Développement Web : dynamique coté client avec AJAX et WebSocket

Master TIIL & ILIADE 1ère année

Eric Cariou

Université de Bretagne Occidentale
UFR Sciences et Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

Génération dynamique de contenu

- Dynamique coté serveur
  - Une URL est associée à une Servlet ou page JSP exécutée par le serveur HTTP
  - Le navigateur Web fait une requête à cette URL
  - La Servlet/JSP génère dynamiquement du code HTML renvoyé au client qui affiche une nouvelle page avec ce contenu
- Dynamique coté client
  - Une requête est faite par le client sur le serveur
    - Le contenu renvoyé modifie la page courante sans en recharger une nouvelle
  - AJAX : communication unidirectionnelle
    - Le client fait une requête sur le serveur pour récupérer du contenu
  - WebSocket : communication bidirectionnelle
    - Le serveur peut envoyer aussi du contenu au client sans que celui-ci ne fasse de requête

AJAX

- Asynchronous Javascript And XML
  - Requêtes coté client en Javascript à une URL pour récupérer du contenu
    - Qui n'est pas obligatoirement du XML : texte brut, JSON …
  - Asynchrone
    - Par défaut et de préférence pour ne pas bloquer la page si le serveur ne répond pas
- Exemples d'usage de chargement coté client
  - Affichage d'une longue série de donnée (ex : Twitter)
    - Charge les prochaines données à la demande quand le client arrive en bas de la page
    - Évite de tout charger en avance
  - Moteur de recherche (ex : Google)
    - On propose des complétions des mots-clés cherchés

Exemple : complétion

- Serveur gère une liste de noms de personnes
- Client affiche une page HTML simple
  - Une zone de texte où l'utilisateur entre un nom de personne
  - Au fur et à mesure, le serveur renvoie les noms qui commencent par ce qui est entré par l'utilisateur
  - Les noms s'affichent sous la zone de texte
  - En cliquant sur un nom, remplit la zone de texte avec ce nom
- Coté serveur
  - On utilise une Servlet Java pour traiter la requête
  - URL sous Tomcat local
    - http://localhost:8084/Test/completion?debut=...
    - completion : nom de l'URL de la Servlet
    - Paramètre debut : début du nom de la personne cherchée

Servlet Completion

- Partie métier
  - Les noms et la recherche de noms

```
// la liste des noms de personnes
private final String noms[] = {
"Roger", "Robert", "Raymonde", "Rachid", "Edouard", "Edmond",
"Elodie", "Eric", "Tatiana", "Théodore", "Laurence" };

// retourne la liste des noms commençant par debut
private ArrayList<String> sousNoms(String debut) {
     ArrayList<String> liste = new ArrayList<>();
     for (String nom : noms) {
         if (nom.startsWith(debut))
              liste.add(nom);
     }
     return liste;
 }
```


Servlet Completion

- Partie HTTP

```
@WebServlet(urlPatterns = {"/completion"})
public class Completion extends HttpServlet {

    // les méthodes doGet et doPost appellent la méthode processRequest
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                  throws ServletException, IOException {
       // récupère le flux de sortie dans un bloc try-with-ressources
       try (PrintWriter out = response.getWriter()) {
            // récupère la valeur du paramètre debut
            String debut = request.getParameter("debut");
            // positionne le contenu retourné comme du XML sans être placé
            // dans le cache car on doit recalculer la valeur à chaque appel
            response.setContentType("text/xml");
            response.setHeader("Cache-Control", "no-cache");
            // construit le XML
            String resultat = "<liste>";
            if ((debut != null) && (!debut.equals(""))) {
                for(String nom : this.sousNoms(debut))
                    resultat +="<nom>"+nom+"</nom>";
            }
            resultat += "</liste>";
            // retourne le XML
            out.write(resultat);
       }
    }
```

Servlet Completion

- Exemple d'appel de la Servlet
  - Si on accède à l'URL
    - http://localhost:8084/Test/completion?debut=Ro
  - Retourne le XML
    - <liste>
      <nom>Roger</nom>
      <nom>Robert</nom>
      </liste>

Page HTML

- La page HTML définit
  - Une zone de texte qui à chaque touche tapée appelle la fonction JS completer()
    - Cette fonction fera l'appel AJAX sur la Servlet
  - Une balise div (identifiant « noms ») qui sera mise à jour par les valeurs retournées par l'appel AJAX
    - Vide par défaut
- En tapant « Ro » dans la zone de texte, on aura

