

Secteur Tertiaire Informatique
Filière « Etude et développement »

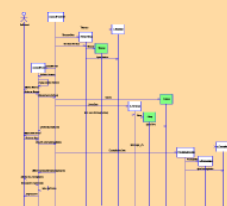
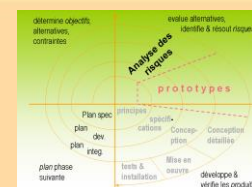
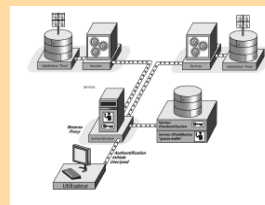
Séquence « Construire une application organisée en couches en mettant en œuvre des Frameworks »

Sécuriser les couches d'une application N Tiers

Apprentissage

Mise en situation

Evaluation



Version	Date	Auteur(s)	Action(s)
1.0	13/10/16	Lécu Régis	Création du document

TABLE DES MATIERES

Table des matières	2
1. Introduction	5
2. JAAS (<i>Java Authentication and Authorization Service</i>)	6
2.1 Introduction	6
2.2 Principes de JAAS	6
2.3 La phase d'authentification.....	7
2.3.1 Commentaire : création du <i>LoginContext</i>	7
2.3.2 Commentaire : appel de la méthode <i>login</i> du <i>LoginContext</i>	8
2.3.3 Résumé	8
2.4 La phase d'autorisation	9
2.4.1 Principe	9
2.4.2 Scénario de l'autorisation.....	10
2.4.3 Exemple du tutoriel.....	10
3. Authentification et autorisation dans une application N Tiers.....	10
3.1 Sécurisation d'une application Web	11
3.2 Sécurisation d'une application N Tiers	12
4. Les protocoles SSL/TLS.....	15
4.1 Présentation des protocoles SSL/TLS.....	15
4.2 Mise en oeuvre en Java	15
4.2.1 Création du certificat.....	15
4.2.2 Codage de la partie serveur.....	16
4.2.3 Codage de la partie client	16

Objectifs

A l'issue de cette séance, le stagiaire sera capable de :

- En partant de la conception d'une application N Tiers sécurisée, mettre en œuvre les mécanismes disponibles selon les technologies et les bonnes pratiques de sécurité dans chaque couche, pour assurer la sécurité globale de l'application.
- Sécuriser la communication entre les couches (certificats, protocoles sécurisés)

Pré requis

Cette séance suppose connu le développement objet en Java, dans une architecture N Tiers (en particulier JSP et EJB).

Méthodologie

Ce document peut être utilisé en présentiel ou à distance.

Il précise la situation professionnelle visée par la séance, la resitue dans la formation, et guide le stagiaire dans son apprentissage et ses recherches complémentaires.

Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

Ressources

Sur la sécurisation d'une application Web ;

Tutoriel : *Securing a Web Application in NetBeans IDE*

<https://netbeans.org/kb/docs/web/security-webapps.html>

Sur **JAAS** (*Java Authentication and Authorization Service*):

Tutoriel Oracle : <http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html>

Sur la sécurité en Java :

- *Oracle Java EE Tutorial*, Part X. Security :

<https://docs.oracle.com/javaee/7/tutorial/partsecurity.htm>

- Tutorial Java de Jm Doudoux : chap. 29 sur la sécurité

1. INTRODUCTION

Cette séance fait partie du projet **CyberEdu**, qui prend en compte la sécurité dans tout le cycle de vie du logiciel, et dans toutes les couches des applications.

La situation professionnelle visée concerne différentes phases du développement, où chaque développeur travaille en général seul sur les composants logiciels et les couches qui lui sont attribués, et pour lesquels il doit développer des tests unitaires. Le développeur code en adoptant une approche sécurité :

- il s'appuie sur le dossier de conception sécurisée de l'application ;
- il applique les règles générales du développement sécurisé ;
- il utilise des bibliothèques spécifiques pour sécuriser la communication entre les couches.

Cette séance va préciser les bonnes pratiques du développement sécurisé dans le cadre des applications N Tiers, et fournir les moyens techniques nécessaires :

- les principes de sécurité sont communs à toutes les couches : authentification des utilisateurs et gestion des autorisations, programmation défensive, validation de toutes les entrées, confiance minimale etc.
- mais les différentes couches n'ont pas le même rôle et le même poids dans la stratégie de sécurité globale de l'application : l'authentification doit être faite immédiatement dans l'interface utilisateur, et communiquée aux autres couches ; les vérifications essentielles doivent être faites côté serveur, dans les composants métier ; la liaison entre les couches doit être sécurisée.

