Construction d'applications web

Chapitre 2

Applicatifs côté serveur en java
Servlets et Container de Servlets

Pages Web dynamiques

- Un serveur Web (par ex. tomcat) écoute sur le port 80 (ou 8080 pour les TP)
- Il répond aux requêtes HTTP des clients (GET ou POST + URL)
- Principe des pages web statiques :
  - une URL = le chemin d'accès à un fichier (HTML, JPEG...)
  - En réponse à un GET, le serveur accède au fichier et en renvoie le contenu
- Principe des pages web dynamiques :
  - une URL correspond à un service/une action
  - en réponse à un GET ou POST, le serveur exécute une **procédure**

Pages Web dynamiques

- Dans le cas de tomcat : une URL donnée est gérée par un objet appelé servlet
- tomcat est un « container de servlets »
- En réponse à une requête :
  - l'objet correspondant à l'URL est créé et initialisé s'il n'existe pas déjà
  - une méthode de cet objet correspondant au type de requête est appelée, avec deux paramètres :
  - en entrée : un objet HttpServletRequest, contenant le contenu de la requête
  - en sortie : un objet HttpServletResponse, dans lequel on écrira ce qu'il faut renvoyer au client

Java EE : principes généraux

- Editions de Java :
  - Java ME (Micro Edition) : embarqué (applications mobiles)
  - Java SE (Standard Edition) : généraliste (appli pour poste de travail)
  - **Java EE (Enterprise Edition) : entreprise (multi-niveaux)**
  - Java FX : clients riches
- Objectif de java EE :
  - Surcouche de Java SE (pas de nouvelle VM)
  - Prendre en charge les aspects “durs”
  - Laisser le développeur se concentrer sur le métier

Principe de Java EE

- Approche “classique”
  - Librairie d'objets techniques
  - Développement d'objets métiers
  - Fournir le “main()”
    - Instancie les objets
    - Appel les instances
- Pattern “Inversion de Contrôle”
  - Environnement Runtime / Moteur d’exécution
    - Prend en charge les aspects techniques
    - Contrôle et coordonne l'activité de l'application
    - Containers
  - Développeur :
    - Confie ses “.class” aux containers

Principe

- JavaEE
  - API
  - Runtimes (Containers)
  - Programmation Internet => Container Web
  - Exemples de containers web
    - Tomcat (implémentation de référence)
    - Jetty
  - Remarque : serveurs d'application
    - Containers web (généralement tomcat)
    - Autres containers (EJB, JMS, WebServices etc)
    - Ex : glassfish, jboss, jonas, websphere, etc
    - Existent sous plusieurs profils dont le profil Web

Container Web : rôle

- Gestion des requêtes / réponses HTTP
  - Déclenchement de méthodes métiers
  - Prise en charge des sessions / cookies
- Mécanismes d'identification / authentification
- Gestion de pools de connexion
- Déploiement des objets métiers (cœur de l'application)
- etc.

