Projet: Mashup

1 Introducción

Le projet consistera à appliquer des technologies de services web (`REST` et `SOAP`) et des techniques de conception en couches pour le développement d'une application web de type "`Mashup`" et d'un petit ensemble de services. Le terme "`Mashup`" désigne une application qui s'appuie sur un ensemble de services à distance (pas nécessairement des services web) pour fournir ses fonctionnalités. Les applications Web de type "mashup" font partie du "Web 2.0", qui désigne une série d'éléments caractérisant les nouvelles applications Web. L'utilisation de services `REST` est également une caractéristique du Web 2.0.

2 Intégration d'Applications Hétérogènes avec des Services Web

2.1 Présentation Générale

Pour définir le contexte du projet, nous supposerons que nous travaillons dans une entreprise qui, pour des raisons historiques, conserve ses informations clients dans deux `CRM` (*Customer Relationship Management*). Il y a quelques années, l'entreprise a décidé d'acheter un `CRM` pour gérer ses informations clients. Ce `CRM` est une application web installée sur l'une des machines de l'entreprise, et utilise une base de données (peut-être installée sur une autre machine) pour stocker les informations sur les clients.

Comme pour toute autre application web, les employés de l'entreprise accèdent à ce `CRM` à partir d'un navigateur. Le personnel du département informatique de l'entreprise est chargé d'administrer le CRM, ce qui implique une maintenance logicielle (par exemple, nouvelles versions et correctifs pour le `CRM` et la base de données, sauvegardes, etc.) et une maintenance matérielle (sur les machines sur lesquelles le `CRM` et sa base de données fonctionnent).

Aujourd'hui, Il existe des entreprises qui proposent des services `CRM` via Internet, comme *Salesforce* ou *SugarCRM*. Avec ces types de *CRM*, le *CRM* ne fonctionne pas en interne, mais plutôt dans l'entreprise qui offre le service. Telles entreprises offrent généralement une interface Web permettant aux employés de l'entreprise d'accéder aux fonctionnalités du `CRM` (équivalent d'un `CRM` classique) à partir d'un navigateur, ainsi qu'un ensemble de services Web permettant de créer des applications qui accèdent aux données des clients. Ces `CRM` libèrent les entreprises des coûts de maintenance des logiciels et du matériel associés aux `CRM` extérieurs à l'entreprise.

En particulier, notre société vient d'acquérir une autre société dans un domaine d'activité similaire. Cette société avait souscrit un service de CRM auprès de Salesforce, de sorte que les données clients de cette nouvelle société sont gérées par Salesforce.

Pour l'instant, l'entreprise constate qu'elle a des données clients dans les deux `CRM`. Pour certaines fonctionnalités (probablement nombreuses), il serait souhaitable de disposer d'une application offrant une vue unifiée des données clients, évitant ainsi aux employés d'avoir à utiliser explicitement les deux `CRM`, ce qui serait très fastidieux. En particulier, notre entreprise décide de construire une application web qui permet de récupérer les informations des clients (*Lead*). **Les informations relatives aux clients comprendront également la latitude et la longitude correspondant à l'emplacement géographique des clients**. Cette information n'étant disponible dans aucun des CRM, l'application utilisera le service de géolocalisation [1], qui permet d'obtenir la latitude et la longitude d'une adresse.

Ce type de client est très intéressant pour le personnel commercial de l'entreprise, qui voudra sans doute leur rendre visite pour tenter de les convertir en véritables clients. La Figure 1 présente l'architecture de l'application mashup à mettre en œuvre. Le code est structuré dans un module : *virtualcrm*.

Lorsque l'utilisateur exécute le client, on invoque l'opération `findLeads`(1) de l'interface `VirtualCRMService` (Figure 2), fourni par le module "virtualcrm". L'opération `findLeads` reçoit trois paramètres. Les deux premiers représentent la fourchette de revenus annuels que les clients doivent avoir ($mathttlowAnnualRevenue \leq$ `gain` $<$ `lowHighRevenue`), et le troisième le nom de la province dans laquelle ils se trouvent. L'opération renvoie des informations sur les clients correspondant à ces critères de recherche.

Pour chaque client (`leadTO`), nous retournons son nom, son prénom, la société pour laquelle il travaille, son revenu annuel attendu, son numéro de téléphone, sa rue, son code postal, sa ville, son département et son pays, la date de son enregistrement dans le CRM et sa position géographique (`GeographicPointTO`). Si le service de géolocalisation n'est pas en mesure de renvoyer la position d'un client, l'attribut `geographicPointTO` doit être `null`.