Il faut contextualiser les principes de sécurité, car une conception N Tiers sécurisée se code différemment selon l'environnement technique.

Le développeur devra donc traduire les patrons de sécurité utilisés dans la conception, par différents moyens selon la technologie et la couche à implémenter, en s'appuyant toujours sur les mécanismes de sécurité standards des Frameworks.

Par exemple, dans l'environnement Java JEE :

- dans l'interface utilisateur Web en **Struts** ou en **JSF**, on utilisera les **Validateurs** prédéfinis pour vérifier les entrées des formulaires, avec les mécanismes d'affichage d'erreurs associés, plutôt qu'un système de validation « maison » ;
- pour authentifier les utilisateurs et gérer leurs autorisations, on utilisera l'API **JAAS** (*Java Authentication and Authorization Service*) ;
- pour sécuriser la communication entre les couches, on utilisera les protocoles **HTTPS** et **SSL/TLS** (*Secure Sockets Layer / Transport Layer Security*) ;
- pour sécuriser les objets métier, on utilisera des annotations pour réserver l'utilisation de chaque EJB session à un utilisateur ou un groupe d'utilisateur ;
- dans les entités, on utilisera des annotations pour garantir les contraintes sur les données liées aux tables SQL.

Pour ne pas faire double emploi avec des technologies que vous connaissez déjà (par exemple les validateurs), nous allons nous concentrer sur les deux aspects les plus techniques :

- **l'authentification et la gestion des autorisations dans une application N Tiers** : comment gérer une authentification de bout en bout, en garantissant que l'utilisateur soit
Sécuriser les couches d'une application N Tiers

reconnu dès l'interface utilisateur, et que son identité soit communiquée à chacune des couches traversées ?

Pour comprendre l'API **JAAS**, nous analyserons un code de bas niveau, entre un client lourd et un serveur.

Puis nous reviendrons sur la sécurisation des EJB¹ et des pages Web, dans GlassFish.

- **la sécurisation de la liaison entre le client et le serveur** (par le protocole **SSL/TSL**).

2. JAAS (JAVA AUTHENTICATION AND AUTHORIZATION SERVICE)

Nous allons nous appuyer sur le guide de référence d'Oracle :

<http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html>

Notre objectif n'est pas de devenir spécialiste JAAS mais de comprendre ses mécanismes généraux et de pouvoir l'utiliser pour sécuriser une application Java.

2.1 INTRODUCTION

JAAS a deux objectifs:

- *authentifier les utilisateurs*, pour déterminer avec certitude qui est en train d'exécuter le code Java, que ce code soit exécuté dans une application, une *applet*, un *bean* ou une *servlet* etc.
- *gérer les autorisations des utilisateurs*, pour s'assurer qu'ils ont les droits (permissions) requis pour exécuter les actions réalisées.

JAAS améliore la sécurité des programmes Java, qui était traditionnellement basée sur le contrôle du code source : « *d'où vient le code ? Par qui est-il signé ?* »

JAAS permet aussi de répondre à la question « *par qui est-il exécuté ?* ».

L'authentification JAAS est extensible et permet aux applications de rester indépendantes des techniques d'authentification qu'elle utilise. On peut ajouter des nouvelles technologies d'authentification dans JAAS, sans avoir à modifier l'application elle-même.

2.2 PRINCIPES DE JAAS

(Les classes Java sont en bleu et renvoient à la documentation en ligne d'Oracle).

Pour lancer le processus d'authentification, l'application crée un objet de la classe [LoginContext](#) qui utilise une instance de [Configuration](#) pour déterminer les technologies d'authentification ([LoginModule](#)) utilisés pour réaliser l'authentification.

Un *LoginModule* classique va saisir dans un formulaire le nom et le mot de passe de l'utilisateur. D'autres vont s'appuyer sur la reconnaissance vocale ou les empreintes digitales.

¹ Voir séance : « *Sécuriser les objets métiers dans un environnement serveur* »
Sécuriser les couches d'une application N Tiers

Une fois que l'utilisateur ou le service ([Subject](#)) qui utilise le code a été authentifié, le composant d'autorisation de JAAS travaille avec le modèle de contrôle d'accès standard de JAVA SE, pour protéger les ressources critiques.

Contrairement aux premières versions de Java, qui ne prenaient en compte que l'emplacement du code et sa signature, pour gérer les accès, le modèle actuel de sécurité prend en compte à la fois le code ([CodeSource](#)), et l'utilisateur du code ([Subject](#)).