Servlets

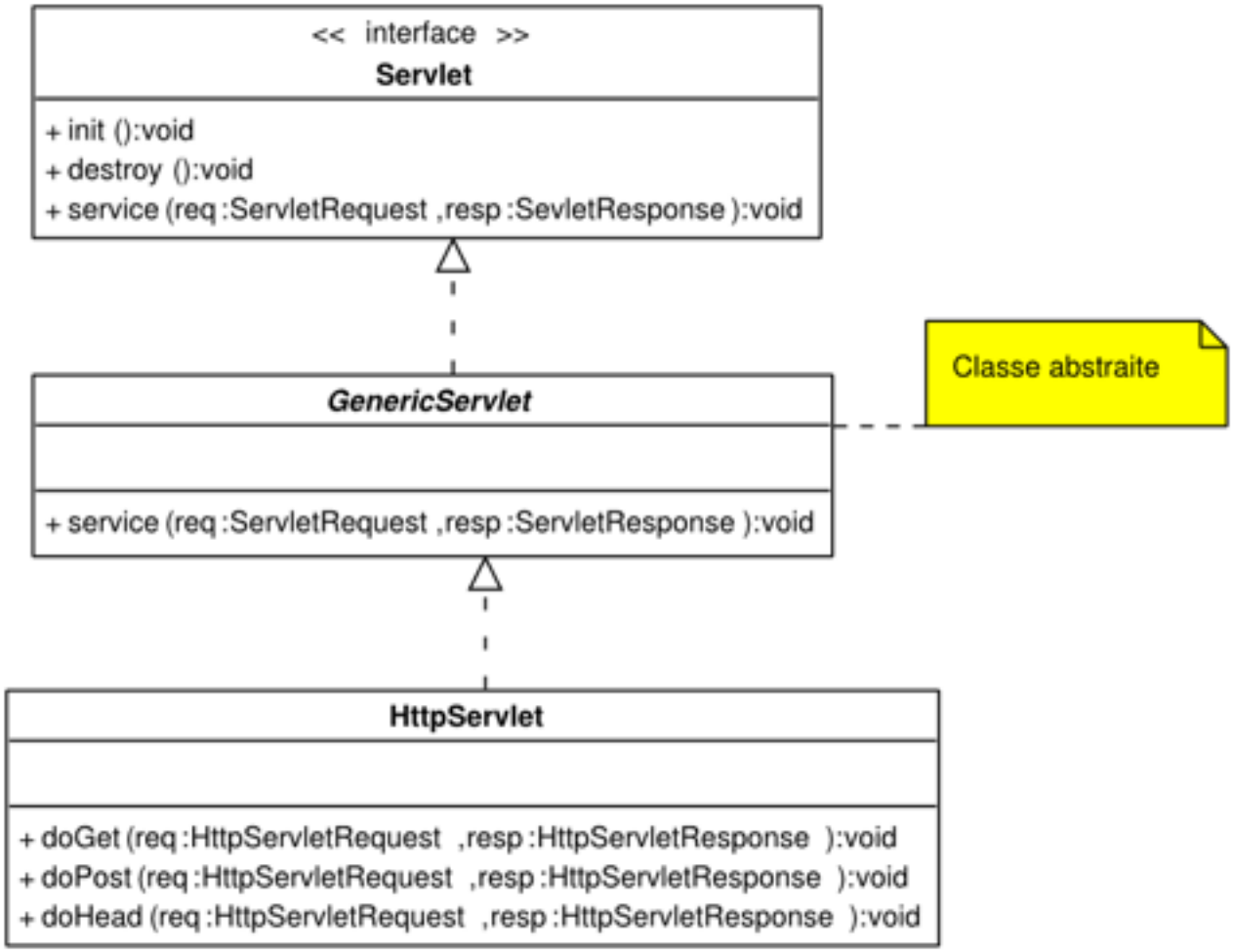
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet(name = "date", urlPatterns = {"/date"})
public class DateServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
            throws ServletException, IOException {

