LISTENERS ET FILTRES WEB

i. Les listeners Web
ii. Déclaration des listeners Web
iii. Exemple d'utilisation d'un listener
iv. Les filtres de Servlet
v. Déclaration des filtres
vi. Implémentation d'un filtre
vii. Cas d'utilisation de filtres

Les servlets ne sont pas les seuls composants gérés par le conteneur Web, ce dernier a également la responsabilité de gérer le cycle de vie des **listeners** et des **filtres** Web.

LES LISTENERS WEB

Un listener (écouteur) est un terme couramment utilisé en Java pour désigner un objet qui sera notifié lors d'une modification de son environnement. Dans les patrons de conception (Design Patterns) Objet, on l'appelle plus généralement Observateur.

Pour le conteneur Web Java EE, un listener designe une classe qui implémente une des interfaces définies dans l'API servlet.

Le principe d'utilisation est le même pour tous les types de listeners Web. Si l'application souhaite être avertie d'un événement particulier survenant pour un `ServletContext` (c'est-à-dire pour l'ensemble de l'application Web), une requête ou une session HTTP alors elle doit fournir une implémentation d'un listener. Une méthode de ce dernier sera appelée par le conteneur à chaque fois que l'événement concerné surviendra lors de la vie de l'application.

Les différents types de listeners sont représentés dans l'API Servlet par les **interfaces** Java suivantes :

javax.servlet.ServletContextListener
Permet d'écouter les changements d'état du `ServletContext`. Le conteneur Web avertit l'application de la création du `ServletContext` grâce à la méthode `contextInitialized` et de la destruction du `ServletContext` grâce à la méthode `contextDestroyed`.

> Il est important de comprendre que le `ServletContext` représente l'application Web. Donc un `ServletContextListener` est un moyen de réaliser des traitements au moment du lancement de l'application Web et/ou au moment de son arrêt.

javax.servlet.ServletContextAttributeListener
Permet d'écouter les changements d'état des attributs stockés dans le `ServletContext` (les attributs de portée application). Le conteneur avertit l'application de l'ajout d'un attribut (appel à la méthode `attributeAdded`), de la suppression d'un attribut (appel à la méthode `attributeRemoved`) et de la modification d'un attribut (appel à la méthode `attributeReplaced`).

javax.servlet.ServletRequestListener
Permet d'écouter l'entrée et/ou la sortie d'une requête de l'environnement de l'application Web. Le conteneur avertit l'application de l'entrée d'une requête à traiter grâce à la méthode `requestInitialized` et de la sortie de la requête grâce à la méthode `requestDestroyed`.

javax.servlet.ServletRequestAttributeListener
Permet d'écouter les changements d'état des attributs stockés dans la `HttpServletRequest` (les attributs de portée requête). Le conteneur avertit l'application de l'ajout d'un attribut (appel à la méthode `attributeAdded`), de la suppression d'un attribut (appel à la méthode `attributeRemoved`) et de la modification d'un attribut (appel à la méthode `attributeReplaced`).

javax.servlet.http.HttpSessionListener

Permet d'écouter la création et la suppression d'une `HttpSession`. Le conteneur avertit l'application de la création d'une session grâce à la méthode `sessionCreated` et de la suppression d'une session grâce à la méthode `sessionDestroyed`. La suppression d'une session signifie que soit elle a été invalidée par l'application elle-même grâce à la méthode HttpSession.invalidate() soit elle est arrivée à expiration et le conteneur a décidé de l'invalider.

javax.servlet.http.HttpSessionAttributeListener

Permet d'écouter les changements d'état des attributs stockés dans la `HttpSession` (les attributs de portée session). Le conteneur avertit l'application de l'ajout d'un attribut (appel à la méthode `attributeAdded`), de la suppression d'un attribut (appel à la méthode `attributeRemoved`) et de la modification d'un attribut (appel à la méthode `attributeReplaced`).