Le client maintient une seule instance (Singleton) de la classe implémentant l'interface `VirtualCRMService`, qui peut être obtenue via `VirtualCRM Factory.getVirtualCRMSErvice()`. L'implémentation du `VirtualCRMService` peut maintenir un état global. Dans ce cas, l'implémentation de cette interface doit initialiser l'état dans le constructeur et ne doit jamais le modifier pendant l'exécution des opérations de l'interface.

L'implémentation de l'interface `VirtualCRMService` doit invoquer les services de recherche du `CRM` interne (2) et de `SalesForce`(3) pour obtenir les données des clients potentiels, et le service de géolocalisation (4) pour obtenir leur position géographique. **l'interface doit retourner les informations des clients, triées par profit potentiel (du**

1. A définir

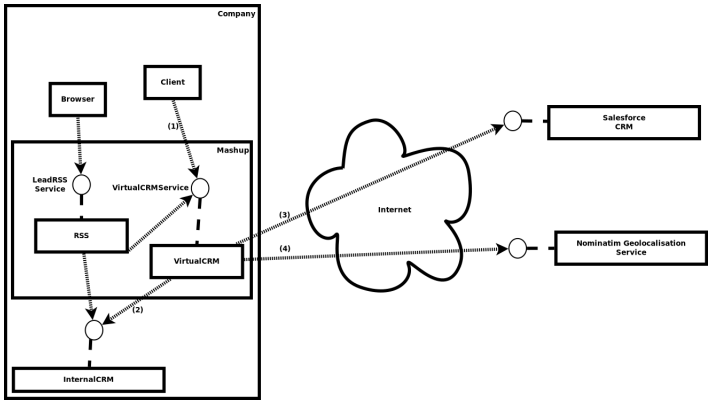

Figure 1. Architecture d'une application `mashup`

.

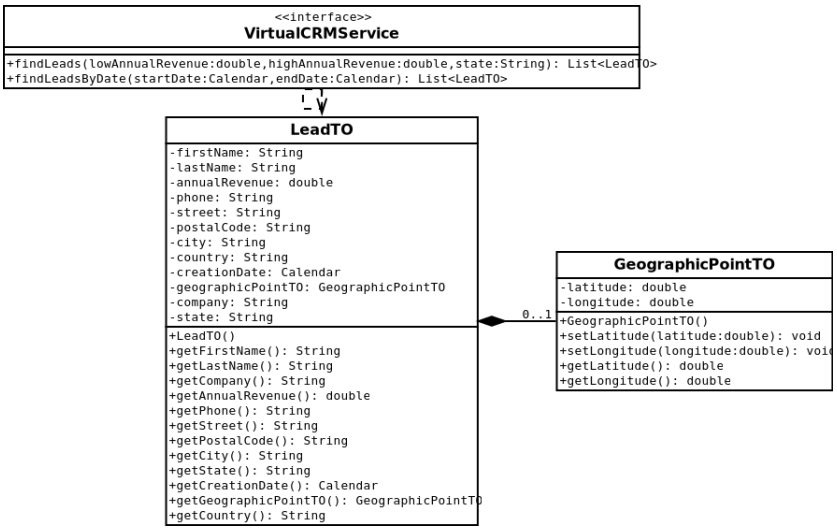

Figure 2. Interface `VirtualCRMService`

.

plus élevé au plus bas). Le client imprimera le résultat de la requête directement sur la console. En pratique, elle devrait mettre en œuvre :

- Une implémentation **fictive** du service de recherche `CRM` interne (le module dénommé "*internalcrm*" sera ajouté à la distribution source).
- L'interface `VirtualCRMService` (module "*virtualcrm*"), utilise une architecture en couches qui cache les API réelles des services invoqués (le `CRM` interne, les service de recherche Salesforce, et le service de géolocalisation). Cette architecture en couches facilite la mise en œuvre de l'interface `VirtualCRMService` (la classe de mise en œuvre peut travailler avec des `API` plus simples que celles des services) et minimise l'impact sur le logiciel lorsqu'on décide de remplacer un service par un autre (il existe plusieurs API de géolocalisation disponibles : google maps, yahoo ! ou openstreetmap)
- Un service qui renvoie des informations récentes sur les clients à partir d'un lecteur `RSS` [4] (par exemple firefox, iexplorer, thunderbird, outlook, etc). Un nouveau module appelé `leadnews` sera ajouté aux sources

Les sections suivantes précisent en détail les fonctionnalités à mettre en œuvre dans chacune des composantes du projet.