    response.setContentType("text/plain;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
      out.println("The time is: " + new java.util.Date());
    }
    // out est automatiquement fermé à l'issue du try
    // quelle que soit l'issue (exception ou non)
  }
}
```

Servlets


<< interface >>
Servlet
+ init ():void
+ destroy ():void
+ service (req :ServletRequest ,resp :SevletResponse):void
GenericServlet
+ service (req :ServletRequest ,resp :ServletResponse):void
Classe abstraite
HttpServlet
+ doGet (req :HttpServletRequest ,resp :HttpServletResponse):void
+ doPost (req :HttpServletRequest ,resp :HttpServletResponse):void
+ doHead (req :HttpServletRequest ,resp :HttpServletResponse):void

Servlets

La classe `HttpServlet`

Rôle : traiter la requête HTTP

Pour chaque méthode HTTP : GET, POST, PUT, DELETE, etc. il y a une méthode correspondante :

`doGet(…)` - répond aux requêtes HTTP GET

`doPost(…)` - répond aux requêtes HTTP POST

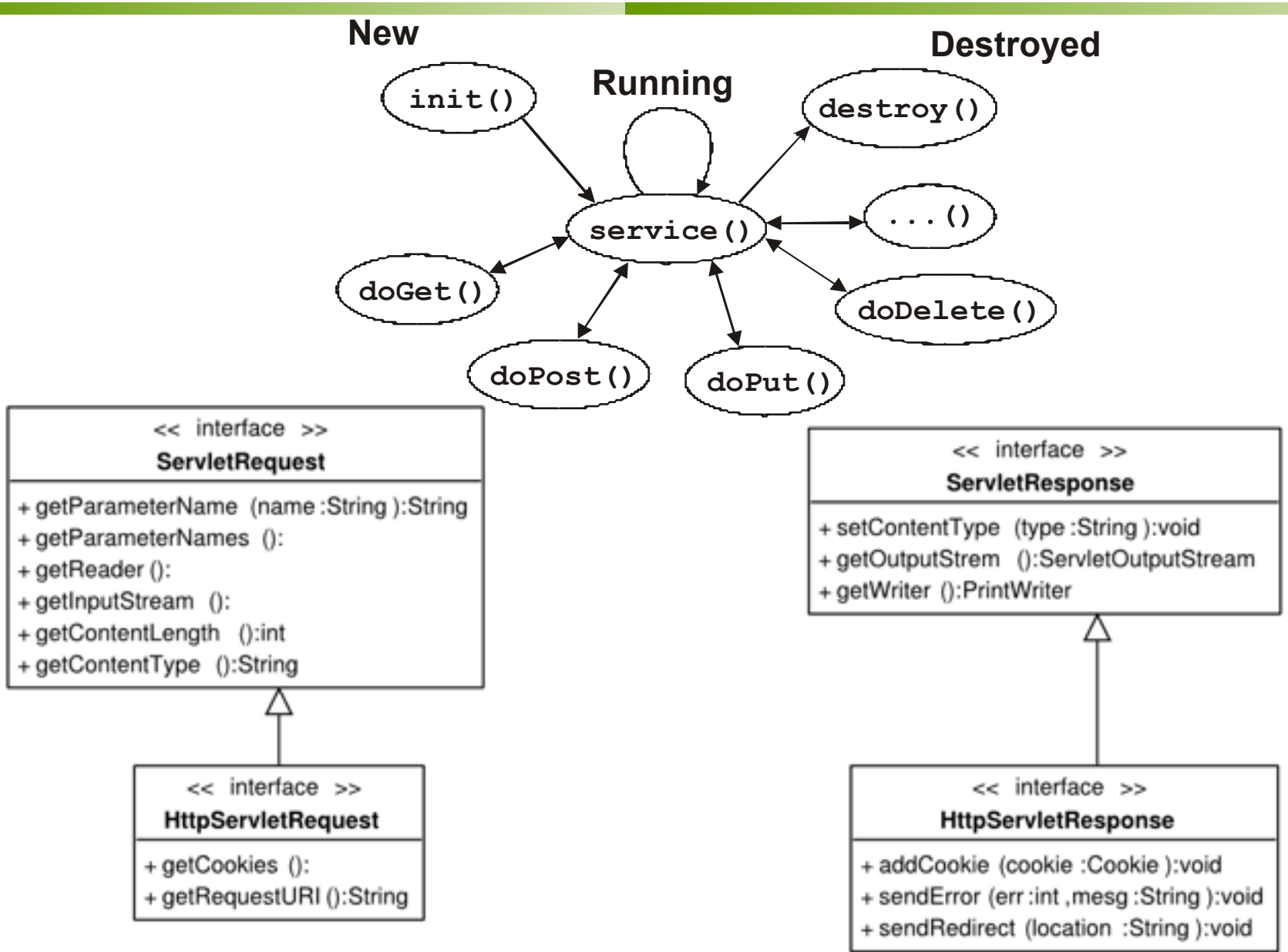
`doPut(…)`, `doHead(…)`, `doDelete(…)`, `doTrace(…)`, `doOptions(…)`

Voir sur la page Chamilo les liens vers la Javadoc de HTTPServlet et Servlet, et également HTTPServletRequest et HTTPServletResponse.

