Servlets

Université Française d'Égypte
Richard Grin
Version 0.9 – 12/10/12

Plan (1/2)

- Application Web
- Présentation des servlets
- Ecrire un servlet
- Traitement des formulaires par les servlets
- Forward et redirect
- Portées
- Filtres
- Ecouteurs
- Traitements asynchrones

Plan (2/2)

- Structure d'une application Web
- Compléments
- Au-delà des servlets

Application Web

Définition

- Une application Web est un logiciel manipulable grâce à un navigateur Web
- L'interface utilisateur est constituée de pages Web
- Lorsque l'utilisateur soumet un formulaire HTML, une requête est envoyée au serveur Web
- Cette requête est traitée par l'application Web qui retourne en réponse une page Web à l'utilisateur
- La page Web renvoyée est presque toujours une page Web dynamique créée « à la volée » par l'application

Exemple

- L'utilisateur a tapé le nom d'un département d'une entreprise dans un formulaire HTML
- Il soumet ensuite le formulaire
- L'application reçoit ce nom sur le serveur
- Elle interroge la base de données pour chercher les noms et salaire des employés de ce département
- Elle construit une page HTML qui contient une table avec ces informations et l'envoie en réponse à l'utilisateur

Contexte de l'application

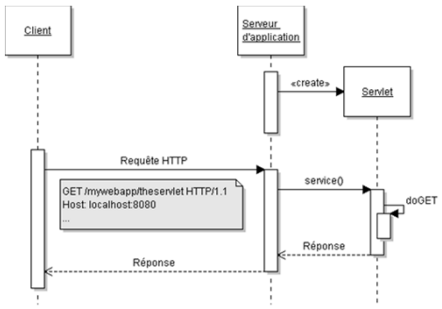
- Un serveur Web peut héberger plusieurs applications Web
- Les requêtes HTTP sont traitées par une application ou une autre suivant l'URL de la requête
- Chaque application a son propre contexte qui est la partie de l'URL placée juste après le nom de la machine serveur :
  `http://www.machin.com/nomAppli/rep/index.html`

Présentation des servlets

Fonctionnalité de base

- De base un serveur Web ne fournit que des pages HTML statiques
- Un servlet est :
  - un composant logiciel écrit en Java
  - qui s'exécute en programme « compagnon » d'un serveur Web
- Il reçoit des requêtes de clients HTTP et construit des pages dynamiques écrites en HTML, qui sont renvoyées en réponse aux clients

Fonctionnement d'un servlet



Exemple de servlet

```
public class HelloServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>");
    out.println("Hello");
    out.println("</title></head><body>");
    out.println("<p>Hello World !</p>");
    out.println("</body></html>");
    out.close();
  }
}
```


Structure d'une méthode get

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
     throws ServletException, IOException {
  // Utiliser request pour lire les paramètres
  // et les headers de la requête
  ...
  // Traiter la requête
  ...
  // Utiliser response pour spécifier le code de
  // statut et les headers de la réponse
  ...
  PrintWriter out = response.getWriter();
  // Envoyer le contenu de la réponse dans out
  ...
}
```

Conteneur de servlets

- Les serveurs HTTP ne savent pas exécuter directement le code Java d'un servlet
- Les servlets nécessitent un conteneur de servlets qui se charge de la gestion des servlets :
  - gestion des noms des servlets (associés à une classe Java)
  - création et initialisation des servlets
  - suppression des servlets
- Le serveur Web lui passe la main quand il reçoit d'un client une requête qui doit être traitée par un servlet (reconnaissable à l'URL)

URL d'un servlet

- Le client ne peut donner une adresse directe du servlet
- L'application Web doit établir une correspondance (un *mapping*) entre le servlet et un URL (avec une annotation ou le fichier web.xml)
- Demander cet URL passera la main au conteneur de servlets qui lancera l'exécution du servlet

Annotations

- Depuis la spécification Servlet 3.0, il est possible de se passer de fichier web.xml pour déclarer un servlet
- Il suffit d'annoter la classe du servlet par l'annotation **`@WebServlet`** qui possède des attributs pour donner toutes les informations nécessaires au fonctionnement du servlet : nom, pattern URL,…

Exemple