2.2 Service de recherche CRM interne

Un service `SOAP` sera mis en œuvre. Ce service fournit une opération de recherche qui renvoie des informations sur les clients gérés par le `CRM` interne. Pour simplifier, l'opération sera similaire à l'opération `findLeads` de l'interface `VirtualCRMService` (Figure 1), c'est-à-dire qu'à partir de l'intervalle de profit annuel (en utilisant deux paramètres) et du nom de la province, elle renvoie les informations des clients qui correspondent à ces critères de recherche.

Une particularité à mentionner est que le service `CRM` interne retournera le nom et le prénom de chaque contact dans un seul champ suivant le format Nom, Prénom (par exemple Laporte, Aymeric). Notez que la classe "`LeadTO`" du module "`virtualcrm`" a deux champs différents pour le nom et pour le prénom. De plus, pour supporter le service `RSS`, le service `SOAP` fournira également une autre opération qui renvoie les clients ajoutés entre deux dates.

Dans un cas réel (par exemple Salesforce), l'opération de recherche serait probablement plus générale, et donc plus compliquée à utiliser. Pour simplicité, la mise en œuvre du service peut disposer d'une liste de clients avec des informations fictives, qui, dans un cas réel, proviendraient d'une (par exemple) base de données

2.3 Service de recherche de Salesforce

Salesforce (`https://www.salesforce.com/`) offre à ses clients un service Web afin que `VirtualCRMService` puisse facilement récupérer les informations contenues dans son `CRM` en ligne.

Salesforce propose deux types d'`APIs` : `SOAP` [6] et `REST` [5]. **Le groupe de développeurs peut choisir celui qu'il veut utiliser.**

Afin de faciliter le développement et le test des applications qui accèdent à leurs données, `Salesforce` permet aux développeurs de créer des **Developer Account** identiques aux comptes réels mais contenant des données fictives. Dans la pratique, nous allons accéder aux données d'une entreprise fictive créée de cette manière. Pour ce faire, chaque groupe créera son propre compte de développeur chez `Salesforce`.

Le modèle de données de `Salesforce` se compose d'un certain nombre d'entités de données, chacune d'entre elles contenant divers types d'informations. Dans notre cas, nous utiliserons l'entité "`Lead`" qui contient diverses données sur les clients potentiels. Parmi ces données, on trouve les différents éléments qui composent leur adresse tels que : la rue, l'État, le code postal ou le pays, ainsi que le chiffre d'affaires annuel de l'entreprise à laquelle appartient le contact (`AnnualRevenue`).

Pour interroger les données d'une entité, il est nécessaire d'utiliser la méthode `query` de l'`API` du service Web. Cette méthode reçoit parmi ses paramètres une expression de requête écrite dans un langage appelé `SOQL` [7], qui est une version très simplifiée de `SQL`. Avant d'invoquer la requête, il est nécessaire de s'authentifier auprès du service en utilisant la méthode logique de l'`API`.

Dans `https://developer.salesforce.com/` on peut y trouver divers exemples et ressources d'intérêt.

2.4 Service de géolocalisation

Plusieurs services web de géolocalisation sont disponibles sur Internet [1,2]. Pour certains d'entre eux, comme `Google Maps`, il faut une `APIKey` et en plus il suive un modèle de tarification de type *pay-as-you-go*.

L'API Nominatim [2] de `OpenStreetMaps` ne nécessite aucun type d'enregistrement et fournit un service `REST` qui prend une adresse comme paramètre et renvoie `latitude` et `longitude` dans des format `JSON` ou `XML` entre autres. Par exemple, si nous voulons obtenir la localisation de la `UFR` (Figure 3) des sciences de l'université d'angers en utilisant l'`API` *Nominating search* [3], cela peut être fait en appelant la `URL` suivante (voir [3] pour plus de détails)