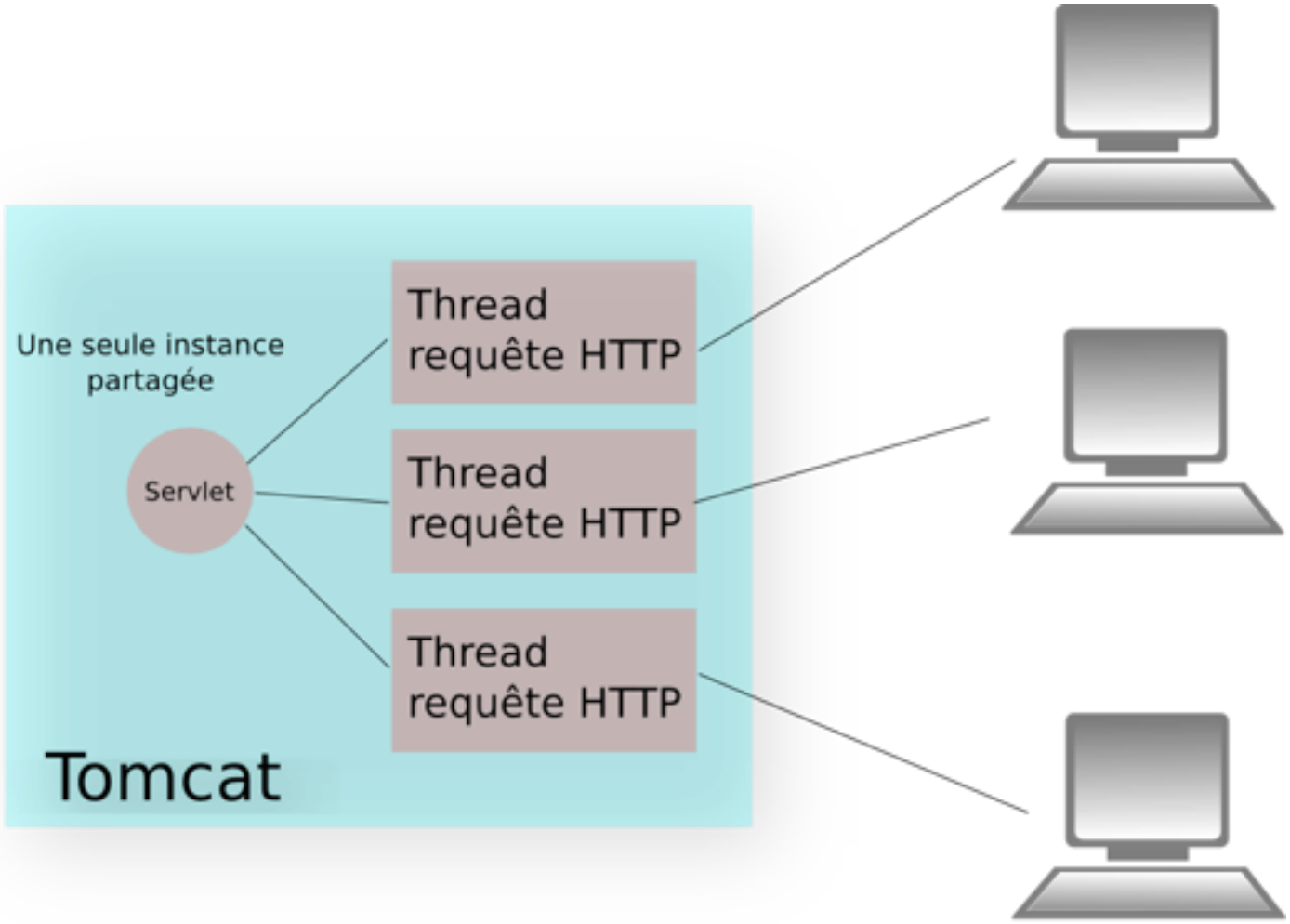
doGet() et doPost()

- Ces méthodes ont deux paramètres : la requête et la réponse
- L'objet `HttpServletRequest`
  - Contient la requête du client
    - En-tête de la requête HTTP
    - Paramètres HTTP (données de formulaire ou paramètres passés avec l'URL)
    - Autres données (cookies, URL, chemin relatif, etc.)
- L'objet `HttpServletResponse`
  - Encapsule les données renvoyées au client
    - En-tête de réponse HTTP (content type, cookies, etc.)
    - Corps de la réponse (en tant que `OutputStream`)

Requests, Responses



Modèle d'exécution



Cycle de vie

1. le serveur crée un pool de threads auxquels il va pouvoir affecter chaque requête
2. La servlet est chargée au démarrage du serveur ou lors de la première requête
3. La servlet est instanciée par le serveur
4. La méthode *init()* est invoquée par le conteneur
5. Lors de la première requête, le conteneur crée les objets Request et Response spécifiques à la requête
6. La méthode *service()* est appelée à chaque requête dans un nouveau thread. Les objets *Request* et *Response* lui sont passés en paramètre
7. Grâce à l'objet *Request*, la méthode *service()* va pouvoir analyser les informations en provenance du client
