Construction d'applications web

Chapitre 3

Reconnaître un utilisateur d'une requête sur la suivante : cookies, sessions

Limitation du protocole HTTP

- HTTP : protocole sans mémoire
- Requêtes indépendantes
  - Pas de moyen simple de reconnaître un client
- Propagation d'une information de requête en requête pour un même client :
  - Envoie de l'information du serveur chez le client
  - Restitution de cette information par le client au serveur à chaque prochaine requête
  - => Cookies

Initialisation d'un cookie (serveur)

- Positionné dans l'entête HTTP
  - Set-Cookie : Nom=Valeur; expires=Date; path=Chemin; domain=NomDomaine; secure
  - Nom=Valeur : associe une valeur à une variable
  - expires=Date : date d'échéance du cookie. Un cookie sans date expire à la fermeture du client
  - domaine=NomDomain : domaine du serveur à qui le cookie doit être renvoyé
  - path=Chemin : préfixe de l'URL associée à ce cookie

Cookie : envoi

L'envoi du cookie vers le navigateur du client se fait grâce à la méthode *addCookie()* de l'objet *HttpServletResponse*.

```
void addCookie(Cookie cookie)
```

Etant donnée que les cookies sont stockés dans les en-têtes HTTP, et que celles-ci doivent être les premières informations envoyées, la création du cookie doit se faire avant tout envoi de données au navigateur (le cookie doit être créé avant toute écriture sur le flot de sortie de la servlet)

```
Cookie MonCookie = new Cookie("nom", "valeur");
response.addCookie(MonCookie);
```

Cookie : récupération

Pour récupérer les cookies provenant de la requête du client, il suffit d'utiliser la méthode *getCookies()* de l'objet *HttpServletRequest*

```
Cookie[] getCookies()
```

Retourne un tableau contenant l'ensemble des cookies présents chez le client. Il est ainsi possible de parcourir le tableau afin de retrouver un cookie spécifique grâce à la méthode **`getName()`** de l'objet **`Cookie`**.

- La récupération de la valeur d'un cookie se fait grâce à la méthode getValue() de l'objet Cookie

```
String Valeur = Cookie.getValue()
```

Voir lien vers la javadoc sur Chamilo

Renvoi d'un cookie par le client

- Accès par le client à serveur/URL
  - Recherche par le client parmi sa liste de cookies de ceux dont
  - Serveur appartient au domaine
  - URL est préfixé par path
- Ajout dans la requête HTTP du client :

  Cookie : Nom1=Valeur1 ; Nom2=Valeur2 ; ...

Limitations des cookies

Résumé du fonctionnement :

- le serveur demande dans sa réponse au logiciel client de stocker des données qu'il lui indique et de les lui renvoyer telles quelles la prochaine fois qu'il le contactera.
- En fonctionnement « normal » (le client est un navigateur web avec une configuration standard), le client fait ce que le serveur lui demande.
- Mais en général le serveur n'a **strictement aucune garantie** sur le comportement du client, qui n'est pas forcément un navigateur mais n'importe quoi qui envoie des requêtes HTTP...
- l'utilisateur peut par exemple écrire une requête HTTP à la main et l'envoyer au serveur avec un logiciel comme telnet.

Limitations des cookies

En particulier, le client peut :

- se faire passer pour n'importe quel navigateur standard (il suffit de copier le format de requêtes de celui-ci)
- ne pas stocker ou ne pas renvoyer les cookies (possible y compris avec un navigateur standard : c'est un choix de configuration utilisateur)
- mais surtout : envoyer des cookies créés de toute pièces contenant des données arbitraires définies par l'utilisateur (et non préalablement fournies par le serveur)
  - le serveur ne doit **jamais** utiliser sans les vérifier des données fournies par un client !

Limitations des cookies

Exemple (injection SQL) :

- un servlet contient :

```
String nom = "inconnu";
Cookies[] cook = request.getCookies();
for (Cookie c : cook) {
  if c.getName().equals("user") nom = c.getValue();
}
String sql =
 ("select * from users where name='" + nom + "';");
```

- le client envoie dans le cookie :

```
user=" '; drop table users; --"
```

- si la requête est exécutée : la table est détruite

Remarque : on a le même problème avec `request.getParameter()` (données de formulaire)

Sessions

Pour contourner ces limitations :

- Le serveur conserve lui-même toutes les données dont il a besoin sur un visiteur du site
- Pour reconnaître ce visiteur d'une requête sur l'autre, il lui confie un cookie comprenant une seule valeur : une longue chaîne aléatoire qui sert d'**identifiant de session**
- Cet identifiant **expire après quelques minutes** quand ce visiteur cesse d'envoyer des requêtes
- Un utilisateur peut présenter un faux identifiant, mais la probabilité qu'il corresponde à un vrai est très faible
- Il reste possible pour un utilisateur de ne pas renvoyer son identifiant.

Gestion de sessions

- Les sessions sont gérées par le container de servlets
  - On récupère un objet `HttpSession` à partir de la requête (`HTTPServletRequest`)
  - Cet objet permet d'associer des valeurs quelconques (de type `Object`) à des chaînes de caractères. Son interface permet :
    - Gestion des objets dans la session : ajout, modification, retrait, recherche, etc.
    - Gestion des informations de la session elle-même : date de création, id de la session, etc.
- Exemple :

```
HttpSession session = request.getSession(true);
```

Renvoie la session associée à l'utilisateur ayant émis cette requête si elle existe, sinon en crée une (true).

