Développement d'applications modulaires en Java

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr

Plan de l'ECUE

1. (Rappels sur le) Développement d'applications Web avec Java
2. Modulariser les applications Java avec Spring
3. Bien structurer une application Web avec Spring MVC
4. Auto-configurer une application Web avec Spring Boot
5. Sécuriser une application Web avec Spring Security
6. Gérer des données massives avec Apache Kafka et Spring
7. Tester une application Web Spring
8. Écrire des applications Web (API) réactives avec Spring WebFlux

Plan de l'ECUE

1. (Rappels sur le) Développement d'applications Web avec Java
2. Modulariser les applications Java avec Spring
3. Bien structurer une application Web avec Spring MVC
4. Auto-configurer une application Web avec Spring Boot
5. Sécuriser une application Web avec Spring Security
6. Gérer des données massives avec Apache Kafka et Spring
7. Tester une application Web Spring
8. Écrire des applications Web (API) réactives avec Spring WebFlux

Plan du cours

Qu'est-ce qu'une application Web ?

- Un ensemble de ressources (pages et services Web) déployé sur un serveur accessible via le protocole HTTP
- Des requêtes HTTP parviennent au serveur Web et sont redirigées vers l'application :
  `http://someserver/myapp/endpoint`
- Le serveur renvoie des réponses HTTP avec du :
  - contenu statique : pages HTML, images, scripts clients, ...
  - contenu généré dynamiquement par des programmes serveurs

Déploiement d'une application Web Java

- Déploiement fait simplement en copiant le contenu de l'application dans un dossier particulier du serveur Web
  - Dossier `webapps/` du répertoire d'installation de Tomcat

Serveur Web Tomcat

» Télécharger Tomcat v10 : `https://tomcat.apache.org/download-10.cgi`
» Décompresser l'archive
» Démarrer le serveur en utilisant le script `catalina` qui se trouve dans le dossier `bin/` de l'archive décompressée (`catalina.sh start`)
» Page d'accueil du serveur Web accessible sur : `http://localhost:8080`

Structure d'une application Web Java

- Une structure standard pour le contenu d'une app : ouvrez le sous-dossier `examples/` de `webapps/` dans Tomcat
- A la racine du dossier d'une application, des fichiers HTML, images, des scripts client ou serveur, ...
- On trouve aussi un dossier `WEB-INF/`, qui comporte :
  - configuration de l'application dans `web.xml` (voir plus loin)
  - classes qui composent l'application
  - bibliothèques utilisées par l'application
- L'application peut exister sous la forme d'une archive ZIP (fichier .war : *Web ARchive*), avec la même structure que ci-dessus

Administrer le déploiement d'applications

- Interface Web : `http://localhost:8080/manager/html`
- Cela nécessite une authentification
- Ajouter d'abord un utilisateur gestionnaire (*manager user*) :
  - Éditer le fichier `tomcat-users.xml` du dossier `conf/` de l'install Tomcat
  - Ajouter la ligne suivante :

```
<user username="bob" password="LoveAlice4Ever;"
roles="manager-gui"/>
```

- On peut démarrer, arrêter, recharger-sans-arrêter, ... les applications existantes
- On peut déployer une nouvelle app (un WAR, par exemple)

Plan du cours

Plan du cours

Notions de base

- Programmes serveurs : *Servlets* (avec ça, on peut tout faire) et *Jakarta Server Pages* (ex-`JavaServer Pages`) – JSP
- Il existe de nombreux frameworks construits au dessus de ces notions
- Exemples de frameworks : Struts & Spring (vu dans ce cours)

Interface implémentée par les servlets

- Interface `javax.servlet.Servlet` (dans les versions récentes de l'API, 4.0+, c'est `jakarta.servlet.Servlet` – remplacer partout `javax.servlet` par `jakarta.servlet`)

Method	Description
public void init(ServletConfig config)	initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once.
public void service(ServletRequest request,ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public void destroy()	is invoked only once and indicates that servlet is being destroyed.
public ServletConfig getServletConfig()	returns the object of ServletConfig.
public String getServletInfo()	returns information about servlet such as writer, copyright, version etc.

- Interface indépendante d'un protocole (HTTP, FTP, ...)

Servlets

- L'interface précédente peut être implémentée directement, mais c'est long à faire
- La bibliothèque qui vient avec le serveur Web (dossier `lib/`) fournit des classes "*Helper*"
- Parmi ces classes, nous retrouvons `GenericServlet` (une classe abstraite) et `HttpServlet`

Classe HttpServlet

```
public class HttpServlet extends GenericServlet {
  public void service(ServletRequest req, ServletResponse resp) {
   service((HttpServletRequest)req, (HttpServletResponse)resp);
  }

  public void service(HttpServletRequest req, HttpServletResponse resp) {
    String verb = req.getMethod();
    if (verb.equals("GET")) doGet(req, resp);
    else if (verb.equals("POST")) doPost(req, resp);
    else ...
  }

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
  {...}

  public void doPost(HttpServletRequest req, HttpServletResponse resp)
  {...}
  ...
}
```

Exemple de servlet

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

public class OurServlet extends HttpServlet {
  public init() {
    // Perform any instance initialisation
  }

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

    // Process the request and generate the response
  }
  ...
}
```

- On redéfinit ces méthodes dans une app Web : `doGet()`, `doPost()`, `doPut()`, ...

Exemple de servlet (*Servlet Mapping*)

- Comment le serveur peut identifier quelle servlet répond à une requête ?

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

@WebServlet("/home")
public class OurServlet extends HttpServlet {
  public init() {
    // Perform any instance initialisation
  }

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

    // Process the request and generate the response
  }
  ...
}
```