8. Grâce à l'objet *Response*, la méthode *service()* va fournir une réponse au client
9. La méthode *destroy()* est appelée lors du déchargement de la servlet, c'est-à-dire lorsqu'elle n'est plus requise par le serveur. La servlet est alors signalée au *garbage collector*

La méthode init()

1. Appelée à la première instanciation,
2. La spécification garantit qu'aucune requête ne sera envoyée avant que init() ne soit exécutée entièrement.
3. On redéfinit `init()` quand :

   On doit ouvrir des ressources (connexions JDBC par ex)

   On doit initialiser l'état d'une Servlet...

La méthode destroy()

- Appelée avant que la Servlet soit collectée par le garbage collector,
- La spécification garantit que toutes les requêtes seront traitées avant l'appel à destroy()
- On la redéfinit lorsque :
  - On doit libérer des ressources,
  - On doit rendre persistant l'état de la servlet (en général fait par des librairies spécialisées comme les Web Services).

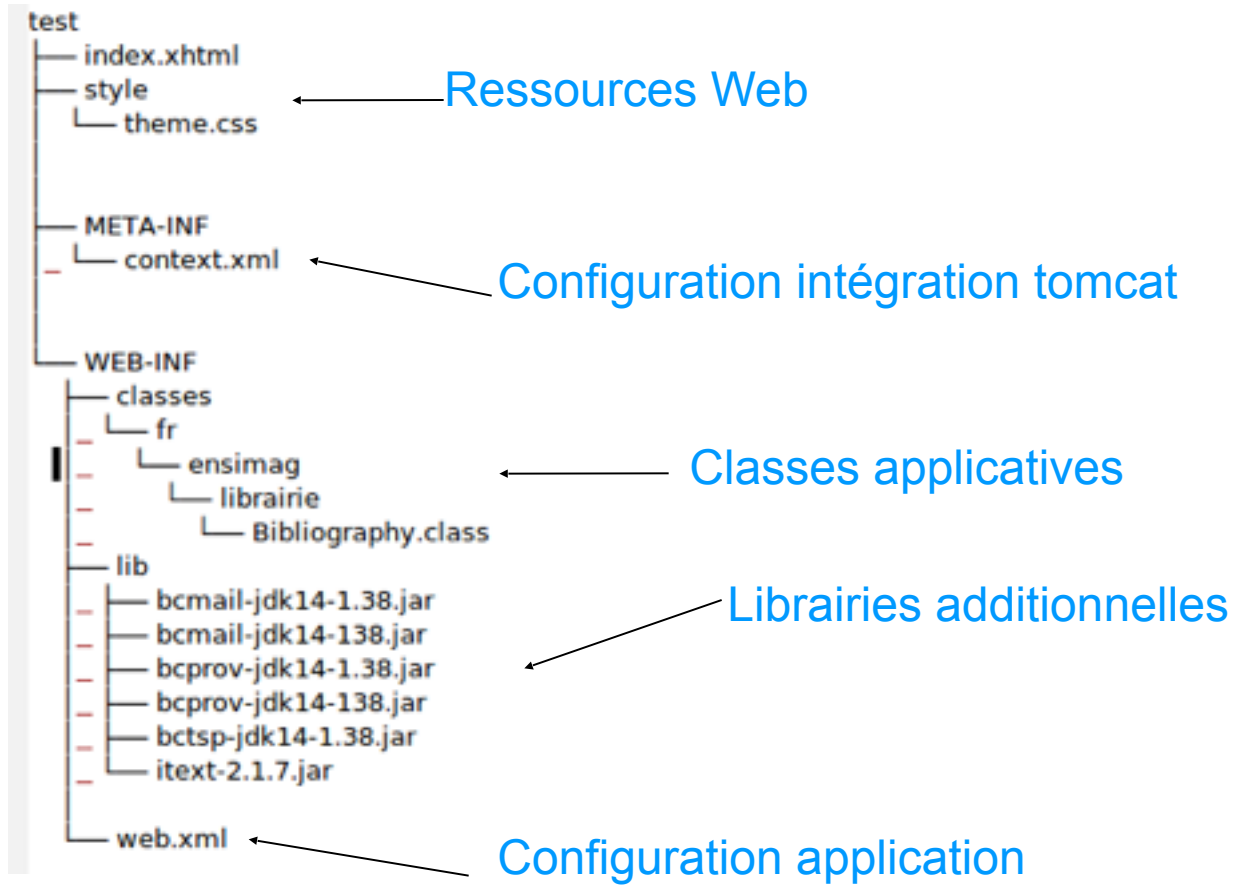
Déploiement

- Ajouter au serveur une application web (“context”)
  - Objets métiers + techniques (.class)
  - Librairies additionnelles (jar)
  - Pages HTML, images, feuilles CSS, scripts Javascript...