> Il existe également deux autres listeners Web : Le HttpSessionActivationListener et le HttpSessionBindingListener. Ils ne sont pas décrits ici car ils sont réservés à des usages plus avancés de l'API Servlet.

DÉCLARATION DES LISTENERS WEB

Une classe implémentant une ou plusieurs interfaces la désignant comme un listener Web doit également être déclarée auprès du conteneur Web. Pour cela, il suffit d'ajouter l'annotation @WebListener à la classe :

Exemple de `ServletRequestListener`

```
package fr.epsi;

import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class MyServletRequestListener implements ServletRequestListener {

  @Override
  public void requestInitialized(ServletRequestEvent sre) {
    // ...
  }

  @Override
  public void requestDestroyed(ServletRequestEvent sre) {
    // ...
  }
}
```

Si l'**on ne souhaite pas utiliser une annotation**, il est également possible de déclarer un listener dans le fichier de déploiement `web.xml` grâce à la balise `listener`.

Déclaration d'un listener dans le fichier web.xml

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

  <listener>
        <listener-class>fr.epsi.MyServletRequestListener</listener-class>
  </listener>

</web-app>
```

EXEMPLE D'UTILISATION D'UN LISTENER

L'exemple (simple) ci-dessous consiste en un `ServletConfigListener` dont le rôle est de réaliser un log applicatif signalant respectivement le lancement et l'arrêt de l'application Web :

Exemple de `ServletRequestListener`

```
package fr.epsi;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class LoggingListener implements ServletContextListener {
  @Override
  public void contextInitialized(ServletContextEvent sce) {
    sce.getServletContext().log("## Lancement de l'application ##");
  }

  @Override
  public void contextDestroyed(ServletContextEvent sce) {
    sce.getServletContext().log("## Arrêt de l'application ##");
  }
}
```

LES FILTRES DE SERVLET

Il est parfois intéressant d'effectuer des opérations avant et/ou après l'invocation de la servlet. Il s'agit souvent d'opérations communes à un ensemble de requêtes d'une application Web.

Un filtre de Servlet est une classe implémentant l'interface Filter. Un filtre a son propre cycle de vie. Une fois créé, le conteneur initialise le filtre en appelant sa méthode `init` et il signalera la destruction du filtre en appelant sa méthode `destroy`. L'opération de filtrage est réalisée grâce à la méthode doFilter.

Exemple d'implémentation d'un filtre Web

```
package fr.epsi;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class MyFilter implements Filter {

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

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
    // ...
  }

  @Override
  public void destroy() {
    // ...
  }

}
```

DÉCLARATION DES FILTRES

La déclaration d'un filtre Web auprès du conteneur se fait soit par l'annotation **`@WebFilter`** soit dans le fichier de déploiement **web.xml**.

Comme pour une Servlet, un filtre est associé à un ou plusieurs motifs d'URL (URL pattern) indiquant au conteneur pour quelles requêtes HTTP le filtre doit être appelé.

Déclaration d'un filtre Web par annotation

```
package fr.epsi;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;

@WebFilter({"/subpart/*", "/otherpart/*"})
public class MyFilter implements Filter {

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

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
    // ...
  }

  @Override
  public void destroy() {
    // ...
  }

}
```

Si l'**on ne souhaite pas utiliser une annotation**, il est également possible de déclarer un listener dans le fichier de déploiement web.xml grâce aux balises `filter` et `filter-mapping`.

Déclaration d'un filtre dans le fichier web.xml

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

  <filter>
        <filter-name>MyFilter</filter-name>
        <filter-class>fr.epsi.MyFilter</filter-class>
  </filter>

  <filter-mapping>
        <filter-name>MyFilter</filter-name>
        <url-pattern>/subpart/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
        <filter-name>MyFilter</filter-name>
        <url-pattern>/otherpart/*</url-pattern>
  </filter-mapping>
</web-app>
```

Il est également possible de déclarer qu'un filtre doit être utilisé pour des requêtes traitées par des Servlets spécifiques plutôt que d'utiliser un modèle d'URL.

