HTTPServlet

Application Web : 2 parties

- Front-end
  - A ce stade, nous avons une bonne maîtrise de ce qu'il se passe dans le navigateur.
- Back-end
  - Mais il faut bien quelque chose pour servir les fichiers.
  - Et il faut aussi quelque chose pour répondre aux requêtes

HTTP

- Le front-end émet des requêtes (page web, fichiers javascript, css, ...)
  - Format HTTP
  - Actuellement : HTTP 1.1
  - http://tools.ietf.org/html/rfc2616

Retournons vers Java

- Il nous faut un serveur pour écouter les requêtes Web et y répondre.

- Le composant de Java dédié au traitement d'une requête :
  - HTTPServlet
  - S'utilise au sein d'un serveur Web
  - Un serveur web traite en général beaucoup de requêtes différentes : de multiples HTTPServlet sur un seul serveur

HTTPServlet : Configuration

- Il faut configurer la ou les servlets pour expliquer à Java leur comportement :
  - Quel(s) port(s) écouter
  - Quel(s) chemin(s) dans l'URL correspond(ent) à quel(s) servlet(s)
  - A quel endroit trouver les fichiers à servir par défaut
  - S'il y a un fichier index.html à utiliser en cas d'absence de précision (www.monsite.com servira www.monsite.com/index.html)
  - Et plein d'autres...

Jetty

- En Java, sur un unique serveur Web, on peut exécuter plusieurs applications distinctes.
  - On appelle cela un ‘Application Server’
  - Chaque application est distinguée par son URL
    - www.monsite.com/app1/...
    - www.monsite.com/app2/...
  - On ‘déploie’ son application sur un application server.
- Nous utiliserons Jetty
  - Open source
  - Solution 100% Java
  - Implémente les dernières technologies
  - http://www.eclipse.org/jetty/

Configuration d’une application Web

- web.xml
  - Description XML de la configuration de l’application.
  - Sur un Application Server, chaque application possède son propre web.xml la concernant.

Configuration d'une application Web

- Embedded
  - A la place d'avoir un application server sur lequel on déploie les applications =>
    un processus Java qui embarque son propre application server pour y déployer son unique application.
  - Moins souple qu'un application server, mais beaucoup plus pratique lors du développement
    - Pas de phase de déploiement, c'est une pure application Java qui s'exécute
    - Les outils de debugging habituels restent 100% fonctionnels
    - Nous utiliserons donc cette approche

Jetty : nomenclature

- Server => ce qui écoute sur un port TCP.
- WebAppContext => ce qui configure le serveur en tant qu'application server.
- ServletHolder => retient le nom et la configuration d'une instance de servlet.
- HTTPServlet => classe à étendre pour répondre aux requêtes HTTP.

Jetty : principe

```
// lie le server à un port
   Server server = new Server(8080);
// instancie un WebAppContext pour configurer le server
   WebAppContext context = new WebAppContext();
// Où se trouvent les fichiers (ils seront servis par un DefaultServlet)
   context.setResourceBase("c://web");
// MaServlet répondra aux requêtes commençant par /chemin/
   context.addServlet(new ServletHolder(new
   MaServlet()), "/chemin/*");
// Le DefaultServlet sert des fichiers (html, js, css, images, ...). Il est en général ajouté en dernier pour que les autres servlets soient prioritaires sur l'interprétation des URLs.
   context.addServlet(new ServletHolder(new DefaultServlet()),
"/");
// ce server utilise ce context
   server.setHandler(context);
// allons-y
   server.start();
```

WebAppContext : quelques méthodes

- `context.setContextPath("/");`
  - Chemin de l'URL pour lequel ce contexte s'applique.
- `context.setWelcomeFiles(new String[] { "index.html" });`
  - Quel est le fichier à servir si l'utilisateur va à l'URL racine sans plus de précision. www.monsite.com affichera www.monsite.com/index.html
- `context.setInitParameter("cacheControl","no-store,no-cache,must-revalidate");`
  - Dans le protocole HTTP, le serveur dicte le comportement du cache du navigateur. Ici on dit que par défaut, il ne faut pas stocker ni retenir en cache les réponses aux requêtes.