  - etc.
- Fichiers de configuration
  - web.xml : configuration de l’application
    - fichier optionnel depuis Java EE6
    - remplacé par des **annotations** (@WebServlet) indiquant quel servlet répond à quelle URL, etc.
  - context.xml : config. l’intégration dans tomcat
    - URL de base (racine de l’application), log, etc.
- Packaging : war (Web Archive)

Structure d'archive

- Arborescence normalisée (Java EE)
- Exemple

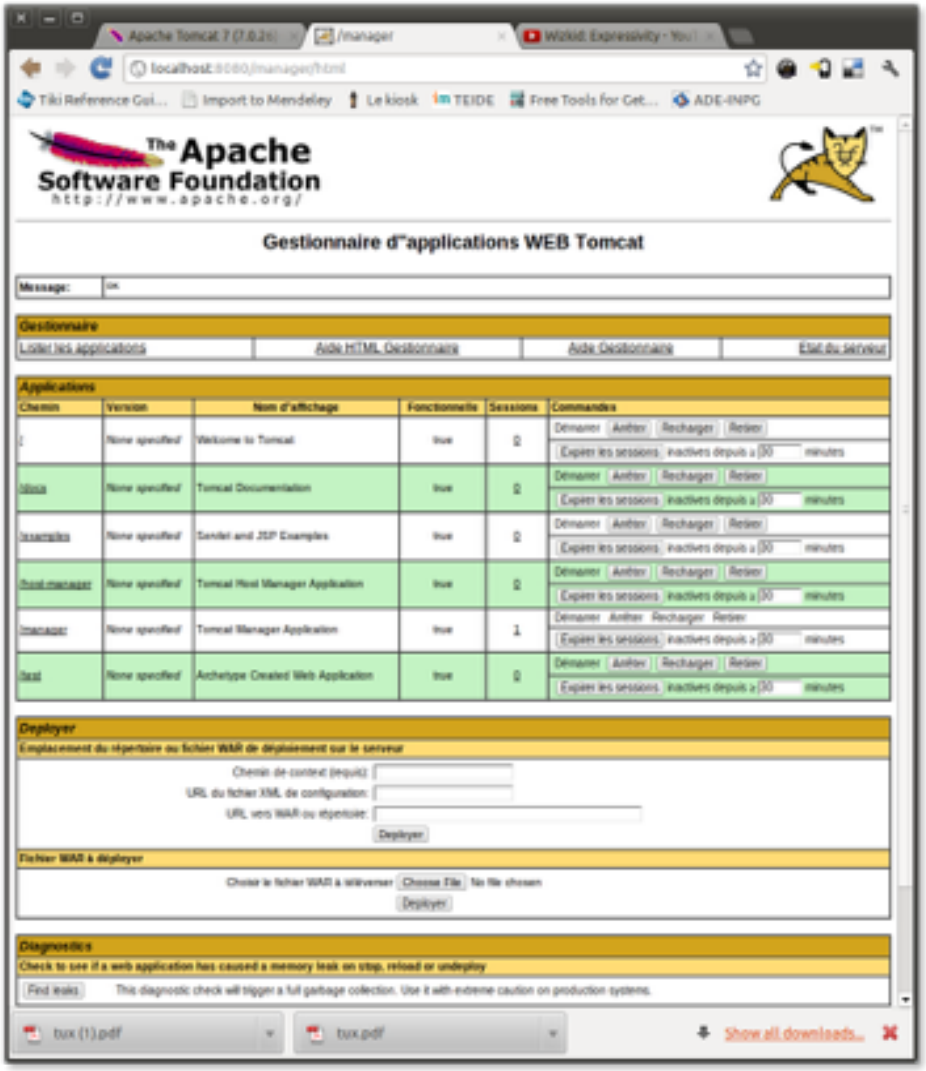
test.war



Déploiement

- 2 modes :
  - Statique : avant de démarrer Tomcat
  - Dynamique : déploiement à chaud
- Comment :
  - Copie du war → webapps de Tomcat
    - Extraction de l'archive lors du démarrage
  - Manager
    - Interface HTML
    - Interface script
  - ....

Déploiement : Manager HTML



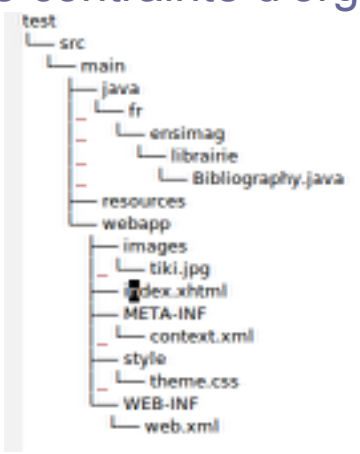
Contrôle d'accès :

conf/tomcat-users.xml

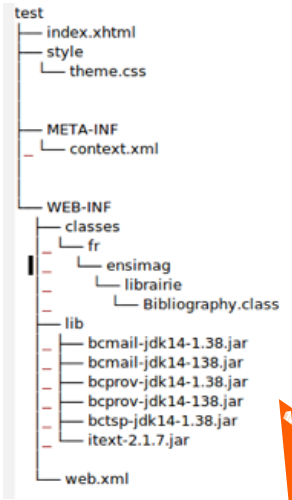
```
<tomcat-users>
  <role rolename="manager-gui"/>
  <role rolename="manager-script"/>
  <role rolename="admin-gui"/>
  <role rolename="admin-script"/>
  <user username="root"
          password="blabla"
          roles="manager-gui,manager script,admi
n-gui,admin-script"/>
