GLG 203
Les Servlets

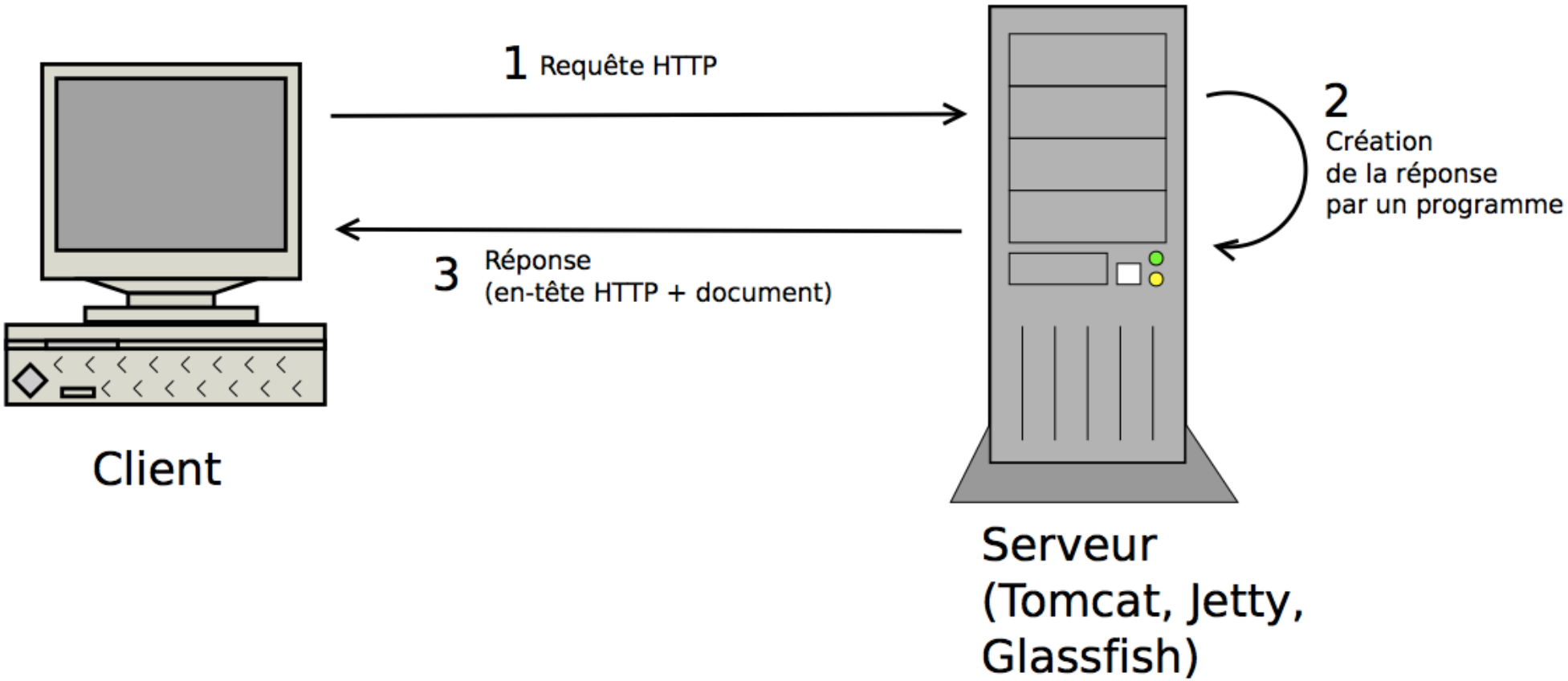
S. Rosmorduc

Sources

- Sources des exemples (en cours) :

  https://gitlab.cnam.fr/gitlab/glg203_204_demos/04_spring_web1.git

Architecture Web



Principes des sites web dynamiques

- le serveur reçoit une requête HTTP
- il fait tourner un programme, dont la sortie standard sera renvoyée au client
- la sortie standard est envoyée comme réponse au client.

Rappel sur les formulaires HTML

```
<form action="inscrire" method="POST">
 mail <input type="text" name="email"/> <br/>
 mot de passe <input type="password" name="pwd"/> <br/>
 signature <input type="text" name="signature"/> <br/>
 <input type="submit"/>
</form>
```

quand l'utilisateur presse le bouton « submit », le navigateur visite la page « inscrire » en envoyant les valeurs des champs email, pwd et signature

Le protocole HTTP

- protocole *sans état* : pas de mémoire ou de notion de session dans http même.
- du coup : passage à l'échelle simple
- mais sur des sites complexes on veut des états… (paniers de provisions, etc…)

