre l'insertion de tous les types d'entités. De plus, chaque nouvelle entité ajoutée dans cettte hiérachie implique la modification de cette table (ajout de nouvelles colonnes, index, etc.).

Ensuite, il est également possible avec InheritanceType.TABLE_PER_CLASS de demander l'utilisation d'une table par classe concrète. Les inconvénients de cette stratégie, outre la redondance des colonnes et donc la dénormalisation de la base de données, sont que les requêtes sur la classe parente impliquent l'utilisation d'unions ou de requêtes différentes pour chaque type. De plus, le support de cette stratégie pour JPA 2 est facultatif; tous les providers ne sont pas obligés de l'implémenter[6], ce qui rend cette solution non portable.

Enfin, la dernière stratégie, InheritanceType.JOINED, celle que nous avons choisie, permet l'utilisation d'une table qui représente la classe parent (Artifact, pour rappel). C'est cette table qui contient les propriétés partagées par les enfants comme title, images, etc. Les entités qui héritent de cette classe possèdent chacune une table qui leur est propre; la clé primaire de ces tables est en fait une foreign key vers la clé primaire de la table parent.

Cette stratégie peut impacter négativement les performances, surtout si la hiérachie de classes est étendue. En effet, elle implique l'utilisation d'un ou plusieurs JOIN au sein des requêtes.

2.2.4. Validation des données

Il est possible d'annoter les propriétés d'une entité avec les annotations de l'API Bean Validation (JSR-303). Ces annotations permettent de définir des contraintes qui empêchent l'insertion de données erronnées dans la base de données.
Par exemple, dans la classe be.isl.desamouryv.sociall.domain.User, lastName est annoté avec @javax.validation.constraints.NotNull, qui empêche l'insertion de valeurs nulles. Ces attributs sont également annotés avec @Size(min=2, max=255), qui spécifie les longueurs minimales et

[5] On parle d'impedance mismatch, http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
[6]Par exemple, Eclipselink, le fournisseur de persistance par défaut de Glassfish et celui que nous utilisons dans cette application, ne supporte pas cette stratégie.

maximales que cet attribut peut prendre. Bien entendu, on essayera de répliquer les contraintes déjà présentes dans la base de données, ou de les restreindre si nécessaire, et non les étendre, ce qui n'aurait aucun sens vu qu'une erreur serait soulevée lors de l'insertion.

2.2.5. Manipulation des données avec l'EntityManager

JPA propose un gestionnaire d'entités qui permet de manipuler ces dernières: l'EntityManager.

2.2.5.1. Contexte de persistance

Dans un environnement Java EE, on obtient le gestionnaire d'entités grâce à l'injection de dépendances: annoter l'attribut avec @javax.persistence.PersistenceContext nous permet de récupérer un EntityManager lié à un PersistenceContext. Ce contexte de persistance est l'ensemble des entités gérées par l'EntityManager à un moment donné.

Il est également possible d'obtenir un EntityManager dans un environnement Java SE, par l'intermédiaire de l'objet EntityManagerFactory, mais dans ce cas, c'est le développeur qui est responsable de la gestion des transactions, contrairement à l'environnement EE où transaction et contexte de persistance sont liés.

Dans notre application, l'EntityManager est injecté dans chaque EJB (module ejb):

```
@PersistenceContext(unitName = "FOODISH_PU")
private EntityManager em;
```

2.2.5.2. Fonctions de l'Entitymanager

Le gestionnaire d'entités offre des fonctions qui correspondent aux opérations couramment effectuées sur une base de données, ce que l'on appelle le CRUD: create/read/update/delete, en français création/lecture/mise à jour/suppression.

- persist(Object entity): persiste une instance dans la base de données,
- <T> T find(Class<T> entityClass, Object primaryKey): recherche une entité sur base de sa clé primaire,
- <T> T merge(T entity): met à jour une entité,
- remove(Object entity): supprime une entité.

2.2.6. Le langage JPQL

Outre les fonctionnalités de CRUD, l'EntityManager permet également d'effectuer des requête dans la base de données, et ce à l'aide de JPQL (Java Persistence Query Language). JPQL est le langage de requête offert par JPA. Il permet d'effectuer des requêtes dans un langage se rapprochant du SQL, mais avec une approche objet.

C'est en JPQL que sont exprimées les requêtes nommées (named queries) de notre application. On peut par exemple trouver dans be.isl.desamouryv.sociall.domain.User la requête User.findByLastName.

```
SELECT i FROM Ingredient i WHERE LOWER(i.name) LIKE :query
```

On peut noter la similitude avec le SQL dans l'usage de SELECT… WHERE… = …, ainsi que l'utilisation d'une fonction LOWER(). On remarque aussi que la requête retourne un objet Java User et non un ensemble de champs.

Ces requêtes JPQL peuvent être exprimées de manière dynamique ou de manière statique.

2.2.6.1. Requêtes dynamiques

Ces requêtes sont créées at runtime par l'application, grâce à la méthode de l'entity manager createQuery(String jpqlQuery). Elles sont utilisées si la forme de la requête peut varier en fonction du contexte.

On passe des paramètres à ces requêtes grâce à des paramètres nommés:

```
SELECT u FROM User u WHERE LOWER(u.email) like LOWER(:email)
```

Nous utilisons ces named parameters pour trois raisons:

1. pour éviter une opération de concaténation superflue,
2. pour empêcher une injection de SQL par un utilisateur malveillant[7],
3. pour permetter au provider de persistance (EclipseLink) d'éventuellement mettre la query en cache.

2.2.6.2. Requêtes statiques

Ces requêtes sont définies au niveau de l'entité grâce à l'annotation @javax.persistence.NamedQueries, puis exécutées avec EntityManager#createNamedQuery(String name). Ces requêtes nommées offrent deux avantages:

1. elles sont centralisées en un seul endroit,
2. elles sont validées par le compilateur lors de la compilation, ce qui permet de détecter d'éventuelles erreurs plus tôt[8].

De plus, les Named Queries sont mises en cache au moment du déploiement. Cela signifie que la traduction JPQL-SQL ne s'effectue qu'une seule fois, ce qui est meilleur pour la performance de l'application.

---

[7] software-security.sans.org/developer-how-to/fix-sql-injection-in-java-persistence-api-jpa

[8] "Fail-fast", Wikipédia http://en.wikipedia.org/wiki/Fail-fast

2.2.7. Criteria

L'API Criteria de JPE est une alternative à JPQL: elle permet d'effectuer des requêtes sur la base de données, mais a l'avantage d'être "type-safe"[9].

Criteria permet de plus d'utiliser un méta-modèle des entités. Ce métamodèle est un ensemble d'objets qui représentent les objets du domaine, et qui permettent d'accroître encore la sûreté du typage en évitant de référencer les noms d'attributs par des chaînes de caractères.

Par exemple, la classe be.isl.desamouryv.sociall.domain.Artifact possède son métamodèle "miroir", la classe abstraite be.isl.desamouryv.sociall.domain.Artifact_, qui nous permettra de faire référence, par exemple, à l'attribut title de Artifact de la manière suivante::

```
Root<Artifact> root = criteria.from(Artifact.class);
root.get(Artifact_.title); // plus sûr que root.get("title")
```

Ces objets peuvent être écrits par le développeur, cependant il s'agit d'une tâche répétitive, qu'il faut adapter à chaque modification d'une entité. C'est la raison pour laquelle nous avons confié cette tâche à un plugin Maven (maven-processor-plugin) qui fera appel au processeur de EclipseLink, org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor.

Criteria est une API particulièrement adaptée pour la création de requêtes dynamiques, qui varient en fonction des paramètres passés. Elle empêche de plus la création de requêtes syntaxiquement fausses, et ces requêtes sont de surcroît vérifiées at compile time (ce qui permet l'auto-complétion par l'IDE). Néanmoins, sa structure est plutôt verbeuse et parfois difficile à lire, et peut dérouter le développeur habitué au JPQL/SQL. C'est pourquoi nous avons préféré utiliser cette API avec parcimonie.

[9] "Sûreté du typage", Wikipédia http://fr.wikipedia.org/wiki/S%C3%BBret%C3%A9_du_typage

3. La couche métier

3.1. Enterprise Java Beans

Les EJBs (Enterprise Java Beans) sont des composants exécutés dans un conteneur d'EJBs. Ce conteneur est un environnement d'exécution (runtime environment) qui fait partie d'un serveur d'applications; il est impossible pour un EJB de s'exécuter hors de cet environnement. Le conteneur d'EJBs fournit de nombreux services, comme par exemple la gestion des transactions ou la sécurité, ce qui simplifie la tâche du développeur en le laissant se concentrer sur la logique propre à son application (voir section *Glassfish*).

Les EJBs, quant à eux, contiennent la logique métier de l'application. Ce sont eux qui seront chargés par exemple de l'accès à la base de données; ces traitements ne s'exécutent que côté serveur, déchargeant par là le client des aspects métier.

Les EJBs ont également la particularité d'être portables, ce qui signifie que l'on peut théoriquement les déployer sur différents serveurs certifés Java EE sans nécessiter d'adaptation.

Les EJBs peuvent être de type Session ou de type Message. Ces derniers utilisent l'API JMS (Java Message Service), qui n'est pas utilisée dans notre application. Nous utiliserons des beans de Session, qui se trouveront dans le module nommé ejb.

3.1.1. Session Beans

Les Session beans peuvent être de trois types: stateful, stateless ou singleton. Pour les distinguer, on les annote avec leurs annotations respectives: @Stateful, @Stateless, @Singleton.

3.1.1.1. Stateful

Les session beans de type stateful conservent leur état durant une session avec un et un seul client. Cela signifie qu'il peut posséder des propriétés dont les valeurs seront conservées d'un appel à l'autre par le client. Pour illustrer ce type d'EJB, on cite souvent l'exemple d'un panier client sur un site d'e-commerce.

3.1.1.2. Stateless

Les session beans de type stateless ne conservent pas leur état d'une requête à l'autre (littéralement, stateless signifie "sans état"). Comme ces beans ne contiennent aucun attribut propre à un client, ils peuvent être partagés entre de multiples clients. C'est la raison pour laquelle on les préfère dans le cas d'une application qui sera utilisée par un grand nombre d'utilisateurs. En effet, leur coût en terme de création, de destruction et de maintien par le serveur est moindre car ce dernier maintient un pool d'instances partagées. Le cahier des charges de notre application spécifiant qu'elle sera amenée à être utilisée par un grand nombre de clients, ils constituent par conséquent un choix logique.

Il est également à noter qu'un bean Stateless peut implémenter un web service, contrairement à un bean stateful.

3.1.1.3. Singleton

Un bean de type singleton n'est instancié qu'une seule fois par application (cf. le pattern Singleton). Comme les beans stateful, il conserve son état.

Dans le cadre de notre application, nous avons fait le choix de conserver les données de session dans la couche web de notre application, à l'aide d'un backing bean de scope session qui utilise la session HTTP. Nos EJBs seront donc exclusivement de type stateless.

3.1.2. Gestion des transactions

Les transactions sont cruciales lors de toute interaction avec une base de données. Une transaction peut être vue comme un ensemble d'opérations indivisible. Cet ensemble doit soit être mené à bien dans son intégralité, soit ne pas être exécuté du tout. Sans une gestion appropriée de ces transactions, l'intégrité des données est menacée.

Dans une application Java EE, les transactions peuvent être gérées par le conteneur d'EJB (on parle alors de Container-Managed Transactions, ou CMTs) ou par l'EJB (Bean-Managed Transactions, BMTs).

Si le développeur choisit de gérer les transactions au niveau applicatif (BMTs), c'est lui qui sera en charge de manuellement appeler les opérations qui y sont relatives (begin, commit, rollback).