```
@WebServlet(urlPatterns={"/liste"})
public class Servlet1 extends HttpServlet {
  ...
}
```

http://machin.com/appli/liste

Fichier **`web.xml`**

- Le *mapping* peut aussi être donné dans le fichier **`web.xml`** qui décrit toute application Web Java (étudié plus loin en détails)
- Ce fichier sera contenu dans un fichier WAR (extension « .war »), fichier ZIP qui enveloppe tous les composants, code et ressources de l'application Web

Exemple de web.xml (1/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>HelloServlet</welcome-file>
  </welcome-file-list>
```

Exemple de web.xml (2/2)

```
<servlet>
  <servlet-name>HelloWorld</servlet-name>
  <servlet-class>p1.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>UneServletSimple</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

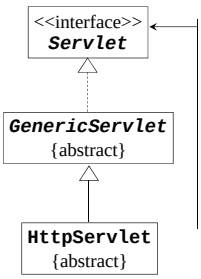

Joker dans les mappings

- Le modèle pour l'URL peut contenir un joker (*) à la fin d'un URL qui commence par « / » ou juste avant une extension
- Par exemple
  - **`<url-pattern>/servlet/*</url-pattern>`**
  - **`<url-pattern>*.serv</url-pattern>`**
- En ce cas, tous les URL qui correspondent à ce modèle lanceront l'exécution du servlet concerné par ce modèle

Les paquetages

- Le paquetage javax.servlet contient les classes et interfaces utilisées pour écrire des servlets
- Le paquetage javax.servlet.http contient les classes et interfaces pour écrire des servlets qui fonctionnent avec le protocole HTTP (c'est presque toujours le cas)

Classes et interfaces de base



`init` appelé par le conteneur à la création du servlet
`service` traite une requête
`destroy` appelé par le conteneur à la suppression du servlet
`getServletConfig` donne accès aux paramètres d'initialisation du servlet (web.xml)
`getServletInfo` informations sous forme de **`String`**

Classes et interfaces

- L'interface ServletRequest représente la requête du client au serveur
- L'interface ServletResponse représente la réponse du serveur
- Les interfaces HttpServletRequest et HttpServletResponse sont des interfaces filles pour travailler avec le protocole HTTP (ce qui est presque toujours le cas)

Traitements effectués par les servlets

- Un servlet traite les requêtes des clients Web par sa méthode **`public void service(ServletRequest, ServletResponse)`**
- C'est la méthode abstraite qu'il faut définir si on hérite de **`GenericServlet`**
- En fait, la plupart des servlets sont des servlets HTTP qui héritent de **`HttpServlet`** et il vaut mieux alors ne pas toucher à la méthode service et redéfinir les méthodes **`doGet`**, **`doPost`**,...

Traitements effectués par les servlets HTTP

- La méthode **`service`** de la classe **`HTTPServlet`** délègue le traitement à une autre méthode, selon le type de requête HTTP envoyée par le client :
  - **`doGet()`**
  - **`doPost()`**
  - **`doPut()`**
  - **`doDelete()`**
  - **`doHead()`**
  - **`doOptions()`**
  - **`doTrace()`**

Écrire un servlet

Code d'un servlet

- Écrire un servlet HTTP c'est
  - écrire une classe qui hérite de **`HttpServlet`**
  - redéfinir au moins une des méthodes **`doGet`**, **`doPost`**,…
  - redéfinir éventuellement une des méthodes **`init`** ou **`destroy`** si le servlet gère des ressources qu'il faut initialiser ou supprimer

HttpServletRequest

- Décrit la requête HTTP reçue par le serveur (headers, cookies envoyés par la requête, paramètres de la requête,…)
- Contient de nombreuses méthodes
- Permet en particulier de récupérer les paramètres reçus depuis un formulaire HTML

HttpServletResponse (1/2)

- Correspond à la réponse qui sera envoyée au client
- Contient de nombreuses méthodes pour décrire la réponse :
  - indiquer le code de statut (par exemple 200 si tout s'est bien passé) de la réponse
  - donner les headers de la réponse
  - ajouter un cookie à la réponse

HttpServletResponse (2/2)

- Contient la méthode **`getWriter()`** pour obtenir le flot de sortie pour générer la page HTML résultat ; le servlet envoie le code de la page HTML qu'il génère dans ce flot
- Contient une méthode pour envoyer au client une réponse de redirection ou un message d'erreur

Paramètres d'initialisation d'un servlet

- Ils permettent de donner des informations qui pourront être utilisées par le servlet ; par exemple l'adresse d'une base de données
- Les valeurs sont données dans le fichier web.xml ou dans l'annotation @WebServlet
- Le fichier web.xml l'emporte sur les annotations ce qui permet de modifier ces informations sans devoir recompiler l'application

Exemple d'initialisation dans web.xml