</tomcat-users>
```

Cycle de développement

Arborescence source
(aucune contrainte d'organisation)

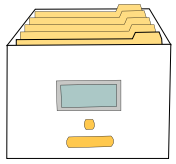


Arborescence binaire (normalisée)



1) compilation

2) archive



test.war

3) déploiement

4) Mise à jour du source

Automatisation du cycle

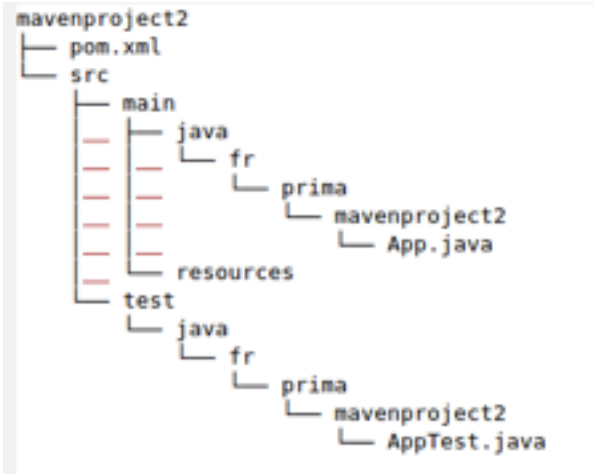
- Nécessité d'automatiser les étapes
  - cf Makefile
- Gestion de production de logiciels java :
  - Ant
  - Maven
  - Ivy
  - Etc
- Utilisation de Maven

Grands principes (survol....)

- Build projet logiciel : repose sur des phases
  - Compile
  - Test
  - Package
  - Install
  - Deploy
- Prises en charge par défaut
  - Inutile de les décrire (‡ makefile, ant)
- Fonction du type de projet :
  - Java : package = jar
  - Projet web : package = war
  - etc.

Gands Principes(survol....)

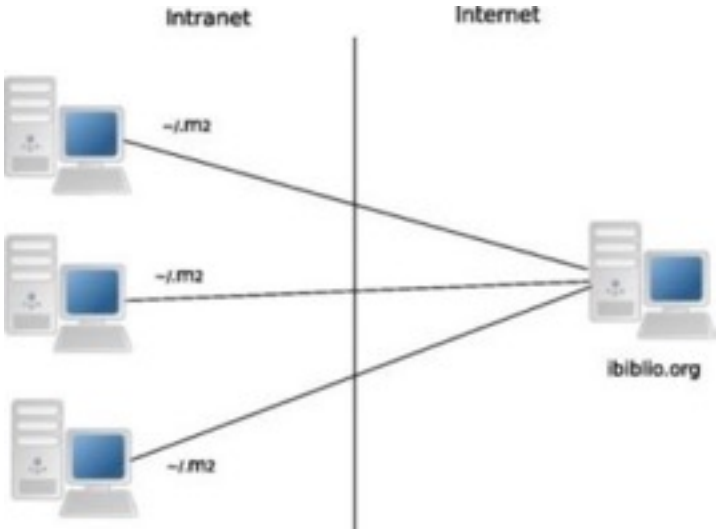
- Organisation standard des sources
  - Limite les besoins de configuration
- Plugins :
  - Permet d'étendre les types de projets
  - De rajouter de nouvelles phases
- Projet Object Model (pom.xml)
  - Fichier de description et de configuration du projet
  - Spécifie les dépendances



Gestion de la dépendance

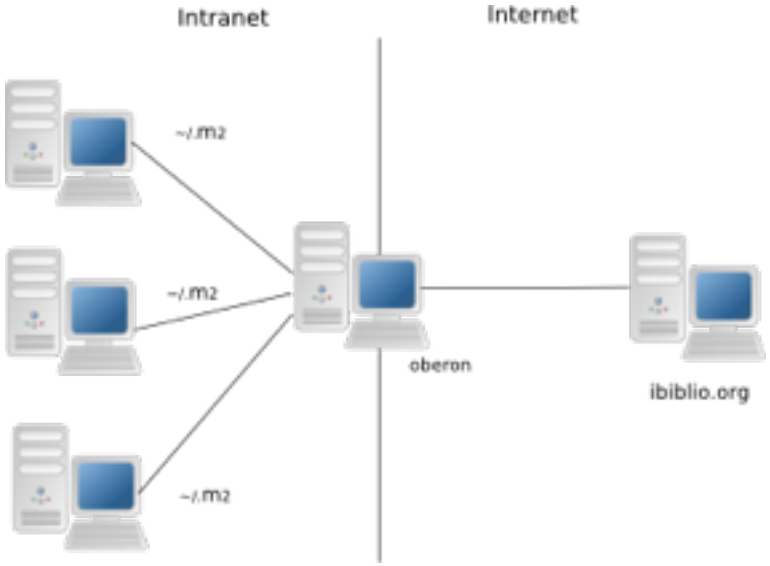