`https://nominatim.openstreetmap.org/search?<params>`

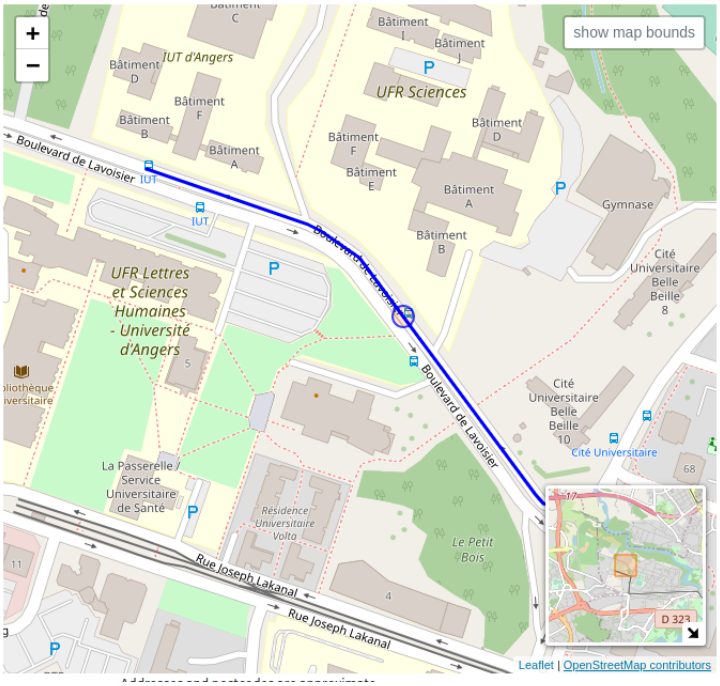

Figure 3. OpenStreetMap visualisation of the UFR of Sciences.

`https://nominatim.openstreetmap.org/search?city=angers&country=france&postalcode=49100&street=2+boulevard+de+lavoisier&format=xml&limit=1`

et analysant la réponse. Cela peut être reçu dans différents formats, parmi lesquels `JSON` et `XML`. **Le choix du format de réponse à utiliser est laissé à chaque équipe**. Dans le cas de l'exemple, nous obtenons le `XML` (simplifié) suivant :

```
<searchresults timestamp="Wed, 14 Sep 22 18:35:30 +0000" .... >
      <place ... lat="47.4795093" lon="-0.6003698" .../>
</searchresults>
```

La réponse doit être analysée afin d'en extraire la **latitude** et la **longitude**.

2.5 Service RSS

Un *parser* de XML sera utilisé (dans le cours nous avons vu JDOM) pour la génération du XML nécessaire à la création d'un service `RSS` nommé *last potential clients*. A partir des données des clients, on va créer un *feed* appelé "Last potential clients', avec le lien `http://www.acme.com/potentialclients.rss` et une description appropriée. Pour chaque nouveau client (ajouté depuis moins de 24 heures), un article `RSS` sera généré avec les éléments suivants :

— **title**. Nom et prénom du contact, concaténés avec le nom de la société, séparés par le caractère "-" (par exemple, Aymeric Laporte - Acme consulting).
— **description**. Concaténation de tous les champs du contact au format '`nom` :`valeur`, séparés par le caractère '-'.
— **pubDate** : date d'enregistrement du client.

Pour plus d'informations sur le RSS, consultez [4, Section *Aspects techniques*].

3 Normes et évaluation

3.1 Groupes

Le projet se fera en groupe de 2 personnes (de préférence) ou individuellement.

3.2 Normes de Codage

Afin d'écrire un code de qualité et facilement lisible, on suivra un système de codage commun, qui définit des règles pour nommer les classes, les attributs et les méthodes, des règles d'identification, etc.

Cela permet à une équipe de développement de conserver l'aspect du code, quel que soit le développeur qui l'a écrit, ce qui facilite la maintenance. La norme de codage [8] sera utilisée pour le projet. Les exemples vus dans le cours suivent ce système de dénomination.

Enfin, il ne sera pas nécessaire de documenter les classes (par exemple, JavaDoc).

3.3 Dépôt de la Version Finale

Une copie du projet (contenant toutes les applications demandées) au format `.tar.gz` ou `.zip` avec les sources de l'application et un mémoire au format `.pdf` seront déposés sur `moodle`. Concernant les sources du projet, le fichier compressé doit contenir un projet qui peut être compilé à l'aide de `maven` ou `gradle` via la ligne de commande. Ce projet doit contenir les sources et les fichiers de configuration mais il n'est pas nécessaire d'ajouter d'autres types de fichiers tels que les `.class` ou les `.war`.

3.4 Format du Rapport

Le rapport contiendra les parties suivantes :

En règle générale, les diagrammes doivent utiliser la notation `UML`. Les diagrammes doivent être de qualité (et non réalisés par rétro-ingénierie ; la conception précède la mise en œuvre !) et être expliqués brièvement mais clairement. Il est important de noter qu'il ne s'agit pas d'un document volumineux, mais d'un document qui explique brièvement mais clairement chacune des sections décrites ci-dessous. La qualité du rapport sera essentielle pour l'évaluation du projet.