```
<servlet>
  <servlet-name>Servlet1</servlet-name>
  <init-param>
    <param-name>p1</param-name>
    <param-value>v1</param-value>
  </init-param>
 <init-param>
    <param-name>p2</param-name>
    <param-value>v2</param-value>
  </init-param>
</servlet>
```


Exemple d'initialisation avec annotation

```
@WebServlet(
  name = "Servlet1",
  urlPatterns = {"/add1"},
  initParams = {
    @InitParam(name="p1", value="v1"),
    @InitParam(name="p2", value="v2")} )
public class Servlet1 extends HttpServlet {
  ...
```


Récupérer les paramètres d'initialisation

- Ces paramètres d'initialisation peuvent être récupérés dans le code du servlet grâce à des méthodes de l'interface **`ServletConfig`**
- L'interface Servlet contient la méthode **`getServletConfig()`** qui renvoie un **`ServletConfig`**
- **`HttpServlet`** implémentant **`ServletConfig`**, ces méthodes sont directement disponibles dans un servlet HTTP

Récupérer les paramètres d'initialisation

- ServletConfig contient les méthodes
  - **`Enumeration<String> getInitParameterNames()`** qui retourne une énumération des noms des paramètres
  - **`String getInitParameter(String)`** qui retourne la valeur du paramètre d'initialisation dont le nom est passé en paramètre

Exemple

```
// Dans le code d'un HttpServlet :
// Récupérer tous les noms
Enumeration<String> e = getInitParameterNames();
while (e.hasMoreElements()) {
  String nom = e.nextElement();
  ...
}
// Récupérer une valeur
String v = getInitParameter(nom);
...
}
```

Contexte d'une application

- Plusieurs applications peuvent être hébergées par le même serveur Web
- Chaque application Web « Java » est décrite par un fichier web.xml (ou par des annotations) et définit un contexte (interface **`ServletContext`**) qui représente le contexte dans lequel l'application s'exécute
- Durant l'exécution c'est l'URL donné par le client qui détermine le contexte (l'application) utilisé

Chemin du contexte

- Si le code Java du servlet a besoin de désigner le contexte de l'application, il peut l'obtenir en interrogeant la requête par : **`request.getContextPath()`**
- Par exemple, pour l'URL « http://machin.com/appli/liste », la méthode renverra « /appli »

Remarque sur les chemins

- Si on est dans la page http://localhost:8080/Servlets/Servlet1 (/Servlets est le contexte) et qu'on veut désigner http://localhost:8080/Servlets/Servlet2, on peut donner le chemin ainsi :
  - « **/**Servlets/Servlet2 » (chemin absolu qui commence par « / » ; il faut préfixer par le contexte)
  - ou « Servlet2 » (chemin relatif)

ServletContext

- Il est possible de récupérer le contexte depuis le code d'un servlet
  - par la méthode **`getServletContext()`** de l'interface **`ServletConfig`**
  - ou directement, par la méthode **`getServletContext()`** de la classe **`HttpServlet`** (qui implémente **`ServletConfig`**)

Paramètre d'initialisation du contexte

- Comme pour les servlets, on peut donner des paramètres d'initialisation d'un contexte
- Ces paramètres seront connus de tous les servlets de l'application
- Initialisation d'un paramètre dans web.xml :

