Plan

Application Web en Java

Intérêts de Java et Java EE

- solution **portable**
- solution **sécurisée**
- solution **standardisée**

Application Web: assemblage de composants, de contenu et de fichiers de configuration

Environnement d'exécution: *Web container*

Archive WAR (*Web Application Archive*)

- contient tous les fichiers de l'application Web
- contient un fichier de déploiement en XML (optionnel en JEE 6/7 suivant le contexte)

Application Web en Java (cont.)

Spécifications de Java EE: contrat entre le conteneur Web et les composants

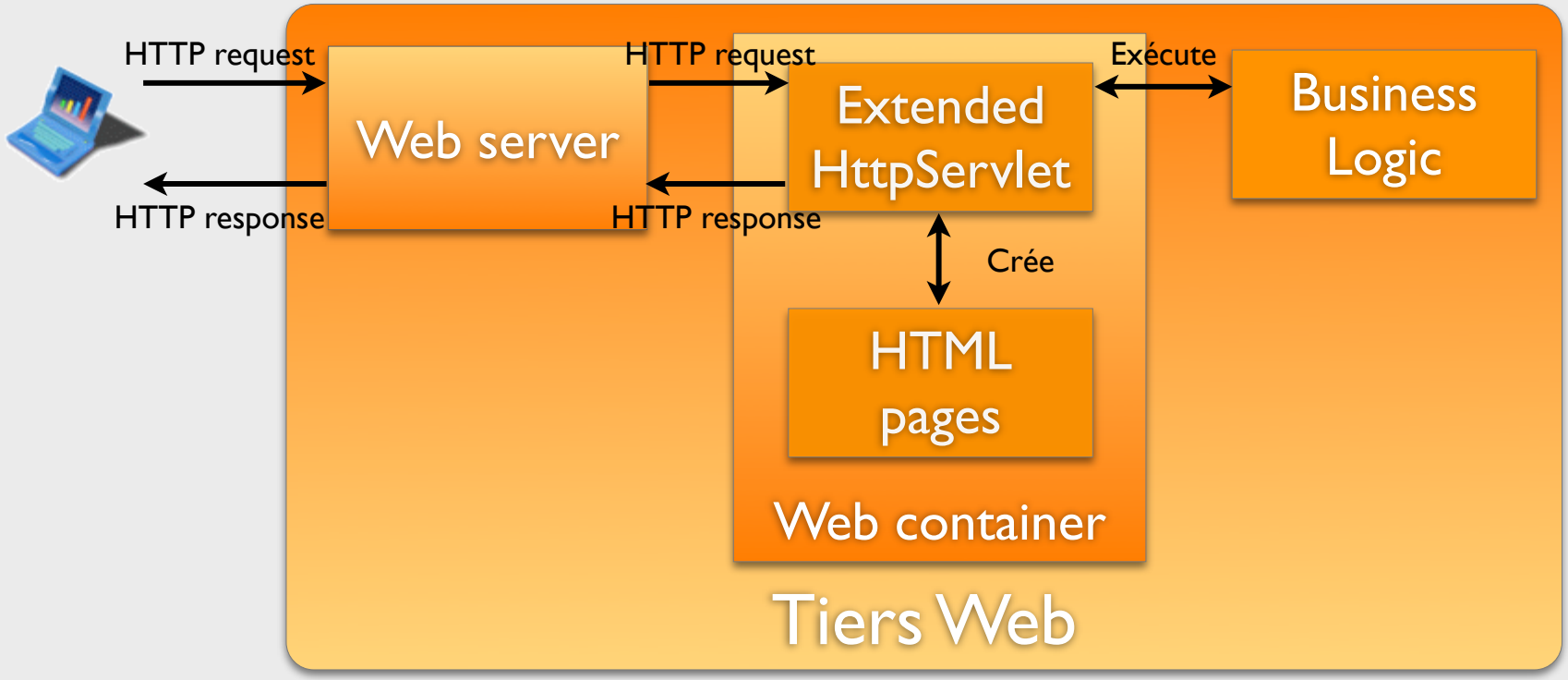
- pour définir le cycle de vie des composants
- pour définir le comportement des composants
- pour définir les services que le serveur offre aux composants

Deux types de composants Web dans la spécification Java EE

- les Servlets
- les JSP (*Java Server Pages*)

Plan

Servlet: en image



Servlet: introduction

Composant de l'architecture Java EE

Classe compilée qui s'exécute dans un conteneur de servlets (*Glassfish* par exemple) hébergé par le serveur

Permet d'étendre les fonctionnalités du serveur Web

Écrit en Java: indépendant de la plate-forme/du serveur

Semblable à d'autres technologies/frameworks

- CGI, RoR (Ruby), Grails (Groovy), Zend, Symfony (PHP), Play (Java)

Servlet: introduction (cont.)

Avantages

- efficacité
  - utilise des threads plutôt que des processus (CGI)
- pratique
  - librairie très développée (moins que PHP, mais plus cohérente)
- portable
  - déployable quelque soit le serveur (ou presque)
- sécurisé
  - fonctionne dans une machine virtuelle (plus facilement maîtrisable)
- peu cher
  - existence de nombreux serveurs gratuits

Servlet: à quoi cela sert ?