Exemple de conversation HTTP

```
$ telnet deptinfo.cnam.fr 80
Trying 163.173.228.28...
Connected to deptinfo.cnam.fr.
Escape character is '^]'.
GET /~rosmorse/aisl-chine/ HTTP/1.0

HTTP/1.1 200 OK
Content-Length: 1194
Content-Type: text/html

<!doctype html>
<html>
<head>
</head>
<body>
  <ul>
  <li> <a href="patterns.pdf">Slides on design patterns</a></li>
  <li> <a href="servlets">JSP/Servlets et architecture</a></li>
  </ul>
</body>
</html>
```

Requête (terminée par une ligne vide)

En-tête

Ligne vide

Contenu

Réponse du serveur

Notion de paramètres

- données expédiée avec la requête
  - typiquement, couple attribut/valeur correspondant à un champ de formulaire
  - un paramètre a un nom (name), qui correspond au name indiqué dans le formulaire
  - la valeur du paramètre est une **chaîne de caractère dans tous les cas**.

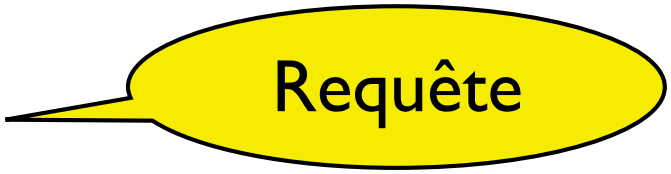
GET

- En mode GET, les valeurs des paramètres sont passés dans l'URL
- On peut enregistrer l'URL pour rejouer la requête
- typiquement utilisé pour récupérer (GET) de l'information depuis le site WEB

```
GET /hello.php?prenom=Alfred HTTP/1.0

HTTP/1.0 200 OK
Date: Tue, 08 Nov 2016 12:59:32 +0000
Connection: close
X-Powered-By: PHP/7.1.0RC3
Content-type: text/html; charset=UTF-8

<!doctype html>
<html>
    <body>
        Bonjour Alfred    </body>
</html>
```



POST

- paramètres passés dans le corps de la requête
- non rejouable
- typiquement, pour envoyer une information à enregistrer sur le site.

```
POST /hello.php HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

prenom=Alfred
HTTP/1.0 200 OK
Date: Tue, 08 Nov 2016 13:01:23 +0000
Connection: close
X-Powered-By: PHP/7.1.0RC3
Content-type: text/html; charset=UTF-8

<!doctype html>
<html>
    <body>
        Bonjour Alfred    </body>
</html>
```

Représentation des paramètres

- en mode GET, introduits par « ? » à la fin de l'URL
- donnés sous la forme nom=valeur
- séparés par « & » quand il y en a plusieurs
- codés (espace remplacé par « + » ou « %20 »
- pour le faire directement :
  - java.net.URLEncoder.encode

Utilisation de GET et de POST

- GET : recherche sur un site; visualisation d'un enregistrement dont on connaît l'identifiant, visite d'une page « statique » ; limitation de taille (pas définie par le standard)
- POST : envoi des données pour un nouvel enregistrement, ou pour une mise à jour, demande d'action (par exemple suppression d'un compte)…

Content-type

- Type MIME *(Multipurpose Internet Mail Extensions)*
- précise le type de contenu **et** son format exact (text/plain, text/html, image/png, application/pdf)
- précise aussi, pour le texte, le codage de celui-ci :

```
Content-Type: text/plain; charset=utf-8
```

Côté client

- formulaire html
- javascript

Technologies côté serveur

- scripts CGI
- API du serveur Web (ISAPI, NSAPI)
- PHP
- JSP/ASP
- Servlets
- Play!

Pile J2EE

- Modulaire, beaucoup de variantes possibles :
- technologies de visualisation : Servlets/JSP ; Wicket, JSF, thymeleaf ;
- + annuaire d'objets (JNDI) ;
- + gestion de transaction (JTA) ; persistence (JTA) ; + load balancing...
- serveurs simples : Tomcat, Jetty...
- serveurs full-stack : JBoss, Glassfish...
- EJB3 vs. Spring : convergence des technologies