```
<web-app>
  <context-param>
    <param-name>p1</param-name>
    <param-value>v1</param-value>
  </context-param>
  ...
```


Récupérer les paramètres d'initialisation

```
ServletContext contexte = getServletContext();
// Récupérer tous les noms
Enumeration<String> e =
  contexte.getInitParameterNames();
while (e.hasMoreElements()) {
  String nom = e.nextElement();
  // Récupérer une valeur
  String v = contexte.getInitParameter(nom);
  ...
}
```

Cycle de vie d'un servlet

Une seule instance de servlet

- Le plus souvent le serveur d'application ne crée qu'un seul objet servlet par application et par type de servlet
- Le servlet est créé à la première connexion à un URL qui correspond (url-mapping) au servlet
- Le servlet est supprimé quand l'application est retirée du serveur (undeploy)

Un thread par client

- Le container de servlet crée un thread pour traiter chaque connexion ultérieure qui doit être traitée par le même type de servlet
- Il ne faut donc pas garder d'information particulière à un client dans les variables d'instance des servlets
- Et les éventuelles informations conservée dans le servlet doivent être protégées contre les accès concurrent

Initialisation du servlet

- Juste après la création par le container du servlet (par le constructeur sans paramètre), la méthode init(ServletConfig) est exécutée
- Cette méthode peut initialiser des ressources qui seront utiles pendant l'existence du servlet, en utilisant éventuellement les paramètres d'initialisation

Méthodes init(ServletConfig) et init()

- Si on redéfinit cette méthode, la première ligne doit être
  super.init(SerletConfig)
  afin que la configuration soit enregistrée et disponible pour les futures utilisation du servlet
- Pour ne pas faire d'erreur, il vaut mieux redéfinir la méthode init() (sans paramètre) qui est appelée par la méthode init(SerletConfig)

Méthode destroy()

- Méthode à redéfinir si on veut faire le ménage avant la suppression du servlet (fermer des ressources par exemple)

Forward et redirect

Forward et redirect

- Le code d'un servlet peut passer la main à une autre partie de l'application (autre servlet par exemple) ou même à une autre application, ou afficher une page Web quelconque
- Il peut passer la main directement (forward = transmettre), ou renvoyer une réponse au client pour lui demander de renvoyer une requête vers l'autre partie de l'application (redirect = rediriger)

Forward

- Le servlet passe la main à une autre partie de l'application, en lui transmettant la requête qu'il a reçu et la référence pour répondre au client de l'application
- Le navigateur Web n'est pas au courant du changement d'URL (d'où l'affichage d'une adresse erronée)

Exemple

```
public void doGet (
       HttpServletRequest request,
       HttpServletResponse response)
   throws ServletException, IOException {
  ...
  request.getRequestDispatcher("/servlet1")
         .forward(request, response);
}
```


Redirect

- Pour passer la main en dehors de l'application, il faut utiliser la redirection
- La redirection peut aussi se faire vers une page de la même application
- Le serveur demande au navigateur de faire une requête vers un autre URL
- Le navigateur affiche le bon URL, ce qui permet d'avoir un URL que l'on peut ajouter dans ses marques-pages

Exemple

```
public void doGet (
       HttpServletRequest request,
       HttpServletResponse response)
   throws ServletException, IOException {
  ...
  response.sendRedirect
    ("http://.../index.html?p1=val1");
}
```

Portées

Conserver des données

- Une requête peut être traitée par plusieurs servlets (par forward)
- Les servlets permettent de conserver des données pendant une requête de l'utilisateur ou même entre 2 requêtes, pendant une session de travail de l'utilisateur
- Ils peuvent même partager des données avec tous les servlets de l'application (y compris ceux qui sont utilisés par d'autres utilisateurs)

3 portées pour conserver des données

- *Requête* : depuis l'arrivée de la requête jusqu'au renvoi de la réponse au client
- *Session* : de la première connexion de l'utilisateur jusqu'à expiration par timeout (optionnel) ou fermeture explicite de la session (ou arrêt du serveur)
- *Application* : toute la durée de vie de l'application

Accès aux données

- On a accès aux données conservées, par l'interface **`HttpServletRequest`** (portée requête) ou par des méthodes de cette interface :
  - session : **`HttpSession getSession()`**
  - application : **`ServletContext getServletContext()`**

Attributs

- On peut conserver des objets (types primitifs interdits) dans les différentes portées en utilisant la notion d'attribut
- Ranger un objet val1 dans la session :