Page HTML

- Contenu de la page HTML testajax.html

```
<html>
  <head>
    <title>Completion en AJAX</title>
    <script type="text/javascript">
      . . .
    </script>
  </head>

  <body>
    <table>
      <tr>
        <td>Nom :</td>
        <td nowrap>
          <input type="text" id="entreeNom" name="entreeNom" size="30" onkeyup="completer();">
        </td>
      </tr>
      <tr>
        <td></td>
        <td>
          <div id="noms"></div>
        </td>
      </tr>
    </table>
  </body>
</html>
```


XMLHttpRequest

- Objet permettant de faire des requêtes HTTP
  - Pas forcément avec du XML malgré son nom
- A l'appel, on précise
  - L'URL de la requête HTTP
  - Les paramètres
    - Directement dans l'URL ou en les précisant dans l'objet requête
  - La fonction qui va traiter les retours de la requête
- La fonction qui traite le retour de la requête
  - Vérifie l'état de la requête (en cours, terminée ...)
  - Récupère le contenu retourné par la requête
    - Contenu textuel, formaté JSON, XML, ou autre
  - Met à jour la page HTML courante avec ce contenu

XMLHttpRequest : fonctions

- open(requete HTTP, URL, boolean async)
  - requete HTTP : GET, POST …
  - async : à true par défaut (asynchrone)
- send(param)
  - Envoie la requête avec les paramètres (cas d'un POST sinon avec un GET on les met dans l'URL)
    - null si pas de paramètre
  - Si param est autre chose que du texte, on doit préciser le type MIME avec setRequestHeader('Content-Type', MIME)
- onreadystatechange = function
  - La fonction est appelée à chaque changement d'état du traitement de la requête

XMLHttpRequest : fonctions

- Vérification de l'état de la requête par les attributs
  - readyState : en fonction de l'état d'exécution de la requête
    - 5 états différents (0 à 4) : la valeur 4 indique que la requête a été exécutée et que le résultat est disponible
  - status : code de retour HTTP de l'appel de la requête
    - 200 si tout s'est bien passé
- Récupération du contenu de la réponse à la requête
  - responseText : sous forme de texte (brut, JSON ou autre)
  - responseXML : contenu XML
    - A parser comme un arbre DOM avec Javascript
- Cas d'une requête synchrone
  - Mettre false en dernier paramètre de open
  - Pas la peine de tester l'état ni d'associer une fonction de traitement, on est bloqué tant que la réponse n'arrive pas

Javascript de la page HTML

- Fonction qui est appelée à chaque touche tapée dans la zone de texte

```
var requete;

function completer() {
   // construit l'URL à appeler avec le contenu de la zone de texte
   var donnees = document.getElementById("entreeNom");
   var url = "completion?debut=" + escape(donnees.value);
   if (window.XMLHttpRequest) {
      requete = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
      // pour de vieux navigateurs IE
      requete = new ActiveXObject("Microsoft.XMLHTTP");
   }
   // définit la requête en mode GET et asynchrone
   requete.open("GET", url, true);
   // associe la fonction majPage pour gérer les retours de la requête
   requete.onreadystatechange = majPage;
   // envoie la requête HTTP
   requete.send(null);
}
```


Javascript de la page HTML

- Fonction appelée à chaque changement d'état de la requête

```
function majPage() {
   // état de la requête = 4 : requête terminée
   // il faut faire ce test car la fonction est appelée pour chaque
   // changement d'état de la requête
   if (requete.readyState == 4) {
      // status (code HTTP) 200 : tout s'est bien passé
      if (requete.status == 200) {
         // on récupère la balise <liste> dans le contenu XML retournée par la requête
         var liste = requete.responseXML.getElementsByTagName("liste")[0];
         // on parcourt la liste des noeuds de la balise <liste> : les noms de personnes
         var noms = liste.childNodes;
         var html="";
         for(var i=0; i < noms.length; i++) {
            nom = noms[i].childNodes[0].nodeValue;
            // pour chaque nom, on le rajoute avec un lien HTML vers
            // la fonction Javascrip majEntree
            html = html+"<a href=\"javascript:majEntree('"+nom+"')\">"+nom+"</a><br />";
         }
         // on remplit le <div> avec le contenu HTML qu'on vient de définir
         mdiv = document.getElementById("noms");
         mdiv.innerHTML = html;
      }
      // si le status n'est pas 200, il y a eu un problème
      else alert("Erreur sur la requête : "+requete.status);
   }
}
```


Javascript de la page HTML

- Fonction quand on clique sur le lien d'un nom

```
// on remplit la zone de texte avec le nom en paramètre
// et on on efface la liste
function majEntree(nom) {
   document.getElementById("entreeNom").value = nom;
   mdiv.innerHTML = "";
}
```