Tomcat

- Serveur applicatif le plus utilisé
  - beaucoup de support

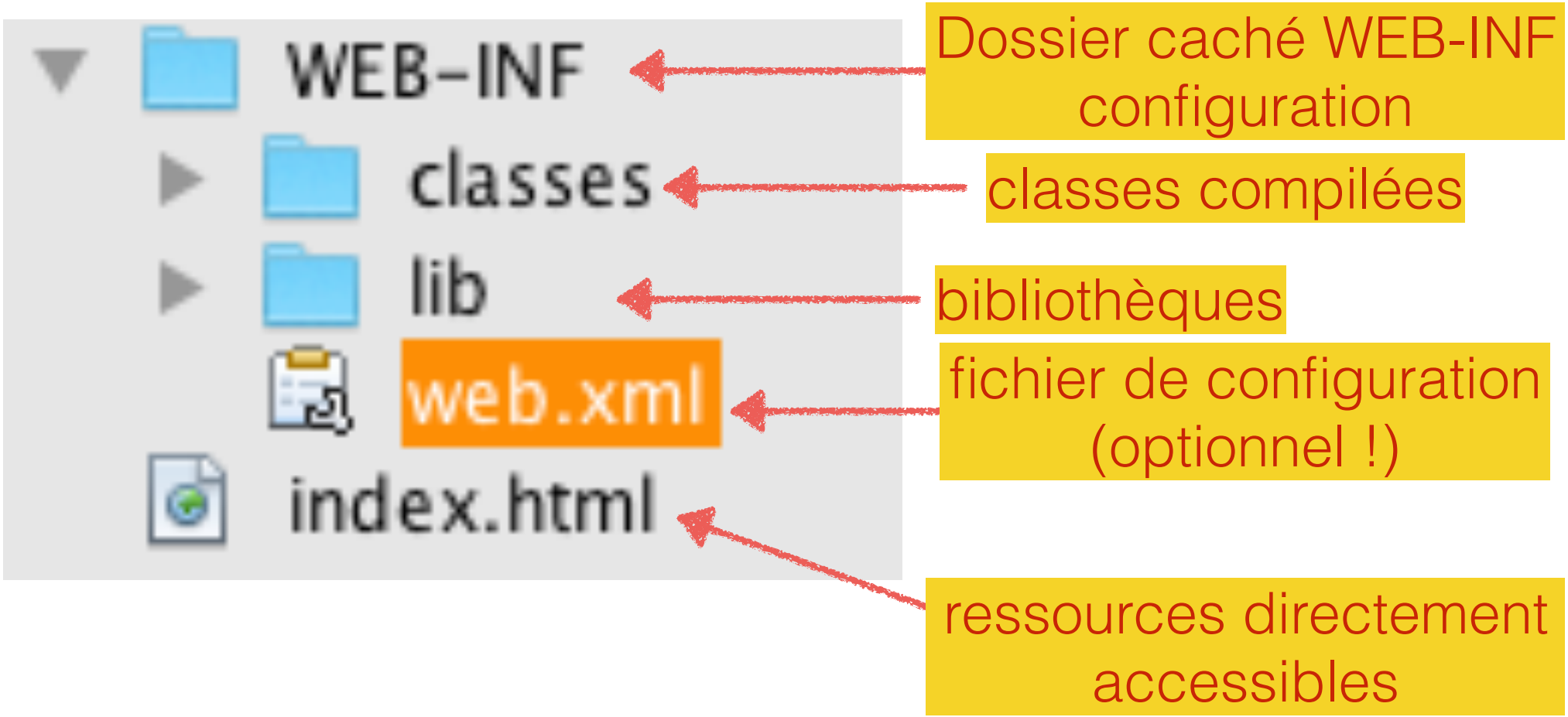
Jetty

- Serveur léger (configuration très simple à modifier/mettre en place)
- Utilisable en stand-alone, mais facile à intégrer *dans* une application java
- très utilisé pour les tests (démarre rapidement)
- documentation parfois un peu succincte.

Notion d'application Web

- Sur un serveur applicatif, les projets J2E sont déployés sous la forme *d'application Web*.
- Une application web regroupe des sources java, et sert d'unité, par exemple pour les sessions utilisateurs
- les applications sur le même serveur sont indépendantes les unes des autres.

Organisation des fichiers dans une application Web



Organisation des fichiers dans une application Web

- Généralement compressée (zippé) dans un fichier .war
- la racine de l'application contient
  - les ressources directement accessibles (fichiers HTML, images, éventuellement les jsp)
  - un dossier WEB-INF qui contient
    - le fichier de configuration web.xml
    - un dossier «classes» pour les fichiers java compilés
    - un dossier «lib» pour les bibliothèques jar.
  - WEB-INF n'est pas visible par les clients web. On peut aussi l'utiliser pour cacher des ressources.