```
HttpSession session = request.getSession();
session.setAttribute("p1", val1);
```

- Récupérer l'objet dans la session :

```
HttpSession session = request.getSession();
Type1 v1 =
  (Type1)session.getAttribute("p1", val1);
```

- Le code est semblable pour les portées application et requête

Attributs récupérables

- Les attributs mis dans les portées session et requête ne sont récupérables que par l'utilisateur qui les y a mis
- Les attributs mis dans la portée application peuvent être récupérés par tous les utilisateurs

Moyens pour suivre une session

- 3 moyens classiques pour suivre un utilisateur durant une session :
  - Cookies : courte information déposée dans le disque de l'utilisateur
  - Réécriture des URL : on ajoute une information sur la session à toutes les url qui référencent le serveur et qui sont envoyées au client ; par exemple, http://serveur/chemin/truc.html;session=132
  - Champs cachés dans les formulaires donnés à remplir par les utilisateurs
    <INPUT TYPE=« HIDDEN » NAME=« session » VALUE=« 132 »>

Suivi de session avec les servlets

- L'API pour les servlets cache la complexité liée au suivi de session
- Elle utilise par défaut les cookies
- Si le navigateur ne sait pas gérer les cookies ou si l'utilisateur ne les a pas autorisés, l'API utilisera la réécriture d'URL
- Tout ce suivi de session est caché au développeur

Session

- La classe HTTPSession représente les sessions
  `HTTPSession session = request.getSession();`
- On peut tester si la session vient d'être créée par **`session.isNew()`**
- On peut ensuite ranger et récupérer des objets dont la valeur sera conservée pendant toute la durée de la session par les méthodes **`setAttribute`** et **`getAttribute`**
- **`getAttributeNames`** renvoie une **`Enumeration`** pour avoir les noms de tous les attributs associés à la session en cours

Fermeture d'une session

- Une session se termine automatiquement quand la durée de timeout est dépassée sans une connexion de l'utilisateur
- Le timeout est donné par **`getMaxInactiveInterval`**
- On peut aussi fermer une session par la méthode **`invalidate`** de l'interface **`HttpSession`**
- Si la classe d'un objet associé à la session implémente l'interface **`HTTPSessionBindingListener`**, il est prévenu au moment où il n'est plus associé à la session (et aussi s'il redevient associé) ; si la session se termine, il est donc prévenu

API pour la réécriture d'URL

- Si on veut pouvoir utiliser la réécriture d'URL pour le suivi de session, on doit utiliser des URL « encodées » pour les URL qui référencent le serveur et qui sont écrites dans les pages envoyées au client en réponse :

```
String url = "….. ";
String urlEncode = response.encodeURL(url);
out.println("<A HREF=\""  + urlEncode
            + "\"> . . . </A>");
```


Traitement des formulaires par un servlet

Traitement des formulaires par un servlet

- Nous allons étudier comment récupérer et traiter les données entrées par l'utilisateur dans un formulaire
- Les données tapées par l'utilisateur sont conservées dans des paramètres dont les noms sont liés aux noms HTML des composants du formulaire (attribut « NAME » du tag HTML)

HttpServletRequest

- Interface qui représente une requête d'un client Web
- Elle contient plusieurs méthodes pour récupérer les valeurs des paramètres :
  - **`String getParameter(String nomParam)`** récupère la valeur d'un paramètre
  - **`String[] getParameterValues()`** si le paramètre peut avoir plusieurs valeurs (par exemple pour les boîtes à cocher)
  - **`Enumeration<String> getParameterNames()`** fournit les noms de tous les paramètres

Traitement des paramètres

- Les méthodes pour récupérer les paramètres sont indépendantes du type de la requête (POST ou GET)
- Pour couvrir les 2 types de requête avec un seul code, il suffit que **`doGet`** appelle **`doPost`** (ou l'inverse)

POST et GET

- Exemple d'URL avec GET : http://www.truc.com/abonnement?id=23378&nom=Toto&op=d&v=56
- Avec la méthode POST de HTML, les paramètres sont passés dans la requête et pas dans l'URL

Exemple de code

- Code HTML :

