CHAPITRE 11

Temps réel

Remy Sharp

LE WEB EN TEMPS RÉEL fait partie de ces aspects d'Internet qu'on apprécie d'utiliser mais qui peuvent être assez effrayants quand il faut les mettre en place. Ce chapitre présente les différentes technologies et la simplicité du code client.

Pour ajouter du temps réel à une application web, deux options s'offrent à vous : WebSockets et Server-Sent Events. WebSockets permet de créer un flux connecté vers votre serveur (une connexion TCP, donc) afin de mettre en place une communication bidirectionnelle et en temps réel entre le serveur et le client. Un exemple d'application classique est un client de chat, mais ses possibilités sont infinies. Les sockets web ont un long chemin à parcourir pour remplacer le code qui repose sur Comet. Ce dernier utilise en effet un grand nombre de techniques – qui sont assez souvent du bricolage – pour mettre en place un flux de données en temps réel à partir d'un serveur. Comme nous le verrons plus loin, WebSockets simplifie ce traitement du côté client.

Server-Sent Events, également appelé EventSource, “pousse” en temps réel des messages du serveur vers le navigateur client. Cette API convient donc parfaitement aux applications qui attendent des informations d’un serveur sans nécessiter d’interaction de la part de l’utilisateur – pour les mises à jour en direct d’informations, par exemple.

WebSockets et flux de données

WebSockets, l’API des sockets web, ne fait pas partie de la spécification HTML5, mais elle est très importante pour certaines des applications web “en temps réel” qui sont apparues ces dernières années.

Les sockets web fournissent une connexion bidirectionnelle entre le serveur et le client. Cette connexion est en temps réel et reste ouverte tant qu’elle n’est pas fermée explicitement. Lorsque le serveur veut envoyer un message au client, ce message est donc immédiatement “poussé” vers le navigateur.

C’est exactement le but que poursuivait Comet : Comet permet de créer une connexion en temps réel vers un serveur, mais en passant par plusieurs astuces différentes. Si aucune de ces astuces ne fonctionne, il se replie vers une interrogation Ajax qui accède constamment au serveur et ne s’adapte donc pas très bien à la montée en charge.

Si vous disposez d’une socket ouverte, en revanche, le serveur peut pousser les données vers toutes les sockets connectées et n’a pas besoin de répondre constamment aux requêtes Ajax. On passe donc du “polling” au “pushing” – du réactif au proactif. Ce que Comet tentait d’obtenir par des astuces, les sockets web l’intègrent nativement dans le navigateur.

> **NOTE** Si votre navigateur ne reconnaît pas nativement les sockets web, vous pouvez vous replier sur Flash. Hiroshi Ichikawa a en effet écrit une rustine en Flash pour les sockets web, disponible à l’URL **http://github.com/gimite/web-socket-js**.

Diminution de la latence des applications temps réel

Un énorme avantage des sockets web est la diminution de la latence. Une socket étant toujours ouverte et en écoute, les données n’ont plus qu’à parvenir à votre navigateur dès qu’elles ont été poussées par le serveur : la latence est donc extrêmement faible par rapport à celle d’une requête Ajax reposant sur `XMLHttpRequest`.

En théorie, avec Google Wave (ce projet a, depuis, été abandonné par Google), les personnes qui partageaient un document voyaient immédiatement toutes les touches tapées par les autres à mesure qu’elles saisissaient du texte. Avec de l’Ajax de base, en revanche, vous devriez créer un objet XHR chaque fois qu’une touche est pressée et il faudrait donc envoyer tous les en-têtes d’une requête XHR classique : agent utilisateur, types de contenus acceptés, etc. Cela fait beaucoup pour une seule frappe de touche…

Avec les sockets, la connexion étant toujours ouverte, il suffit d'envoyer le caractère saisi, qui sera ensuite diffusé à tous les clients connectés au serveur. Seule cette information sera envoyée.

Le volume des données transmises passe donc de 200-300 octets (avec Ajax) à 10-20 octets (avec une socket) : l'ensemble sera donc plus réactif et les données seront transmises plus rapidement aux clients connectés.

L'API WebSocket

L'API WebSocket est très simple à utiliser. Comme nous l'avons vu avec l'API de messagerie et les Web Workers, les navigateurs actuels (à l'exception de Firefox et des Web Workers) ne savent envoyer que des chaînes avec `postMessage` et `onmessage`. Il en va de même pour les sockets.

NOTE La mise en place d'un serveur pour le protocole `ws://` sort du cadre de ce livre, mais il existe déjà plusieurs bibliothèques permettant de l'ajouter. En utilisant des serveurs comme Node.js, on peut obtenir un serveur de sockets web en moins de 20 minutes. Cette mise en œuvre est décrite dans l'article **http://remysharp.com/slicehost-nodejs-websockets/**.