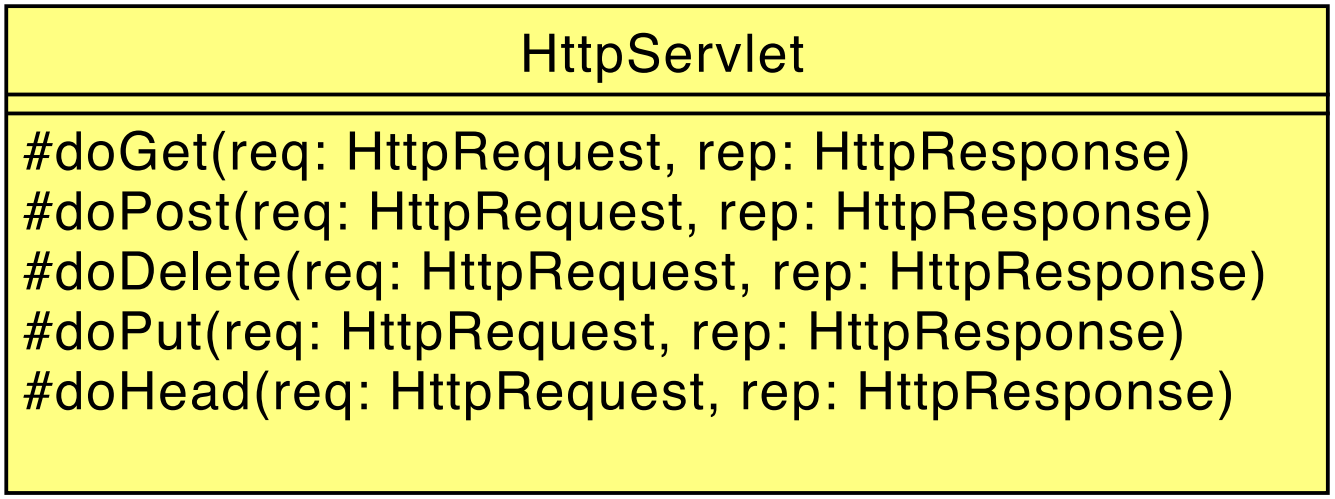
Spring et serveurs applicatifs

- Une archive .war peut se baser sur Spring
- Springboot permet de réaliser facilement un exécutable indépendant qui embarque le serveur comme une dépendance (tomcat par défaut)
- un *war* peut être créé à partir d'une application Springboot (mettre le plugin 'war' dans gradle)

Les servlets

- Au départ, classe permettant d'étendre les fonctionnalités d'un serveur **quelconque**
- méthodes
  - init(ServletConfig) : appelée lors de la configuration de la servlet par le conteneur
  - destroy() : appelée quand le conteneur décide de détruire la servlet
  - service(ServletRequest req, ServletResponse res) : appelée lorsque le serveur est interrogé

HttpServlet



- ne redéfinir que les méthodes qui nous intéressent
- les autres renvoient une erreur 405 (méthode non définie)

H

Annotation

Fixe l'URL

```
@WebServlet(name = "HelloServlet", urlPatterns = {"/hello"})
public class HelloServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<body>");
        out.println("Bonjour " +
                request.getParameter("nom"));
        out.println("</body>");
        out.println("</html>");
        out.close();

    }
}
```

Nous écrivons la réponse

On envoie de l'UML codé en UTF-8

récupération d'un paramètre

Servlet

- Annotation : @WebServlet(name = "HelloServlet", urlPatterns = {"/hello"}) : associe la servlet et l'URL «/hello»
- request : représente la requête ; donne accès aux paramètres
- méthodes utiles sur request :
  - String getParameter(String): valeur d'un paramètre
  - String [] getParameterValues(String) : valeur**s** d'un paramètre multi-valué (select, checkboxes...)
- response : représente la réponse ; permet d'écrire le résultat, de renvoyer des redirections…

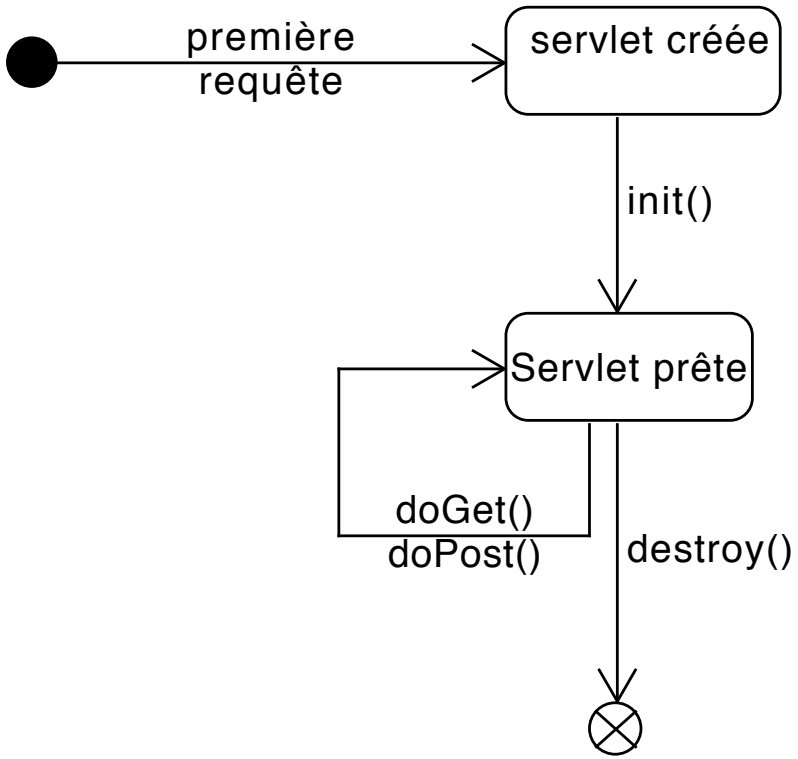
Place des servlets dans l'architecture d'une application