Router les requêtes

Exécuter un traitement

Servlet: fonctionnement

Association URL(s) / servlet

Lecture des données envoyées par le client

- données explicites (formulaire)
- données implicites (*Request Header*)

Exécution du servlet par le conteneur

Génération et envoi de la réponse au client

- données explicites (HTML/XML)
- données implicites (*Response Header*, *Status Code*)

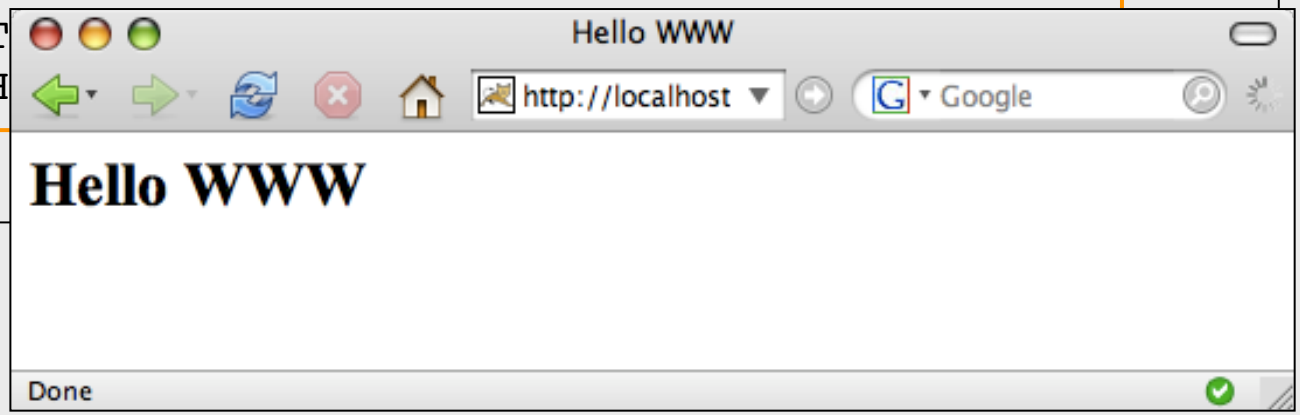
Un exemple

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">\n" +
                "<HTML>\n" +
                "<HEAD><T
                "<BODY><H
  }
}
```

API à importer

HelloWWW est un objet de type HttpServlet

Redéfinition de doGet pour traiter une requête GET



Servlet: l'API

2 packages importants

- *javax.servlet*
  - package générique indépendant des protocoles
- *javax.servlet.http*
  - spécifique à HTTP
  - permet la gestion des sessions

pour le protocole HTTP

Les classes

- 2 classes abstraites…
  - *javax.servlet.GenericServlet* (indépendante du protocole)
  - *javax.servlet.http.HttpServlet* (spécifique à HTTP)
- qui implantent 2 interfaces
  - *javax.servlet.Servlet*
  - *javax.servlet.ServletConfig*

La classe *GenericServlet*

javax.servlet.GenericServlet

- indépendant des protocoles réseaux
- classe abstraite (une implantation de base de l'interface Servlet)

Trois méthodes

- *init()*
- *service()* ← point d'entrée principal
  méthode abstraite qui doit être implantée par la sous-classe
- *destroy()*

Structure de base d'un servlet

```
import javax.servlet.*;

public class First extends GenericServlet {

   public void init(ServletConfig config)
      throws ServletException {
   ...
   }

   public void service(ServletRequest req, ServletResponse rep)
      throws ServletException, IOException {
   ...
   }

   public void destroy() {
   ...
   }
}
```

Communiquer avec le client

Deux arguments pour la méthode *service()*

- *javax.servlet.**ServletRequest***
  - encapsule la requête du client
  - contient le flux d'entrée et les données en provenance du client
- *javax.servlet.**ServletResponse***
  - contient le flux de sortie pour répondre au client

Développer des *HttpServlet*

- *javax.servlet.http.HttpServlet* *c'est la classe que nous utiliserons !*
- hérite de *GenericServlet* et fournit une implantation spécifique à HTTP de l'interface *Servlet*

HttpServlet: la méthode *service()*

HttpServlet surcharge la méthode *service()*

- lit la méthode HTTP (GET, POST,…) de la requête
- transmet la requête à la méthode appropriée pour traitement

Correspondance méthode HTTP/méthode "handler"

```
public void doXXX(HttpServletRequest req, HttpServletResponse rep)
```

Un servlet est une sous-classe de *HttpServlet* et on surcharge la ou les méthodes *doXXX()* nécessaires

- requête GET, HEAD: *doGet()*
- requête POST: *doPost()*
- requête PUT: *doPut()*

Structure de base d'un servlet Http

```
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {

   public void init(ServletConfig config)
      throws ServletException {
   ...
   }

   public void doGet(HttpServletRequest req, HttpServletResponse rep)
      throws ServletException, IOException {
   ...
   }

   public void destroy() {
   ...
   }
   public String getServletInfo() {…}
}
```

Structure de base d'un servlet Http (cont.)

Détail de la méthode *doGet()*

```
public void doGet(HttpServletRequest req,
                  HttpServletResponse rep)
  throws ServletException, IOException {