WebAppContext : quelques méthodes

- `context.setInitParameter("redirectWelcome", "true");`
  - Quand c'est 'true', la page de bienvenue est affichée par redirection plutôt que par suivi. En d'autres termes, l'URL change à la page de bienvenue. Dans tous les cas le contenu de cette page sera affichée.
- `context.setClassLoader(Thread.currentThread().getContextClassLoader());`
  - Pour des raisons de sécurité, il faut préciser sous quelle autorité doit s'effectuer le chargement des classes Java. Ici on utilise simplement l'autorité du Thread en cours.
- `context.setMaxFormContentSize(50000000);`
  - Spécifie la taille limite des données qu'un front-end peut soumettre à ce back-end.

WebAppContext : quelques méthodes

- `context.addServlet(new ServletHolder(new MyServlet()), "/url1");`
  - Enregistre une servlet pour répondre à l'url /url1 exactement.
- `context.addServlet(new ServletHolder(new MyOtherServlet ()), "/url2/*");`
  - Enregistre une servlet pour répondre à toute url commençant par /url2
  - Si plusieurs servlets sont ajoutées, chaque requête est traitée dans l'ordre de cet ajout, jusqu'à trouver une servlet qui accepte de la traiter.
- `context.addServlet(new ServletHolder(new DefaultServlet()), "/");`
  - La DefaultServlet sert des fichiers.
  - Ici toutes les URLs seront traitées comme des fichiers à trouver puisque cela commence à / directement
  - C'est pour cela qu'on ajoute la DefaultServlet en dernier en général.

HTTPServlet

- Cette classe a pour but d'être étendue.
- Chaque requête HTTP servlet appellera la méthode:

```
service(HttpServletRequest i,HttpServletResponse o)
```

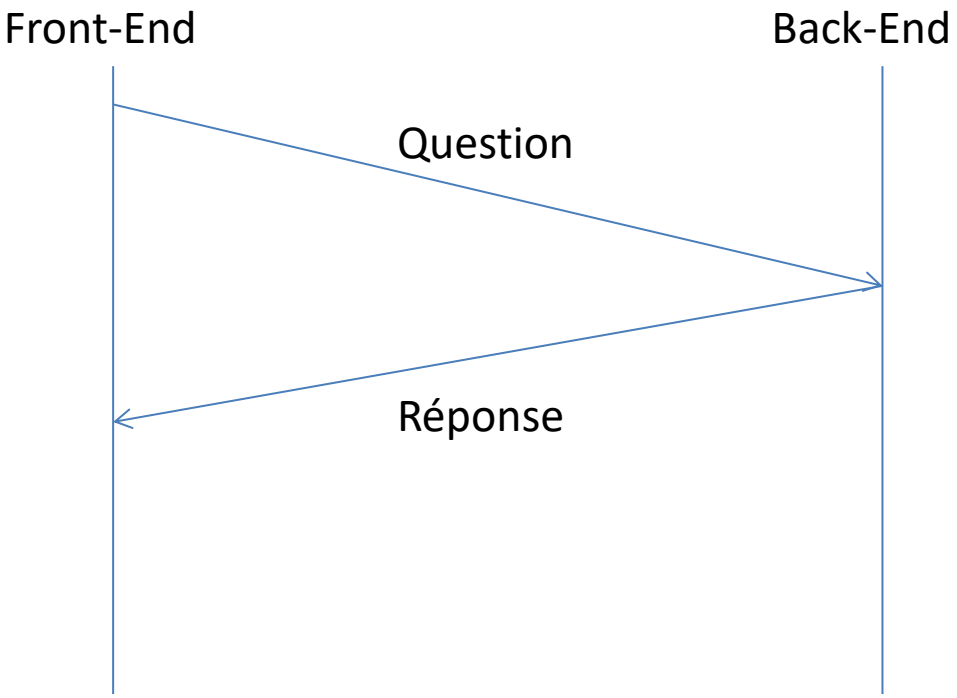
  - `HttpServletRequest : contient tout ce qui concerne la requête (son chemin, l'adresse de l'émetteur, les cookies, les données transmises, etc...)`
  - `HttpServletResponse : possède les méthodes pour préparer la réponse à renvoyer au front-end.`