Si l'authentification réussit, l'instance de *Subject* sera mise à jour avec l'identité de l'utilisateur réel ([Principals](#)) et ses autorisations.

2.3 LA PHASE D'AUTHENTIFICATION

Nous allons commenter l'exemple fourni par le tutoriel Oracle, après l'avoir essayé :

<http://docs.oracle.com/javase/7/docs/technotes/guides/security/jgss/tutorials/AcnOnly.html>



Mise en situation dans GlassFish : authentification JAAS

Ouvrez une fenêtre console, dans le dossier [sources / SampleAcn](#), et lancez le batch [script.bat](#) qui compile et exécute l'exemple.

Le programme exécute un *LoginModule* ([SampleLoginModule](#)) qui demande un nom d'utilisateur et un mot de passe.

Testez que l'authentification réussit avec [testUser](#) et [testPassword](#).

Relancez le programme et faites trois échecs successifs : le programme s'arrête.

2.3.1 Commentaire : création du *LoginContext*

```
LoginContext lc = new LoginContext ("Sample", new MyCallbackHandler() );
```

Crée un *LoginContext*, en précisant la technologie à employer pour l'authentification (le *LoginModule*) et la classe à utiliser pour communiquer avec l'utilisateur (*MyCallbackHandler*)

Le premier paramètre *Sample* est une clé qui référence le *LoginModule* dans le fichier de configuration [sample_jaas.config](#) (à la racine du dossier) :

```
/** Login Configuration for the JAAS Sample Application */
Sample
{
    sample.module.SampleLoginModule required debug=true;
};
```

Le *LoginModule* correspondant est dans le dossier [sample / module](#). C'est une classe qui implémente l'interface *LoginModule*

```
public class SampleLoginModule implements LoginModule
{
    etc.
```

Dans le tutoriel, la classe fournie est très simple : elle compare le nom de l'utilisateur et son mot de passe avec des chaînes en clair dans le code (pas une bonne idée dans la réalité).

Mais il est facile d'écrire un module plus performant, en respectant la même interface :

Sécuriser les couches d'une application N Tiers

Methods	
Modifier and Type	Method and Description
boolean	abort() Method to abort the authentication process (phase 2).
boolean	commit() Method to commit the authentication process (phase 2).
void	initialize() (Subject subject, CallbackHandler callbackHandler, Map<String,?> sharedState, Map<String,?> options) Initialize this LoginModule.
boolean	login() Method to authenticate a subject (phase 1).
boolean	logout() Method which logs out a subject.

Quand le *LoginModule* communique avec l'utilisateur, par exemple pour demander son nom et son mot de passe, il ne le fait pas directement, car il existe de nombreuses manières de communiquer avec un utilisateur, et il est souhaitable que le *LoginModule* ne soit pas couplé à l'interface utilisateur. Il appelle donc une classe *MyCallbackHandler* pour réaliser l'interaction avec l'utilisateur et obtenir les informations requises. Cette classe spécifique au contexte de l'utilisateur doit implémenter l'interface *CallbackHandler*.

A la création du *LoginContext*, il faut lui passer l'instance de la classe qui implémente *CallbackHandler*. Comme le *LoginContext* n'est qu'un intermédiaire vers les *LoginModule*, il passe la référence de cette classe au *LoginModule*, qui va l'utiliser pour récupérer les informations nécessaires à l'authentification.

2.3.2 Commentaire : appel de la méthode *login* du *LoginContext*

Lors de son instantiation, le *LoginContext* crée un objet vide [javax.security.auth.Subject](#) (qui représente l'utilisateur ou le service en cours d'authentification). Il construit le *LoginModule* défini par le paramètre du constructeur (*Sample* -> *SampleLoginModule*) et il l'initialise avec le nouveau *Subject* et la classe de *CallbackHandler* (*MyCallbackHandler*).

Après avoir instancié le *LoginContext* *lc*, il faut appeler sa méthode *login* pour réaliser le processus d'authentification :

```
lc.login();
```

La méthode *login* du *LoginContext* appelle les méthodes du *LoginModule*, pour qu'il effectue les opérations d'authentification. Le *LoginModule* utilise la classe *MyCallbackHandler* pour récupérer le nom de l'utilisateur et son mot de passe. Puis il communique ces informations au système d'authentification (par exemple *Kerberos KDC*, [Kerberos reference documentation](#)).