    // Lire la requête du client

    // La traiter

    // Répondre

}
```

Objet requête

Objet réponse

Lire la requête du client

HttpServletRequest hérite de *ServletRequest*

- encapsule la requête HTTP et fournit des méthodes pour accéder à toutes les informations
- contient les informations sur l'environnement du serveur

Méthodes de *HttpServletRequest*

- *String getMethod()*: retourne la méthode HTTP
- *String getHeader(String name)*: retourne le paramètre name de l'entête HTTP de la requête
- *String getRemoteHost()*: retourne le nom d'hôte du client
- *String getRemoteAddr()*: retourne l'adresse IP du client

Lire la requête du client (cont.)

Méthodes (suite)

- *String getParameter(String name)*: retourne la valeur d'un champ de formulaire
- *Enumeration getParameterNames()*: retourne tous les noms des paramètres
- *String getServerName()*: retourne le nom du serveur
- *String getServerPort()*: retourne le port sur lequel le serveur écoute

Répondre au serveur

HttpServletResponse hérite de *ServletResponse*

- utilisé pour construire un message de réponse HTTP renvoyé au client
- contient les méthodes nécessaires pour définir: le type de contenu, l'entête, et le code de retour
- contient un flux de sortie pour envoyer des données (HTML ou XML ou autre) au client

Méthodes de *HttpServletResponse*

- *void setStatus(int statusCode)*: définit le code de retour de la réponse
- *void setHeader(String name, String value)*: définit la valeur de l'entête name

Répondre au serveur (cont.)

Méthodes (suite)

- *void setContentType(String type)*: définit le type MIME du contenu
- *PrintWriter getWriter()*: pour envoyer des données texte au client
- *ServletOutputStream getoutputStream()*: flux pour envoyer des données binaires
- *void sendRedirect(String url)*: redirige le client vers l'URL

On peut

- renvoyer du HTML
- rediriger vers une page donnée
- faire une action (upload/download,…)

Cycle de vie d'un servlet

① Chargement

spécifié lors du déploiement

② Initialisation

`init()`

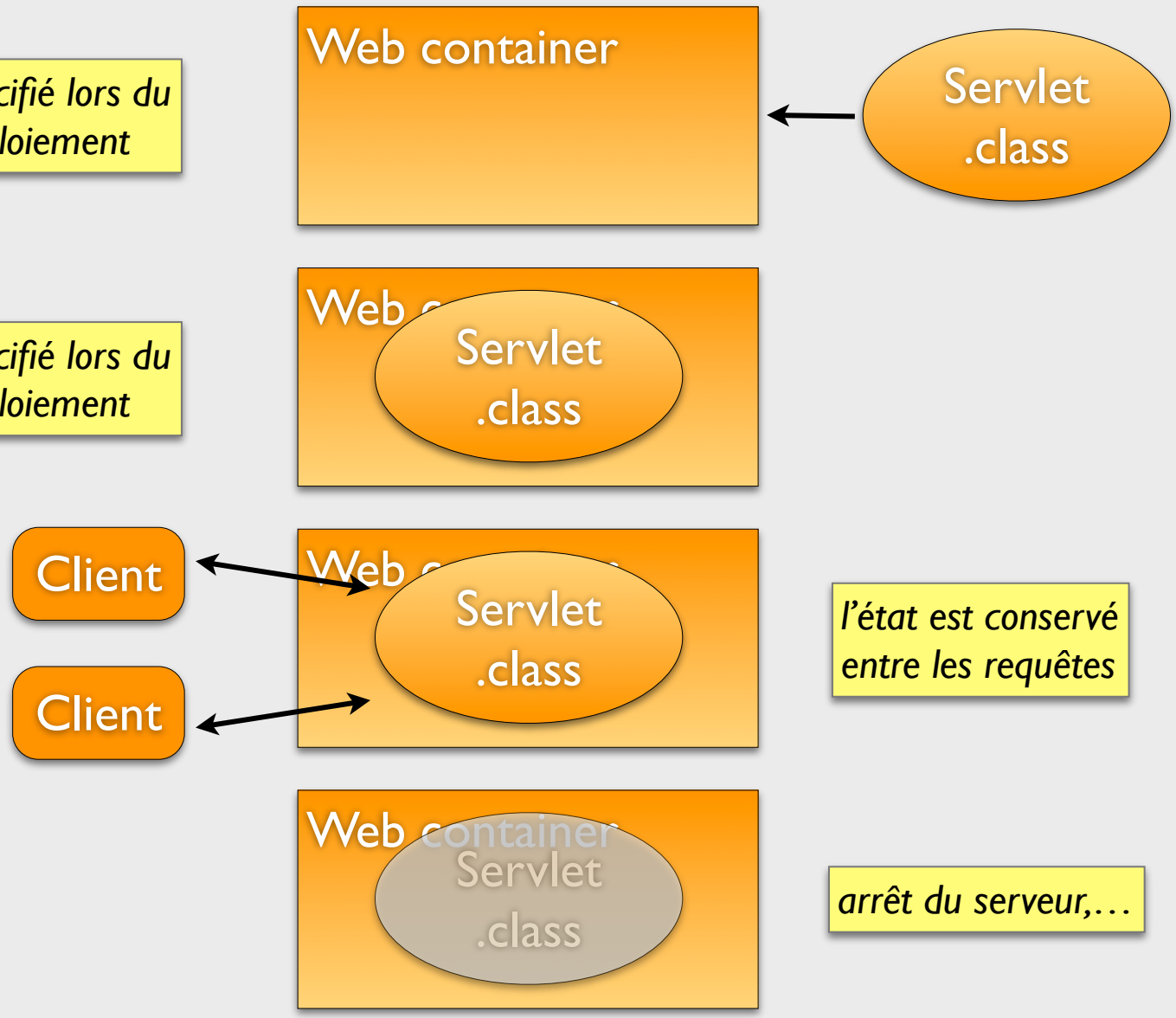
spécifié lors du déploiement

③ Traitement de 1 à n requêtes

`service()`

④ Destruction

`destroy()`



Cycle de vie d'un servlet: initialisation

init()

- une fois crée, le servlet est initialisé avec cette méthode
- elle n'est appelée qu'une seule fois par le serveur
- similaire à un constructeur
- on peut y placer
  - les connexions réseaux
  - les connexions BD
  - la récupération des paramètres d'initialisation
  - la récupération des paramètres de contexte de l'application

Cycle de vie d'un servlet: initialisation (cont.)

WEB-INF/web.xml

```
<servlet>
<servlet-name>MonServlet</servlet-name>
<servlet-class>monapp.servlets.MonServlet</servlet-class>
<init-param>
    <param-name>langue</param-name>
    <param-value>fr</param-value>
</init-param>
<init-param>
    <param-name>devise</param-name>
    <param-value>EUR</param-value>
</init-param>
</servlet>
```

Code