Nous avons choisi de laisser le conteneur gérer les transactions. Suivant le principe de "convention over configuration", nous n'annotons pas nos EJBs de manière spécifique, ce qui revient au même que de les annoter au niveau de la classe avec la valeur par défaut, à savoir:

```
@TransactionManagement(TransactionManagementType.CONTAINER)
```

Typiquement, une transaction est démarrée au début de chaque méthode d'un EJB, et est terminée en fin de méthode. Dans le cas où une transaction serait déjà en cours (la méthode x a été appelée dans une méthode y qui a elle-même débuté une transaction), l'appel à cette méthode serait alors englobé dans la transaction. En cas d'erreur, toutes les opérations qui en font partie seraient annulées (rollback). Ce comportement par défaut correspond à l'annotation:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
```

3.1.3. Le pattern Façade

Le design pattern Façade cherche à offrir à l'utilisateur une interface simplifiée, qui cache la complexité des opérations effectuées par un sous-système Cette interface permet aussi de réduire le couplage entre l'utilisateur et le traitement.

Par exemple, les opérations relative à l'inscription et à la gestion du compte d'un membre font appel à de nombreux sous-processus: création d'un token à usage unique, création d'un mot de passe, upload d'une photo de profil, envoi de mails de confirmation… Ce qui résulte en l'utilisation de nombreux EJBs différents, car de nombreux domaines de responsabilité sont partagés. C'est la raison pour laquelle nous avons implémenté le pattern Façade avec le bean be.isl.desamouryv.sociall.service.AccountService.

Ce bean propose des méthodes simples au client (l'application web), comme AccountService#signinUser(User user) qui se charge, dans l'ordre, de:

1. générer un token grâce à l'EJB TokenGenerator,
2. récupérer le role "MEMBER" grâce à l'EJB RoleFacade,
3. enregistrer dans la base de données le nouvel utilisateur et le record correspondant dans la table relative à l'activation du compte,
4. envoyer un mail de confirmation à l'utilisateur.

L'ensemble de ces opérations constitue une transaction, qui peut être annulée (rollbacked) si nécessaire.

Il ne faut pas oublier de préciser que le pattern Façade n'empêche pas l'utilisation du sous-système. Il est toujours possible pour le client d'utiliser directement les méthodes des EJBs.
.

3.2. Injection de dépendances

L'injection de dépendances est assurée en Java EE par l'API Context and Dependency Injection for Java EE (CDI).

CDI offre au développeur la possiblité d'injecter automatiquement les dépendances requises au sein de ses objets, le libérant de l'instanciation et assurant par là un faible couplage entre les classes. On essayera de favoriser au maximum ce faible couplage pour différentes raisons, entre autres:

- minimiser la dépendance entre les objets,
- faciliter la réutilisation des objets,
- faciliter le test des classes.

Une fois injectées, le cycle de vie de ces resources est assuré par CDI.

4. La couche de présentation

4.1. Java Server Faces

Java Server Faces (JSF) est le framework proposé par la plate-forme Java EE pour le développement d'applications web riches. On peut discerner deux grands aspects de ce framework:

1. Une API qui représente des composants, permet de les valider, d'établir des règles de navigation, etc.
2. Des librairies de tags qui servent à déclarer des composants dans une page web, et à les lier à l'objet qui leur correspond côté serveur.

Nous avons choisi d'utiliser JSF comme framework pour l'interface utilisateur de notre application pour plusieurs raisons. Tout d'abord, il s'agit d'un composant standard de la plate-forme Java EE, il ne nécessite l'ajout d'aucune librairie ni framework pour fonctionner. Ensuite, il s'agit d'une technologie mature, bien documentée et à la large communauté. Enfin, des projets comme Primefaces permettent de bénéficier d'une large panoplie de composants qui nous permettront d'enrichir notre application.

4.1.1. Modèle-Vue-Contrôleur

JSF offre au développeur un modèle de programmation qui lui permet de séparer clairement la logique métier de la présentation: le design pattern Model-View-Controller (MVC).

Ce patron de conception définit trois composants.

- Le modèle: ce sont les données que l'on traite, leur modélisation. Dans notre cas, ce sont les entités JPA.
- La vue: la vue est la manière dont on va présenter les données; c'est avec la vue que l'utilisateur interagit. Ce sont nos pages web (les Facelets).
- Le contrôleur: c'est lui qui contient la logique métier, et c'est lui qui est chargé d'appliquer les actions de l'utilisateur, et de mettre le modèle à jour en conséquence. C'est ici qu'interviennent les managed beans.

En séparant clairement la vue du modèle, le modèle MVC permet de réduire les dépendances entre eux. On construit de cette manière un design plus flexible.

On peut remarquer que le modèle MVC reflète à une échelle différente l'architecture n-tier mentionnée à la section *Architecture*. Une architecture n-tier implique généralement que les couches soient séparées par le réseau, tandis que dans le modèle MVC, ce sont différentes portions du code qui délimitent les responsabilités.

4.1.2. Facelets

Facelets est la technologie utilisée par JSF pour construire les vues. Quelques caractéristiques:

- pages écrites en xhtml,
- utilisation de bibliothèques de tags,

- support de l'expression language,
- existence d'un système de templating.

4.1.2.1. Bibliothèques de tags

Facelets utilise des bibliothèques de tags pour ajouter des composants aux pages. Pour avoir accès à une bibliothèque spécifique, il faut ajouter son namespace (son espace de nom) à la balise html de la facelet::

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
```

En la déclarant de cette manière, nous aurons accès à ses éléments sous la forme préfixée ui:.

Parmi ces bibiliothèque, on peut citer:

- HTML: décrit les composants qui correspondent aux éléments HTML traditionnels, et qui font partie de la famille JSF UIComponent. Dans notre application, ce sont tous les champs de type outputText, ou les éléments form: <h:form/>, <h:outputText/>;
- Core: on y retrouve les composant utilisés pour les conversions, les validations, etc. <f:convertDateTime/> et <f:metadata/> en font partie;
- UI: il s'agit des composants utilisés pour l'interface utilisateur et le templating, comme <ui:include/> ou <ui:define/>.

JSF 2.2 introduit également le namespace *xmlns.jcp.org/jsf/passthrough* afin de permettre un support des nouveaux attributs HTML5 grâce aux attributs pass-through. Nous utilisons par exemple l'attribut placeholder dans la barre de recherche de l'application. Cet attribut permet d'afficher une valeur par défaut dans un champ de type input, qui disparaît lors de la frappe dans le champ:

```
<p:inputText pt:placeholder="le texte à afficher"/>
```

Ces nouveaux attributs permettent également d'effectuer, par exemple, des validations sur le contenu des champs, de l'auto-complétion, d'afficher des indicateurs de type barre de progression, etc. Cependant, ces attributs ne sont pas encore tous supportés par tous les navigateurs. Par exemple, placeholder n'est supporté par Internet Explorer qu'à partir de la version 10 du navigateur[10]. Le développeur doit donc faire attention à soit utiliser des attributs non existentiels pour l'application, soit être en possession d'un plan B dans le cas où la technologie ne serait pas supportée. Dans notre cas, nous avons choisi d'utiliser un fallback en Javascript qui sera appliqué si le navigateur est détecté par la bibliothèque Modernizr comme non compatible[11]. Nous aurions bien sûr pu utiliser le composant watermark de Primefaces qui

[10] http://caniuse.com/#feat=input-placeholder
[11] http://www.hagenburger.net/BLOG/HTML5-Input-Placeholder-Fix-With-jQuery.html

offre nativement les mêmes fonctionnalités; nous avons retenu la solution passthrough pour démontrer simplement ses fonctionnalités.

4.1.2.2. Composants composites

Il est aussi possible de créer sa propre bibliothèque de tags. JSF offre la possiblité de créer ses propres composants afin de pouvoir les réutiliser au travers de l'application. Il est possible de passer des paramètres à ces composants afin de les faire correspondre au contexte.

Dans notre cas, nous devons régulièrement afficher la photo de profil d'un utilisateur. Si l'utilisateur a uploadé une photo, c'est cette image qui sera utilisée, dans le cas contraire, c'est une image par défaut qui sera visible. Nous souhaitons également que cette image soit cliquable et redirige vers la page de profile de l'utilisateur. C'est pour répondre à ce cas d'utilisation que nous avons développé le composite profilePic. Ce composant prendra comme paramètre l'utilisateur dont nous souhaitons afficher la photo (un objet de type be.isl.desamouryv.sociall.domain.User qui fait partie de notre domaine), et en fonction de la valeur de cet objet (nul ou non), l'image adéquate sera affichée, de même que le lien.

Nous pourrons ensuite utiliser ce composant comme n'importe quel autre composant JSF, après avoir déclaré le namespace correspondant. On notera l'utilisation du mot "components" à la suite du namespace standard: il s'agit du nom du dossier dans lequel nous aurons choisi de placer les composants, au sein du dossier javax.faces.resources; le choix de ce nom de dossier est complètement arbitraire.

```
xmlns:sc="http://xmlns.jcp.org/jsf/composite/components"
```

```
<sc:profilePic user="#{userSession.user}"/>
```

4.1.2.3. Expression Language

Le langage d'expression (expression language, EL), permet de faire le lien entre les resources serveur et le client. Il offre plusieurs facilités.

Tout d'abord, on peut grâce à lui accéder aux attributs d'un managed bean, ou encore d'un objet présent dans le scope requête, session ou application. Par exemple, l'expression #{userSession.user} permet de faire référence à l'objet nommé user présent dans un managed bean nommé userSession.

Ensuite, il permet d'effectuer des opérations de base, comme des opérations arithmétiques ou de comparaison, à l'aide des opérateurs courants: +, -, and, or… Une expression comme rendered="#{postController.artifacts.size() > 0}" permet d'utiliser le booléen qui résulte de la comparaison comme valeur pour l'attribut rendered.

Enfin, on peut également appeler des méthodes publiques ou statiques, et depuis l'EL 2.2 (présent depuis JSF 2.0), on peut également passer des paramètres à ces méthodes, que ce soit pour exécuter un traitement ou pour obtenir une valeur de retour. C'est de cette manière que nous procédons pour afficher les première lignes d'un message, avec l'expression #{format.overview(message.text)} qui fait appel à la méthode Format#fomat(String string) qui retourne une chaîne de caractères.

4.1.2.4. Templates

JSF possède une librairie de tags qui facilite le templating des interfaces utilisateur. Pour créer un template, il faut créer une facelet qui contient des éléments ui:insert. Ces éléments seront définis de manière spécifique aux pages clientes de ce template.

Le template principal de notre application contient le header (barre de recherche, icône de profil, etc.), ainsi qu'un bloc "content", le bloc principal. De cette manière, nous pouvons nous contenter dans chaque client de ne définir que les éléments de ce bloc, ce qui nous permet de rester fidèles à une philosophie DRY[12].

4.1.3. Managed Beans

Les managed beans sont les composants serveur qui contiennent les propriétés et les fonctions utilisés par les composants présentés dans la Facelet. Dans notre application, ces objets se trouvent dans le package be.isl.desamouryv.sociall.ui: UserController, SearchController, UserSession, etc.

Les managed beans sont des POJOs annotés avec l'annotation @javax.inject.Named, ce qui les rend, eux et leurs membres, accessibles en EL. L'annotation peut prendre un attribut *name* qui sert à définir le nom du bean en EL; si rien n'est spécifié, c'est le nom de la classe (dont la première lettre aura été mise en minuscule[13]) qui sera utilisé.

4.1.4. Conversion

Dans la couche de présentation, les données sont affichées sous forme de chaînes de caractères, de manière à être lues et modifiées par l'utilisateur. Ces données correspondent cependant côté serveur à des objets Java. JSF propose une conversion automatique des données, dès lors que l'objet côté serveur correspond à un des types supportés par le composant.

Il est également possible de développer des convertisseurs adaptés à notre application, qui se chargeront par exemple de la conversion des entités de notre domaine. Pour cela, il faut créer une implémentation de javax.faces.convert.Converter. Cette interface nous oblige à redéfinir deux méthodes, getAsObject et getAtString.

---

[12] Do not Repeat Yourself, http://fr.wikipedia.org/wiki/Ne_vous_r%C3%A9p%C3%A9tez_pas

[13] On utilise ici le CamelCase tout comme dans les conventions Java, http://fr.wikipedia.org/wiki/CamelCase

- Object getAsObject(FacesContext context, UIComponent component, String value): sur base d'une chaîne de caractères passée en paramètre, cette méthode se charge de retourner l'objet qui lui correspond.
- String getAsString(FacesContext context, UIComponent component, Object value): à l'inverse, cette méthode retourne une représentation écrite de l'objet passé en paramètres.

Dans notre application, nous souhaiterions par exemple afficher une liste d'entités User au moyen du composant approprié:

```
<h:selectOneMenu value="#{bean.selectedUser}">
       <f:selectItems value="#{bean.users}"/>
</h:selectOneMenu>
```

JSF ne sait ni comment afficher un utilisateur, ni comment convertir sa forme telle qu'affichée à l'écran vers une entité. Si nous essayions d'utiliser ce composant tel quel dans notre application, nous obtiendrions une erreur dès que nous essayerions de soumettre les données au serveur::

Erreur de conversion lors de la définition de la valeur «be.isl.desamouryv.sociall.domain.User[userId=1]» pour «null Converter».