```
<input type="text" name="nom">
<input type="checkbox" name="ville"
       value="Nice">Nice<br>
<input type="checkbox" name="ville"
       value="Paris">Paris<br>
```

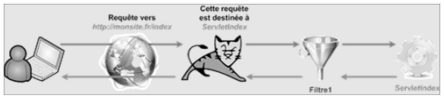
- Code servlet :

```
String nom = req.getParameter("nom");
String[] villes =
  req.getParameterValues("ville");
```


Filtres

Utilité

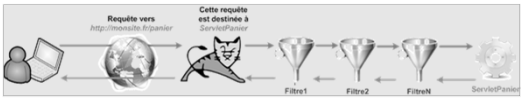
- Un filtre est un objet qui intercepte les requêtes avant qu'elles ne soient traitées par un servlet ou/et qui traite les réponses juste après qu'elles aient été générées par un servlet



Cette image et la suivante sont reprises
de l'article de « cysboy » sur les filtres :
http://www.siteduzero.com/tutoriel-3-427660-les-filtres.html#ss_part_1

Chaînage de filtres

- Les filtres peuvent être chaînés à la manière des commandes dans un pipe Unix :



Déclaration d'un filtre

- On déclare un filtre par la balise **`<filter>`** du fichier web.xml ou en annotant la classe du filtre par **`@WebFilter`**
- On indique les requêtes qui seront filtrées par la balise **`<filter-mapping>`** de web.xml ou en renseignant un attribut de l'annotation **`@WebFilter`** ; on peut donner
  - le nom d'un ou plusieurs servlets
  - ou un ou plusieurs modèles d'URL

Paramètres d'initialisation

- Comme pour les servlets, on peut donner des paramètres d'initialisation à un filtre
- On peut utiliser pour cela
  - la balise <init-param> (dans la balise <filter>) dans le fichier web.xml
  - ou l'annotation @WebInitParam, directement ou dans l'attribut initParams de l'annotation @WebFilter

Exemple d'annotation

```
@WebFilter(
  filterName = "Filtre1",
  urlPatterns = {"/*"},
  initParams = {
    @WebInitParam(name="p1", value="v1")
  }
)
public class Filtre1 implements Filter {
```


Dans web.xml

```
<filter>
  <filter-name>Filtre1</filter-name>
  <filter-class>fr...Filtre1</filter-class>
  <init-param>
    <param-name>p1</param-name>
    <param-value>v1</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Filtre1</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Exemple (1)

```
@WebFilter(filterName="filtre1",
urlPatterns={"/servlet1}", "/autre")
public class Filtre1 implements Filter {
  private FilterConfig filterConfig;

  @Override
  public void init(FilterConfig config) {
    this.filterConfig = config;
  }
```


Exemple (2)

```
@Override
 public void doFilter(ServletRequest request,
   ServletResponse response,
   FilterChain chain) {
   // Traitement requête avant servlet
   ...
   filterChain.doFilter(request, response);
   // Traitement de la réponse du servlet
   ...
 }
```


Ordre d'exécution des filtres

- Si plusieurs filtres peuvent s'appliquer, l'ordre d'exécution est déterminé par l'ordre de déclaration des **`<filter-mapping>`** dans le fichier web.xml (pas possible avec l'annotation)
- En appelant doFilter, on appelle le prochain filtre (dans le cas où on est dans le dernier filtre, on envoie la requête au servlet, ou on envoie la réponse au client)

Écouteurs

Définition

- Objet qui va intervenir à certains moments du traitement des requêtes
- La classe de l'objet doit implémenter des interfaces définis par l'API des servlets qui déterminent le type de l'écouteur : ServletContextListener, ServletContextAttributeListener, ServletRequestListener, ServletRequestAttributeListener, HttpSessionListener, HttpSessionAttributeListener