```
public class MonServlet extends GenericServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        out.println("Servlet init parameters:");
        Enumeration e = getInitParameterNames();
        while (e.hasMoreElements()) {
            String key = (String)e.nextElement();
            String value=getInitParameter(key);
            out.println("  " + key + " = " + value);
        }
        …
}
```

Cycle de vie d'un servlet: initialisation (cont.)

En JEE 7:

```
package monapp.servlets.MonServlet;

@WebServlet(name="MonServlet",
   urlPatterns={...},
   initParams={
      @WebInitParam(name="langue", value="fr"),
      @WebInitParam(name="devise", value="EUR")
   }
)
public class MonServlet extends GenericServlet {
   public void init(ServletConfig config) throws ServletException {
      super.init(config);
      out.println("Servlet init parameters:");
      Enumeration e = getInitParameterNames();
      while (e.hasMoreElements()) {
         String key = (String)e.nextElement();
         String value=getInitParameter(key);
         out.println("  " + key + " = " + value);
      }
      …
}
```

Cycle de vie d'un servlet: initialisation (cont.)

WEB-INF/web.xml

```
<context-param>
   <param-name>langue</param-name>
   <param-value>fr</param-value>
   <description>Paramètre de contexte</description>
</context-param>
```

Code

```
public class MonServlet extends GenericServlet {
   public void init(ServletConfig config) throws ServletException {
      super.init(config);
      out.println("Context init parameters:");
      ServletContext sc = getServletContext();
      Enumeration e = sc.getInitParameterNames();
      while (e.hasMoreElements()) {
         String key = (String)e.nextElement();
         Object value = sc.getInitParameter(key);
         out.println("  " + key + " = " + value);
      }
      …
}
```

Cycle de vie d'un servlet: traitement

service()

- appelée automatiquement par le serveur à chaque requête
- dispatch en fonction du type de requête (GET/POST)

Utilisation des méthodes de *HttpServletRequest*

Exemple

- http://localhost/exemples/servlets/TraceServlet/tutu?toto=titi
  - *getMethod()* → GET
  - *getPathInfo()* → /tutu
  - *getQueryString()* → toto=titi
  - *getRequestURI()* → /exemples/servlets/TraceServlet/tutu
  - *getServletPath()* → /servlets/TraceServlet
  - *getContextPath()* → /exemples

Cycle de vie d'un servlet: destruction

destroy()

- destruction du servlet
- appelée qu'une seule fois quand le servlet est déchargé par le serveur, ou quand le serveur est arrêté
- on y place la fermeture des connexions réseaux, BD

Cycle de vie d'un servlet: concurrence

Un servlet gère les requêtes de plusieurs clients: un thread est généré à chaque requête

Conséquence: problème de concurrence d'accès

- les attributs du servlet sont partagés
- les méthodes accèdent à des ressources partagées

Pour les portions critiques

- synchroniser les accès à la ressource (*synchronized*)
- implanter l'interface *SingleThreadModel*
  - le moteur de servlet crée un pool d'objets servlets, et le servlet ne répond aux requêtes que d'un seul client à la fois

```
public class ExempleServlet extends HttpServlet
    implements SingleThreadModel {
…
}
```

Traitement de données

Un formulaire HTML

- permet de saisir les données
- mécanisme pour envoyer les données au serveur Web

Rappel: les balises

- ⟨FORM⟩: pour définir un formulaire
- ⟨INPUT⟩: pour définir un élément d'interface
  - text,
  - password,
  - checkbox,
  - radio,
  - submit, reset
- ⟨TEXTAREA⟩: zone de texte sur plusieurs lignes
- ⟨SELECT⟩: liste déroulante

Traitement des données (cont.)

Attributs de <FORM>

- action: URL où envoyer les données
- method: méthode à utiliser pour l'envoi (GET/POST)

Les données sont envoyées lorsque l'on clique sur un bouton de type submit

```
<FORM METHOD="POST"
   ACTION="http://host/servlet/demandeServlet">

Nom: <INPUT type="text" name="Nom"/>
<BR>
<INPUT type="submit" valeur="Envoyer"/>

</FORM>
```

Traitement des données (cont.)

Les méthodes *GET* et *POST*

- *GET*
  - les champs du formulaire sont ajoutés à l'URL de l'attribut ACTION
  - forme nom=valeur

```
URL?nom1=val1&nom2=val2&…
```

- *POST*
  - les données sont passées dans le corps du message HTTP
  - envoi de données binaires par exemple

Traitement des données (cont.)

Lecture des données

- méthode *String getParameter(String name)*
- dans le formulaire

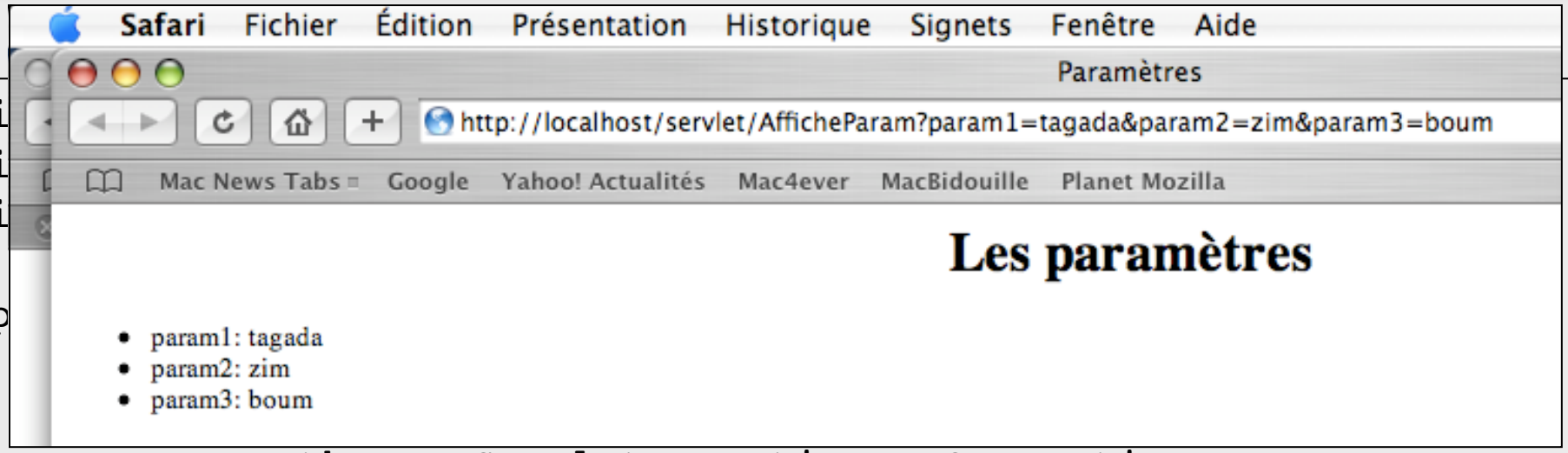
```
<INPUT type="text" name="nom" />
```

- dans la méthode *doXXX()*