- Si on entre « Ro » dans la zone de texte, la balise est modifiée dynamiquement pour devenir

```
<div id="noms">
<a href="javascript:majEntree('Roger')">Roger</a><br>
<a href="javascript:majEntree('Robert')">Robert</a><br>
</div>
```

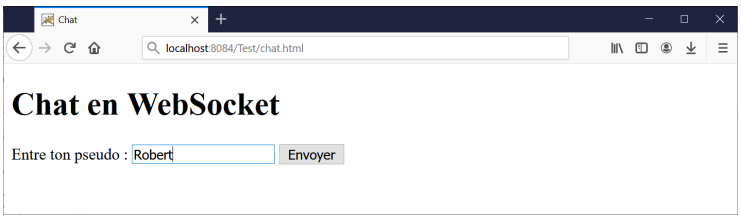
- Si on avait utilisé une exécution synchrone
  - Tant que la requête AJAX du « R » n'est pas terminée, le navigateur ne laisse pas l'utilisateur entrer le « o »

WebSocket

- Dans le principe, similaire aux sockets TCP
  - Un client ouvre une connexion sur le serveur
  - Une fois les sockets connectées, communication bi-directionnelle
- Particularités
  - Protocole niveau applicatif (couche 7 modèle OSI) comme HTTP
    - TCP : couche transport, niveau 4
    - Les Web sockets peuvent aussi s'utiliser au-dessus de HTTP
  - Peut s'utiliser dans des navigateurs Web
  - Multi-langages
    - Dans l'exemple du cours : serveur en Java et client en Javascript dans un navigateur Web
  - Programmation événementielle
    - On associe une fonction listener pour gérer un événement : une connexion établie, un message reçu, une erreur ...

Exemple : chat

- Discussion par chat
  - Un utilisateur entre son pseudo
  - Il peut ensuite envoyer des messages à tous les autres utilisateurs
  - Dès qu'un message est envoyé, toutes les pages des utilisateurs sont dynamiquement modifiées pour afficher le nouveau message
- Coté client
  - La page HTML affichée reste la même tout du long
  - Pour l'entrée du pseudo :



Exemple : chat

- Forme de la page HTML pendant la communication
  - Robert a validé son pseudo
  - Gérard et Simone sont également connectés et ont chacun envoyé un message

Page HTML

- Contenu de la page HTML chat.html

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Chat</title>
    <script language="javascript" type="text/javascript">
    . . .
    </script>
  </head>
  <body>
    <h1>Chat en WebSocket</h1>
    <div id="zonePseudo">
      <p>Entre ton pseudo :
      <input id="entreePseudo" value="" type="text">
      <input id="boutonPseudo" value="Envoyer" type="button" onclick="initialisation()">
      </p>
    </div>
    <div id="zoneMessage" style="display:none;">
      <p>Entre ton message <span id="pseudo"> </span> :
      <input id="message" name="message" value="" type="text" size=70>
      <input onclick="envoyerMessage()" value="Envoyer" type="button"><br />
      <input onclick="fermerSocket()" value="Fermer Communication" type="button">
      </p>
      <h2>La liste des messages :</h2>
      <p id="listeMessages"></p>
    </div>
  </body>
</html>
```


Page HTML

- La page définit deux
  - Identifiant zonePseudo pour la partie pour renseigner son pseudo
  - Identifiant zoneMessage pour la partie gérant les messages
  - Le second est masqué par défaut avec le style `style="display:none;"`
    - Il sera affiché quand le pseudo sera rentré dans le premier avec l'appel de la fonction Javascript initialisation() du bouton Envoyer

```
// initialisation : récupère le pseudo puis ouvre la connexion
function initialisation() {
   // récupère le pseudo de l'utilisateur
   pseudo = document.getElementById("entreePseudo").value;
   // remplit le <span> d'id pseudo avec le pseudo de l'utilisateur
   document.getElementById("pseudo").innerHTML = pseudo;
   // cache la zone d'entrée du pseudo et affiche celle des messages
   document.getElementById("zonePseudo").style.display = "none";
   document.getElementById("zoneMessage").style.display = "block";

   // ouvre la connexion avec la partie serveur
   ouvrirConnexion();
}
```


Ouverture connexion coté client

- Fonctionnement client/serveur classique
  - Le client connaît l'adresse du serveur : une URI de protocole ws
    - wss pour la version sécurisée (comme https vs http)
  - Le client ouvre la connexion sur la partie serveur
- Particularité Web socket
  - Associe une fonction Javascript pour chaque événement : ouverture de connexion, message reçu (du serveur), une erreur