Si l'authentification réussit, le *LoginModule* remplit l'objet *Subject* avec l'identifiant (*Principal*) renvoyé par le système de sécurité, qui représente l'utilisateur et ses droits (par exemple un *Kerberos Principal*).

L'application appelante peut alors retrouver l'objet *Subject* (authentifié et détaillé) en appelant la méthode *getSubject* de la classe *LoginContext*.

2.3.3 Résumé

Cette traduction libre de la documentation Oracle *JAAS* résume l'enchainement des opérations, mais elle peut vous paraître obscure, à bon droit !

Il faut donc résumer et insister sur les points intéressants pour une approche objet et sécurité :

1) Ce n'est pas à proprement parler *JAAS* qui authentifie :

Comme beaucoup d'API Java (*JNDI*, *JMS*, *JDBC* etc.) *JAAS* n'est qu'un intermédiaire.

Sécuriser les couches d'une application N Tiers

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Dans une véritable application sécurisée, le travail d'authentification est délégué à un système de sécurité protégé et indépendant du langage (par exemple *Kerberos*).

2) JAAS n'est pas un système monolithique, mais modulaire et évolutif :

La classe *LoginContext* ne fait rien par elle-même : elle sert de « **Contrôleur** » ou de « **Façade** » pour l'application appelante, qui ne voit qu'elle.

Elle sert aussi de « **Fabrique** » (*factory*) puisqu'elle instancie le *LoginModule* (qui va effectuer l'authentification dans le cas simple de l'exemple, ou la sous-traiter à un système comme *Kerberos*) et l'objet *Subject* qui va recevoir les informations de l'utilisateur une fois authentifié.

2.4 LA PHASE D'AUTORISATION

Nous allons donner quelques repères, d'après le tutoriel Oracle :

<http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/tutorials/GeneralAcnAndAzn.html>

2.4.1 Principe

L'autorisation JAAS étend l'architecture de sécurité du langage Java, dont la politique de sécurité par défaut (*policy*) spécifie les droits d'accès accordés à l'exécution d'un code. L'architecture de sécurité depuis Java 2 est centrée sur le code. Ce qui veut dire que les permissions sont accordées en se basant sur les caractéristiques du code : *d'où vient le code, est-il signé et par qui ?*

Ceci donne le type de permissions qui sont attribuées au code contenu dans le fichier *SampleAcn.jar* situé dans le répertoire courant (il n'y a pas de signature du code, et on ne se préoccupe pas de savoir si le code est signé ou non) :

(fichier *sampleazn.policy*)

```
grant codebase "file:./SampleAcn.jar"
{
    permission javax.security.auth.AuthPermission "createLoginContext.Sample";
};
```

Les autorisations JAAS augmentent ces contrôles d'accès centrés sur le code, avec de nouveaux contrôles centrés sur l'utilisateur.

Les permissions ne sont pas attribuées seulement sur le critère « *de quel code s'agit-il ?* » mais aussi « *qui est en train de l'exécuter ?* ».

Ci-dessous, les droits de lecture sur *java.home* et *user.home* et le fichier *foo.txt* sont attribués au Principal *testUser*, pour les actions effectuées dans *SampleAction.jar*.

```
/** User-Based Access Control Policy for the SampleAction class
** instantiated by SampleAzn
**/
grant codebase "file:./SampleAction.jar",
    Principal sample.principal.SamplePrincipal "testUser"
{
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "user.home", "read";
    permission java.io.FilePermission "foo.txt", "read";
};
```

2.4.2 Scénario de l'autorisation

Quand une application utilise JAAS pour authentifier un utilisateur (ou une autre entité comme un service), JAAS renvoie un objet [Subject](#) qui représente l'utilisateur authentifié.

Un *Subject* est composé d'un ensemble de [Principals](#) où chaque *Principal* représente une identité pour cet utilisateur. Par exemple, un *Subject* peut avoir comme *Principals* son nom ([Lecu](#)) et son numéro de sécurité sociale ([1590159610110](#)), qui le distingue des autres *Subjects*.

Dans une politique de sécurité, on peut attribuer des permissions à des *Principals* distincts. Après l'authentification, l'application associe le *Subject* au contexte de contrôle d'accès en cours. Pour toutes les opérations suivantes qui exigent une vérification de sécurité (par exemple un accès à un fichier local), la machine virtuelle Java (*JRE*) déterminera automatiquement si la politique de sécurité accorde les permissions demandées par un *Principal* donné, et ainsi les opérations ne seront permises que si le *Subject* défini dans le contexte de contrôle d'accès en cours contient ce *Principal*.