```
String n = req.getParameter("nom");
```

Exemple de lecture des paramètres

Traitement des données (cont.)




```
            throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HEAD>\n <TITLE>Paramètres</TITLE>\n </HEAD>\n"
      +"<BODY>\n" + "<H1 ALIGN=CENTER>Les paramètres</H1>\n"
      + "<UL>\n "
      + "<LI>param1: "+ request.getParameter("param1") + "\n"
      + "<LI>param2: "+ request.getParameter("param2") + "\n"
      + "<LI>param3: "+ request.getParameter("param3") + "\n"
      + "</UL>\n" + "</BODY>");
  }
```

Traitement des données issues d'un formulaire

```
<!DOCTYPE HT
<HTML><HEAD>
<BODY>
<H1 ALIGN="C
<FORM ACTION
 Référence:
 Description
 Prix: <INPU
 <HR>
 Nom: <INPUT
 Prénom: <IN
 Adresse: <T
 Carte banca
   
   
VALUE="Maste
   
Express<BR>
  Numéro: <I
  Confirmati
  <CENTER><I
</FORM>
</BODY></HTM
```

Formulaire avec la méthode POST

http://localhost:8080

Google

Formulaire à remplir

Référence: ID3
Description: Polaire
Prix: 34

Nom: Perrin
Prénom: Olivier
Adresse: 21 rue Machin
54000 Nancy
Carte bancaire:
- Visa
- MasterCard
- American Express

Numéro: *****
Confirmation du numéro: *****

Valider

Done

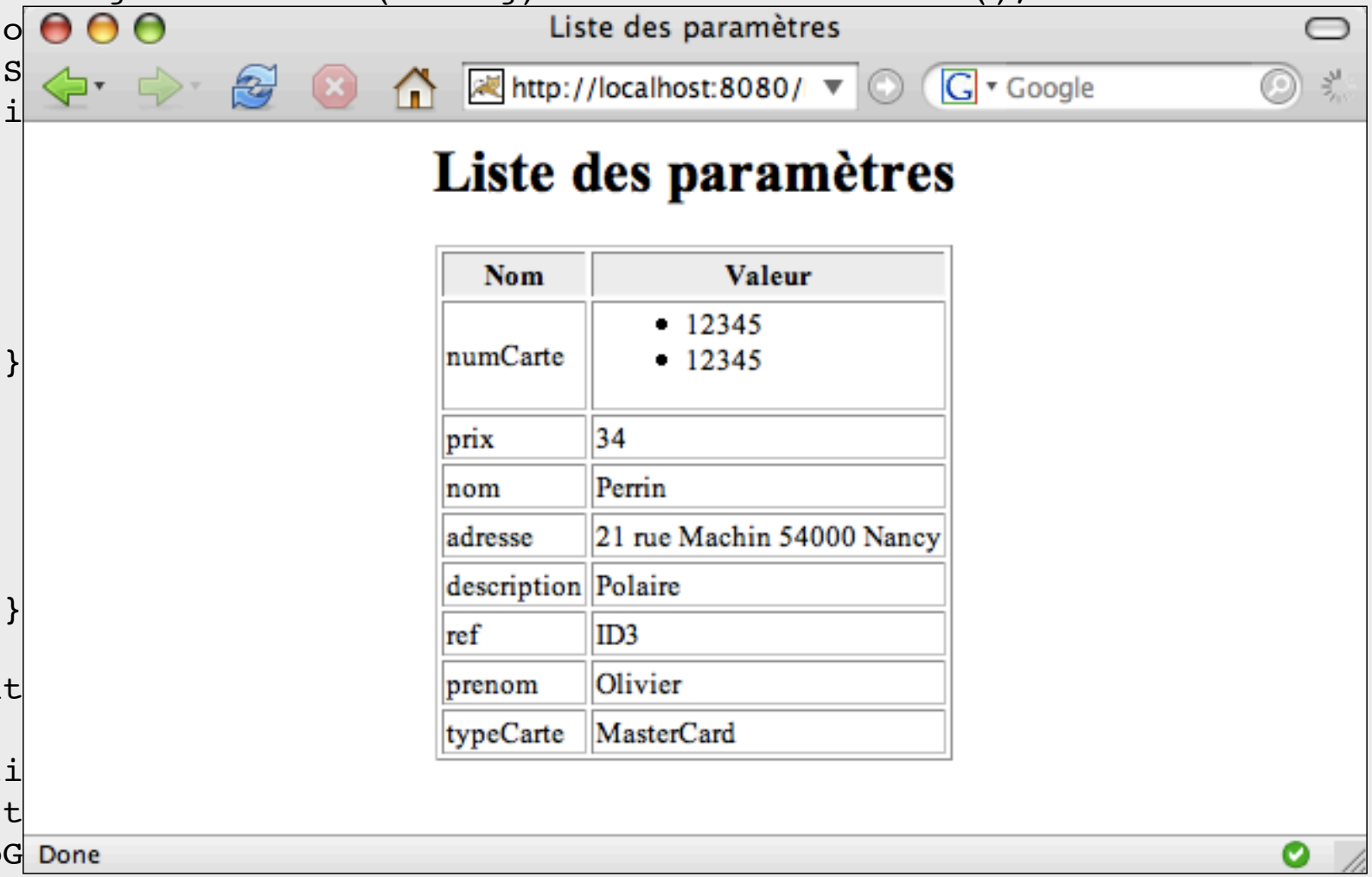
```
R>