- Reçoivent les requêtes
- font éventuellement de l'affichage **En fait, on fera gérer l'affichage par d'autres système, par ex. les JSP ou thymeleaf** (cours suivant)
- font donc partie de **l'interface utilisateur**

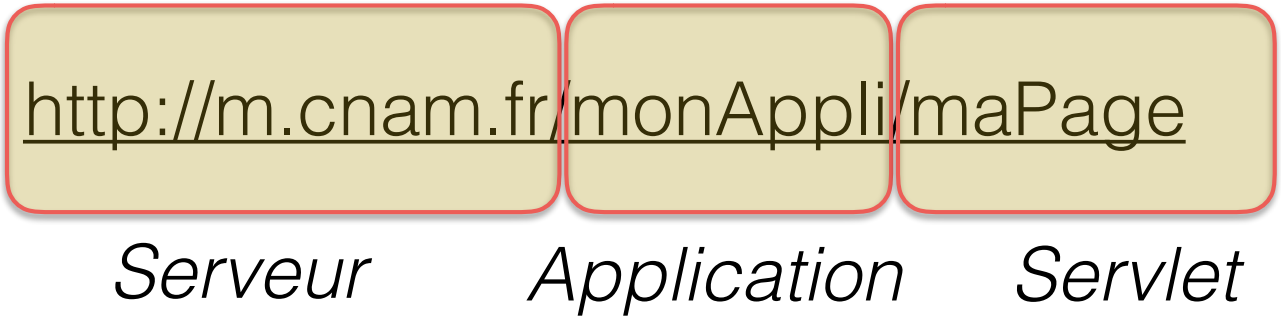
Fonctionnement d'une application à base de servlets

- Quand on visite l'URL associée à une servlet
  - l'objet servlet correspondant est créé s'il n'existe pas
  - il est utilisé pour répondre à la requête
  - il est *généralement* conservé en mémoire
    - les requêtes à la même adresse suivantes l'utiliseront
    - si plusieurs requêtes simultanées pour la même URL, traitement en parallèle !
    - problèmes éventuels de multi-threading **(pas thread-safe)**
- Avec Spring: injection de dépendances dans la servlet

Cycle de vie d'une servlet



URL et servlet



- l'URL donnée dans la configuration est relative à *l'application*
- *req.getContextPath()* renvoie le chemin de l'application. (ici: /monAppli)

L'annotation @WebServlet

- urlPatterns : liste des patterns d'URL pris en charge par cette servlet :

```
@WebServlet(name = "AutreServlet", urlPatterns = {"/home", "/accueil"})
```

- le pattern peut se terminer par une « * » :

```
@WebServlet(name = "Control", urlPatterns = {"/faire/*"})
```

- (utile pour les front controllers)
- on récupère la partie variable avec request.getPathInfo()

@WebServlet

- name: nom de la servlet (utilisé pour y faire référence ailleurs)
- asyncSupported : peut fonctionner de manière asynchrone (voir https://blogs.oracle.com/enterprisetechtips/entry/asynchronous_support_in_servlet_3)
- loadOnStartup : si spécifié, permet la création de la servlet au chargement de l'application.

Path avec « * »

- Pour un path du type /faire/*, on attends par exemple des URL de la forme /faire/afficher/3
- on veut alors extraire « afficher/3 » (avec request.getPathInfo() et le traiter
- (en réalité, on utilisera un framework qui gère ce type de problèmes, comme JAX-RS) ;

Expédition de données non HTML

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
            throws ServletException, IOException {
 response.setContentType("image/png");
 BufferedImage buff= new BufferedImage(100, 100,
                                       BufferedImage.TYPE_INT_RGB);
 Graphics g= buff.getGraphics();
 g.setColor(Color.YELLOW);
 g.fillRect(0, 0, 100, 100);
 g.setColor(Color.RED);
 g.drawLine(0, 0, 100, 100);
 g.dispose();
 ServletOutputStream out = response.getOutputStream();
 ImageIO.write(buff, "png", out);
 out.close();
}
```

web.xml

- Au lieu des annotations: respecte mieux la séparation configuration/code
- mais pénible… (ben oui, le XML c'est #!*)
- deux déclarations pour une servlet: la servlet, et son mapping URL