- Importer l'annotation `@WebServlet` de `javax.servlet.annotation` (`jakarta.servlet...`)

A vos claviers

- Écrire une servlet qui répond à une requête HTTP Get avec le message "*Hello Java Web World !*"
- D'abord, faire cela à la main (étapes élémentaires) :
  1. copier la classe qui se trouve sur le dépôt Git et la compiler, en ajoutant dans le CLASSPATH le jar `servlet-api.jar` qui se trouve dans le dossier `lib/` de Tomcat,
  2. créer un dossier simpleapp/ dans webapps/, créer dedans un dossier WEB-INF/classes/ et mettre le .class dans ce dossier,
  3. démarrer Tomcat, en allant dans le dossier bin/ de Tomcat et en tapant la commande ./catalina.sh start (changer les droits sur le fichier .sh si celui-ci n'est pas exécutable)
  4. Dans votre navigateur, aller à : `http://localhost:8080/simpleapp/home`
- Ensuite, utiliser IntelliJ IDEA pour éditer le code et Gradle pour construire et déployer le projet (voir prochaines diapos)

A vos claviers -suite-

- Installer Gradle (dans les systèmes Unix, vous pouvez installer SDKMAN ! d'abord : `https://sdkman.io/`)
- Ceux qui ne connaissent pas Gradle, c'est un système de build, comme Maven –même structure pour les projets, mais pour les scripts de build on utilise un DSL (*Domain Specific Language*) Groovy ou Kotlin et non XML
- Tester votre installation de Gradle sur un terminal : `gradle -v`
- Initialiser un projet Java simple : `gradle init`
  - Choisir comme type de projet "`2: application`"
  - Choisir comme langage d'implémentation "`3: Java`"
  - Choisir comme DSL pour le script de build : "`2: Kotlin`"
  - Choisir l'option par défaut pour le framework de Test (`JUnit 4`)
  - Donner un nom à votre projet : SimpleApp et donner un nom de package : `fr.polytech.mtp.ig5.iaw2.c1`

A vos claviers -suite-

- Que s'est-il passé ? inspecter le dossier courant et ouvrez le fichier `build.gradle.kts`
- Expliquer les lignes de ce fichier
- Où se trouve la classe principale de l'application ?
- Lister les tâches possibles à faire dans ce projet : `gradle tasks`
- Exécuter votre app : `gradle run`
- Ceci déclenchera plusieurs tâches. Identifier ces tâches

* Utiliser gradle run -i

A vos claviers – transformer le projet en app Web

- Éditer le script de build, `build.gradle.kts` pour y mettre le contenu suivant :

```
plugins {
  id "war"
  id "org.gretty" version "3.0.6" // ou version plus
                // recente pour l'API 4.0+ des servlets
}
repositories {
  mavenCentral()
}
dependencies {
  providedCompile("javax.servlet:javax.servlet-api
    :4.0.1")
  testImplementation("junit:junit:4.12")
}
```

Le plugin gretty permet d'installer et démarrer un serveur Web Tomcat (Fondation Apache) ou Jetty (Fondation Eclipse)

A vos claviers – transformer le projet en app Web

- Ajouter un dossier `webapp/` sous `src/main`
- Ajouter dans ce dossier une page Web dans laquelle est affichée la phrase "Hello World !"
- Changer de wrapper Gradle (pour utiliser une version standard de Gradle) : *gradle wrapper* —*gradle* — *version* = 6.6.1
- Lancer le build : `gradlew build`
- Démarrer le serveur Web Jetty et y déployer votre app : `gradlew jettyStart`
- Aller dans votre navigateur Web à la page : `http://localhost:8080/SimpleApp` où `SimpleApp` est le nom de votre projet

A vos claviers – transformer le projet en app Web

- Ajouter dans la page Web un formulaire pour saisir le nom d'un utilisateur
- Quand le formulaire est soumis, une servlet renvoie le message "Hello" suivi du nom de l'utilisateur
  - L'attribut `action` du formulaire doit avoir comme valeur "`home`"
  - La servlet est placée dans le dossier `src/main/java` (vous pouvez simplement modifier la classe `App` créée précédemment)
  - Pour récupérer la valeur d'un champ de formulaire dans la servlet : (le *name* du champ est "`name`")

    ```
    String name = request.getParameter("name");
    ```
- Tester l'application
- Arrêter le serveur ...

Servlet Mappings – routes des requêtes HTTP

1. A l'ancienne (fichier `WEB-INF/web.xml` dans `webapp/`) :

```
<web-app>
        <servlet>
                <servlet-name>Home</servlet-name>
                <servlet-class>com.mantiso.OurServlet</servlet-class>
        </servlet>

        <servlet-mapping>
                <servlet-name>Home</servlet-name>
                <url-pattern>/Bar</url-pattern>
        </servlet-mapping>

        <servlet-mapping>
                <servlet-name>Home</servlet-name>
                <url-pattern>*.abc</url-pattern>
        </servlet-mapping>
</web-app>
```

2. Avec les annotations :

```
@WebServlet(urlPatterns="hello","*.do")
```

Objet `HttpServletRequest`

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

  // Read HTTP headers
  String strHost = req.getHeader("Host");

  // Read Content-Type of request body
  String strContentType = req.getContentType();