an

<BR>
```

Traitement des données issues d'un formulaire

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class AfficheFormulaire extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse rep)
      throws ServletException, IOException {
    rep.setContentType("text/html");
    PrintWriter out = rep.getWriter();
    String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n";
    String titre = "Liste des paramètres";
    out.println(docType + "<HTML>\n" +
                "<HEAD><TITLE>" + titre + "</TITLE></HEAD>\n" +
                "<BODY>\n" + "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#eeeeee\">\n" +
                "<TH>Nom<TH>Valeur");
    Enumeration nomParams = req.getParameterNames();
```

Traitement des données issues d'un formulaire

```
    while(nomParams.hasMoreElements()) {
      String nomParam = (String)nomParams.nextElement();
      o
      S
      i
      }
      }
    }
    out
  }
  publi
      t
    doG
  }
}
```



Une application Web en Java EE

Une application Web contient

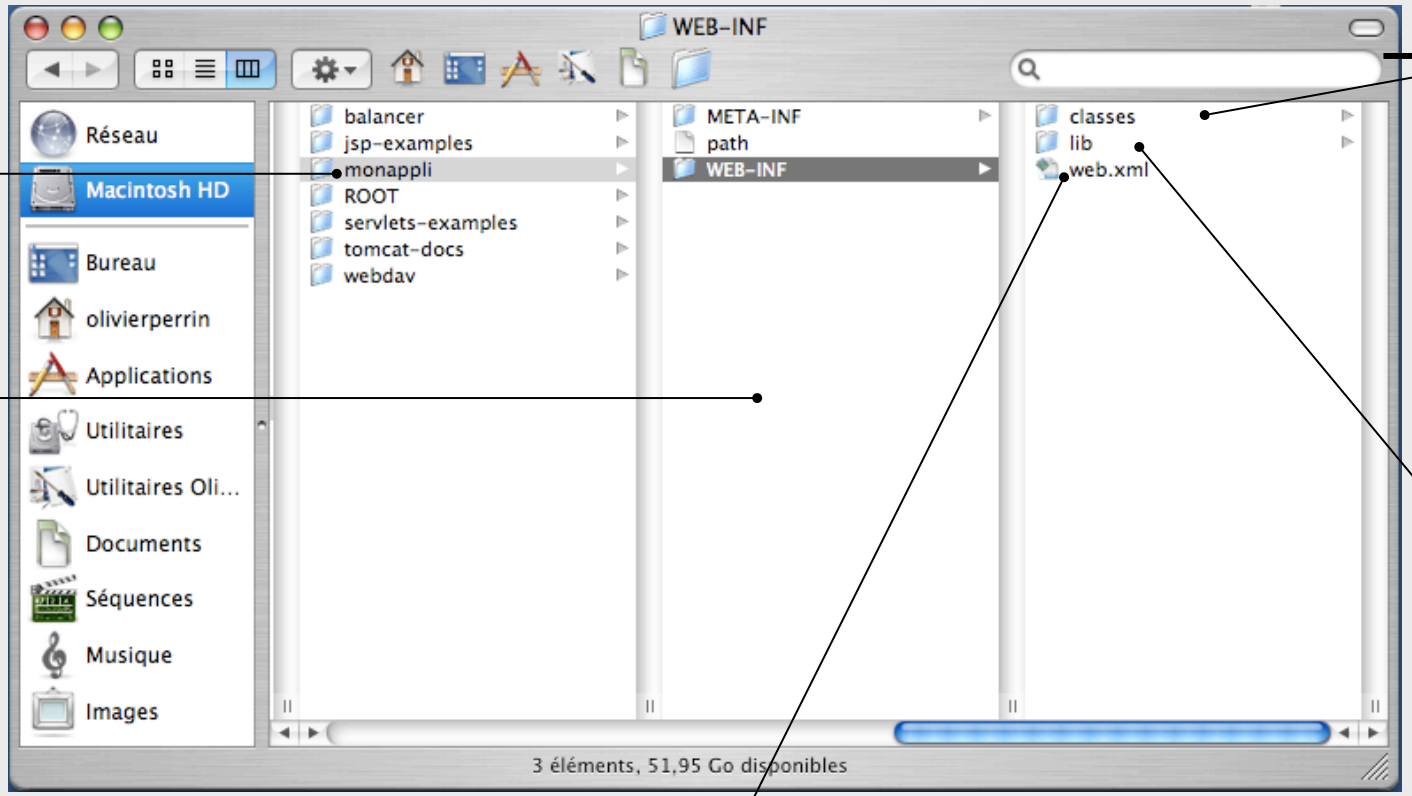
- l'ensemble des classes
- l'ensemble des librairies nécessaires (parser XML,…)
- les pages HTML
- les pages JSP
- le fichier de déploiement (optionnel en JEE 6/7)

Elle est stockée dans un fichier d'archive WAR

Elle est déployée simplement sur le serveur

Une application Web en Java EE: structure

L'application est située dans un répertoire qui possède une structure donnée



Classes des servlets

Répertoire de l'application

Fichiers de présentation (html, jsp,…)

Librairies (jar,…)

Fichier de configuration (optionnel en JEE 6/7)

Une application Web en Java EE: *web.xml*

Le fichier *web.xml* donne les instructions concernant le déploiement du servlet dans le conteneur

web-app

- description de l'application Web

servlet

- relation entre le nom du servlet et la classe qui l'implante

servlet-mapping

- relation entre le nom du servlet et l'URL qui permet d'y accéder
- possibilité d'avoir plusieurs chemins différents

Optionnel en JEE 6/7 !