```
<web-app>
    <servlet>
        <servlet-name>SalutServlet</servlet-name>
        <servlet-class>glg203.SalutServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SalutServlet</servlet-name>
        <url-pattern>/salut</url-pattern>
    </servlet-mapping>
</web-app>
```

web.xml

- permet aussi de configurer session-timeout

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 …
   <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Avec Springboot

- Dans gradle

```
dependencies {
    implementation
          'org.springframework.boot:spring-boot-starter-web'
  …
}
```

- Annoter la configuration avec @ServletComponentScan (ne fonctionne pas pour des fichiers .war)

Spring seul

- Dans ce cas, c'est le serveur applicatif qui va mettre Spring en place et non l'inverse
- … on en reparle à la fin du prochain cours.

Les beans

Beans et JSP/Servlets

- HTTP est sans état : où conserver de l'information ????
- dans une base de données ?
  - pour le long terme
  - aujourd'hui, très souvent utilisé pour le cours terme dans le cas de load balancing, souvent avec des bases noSQL
- en mémoire : **les beans**

Notion de bean

- Au départ, développée pour les interfaces graphiques
- idée: un objet java qu'on peut facilement sauvegarder, et dont on peut découvrir et manipuler les propriétés

bean

- ne pas confondre avec un EJB (enterprise java bean !)
- Conditions pour qu'un objet puisse être un bean
  - il doit avoir un constructeur par défaut
  - ses propriétés doivent être manipulables par des getters et des setters
  - il doit implémenter sérializable
- en fait, pour notre cas, on n'a pas toujours besoin de « vrais » beans

Beans dans le contexte JSP/Servlet

- Un bean est un objet java presque quelconque, stocké en mémoire le plus souvent
- un bean a un **nom** qui l'identifie
- il peut éventuellement être sérialisé pour libérer de l'espace mémoire
- un bean a une portée ou durée de vie, qui indique qui a accès à sa valeur

bean et durée de vie

- bean application : partagé par tous les utilisateurs, disparaît quand l'application se termine ;
- bean session : permet de conserver les données pour une session d'un utilisateur ;
- bean request : vit le temps d'une requête
- bean page : vit le temps de l'affichage d'une JSP

Les beans application

- Permettent de conserver en mémoire des données partagées par tous
- risque majeur : accès concurrent. Le bean doit être soit immuable, soit thread-safe (difficile et/ou long)
- adapté aux informations invariables
- envisager l'usage de ConcurrentHashMap, par exemple, dans le cas contraire
- ***à utiliser avec précaution !!!***
- En Spring, on n'a pas vraiment de raison de s'en servir…

Création d'un bean application

- Le bean pouvant être sérialisé, il est conseillé qu'il soit sérialisable
- L'objet servletContext, accessible à l'aide de request.getServletContext(), gère une map String → bean
  - req.getServletContext().setAttribute(name, valeur) pour stocker l'objet dans l'application
  - req.getServletContext().getAttribute(name) : renvoie l'objet (ou null)

Bean application (exemple)

```
@Override
protected void doGet(HttpServletRequest req,
          HttpServletResponse resp)
          throws ServletException, IOException {
 Repertoire rep=
    (Repertoire)req.getServletContext().getAttribute("repertoire");
 if (rep== null) {
    rep= RepertoireFactory.build();
      req.getServletContext().setAttribute("repertoire", rep);
 }
 …
```

attention, cet exemple est erroné (voir fin du cours)

Les beans session

- Une *session* est un espace de travail où l'on stocke temporairement les données relatives à un utilisateur et à une application précise
- la session est fermée quand l'utilisateur quitte son navigateur
- la session se termine au bout d'un certain temps d'inactivité (30min par défaut)
- exemple typique de bean session : un panier d'achat (*shopping cart*)
- utilisable aussi pour gérer les données de connexion d'un utilisateur

Comment fonctionne la session ??????????

- http sans état →pas de session… mais :
- quand on veut créer une session, le serveur applicatif crée un très grand nombre aléatoire, l'identifiant de session
- ce numéro est utilisé en mémoire comme clef pour la session ;
- il est transmis à l'utilisateur, et géré :
  - soit comme cookie
  - soit comme paramètre dans l'URL :

    index;jsessionid=f0e4870fd0fe697512eb2edfc0fd
- la session consomme des resources: par défaut, pas de session !

Session

- On accède ou on crée la session à travers l'objet request :
  - HttpSession s= request.getSession()
    - retourne la session courante (en la créant si elle n'existe pas)
  - HttpSession s= request.getSession(false)
    - retourne la session courante, ou null s'il n'y en a pas

Beans session