  // Read parameters
  // e.g. http://someserver/someapp/someservlet?uid=bob
  String strName = req.getParameter("uid");

  // Access request body
  BufferedReader = req.getReader();
  ...
}
```

Objet `HttpServletResponse`

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {
  try {
    if (notLoggedOn()) {
      resp.sendRedirect("/logon");
      return;
    }
    resp.setContentType("text/xml");
    resp.setHeader("X-Custom-Header", new Date());
    PrintWriter out = resp.getWriter();
    out.write("<message>Hello world</message>");
  } catch (Exception e) {
    resp.sendError(response.SC_INTERNAL_SERVER_ERROR);
  }
}
```

Initialiser une servlet

- Pour récupérer les données de connexion à un seveur de BdD, par ex

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

@WebServlet(urlPatterns = {"/home", "*.do"},
            initParams =
            {@WebInitParam(name = "connstr",
                           value="server=...")})
public class OurServlet extends HttpServlet {
  public init() {
    connstr = getInitParameter("connstr");
  }

  ...
}
```

Possibilité de faire la même chose dans `web.xml`

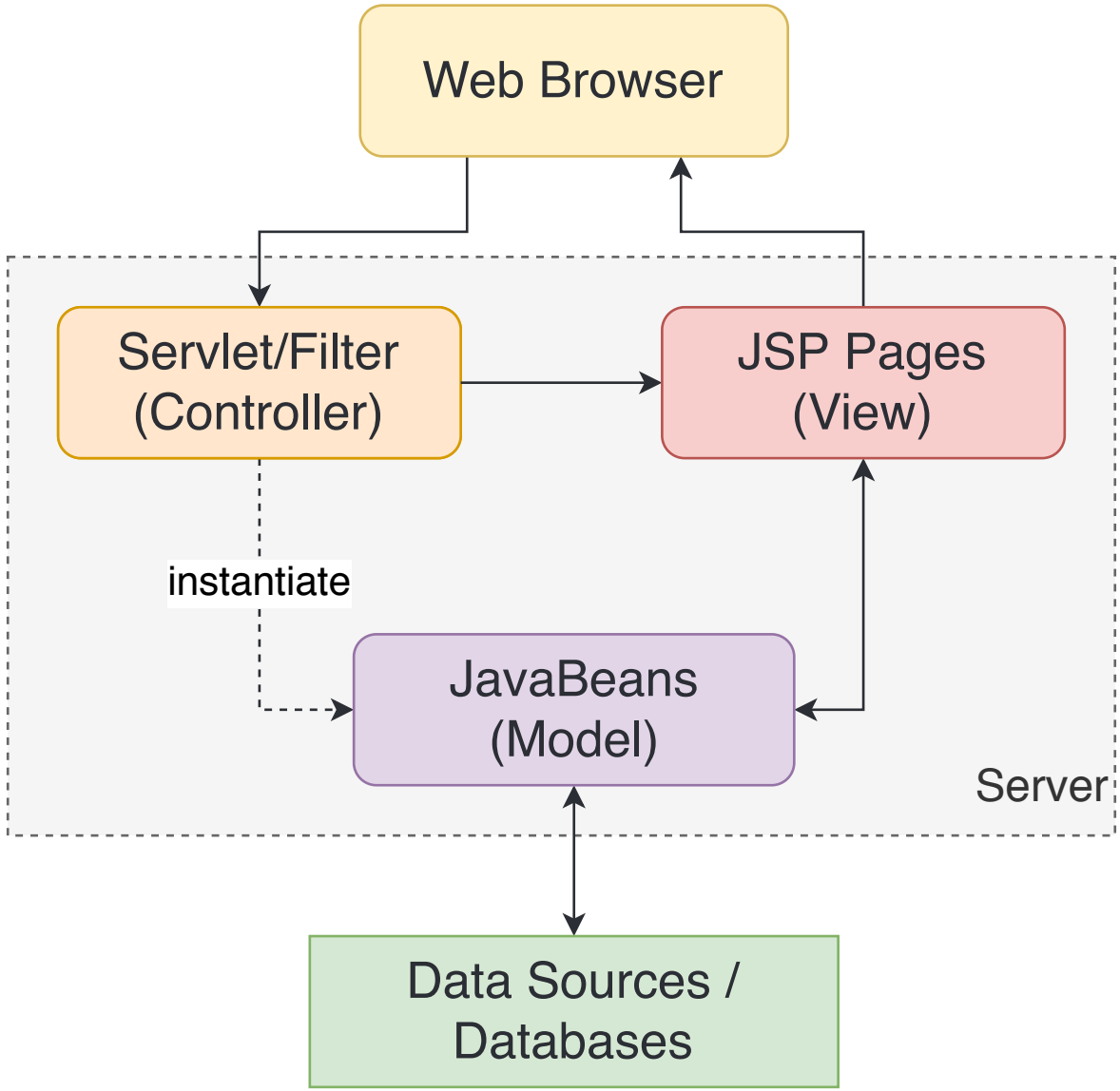
Initialiser toute l'application

```
<web-app>
    <context-param>
      <param-name>connstr</param-name>
      <param-value>server=homer;catalog=pubs;uid=sa;pwd=;</param-value>
    </context-param>
</web-app>
```