Cela signifie que vous ne pouvez pas (actuellement) envoyer des données binaires. Mais, dans le monde du Web, nous avons l'habitude de travailler avec JSON et il n'est pas très difficile d'encoder les messages en JSON à mesure qu'ils arrivent d'une socket, puisque c'est ce que nous faisons déjà pour les requêtes JSON Ajax.

L'API se borne à créer la connexion, à envoyer et recevoir des données sur la socket, et à fermer celle-ci. Elle propose également un gestionnaire d'erreur et un indicateur d'état signalant que la connexion est en cours d'établissement, qu'elle est ouverte, en cours de fermeture ou fermée. Une socket fermée est définitivement inutilisable et ne peut plus être rouverte : vous devrez créer une nouvelle socket.

La création d'une socket web est très simple et ressemble beaucoup à celle d'un Web Worker. Le protocole de l'URL doit être `ws://`, mais le reste peut être structuré comme n'importe quelle autre URL :

```
var socket = new WebSocket('ws://monserveur.com/tweets:8080/');
```

Dans cet exemple, j'attends les messages provenant de l'URL `tweets`. Chacun d'eux est un nouveau tweet de Twitter, que mon serveur écoute puisqu'il a été configuré pour cela (voir Figure 10.4).

CONSEIL L'URL utilisée pour la socket web ne doit pas nécessairement avoir la même origine que votre document. Cela signifie que vous pouvez vous connecter à des serveurs tiers, ce qui étend d'autant plus les possibilités qui s'offrent à vous.

Les messages du serveur, récupérés à partir de l'API Twitter, sont délivrés en JSON. Quand ils arrivent, il faut donc extraire les données et afficher le tweet à l'écran :

```
socket.onmessage = function(event) {
  var tweetNode = renderTweet(JSON.parse (event.data));
  document.getElementById('tweets').appendChild(tweetNode);
};
```

En quatre lignes de JavaScript (si l'on ne tient pas compte du code de la fonction `renderTweet`, qui se contente de transformer les données JSON en fragment HTML pour l'ajouter à la page), je peux donc afficher en temps réel les tweets sur ma page.

FIGURE 11.1 Connexion montrant les tweets que mon serveur écoute.

Manipulation d'une socket

Comme nous l'avons mentionné, il existe également des méthodes permettant de faire autre chose que simplement écouter une socket. À titre d'exemple, voici à quoi ressemblerait le code d'un client chat programmé avec WebSocket :

```
var socket = new WebSocket("ws://mon_serveur_chat.com:8080/"),
        me = getUsername();

socket.onmessage = function(event) {
  var data = JSON.parse(event.data);
  if (data.action == 'joined') {
    initialiseChat();
  } else {
    showNewMessage(data.who, data.text);
  }
};
```

```
socket.onclose = function () {
  socket.send(JSON.stringify({
    action: 'logoff',
    username: me
  }));
  showDisconnectMsg();
};
socket.onopen = function() {
  socket.send(JSON.stringify ({
    action: 'join',
    username: me
  }));
};
```

Cet extrait utilise les mêmes techniques que celles que nous avons employées avec l'API de messagerie pour contourner la limite des messages en texte pur : l'API des sockets est aussi simple que cela. Toute la négociation de la communication est prise en charge par le navigateur, sans que vous ayez à vous en soucier, et il en va de même pour la gestion du tampon (bien que vous puissiez vérifier la valeur de la propriété `bufferedAmount` de la socket). En fait, le processus de communication est encore plus simple que la configuration d'un objet XHR !

Server-Sent Events

Dans certaines situations, vous avez simplement besoin que le serveur envoie des messages à votre application. L'API Server-Sent Events est particulièrement bien adaptée aux applications qui affichent en temps réel des changements de prix, ou les dernières dépêches, ou toute information devant parvenir unilatéralement au navigateur en temps réel – si, en revanche, vous avez besoin d'une communication en temps réel bidirectionnelle, c'est WebSockets qu'il vous faut.

Server-Sent Events fournit `EventSource`, dont le fonctionnement ressemble beaucoup à celui d'une socket web : on crée un nouvel `EventSource` en lui passant l'URL à laquelle se connecter et le navigateur commence immédiatement à établir une connexion.

Un objet `EventSource` reconnaît trois événements simples :

- `open`. Lorsque la connexion a été établie.
- `message`. Lorsqu'un nouveau message arrive – la propriété `data` de l'événement contient alors le message brut.
- `error`. Si une erreur est survenue.