Ce message relativement explicite spécifie que le convertisseur pour ce type de données est null. Nous devons donc créer un convertisseur spécifique pour la classe User. Comme prévu dans le "contrat" de l'interface, ce convertisseur implémente les deux méthodes précédemment mentionnées:

- getAsString reçoit un objet de type User et retourne son id sous forme de String,
- getAsObjet reçoit cet id et, afin de retourner l'objet qui lui correspond, fait appel à l'EJB UserFacade pour aller le rechercher dans la base de données.

4.1.4.1. Limites de @FacesConverter

Cependant, cette approche nécessite la prise en compte de deux aspects..

Tout d'abord, il existe une limitation au sein de JSF 2.2 qui empêche d'injecter une resource dans une classe annotée avec @FacesConverter (l'annotation qui permet à JSF de détecter cette classe en tant que Converter). Or, cette annotation permet de définir un attribut forClass qui permet de définir que c'est ce converter qui sera automatiquement appelé pour une classe d'un certain type (forClass=User.class), sans avoir à le spéficier dans le composant. Il existe deux solutions à ce problème.

La première est de déclarer le convertisseur comme une ressource @Named CDI classique. Il devient alors possible d'injecter des ressources dans cette dernière, et le convertisseur est toujours accessible au travers de l'expression language; cependant, on perd alors le bénéfice

de l'attribut forClass. Chaque composant qui utilise nécessite ce convertisseur devra en déclarer son utilisation au moyen de l'attribut converterId.

La deuxième solution est d'aller récupérer manuellement l'EJB dans le contexte JNDI (Java Naming and Directory Interface) au lieu de l'injecter grâce à une annotation. Cette technique, qui permet de conserver l'annotation JSF @FacesConverter, est celle que nous allons préférer. C'est au travers de la classe utilitaire be.isl.desamouryv.sociall.ui.util.EJB que nous effectuerons la recherche dans le contexte. Cette classe possède une méthode générique qui, à l'aide d'un paramètre de type Class, retourne l'EJB correspondant. Cette méthode que nous déclarons statique pourra ainsi être réutilisée au besoin, indifféremment de l'EJB recherché.

4.1.4.2. Convertisseur générique

Au cours du développement, nous sommes rapidement amenés à constater que la création de tous les convertisseurs liés à nos entités JPA est répétitive et fait appel au même traitement: sur base de son id, récupérer l'entité dans la base de données grâce à la méthode find de l'EntityManager. De plus, toutes nos entités héritent de la classe abstraite be.isl.desamouryv.sociall.domain.Bean et implémentent sa méthode getId.

Nous avons donc décidé de créer un convertisseur générique (be.isl.desamouryv.sociall.ui.converter.BeanConverter) qui va implémenter les deux méthodes de l'interface et traiter toutes les entités comme des Bean. Ensuite, pour chaque entité, il nous suffira de créer un convertisseur qui étend cette classe abstraite et qui redéfinit la méthode getFacade afin d''utiliser le bon EJB.

De cette manière, nous facilitons la maintenance et réduisons tant que possible la duplication de code au travers de tous nos convertisseurs.

4.1.5. Ajax

L'ajax (Asynchronous Javascript and XML) représente un ensemble de techniques (HTML, Javascript, CSS, DOM, XML ou, fréquemment, JSON) qui permettent un échange de données entre une application web et le serveur de manière asynchrone. A l'inverse des requêtes HTTP traditionnelles qui nécessitent un rechargement complet de la page pour la mettre à jour avec la réponse du serveur, l'ajax permet de ne mettre à jour que la portion requise, et ce en arrière-plan, sans rendre l'application indisponible durant ce chargement (d'où l'expression asynchrone).

Dans une application web, ces échanges sont effectués par l'intermédiaire de l'objet XMLHttpRequest, en Javascript. Depuis JSF 2, les composants JSF proposent une gestion de l'ajax intégrée, de même que les composant Primefaces, ce qui évite au développeur d'avoir à maîtriser le Javascript pour le mettre en oeuvre.

4.1.6. Primefaces

Comme mentionné précédemment, il est possible d'enrichir le jeu de composants html à l'aide de blbiothèques supplémentaires. Il existe de ce fait plusieurs suites de composants, les plus populaires étant Richfaces, IceFaces et Primefaces.

C'est Primefaces que nous avons choisi d'utiliser, et ce pour les raisons suivantes:

- choix parmi de nombreux composants riches qui intègrent Ajax,
- grand nombre d'utilsateurs, par conséquent les composants sont largement testés et les ressources nombreuses (tutoriaux, forums, etc.),
- possibilité de développer facilement son propre thème à l'aide de l'outil JQuery Theme Roller.

Nous avons débuté le développement de l'application avec la version 4.0 de Primefaces, cependant, une nouvelle version majeure, la 5.0, suivie peu de temps après par la 5.1, a été publiée en mai 2014. La migration étant aisée, nous avons choisi d'utiliser cette version à partir de ce moment car elle corrige certains bugs et propose de nouvelles fonctionnalités, comme par exemple le fait de pouvoir ajouter un masque de saisie sur les champs de type date. Néanmoins, nous n'avons pas eu le temps d'adapter certaines fonctionnalités, comme le push; nous avons donc continué à utiliser l'API 4.0 qui est simplement dépréciée en 5.x.

5. Fonctionnalités spécifiques de l'application

5.1. Inscription

Nous renvoyons le lecteur vers les annexes pour le diagramme de cas d'utilisation concernant les inscriptions ainsi que les actions relatives au statut de membre.

5.1.1. Authentification

Lors de la conception des processus d'inscription et d'authentification d'un utilisateur, deux aspects sont à prendre en compte.

D'une part, le nombre grandissant d'applications et de sites engendre une multiplication pour l'utilisateur de noms d'utilisateurs, mais surtout de mots de passe à mémoriser. Pour ne pas avoir à en retenir des dizaines, la plupart du temps l'utilisateur réutilise le même mot de passe pour tous ses comptes, ce qui représente un risque de sécurité important si l'un des sites connait une faille. De plus, pour pouvoir retenir facilement ce mot de passe, l'utilisateur aura tendance à choisir un mot de passe peu sécurisé, ce qui représente un deuxième risque. Cet aspect se trouve de surcroît renforcé par l'utilisation des terminaux mobiles: en effet, il est encore plus compliqué d'entrer une longue combinaison de lettres, chiffres et symboles sur un clavier de taille réduite.

D'autre part, l'authentification d'un utilisateur et le stockage d'un mot de passe ne doivent pas être pris à la légère. Les mécanismes qu'ils emploient doivent être conçus de manière robuste, et testés extensivement. La complexité de ces processus est telle que selon Tim Bray, développeur chez Google, cette tâche devrait être confiée à un nombre restreint de fournisseurs, parmi lesquels on pourrait retrouver des universités ou des gouvernements[14].

5.1.2. OAuth 2.0 et OpenID Connect

OAuth et OpenID répondent à ces deux problématiques. OpendID, dans sa première version, confie le soin de l'authentification à un tiers, nommé Identity Provider[15] (IdP). Le plus connu de ces IdPs est sans aucun doute Google. Contrairement à ce protocole d'authentification, OAuth, quant à lui, était initialement un framework d'autorisation. Dans sa version 2, cependant, il est également utilisé pour l'authentification des utilisateurs, notamment grâce à l'extension OpenID Connect.

5.1.3. Inscription et authentification à l'aide des réseaux sociaux

Pour les raisons citées précédemment, nous avons choisi de donner la priorité à l'inscription sur notre site à l'aide d'OAuth 2.0, ce au travers des trois réseaux sociaux les plus utilisés en Europe[16]: Facebook, Google+ et Twitter.

[14]Fontana, John, "Quit peeing in identity pool, Google dev advocate says". http://www.zdnet.com/quit-peeing-in-identity-pool-google-dev-advocate-says-7000015979/

[15]Mozilla Developer Network, "Identity Provider Overview". https://developer.mozilla.org/en-US/Persona/Identity_Provider_Overview

[16]IAB Belgium, "Sept belges sur dix sont actifs sur les réseaux sociaux", pdf. http://www.iab-belgium.be/wp-content/uploads/2013/01/CP-IAB-Sept-belges-sur-dix-sont-actifs-sur-les-rseaux-

Pour pouvoir utiliser les API de ces réseaux, il est nécessaire d'enregistrer un compte auprès d'eux. Grâce à ce compte, nous recevrons une clé et un secret qui seront utilisés lors du processus d'autorisation et nous permettront de nous faire reconnaître auprès du site.
Pour ce faire, nous nous enregistrons donc aux adresses suivantes:

- https://developers.facebook.com,
- https://dev.twitter.com,
- https://developers.google.com.

Nous renvoyons le lecteur vers les annexes pour un diagramme de séquence représentant les processus de OAuth.

5.1.3.1. Callback URL et modification du fichier host

Parmi les informations requises par ces réseaux, nous devons indiquer une adresse de callback: c'est vers cette adresse que sera redirigée notre requête une fois qu'elle aura été approuvée par l'utilisateur. Cependant, la plupart des API n'acceptent pas que l'on indique une adresse de type http://localhost. Nous devons donc modifier le fichier host de notre machine de développement afin d'utiliser une adresse valide, ce qui nous permettra d'accéder à l'application à l'adresse http://foodish.com:8080.

```
127.0.0.1             foodish.com
```

5.1.3.2. SocialAuth

Nous avons choisi d'utiliser le framework SocialAuth[17] qui va nous permettre d'interagir avec les différentes API. Ce framework permet de manipuler ces API de manière unifiée, et ce en langage Java. Il nous permettra également par la suite d'accéder à des fonctionnalités avancées comme publication de statuts ou la récupération d'une liste de contacts.

Lors de l'analyse technique de la phase d'authentification via un tiers, nous nous sommes trouvés face à un problème: la sécurité de l'application étant conçue autour de JAAS, nous sommes dépendants de son implémentation par Glassfish. Cette dernière impose l'utilisation de realms et de classes dont les implémentations lui sont propres. Dans notre cas, nous utilisons un JDBCRealm dont le login module utilise un nom d'utilisateur et un mot de passe pour authentifier un utilisateur grâce à la base de données. Cependant, dans le cas d'une authentification via un réseau social, aucun transfert de mot de passe n'est évidemment présent.

Nous avons donc réfléchi à la manière d'authentifier les utilisateurs de deux manières différentes: d'un côté, de manière classique, à l'aide d'un couple username/password, d'un

sociaux.pdf

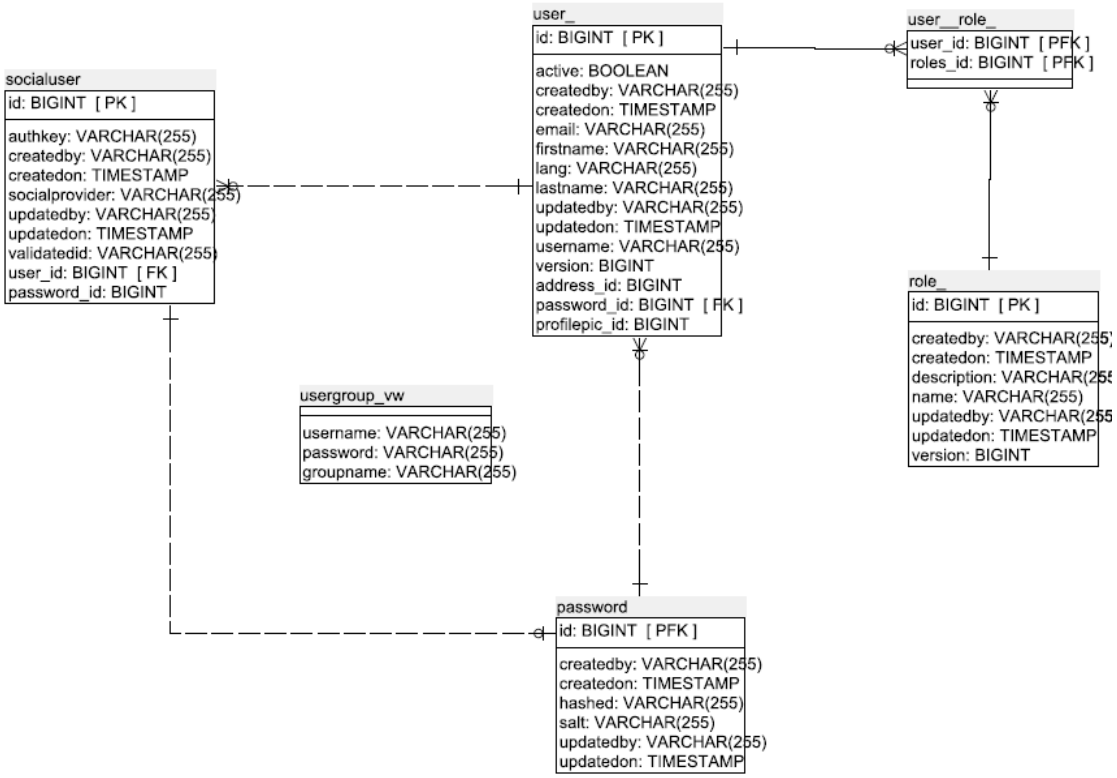
[17]"SocialAuth, Java Library for authentication, getting profile, contacts and updating status on Google, Yahoo, Facebook, Twitter, LinkedIn, and many more providers."
https://code.google.com/p/socialauth/

autre côté, uniquement avec nom d'utilisateur récupéré après authentifcation via OAuth. Des recherches nous ont appris qu'une solution était le développement d'un realm personnalisé, qui utiliserait deux login modules au lieu d'un. Néanmoins, cette option nécessite une connaissance relativement approfondie du développement Glassfish, et nous avons eu le sentiment qu'approfondir ce domaine sortait un peu du contexte de cette épreuve intégrée.