HTTPServlet

- En général on ne redéfinit pas directement service :
  - `doGet(HttpServletRequest i,HttpServletResponse o)`
  - `doPost(HttpServletRequest i,HttpServletResponse o)`
  - `doPut(HttpServletRequest i,HttpServletResponse o)`
  - `doDelete(HttpServletRequest i,HttpServletResponse o)`
  - `...`

Détours : le protocole HTTP

- Le dialogue est toujours :



Front-End -> Back-End

- La question est composée de :
  - Une URL
  - Une méthode : Get/Post/Put/Delete/...
  - Une adresse d'origine
  - Des cookies
  - De données à envoyer
  - D'un mimetype pour les données à envoyer
    - http://www.sitepoint.com/web-foundations/mime-types-complete-list/
  - Des en-têtes, p.ex. le user agent=browser
  - ...

Back-End -> Front-End

- La réponse est composée de :
  - Un code d'état : 200 = ok, 404 = manquant etc
    - http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
  - Des données
  - Un mimetype pour ces données
  - Des modification de cookies (ajout/suppression/modification)
  - Des en-têtes, p.ex. des informations sur la manière de cacher les entrées
  - …

Chrome : F12 / Network

Les méthodes des requêtes

- GET : obtenir une ressource.
  - L'URL entrée dans le navigateur effectue un GET sur cette adresse.
    - Les paramètres sont encodés dans cet URL : ?param1=valeur1&param2=valeur2
  - Bookmarkable.
  - En général peut être mis en cache.
- POST : soumettre une ressource.
  - Soumission d'un formulaire : les données sont envoyées en POST.
    - Les paramètres sont encodés dans le 'payload' qui est distinct de l'URL.
  - Non bookmarkable.
  - En général ne peut pas être mis en cache : la réponse à la soumission d'un formulaire dépend de ce formulaire et change donc à chaque fois.

Les autres méthodes...

- DELETE, PUT
  - utilisés dans les API Webs, dans le standard REST
  - peu voire pas du tout utilisés à partir d'un navigateur
- Dans le cadre de ce cours, on se concentrera sur GET et POST
  - GET : quand on veut pouvoir bookmarker la requête.
  - POST : quand on **ne** veut **pas** pouvoir bookmarker la requête.

De retour aux servlets

```
public class DemoServlet extends HttpServlet {
	@Override
	protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
		...
	}

	@Override
	protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
		...
	}

}
```

HttpServletRequest

- Tout ce qui concerne la requête.
- http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html
- Quelques méthodes :
  - getPathInfo() : renvoie la partie d'URL supplémentaire à ce qui est configuré pour le servlet.
  - getParameter(String) : obtient un des paramètres de la requête, çad des données transmises.
  - getCookies() : retourne les cookies.

HTTPServletResponse

- Tout ce qui concerne la réponse
- http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html
- setStatus(int): définit le statut de la réponse
- setContentLength(int): définit la longueur en bytes de la réponse
- setCharacterEncoding(String): définit l'encodage de la réponse ("utf-8")
- setContentType(String): définit le mimetype de la réponse
- getOutputStream(): retourne l'OutputStream permettant d'écrire la réponse

Exceptions dans doGet ou doPost

- Un fichier non trouvé = erreur 404

```
resp.setStatus(404);
resp.setContentType("text/html");
String msg="<html><body>Fichier non
trouvé</body></html>";
byte[] msgBytes=msg.getBytes("UTF-8");
resp.setContentLength(msgBytes.length);
resp.setCharacterEncoding("utf-8");
resp.getOutputStream().write(msgBytes);
```

- Ceci est un traitement normal du côté Java, pas une exception jetée !

Exception en Java

- Si une exception s'échappe :
  - C'est Jetty lui-même qui la gérera.
    - Redirection vers une page d'erreur.
    - Ou envoi d'un message standard.
- En général on évitera de laisser s'échapper les exceptions.

Gestion des exceptions

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
                    throws ServletException, IOException {
          try {
                    ....
          } catch (Exception e) {
                    e.printStackTrace(); // pour logger l'erreur
                    resp.setStatus(500); // erreur au niveau du serveur
                    resp.setContentType("text/html");
                    byte[] msgBytes=e.getMessage().getBytes("UTF-8");
                    resp.setContentLength(msgBytes.length);
                    resp.setCharacterEncoding("utf-8");
                    resp.getOutputStream().write(msgBytes);
          }
}
```