1. **Conception**
   (a) **Architecture globale** : expliquez brièvement les modules qui ont été conçus et mis en œuvre. Deux diagrammes UML de "haut niveau" seront utilisés pour étayer l'explication.
      — Un diagramme de classes illustrant les **principales** abstractions (interfaces, *factories*, classes d'implémentation des interfaces, etc.) des différents modules impliqués dans l'implémentation de l'interface `VirtualCRMService`. Ce diagramme ne devra pas montrer les noms des opérations et des attributs dans les interfaces et les classes. Elle a pour but d'illustrer les dépendances entre les principales abstractions.
      — Un diagramme de séquence illustrant le traitement d'une invocation de l'opération `findLeads` de l'interface `VirtualCRMService`, en termes d'abstractions présentées dans le diagramme de classes ci-dessus.
   (b) **i-ème module** : Pour chaque module, il est nécessaire d'écrire une section expliquant sa conception. Cette explication doit inclure :
      — La structure globale du paquet du module. Il n'est pas nécessaire d'utiliser la notation UML. Il n'est pas non plus nécessaire de montrer tous les paquets, mais seulement les paquets de premier niveau qui reflètent l'architecture du module et ses principaux sous-paquets. Pour chaque paquet, une brève explication de ce qu'il contient doit être donnée.

— Un ou plusieurs schémas illustrant l'architecture du module. En particulier, les diagrammes spécifient en détail les opérations des interfaces et des classes por `transferir les objets` (*transfer object*). Pour le reste des abstractions (classes concrètes), il sera seulement nécessaire de spécifier les opérations et les attributs qui sont considérés comme pertinents pour comprendre l'explication du module (par exemple, un attribut important pour comprendre l'explication d'une interface). En d'autres termes, les diagrammes UML ne doivent pas rassembler toutes les abstractions mises en œuvre, mais seulement celles qui sont pertinentes et, pour chacune d'entre elles, ne montrer que ce qui est nécessaire pour comprendre la conception du module.

2. **Compilation et Installation de l'Application**
   Il doit être clairement expliqué comment compiler et installer la pratique, en supposant un environnement correctement configuré (par exemple, serveur d'application installé et configuré, paquets de logiciels installés dans les mêmes répertoires que dans le laboratoire, etc. Les instructions d'installation doivent être aussi simples que possible. En particulier, elles devraient préciser :
   — Comment décompresser la distribution source
   — Tout aspect particulier de la configuration qui doit être mis en évidence
   — Le projet doit pouvoir être exécuté en utilisant soit `gradle` ou `maven`.
   — **Lors de la correction de la version finale de chaque application, les instructions d'installation seront suivies**
3. **Problèmes connus** : liste des *bugs/erreurs* connus dans le code.

3.5 Itérations et Livraisons

Pour la réalisation de chaque application, une approche par itérations sera suivie. Chaque itération incorpore plus de fonctionnalités que la précédente, jusqu'à ce que la dernière itération se termine par un logiciel qui met en œuvre toutes les fonctionnalités. En particulier, chaque mise en œuvre sera effectuée en deux itérations.

— **La correction de la première itération ne donnera pas lieu à une note associée et il ne sera pas nécessaire de soumettre un rapport.**
— Il suffira de montrer les diagrammes liés à cette itération en utilisant un outil de conception UML.
— L'objectif de cette première itération est d'essayer de détecter les erreurs majeures et, le cas échéant, de guider l'apprenant vers leur résolution.
— Lors de la deuxième itération, la pratique sera achevée.
— **Lors de la deuxième itération, une note sera attribuée et le code et le mémoire devront être livrés via moodle.**.

3.6 Évaluation

L'évaluation du projet prendra en compte :
— Son bon fonctionnement
— La qualité de la conception
— La qualité de la mémoire
Un projet copié signifie un échec pour le groupe qui a autorisé la copie et pour le groupe qui a copié, aucune distinction ne sera faite.

3.7 Dates importantes

— **1ère itération :** `10/10/2022`
— **Dépôt de la version finale :** `31/10/2022`
— **Défense :** En contactant l'enseignant après le `31/10/2022`

Références

1. Google geocoding api, `https://developers.google.com/maps/documentation/geocoding/overview`
2. Nominatim : Open street map geocoding, `https://nominatim.org/release-docs/develop/api/Overview/`
3. Nominatim : Open street map geocoding, `https://nominatim.org/release-docs/develop/api/Search/`
4. Rss, `https://fr.wikipedia.org/wiki/RSS`
5. Salesforce rest api developer guide, `https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/intro_rest.htm`
6. Salesforce soap api developer guide, `https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_quickstart_intro.htm`
7. Soql, `https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql.htm`
8. Java code conventions (1997), `https://www.oracle.com/technetwork/java/codeconventions-150003.pdf`