2.4.3 Exemple du tutoriel



Mise en situation dans GlassFish : authentification et autorisation JAAS

Ouvrez une fenêtre console, dans le dossier [sources / SampleAzn](#), et lancez le batch [script.bat](#) qui compile et exécute l'exemple.

Le programme exécute le même *LoginModule* ([SampleLoginModule](#)), qui demande un nom d'utilisateur et un mot de passe.

L'authentification réussit avec [testUser](#) et [testPassword](#), et l'application affiche les droits de l'utilisateur (*Subject*) identifié.

```
D:\Etude Securite\Livrables\Partie 2 - Séances\SEANCES\14 Sécuriser les couches
logicielles dans une application N Tiers\sources\SampleAzn>java -classpath Sampl
eAzn.jar;SampleAction.jar;SampleLM.jar -Djava.security.manager -Djava.security
.policy==sampleazn.policy -Djava.security.auth.login.config==sample_jaas.config
sample.SampleAzn
user name: testUser
password: testPassword
[SampleLoginModule] user entered user name: testUser
[SampleLoginModule] user entered password: testPassword
[SampleLoginModule] authentication succeeded
[SampleLoginModule] added SamplePrincipal to Subject
Authentication succeeded!
Authenticated user has the following Principals:
SamplePrincipal: testUser
User has 0 Public Credential(s)
Your java.home property: C:\Program Files\Java\jre1.8.0_101
Your user.home property: C:\Users\regis
foo.txt does exist in the current working directory.
```

3. AUTHENTIFICATION ET AUTORISATION DANS UNE APPLICATION N TIERS

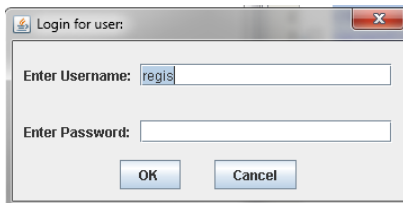
Nous venons d'analyser une utilisation bas niveau de JAAS qui demande beaucoup d'informations techniques pour sa mise en œuvre. Mais il y a plus simple : JAAS est intégré à de nombreux composants JAVA SE et JAVA EE.

Dans la séance sur la sécurisation des objets métier, nous avons vu comment déclarer des utilisateurs et des groupes d'utilisateurs dans des *Realms* du serveur GlassFish : cela ne demande pas de code technique, et peut se faire entièrement sous la console d'administration de GlassFish.

Sécuriser les couches d'une application N Tiers

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Dans cette séance, les objets métier étaient appelés par un client lourd avec une authentification de base (fenêtre de connexion prédéfinie) :



Nous allons reprendre cette démarche dans une application N Tiers en développant notre propre formulaire de connexion, et en identifiant l'utilisateur de bout en bout : la couche Web va communiquer à la couche métier (EJB) l'identité de l'utilisateur authentifié, de façon à gérer ses autorisations sur les objets métier.

3.1 SECURISATION D'UNE APPLICATION WEB

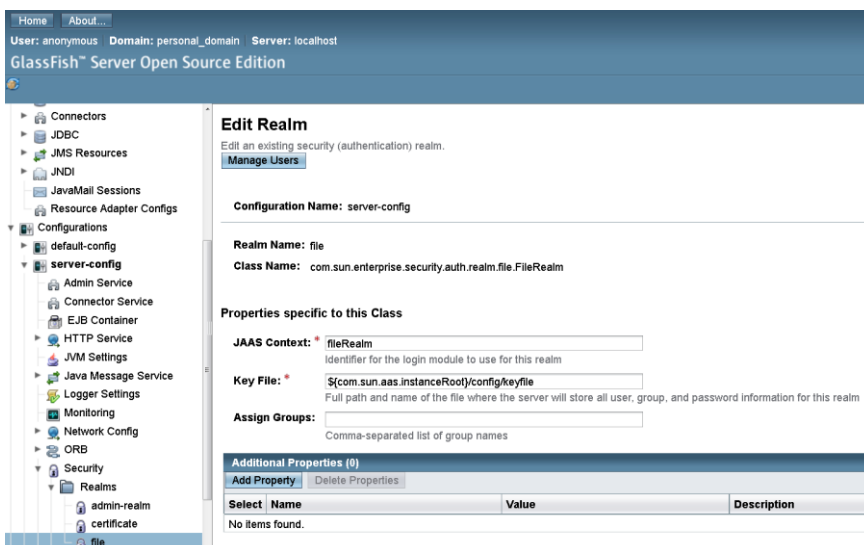
Nous allons suivre le tutoriel de NetBeans sur la sécurisation des applications Web (environ 40 minutes) : <https://netbeans.org/kb/docs/web/security-webapps.html>

Parmi les serveurs décrits, nous prendrons GlassFish.