Idéalement, nous aurions aimé implémenter une solution basée sur JASPIC (Java Authentication Service Provider Interface for Containers). JASPIC est une spécification Java qui permet le développement de modules d'authentification indépendants du serveur et donc totalement portables. Une fois de plus cependant, faire le tour de cette spécification aurait pu constituer une épreuve intégrée à part entière.

C'est pourquoi nous avons fait le choix d'implémenter une solution quelque peu hybride, qui tire parti de JAAS sans demander de développement spécifique au serveur. Sur le schéma suivant, on peut voir que nous avons créé une classe SocialUser qui contient les informations relatives au compte d'un utilisateur sur un réseau spécifique. La table correspondante dans la base de données contient une foreign key qui permet d'identifier l'utilisateur.

Nous avons donc décidé de modifier la vue utilisée par Glassfish pour le login, de manière à utiliser la propriété validatedId retournée par OAuth comme nom d'utilisateur; quant au mot de passe, il s'agit d'une clé que nous générerons lors de la création du SocialUser et que nous assignons à la propriété authKey. Cette clé sera ensuite chiffrée en SHA-256 et enregistrée dans la table Password. De cette manière, il sera possible d'authentifier un utilisateur à l'aide du couple validatedId/authKey en lieu et place de email/password.



Nous avons ensuite modifié la vue SQL de manière à faire une union entre les données.

```
CREATE OR REPLACE VIEW usergroup_vw AS
SELECT u.email AS username, p.hashed AS password, r.name AS groupname
FROM user_ u
JOIN password p ON u.password_id = p.id
JOIN user__role_ ur ON u.id = ur.user_id
JOIN role_ r ON ur.roles_id = r.id
UNION
SELECT s.validatedid AS username, p.hashed AS password,
r.name AS groupname
FROM socialuser s
LEFT JOIN user_ u ON s.user_id = u.id
LEFT JOIN user__role_ ur ON u.id = ur.user_id
LEFT JOIN password p ON s.password_id = p.id
LEFT JOIN role_ r ON ur.roles_id = r.id;
```

L’enregistrement d’un utilsateur suivra plus ou moins le même processus, en exploitant les données reçues en callback: nom d’utilisateur, langue, etc.

5.1.4. Inscription classique

Malgré les réserves émises plus tôt, nous offrons néanmoins la possibilité à nos utilisateurs de s'inscrire de manière classique.

Pour empêcher l'inscription d'un utilisateur avec un email fictif ou qui ne lui appartient pas, nous procédons à une validation de l'adresse mail à l'aide d'un lien que nous lui envoyons par courier électronique à l'adresse renseignée. Ce processus nous permettra de surcroît d'être en possession de l'adresse de l'utilisateur, nécessaire par exemple dans le processus de réinitialisation du mot de passe. L'utilisateur devra cliquer sur ce lien endéans un laps de temps défini (quarante-huit heures dans notre cas); dans le cas contraire, son compte sera désactivé.

Ce mécanisme comporte plusieurs étapes que nous allons détailler aux points suivants.

5.1.4.1. Formulaire d'inscription

L'utilisateur s'inscrit sur le site: il remplit un formulaire comprenant son adresse mail et le mot de passe qu'il choisit d'utiliser. Ces données sont enregistrées en base de données. Pour permettre une navigation fluide et rapide, l'utilisateur accède au site dès l'envoi de ce formulaire au serveur.

5.1.4.2. Cryptage du mot de passe

Le mot de passe est envoyé de manière sécurisée au serveur via SSL (voir section *SSL et HTTPS*). Cependant, il est évident que nous n'allons pas le stocker tel quel dans la base de données. En effet, nous évitons de cette manière que quelqu'un ayant accès à la base de données ait accès aux mots de passe.

Par conséquent, nous le sauvegardons sous une forme cryptée à l'aide de l'algorithme SHA-256, dont le module de sécurité du conteneur saura de plus tirer parti facilement. Nous avons choisi cet algorithme, qui est une version de SHA-2, car il ne comporte pas les failles de SHA-1.

5.1.4.3. Génération du lien de validation

La table contenant les données de l'utilisateur comporte un attribut de type booléen nommé "active". Ce champ indique si l'utilisateur a validé son adresse email ou non. Lors de l'enregistrement des données de l'utilisateur, nous mettons ce flag à false. Durant ce processus, nous générons également un token unique; ce token, connu de nous seul, sera inclus en tant que paramètre GET dans l'URL que nous envoyons par email à l'utilisateur. Lorsque l'utilisateur accèdera à ce lien, ce token sera récupéré et vérifié par notre code métier. Si le code est accepté, le compte sera activé: active sera passé à true, le token sera supprimé de la base de données.

5.1.4.4. Désactivation des comptes inactifs à l'aide d'un EJB timer

Nous avons spécifié plus tôt que l'utilisateur disposait de 48h pour valider son adresse mail. Pour détecter les comptes qui n'ont pas été vérifiés, nous utilisons un EJB qui déclenchera une méthode spécifique régulièrement, à un moment choisi par le développeur. Cette méthode est

annotée avec @Schedule et un attribut qui contient l'expression qui nous permet de définir l'intervale de temps au terme duquel nous souhaitons que la méthode soit exécutée.

Dans le cadre de la version de test de cette application, nous déclenchons cette action toutes les cinq minutes. Néanmoins, dans une situation réelle, nous aurions choisi de la déclencher toutes les 24h, vraisemblablement à une heure de faible trafic sur le site afin de ne pas impacter les performances de l'application. L'expression aurait donc ressemblé à l'expression suivante, où l'on indique que l'on souhaite que l'action s'effectue tous les jours à 4h.

```
@Schedule(hour="4")
```

La méthode qui possède cette annotation déclenche un scan des comptes qui n'ont pas encore été validés. Leur date de création est comparée avec la date actuelle, et dans le cas où le délai de 48h est expiré, le compte et son token sont supprimés. Un mail est également envoyé à l'utilisateur pour le notifier de cette suppression.

5.2. Sécurité

Comme on peut le voir sur le diagramme d'acteurs[18], le cahier d'analyse spécifie que l'application distingue trois types d'utilisateurs:

- les visiteurs,
- les membres (des visiteurs authentifiés),
- les administrateurs.

En fonction du groupe auquel il appartient, un internaute a accès à un nombre plus ou moins restreint de pages: les utilisateurs ont accès au contenu public du site, les membres à leur espace personnel, les administrateurs aux pages dédiées à la gestion.

Il est donc nécessaire d'établir une stratégie de sécurité. Le processus qui permet de déterminer si un utilisateur est autorisé ou non à accéder à une ressource consiste en trois phases. En premier lieu, on rencontre une phase d'identification qui permet de distinguer l'utilisateur au sein du système. Vient ensuite une phase d'authentification dont le but est d'assurer au système que l'utlisateur est bien celui qu'il prétend être. Pour terminer, c'est la phase d'habilitation qui détermine précisément si l'accès à la resource est autorisé, en fonction du ou des rôles qui ont été attribués à l'utilisateur.

5.2.1. JAAS (Java Authentication and Authorization Service)

JAAS est une API qui fait partie de la couche Java SE, et sur laquelle reposent les processus de sécurisation des applications Web ainsi que des EJBs. Si la phase d'identification est assurée par l'application Web, le plus souvent au moyen d'un formulaire envoyé avec une requête HTTP au serveur Web, c'est par contre le conteneur qui procède aux phases d'authentification et d'habilitation. Pour ce faire, il utilise JAAS pour accéder de manière transparente au système choisi.

[18] Voir annexes.

5.2.2. Domaine, rôle, principal

Domaine, rôle et principal sont les trois éléments clefs sur lesquels repose la sécurisation de notre application.

Pour commencer, le domaine comprend l'ensemble des règles choisies pour la sécurisation. Ce domaine comprend la liste des rôles, la liste des utilisateurs, ainsi que le liens qui les unissent: quels rôles sont attribués à quel utilisateur. Comme mentionné plus haut, ce domaine peut reposer sur différents principes. Dans notre cas, ces données proviendront de notre base de données, mais ce domaine pourrait par exemple être basé sur un annuaire LDAP, un Active Directory, ...

Ensuite, la notion de rôle est évidente. Chaque utilisateur a qui on a attribué un rôle devient membre d'un groupe qui partage des permissions sur un certain nombre de resources. Un utilisateur peut faire partie d'un ou plusieurs groupes.

Enfin, le principal est un utilisateur du site qui, suite à un processus d'authentification, est reconnu comme faisant partie d'un ou plusieurs groupes.

5.2.3. Création d'un domaine JDBC avec Glassfish

Nos règles de sécurité (utilisateurs, rôles, groupes) étant stockées dans la base de données, nous devons créer un domaine utiisable par Glassfish à l'aide de JDBC. C'est de cette manière que le serveur prendra connaissance de ces règles.

Pour effectuer cette manipulation à l'aide d'une interface graphique, il faut se rendre dans la console d'administration, dans la section dédiée à la sécurité de notre instance de serveur. On peut trouver dans cette section un outil qui nous permet de créer un nouveau domaine, dans notre cas de type com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm.

Cet outil demande de renseigner plusieurs informations, entre autres:

- le nom JNDI de la resource JDBC utilisée,
- le nom de la table reprenant les utilisateurs autorisés et la colonne qui reprend les noms d'utilisateurs,
- le nom de la table reprenant les groupes et la colonne qui reprend les noms de groupe,
- l'algorithme de cryptage des mots de passe.

5.2.4. Création d'une vue SQL

Le JDBCRealm proposé par Glassfish nécessite de trouver dans la même table les noms d'utilisateurs et les noms de groupe; le nom d'utilisateur et le nom du groupe constituent en quelque sorte une clé primaire à ses yeux. Cependant, notre modèle relationnel utilise une clé technique comme clé primaire (une value numérique issue de la séquence user_id_seq).

Nous pourrions créer une table de jointure qui contiendrait les données requises, mais la base serait par conséquent dénormalisée en raison de la redondance de ces données.

Afin de préserver d'éviter de modifier la structure de la base de données et de dupliquer l'information, nous créons une vue SQL qui reprend les informations requises par le domaine JDBC: les noms d'utilisateurs, les mots de passe et les groupes.

```
CREATE OR REPLACE VIEW usergroup_vw AS
SELECT u.email AS username, p.hashed AS password, r.name AS groupname
FROM user_ u
JOIN password p ON u.password_id = p.id
JOIN user__role_ ur ON u.id = ur.user_id
JOIN role_ r ON ur.roles_id = r.id
```

Nous renseignons enfin cette vue dans la console d'administration, en lieu et place d'une ou plusieurs tables[19].

5.2.5. Cryptage du mot de passe

Le realm de Glassfish permet d'utiliser des mots de passe cryptés avec différents algorithmes. Il suffit d'indiquer le type choisi dans le champ prévu à cet effet. Par défaut, c'est le SHA-256 qui est utilisé. Nous ne changeons pas cette valeur car c'est l'algorithme que nous utilisons également (voir section *Inscription*).

5.2.6. Point de vue applicatif

Il faut maintenant refléter cette sécurisation côté applicatif: indiquer à l'application quel type de login on choisit, quel realm utiliser, et quelles sont les contraintes à appliquer à quelle ressource.