```
@Override
protected void doGet(HttpServletRequest req,
     HttpServletResponse resp)
     throws ServletException, IOException {
  ShoppingCart cart= (ShoppingCart)
           req.getSession().getAttribute("cart");
  if (cart==null) {
    cart= new ShoppingCart();
    req.getSession().setAttribute("cart", cart);
  }
…
```

:tention, cet exemple est erroné, nous reviendrons dessus à la fin du cours.

Les beans request

- Utilisés pour passer de l'information entre une servlet et une jsp, quand la servlet sert de contrôle et la jsp de vue
- on stocke les beans requests dans l'objet request, avec
  - req.setAttribute(name, valeur)
  - req.getAttribute(name)
- pas de problème de concurrence

Les cookies

- font partie du protocole HTTP
- leur valeur est passée lors des envois de requêtes et de réponses dans l'en-tête HTTP

Set-Cookie : NOM=VALEUR; domain=NOM_DE_DOMAINE; expires=DATE

- On crée des objets de classe Cookie, et on les expédie grâce à reps.addCookie(cookie) ;
- On récupère les cookies envoyés par le client avec req.getCookies()
- un cookie est du TEXTE
- la taille de NOM=VALEUR est limitée à 4096 octets

Cookies

```
@WebServlet(name = "Compter", urlPatterns = {"/Compter"})
public class Compter extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse resp)
                         throws ServletException, IOException {
        resp.setContentType("text/plain");
        int cpt= 0;
        Cookie[] cookies= req.getCookies();
        for (Cookie c: cookies) {
            if (c.getName().equals("compteur")) {
                cpt= Integer.parseInt(c.getValue());
            }
        }
        cpt= cpt + 1;
        resp.addCookie(new Cookie("compteur", ""+ cpt));
        resp.getWriter().write(""+cpt);
        resp.getWriter().close();
    }
}
```

cookies

- On peut changer la durée de vie d'un cookie : cookie.setMaxAge(n); (en secondes)
- on peut détruire un cookie: cookie.setMaxAge(0);

Session et Codage des URL

- Si les cookies sont désactivés sur le navigateur, il faut envoyer le numéro de session en paramètre
- pour écrire les liens et les URL, on utilise alors :
  - response.encodeURL(url)
  - ou response.encodeRedirectURL(url)

Servlets et multitâche

- Quand il y a plusieurs requêtes simultanées, elles sont traitées en parallèle dans des tâches différentes ;
- le même objet servlet peut être utilisé par plusieurs tâches simultanément ;
- pas de problème pour les variables locales de doGet et doPost ;
- problèmes potentiels si on a des variables d'instance ;
- conseil : ne pas avoir de variables d'instance (ou alors, des constantes).

Beans et multitâche

- https://www.ibm.com/developerworks/library/j-jtp09238/

Initialisation du code d'une application

- On met en place des listeners (pattern observateur) qui sont prévenus quand l'état de l'application change, et en particulier au début de celle-ci
- la méthode contextInitialized est appelée au lancement de l'application, et peut stocker des informations dans le servletContext.

```java
@WebListener
public class Informations implements ServletContextListener{

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.err.println("ON VIENT DE COMMENCER");
        sce.getServletContext().setAttribute("debut", new Date());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.err.println("ON VIENT DE SE TERMINER");
    }

}
```

Les filtres

- Utilisent le pattern « chaîne de responsabilités » pour ajouter des fonctionalités aux servlets
  - codage
  - log
  - sécurité

```java
@WebFilter(urlPatterns = "/*")
public class LogWebFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig)
          throws ServletException {
        System.err.println("ON crée le filtre");
    }

    @Override
    public void doFilter(ServletRequest request,
       ServletResponse response, FilterChain chain)
       throws IOException, ServletException {
        System.err.println("ON FILTRE !!!!");
        // on passe au suivant !
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
        System.err.println("ON détruit le filtre");
    }

}
```

Filtres

- pour la sécurité: la méthode doFilter peut bloquer l'affichage d'une page (en ne s'appelant pas sur *chain*)
- on peut factoriser la mise en place d'un codage pour le texte :

```
ServletResponse response, FilterChain filterChain) … {
        request.setCharacterEncoding("utf-8");
        response.setCharacterEncoding("utf-8");
        filterChain.doFilter(request, response);
}
```

Limites des servlets

- La génération directe de HTML est pénible
  - illisible
  - difficile à maintenir
- Solution : langages de templates
  - maison (bof)
  - JSP (probablement trop générique)
  - JSF (puissant et pratique; cycle de vie **très** spécifique)
  - **Thymeleaf**
- De plus en plus : HTML fixe, partie variable: JavaScript plus REST

Concurrence et création des beans

- Le code