Un rappel avant de commencer :

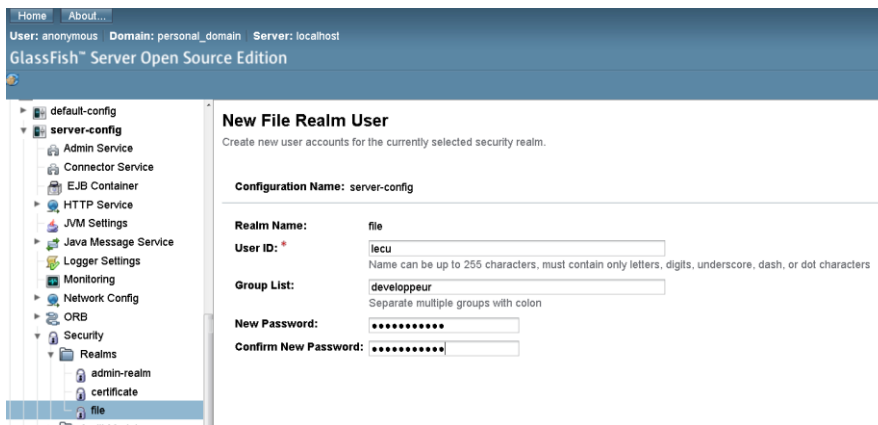
GlassFish gère les utilisateurs avec des *realms* (royaumes, domaines).

Dans la console de GlassFish, il faut cliquer dans *Security / Realms / file*, pour ouvrir le formulaire *Edit Realm* :



Manage Users permet d'ajouter des utilisateurs et des groupes :

Sécuriser les couches d'une application N Tiers



Mise en situation dans GlassFish : *Securing a Web application in NetBeans IDE*

L'application complète est fournie dans : [corrige / WebApplicationSecurity](#)

3.2 SECURISATION D'UNE APPLICATION N TIERS

Comment réunir les deux maquettes : couche Web et couche métier ?

1) La maquette [WebApplicationSecurity](#) définit des contraintes de sécurité sur les accès aux pages Web :

Chaque contrainte est associée à un *role-name* ([AdminRole](#)) et s'applique à un ensemble de pages ([/secureAdmin/*](#))

(Extrait du fichier de configuration *web.xml*) :

```
<security-constraint>
  <display-name>AdminConstraint</display-name>
  <web-resource-collection>
    <web-resource-name>Admin</web-resource-name>
    <description/>
    <url-pattern>/secureAdmin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>AdminRole</role-name>
  </auth-constraint>
</security-constraint>
```

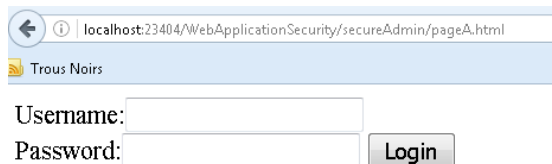
Chaque *role-name* est lui-même associé à un *Principal* ([admin](#)) défini dans GlassFish :

(Extrait du fichier *glassfish-web.xml*)

```
<security-role-mapping>
  <role-name>AdminRole</role-name>
  <principal-name>admin</principal-name>
</security-role-mapping>
```

Lorsque l'utilisateur ouvre une page sécurisée, il est automatiquement redirigé vers le formulaire de connexion :

Sécuriser les couches d'une application N Tiers



- 2) La maquette [cart-ejbV1](#) (fournie dans [sources](#)) utilise des annotations `@DeclareRoles` avant la déclaration de la classe EJB :

```
@DeclareRoles("TutorialUser")
public class CartBean implements Cart, Serializable { etc.
```

Et restreint l'accès à une méthode de cet *ejb* à un rôle de sécurité, par l'annotation `@RolesAllowed` :

```
@RolesAllowed("TutorialUser")
public void addBook(String title) { etc.
```

Les rôles de sécurité sont définis et associés aux Principals de Glassfish dans le fichier *glassfish-ejb-jar.xml* :

```
<security-role-mapping>
  <role-name>TutorialUser</role-name>
  <principal-name>lecu</principal-name>
</security-role-mapping>
```

3) Circulation des identifiants de sécurité (*principal*)

Si nous appelons un EJB session depuis une page Web sécurisée, nous endossons l'identité de l'utilisateur connecté, qui est transmise au container d'EJB.

