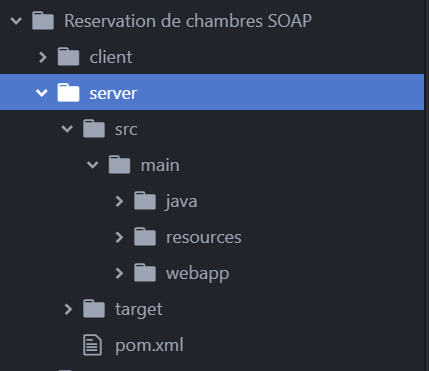
# TP Web Service sécurisés

# Partie I

### 1 – Mise en place de l’application SOAP

**1a**. Nous avons commencé par créer un répertoire « ***Reservation de chambres SOAP*** » pour la mise en place de la partie API SOAP. Ce répertoire est divisé en deux sous-répertoires « ***server*** » et « ***client*** ».  
A l’aide de Maven en ligne de commande dans le répertoire « ***server*** », nous avons généré l’arborescence et les fichiers de base du projet de type webapp.



Pour cette partie, nous avons décidé de créer notre programme Java « à l’ancienne » avec notre éditeur Atom et les commandes ***javac*** et ***java*** pour compiler et exécuter le code.   
Nous utiliserons un dépôt Github et un Git en ligne de commandes pour gérer le partage du code.

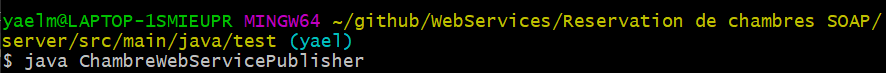
**1b**. Nous avons créé nos deux POJO représentant les objets principaux pour la gestion des réservations de chambres d’hôtel : ***chambre*.java** et ***reservation.java.***

L’interface ***ChambreWebService*** constitue la base de notre WebService. Nous l’avons implémentée dans la classe ***ChambreWebServiceImpl***. C’est dans cette classe que figureront les Web Methods utiles à l’API.

Nous avons ensuite codé les fonctions utiles au système de réservation.

**1c**. Afin de ne pas utiliser d’application tierce et tout centraliser dans notre code java, nous avons décidé de publier notre Web Service sur le réseau via l’objet Endpoint.   
Pour cela, nous avons créé la classe ***ChambreWebServicePublisher***, constituant le main de notre projet.

Le serveur est prêt. Pour le lancer, il suffit de se rendre dans le répertoire ***server/src/main/java/test*** et de lancer le programme java compilé à l’aide de ***java***:



Recompiler la classe à l’aide ***javac*** si nécessaire :



Le serveur tourne.

**1d**. Nous avons ensuite testé notre serveur en accédant au WSDL. Le fichier XML s’affiche bien dans notre navigateur.

Pour y accéder, saisissez dans votre navigateur favori cette URL : <http://localhost:10000/ReservationVoyage?wsdl>

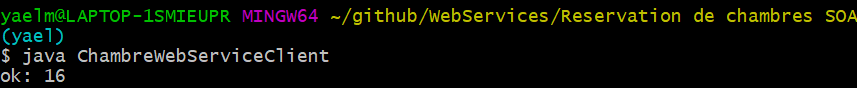
### 2 – Construire un client Java SOAP à partir du WSDL

2a. Nous restons avec notre simple éditeur Atom et notre java en ligne de commandes. Nous travaillons dans le répertoire « **Reservation de chambres SOAP/client** « .

2b. Avec Maven, nous générons les classes clientes dans un package « client » à partir du WSDL à l’aide de la commande suivante :

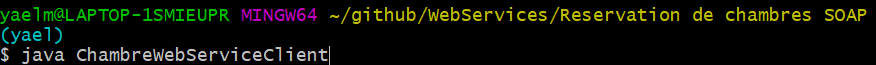


2c. On code notre classe cliente principale qui devra accéder au Web Service créé précédemment. Il s’agit de la classe ***ChambreWebServiceClient*** dans le package ***client***. On compile et exécute cette classe. Afin de tester simplement, nous avions créé une fonction très sommaire dans notre Web Service renvoyant « ok » suivant d’un numéro de chambre pour valider la connexion.