```
var websocket;

function ouvrirConnexion() {
   websocket = new WebSocket("ws://localhost:8084/Test/chat");
   websocket.onopen = function(evt) {
      onOpen(evt)
   };
   websocket.onmessage = function(evt) {
      onMessage(evt)
   };
   websocket.onerror = function(evt) {
      onError(evt)
   };
}
```


Fonctions Javascript coté client

- Listeners d'événements

```
// appelée quand la connexion est ouverte
function onOpen(evt) {
   alert("Connexion établie");
}

// appelée quand le serveur envoie un message : rajoute la donnée de
// l'événement à la fin du paragraphe qui contient la liste des messages
function onMessage(evt) {
   liste = document.getElementById("listeMessages");
   liste.innerHTML = liste.innerHTML + "<br />" + evt.data;
}

// appelée quand il y a une erreur
function onError(evt) {
   alert("Erreur : " + evt.data);
}
```

- Quand on clique sur le bouton Envoyer

```
function envoyerMessage() {
   // récupère le contenu de la zone de texte
   message = document.getElementById("message").value;
   // envoie le message au serveur avec le pseudo
   websocket.send(pseudo + " : " + message);
   // efface le contenu de la zone de texte
   document.getElementById("message").value = "";
}
```


WebSocket Java coté serveur

- Dans l'implémentation Java : package javax.websocket
- Spécialisation de EndPoint avec 3 méthodes à redéfinir
  - Quand un client se connecte
    - void onOpen(Session session, EndpointConfig config)
  - Quand la connexion est fermée avec le client
    - void onClose(Session session, CloseableReason reason)
  - Quand une erreur a eu lieu
    - void onError(Session session, Throwable thr)
  - Peut utiliser à la place une classe annotée par @ServerEndpoint et annoter 3 de ses méthodes avec @OnOpen, @OnClose et @OnError
    - Définira aussi une méthode annotée par @OnMessage pour traiter un message reçu du client
- Session : objet représentant la session ouverte avec un client
  - RemoteEndpoint.Basic getBasicRemote() : point de communication synchrone avec l'élément connecté avec la socket
  - RemoteEndpoint.Async getAsyncRemote() : idem mais en asynchrone

Code Java coté serveur

- On définit une classe WSChat
  - Associée à l'URI « /chat » sur le serveur
    - L'URI utilisée par le navigateur Web coté client
    - Exécuté par un serveur Tomcat utilisant le port 8084
  - A chaque connexion d'un client, on rajoute son point de communication synchrone dans une liste
    - Liste définie en static car il y a une instance de WSChat par client connecté
      - Il faut pouvoir la partager facilement entre toutes les sockets
      - Ne se soucie par ici de l'accès concurrent (simplification)
  - A chaque réception d'un message venant d'un client
    - On parcourt la liste pour envoyer le message à chacun des clients connectés

Code Java coté serveur

```java
@ServerEndpoint("/chat")
public class WSChat {

    // la liste des websockets : en static pour être partagée
    private static ArrayList<Basic> listeWS = new ArrayList<>();

    @OnMessage
    public void message(String message) {
        // on parcourt toutes les WS pour leur envoyer une à une le message
        for(Basic ws : WSChat.listeWS) {
            try { ws.sendText(message); }
            catch (IOException ex) { System.err.println("Erreur de communication"); }
        }
    }

    @OnOpen
    public void open(Session session) {
        // à l'ouverture d'une connexion, on rajoute la WS dans la liste
        WSChat.listeWS.add(session.getBasicRemote());
    }

    @OnClose
    public void onClose(CloseReason reason) {
        System.out.println("Fermeture de la WS");
    }

    @OnError
    public void error(Throwable t) {
        System.err.println("Erreur WS : "+t);
    }
}
```


WebSocket

- Chat
  - Exemple simple d'utilisation avec échange de chaînes de caractères
- Données échangées
  - Autres formats : chaînes structurées (JSON...), binaires ...
  - L'URI de connexion peut accepter des paramètres
- Coté serveur
  - La méthode annotée @OnMessage peut retourner une valeur
  - Dans ce cas, le « return » de Java résulte en un envoi de message pour le client avec la valeur retournée
    - Pas utilisé dans l'exemple, on avait un retour void
- En fonction des implémentations, il peut y avoir des déconnexions automatiques au bout d'un certain temps
  - A gérer au besoin
- Voir les documentations spécialisées pour plus de détails sur le fonctionnement des Web sockets