L'identité de l'utilisateur est donc reconnue dès l'interface utilisateur, et il est tracé de bout en bout, jusqu'à la couche métier.



Mise en situation dans GlassFish : application N Tiers sécurisée.

Nous allons construire une application N Tiers, avec deux identifiants (*Principal*) :

- *admin* : il aura accès à la page administrateur et à la page utilisateur dans le site Web ; il pourra exécuter toutes les méthodes du panier (classe *CartBean*) y compris la méthode *remove* qui supprime tous les articles ;
- *user* : il n'aura accès qu'à la page utilisateur dans le site Web ; il pourra exécuter les méthodes du panier, à l'exclusion de la méthode *remove* qui est réservée à l'administrateur.

Placez les bonnes annotations dans les deux projets et vérifiez le fonctionnement global :

- possibilité pour l'administrateur de se connecter sur les pages administrateur et utilisateur

(Contrôle de sécurité au niveau de la couche Web) ;

- impossibilité pour l'utilisateur d'accéder à une page administrateur

(Contrôle de sécurité au niveau de la couche Web) ;

Sécuriser les couches d'une application N Tiers

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

HTTP Status 403 - Forbidden

type Status report

message Forbidden

description Access to the specified resource has been forbidden.

GlassFish Server Open Source Edition 4.1.1

- la propagation de l'identité de l'utilisateur connecté, de la page Web vers le *container* d'EJB : par exemple, un utilisateur ne pourra pas appeler la méthode *remove* sur le panier, depuis sa page personnelle (contrôle de la couche métier, EJB)

Trace dans la console GlassFish :

```
Infos: JACC Policy Provider: Failed Permission Check, context(cart-ejbV2/cart-ejbV2)-
permission(("javax.security.jacc.EJBMethodPermission" "CartBean" "remove,Remote,"))
Avertissement: A system exception occurred during an invocation on EJB CartBean, method:
public void javaeetutorial.cartsecure.ejb.CartBean.remove()
Avertissement: javax.ejb.AccessLocalException: Client not authorized for this invocation
```

- un utilisateur malveillant, depuis une page sans autorisation Web, ne pourra utiliser aucune des méthodes des EJB (voir page : [malveillant.jsp](#) dans corrigé)

Trace dans la console GlassFish :

```
Infos: JACC Policy Provider: Failed Permission Check, context(cart-ejbV2/cart-ejbV2)-
permission(("javax.security.jacc.EJBMethodPermission" "CartBean"
"addBook,Remote,java.lang.String"))
Avertissement: A system exception occurred during an invocation on EJB CartBean, method:
public void javaeetutorial.cartsecure.ejb.CartBean.addBook(java.lang.String)
Avertissement: javax.ejb.AccessLocalException: Client not authorized for this invocation
```



Mais attention : le mécanisme d'authentification Web repose sur des cookies de session.

La maquette est simple et n'intègre pas de mécanismes de déconnexion : il faut quitter le navigateur ou supprimer le cookie *localhost* pour se déconnecter.

Si vous lancez maintenant la page [malveillant.jsp](#), sans vous déconnecter du mode administrateur, la page fonctionne : on n'échappe donc pas à l'attaque par **CSRF** (*Cross Site Request Forgery*)

Corrigé dans le dossier : [corriges / N Tiers](#)

4. LES PROTOCOLES SSL/TLS

Une activité indispensable et complémentaire de l'authentification, consiste à sécuriser la communication entre le client et les différentes couches serveur, afin d'éviter la capture d'informations utilisateur ou d'informations de sécurité sur le réseau.

En Java, on utilise le protocole sécurisé **SSL** (*Secure Socket Layer*) et son successeur **TLS** (*Transport Layer Security*).

Dans cette séance, nous allons nous centrer sur l'utilisation bas niveau, dans le code.

La configuration d'une liaison **HTTPS** sur un serveur Web a été vue dans la séance : « *Prendre en compte le réseau dans la sécurité du Web* ».

4.1 PRESENTATION DES PROTOCOLES SSL/TLS



Lire l'article de Wikipedia : https://fr.wikipedia.org/wiki/Transport_Layer_Security

4.2 MISE EN OEUVRE EN JAVA

4.2.1 Création du certificat

Pour pouvoir utiliser le protocole SSL/TLS, il faut d'abord récupérer un certificat d'une autorité de certification, ou en créer un à des fins de test (auto-signé).