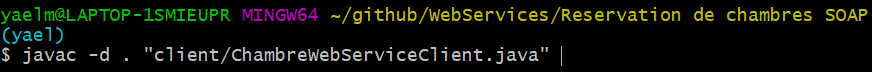


Le client affiche bien le résultat de cette fonction, tout semble fonctionner.

Afin de lancer le client, se placer dans le répertoire « ***Reservation de chambres SOAP*** » et exécuter la classe compilée :



Recompiler si nécessaire avec ***javac*** :



### 4 – Mise en place de l’application REST

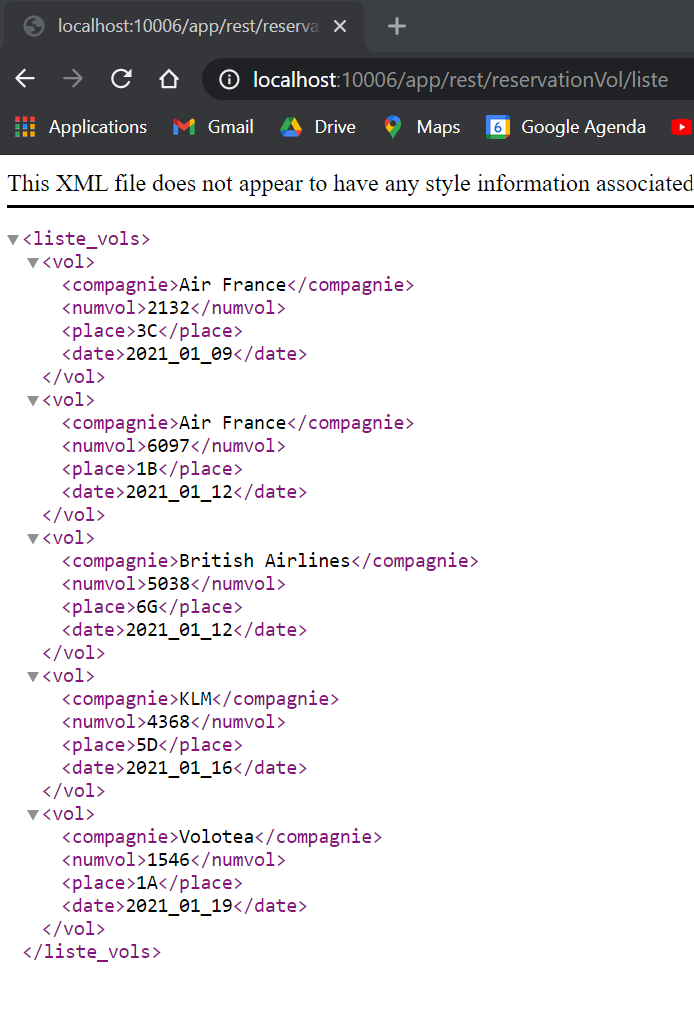
**4a**. Cette fois-ci, nous utilisons un IDE (Eclipse), ce qui nous facilitera la tâche pour crée notre application REST. Nous créons un nouveau projet Maven de type Webapp.

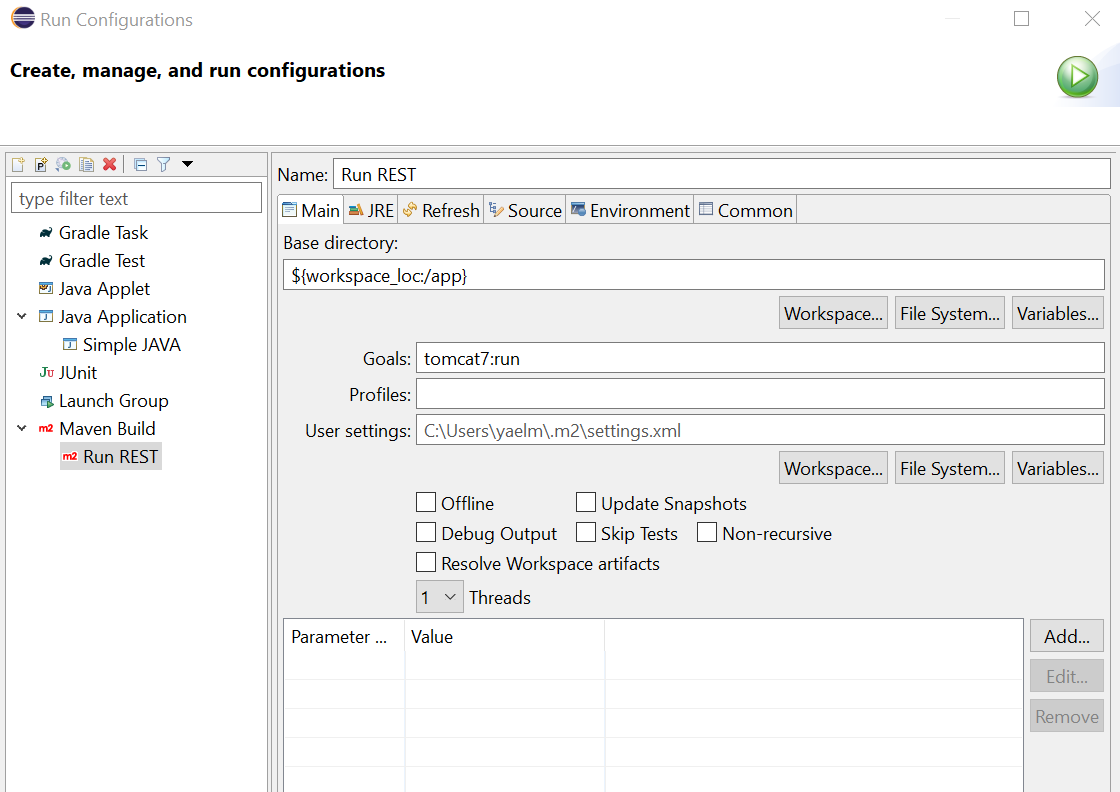
Le contenu de cette application figure dans le répertoire « ***Reservation de vols REST/Application REST*** » .

Dans le fichier ***pom.xml***, nous ajoutons les dépendances utiles à l’application : le servlet javax, Jersey, Json ainsi que le plugin Tomcat pour Maven, qui nous servira de serveur http.

**4b 4c.**

Le service rest est accessible à <http://localhost:10008/app/rest/reservationVol/liste>

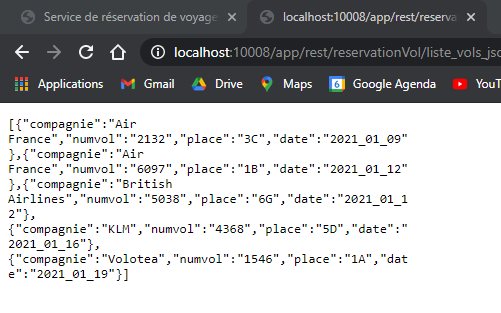
  
La fonction appelée renvoi tous les vols sous forme de XML.

Comment l’exécuter : Avec Eclipse, ouvrir le projet et exécuter le code à l’aide du configuration Maven. 

Le Base Directory est la racine du projet (répertoire app).   
Se placer ensuite sur le fichier VolWebService.java et faire exécuter.

4d. Passage en JSON

Le fonction renvoyant tous les vols, mais cette fois-ci en JSON est accessible à <http://localhost:10008/app/rest/reservationVol/liste_vols_json>



4e. Le code source du client SOAP fonctionne séparément (voir partie 2) mais nous n’avons pas réussi à l’incorporer dans ce projet Maven

### 5 – Mise en place de l’application JavaScript

Cette fois-ci dans le répertoire « ***Application JavaScript***», ouvrir le fichier ***index.html*** avec votre navigateur favori. Le code JS y est inclus dans la balise ***<script>.*** Ce code récupère le JSON via requête http (voir partie 4) et affiche les données des objets formés à partir du JSON.

Note : dans la partie 4 (application REST), nous renvoyons les résultats des requête via ***Response***:   
en ajoutant dans le header « ***Access-Control-Allow-Origin*** », ce qui évite d’obtenir une erreur de sécurité côté JS au moment de récupérer le JSON.



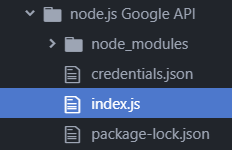
# Partie II

### 1 – Mettre en place une délégation d’autorisation Oauth 2.0 auprès d’un fournisseur d’API public (Google, Facebook, Twitter…)

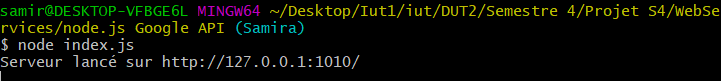
Nous avons décidé d’inscrire notre application consommatrice REST auprès du fournisseur d’API google. Nous avons choisi ce fournisseur d’API car il est très complet et nous permet de mettre en place le processus Oauth 2.0.



Nous avons développé notre application REST en Node.js. Celle-ci dispose d’un fichier index.js qui fournit une page web avec un lien de connexion vers un compte google, d’un fichier credentials JSON comportant l’ID client ainsi que le secret et de « node modules » regroupant les librairies google API.

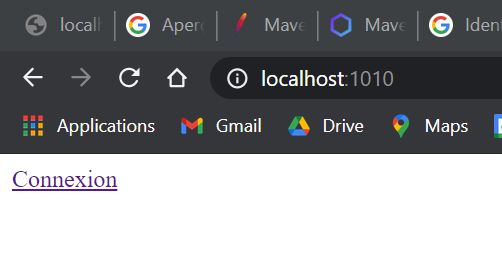


Pour accéder à la page web, nous devons tout d’abord exécuter le fichier index.js sur un terminal pour faire tourner le server.

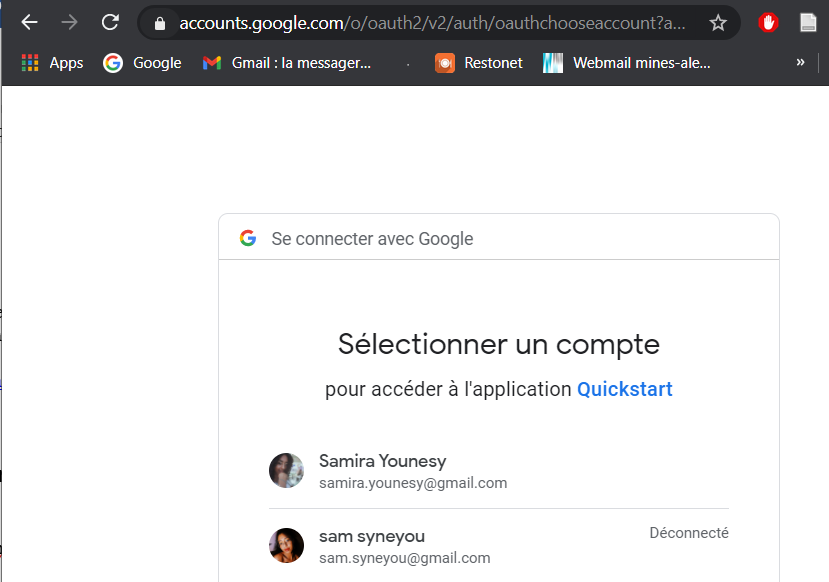


Une fois le server lancé, on peut accéder à notre page web avec l’URL <http://localhost:1010>

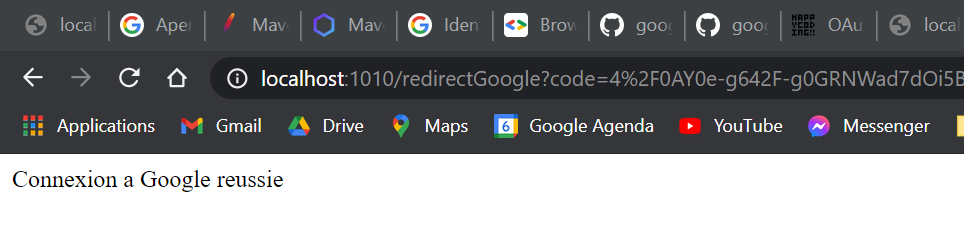
Nous arrivons sur une page Web avec un lien ‘Connexion’.



Après avoir cliquer sur le lien « Connexion », on dirige sur une page où l’on doit choisir un compte google.



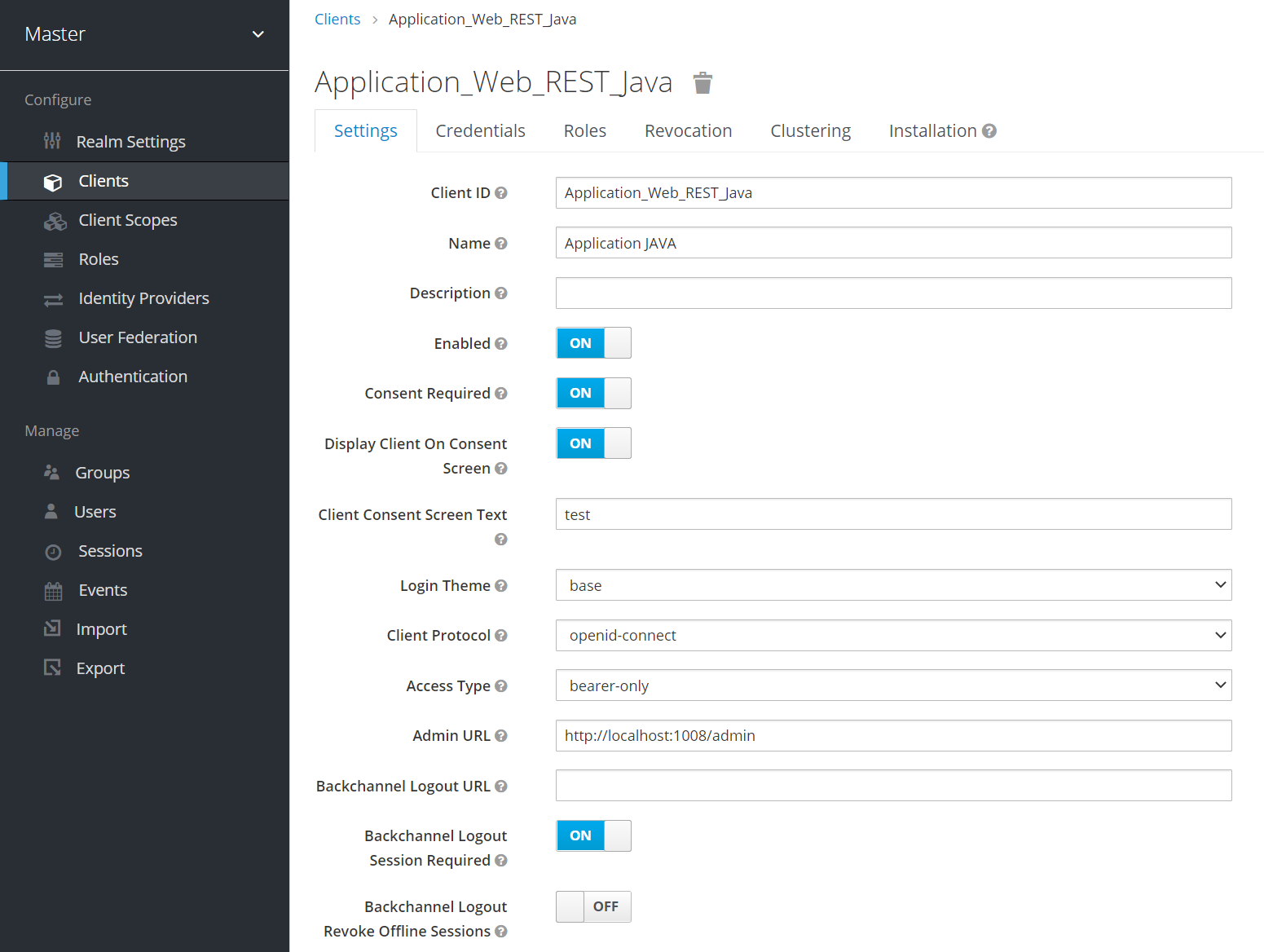
Après avoir choisis le compte google, on nous redirige vers une autre page sur laquelle il est ecrit « Connexion a Google reussie »



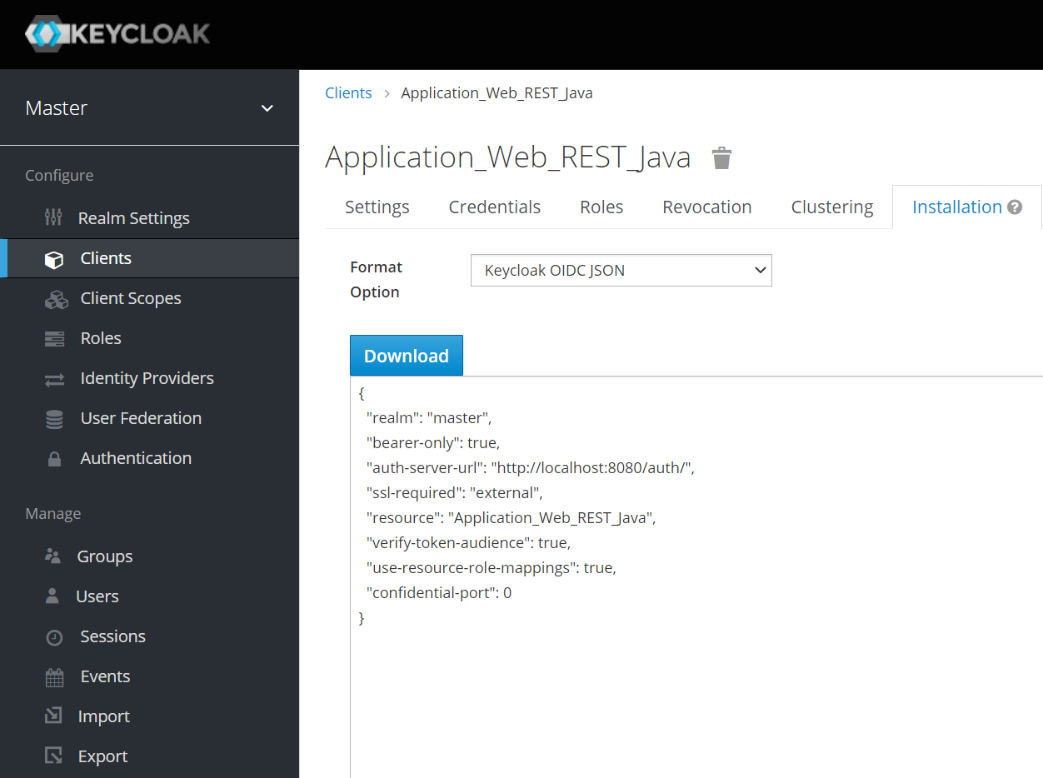
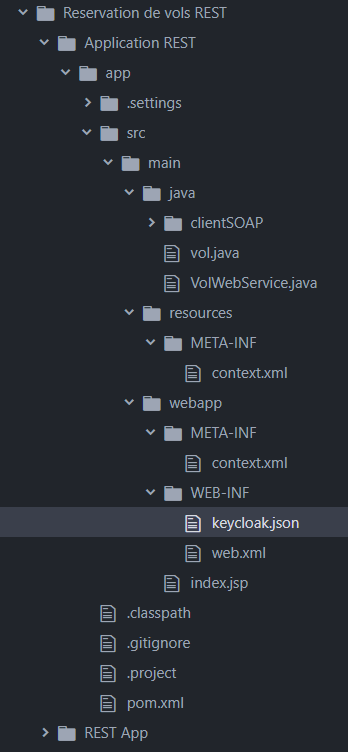
Après connexion, google-nous renvoie une URL de redirection avec en paramètre un code. En échange des « credentials » et de ce code, google-nous renvoie un token.

### 2 – Déléguer l’autorisation d’accès à l’API REST auprès d’un serveur OpenId Connect

Nous avons installé Keycloak localement. Une fois ce server installé, nous avons configuré notre application REST en tant que client de type « bearer only ». Nous avons récupéré l’application REST de la partie I du TP que nous avons développé en java. L’adapter choisis pour l’application REST est Tomcat. Le port 10008 a été choisis pour l’admin URL.



Après configuration de notre client, Keycloak nous génère un fichier JSON pour configurer le java adapter. Nous avons ensuite récupéré ce fichier JSON afin de l’inclure dans notre application REST.



Lorsqu’on lance notre server et que nous accédons ensuite à l’URL <http://localhost:10008/app/rest/reservationVol/liste>, nous pouvons observer que notre API est bien sécurisée car nous avons un retour 401.