```
public class OurServlet extends HttpServlet {
  String connstr;
  public void init() {
    connstr = getServletContext().getInitParameter("connstr");
  }
}
```

Plan du cours

Architecture des applications JSP (Patron MVC)



Pages JSP

- Mélange de code HTML et de code Java (éq. Php, ASP, ...)
- Pages JSP transformées en servlets à l'exécution :
  - Code source de la servlet généré puis compilé lors de la réception de la première requête
- Scriptlet : balises `<% et %>`
  - instructions Java exécutées pour réaliser des traitements lors de la réception d'une requête HTTP et pendant la fabrication de la réponse HTTP
- Expressions : balises `<%= et %>`
  - expression dont la valeur, convertie en chaîne, est incluse dans le flot HTML produit dans la réponse HTTP

Pages JSP -suite-

- Déclarations : balises `<%! et %>`
  - déclaration de classe, de méthode, d'attribut, etc, utilisables dans les scriptlet et expressions précédentes
- Directives d'inclusion : balises `<%@ et %>`
  - directives d'inclusion de bibliothèques ou de fichiers (de code JSP)
- Commentaires : balises `<%- et -%>`

Variables pré-définies et pré-initialisées dans les scripts

- `request` (HttpServletRequest) : objet correspondant à la requête HTTP (durée de vie = celle d'une seule interaction avec un client)
- `response` (HttpServletResponse) : objet réponse HTTP
- `out` (PrintWriter) : objet utilisé pour écrire dans le flot HTML de la réponse HTTP ⟶ `out.print(...);`
- `session` (HttpSession) : objet correspondant à la session (durée de vie = celle de la session avec un client)
- `application` (ServletContext) : objet dont la durée de vie est égale à celle de l'application (initialisation de l'app)
- ...

Exemple de page JSP

```
<html>
  <head> <title>Converter</title> </head>
  <body>
    <h1><center>Converter</center></h1> <hr/>
    <p>Enter an amount to convert:</p>
    <form method="get">
      <input type="text" name="amount" size="25"> <br>
      <input type="submit" value="Submit"><input type="reset" value="Reset">
    </form>
    <% String amount = request.getParameter("amount");
    if ( amount != null && amount.length() > 0 ) {
      Double d = new Double (amount); %> <p>
      <%= amount %> dollars =
      <%= converter.dollarToYen(d.doubleValue()) %> Yen.</p><p>
      <%= amount %> Yen = <%= converter.yenToEuro(d.doubleValue()) %> Euro.
      </p>
    <% } %>
  </body>
</html>
```

A vos claviers – reprendre l'app Web

- Nous allons maintenant créer une page JSP qui affiche le message "Hello" suivi du nom de la personne
  - ce qui correspond à la vue (on garde la servlet comme contrôleur)
- Créer une page JSP (response.jsp) dans le dossier `webapp/`
- Ajouter dans le *body* de cette page la ligne suivante :

  `<h2>Hello, ${user}!</h2>`

  D'où vient `${user}` ?
- C'est un attribut qui est inséré dans la requête HTTP qui est redirigée par la servlet :

```
String name = request.getParameter("name");
if (name == null) name = "World";
request.setAttribute("user", name);
request.getRequestDispatcher("response.jsp").forward(
    request, response);
```

Gestion des erreurs dans JSP

- Dans une section de code JAVA à l'intérieur d'une page JSP, on peut signaler une exception :

```
if ( amount != null && amount.length() > 0 ) {
   ...
}
else {
  throw new ServletException("Il faut indiquer un
     montant");
}
```

- Qu'est-ce qu'on a sur le navigateur si une exception est levée ?

Exception renvoyée au navigateur Web (par Jetty)

HTTP ERROR 500 javax.servlet.ServletException: No name provided

URI:	/SimpleServer/hello
STATUS:	500
MESSAGE:	javax.servlet.ServletException: No name provided
SERVLET:	HelloServlet
CAUSED BY:	javax.servlet.ServletException: No name provided

Caused by:

```
javax.servlet.ServletException: No name provided
        at fr.polytech.ig5.iaw.App.doPost(App.java:27)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:707)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:790)
        at org.eclipse.jetty.servlet.ServletHolder$NotAsyncServlet.service(ServletHolder.java:1391)
```

Capturer l'exception et afficher une page d'erreur

- D'abord, prévoir une redirection vers une page d'erreur dans web.xml

```
<error-page>
  <location>/error.jsp</location>
</error-page>
```

- Parenthèse : on peut gérer plus finement les erreurs en ajoutant des pages d'erreurs spécifiques