Une application Web en Java EE: *web.xml* (2)

La description d'une application Web se trouve entre <*web-app*> et </*web-app*>

Configuration du servlet grâce à l'élément <*servlet*>

- <*icon*>*test.gif*</*icon*>: optionnel
- <*servlet-name*>*Test*</*servlet-name*>: utilisé comme référence pour la définition du servlet
- <*display-name*>*Servlet de test*</*display-name*>: optionnel
- <*description*>*Blabla*</*description*>: optionnel
- <*servlet-class*>*servlet.TestServlet*</*servlet-class*>: classe d'implantation
- <*jsp-file*>*JSP/test.jsp*</*jsp-file*>: fichier JSP associé
- <*init-param*>: paramètre d'initialisation
- <*security-role-ref*>: autorisations sur le servlet

Une application Web en Java EE: *web.xml* (3)

Un servlet peut avoir des paramètres d'initialisation

Ils peuvent être modifiés sans avoir à recompiler l'application

Utilisation avec la méthode *javax.servlet.ServletConfig.getInitParameter()*

Exemple

```
<init-param>
   <param-name>valeurTest</param-name>
   <param-value>12</param-name>
   <description>une valeur pour mon servlet</description>
</init-param>
```

Une application Web en Java EE: *web.xml* (4)

Le servlet mapping permet de construire la relation entre un servlet et son URL d'accès

```
<servlet-mapping>
   <servlet-name>Test</servlet-name>
   <url-pattern>/Test/*</url-pattern>
</servlet-mapping>
```

Tous les URLs correspondant à *http://host/webapp/url-pattern* exécuteront le servlet

Exemples

- http://host/webapp/Test/*.do
- http://host/webapp/Test
- http://host/webapp/Test/cours/test/*

Une application Web en Java EE: *web.xml* (5)

Paramètres de sécurité

Permet de lier un nom de rôle au nom de rôle utilisé en dur dans le servlet pour la sécurité

Cela permet d'éviter la modification du code du servlet pour l'adapter à l'environnement

```
<security-role-ref>
   <role-name>monadmin</role-name>
   <role-link>admin</role-link>
</security-role-ref>
```

Une application Web en Java EE: *web.xml* (6)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">
   <display-name>Mon Hello World Servlet</display-name>
   <servlet>
       <servlet-name>HelloWorld</servlet-name>
       <servlet-class>HelloWorldServlet</servlet-class>
   </servlet>
   <servlet-mapping>
       <servlet-name>HelloWorld</servlet-name>
       <url-pattern>/hello</url-pattern>
   </servlet-mapping>
</web-app>
```

Une application Web en Java EE: web.xml (7)

Annotations disponibles en JEE 6/7:

- *@WebServlet*: pour définir un servlet
- *@WebFilter*: pour définir un filtre
- *@WebListener*: pour définir un Listener
- *@WebInitParam*: pour définir un paramètre à l'initialisation
- *@ServletSecurity*: pour configurer la sécurité
- *@MultipartConfig*: chargement de fichiers

Possibilité de surcharger les valeurs dans le *web.xml*

Annotation *@WebServlet*

@WebServlet: définit un servlet

Doit hériter de *HttpServlet*

Attributs:

- name
- value
- urlPatterns: **obligatoire** !
- loadOnStartup
- initParams
- asyncSupported
- smallIcon, largeIcon
- description
- displayName

Annotation *@WebFilter*

@WebFilter: définit un filtre

Possibilité d'enregistrer dynamiquement lors de l'initialisation du *ServletContext* un servlet ou un filtre

```
ServletRegistration.Dynamic dynamic =
            servletContext.addServlet(...);
dynamic.addMapping(...);
```

Attributs:

- urlPatterns
- initParams
- filterName
- servletNames
- urlPatterns
- dispatcherTypes

```
@WebFilter("/path")
public class MonFiltre implements Filter {
  public void doFilter(
       HttpServletRequest req, HttpServletResponse res) {
    ...
    }
}
```

Annotation *@WebListener*

@WebListener: définit un listener

```
@WebListener
public class MonListener implements ServletContextListener {
   public void contextInitialized(ServletContextEvent sce) {
      ...
   }
}
```

Attribut:

- value: description du listener

Annotation *@ServletSecurity*

@ServletSecurity: définit des contraintes de sécurité

Support pour l'authentification, login et logout

Attention: attaché à une classe !