Déclaration / initialisation

- Balise <listener> dans web.xml ou annotation @WebListener

Implémentation

- Les méthodes à implémenter dépendent du type de l'écouteur
- Pour le type **`ServletContextListener`**, les méthodes à implémenter sont les suivantes :
- **`contextInitialized(ServletContextEvent)`** qui sera exécutée au démarrage de l'application
- **`contextDestroyed(ServletContextEvent)`** qui sera exécutée à la fin de l'application

Exemple d'utilisation

- Initialiser des attributs du contexte de l'application qui seront ensuite utilisés tout au long du fonctionnement de l'application

Exemple

```
@WebListener
public class MonEcouteurDeContext
       implements ServletContextListener {
  public void
    contextInitialized(ServletContextEvent ev){
    Executor executor =
      new ThreadPoolExecutor(...);
    ev.getServletContext().setAttribute(
      "pool", executor);
  }
 ...
```


Traitements asynchrones

Présentation

- Depuis Servlet 3.0 il est possible de lancer un traitement en arrière-plan pour libérer plus rapidement le thread qui traite la requête
- Il faut pour cela configurer le servlet pour qu'il supporte les traitements asynchrones dans web.xml ou par l'attribut « asyncSupported=true » de l'annotation@WebServlet

Fonctionnement

- La méthode **`HttpServletRequest.startAsync()`** indique que la réponse ne sera pas rendue à la fin de la méthode
- Cette méthode renvoie un **`AsyncContext`** qui pourra être utilisé pour traiter la fin de la requête et rendre la réponse plus tard, par un pool de threads dédiés à ce travail (le plus souvent les instances de **`AsyncContext`** sont mises dans une file d'attente pour être traitées à tour de rôle par les threads du pool)

AsyncContext

- Représente un context d'exécution pour une requête asynchrone ; contient le contexte de la requête et en particulier les paramètres et la réponse (**`Response`**)
- La méthode **`complete()`** termine l'opération asynchrone
- Une alternative pour terminer l'opération asynchrone est d'appeler la méthode **`dispatch`** pour indiquer de passer la main à l'URI de départ ou à un autre URI
- Il est possible aussi de donner un délai (timeout) après lequel l'opération asynchrone se termine

Utilisation typique de AsyncContext (1/2)

- Lorsque le servlet, disons la méthode doGet, doit accomplir une tâche qui peut prendre du temps, le servlet appelle la méthode **`startAsync`** de la classe **`Request`**, qui renvoie un **`AsyncContext`**
- Ca signifie que la fin de la méthode **`doGet`** ne sera pas la fin du traitement de la requête
- Le servlet range le contexte asynchrone (ou un **`Runnable`**, ou une variante comme **`Callable`**, qui connait ce contexte) dans un attribut de la requête (le plus souvent de type liste)

Utilisation typique de AsyncContext (2/2)

- Un autre composant de l'application récupère les éléments de la liste pour fournir la tâche qui peut prendre du temps à un pool de threads
- La tâche peut utiliser le contexte asynchrone pour envoyer des messages au client : asyncContext.getResponse().getWriter().print(…), pour fournir une page HTML et appeler complete, ou pour donner la main à un autre composant (souvent une page JSF) par la méthode dispatch

Utilité

- Dans le cas où le thread est en attente d'une ressource longue à obtenir, cette façon de faire permet de libérer le thread pour d'autres tâches, en attendant d'avoir la ressource
- C'est aussi une façon de simuler du « *push* » par le serveur : le serveur peut « pousser » des informations sur le client sans avoir été sollicité juste avant par une requête du client (utile pour les applications où plusieurs clients peuvent interagir pour changer leur environnement)

Exemple (1)

```
@WebServlet(name=…, urlPatterns=…,
    asyncSupported=true)
public class Servlet1 extends HttpServlet {
  ...
  public void doGet(…) throws … {
    ...
    // final car utilisé dans la classe
    // interne Runnable qui suit
    final AsyncContext ac =
      request.startAsync();
```