```
<error-page>
  <error-code>404</error-code>
  <location>/404.html</location>
</error-page>
```

Capturer l'exception et afficher une page d'erreur

- Ensuite, créer la page `error.jsp` et dans cette page, on peut obtenir et afficher le message de l'exception

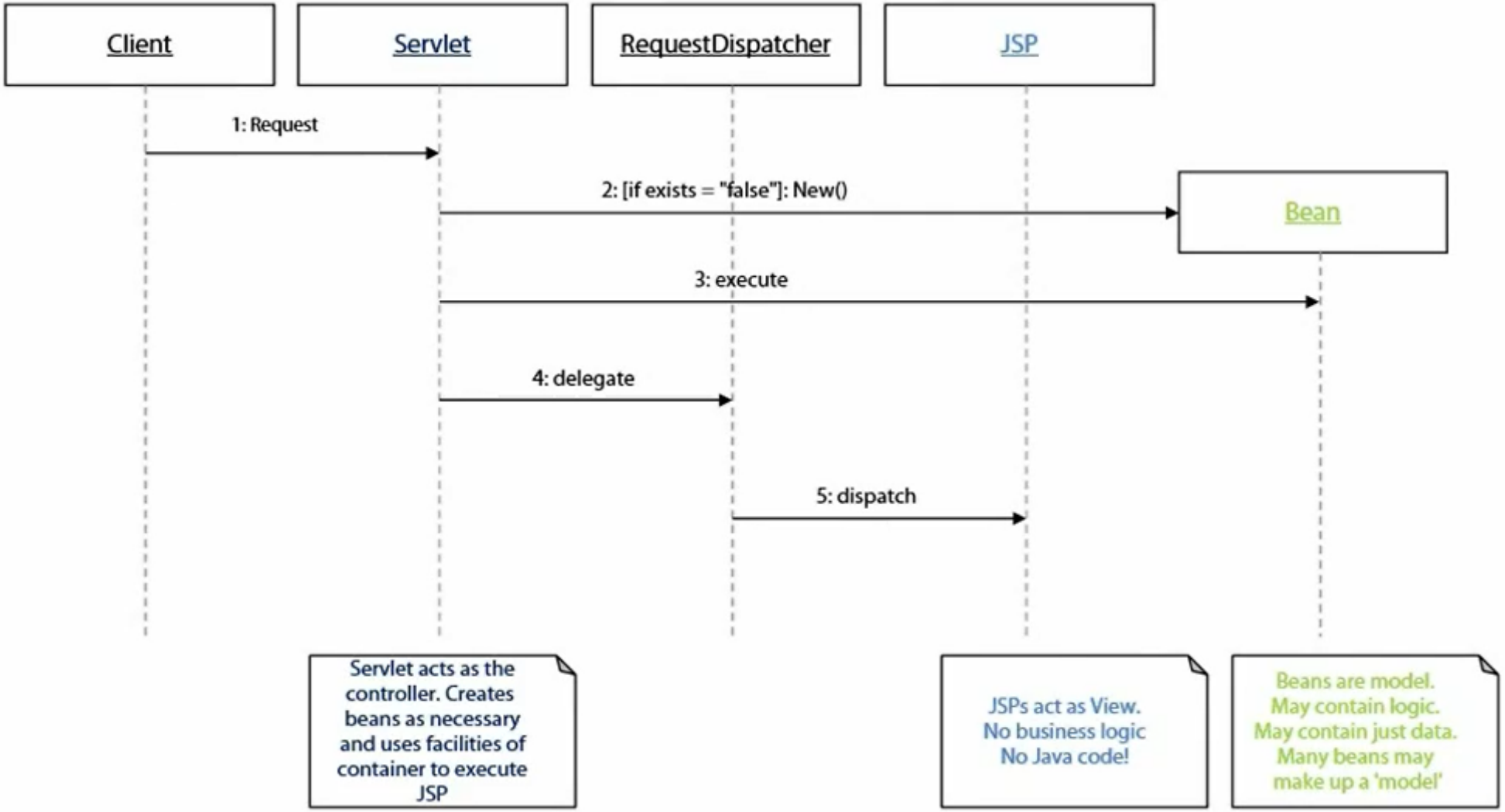
```
<%@ page contentType="text/html;charset=UTF-8"
    language="java" %>
<%@ page isErrorPage="true" %>
<html>
  <head>
    <title>Error Page</title>
  </head>
  <body>
    <h2>Error Page!</h2>
  <%= exception.getMessage() %>
  </body>
</html>
```

La directive "page" avec `isErrorPage="true"` donne accès à l'attribut `exception` (utilisé ci-dessus)

A vos claviers – reprendre l'app Web

- Ajouter une page d'erreur si jamais un nom n'est pas fourni
- Tester à nouveau l'application
- Dans une vraie application Web, nous n'utilisons pas les exceptions pour ce genre de contrôles (c'est juste pour tester la gestion des erreurs ici)

MVC avec des *Servlet Dispatcher*



MVC avec des *Servlet Dispatcher* -suite-

- Un objet `ServletDispatcher` peut être obtenu à partir du contexte de la servlet : (extrait d'un précédent exemple)

```
ServletDispatcher dispatcher = request.
    getRequestDispatcher("/response.jsp");
dispatcher.forward(request,response);
// ou bien dispatcher.include(request,response);
// qui permet de deleguer et ensuite reprendre la main
```

- Des objets (données) peuvent être inséré(e)s par la servlet (le contrôleur) avant de déléguer la requête
- Cette insertion se fait en invoquant la méthode `setAttribute("nom",valeur);` avant la délégation
- L'insertion de ces objets/données peut avoir plusieurs portées

Portées des objets/données dans une app Web

- **Application** (durée de vie de toute l'application) :

```
getServletContext().setAttribute("db_server_url",url);
```

- **Session** (une session d'un utilisateur sur son navigateur – plusieurs requêtes) :

```
getServletContext().getSession().setAttribute("user",
    user);
```

  Typiquement, c'est la portée qu'on utilise pour un panier dans une app de e-commerce

- **Requête** (durée de vie de la requête HTTP) :

```
request.setAttribute("locale",locale);
```

Une dernière chose : masquer les pages JSP

- Essayer d'accéder à la page `http://localhost:8080/SimpleServer/error.jsp`
  Surprise!!!
- Déplacer les .jsp (avec refactoring) dans le dossier `WEB-INF/` (son contenu n'est pas rendu accessible par le serveur Web)
- S'assurer que toutes les références aux fichiers JSP sont correctes. Ex :