```
@Override
protected void doGet(…) …{
 Repertoire rep=
    (Repertoire)req.getServletContext().getAttribute("repertoire");
 if (rep == null) {
    rep= RepertoireFactory.build();
    req.getServletContext().setAttribute("repertoire", rep);
 }
 …
```

- est incorrect: si rep est null, plusieurs threads pourraient vouloir le créer simultanément !

Bean application et multithreading

- Les méthodes getAttribute() et setAttribute() sont thread-safe… pas de problème de ce côté
- les beans eux-même ne le sont pas forcément !
- et les **séquences** d'appels aux beans ne sont pas thread-safe !

multithreading

appel 1

```
Repertoire rep=
   (Repertoire)req.getServletContext().getAttribute("repertoire");

if (rep== null) {
    rep= RepertoireFactory.build();
    req.getServletContext().setAttribute("repertoire", rep);
}
…
```

appel 2

- entre l'appel 1 et l'appel 2, un autre thread a pu créer et enregistrer un autre répertoire
- pour que l'appel soit sûr :
  - synchroniser la séquence
  - … ou faire créer les beans applications en début d'application par des ServletContextListener. C'est le plus simple et le plus sûr.

Bean session

- Attention:
  - comme un bean application, les beans sessions peuvent potentiellement être utilisés de manière concurrente
  - si l'utilisateur charge plusieurs pages
  - si la page courante charge des pages secondaires (ajax par exemple)
- avoir ce problème à l'esprit.

Initialisation correcte des beans sessions

- première solution : utiliser un HttpSessionListener
- seconde solution: utiliser la synchronisation. Sur quoi ?
  - *pas de garantie indépendante du serveur quand à l'unicité de l'objet renvoyé par getSession (ça peut être un wrapper temporaire)*
  - *dans les JSP, c'est ce que fait glass fish.*
  - seule possibilité sûre : synchroniser sur un bean session.
- voir http://stackoverflow.com/questions/9802165/is-synchronization-within-an-httpsession-feasible et surtout http://www.ibm.com/developerworks/library/j-jtp09238/index.html

Utilisation de la synchronisation

- pas très portable (dépend du serveur)
- sous glass fish, utilisé dans les jsp :

```
demo.Panier panier = null;
synchronized (session) {
    panier = (demo.Panier) session.getAttribute("panier");
    if (panier == null){
        panier = new demo.Panier();
        session.setAttribute("panier", panier);
    }
}
```

Double-checked locking (hors programme)

- On trouve sur le web des solutions qui utilisent le pattern du « double checked locking »
- celui-ci permet d'éviter de poser un verrou si l'objet existe déjà
- mais il est délicat à utiliser (et la plupart des exemples donnés sont faux) :
  - https://fr.wikipedia.org/wiki/Double-checked_locking

Solution avec un HttpSessionListener

```
@WebListener
public class CompteurSetter implements HttpSessionListener{

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.err.println("On initialise la session...");
        se.getSession().setAttribute("cpt", new Compteur());
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
    }

}
```

Dans la servlet…

- Le bean existe déjà !

```
@WebServlet(urlPatterns = {"/demoInit"})
public class DemoInitBeanSession extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse resp)
                         throws ServletException, IOException {
        Compteur c = (Compteur) req.getSession().getAttribute("cpt");
        int v = c.nextVal();
        resp.getWriter().append("Compteur " + v);
        resp.getWriter().close();
    }
}
```

Approches pour beans partagés

- une classe avec des méthodes synchronisées. Difficile à bien réaliser (une ***séquence*** d'appels de méthodes synchronisées n'est pas synchronisée)
- classes immuables… pas de problème de modification
- ConcurrentHashMap et les classes de java.util.concurrent en général
- AtomicInteger et les classes de java.util.concurrent.atomic.
  - exemple pour un compteur :
  - int newVal= atomicInt.addAndGet(1);
- frameworks comme akka

Servlets asynchrones

- Une servlet capable de mettre des requêtes « en attente » sans surcharger le serveur.
- Idée :
  - on tient à jour une liste de requêtes en attente
  - quand on a les données nécessaires pour leur répondre, on répond aux clients
- évite de faire de bombarder le serveur de requêtes inutiles

Servlets asynchrones

- déclare asyncSupported = true dans son annotation ;
- tient à jour une liste de AsyncContext
- quand on reçoit une requête, on crée un nouvel AsyncContext (avec les données de la requête) et on l'ajoute à la liste
- les clients attendent la réponse…
- quand on peut répondre, on boucle sur les AsyncContext, on écrit leurs réponses, puis on les ferme avec la méthode complete().
- généralement utilisé avec AJAX.