- Transitivité
  - Le projet dépend de A
  - Maven analyse A (à partir de son pom.xml)
    - A dépend de B et C
    - Analyse de B et de C etc.
  - ➔ Le projet dépend de A, B, C etc.
- Les dépendances sont automatiquement téléchargées sur internet et installées en local pour accélérer les prochaines requêtes
- Cache local : ~/.m2

Architecture



Utilisation directe

Mise en place d'un proxy



Quelques avantages de Maven

- Gestion simplifiée des dépendances
- Contrôle précis des versions de librairies
- Un même projet peut être :
  - Compilé en ligne de commande
  - Reconnu nativement par netbeans
  - Reconnu par eclipse par ajout d'un plugin
  - => Le meilleur de tous les mondes
- Inconvénient
  - Syntaxe pas simple à mémoriser
  - Ticket d'entrée

Exemple

- Création d'un projet vide (Archetype)
  - 1 Archetype par type de projet
  - mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp
  - GroupId : classification du projet (ex : fr.ensimag)
  - ArtifactId : nom du projet
- Ajout de la dépendance à tomcat
  - Référencer le plugin dans pom.xml
- Configurer l'accès au manager
  - Login, mot de passe
  - ~/.m2/settings.xml
- Cycle de développement
  - mvn package tomcat:redeploy