```
ServletDispatcher dispatcher = request.
    getRequestDispatcher("/WEB-INF/response.jsp");
```

- Les pages JSP font partie de la vue (elles ne doivent pas être accessibles directement – voir schéma du patron MVC)

Expression Language

- Ce langage permet d'écrire des expressions qui permettent de réduire la quantité de code Java
- Syntaxe : une expression placée entre `${` et `}`
- Exemple : `<h2> Bienvenue ${user.name} </h2>`
  ça récupère du contexte de la servlet (requête, session ou app) un objet *JavaBean* user, puis exécute `user.getName()` :

```
<% User user = (User)request.getAttribute("user"); %>
<h2>Bienvenue <%= user.getName() %> </h2>
```

  C'est plus simple, non ?
- On peut scripter du CSS également :
  `<div class=${app.formCssClass.name}></div>`

Utiliser des JavaBeans et plein d'opérateurs dans JSP EL

- Syntaxe JavaScript : `user.name` ou `user["name"]`
- Opérateurs mathématiques : +, -, *, /, %
- Opérateurs de comparaison : ==, !=, <, >, <=, >=
- Opérateurs logiques : &&, ||, !
- Exemple : `<h3>${user.name == "Eva"}</h3>`
- Opérateur vide : permet de tester si une expression est égale à true, ou à une valeur différente de 0 ou null (selon son type), ça retourne la valeur true, sinon ça retourne false
  `${!empty user.name}` (éq. `user.name != null`)

Une dernière chose sur JSP : vous savez ce qu'est JSTL ? Googler ce mot pour vous faire une idée

Plan du cours

Plan du cours

Annotations

- Les annotations remplacent parfois le code XML dans web.xml
- Seules les classes dans `WEB-INF/classes/` et les JAR dans `WEB-INF/lib/` sont scannés
- Nous avons déjà utilisé l'annotation `@WebServlet` pour remplacer une déclaration de servlet dans `web.xml`
- Nous pouvons déclarer des écouteurs en utilisant les annotations

Qu'est-ce qu'un écouteur ?

- Une section de code qui écoute un type d'événement particulier
- Utilisé souvent pour la journalisation – *logging* ou pour déclencher des traitements asynchrones
- Un événement est déclenché par le serveur à des moments précis de la vie d'une app Web :
  - Événements d'applications au démarrage/arrêt de l'app
  - Événements de session au démarrage/arrêt d'une session
  - Événements de requête au démarrage/arrêt d'une requête
  - Événements d'attribut lors de l'ajout/suppression d'un attribut (au niveau App, Session ou Requête)

Comment définir un écouteur ?

- Écrire une classe annotée par `WebListener`
- Implémenter la bonne interface `Listener` :
  `ServletContextListener` (app),
  `ServletContextAttributeListener` (attribut/app),
  `HttpSessionActivationListener` (session),
  `HttpSessionAttributeListener` (attribut/session),
  `ServletRequestListener` (requête),
  `ServletRequestAttributeListener` (requête/attribut)
- Dans la classe “écouteur d’évènement”, les méthodes callback à définir reçoivent en paramètre un objet “évènement”

Exemple d'écouteur d'évènements d'application

```
public class WhitePagesServletListener
             implements javax.servlet.ServletContextListener
{
    java.util.Hashtable whitePages;

    public void contextDestroyed(ServletContextEvent sce)
    {
    }

    public void contextInitialized(ServletContextEvent sce )
    {
        whitePages = new java.util.Hashtable();
        sce.getServletContext().setAttribute("addrbook", whitePages);
    }
}
```

- Il est possible d'étendre la classe Adapter "`EventListener`" pour ne pas devoir implémenter toutes les méthodes abstraites

Un écouteur d'événements liés aux attributs d'une session

- D'abord, la durée d'une session peut être paramétrée. Dans `web.xml` :