Exemple (2)

```
    ac.start(new Runnable() {
      ...
      ac.getResponse().getWriter()
        .println(…);
      ac.complete();
    }
  }
}
```

Structure d'une application Web

Généralités

- La spécification des servlets normalise la présentation des applications Web
- Cette normalisation améliore la portabilité des applications

Fichier .war

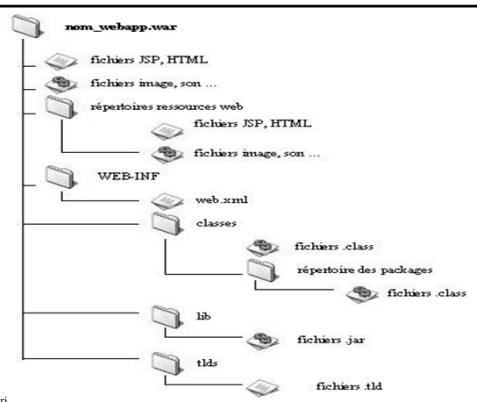
- Tous les fichiers utilisés par une application Web peuvent être packagés sous forme d'un fichier jar
- Un tel fichier doit avoir un suffixe .war pour les distinguer des autres fichiers jar

Structure d'un fichier war

- Répertoire racine contient les fichiers servis aux clients et les ressources qu'ils utilisent :
  - pages statiques html
  - pages JSF ou JSP
  - ressources (images, fichiers CSS, fichiers javascript,…)
- Sous-répertoire WEB-INF qui contient les classes Java utilisées par l'application et les fichier de configuration de l'application
- Sous-répertoire META-INF qui contient le fichier manifest (jar)

Répertoire WEB-INF

- Sous-répertoire **`classes`** qui contient les classes servlets et les classes utilisées par ces servlets (javabeans par exemple)
  - L'emplacement exact d'une classe est déterminé par le nom de son paquetage
- Sous-répertoire **`lib`** qui contient les fichiers jar utilisés par l'application
- Fichier standard **`web.xml`** qui contient les indications pour configurer et déployer l'application, et d'autres fichiers de configuration (beans.xml, faces-config.xml, glassfish-web.xml,…)



Exemple de web.xml

```
<servlet>
  <servlet-name>abonnement</servlet-name>
  <servlet-class>
    abonnement.Abonnement
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>abonnement</servlet-name>
  <url-pattern>index.html</url-pattern>
</servlet-mapping>
```


Divers compléments

Lire/écrire dans un fichier

- La difficulté est d'indiquer où se trouve le fichier dans lequel on veut lire ou écrire
- Plusieurs possibilités :
  - donner un nom de ressource (par rapport au *classpath* ou à la classe qui travaille avec le fichier) et utiliser **`Class.getResource`**
  - utiliser la méthode **`ServletContext.getRealPath`** qui donne le chemin réel qui correspond à un chemin virtuel (le chemin pour l'application) ; retourne une **`String`** (**`null`** si rien n'a été trouvé) ; une instance de **`ServletContext`** peut s'obtenir par **`this.getServletContext()`**

Lire/écrire dans un fichier

- Si on veut pouvoir changer le chemin du fichier sans avoir à recompiler, on peut utiliser un paramètre de l'application (**`<init-param>`** de **`<servlet>`** du fichier **`web.xml`**) pour indiquer le chemin

Mettre au point

- Regarder le HTML généré
- Envoyer des messages de log sur le serveur
- Utiliser des outils tels que Firebug pour lire le contenu des requêtes et des réponses HTTP
- Méthode log de HttpServlet pour envoyer des messages de log sur le serveur
- 2 variantes : avec String et avec Throwable

Au delà des servlets

Plus loin avec les servlets...

- Les servlets ne favorisent pas la séparation entre la présentation des données et les traitements métier
- Cette séparation peut être améliorée en utilisant les pages JSF (un autre cours porte sur JSF)

Frameworks au-dessus des servlets

- Les applications Web actuelles utilisent rarement directement les servlets
- Elles utilisent des frameworks qui utilisent en interne des servlets