Voir le lien vers la documentation javadoc sur Chamilo

Gestion de sessions

- Quand on appelle `getSession(true)` : l'utilisateur se voit attribuer un Session ID
- Le Session ID est communiqué au client dans la réponse :
  - Option 1: si le navigateur supporte les cookies, le serveur crée un cookie avec le session ID. Dans Tomcat, ce cookie est appelé `JSESSIONID`.
  - Option 2: si le navigateur ne supporte pas les cookies, le serveur peut essayer d'ajouter l'identifiant aux URL des liens internes :

http://mon.serveur/mon.appli/ma/page;jsessionid=123abc...

Inconvénient : si l'utilisateur envoie à quelqu'un d'autre un lien vers la page, l'autre personne récupère aussi la session...

Mise à jour des données dans la session

- On parle d'« attributs » de session :

```
HttpSession session = request.getSession();
session.setAttribute("accessCount", "10");
```

- Et on peut les supprimer :

```
session.removeAttribute("name");
```

Récupération des données

- L'objet session fonctionne comme une `Map`
  - Peut stocker n'importe quel type d'objet,
  - Les objets sont accessibles par leur nom (qui joue le rôle de clé), qui est une chaîne de caractères
- Exemple de récupération :

```
Integer accessCount =
  (Integer) session.getAttribute("accessCount");
```

- Récupérer toutes les « clés » de tous les objets dans la session :

```
Enumeration<String> attributes =
   request.getAttributeNames();
```

Autres informations de session

- Récupérer le session ID, par exemple : `gj9xswvvw9p`

```
public String getId();
```

- Voir si la session vient juste d'être créée :

```
public boolean isNew();
```

- Récupérer la date de création :

```
public long getCreationTime();
```

- Dernière fois que la session a été activée (ex dernière date de connexion)

```
public long getLastAccessedTime();
```

- Pour terminer (détruire) une session :

```
public void invalidate();
```

Listeners

- Ecouter les phases de vie des objets serveurs
- Exemple : destruction d'une session
  - Réservation de places de spectacles
  - Panier : places réservées mais non payées
  - Rangées en session mais aussi en BD (sinon, risque de réservation par plusieurs clients)
  - Si le propriétaire du panier ne concrétise pas :
    - Places perdues
  - Solution :
    - Détecter la fin de session (timeout)
    - Libérer les places du panier

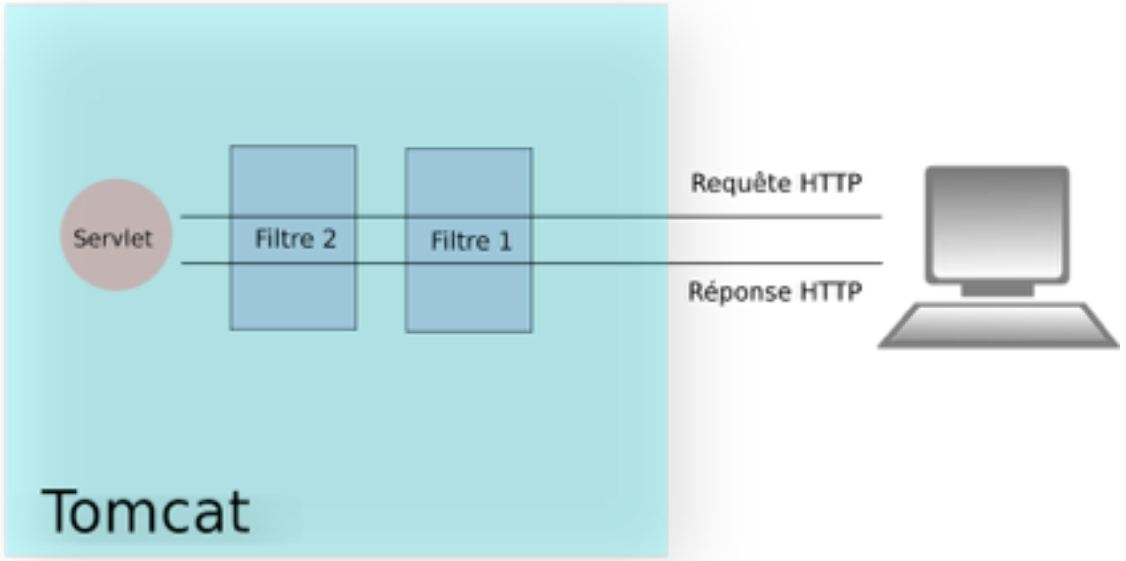
Listeners

Exemple de code :

```
package fr.ensimag;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;


@WebListener()
public class SessionListener implements HttpSessionListener {

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.err.println("Creation de la session");
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        System.err.println("Destruction de la session");
    }
}
```

Filtres

- Fonctionnalité orthogonale :
  - S'ajoute de manière transparente à une application
- Principe :
  - Modifier le contenu de la requête et de la réponse
  - Possiblité de mise en place d'une chaîne de filtres



Filtres

- Exemple
  - Authentification
  - Log
  - Mesure de performance
  - Cache
  - Compression
  - Watermarking
  - Etc.

Exemple : durée d'une requête


```
@WebFilter(filterName = "Perfomance", urlPatterns = {"/*"})
public class Perfomance implements Filter {

    public void doFilter( ServletRequest request,
                          ServletResponse response,
                          FilterChain chain)
            throws IOException, ServletException {

            long start = System.currentTimeMillis();
            chain.doFilter(request, response);
            long end = System.currentTimeMillis();
            System.err.print("Duree : ");
            System.err.println(end-start);
    }

    @Override
    public void init(FilterConfig filterConfig)
                          throws ServletException {
    }

    @Override
    public void destroy() {
    }
}
```

Appel du prochain Filtre (ou de la servlet si dernier)