```
<session-config>
  <session-timeout>1</session-timeout>
</session-config>
```

  La durée est exprimée en minutes ici

- L'objet événement de type `HttpSessionEvent` fournit une méthode `getSession()` qui permet d'obtenir le contexte de la servlet (méthode `getServletContext()`). Ce dernier objet permet de faire des logs :

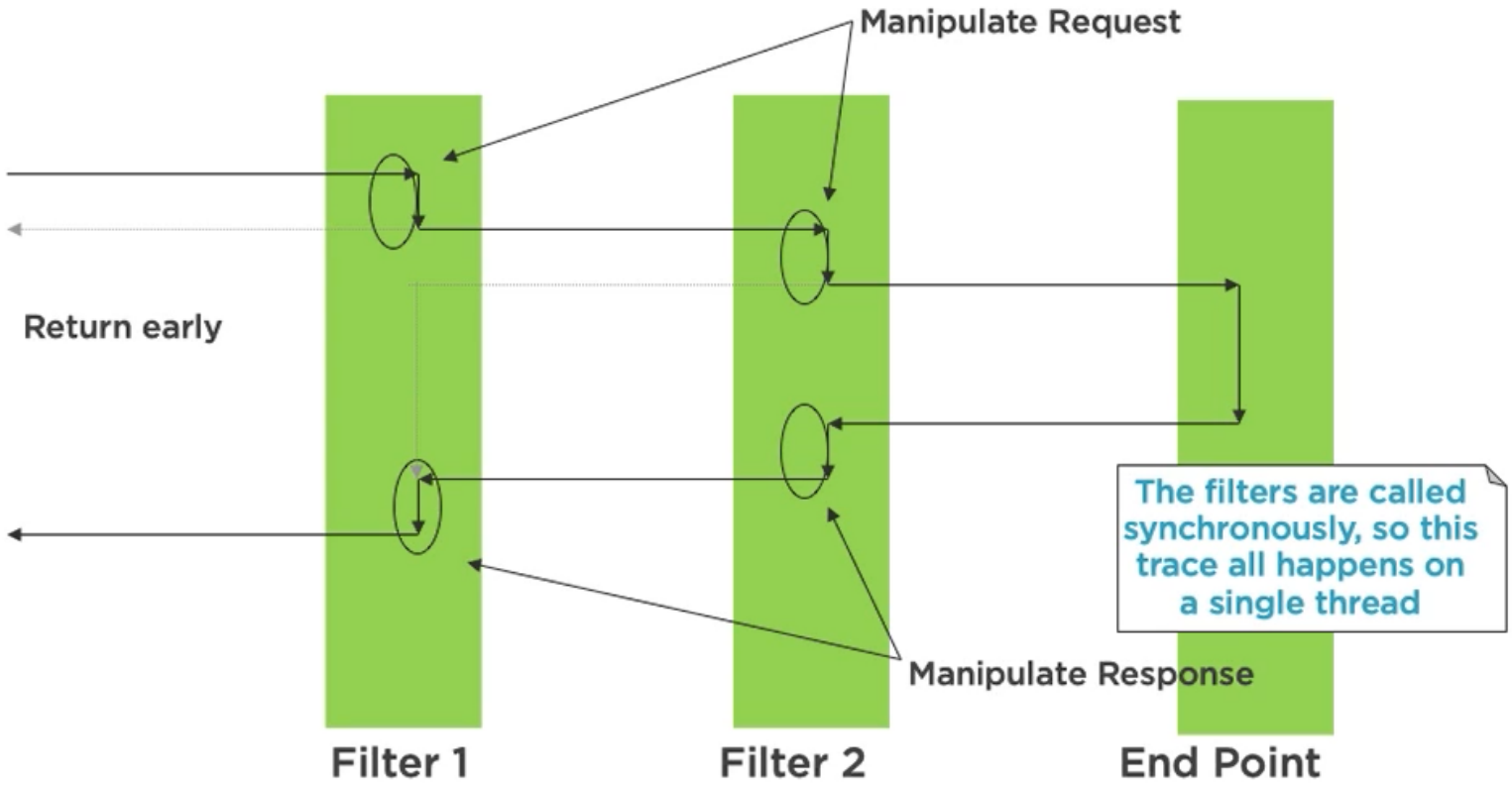
```
event.getSession().getServletContext().log("Session
utilisateur démarrée);
```

Plan du cours

Qu'est-ce qu'un filtre

- Une section de code qui intercepte une requête et qui effectue dessus un traitement supplémentaire :
  - Ils sont exécutés avant/après l'exécution d'une requête
  - La requête peut être pour une page HTML, page JSP, une servlet, ...
  - La requête et la réponse peuvent être modifiées
  - Les filtres peuvent être exécutées en chaîne
  - Les filtres peuvent intercepter la requête originale des `forward` et `include` (voir `ServletDispatcher`)
  - Les filtres sont utilisés pour gérer des sessions, la journalisation (*logging*), la sécurité, ...

Chaîne de filtres (synchrones – appels bloquants)



- Le filtre 1 peut être un filtre de sécurité, qui ne laisse pas passer la requête si utilisateur non-authentifié/autorisé
- Le filtre 2 peut gérer le *logging* par exemple

Comment définir un filtre ?

- Une classe qui implémente l'interface `javax.servlet.Filter` :

```
// Called once at start
public void init(FilterConfig config);
// Called once at end
public void destroy(FilterConfig config);
// where the work is done
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain);
```

- Appeler `chain.doFilter(...)` pour continuer l'exécution de la chaine de filtres

Configuration d'un filtre

```
<filter>
    <filter-name>Logging Filter</filter-name>
    <filter-class>com.mantiso.filter.LoggingFilter</filter-class>
    <init-param>
        <param-name>jbdc-url</param-name>
        <param-value>jdbc:mssql:localhost:1024</param-value>
    </init-param>
</filter>

<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/*</url-pattern>
  <!--    <servlet-name>SessionTest</servlet-name> -->
</filter-mapping>
```

- Les filtres peuvent être associés à une ressource ou un URL
- Possible de le faire avec des annotations : `@WebFilter(urlPatterns="*.do")` et `@WebInitParam`

Structure-type d'un filtre

```
public void doFilter(ServletRequest req,
                     ServletResponse resp,
                     FilterChain chain)
      throws java.io.IOException, ServletException
{
  HttpServletRequest request = (HttpServletRequest)req;
  HttpServletResponse response = (HttpServletResponse)resp;