```
@WebServlet(name="MonServlet", urlPatterns={"/path1", "/path2"})
@ServletSecurity(@HttpConstraint(rolesAllowed={"user", "guest"}),
    httpMethodConstraints=@HttpMethodConstraint("POST", rolesAllowed={"admin"}))

public class MonServlet extends HttpServlet {...
```

Attribut:

- httpMethodConstraint
- value

Annotation *@MultipartConfig*

@MultipartConfig: support natif du chargement de fichiers

Gestion native des requêtes de type "mime/multipart"

- upload de fichiers depuis des formulaires

En utilisant *@MultipartConfig* sur la classe servlet, on dispose d'une API similaire à celle de *Commons FileUpload*

```
Collection<Part> parts = request.getParts()
for (Part part : parts) {
    part.getName();
    part.getSize();
    part.getInputStream();
}
```

Extensibilité

Modularité de *web.xml*

- *web-fragment.xml*

Possibilité d'intégrer des bibliothèques

- fichiers jar dans le répertoire *META-INF* de *WEB-INF/lib*
- le serveur recherche et fusionne les fragments

Intéressant pour servlets, filtres, listeners

```
<web-fragment>
    <servlet>
        <servlet-name>welcome</servlet-name>
        <servlet-class>WelcomeClass</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>welcome</servlet-name>
        <url-pattern>/Welcome</url-pattern>
    </servlet-mapping>
    ...
</web-fragment>
```

```
<web-fragment>
    <filter>
        <filter-name>filtre.helloworld</filter-name>
        <filter-class>lpro.MonFiltre</filter-class>
        <init-param>...</init-param>
    </filter>
    <filter-mapping>
        <filter-name>filtre.helloworld</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-fragment>
```

Extensibilité (cont.)

Problème de l'inter-dépendance et des conflits

- possibilité de fixer un ordre de chargement

web.xml peut déclarer un ordre d'intégration des fragments découverts

- *<absolute-ordering>*

Les fragments peuvent aussi déclarer leurs contraintes

- ordre relatif: *<ordering>*
- *<before>* et *<after>*
- ignoré si *<absolute-ordering>* est spécifié
- *<others/>* pour les fragments découverts mais non répertoriés

Extensibilité (cont.)

Synthèse

	web.xml	web-fragment.xml
Location	WEB-INF of the war file	META-INF directory of JAR file inside WAR file's WEB-INF/lib
Ordering related element	<absolute-ordering>	<ordering>

Ordre absolu dans *web.xml*

```
<web-app>
    ...
    <absolute-ordering>
        <name>A</name>
         <others/>
        <name>B</name>
    <absolute-ordering>
</web-app>
```

Ordre dans *web-fragment.xml*

```
web-fragment>
    <name>A</name>
    ...
    <ordering>
        <before>
            <others/>
        </before>
    </ordering>
</web-fragment>
```

```
<web-fragment>
    <name>B</name>
    ...
    <ordering>
        <after>
            <name>A</name>
        </after>
        <before>
            <others/>
        </before>
    </ordering>
</web-fragment>
```

Extensibilité (cont.)

Nouvelle API

Ajout de composants/accès à la liste des composants

- *addServlet(String servletName)*
- *addServlet(String className)*
- *addServlet(Class<?> servletClass)*
- *addServlet(Servlet servlet)*
- *setLoadOnStartup(boolean loadOnStartup)*
- *setServletSecurity, setMultipartConfig...*

Le code utilisant l'API doit être exécuté **avant** que l'application ne soit chargée

Async

Problème du cyle requête/réponse classique

- le thread de communication avec le client est maintenu ouvert tout le temps de traitement de la requête
- peut être très long !

Solution proposée: servlet à traitement asynchrone

Déclaration

- annotation: *@WebServlet(asyncSupported=true)*

Attention !

- il faut gérer le timeout soi-même
- *setTimeout()*

Async (cont.)

Utilisation

- la méthode *startAsync()* renvoie un *AsyncContext*
  - *AsyncContext ctx = ServletRequest.startAsync(req, res)*
- *AsyncContext* représente la requête détachée de la couche transport
- on peut utiliser alors
  - *dispatch(String path)*
  - *start(Runnable action)*
- il faut clore explicitement la requête
  - *AsyncContext.complete()*

Contrainte majeure: il faut que toute la chaîne (filtres, listeners) soit asynchrone !

Async API

ServletRequest.isAsyncSupported()

- vrai si TOUT [filtres | servlets] supportent le mode asynchrone dans
  - la chaîne des filtres
  - la chaîne de dispath de la requête
- configuré via:
  - web.xml ou annotation ou API

AsyncContext ServletRequest.startAsync

- appel par le servlet | filtre
- réponse non automatique à la fin de la méthode ou du filtre

Async API (cont.)

AsyncContext.dispatch()

- appelé par un gestionnaire asynchrone
- réponse générée par [Servlet | Filtre] en utilisant
  - le pool de thread du conteneur
  - JSP, JSF ou le framework utilisé

AsyncContext.complete()

- appelé par un gestionnaire asynchrone ou un conteneur
- génération de la réponse

Async: exemple

```
@WebServlet(name="AsyncServlet", urlPatterns={"/AsyncServlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
      try {
        System.out.println("Exécution de doGet");
        AsyncContext ac = request.startAsync();
        ac.addListener(new AsyncListener() {
          public void onComplete(AsyncEvent event) throws IOException {System.out.println("onComplete");}
          public void onTimeout(AsyncEvent event) throws IOException {System.out.println("onTimeout");}
          public void onError(AsyncEvent event) throws IOException {System.out.println("onError");}
          public void onStartAsync(AsyncEvent event) throws IOException {System.out.println("onStartAsync");}
        });

        System.out.println("Je suis dans doGet ...");

        // Start another service
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(10);
        executor.execute(new MyAsyncService(ac));

        System.out.println("Encore des trucs à faire dans doGet ...");
      } finally {
    }
  }
  class MyAsyncService implements Runnable {
    AsyncContext ac;

    public MyAsyncService(AsyncContext ac) {
      this.ac = ac;
      System.out.println("Dispatché vers " + "\"MyAsyncService\"");
    }
    public void run() {
      System.out.println("Traitement très long dans \"MyAsyncService\"");
      ac.complete();
    }
  }
}
```

```
Exécution de doGet
Dispatché vers "MyAsyncService"
Je suis dans doGet doGet ...
Encore des trucs à faire dans doGet ...
Traitement très long dans "MyAsyncService"
onComplete
```