IMPLÉMENTATION D'UN FILTRE

L'opération de filtrage est réalisée par la méthode doFilter.

Principe général d'implémentation d'un filtre

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
  // réaliser des opérations avant le traitement de la requête

  // appeler l'élément suivant dans la chaîne de filtrage
  chain.doFilter(request, response);

  // réaliser des opérations après le traitement de la requête
}
```

Si plusieurs filtres doivent être déclenchés pour le traitement d'une requête, alors l'appel à `chain.doFilter(...)` permet de passer au filtre suivant. Un fois le dernier filtre appelé, l'appel à `chain.doFilter(...)` passera au traitement normal de la requête (Servlet, JSP ou ressource statique).

Il est recommandé d'implémenter des filtres de manière à ce qu'ils soient indépendants les uns des autres. En effet, si plusieurs filtres sont appelés pour le traitement d'une requête, l'ordre dans lequel ces filtres seront appelés n'est pas prédictible s'ils ont été déclarés avec l'annotation `@WebFilter`. En revanche, ils seront appelés dans l'ordre des balises `filter-mapping` s'ils ont été déclarés à partir du fichier de déploiement `web.xml`

Une implémentation de filtre peut très bien ne pas appeler `chain.doFilter(...)` et choisir de générer directement une réponse.

CAS D'UTILISATION DE FILTRES

Deux exemples d'implémentation de filtres simples mais efficaces.

GESTION DE L'UTF-8

Un cas facilement compréhensible est celui d'une application Web qui poste les données de tous ses formulaires HTML en UTF-8. Nous avons vu que par défaut, le conteneur Web utilise l'encodage ISO-8859-1 (Latin-1). Il est donc nécessaire de positionner le bon encodage grâce à la méthode `ServletRequest.setCharacterEncoding(String)`. Cette opération répétitive est source d'oubli (et donc de bug). Il serait préférable de garantir que cette méthode soit systématiquement appelée avant chaque traitement de Servlet. Ce type de comportement peut très facilement être implémenté au moyen d'un filtre Web.

Filtre UTF-8

```
package fr.epsi;
```

```
import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;

@WebFilter("/*")
public class Utf8RequestEncodingFilter implements Filter {

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
    request.setCharacterEncoding("UTF-8");
    chain.doFilter(request, response);
  }

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

  @Override
  public void destroy() {
  }

}
```

GÉNÉRATION DE LOG

Il peut être intéressant de garder une trace des paramètres HTTP reçus lors des tests ou pour des statistiques. Le filtre ci-dessous écrit dans les logs du serveur le nom et la valeur de tous les paramètres reçus :

Filtre de log de paramètres

```
package fr.epsi;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;

@WebFilter("/*")
public class LogFilter implements Filter {

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
    request.getServletContext().log("parameters received: " + parametersToString(request));
    chain.doFilter(request, response);
  }

  private List<String> parametersToString(ServletRequest request) {
    List<String> parameters = new ArrayList<>();
    request.getParameterMap().forEach((k, v) -> parameters.add(k + "=" + Arrays.toString(v)));
    return parameters;
  }

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

  @Override
  public void destroy() {
  }

}
```

L'utilisation conjointe des deux filtres ci-dessus peut poser problème. En effet, la méthode `request.setCharacterEncoding(...)` dans la classe `Utf8RequestEncodingFilter` doit être appelée avant que les paramètres de la requête ne soient accédés. Le filtre `Utf8RequestEncodingFilter` doit donc être placé **avant** le filtre `LogFilter`. Malheureusement, cela ne peut pas être garanti par l'utilisation de l'annotation `@WebFilter`.

On peut imaginer des traitements bien plus complexes grâce aux filtres : contrôle des droits d'accès (autorisation), optimisation d'image, chiffrement des données...

Cette œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution - Partage dans les Mêmes Conditions 3.0 France.