  // pre process request
  // optionally 'wrap' request and response

  chain.doFilter(req, response); // also optional

  // post process response
}
```

ServletRequest and ServletResponse

Filter could simply return or use a RequestDispatcher

Wrappers des requêtes et réponses HTTP

- Dans certains cas, on est amené à écrire des wrappers pour les requêtes ou les réponses HTTP
- Ce sont des classes, qui étendent des classes Adapter (`HttpServletRequestWrapper` ou `HttpServletResponseWrapper`) pour :
  - utiliser une journalisation log4j de certaines informations dans les requêtes
  - transformer ou compresser les réponses
  - ...

Plan du cours

Pourquoi on recherche parfois l'asynchronisme ?

- Lorsqu'on a des traitements au backend qui peuvent être lents (invocation d'un service externe lent)
- Lorsque l'on fait des entrées/sorties dans le backend qui peuvent bloquer (la tendance maintenant dans la plupart des API est au NIO – *Non-blocking IO*, par défaut)
- Pour permettre au serveur Web de réutiliser le thread d'accueil des requêtes du client (en réalité, il y en a plusieurs, des *HTTP worker threads*. Tomcat peut en créer jusqu'à 200, valeur par défaut)
- Pour permettre le "*Server Push*" : le serveur peut notifier les clients, qui font du "*long polling*", d'informations qui sont mises à jour régulièrement (réseaux sociaux, ...)

L'objectif ultime est rendre une application Web **scalable**

Mode d'emploi des servlets asynchrones

1. Démarrer un contexte de servlet asynchrone en utilisant l'objet représentant la requête HTTP
2. Utiliser ce contexte pour accéder à la requête et retourner la réponse, mais en effectuant cela dans un thread à part, afin de libérer le thread worker (HTTP) du serveur
3. Éventuellement, ajouter des écouteurs au contexte pour gérer les événements (timeout, ...)
4. Indiquer la fin du contexte asynchrone pour déclencher le renvoi de la réponse au client

On peut définir des filtres asynchrones également (non présentés ici)

Écrire une servlet asynchrone

- D'abord annoter la servlet comme asynchrone :

```
@WebServlet(urlPatterns="/home", asyncSupported=true)
public class FirstAsyncServlet extends HttpServlet {
}
```

- Démarrer le contexte de servlet asynchrone :

```
@Override
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
  final AsyncContext ctx = req.startAsync();
  // ...
}
```

L'invocation de `startAsync()` indique au serveur que la réponse ne doit pas être retournée au client à la fin du `doGet(...)`

Écrire une servlet asynchrone -suite-

- Démarrer et arrêter le traitement asynchrone dans un autre thread :

```
ctx.start(() -> {
  try { // ...
    ctx.getResponse().getWriter().write(...);
  }
  catch(IOException e) {
    log("Un probleme est survenu lors du traitement
    asynchrone",e);
  }
  ctx.complete();
});
```

- Autre possibilité : déléguer le traitement à une autre servlet

```
final AsyncContext ctx = req.startAsync();
ctx.dispatch("/uneUrl");
```

Fin du traitement asynchrone

- Possibilité de définir un timeout pour une servlet asynchrone
- Fixer la durée du timeout :

```
ctx.setTimeout(5 * 1000); // temps en millisecondes
```

- A la fin, indiquer que le traitement de la requête asynchrone est terminé : `ctx.complete();`

Ecouteurs d’événements pour une servlet asynchrone

```
ctx.addListener(new AsyncListener() {
    public void onComplete(AsyncEvent event) {
        log( msg: "FirstAsyncServlet onComplete called, thread id:  " + Thread.currentThread().getId());
    }
    public void onTimeout(AsyncEvent event) {
        log( msg: "FirstAsyncServlet onTimeout called, thread id:  " + Thread.currentThread().getId());
    }
    public void onError(AsyncEvent event) {
        log( msg: "FirstAsyncServlet onError called , thread id: " + Thread.currentThread().getId());
    }
    public void onStartAsync(AsyncEvent event) {
        log( msg: "FirstAsyncServlet onStartAsync called, thread id:  " + Thread.currentThread().getId());
    }
});
```

A vos claviers

- Faire le sujet de TP sur le dépôt Git, après avoir visionné les derniers slides

Plan du cours

Présence de Java dans le back-end de grandes app Web (Source : Wikipedia, Oct. 2021)

Back-end (Server-side) table in most popular websites

Websites	C#	C	C++	D	Erlang	Go	Hack	Haskell	Java	JavaScript	Perl	PHP	Python	Ruby	Scala	XHP
Google	No	Yes	Yes	No	No	Yes	No	No	Yes	Yes	No	No	Yes	No	No	No
YouTube	No	Yes	Yes	No	No	Yes	No	No	Yes	No	No	No	Yes	No	No	No
Facebook	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes
Yahoo	No	Yes	Yes	No	No	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Amazon	No	No	Yes	No	No	No	No	No	Yes	No	Yes	No	No	No	No	No
Wikipedia	No	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No
Twitter	No	No	Yes	No	No	No	No	No	Yes	No	No	No	No	Yes	Yes	No
Bing	Yes	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No
eBay	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	Yes	No
MSN	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
LinkedIn	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	Yes	No
Pinterest	No	No	No	No	Yes	No	No	No	No	No	No	No	Yes	No	No	No
WordPress.com	No	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No

Présent dans le plus grand nombre de ces grandes app Web (+ Netflix, ...)

Wrap-up

- Programmation de back-end (et un peu de Front) d'applications Web en Java
- Back-end structuré, qui profite des forces de la prog par objets et des design patterns associés
- Programmation avec un langage à typage statique (détecter certaines erreurs, de typage notamment, au plus tôt)
- Mécanisme déclaratif (à base d'annotations) pour la définition de filtres, d'écouteurs d'événements, de servlets asynchrones, ...
- Scalabilité d'une appli Web grâce aux servlets asynchrones
- Prochains cours : framework qui exploite toutes les notions de ce cours et rend leur utilisation plus simple

Vous pouvez envoyer vos questions
ou commentaires par mail