5.2.6.1. Mécanisme d'authentification

Java EE propose plusieurs types d'authentification, parmi lesquelles nous citerons basic et form. Ces deux méthodes fonctionnent de manière similaire: lorsque l'utilisateur fait une requête vers une ressource protégée, le serveur lui demande un nom d'utilisateur et un mot de passe. Dans le cas de l'authentification basic, c'est à l'aide d'un pop up propre au browser, dans le cas de l'authentification form, à l'aide d'une page contenant un formulaire et référencée dans le fichier web.xml. Ces données seront ensuite envoyées au serveur, encodées en Base64 dans le cas basic, et en clair dans le cas form. On se rend bien compte que ces méthodes ne sont pas suffisamment sécurisées, c'est là que SSL intervient (voir section *SSL et HTTPS*).

Nous choisissons d'utiliser la méthode form, et pour ce faire nous devons l'indiquer dans le web.xml, de même que les pages qui correspondent au formulaire et à la page à afficher en cas d'erreur.

[19] Nota bene: cette vue a été modifiée en cours de développement pour permettre une authentification à l'aide du framwork SocialAuth, nous renvoyons le lecteur à la section *SocialAuth*.

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>SociallRealm</realm-name>
    <form-login-config>
        <form-login-page>/pages/signin/login.xhtml</form-login-page>
        <form-error-page>/pages/signin/loginerror.xhtml</form-error-page>
    </form-login-config>
</login-config>
```

5.2.6.2. Déclaration des rôles

Il faut ensuite déterminer quel rôle a accès à quelle resource, au moyen de motifs d'urls. On déclare par exemple de la sorte que seuls les utilisateurs faisant partie du groupe MEMBER ont accès aux pages situées dans le dossier member:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Member pages</web-resource-name>
        <url-pattern>/pages/member/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>MEMBER</role-name>
    </auth-constraint>
</security-constraint>
```

5.2.6.3. Mapping application/base de données

Pour terminer, les rôles utilisés dans le deployment descriptor ne reflètent pas nécessairement aux groupes de la base de données. Pour réaliser la correspondance au niveau du serveur, nous ajoutons ces lignes au fichier glassfish-web.xml.

```
<security-role-mapping>
    <role-name>MEMBER</role-name>
    <group-name>MEMBER</group-name>
</security-role-mapping>
```

5.3. SSL et HTTPS

Dans la section *Inscription*, nous avons vu que les utilisateurs peuvent se connecter sur le site à l'aide d'un couple nom d'utilisateur/mot de passe. Pour éviter que ce mot de passe ne transite en clair sur le réseau et ne soit potentiellement récupéré par quelqu'un de malveillant, nous utiliserons le protocole HTTPS. Ce protocole est une version de HTTP utilisant SSL (Secure Socket Layer). Il nous assure que les données qui transitent entre le client et le serveur sont protégées grâce à un système de chiffrement et de signature des messages.

5.3.1. Côté applicatif

Pour que notre application utlise HTTPS, il faut ajouter une contrainte de sécurité dans le web.xml, semblable à celles déclarées pour le realm de Glassfish (voir *Sécurité*). Nous décidons que les pages membres, administrateurs et relatives au login devront être utilisées. Nous déclarons la contrainte suivante pour les pages relatives au login:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>SecureConnection</web-resource-name>
        <url-pattern>/pages/signin/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Et nous ajoutons l'élément user-data-constraint aux contraintes déjà existantes pour les pages membre et admin.

C'est l'élément transport-guarantee et la valeur CONFIDENTIAL qui forcera toute requête sur une adresse spécifiée dans le ou les éléments url-pattern à transiter sur une connection sécurisée.

5.3.2. Signatures et certificats

Comme nous l'avons mentionné plus haut, SSL demande aussi une authentification de la part du serveur. Cette authentification permet s'assurer que l'on s'adresse bien au bon interlocuteur, et pour ce faire, le protocole utilise des certificats. Un certificat correspond à une carte d'identité du serveur. Elle contient des informations sur ce dernier, et quand elle est signée par une autorité de certification, elle garantit son identité. Les autorités de certification sont "des organismes enregistrés et certifiés auprès d'autorités publiques ou de gouvernance de l'internet qui établissent leur viabilité comme intermédiaire fiable"[20].C'est ce certificat que SSL évaluera avant de remettre son message.

5.3.3. Clé publique et clé privée

Les messages transportés par SSL sont chiffrés de manière asymétrique, c'est-à-dire à l'aide d'une clé publique et d'une clé privée. La clé publique permet de chiffrer les messages de manière à ce que seule le possesseur de la clé privée puisse les déchiffrer, tandis que la clé privée chiffre des messages destinés à tous les possesseurs de la clé publique. C'est par l'intermédiaire de certificats signés par une autorité de certification que la clé publique est transmise, afin de sécuriser cette transmission.

[20] http://fr.wikipedia.org/wiki/Certificat_%C3%A9lectronique

5.3.4. Utilisation de HTTPS avec Glassfish

Le support du SSL est configuré “out-of-the-box” pour Glassfish, avec l’utilisation d’un certificat self-signed, ce qui signifie qu’il n’a pas été signé par une autorité de certification. Cela peut être le cas pour différentes raisons: environnement de développement, réseau d’entreprise, … Dans cette situation, le navigateur détecte que ce certificat n’a pas été signé par quelqu’un faisant partie de sa liste préétablie d’autorités, et affiche un message d’avertissement à l’utilisateur.

Pour nous rapprocher d’un cas d’utilisation réel, nous avons néanmoins décidé de générer nous-même ce certificat, et de le signer nous-même à l’aide d’une clé fraîchement générée elle aussi. Pour ce faire, nous utiliserons OpenSSL, l’outil keytool de Java et Strawberry Perl, Perl n’étant pas installé par défaut sur une machine Windows, contrairement à un environnement Linux.

Glassfish utilise l’alias s1as, nous allons le remplacer par nos données après l’avoir retiré du keystore, le trousseau de clés. Nous pouvons ensuite générer un certificat, foodish.csr. Ces manipulations s’effectuent dans le dossier config du domaine Glassfish dont fait partie notre application.

```
keytool -delete -alias s1as -keystore keystore.jks
keytool -genkeypair -dname "CN=foodish.com,O=Foodish,L=Liege,S=Liege,C=BE" -alias s1as
-keystore keystore.jks
keytool -certreq -alias s1as -keystore keystore.jks -file foodish.csr
```

Dans un environnement de production, le certificat serait transmis à une autorité de certification pour signature. C’est un acte payant qui peut être effectué auprès d’organismes comme GoDaddy, Comodo, et de nombreux autres.

Dans notre cas, nous allons créer une autorité de certification et signer nous-même notre certificat, ce qui ne garantit rien à l’utilisateur mais nous permettra tout de même d’utiliser HTTPS. C’est ici qu’intervient OpenSSL et son script perl CA.pl. Dans l’ordre, nous créons l’autorité (CA), puis nous signons le certificat, avant enfin de modifier l’encodage de ce certificat pour Glassfish.

```
perl <dossier OpenSSL>\CA.pl -newca
copy foodish.csr foodish.pem
perl <dossier OpenSSL>\CA.pl -sign
openssl x509 -in newcert.pem -out foodish.der -outform DER
openssl x509 -in cacert.pem -out demoCA.der -outform DER
```

Une fois en possession de nos certificats fraîchement signés, il faut les ajouter au keystore de Glassfish (après les avoir copiés dans config).

```
keytool -importcert -alias demoCA -file demoCA.der -keystore keystore.jks
```

```
keytool -importcert -alias s1as -file foodish.der -keystore keystore.jks
```

5.4. Stockage des fichiers uploadés

L'application offre la possiblité aux utilisateurs d'uploader des photos. Ces photos seront utlisées soit comme image de profil, soit pour illustrer les articles.

5.4.1. Choix de la stratégie de stockage

Il existe deux possiblités pour stocker des médias: soit dans la base de données en tant que BLOB (Binary Large Object), soit sur un disque. C'est alors un chemin vers le fichier sur le disque qui sera sauvegardé dans la base de données. Les deux méthodes présentent des avantages et des inconvénients[21].

	Système de fichiers	**Base de données**
+	• Pas de limite dans la taille des fichiers • Accès rapide aux images	• Meilleure sécurisation des resources grâces aux permissions SQL • Garantie du respect des critères ACID[22].
-	• Il faut envisager un backup du filesystem en plus du backup de la base de données	• Limite de taille maximale des fichiers (2GB). • Les BLOB font rapidement grossir la base de données

Nous avons choisi de stocker ces médias dans un système de fichiers pour plusieurs raisons.

Tout d'abord, le cahier des charges spécifie que pour chaque page, de nombreuses photos peuvent être ajoutées par les utlisateurs; nous souhaitons donc éviter une croissance importante de la taille de la base de données, ce qui pourrait impacter ses performances.

Ensuite, les photos uploadées étant toutes publiques, leur sécurisation n'est pas une priorité.

Pour terminer, ces photos ne sont pas d'une importance cruciale, elles ne représentent pas un enjeu important. On peut donc se permettre un petit manque de respect de l'intégrité des données dans le cas d'une erreur qui aurait dû causer un rollback. On pourra envisager une maintenance régulière de la base de données pour détecter ces erreurs et retirer les records erronnés, à l'aide d'un batch ou d'un EJB Timer par exemple.

---

21 https://wiki.postgresql.org/wiki/BinaryFilesInDB
http://blog.sqlauthority.com/2009/07/13/sql-server-blob-pointer-to-image-image-in-database-filestream-storage/

22 Atomicité, cohérence, isolation et durabilité. http://fr.wikipedia.org/wiki/Propri%C3%A9t%C3%A9s_ACID

5.4.2. Configuration de Glassfish

Une application web est capable de servir des ressources situées dans son docroot, c'est-à-dire dans un dossier faisant physiquement partie de l'application. Cependant, nous souhaitons que le répertoire dans lequel nous stockons les uploads soit indépendant de l'application, pour éviter de perdre ces ressources lors de l'undeploy de l'application, par exemple.

Glassfish propose de définir des alternate docroot, qui permettent à l'application de servir des ressources situées hors de son document root lorsque les requêtes correspondent à une URI définie.

Pour définir un alternate docroot, il suffit d'ajouter ces lignes au fichier glassfish-web.xml:

```
<property name="alternatedocroot_1"
          value="from=/images/* dir=D:\java\uploads"/>
```

Ce qui se traduira par "aller récupérer toutes les requêtes de type "<contexte>/images/… dans le dossier D:\java\uploads\images".

Il est également possible de définir un serveur virtuel dans le fichier domain.xml de Glassfish (ou à l'aide de la console d'administration). Nous aurions préféré utiliser cette solution afin que le chemin du dossier utilisé reste lié à l'environnement de développement/d'exécution, mais à l'heure où nous écrivons ces lignes, nous n'avons pas réussi à faire fonctionner cette solution.

5.4.3. Primefaces FileUpload

Nous utilisons le composant Primefaces fileUpload pour effectuer le transfert des photos à proprement parler. Ce composant tire parti des fonctionnalités d'upload de HTML5 tout en proposant une solution de remplacement pout les browsers non compatibles.

Ce composant nous offre également la possiblité de restreindre les types de fichier que nous acceptons. Nous avons choisi d'accepter les formats .png, jp(e)g, .gif et .bmp. Nous avons également limité la taille maximale des fichiers à 5000 bytes.

```
<p:fileUpload fileUploadListener="#{userController.uploadProfilePic}"
                        allowTypes="/(\.|\/)(gif|jpe?g|png)$/"
                        sizeLimit="5000"/>
```

5.5. Utilisation du Server-Push et des Server-Sent Events HTML5

Le modèle HTTP traditionnel fonctionne selon un principe de requête et de réponse: le client fait une demande explicite au serveur, qui lui retourne une réponse qui entraîne un rechargement de la page. Si ce modèle est adapté dans la plupart des cas d'utilisation, il ne convient pas, par exemple, dans le cas de la messagerie instantanée que nous souhaitons implémenter. En effet, nous souhaitons que l'utilisateur soit averti immédiatement de l'arrivée d'un nouveau message.

On pourrait imaginer effectuer des requêtes au serveur à intervale régulier pour vérifier si de nouveaux messages sont disponibles. Cependant, d'une part, les messages ne seraient toujours pas instantanés, et d'autre part, un traffic non négligeable et inutile serait engendré.

C'est la raison pour laquelle nous avons implémenté une solution qui utilise la technique du server push: c'est le serveur qui, à la suite d'un événement, envoie un message au client. Ce client doit au préalable s'être enregistré (on parle de registration) auprès du serveur pour pouvoir recevoir ces messages. On remarque qu'il s'agit là d'une implémentation du pattern Observer.

Dans notre cas, ce processus est déclenché par l'envoi d'un message par un utilisateur A vers un utilisateur B. Le message est sauvegardé en base de données car nous souhaitons que les conversations soient persistantes: l'utilisateur peut les retrouver en revenant sur l'application, il peut aussi envoyer et recevoir des messages en différé.