En Java, on utilisera la commande *keytool* qui fait partie du SDK Java SE :

```
keytool -genkey -keystore serveurKeystore -keyalg RSA
```

La commande crée un magasin de clés (*keystore*), dans le fichier *serveurKeystore* du répertoire courant, où est passée la commande.

L'algorithme choisi est *RSA* (paire de clés privée/publique).

Il faut rentrer interactivement le mot de passe du *keystore*, le nom et prénom, l'unité organisationnelle etc.

```
>keytool -genkey -keystore serveurKeystore -keyalg RSA
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Quels sont vos nom et prénom ?
[Unknown]: lecu
Quel est le nom de votre unité organisationnelle ?
[Unknown]: cdi
Quel est le nom de votre entreprise ?
[Unknown]: afpa
Quel est le nom de votre ville de résidence ?
[Unknown]: grenoble
Quel est le nom de votre état ou province ?
[Unknown]: france
Quel est le code pays à deux lettres pour cette unité ?
[Unknown]: fr
Est-ce CN=lecu, OU=cdi, O=afpa, L=grenoble, ST=france, C=fr ?
[Inon]: oui
Entrez le mot de passe de la clé pour <mykey>
(appuyez sur Entrée s'il s'agit du mot de passe du fichier de clés) :
```

Pour utiliser la commande *keytool*, pensez à ajouter le dossier java dans votre PATH :

```
path=%path%;C:\Program Files\Java\jdk1.8.0_101\bin
```



Détail de la commande *keytool* :

<http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html#Changes>

Sécuriser les couches d'une application N Tiers

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

4.2.2 Codage de la partie serveur

Les étapes sont sensiblement les mêmes que pour un serveur de sockets IP.

- 1) récupération de la « *factory* » par défaut de serveur de sockets SSL

```
SSLServerSocketFactory sslFact=(SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
```

- 2) création d'un serveur de socket SSL sur un port utilisateur (9000)

```
SSLServerSocket sslServerSock = (SSLServerSocket) sslFact.createServerSocket(9000);
```

- 3) attente d'une connexion sur le port par un socket SSL

```
SSLSocket sslsock = (SSLSocket) sslServerSock.accept();
```

- 4) lecture ligne à ligne sur le socket et traitement de la ligne

Dans cette démonstration, on utilisera un *BufferedReader* pour lire sur le socket.

Le traitement consistera simplement à réafficher les lignes reçues.



Au lancement du programme serveur (*ServeurSSL*), il faut préciser à la machine virtuelle le nom du magasin de clés (*serveurKeyStore*) et son mot de passe :

```
java -Djavax.net.ssl.keyStore=serveurKeystore -Djavax.net.ssl.keyStorePassword=12345678 ServerSSL
```

4.2.3 Codage de la partie client

- 1) récupération de la « *factory* » par défaut de sockets

```
SSLSocketFactory sslSockFact = (SSLSocketFactory) SSLSocketFactory.getDefault();
```

- 2) création d'un socket pour l'envoi des messages sur la machine locale, port 9000

```
SSLSocket sslsocket = (SSLSocket) sslSockFact.createSocket("localhost", 9000);
```

- 3) Envoi de message ligne à ligne sur le socket

Dans cette démonstration, on utilisera un *BufferedWriter* pour envoyer sur le socket des messages lus au clavier.



Au lancement du programme client (*ClientSSL*), il faut préciser à la machine virtuelle le nom du magasin de confiance (*serveurKeyStore*) et son mot de passe :

```
java -Djavax.net.ssl.trustStore=serveurKeystore -Djavax.net.ssl.trustStorePassword=12345678 ClientSSL
```

Corrigés dans le dossier : [corriges](#) / [SSL](#)

Les deux scripts sont à lancer dans une fenêtre console :

- *scriptCertificat* crée le magasin de clés
- *scriptLancement* lance les programmes serveur et client.

Vous aurez sans doute à changer le PATH de Java dans les deux scripts.

Sécuriser les couches d'une application N Tiers

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

CRÉDITS

OEUVRE COLLECTIVE DE L'AFPA

Sous le pilotage de la DIIP
et du centre sectoriel Tertiaire

EQUIPE DE CONCEPTION

Chantal PERRACHON – IF Neuilly-sur-Marne
Régis Lécu – Formateur AFPA Pont de Claix

Sécuriser les couches d'une application N Tiers

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »