Sécurité Java EE

ITU - Université de Nice Sophia Antipolis
Richard Grin
Version 0.5 – 2/3/19

Plan du support

- Présentation
- Concepts
- Authentification
- Entrepôts d'identités
- Protection des pages Web
- Protection des méthodes des EJB
- Références

But de ce support

- Présenter rapidement l'essentiel des concepts et du code pour utiliser l'API de sécurité introduite par Java EE 8, dans le cadre d'une application Web qui utilise JSF pour l'interface utilisateur

Le but de la sécurité

- N'autoriser l'utilisation de pages Web ou de méthodes Java qu'à certains utilisateurs
- Pour cela il faut pouvoir
  - identifier/authentifier l'utilisateur qui utilise l'application (celui qui fait afficher les pages de l'application)
  - indiquer les restrictions d'accès sur certaines pages ou méthodes
- L'authentification va se faire en s'appuyant sur des informations contenues dans un entrepôt d'identités

API de sécurité

- Intégrée dans le profil Web
- Facilite la tâche du développeur et fournit des implémentations concrètes utilisables telles quelles par le développeur

Concepts

- Caller
- Authentification
- Document d'identification (*credential*)
- Principal
- Rôle
- Autorisation

Caller

- Caller = visiteur, interlocuteur, demandeur, utilisateur de l'application
- À l'origine d'une requête Web au serveur, ou d'un appel d'une méthode

Authentification

- Mécanisme qui permet d'identifier un utilisateur et de vérifier qu'il est bien ce qu'il prétend être
- Interagit avec le caller pour obtenir des *credentials* prouvant l'identité du caller
- Les credentials sont le plus souvent obtenus par la saisie d'un nom de login et d'un mot de passe

Principal

- Une fois qu'il est authentifié, l'utilisateur devient un *principal* (mandant, donneur d'ordre) au nom duquel les opérations seront effectuées dans l'application ; l'identité du caller sera représentée par ce principal
- Un caller agit en tant qu'un de ses principals ; c'est le principal qui détermine ses droits ; par exemple comme étudiant ou comme conducteur de voiture

Rôle

- La politique de sécurité des serveurs d'application repose sur la notion de rôle
- Exemples de rôle : administrateur, organisateur, participant, chef de service
- Un principal peut avoir un ou plusieurs rôles dans une application
- Un rôle donne des droits aux utilisateurs qui ont ce rôle
- La notion de rôle augmente la portabilité de l'application puisqu'on ne fait aucune prévision sur l'existence d'utilisateurs particuliers

Groupe et rôle

- C'est la même chose par défaut pour l'API de sécurité de Java EE 8 : les rôles correspondent aux valeurs dans la table utilisée pour les groupes (voir la section « Mécanismes d'authentification standards »)

Autorisations

- Quand l'utilisateur a été authentifié, il faut veiller à ce ne puisse accéder qu'aux ressources qui lui sont autorisées :
  - les pages Web (voir section « Protection des pages Web »)
  - les méthodes des EJB (voir section « Protection des méthodes des EJB »)

Authentification

JASPIC

- *Java Authentication SPI for Containers* permet de développer des modules d'authentification pour prendre en compte n'importe quel type de preuves d'identité
- Très souple mais complexe
- La nouvelle API s'appuie sur JASPIC mais s'utilise plus simplement ; en particulier elle fournit des implémentations standards pour les types d'authentification les plus fréquents

Mécanisme de base de l'API

- La nouvelle API de sécurité de Java EE 8 s'appuie sur l'interface `HttpAuthenticationMechanism`
- Ce mécanisme permet d'obtenir les *credentials* d'un utilisateur, en utilisant le protocole HTTP
- La validation de ces *credentials* est le plus souvent délégué à un IdentityStore (voir section suivante)

Implémentations de HttpAuthenticationMechanism

- Le container doit fournir des implémentations standards, configurables par des annotations standards
- Le développeur peut aussi créer sa propre implémentation d'un bean CDI qui implémente `HttpAuthenticationMechanism`

- Avant d'étudier les mécanismes d'authentification standards fournis par le serveur d'application, les transparents de la fin de cette section donnent du code qui fonctionne quelle que soit l'implémentation de `HttpAuthenticationMechanism` utilisée

Logout

- Injecter `HttpServletRequest` et appeler la méthode `logout()`
- Il est préférable d'invalider aussi la session

Exemple

```
@Named
@RequestScoped
public class Logout {
  @Inject
  private HttpServletRequest request;

  public void submit() throws ServletException {
    request.logout();
    request.getSession().invalidate();
  }
}
```


Obtenir le login

- Si on injecte un bean CDI de type `javax.security.enterprise.SecurityContext` on peut obtenir un utilisateur authentifié par `securityContext.getCallerPrincipal()` qui retourne un `Principal`
- `Principal` contient la méthode `getName()` qui retourne le nom de l'utilisateur connecté
- Plus simplement on peut injecter `Principal`

Exemple

```
@Inject
Principal principal;
…
  … principal.getName() …
```


Affichage conditionnel

- Pas encore d'objet implicite de type `SecurityContext` dans le langage EL (sans doute ajouté dans les prochaines versions) ; à la place on utilise l'objet implicite `request`
- Test dans une page JSF pour savoir si l'utilisateur est connecté :

```
<h:panelGroup
  rendered="#{not empty request.userPrincipal}">
```

- Pour savoir si l'utilisateur a un certain rôle :

```
#{request.isUserInRole('admin')}
```


Lien conditionnel

- Pour afficher ou pas un lien vers une page Web, suivant les droits de l'utilisateur
- Par exemple pour afficher ou non une entrée de menu
- Utiliser la méthode `hasAccessToWebResource` de l'interface `SecurityContext`
- Comme le langage EL n'a pas encore d'objet implicite pour `SecurityContext`, il faut écrire un peu de code dans un backing bean CDI

Exemple

- Le bean :

```
@Named @ApplicationScoped
public class Security {
  @Inject
  private SecurityContext securityContext;

  public boolean hasAccess(String resource) {
  return securityContext
         .hasAccessToWebResource(resource, "GET");
  }
}
```

- Dans la page :

```
<h:link value="Go to Bar" outcome="/bar“
  disabled="#{not security
              .hasAccess('/bar.xhtml')}" />
```


Mécanismes d'authentification standards

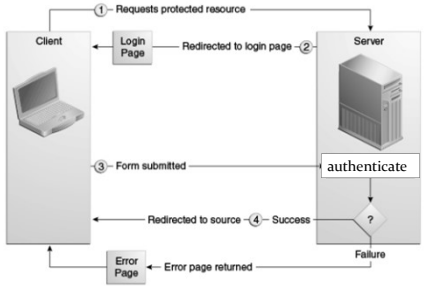
Mécanismes standards

- Des mécanismes standards sont fournis par la plateforme Java EE (spécification des servlets) : BASIC, DIGEST, FORM, CERT
- La nouvelle API de sécurité implémente BASIC et FORM et ajoute CustomForm
- BASIC utilise une fenêtre pop-up du navigateur pour demander le login et le mot de passe
- FORM utilise une page qui contient un formulaire pour obtenir le login et le mot de passe
- CustomForm est une variante de FORM qui convient mieux aux pages JSF

Authentification avec formulaire

- Quand l'utilisateur veut accéder à une page protégée et qu'il ne s'est pas encore authentifié, les mécanismes standards d'authentification affichent *automatiquement* un formulaire
- L'utilisateur saisit son login et son mot de passe dans le formulaire
- Si l'authentification est réussie (login et mot de passe corrects) et si l'utilisateur authentifié a bien les autorisations d'accès à la page protégée, cette page est affichée

Form Based



Erreur à ne pas faire

- Lorsque le formulaire est écrit dans une page de l'application ça serait une erreur de naviguer vers la page qui contient le formulaire de saisie du login ; celui-ci doit être affiché *automatiquement* au moment où l'URL d'une page protégée est demandée
- Le mécanisme d'authentification utilisé par l'API de sécurité est déclenché par le container, pas par l'application
- Il est possible de faire déclencher l'authentification par l'application mais ça n'est pas étudié dans cette introduction

Choix mécanisme authentification

- Le choix du mécanisme se fait par une annotation à placer sur un bean CDI
- Ce support n'étudie que le mécanisme customForm qui est bien adapté à une application qui utilise JSF
- Le plus naturel est de placer l'annotation `@CustomFormAuthenticationMechanismDefinition` sur un bean de configuration de l'application, de portée CDI « application », sans qualificateur
- On ajoute l'annotation `@FacesConfig` pour permettre l'utilisation des ajouts de JSF 2.3

@CustomFormAuthenticationMechanismDefinition

- L'attribut `loginToContinue` est obligatoire
- Sa valeur est du type de l'annotation `@LoginToContinue`

Exemple

```
@CustomFormAuthenticationMechanismDefinition(
    loginToContinue = @LoginToContinue(
           loginPage = "/login",
           errorPage = "/login-fail"
           )
)
@ApplicationScoped
@FacesConfig
public class ApplicationConfig { ... }
```


@LoginToContinue

- Paquetage javax.security.enterprise.authentication.mechanism.http
- Attributs :
  - `loginPage` : la page qui contient le formulaire de login (par défaut « `/login` »)
  - `errorPage` : page d'erreur qui s'affiche si les login et mot de passe ne sont pas corrects (par défaut « `/login-error` »)
  - `useForwardToLogin` : utilise un *forward* pour afficher la page de login (`true` par défaut ; sinon redirection)
  - `useForwardToLoginExpression` : idem `useForwardToLogin` mais valeur donnée par une expression EL évaluée à chaque utilisation du formulaire de login ("" par défaut)

Attribut `errorPage`

- Si on veut afficher les messages d'erreur dans la page de login (c'est le cas le plus souvent avec JSF) plutôt que sur une page à part, écrire `errorPage = ""`

Attribut `useForwardToLogin`

- Comme la page qui contient le formulaire de login est affichée par *forward*, l'adresse affichée par le navigateur pendant que le formulaire est affiché est l'adresse de la page protégée demandée
- Si on donne la valeur `false` à l'attribut `useForwardToLogin`, la page du formulaire est affichée par redirection et l'adresse affichée par le navigateur reflète le contenu affiché

Formulaire défini par l'application

- La page qui contient le formulaire de saisie du login et du mot de passe est définie par l'application
- La méthode du backing bean de la page qui est appelée à la soumission du formulaire doit utiliser la méthode `authenticate` du `SecurityContext` injecté, pour valider le nom et le mot de passe

SecurityContext.authenticate()

- Prend en paramètre la requête, la réponse et des paramètres de type `AuthenticationParameters`, retourne un `AuthenticationStatus` qui indique si le nom et le mot de passe sont validés
- L'exemple suivant illustre son utilisation

Exemple - page de login

```
<h:form>
  <h:outputLabel for="email" value="Email" />
  <h:inputText id="email" value="#{login.email}" />
  <br />
  <h:outputLabel for="mdp" value="Mot de passe" />
  <h:inputSecret id="mdp" value="#{login.motDePasse}" />
  <br />
  <h:commandButton value="Login" action="#{login.submit}" />
  <h:messages />
</h:form>
```


Exemple - backing bean (1/3)

```
@Named
@RequestScoped
public class Login {
  @NotNull @Email
  private String email;
  @NotNull @Size(min = 8, message = "Au moins 8 caractères")
  private String motDePasse;
  @Inject
  private SecurityContext securityContext;
  @Inject
  private ExternalContext externalContext;
  @Inject
  private FacesContext facesContext;
```


Exemple - backing bean (2/3)

```
  public void submit() {
    switch (continueAuthentication()) {
    case SEND_CONTINUE:
      facesContext.responseComplete();
      break;
    case SEND_FAILURE:
      facesContext.addMessage(null, new FacesMessage(
        FacesMessage.SEVERITY_ERROR, "Échec login ", null));
      break;
    case SUCCESS:
      facesContext.addMessage(null, new FacesMessage(
        FacesMessage.SEVERITY_INFO, "Login réussi", null));
      break;
    case NOT_DONE: // n'arrivera jamais
    } }
```


Exemple - backing bean (3/3)

```
  private AuthenticationStatus continueAuthentication() {
    return securityContext.authenticate(
      (HttpServletRequest) externalContext.getRequest(),
      (HttpServletResponse) externalContext.getResponse(),
      AuthenticationParameters.withParams().credential(
        new UsernamePasswordCredential(email, motDePasse))
    );
  }
  // getters et setters pour email et motDePasse
  ...
}
```

Entrepôts d'identités (Identity store)

Entrepôt d'identités

- Entrepôt pour garder des informations sur les utilisateurs, pour les authentifier
- Contient les noms de groupe, noms de login et mots de passe pour le cas étudié dans ce support

2 types d'entrepôt standards fournis par Java EE

- LDAP configuré avec **`@LdapIdentityStoreDefinition`**
- Base de données configuré avec **`@DatabaseIdentityStoreDefinition`** (le seul qui sera étudié dans ce support)
- Ces annotations doivent annoter un bean CDI de portée **`@ApplicationScoped`** et le qualificateur **`@Default`** (pas de qualificateur)

`@DatabaseIdentityStoreDefinition`

- Quelques attributs (de type `String`, sauf précisé) :
  - `dataSourceLookup`, par défaut "java:comp/DefaultDataSource" : nom JNDI de la source de données qui contient les informations sur les logins, les mots de passe et les groupes
  - `callerQuery` : requête SQL dont le seul paramètre correspond au nom de login et qui retourne le mot de passe en 1ère colonne
  - `groupsQuery` : requête SQL dont le seul paramètre correspond au nom de login et qui retourne les groupes en 1ère colonne (plusieurs groupes possibles)

`@DatabaseIdentityStoreDefinition`

- Class<? extends PasswordHash> `hashAlgorithm`, par défaut Pbkdf2PasswordHash.class du paquetage javax.security.enterprise.identitystore : hash utilisé pour les mots de passe
- String[] `hashAlgorithmParameters`, par défaut {} : paramètres de l'algorithme ; taille du sel par exemple

Algorithme de hachage

- Par défaut PBKDF2WithHmacSHA256
- On peut paramétrer l'algorithme (taille de la clé, nombre d'itérations, taille du sel)
- Par défaut (si on ne donne pas de paramètres pour l'attribut hashAlgorithmParameters) :
  - 2048 itérations
  - taille du sel 32 octets
  - taille de la clé 32 octets
- Les algorithmes supportés : PBKDF2WithHmacSHA224, PBKDF2WithHmacSHA256, PBKDF2WithHmacSHA384, PBKDF2WithHmacSHA512

Exemples

- Ces exemples utilisent une base de données qui contient les tables LOGIN et GROUPE avec les structures suivantes :
  - LOGIN a les colonnes login, mot_de_passe
  - GROUPE a les colonnes login, nom_groupe
- Les noms et les structures des tables et des colonnes peuvent être variés, du moment que l'on donne les bonnes requêtes SQL pour récupérer le mot de passe et les groupes

Exemple pour base de données

```
@DatabaseIdentityStoreDefinition(
  dataSourceLookup="java:app/monstore",
  callerQuery="select mot_de_passe from login where
login=?",
  groupsQuery="select nom_groupe from groupe where
login=?")
@ApplicationScoped
public class ConfigApplication {
}
```


Exemple avec hachage (1/2)

```
@DatabaseIdentityStoreDefinition(
  dataSourceLookup="java:app/monstore",
  callerQuery="select password from principals where
username=?",
  groupsQuery="select group from principals where
username=?",
  hasAlgorithm=Pbkdf2PasswordHash.class,
  hashAlgorithmParameters = {
    "Pbkdf2PasswordHash.Iterations=3072",
    "${applicationConfig.hash}" // juste pour tester
)
```


Exemple avec hachage (2/2)

```
@ApplicationScoped
@Named // obligatoire car référencé dans expression EL
public class ConfigApplication {
  public String[] getHash() {
    return new String[]
      {"Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512",
       "Pbkdf2PasswordHash.SaltSizeBytes=64"};
}
```


Hachage de mots de passe

- Pour des raisons de sécurité il n'est pas bon de conserver les mots de passe en clair dans le serveur
- Avant d'être enregistrés les mots de passe doivent être transformés par une fonction de hachage, SHA-256 par exemple, qui transforme un tableau d'octets de taille quelconque en un tableau d'octets de taille constante (32 octets pour SHA-256)
- Un codage « HEX » ou « Base64 » est appliqué à la valeur de hachage si on veut manipuler la valeur sous la forme de caractères

Sites pour hacher les mots de passe

- Pour savoir si les données enregistrées dans la base de données correspondent bien aux mots de passe non cryptés
- Pour avoir ces sites, taper, par exemple, « SHA-256 online » sur Google
- On trouve, par exemple https://hash.online-convert.com/sha512-generator ou (il y a d'autres algorithmes disponibles à cet URL) http://hash.online-convert.com/sha256-generator qui donne le mot de passe (ou le contenu d'un fichier) codé en SHA-512 avec Hex et Base64

Interfaces pour le hachage

- Le paquetage `javax.security.enterprise.identitystore` contient l'interface `Pbkdf2PasswordHash` qu'on peut utiliser pour définir un entrepôt d'identité
- Elle hérite de l'interface `PasswordHash`

Hacher un mot de passe

- Si on a besoin de générer des mots de passe hachés, on injecte par CDI une classe fournie par Java EE qui permet de faire cette génération (ici à partir du mot de passe en clair « lemdp ») :

```
@Inject
private Pbkdf2PasswordHash passwordHash;
...
... // initialisation ; voir transparents suivants
passwordHash.generate("lemdp".toCharArray())
```


Exemple

```
// Initialise la fonction pour le hachage
Map<String, String> parameters= new HashMap<>();
parameters.put("Pbkdf2PasswordHash.Iterations", "3072");
parameters.put("Pbkdf2PasswordHash.Algorithm",
               "PBKDF2WithHmacSHA512");
parameters.put("Pbkdf2PasswordHash.SaltSizeBytes", "64");
passwordHash.initialize(parameters);
...
execute(c,
  "INSERT INTO login VALUES(('toto', '"
    + passwordHash.generate("lemdp".toCharArray())
  + "')");
```


Initialisation de l'entrepôt

- Si les tables et les données utilisées par l'entrepôt d'identité n'existent pas déjà, l'application peut les créer au démarrage
- Le plus simple est d'écrire un EJB Singleton qui est créé au démarrage de l'application et qui contient une méthode annotée par `@Startup`
- Le code d'initialisation peut utiliser JPA ou JDBC

Exemple initialisation

- Les transparents qui suivent donnent 2 exemples de code
  - Le 1er exemple reprend la structure simple des tables LOGIN et GROUPE des exemples précédents et utilise JDBC
  - Le 2ème exemple utilise JPA au lieu de JDBC, avec une structure un peu plus complexe pour les tables

Création des tables

- On se place dans le cas où l'application doit créer de nouvelles tables pour enregistrer les logins et les groupes (le plus souvent ces tables existent déjà)
- S'il y a des entités pour les logins et les groupes dans l'application, le plus simple est de les faire créer automatiquement en configurant la source de données est en mode « create-table » dans persistence.xml
- Sinon, les tables peuvent être créées par du code JDBC
- Les créer par du code JPA (avec une requête native) pose des difficultés pour tester si les tables existent déjà

Exemple 1, avec JDBC (1/2)

```
@Singleton
@Startup
@DataSourceDefinition(
  className = "org.apache.derby.jdbc.ClientDataSource",
  name = "java:app/jdbc/securite",
  serverName = "localhost",
  portNumber = 1527,
  user = "app", // nom et mot de passe
  password = "app", // donnés à la création BD
  databaseName = "securite"
)
public class Init {
  @Resource(lookup = "java:app/jdbc/securite")
  private DataSource dataSource;
  @Inject
  private HashMdp passwordHash; // Pour coder mot de passe
```


Exemple 1, avec JDBC (2/2)

```
  @PostConstruct
  public void init() {
    try (Connection c = dataSource.getConnection()) {
      execute(c, "CREATE TABLE login (login VARCHAR(20)
PRIMARY KEY, mot_de_passe VARCHAR(160))");
      execute(c, "CREATE TABLE groupe (login VARCHAR(20)
references login, groupe VARCHAR(20))");
      String hashMdp = passwordHash.generate("toto");
      execute(c, "INSERT INTO login (LOGIN, MOT_DE_PASSE)
VALUES('admin', '" + hashMdp + "')");
      execute(c, "INSERT INTO groupe(login, groupe)
VALUES('admin', 'admin')");
      ...
    } catch (SQLException ex) { // init ne peut lancer
      // d'exception contrôlée par compilateur
```


Exemple 2, avec JPA

- Utiliser une requête native pour créer les tables (pas standardisé dans JPA) et l'exécuter dans une transaction
- Les tables qui contiennent les logins, mots de passe et groupes peuvent contenir d'autres informations que le minimum requis
- Il est plus simple d'ajouter des identificateurs non significatifs générés automatiquement et de faire générer les tables en configurant persistence.xml

Exemple (1/4) – Entité User

```
@Entity
public class Login implements Serializable {
  @Id
  private String nom;
  private String mdp;
  private List<Groupe> groupes = new ArrayList<>();

  public Login(String nom, String mdp) {
    this.nom = nom;
    this.mdp = mdp;
  }
  ... // En particulier constructeur sans paramètre
}
```


Exemple (2/4) – Entité Groupe

```
@Entity
public class Groupe implements Serializable {
  @Id
  private Long id;
  private String nom;
  @ManyToMany(mappedBy="groupes")
  private List<Login> logins = new ArrayList<>();

  public Groupe(Long id, String nom) {
    this.id = id;
    this.nom = nom;
  }
  ... // En particulier constructeur sans paramètre
      // et méthode pour mettre login dans groupe
}
```


Exemple (3/4) – Données

```
@Singleton @Startup
public class Init {
  @PersistenceContext
  private EntityManager em;
  @Inject
  private Pbkdf2PasswordHash passwordHash;
  @PostConstruct
  public void populate() {
    Map<String, String> parameters = new HashMap<>();
    parameters.put("Pbkdf2PasswordHash.Iterations", "3072");
    parameters.put("Pbkdf2PasswordHash.Algorithm",
                   "PBKDF2WithHmacSHA512");
    parameters.put("Pbkdf2PasswordHash.SaltSizeBytes", "64");
    passwordHash.initialize(parameters);
```

Exemple (4/4) – Données

```
    User admin = em.find(Login.class, "admin");
    if (user == null) {
      admin = new Login("admin",
               passwordHash.generate("a***n".toCharArray()));
      em.persist(user);
    }
    Groupe groupe = em.find(Groupe.class, 1L);
    if (groupe == null) {
      groupe = new Groupe(1L, "admin", admin);
      em.persist(groupe);
    }
    ... // Met login dans groupe
  }
}
```


Protection des pages Web

Utilité

- Cette section explique comment restreindre l'accès des pages d'une application à certains utilisateurs

Erreurs HTTP pour la sécurité

- Lorsqu'un utilisateur veut accéder à une page protégée, il peut arriver les erreurs suivantes :
  - erreur HTTP 401, *Unauthorized*, si l'utilisateur ne fournit pas les *credentials* (le plus souvent le bon mot de passe)
  - erreur HTTP 403, *Forbidden*, si l'utilisateur a fournit les bons *credentials* mais n'a pas l'autorisation d'accéder à la page

Schéma de protection

- Le plus simple est d'utiliser le système de sécurité offert par les containers de page Web des serveurs d'application
- Il est simple de grouper les pages à protéger dans un ou plusieurs répertoires et de filtrer l'accès à ces répertoires
- Pour les cas plus complexes de filtrage, il est possible de coder le filtrage directement dans le code Java de l'application

Déclaration des protections

- La protection est déclarée dans le fichier `web.xml` par des contraintes de sécurité déclarées sur des ressources (les pages Web désignées par des modèles d'URL)

Notions utilisées

- L'application peut définir des contraintes de sécurité (`<security-constraint>`)
- Elles restreignent l'accès à des ressources (page Web définies par leur URL) aux seuls utilisateurs qui ont un certain rôle (plusieurs rôles peuvent être indiqués)
- Lorsqu'un utilisateur veut accéder à une ressource protégée, il doit être authentifié et il peut accéder à la ressource s'il a les rôles requis par la ressource

Types de contraintes de sécurité

- D'exclusion
- Non contrôlées
- Par rôle

D'exclusion

- On définit l'URL (`<url-pattern>`) pour la contrainte de sécurité mais sans lui associer de contrainte d'authentification (pas de `<auth-contraint>`)
- Les ressources qui correspondent à l'URL ne sont pas accessibles directement par une requête HTTP GET
- Cependant ces ressources seront toujours accessibles par une requête POST qui retournerait une de ces ressources

Non contrôlées

- Si une page ne correspond à aucun des URLs définis par les contraintes de sécurité, elle peut être accédée par tous, même les utilisateurs non authentifiés

Par rôle

- Un rôle correspond à un nom quelconque qui correspond à un type d'utilisateur (« admin » par exemple) ou bien à une protection spéciale pour certaines actions (« can-remove-account » par exemple)
- Les rôles apparaissent dans les contraintes d'autorisation pour indiquer qui peut accéder aux pages protégées par une contrainte de sécurité
- Un utilisateur aura accès à une page s'il a un des rôles listés dans la contrainte d'authentification
- Les rôles sont le plus souvent associés aux utilisateurs au moment de leur authentification

Contraintes de sécurité

- Les sous balises de `<security-constraint>` :
  - `<web-resource-collection>` indique les URLs des pages (`<url-pattern>`) et les méthodes HTTP protégées (`<http-method>`)
  - `<auth-constraint>` indique les rôles qui permettent d'accéder aux pages (il suffit d'avoir un de ces rôles pour y avoir accès)
  - `<user-data-constraint>` indique comment sont protégées les données échangées entre le client et le serveur (aucune protection par défaut)

Exemple

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>r1</web-resource-name>
    <url-pattern>/liste.xhtml</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>employe<role-name>
    <role-name>drh<role-name>
  </auth-constraint>
</security-constraint>
```


<url-pattern>

- Les pages protégées sont désignées par la sous-balise `<url-pattern>` qui donne un chemin qui peut comporter le joker *
- 3 Formats possibles pour un url-pattern :
  - commence par « / » et se termine par « /* » (toute page située sous le répertoire qui précède « /* », à n'importe quelle profondeur
  - commence avec « *. » et se termine par un caractère (par exemple *.jsf)
  - commence par « / » et ne comporte pas de *

Exemple

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>p1</web-resource-name>
    <url-pattern>/faces/candids/*</url-pattern>
    <url-pattern>/candids/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>admin</role-name>
     <role-name>rsr</role-name>
  </auth-constraint>
</security-constraint>
```


<http-method>

- Une contrainte d'authentification peut restreindre l'accès par une ou plusieurs méthodes HTTP
- Par défaut (si on ne cite aucune méthode HTTP), les accès aux ressources sont protégés pour n'importe quelle méthode HTTP
- Mais si on cite au moins une méthode HTTP, la protection ne concerne que les accès de la ressource avec les méthodes HTTP données ce qui signifie que les accès par toutes les autres méthodes HTTP ne seront pas protégés

Exemple

Seuls les accès à « `r1` » par les méthodes HTTP GET et POST sont protégés ; l'accès par PUT est autorisé pour tous

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>r1</web-resource-name>
    <url-pattern>/liste.xhtml</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>employe<role-name>
  </auth-constraint>
</security-constraint>
```


<deny-uncovered-http-methods/>

- Pour éviter d'oublier de protéger une ressource pour une méthode HTTP, on peut utiliser la balise top-level `<deny-uncovered-http-methods/>` (placée en dehors de toute contrainte de sécurité)
- Toutes les méthodes non citées seront alors interdites ; on cite les méthodes que l'on veut autoriser
- Dans l'exemple suivant, seuls les accès par GET et POST sont autorisés pour le rôle « employe »

Exemple – depuis Servlet 3.1

```
<deny-uncovered-http-methods/>>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>r1</web-resource-name>
    <url-pattern>/liste.xhtml</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>employe<role-name>
  </auth-constraint>
</security-constraint>
```

Les accès à « r1 » par les méthodes HTTP GET et POST sont protégés ; l'accès par PUT est interdit pour tous

Lien vers une page protégée

- C'est l'URL de la page que l'on veut afficher qui est comparé aux patterns des pages protégées
- Si une page contient un lien vers une page protégée, il faut écrire ce lien avec les composants `<h:outputLink>`, `<h:button>` ou `<h:link>`, ou ajouter une redirection à un lien avec un composant `<h:commandButton>` ou `<h:commandLink>`

Et avec un forward interne ?

- Un forward interne au serveur (méthode `forward` de `RequestDispatcher`) ne déclenche pas une vérification des contraintes de sécurité puisque l'URL n'est pas modifié
- Ce qui signifie qu'un `<h:commandButton>` ou un `<h:commandLink>` (sans redirection) qui retourne une page protégée en réponse à la requête POST, donnera accès à la page, même si l'utilisateur n'a pas l'autorisation d'accès à cette page

`<auth-constraint>`

- L'autorisation sera accordée si l'utilisateur a un des rôles indiqués
- Cas particuliers pour les rôles :
  - « * » indique que tous les rôles sont acceptés
  - « ** » indique que tous les utilisateurs authentifiés sont acceptés
- S'il y a une balise `<auth-constraint>` mais aucune sous-balise `<role-name>`, les pages ne pourront être accédée par aucun utilisateur

Exemple

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>r1</web-resource-name>
    <url-pattern>/liste.xhtml</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>**<role-name>
  </auth-constraint>
</security-constraint>
```


`<user-data-constraint>`

- La balise `<user-data-constraint>` peut être ajoutée dans une balise `<security-constraint>` pour indiquer que la connexion avec les pages protégées utilisera le protocole HTTPS
- Elle contient une balise `<transport-garantee>` dont la valeur peut être égale à `NONE` (valeur par défaut), `CONFIDENTIAL` ou `INTEGRAL`

Valeurs pour <transport-garantee>

- CONFIDENTIAL : les données transmises ne peuvent être vues par des tiers
- INTEGRAL : les données transmises ne peuvent être modifiées par des tiers
- NONE : pas de protection particulière
- En pratique, les 2 modes CONFIDENTIAL et INTEGRAL sont traités de la même façon par les serveurs d'application (utilisation de HTTPS)
- Il est déconseillé de repasser en mode « normal » après être passé en mode « SSL » ; si on veut le faire tout de même, il est nécessaire d'écrire du code

Exemple

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>r1</web-resource-name>
    <url-pattern>/faces/restr/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>employe<role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>
      CONFIDENTIAL
    </transport-guarantee>
  </user-data-constraint>
</security-constraint>
```


Page d'erreur

- Pour faire afficher une page spéciale si l'utilisateur n'a pas les autorisations pour accéder à une page protégée il suffit d'ajouter ceci dans web.xml :

```
<error-page>
  <error-code>403</error-code>
  <location>/login/noauth.xhtml</location>
</error-page>
```

- Pour faire afficher dans la page noauth.xhtml l'URL de la page interdite :

```
#{requestScope["javax.servlet.error.request_
uri"]}
```


Protection des méthodes des EJB

Par déclaration

- Par défaut, toutes les méthodes sont accessibles à tous les utilisateurs
- Il est possible de restreindre l'accès
  - avec une annotation
  - avec les fichiers de déploiement XML
- Les fichiers XML l'emportent sur les annotations

Annotation @RolesAllowed

- Annote une méthode ou la classe EJB (concerne alors toutes les méthodes de la classe) protégée
- En paramètre, un seul rôle ou plusieurs rôles de type String entourés d'accolades (les utilisateurs ayant un des rôles peuvent utiliser la méthode ou la classe) :

```
@RolesAllowed("role1")
@RolesAllowed({ "role1", "role2" })
```

Annotation @PermitAll

- Peut annoter une classe ou une méthode ; elle indique que toutes les méthodes de la classe annotée, ou la méthode annotée, sont autorisées à tous les utilisateurs
- Si la classe est protégée, cette annotation permet d'exclure une méthode de la protection : tous les utilisateurs seront autorisés à utiliser la méthode annotée par `@PermitAll`

Annotation @DenyAll

- Ne peut annoter qu'une méthode ; elle indique que la méthode annotée n'est accessible par aucun utilisateur
- Cette annotation est rarement utilisée car la méthode annotée ne pourra pas être utilisée à l'intérieur d'un container Java EE

Exemple de déclarations (1/2)

```
<ejb-jar>
<assembly-descriptor>
  <security-role>
    <description>Employés chargés des clients
    </description>
    <role-name>Caissier</role-name>
  </security-role>
  <security-role>
    <description>Contrôleur</description>
    <role-name>Controleur</role-name>
  </security-role>
```


Exemple de déclarations (2/2)

```
  <method-permission>
    <role-name>Controleur</role-name>
    <method>
      <ejb-name>Compte</ejb-name>
      <method-name>getBalance</method-name>
    </method>
    <method>
      <ejb-name>Compte</ejb-name>
      <method-name>setBalance</method-name>
    </method>
  </method-permission>
</assembly-descriptor>
</ejb-jar>
```


Méthodes non protégées

- Certaines méthodes sont autorisées à tous (`<unchecked/>`):

```
<method-permission>
  <unchecked/>
  <method>
    <ejb-name>Compte</ejb-name>
    <method-name>getNames</method-name>
  </method>
  ...
```


Méthodes non accessibles

- Des méthodes peuvent ne pas être accessibles pour un déploiement :

```
<exclude-list>
  <description>Méthode m1 de Ejb1 interdite
  </description>
  <method>
    <ejb-name>Ejb1</ejb-name>
    <method-name>m1</method-name>
  </method>
</exclude-list>
```

Annotation @RunAs

- Peut annoter une méthode d'un EJB (si la classe EJB est annotée, toutes les méthodes de la classe sont concernées)
- Le paramètre est le nom du rôle que l'utilisateur prendra pendant qu'il exécutera la méthode
- Exemple :
  `@RunAs("admin")`
- Même utilité que le « suid » d'Unix : donner temporairement une autorisation à un utilisateur

Balise run-as

- Dans un fichier de configuration :

```
<session>
  <ejb-name>. . . </ejb-name>
  . . .
  <security-identity>
    <run-as>
      <role-name>admin</role-name>
    </run-as>
  </security-identity>
  . . .
</session>
```


use-caller-identity

- Pour être certain qu'on utilisera le principal et pas un rôle donné par une méthode EJB « `run-as` », on peut indiquer `use-caller-identity` pour un EJB ; en ce cas, les rôles transmis seront les rôles liés à la vraie identité de l'utilisateur (sinon ça serait le rôle donné par « `run-as` ») :

```
<security-identity>
  <use-caller-identity/>
</security-identity>
```


AccessLocalException

- Cette exception du paquetage `javax.ejb` est levée si une méthode est appelée alors que les autorisations ne sont pas suffisantes

Références

Implémentation de référence

- Soteria : https://github.com/javaee/security-soteria

- JSR 375 : https://jcp.org/en/jsr/detail?id=375

Exemples

- De nombreux exemples peuvent être trouvés dans l'implémentation de référence Soteria aux adresses https://github.com/javaee-security-spec/soteria/tree/master/test/app-db/src/main/java/org/glassfish/soteria/test et https://github.com/javaee-security-spec/soteria/tree/master/test ; lire en particulier le fichier README.md

Webographie

- Vidéo : https://blogs.oracle.com/theaquarium/entry/devoxx_replay_java_ee_security
- Page pour le projet : https://java.net/projects/javaee-security-spec
- Code dans GitHub : https://github.com/javaee/security-api
- Spécification : https://github.com/javaee/security-spec (on peut générer le pdf à partir des fichiers ou bien les consulter directement, par exemple à l'adresse .../security-spec/blob/master/src/main/doc/authenticationMechanism.asciidoc)

Tutoriel

- Java EE 8 : https://javaee.github.io/tutorial/security-intro.html (peu d'information sur les nouveautés de JSR 375)
- @HashServiceType : https://medium.com/@swhp/explore-hash-feature-on-java-ee-security-jsr-375-soteria-7fe3b1953785
- Playing with JSR 375 : https://medium.com/@swhp/playing-with-java-ee-security-jsr-375-soteria-38e8d2b094d4
- Devoxx (vidéo YouTube de la présentation) : https://www.youtube.com/watch?v=DoSfxPvUj18

- https://www.youtube.com/watch?v=TivynzrVZoc
- https://dzone.com/articles/an-overview-of-java-ee-8s-security-api
- Série de 4 articles très complets, avec des quiz (et format PDF à la fin de chaque article pour imprimer) : https://www.ibm.com/developerworks/java/library/j-javaee8-security-api-1/index.html
- https://www.youtube.com/watch?v=swE9IG3VT_k
- https://www.youtube.com/watch?v=TivynzrVZoc&t=
- Exemple de code : https://github.com/hantsy/ee8-sandbox/tree/master/security-custom-form-db

Bibliographie

- Chapitre 13 du livre « The Definitive Guide to JSF in Java EE 8 » de Bauke Scholtz et Arjan Tijms
- Chapitre 2 du livre « Java EE 8 Only What's New » de Alex Theedom
- Chapitre 9 du livre « Java EE 8 Application Development » de David R. Heffelfinger »