Suite à cet envoi, l'application envoie en parallèle le contenu du message au browser de l'utilisateur B, si il est connecté; cette notification permet de rafraîchir la portion adéquate de la page et d'afficher le nouveau message.

Il existe plusieurs manières de transmettre des données en push. HTML5 permet par exemple, à l'aide des server-sent events et de l'objet EventSource, d'envoyer les notifications vers le browser. Les websockets, quant à eux, permettent de recevoir des notifications mais aussi de les envoyer. à l'aide d'une connection client-serveur. Ces technologies présentent chacune leurs avantages et leur inconvénients[23], et elles ne sont pas supportées de manière homogène par tous les navigateurs. De plus, les implémentations des websockets sont liées au serveur, et pas toujours portables.

C'est ici qu'intervient le framework Atmosphere[24]. Ce framework permet d'utiliser ces différentes technologies en toute transparence, tout en proposant des solutions de dégradation quand elles ne sont pas supportées par le browser. C'est au travers de Primefaces Push que nous emploierons Atmosphere, ce qui nous permet de simplifier son utlisation au travers d'une API adaptée à JSF.

Ainsi, il nous sera permettre d'envoyer une notification de la manière suivante:

```
PushContext pushContext = PushContextFactory.getDefault().getPushContext();
pushContext.push("/conversation/" + userId, message);
```

[23] http://stackoverflow.com/a/5326159

[24]Atmosphere, Realtime Client Server Framework for the JVM, supporting WebSockets and Cross-Browser Fallbacks Support, https://github.com/Atmosphere/atmosphere

L'utilisation de l'id (userId) nous permet de créer un channel de communication dynamique: les notifications ne sont envoyées qu'à l'utilisateur qui s'est abonné à ce channel. C'est le composant Primefaces p:socket qui permet d'effectuer cette registration:

```
<p:socket channel="/conversation/#{userSession.user.id}" onMessage="handleMessage"/>
```

Dans l'attribut onMessage, on retrouve le nom de la fonction Javascript qui sera déclenché lors de l'événement onMessage (donc à la réception d'une notification). Dans notre cas, cette méthode déclenchera, comme mentionné précédemment, un rafraîchissement d'une portion spécifique de la page.

5.6. Internationalisation

JSF facilite l'internationalisation des applications en proposant de stocker tous les messages à afficher dans un fichier de properties, sous forme de paires clé-valeur. Il suffira ensuite de référencer ces messages à l'aide de leur clé. Pour avoir accès à ces ressources, il faut déclarer quelques éléments dans le fichier de configuration spécifique à JSF, faces-config.xml.

```
<application>
	<locale-config>
		<default-locale>fr</default-locale>
		<supported-locale>en</supported-locale>
	</locale-config>
	<resource-bundle>
		<base-name>be.isl.desamouryv.sociall.i18n.messages</base-name>
		<var>msg</var>
	</resource-bundle>
</application>
```

A l'aide de ces éléments, on spécifie quelles sont les locales supportées par l'application (ici, le français et l'anglais), ainsi que le nom complet (fully qualified name) du bundle de resources. Ce bundle contient les différents fichiers de properties dans lequel se trouvent les messages. A l'exception de la locale par défaut qui correspondra au fichier messages.properties, les noms des fichiers seront suffixés par la locale qui leur correspond; ainsi, on trouvera un fichier messages_en.properties.

Il est ensute aisé d'accéder au resource bundle. Au sein d'une facelet, ce sera à l'aide du nom spécifié dans l'élément <var/>:

```
<h:outputText value="#{msg['common.appname']}"/>
```

On pourra également récupérer un message dans le code Java:

```
ResourceBundle bundle = ResourceBundle.getBundle("be.isl.desamouryv.sociall.i18n.messages",
                    FacesContext.getCurrentInstance().getViewRoot().getLocale());
```

```
String translation = bundle.getString("common.appname");
```

On constate qu'il est nécessaire d'indiquer la locale dans laquelle on souhaite recevoir le message; c'est également ce qui se passe en interne quand on référence le bundle sous le nom msg. Par défaut, c'est la locale du browser qui est prise en compte, mais il est également possible de la modifier programmatiquement, afin par exemple de laisser l'utilisateur choisir la langue dans laquelle il souhaite afficher l'application, indépendamment de la configuration de son navigateur.

C'est ainsi que nous avons placé une combox dans le footer de l'application qui permet d'afficher le site soit en français, soit en anglais. Cette combox fait appel à un managed bean, LanguageController, qui se charge de modifier la locale courante.

5.7. API REST

Tout comme nous utilisons les APIs d'autres réseaux, notamment pour identifier les utilisateurs, nous souhaitons permettre à d'autres utilisateurs d'interagir avec notre application et ses données. C'est la raison pour laquelle nous allons développer notre propre API, accessible au travers d'un Web Service REST.

Dans un premier temps, nous allons seulement rendre accessibles certaines de nos données publiques au travers de quelques méthodes simples. Néanmoins, on pourrait envisager par la suite l'élargissement de cette API en lui permettant d'interagir avec les comptes des utilisateurs: création, modification, etc. Pour cela, il faudra envisager l'utilisation d'un token d'authentification.

Un web service "est un programme informatique de la famille des technologies web permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués". Il permet donc d'échanger des informations entre des systèmes qui peuvent utiliser des technologies différentes, et ce de manière plus ou moins standardisée, au travers d'un réseau. Il s'agira généralement d'Internet et HTTP.

5.7.1. SOAP et REST

Tous les web services n'utilisent pas les mêmes technologies. On distingue deux types de web services différents: les services WS-* et les services REST (Representational State Transfer).
Ces deux technologies sont représentées dans le monde Java par les API JAX-WS et JAX-RS.
Nous avons choisi de développer un web service REST en raison de la simplicité et de la légèreté de REST en comparaison avec l'alternative WS-* et SOAP, mais également parce que nous n'avons pas besoin des fonctionnalités supplémentaires offertes par SOAP (gestion des transactions, sécurité, etc.). Il ne faut cependant pas croire que la simplicité offerte par REST rime avec simpliste: en effet, c'est le protocole utilisé par les grands acteurs du web d'aujourd'hui (Twitter, Amazon pour ne mentionner qu'eux) et son efficacité n'est pas à remettre en cause.

De plus, REST se repose sur les principes du web, comme l'accès aux ressources au travers d'une URI, ou l'utilisation des méthodes du protocole HTTP (GET, PUT, DELETE).

5.7.2. JAXB

Pour que nos réponses soient lisibles par notre client qui n'est pas nécessairement écrit en langage Java, il faut les envoyer dans un format qu'il peut comprendre. JAXB (Java Architecture for XML Binding) permet à JAX-RS de convertir nos objets en XML (Extensible Markup Language) et vice-versa grâce à des opérations de sérialisation/désérialisation (marshalling/unmarshalling); le but du XML est d'offrir un format d'échange standardisé.

Outre le XML, JAXB va aussi nous permettre d'écrire nos réponses en format JSON (JavaScript Object Notation), format notamment facile à manipuler par des clients écrits en JavaScript. C'est également le format communément utilisé par la plupart des APIs de réseaux ou sites à grande fréquentation.

5.7.3. Implémentation

Depuis la version 3.0 des Servlet, aucune configuration en xml n'est nécessaire pour exposer des web services. Il suffit d'intégrer dans l'application une implémentation de la classe javax.ws.rs.core.Application. Dans cette classe, nous déclarons les différentes ressources et nous les retournons dans la méthode overridée Application#getClasses, ce qui permet d'éviter que tout le classpath de l'application soit scanné à la recherche des classes adéquates.

Nous annotons cette classe avec l'annotation suivante:

```
@javax.ws.rs.ApplicationPath("/")
```

L'attribut de l'annotation représente la portion d'URI qui identifiera nos différents services. Nous souhaitons que nos services soient accessibles immédiatement à la racine du context root, c'est pour laquelle nous indiquons "/".

En ce qui concerne nos services à proprement parler, il s'agit de simples classes pourvues de méthodes et annotées de manière spécifique. Il aurait été tout à fait possible d'annoter directement nos EJBs déjà existants de manière à les exposer en tant que web services. Cependant, la structure de notre application et sa découpe en différents module veut que ces EJBs fassent partie d'un package séparé du package war; or, un web service REST, qui doit être accessible via HTTP, doit faire partie d'un projet de type war. C'est pour cette raison que nous avons créé le war nommé ws, dans lequel on trouvera les services. Les classes que nous utilisons comme services sont néanmoins annotées avec @Stateless, ceci afin de permettre l'injection de dépendances.

Ces classes possèdent également une annotation @Path, dont l'attribut indique leur portion d'URI spécifique. Nous choisissons par exemple "artifact" pour le service chargé de retourner des données relatives aux artifacts. Les méthodes exposées sont également annotées de

manière à leur donner une URI propre. Enfin, l'annotation @GET spécifie que cette URI doit être appelée avec la méthode HTTP GET.

La configuration suivante permettra d'appeler la méthode latest à l'URI http://localhost:8080/foodishws/artifacts/latest/XXX, la dernière portion correspondant au pramètre déclaré avec @PathParam. On notera aussi l'annotation @Produces qui nous permet de spécifier le ou les formats de la réponse à retourner.

```
@Path("artifacts")
@Stateless
public class ArtifactWebService {

        @EJB
        private ArtifactFacade artifactFacade;

        @GET
        @Path("latest/{max}")
        @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
        public List<Artifact> latest(@PathParam("max") Integer max) {
            return artifactFacade.findLatest(max);
        }
}
```

5.7.4. Annotation des objets

Pour terminer, il est nécessaire de donner quelques informations à JAXB sur la manière dont il faut sérialiser les objets.

Tout d'abord, l'annotation @XmlRootElement spécifie l'élément racine; c'est la balise qui contient les autres balises qui décrivent les données. Par exemple, pour un objet de type Recipe, c'est de la balise <recipe/> qu'il s'agit.

Ensuite, l'annotation @XmlAccessorType(XmlAccessType.FIELD) spécifie que l'on souhaite accéder aux attributs par le champ et non par le getter. Elle permet aussi que tous les champs fassent partie du contenu XML. On souhaite néanmoins parfois que ce ne soit pas le cas pour certaines propriétés, parce que leur contenu n'est pas utile ou par sécurité. Dans ce cas, on a la possiblité de les annoter avec @XmlTransient afin de les rendre invisibles pour le XML.

Enfin, nous avons également annoté la classe Artifact avec l'annotation @XmlSeeAlso:

```
@XmlSeeAlso({Event.class, Recipe.class, Place.class, Product.class})
```

Cette annotation permet, lors de la génération d'une réponse contenant un ou plusieurs Artifact, d'envoyer aussi les données spécifiques au type réel de l'objet, comme par exemple l'adresse d'un lieu ou les ingrédients d'une recette.

6. Procédure d'installation de l'application

L'application sera distribuée sous forme de fichier ear à déployer sur un serveur d'application Java EE 7 compliant, appuyé par une base de données de n'importe quel type. Nous décrirons ici les différentes procédures à suivre pour l'installation sur un serveur Glassfish 4 et une base de données PostgreSQL.

6.1. Création de la datasource

Pour utilise la base de données, l'application utilise une datasource JTA. Le nom JNDI sous lequel cette datasource doit être déclarée est spécifiée dans le fichier persistence.xml.

```
<jta-data-source>jdbc/sociall</jta-data-source>
```

Il sera donc nécessaire de créer une datasource portant ce nom dans Glassfish. Cette procédure comporte deux étapes: tout d'abord la création d'un connection pool, ensuite la création de la datasource à proprement parler. L'utilisateur devra renseigner les informations de connexion propres à sa base de données: le nom de la base, l'hôte, le nom d'utilisateur et le mot de passe. Dans la console d'administration, ces données sont configurées dans Resources\JDBC\JDBC Resources et Resources\JDBC\JDBC Connection Pool.

6.2. Création du realm de sécurité

Cette procédure a été décrite dans les sections SocialAuth et Création d'un domaine JDBC avec Glassfish. Grosso modo, ce realm utilisera les propriétés suivantes telles que déclarées dans le fichier domain.xml de Glassfish:

```
<auth-realm classname="com.sun.enterprise.security.ee.auth.realm.jdbc.JDBCRealm"
name="SociallRealm">
       <property name="jaas-context" value="jdbcRealm"></property>
       <property name="password-column" value="password"></property>
       <property name="datasource-jndi" value="jdbc/sociall"></property>
       <property name="group-table" value="usergroup_vw"></property>
       <property name="charset" value="UTF-8"></property>
       <property name="user-table" value="usergroup_vw"></property>
       <property name="group-name-column" value="groupname"></property>
       <property name="digestrealm-password-enc-algorithm" value="SHA-256"></property>
       <property name="user-name-column" value="username"></property>
</auth-realm>
```

6.3. Création de la session Javamail

L'application utilise une session Javamail pour envoyer les notifications aux utilisateurs. Cette session doit porter le nom mail/sociall car c'est à travers ce nom que l'EJB MailSender y accède:

```
@Resource(name = "mail/sociall")
 private Session mailSession;
```

6.4. Mise en place de la base de données

Pour la création de la base de données, deux stratégies sont possibles: soit un script SQL permettant la création des tables peut être distribué, soit ces tables peuvent être créées automatiquement au déployement de l'application grâce à ces lignes déclarées dans le fichier persistence.xml

```
<property name="javax.persistence.schema-generation.database.action" value="create"/>
<property name="javax.persistence.sql-load-script-source" value="data/data.sql"/>
```

La première propriété permet de créer les tables au départ des annotations présentes au niveau des entités. La seconde propriété permet l'exécution du script data.sql présent dans le dossier data, dont le chemin est relatif à la racine du persistence unit (chez nous, dans resources\META-INF). Par exemple, en développement, c'est dans ce fichier que nous avons inséré nos données de test afin que les tables soient populées automatiquement à chaque déploiement.

7. Problèmes (et solutions)

7.1. Bug de la version 2.2.0 de Mojarra

Durant le développement, nous avons constaté que les paramètres passés à une facelet à l'aide du tag <f:viewParam/> étaient toujours null sans raison. Quelques recherches sur Internet nous ont permis de prendre connaissance d'un bug[25] lié à la version 2.2.0 de Mojarra, que Glassfish utilise par défaut.

7.1.1. Solutions

Au niveau applicatif: ajouter dans le classpath du projet une autre version de Mojarra (dans notre cas, à l'aide de Maven), et ajouter les lignes suivantes au fichier glassfish-web.xml afin de spécifier au serveur d'utiliser la version de l'application:

```
<class-loader delegate="false" />
<property name="useBundledJsf" value="true" />
```

Au niveau du serveur: remplacer le jar javax.faces dans le dossier module.
Nous avons retenu cette deuxième solution afin de ne pas renconter à nouveau le problème avec les autres applications que nous pourrions déployer sur le même serveur.

7.1.2. Sources

http://stackoverflow.com/questions/10782528/how-to-update-mojarra-version-in-glassfish
http://miageprojet2.unice.fr/Intranet_de_Michel_Buffa/Cours_composants_distribu%C3%A9s_pour_l'entreprise_2013-2013/Mettre_%C3%A0_jour_la_version_de_JSF_%2F%2F_Mojarra

[25] https://java.net/jira/browse/JAVASERVERFACES-2900

7.2. Noms JNDI portables

Il est parfois nécessaire de recourir à un lookup JNDI pour récupérer une instance d'un EJB. C'est le cas par exemple avec les classes annotées avec @FacesValidator ou @FacesConverter[26], comme nous l'avons vu à la section *Limites de @FacesConverter*. Pour ce faire, il est nécessaire d'avoir des noms JNDI portables et dynamiques; on ne peut envisager de dépendre d'un numéro de version.

7.2.1. Solution

Pour que le nom JNDI du module ejb soit fixe, il suffit d'overrider son nom dans le fichier ejb-jar.xml (après l'avoir créé, le cas échéant) grâce au tag <module-name>.

Il serait également possible d'overrider le nom du module ear dans le fichier application-xml, mais nous n'y sommes pas arrivé malgré nos tentatives (par exemple avec maven-ear-plugin et le tag <generateApplicationXml/>). Cela n'a toutefois pas posé de problème car la ligne suivante renvoie le nom complet du module ear, version comprise:

```
new InitialContext().lookup("java:app/AppName");
```

La classe be.isl.desamouryv.sociall.ui.util.EJB sera pour finir chargée de récupérer les EJB au moyen de leur nom JNDI de type:

```
java:global/ear-<version>/ejb/RoleFacadeImpl
```

7.2.2. Sources

http://balusc.blogspot.be/2011/09/communication-in-jsf-20.html#GettingAnEJBInFacesConverterAndFacesValidator
https://blogs.oracle.com/MaheshKannan/entry/portable_global_jndi_names

[26] https://java.net/jira/browse/JAVASERVERFACES_SPEC_PUBLIC-763
http://jdevelopment.nl/jsf-22/#injection

7.3. Niveau des messages d'erreur du module de sécurité de Glassfish

Durant les phases de test de cette configuration relative à la sécurité, nous avons été confrontés à des erreurs dans le processus d'authentification. En effet, nos tentatives d'accès restaient infructueuses, sans provoquer d'erreur dans les fichiers de log du serveur. Cependant, les messages d'erreurs que nous recevions dans la console d'administration n'étaient pas suffisamment détaillés que pour nous puissions comprendre la cause de ces échecs:

```
WARNING:   WEB9102: Web Login Failed:
com.sun.enterprise.security.auth.login.common.LoginException: Login failed: Security
Exception
```

7.3.1. Solution

Pour avoir une idée plus claire des problèmes rencontrés durant ce processus, nous avons modifié le niveau des messages de log afin de pouvoir recevoir des informations plus précises. Dans la console d'administration de Glassfish, nous avons spécifié le niveau du logger javax.enterprise.system.core.security à "FINEST" (le plus détaillé), et nous avons ajouté un logger com.sun.enterprise.security, lui aussi mis à "FINEST". Nous avons ensuite reçu des messages bien plus détaillés:

```
FINE:   Cannot validate user
java.sql.SQLSyntaxErrorException: Column 'PASSWORD' is either not in any table in the FROM
list or appears within a join specification and is outside the scope of the join
specification or appears in a HAVING clause and is not in the GROUP BY list. If this is a
CREATE or ALTER TABLE  statement then 'PASSWORD' is not a column in the target table.
```

C'est de cette manière que nous avons pu remarquer, lors des premières phases de développement sur une base de données intégrée Derby, que les noms des tables et des colonnes tels que spécifiés dans Glassfish devaient respecter la casse (ce qui, par contre, n'est pas le cas avec une base PostgreSQL), et corriger ces erreurs.

7.3.2. Sources

http://www.developpez.net/forums/d1278581/java/serveurs-conteneurs-java-ee/glassfish/roles-jdbc-realm/

8. Conclusion

Le développement de cette application dans le cadre de l'épreuve intégrée nous a permis de mettre en oeuvre toutes les connaissances acquises durant le baccalauréat en informatique de gestion: du cours de principes et méhodes de programmation au cours de langage orienté gestion, en passant par le cours de réseaux ou encore celui de gestion de base de données.

Nous avons pu constater que chacune de ces connaissances est utiles au quotidien en tant que développeur d'application, car ses missions sont variées et ses domaines d'expertises vastes.

Nous conclurons en disant que nous avons la chance d'avoir acquis les bases d'un savoir qu'il sera nécessaire d'enrichir de manière continue. En effet, le métier d'informaticien demande d'être proactif et nous sommes convaincus que se tenir informé de l'évolution rapide des technologies est la clé d'une carrière réussie.

9. Pistes d'amélioration

9.1. Internationalisation de la base de données

Comme mentionné dans la section *Internationalisation*, il est possible d'afficher le site en anglais ou en français. Cependant, le contenu de la base de données ne possède pas d'indicateur permettant de distinguer le contenu rédigé dans l'une ou l'autre langue. Il pourrait être utile de modifier ceci afin d'afficher en priorité le contenu dans la langue de l'utilisateur.

9.2. Administration du site

Bien que le statut d'utilisateur existe, les actions qui lui sont réservées sont actuellement peu nombreuses et pourraient être élargies Par exemple, il serait envisageable de prévoir une validation des nouveaux articles et commentaires afin d'éviter un contenu offensant. De même, le contenu des mails envoyés par la classe MailSender pourrait provenir de la base de données et être éditable par l'administrateur.

9.3. Tests unitaires

Lors des premières phases de développement, nous avons souhaité développer en parallèle de nos méthodes métier des test unitaires avec Junit. Pour ce faire, nous avons souhaité utiliser un conteneur d'EJB embedded (EJBContainer), pour nous permettre de tester nos EJBs dans un environnement de test. Malheureusement, nous avons pu constater qu'il existe une grande incompatiblité entre Maven et le container intégré Glassfish 4. Nous avons tenté d'utiliser le container OpenEJB; cependant il n'existe à l'heure actuelle pas d'implémentation compatible Java 7.

Face à ces difficultés, le projet ne comprend malheureusement pas de test unitaires, ce qui est problématique.

9.4. Traitement des images uploadées

Il faudrait prévoir une solution de backup des images uploadées par les utilisateurs. De plus, comme il est impossible d'inclure la manipulation de fichiers dans des transactions, il faudrait également de manière régulière vérifier que les données présentes dans la base de données sont correctes, afin de présever leur intégrité.

10. Bibliographie

Gonvalves, Antonio, Java EE 6 et GlassFish 3, Pearson, 2010, 554 pages.
Freeman, Eric et Freeman, Elisabeth, Head First Design Patterns, O'Reilly, 2004, 678 pages.

11. Webographie

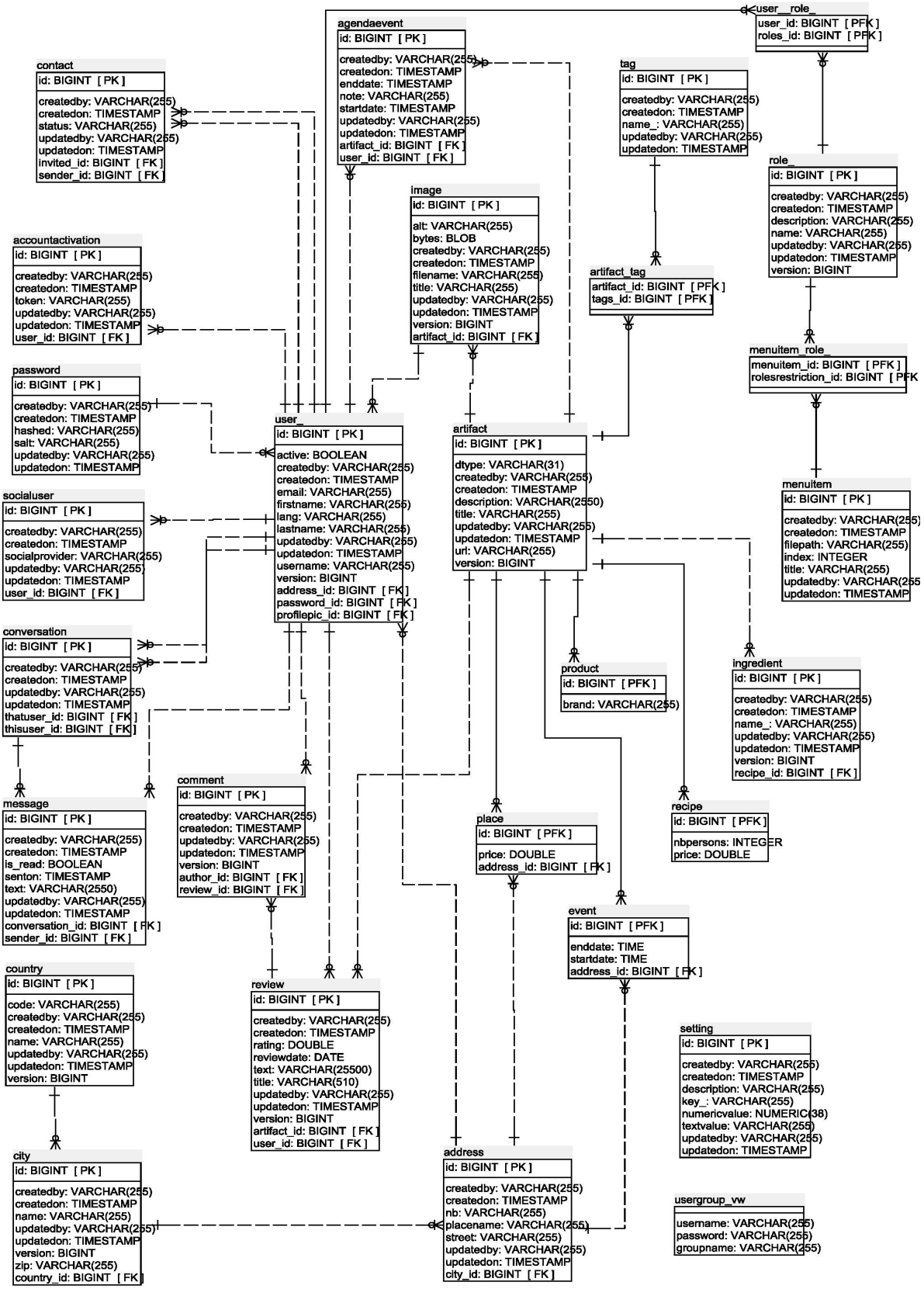
Cervera-Navarro, Ricardo, Evans, Ian, Haase, Kim, Jendrock, Eric et Markito, William. "The Java EE 7 Tutorial, Release 7 for Java EE Platform", pdf. Oracle, mai 2014.
http://docs.oracle.com/javaee/7/tutorial/doc/javaeetutorial7.pdf

Çivici, Çağatay. "Primeface User's Guide 4.0", pdf.
http://primefaces.googlecode.com/files/primefaces_users_guide_4_0_edtn2.pdf

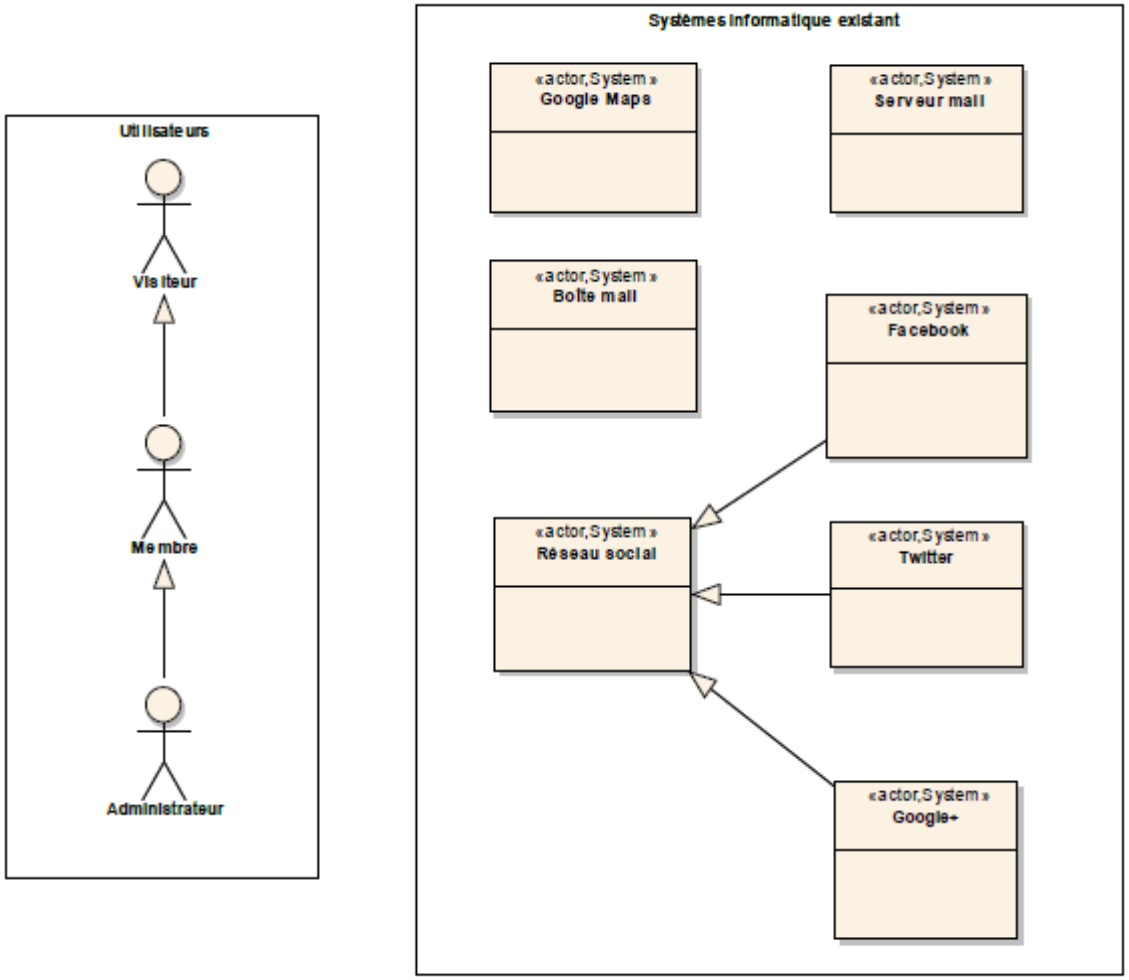
"GlassFish Server Open Source Edition Application Development Guide, Release 4.0", pdf. Oracle, mai 2013.
https://glassfish.java.net/docs/4.0/application-development-guide.pdf

Annexes

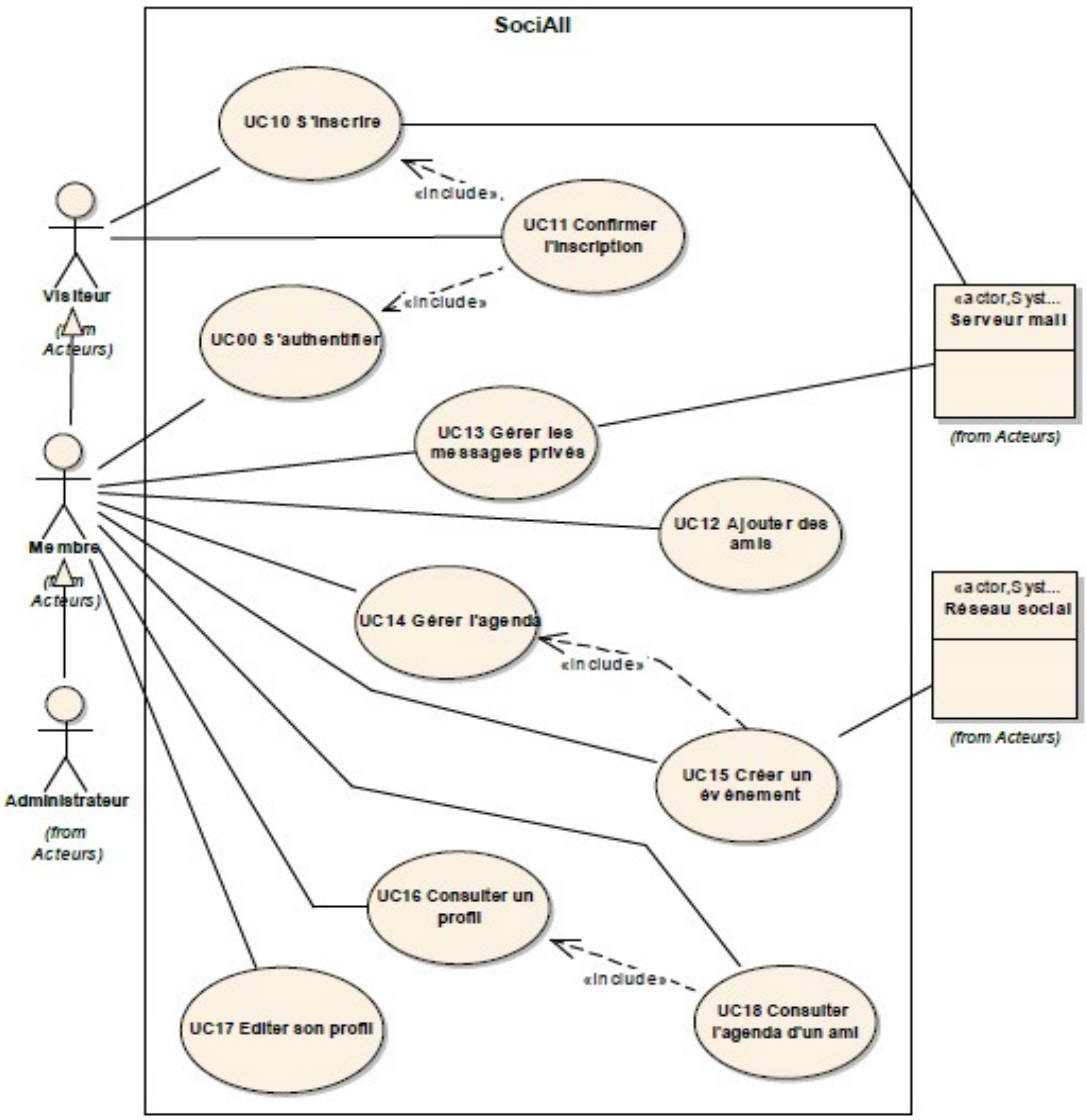
1. Schéma de la base de données



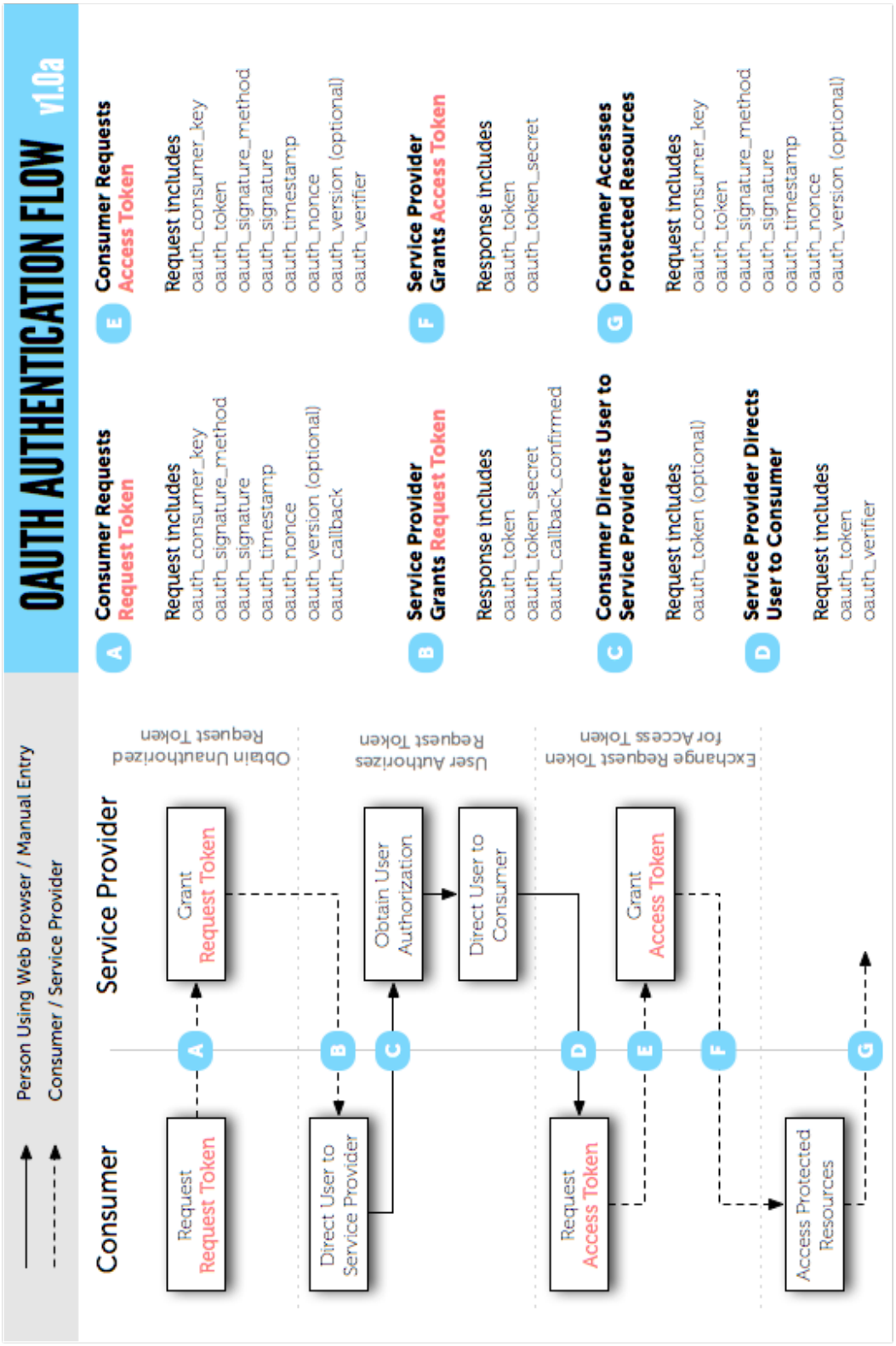
2. Diagramme d'acteurs



3. Diagramme de cas d'utilisation Profil/Membre



4. Diagramme de séquence Oauth



Auteur du diagramme : @idangazit (http://twitter.com/intent/user?screen_name=idangazit)