Systèmes d'exploitation
Programmation réseaux

TP 5 - Mise en place d'une architecture client-serveur Java classique avec sécurité logicielle hybride symétrique et asymétrique.

Liège, Avril 2015
HEPL - Ingénieur Industriel

Thibault Reinders
Nassim Rahali
Kaoutare Ahallouch

M 18

Professeur : C. Vilvens

Table des matières

1 Méthodologique de travail **2**

2 Bank Server **2**

3 Bank Client **3**

4 Agent Bancaire **3**
4.1 Vers Bank Server 3
4.2 Vers Data Server 4

5 Data Server **4**

6 Web Server **4**
6.1 Controller 4
6.2 JSP 4
6.3 Configuration HTTPS 5
6.4 Quelques exemples 5

7 AS & TGS **5**

1 Méthodologique de travail

Nous nous sommes tout d'abord réparti le travail selon les 3 serveurs à faire (Bank, Data et Web).

- Bank Server : Thibault
- Data Server : Kaoutare
- Web Server : Nassim

Ensuite, en fonction de l'avancement de chacun dans son serveur, nous avons fait le reste du travail comme suit

- Kerberos (AS & TGS) : Nassim et Thibault
- Bank Client : Thibault
- Agent Bancaire : Kaoutare et Thibault (chacun la partie utilisant son serveur.)

Pour faire les 3 serveurs, nous n'avons pas eut besoin de nous mettre d'accord puisqu'il n'y a pas d'interactions directes entre les différents serveurs, hormis le Web Server qui devait avoir un accès direct aux bases de données des deux autres serveurs.

Pour l'Agent Bancaire, les interactions entre la partie utilisant le Bank Server et celle utilisant le Data Server n'est pas très importante et nous n'avons pas eut de problème pour développer les 2 en parallèle.

Pour l'AS et le TGS, il nous a fallu travailler ensemble et communication par communication pour ne pas se tromper dans qui chiffre quoi et avec quelle clé.

2 Bank Server

Le serveur a 2 tâches à remplir. Il communique avec les clients Bank Client et Agent Bancaire. On utilise donc 2 *ThreadPoolExecutor* pour qu'un trop grand nombre de clients d'un type ne bloque pas l'autre type de client. Logiquement, on donnera plus de threads au pool destiné à communiquer avec les Banques Clientes qu'à celui pour les Agents Bancaires.

Le serveur a accès a une base de données *MongoDB* qui contient les banques clientes et leurs demandes d'opérations de crédit et de débit avec notre banque. Typiquement

```
{
  "_id" : 1,
  "type" : "debit",
  "montant" : 1000,
  "valide" : "false",
  "date" : "2015-04-15"
  "banque" :
  {
    "_id" : 1,
    "name" : "Belfius"
  }
}
```

La communication avec les Banques Clientes se fait par SSL, on a donc créé des clés (vérifiées par un CA) enregistrées dans un *keystore*, côté client et côté serveur. Pour ne pas alourdir le code, la création d'une socket SSL a été mise dans un petit jar.

Par défaut, les demandes de crédit ou de débit des Banques Clientes sont à *false*. Il faudra l'intervention d'un Agent Bancaire pour les valider.

L'Agent Bancaire peut demander à voir toutes les opérations ou alors les filtrer selon la Banque ou le type d'opération, pour ensuite en sélectionner une ou plusieurs et les valider.

3 Bank Client

La Banque Cliente communique donc par SSL avec son serveur. Lors de la connexion, le client doit choisir une banque partenaire de notre banque sous peine d'être refusé. Il peut ensuite faire des demandes de débit et/ou de crédit ou vérifier l'état d'une opération appartenant à sa banque.

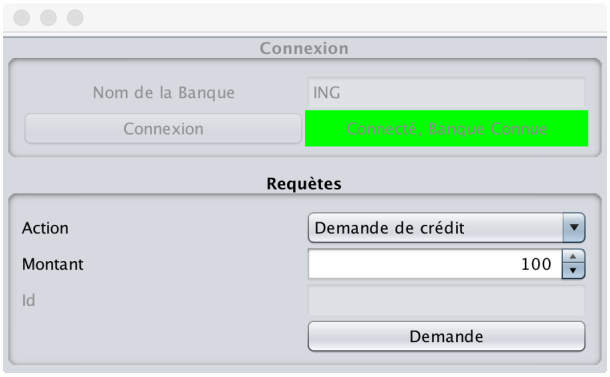

FIGURE 1 – Application Banque Cliente

4 Agent Bancaire

4.1 Vers Bank Server

L'Agent Bancaire se connecte au Bank Server et s'authentifie au moyen de l'architecture Kerberos. Il peut demander à voir l'ensemble des opérations, ou alors filtrer celles-ci par *Banque* et *Etat*.

Pour valider une ou plusieurs opérations, l'Agent Bancaire va envoyer la liste des *ids* à valider et le serveur va se charger de faire les updates correspondants sur la base de données.

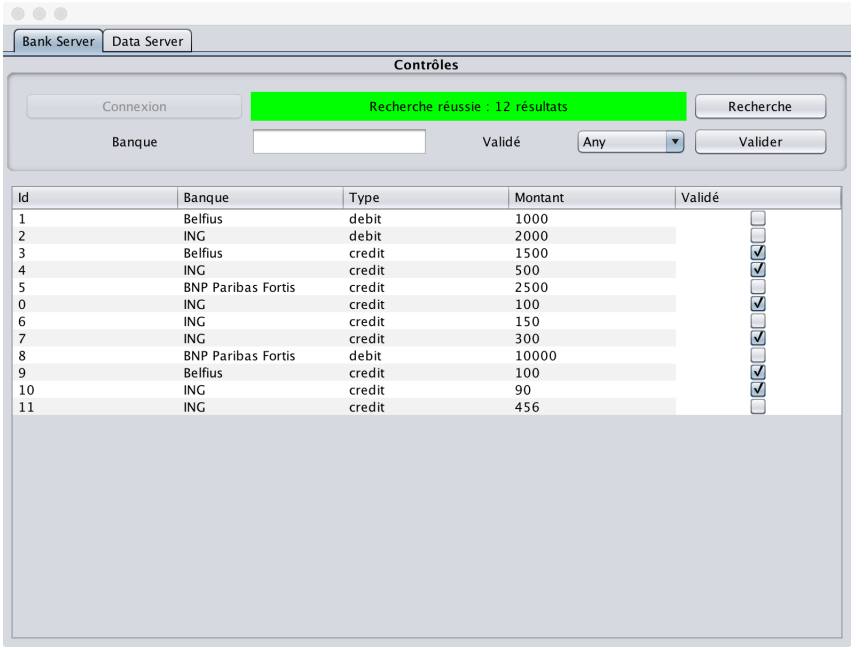

FIGURE 2 – Agent Bancaire (Bank Server)

4.2 Vers Data Server

5 Data Server

6 Web Server

6.1 Controller

Nous avons commencé par développer le controlleur de l'architecture MVC. Il se nomme `BanqueAdminController`. Ce servlet ne va faire qu'aiguiller vers le JSP qui correspond à la requête effectuée par le client. À chaque fois qu'une requête concernant un graphique généré via JFreeChart sera effectué, celui-ci sera enregistré dans un dossier sur le serveur. Comme la page JSP fournie, dans ce cas, contient une balise image ayant pour source le controller `BanqueAdminController` et qu'une session est créée permettant de savoir quelle image fournir au client, celui-ci recevra (via un GET du navigateur) le graphique.

6.2 JSP

Plusieurs JSPs ont été créés. Un scénario standard serait :

1. Le client arrive sur la page `index.jsp`.
2. Il renseigne un login et un mot de passe.
3. Si ceux-ci sont corrects, il va arriver sur la page `welcome.jsp`
4. Sur `welcome.jsp`, il pourra effectuer son choix (consulter une moyenne, avoir une répartition des crédits, ...).
5. Il sera ensuite envoyé sur une page qui correspond à son choix. Éventuellement, une image sera chargée correspondant à la requête effectuée.

Notons l'existence de la page `error.jsp` fournie si l'utilisateur n'est pas authentifié.

6.3 Configuration HTTPS

Nous avons configuré Apache Tomcat pour fonctionner en HTTPS avec keystore.

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
clientAuth="false" sslProtocol="TLS" keystoreFile="web.keystore"
keystorePass="cisco123" />
```

6.4 Quelques exemples

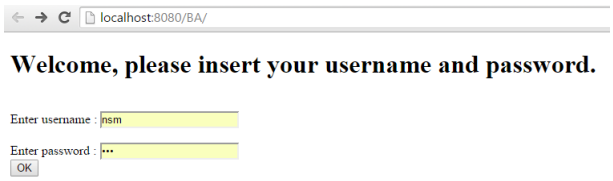

FIGURE 3 – index.jsp

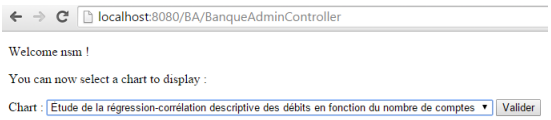

FIGURE 4 – welcome.jsp

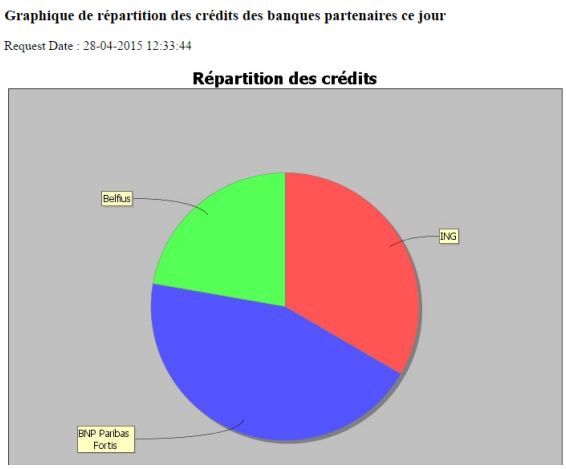

FIGURE 5 – chart.jsp

7 AS & TGS

C'est la partie qui a été la plus 'délicate' à implémenter. En effet presque toutes les applications de l'architecture sont impactées et il faut travailler en groupe et étape par étape pour être sûr de ne pas chiffrer la mauvaise information ou avec la mauvaise clé pour que le suivant puisse bien décrypter les données.

Point de vue technique, on notera juste l'utilisation de *SealedObject* pour encapsuler des objets chiffrés puisqu'on ne peut pas simplement crypter un objet et en récupérer les bytes.