Ce qui rend un `EventSource` unique est la façon dont il gère les connexions coupées et le suivi des messages.

Si la connexion d'un objet EventSource est coupée pour une raison ou une autre, l'API tentera automatiquement de se reconnecter. Si vous utilisez des identifiants de message, l'objet indiquera au serveur lors de sa reconnexion l'identifiant du dernier message qu'il a reçu, ce qui permettra au serveur (si votre application le demande) d'envoyer au client la liste des messages qu'il a manqués.

Supposons, par exemple, que votre application trace un graphique en temps réel représentant le nombre de fois où Bruce évoque sa peluche rose favorite sur Twitter. Cette application représentera donc les sentiments de Bruce au cours du temps – vous saurez ainsi s'il est content ou non de la couleur, de la texture et de l'aspect général de l'objet.

Le navigateur se contentant de recevoir passivement des données du serveur, l'API Server-Sent Events semble tout particulièrement adaptée.

Supposons maintenant que votre connexion se coupe alors que vous surveillez attentivement les émerveillements de Bruce. Lorsque vous vous reconnecterez, EventSource indiquera au serveur que le dernier message avait l'identifiant 69. Si le serveur en est au message 78, l'application sur le serveur réalisera qu'elle en a manqué un certain nombre et le serveur lui renverra alors tous les messages à partir du 70. Le code du client n'a pas besoin d'être modifié car chacun de ces messages manquants déclenchera simplement l'événement message et tout sera représenté correctement sur le graphique.

Voici un exemple de code de cette application :

```
var es = new EventSource('/bruces-pink-toy');
es.onopen = function () {
  initialiseChart();
};
es.onmessage = function (event) {
  var data = JSON.parse(event.data);
  chart.plot(data.time, data.sentiment);
};
```

Server-Side Events – technologie côté serveur

Côté serveur, vous pourriez utiliser une configuration reposant sur PHP (LAMP, par exemple) mais, Apache (le "A" de LAMP) ne supportant pas très bien les connexions persistantes, la connexion sera sans cesse coupée. L'objet EventSource passera donc son temps à se reconnecter automatiquement, ce qui produira un résultat semblable à une application Ajax de type "polling".

Ce n'est donc pas la meilleure façon de procéder, mais il faut bien reconnaître que PHP est sûrement le ticket d'entrée le moins cher pour la plupart d'entre nous. Quoi qu'il en soit, pour vraiment tirer parti d'un EventSource, vous avez besoin d'une connexion persistante au serveur ce qu'une configuration LAMP classique ne peut pas vous fournir.

Vous pouvez, et vous devriez sûrement, choisir un serveur reposant sur les événements. Entrer dans les détails sort du cadre de ce livre mais je vous conseille d'étudier Node.js (une plateforme serveur reposant sur JavaScript) ou Twisted for Python.

Le serveur doit garder ouverte la connexion avec le client et il doit lui envoyer un en-tête avec le type MIME `text/event-stream`.

Il doit envoyer les nouveaux messages sous la forme suivante :

```
id: 1\n

data: { "sentiment": "aime", "time": "2011-06-23 16:43:23"}\n\n
```

Les deux retours à la ligne servent à signaler la fin du message. L'API permet également d'envoyer plusieurs lignes, grâce à quoi nous pouvons envoyer des phrases en texte clair (et non en JSON comme ci-dessus) :

```
data: C'est ma première ligne vraiment, vraiment, vraiment très très\n
data : longue, mais je n'ai pas encore fini.\n\n

data: Comme je suis placé après deux retours à la ligne, je suis un
data: nouveau message.\n\n
```

Dans cet exemple deux messages seulement seront envoyés. En outre, vous noterez qu'aucun identifiant n'est utilisé – ils ne sont pas obligatoires mais vous en aurez besoin si vous voulez pouvoir reprendre à partir d'où vous avez été coupé.

Exemple de serveur EventSource simple

Voici un code Node.js très simple qui accepte les connexions à un serveur `EventSource` et qui envoie des messages. Là encore, nous n'expliquerons pas le fonctionnement du serveur car cela sortirait du cadre de ce livre, mais ce code devrait vous donner un bon point de départ. Nous avons également simplifié la solution afin que le serveur se contente de prévenir les utilisateurs connectés en leur envoyant le nom de l'agent utilisateur des autres visiteurs connectés au même service. Nous garderons les graphiques sur le jouet spécial de Bruce pour un autre jour !

```
/** Lorsqu'ils créent un nouvel EventSource */
response.writeHead(200, {'Content-Type': 'text/event-stream',
                         'Cache-Control': 'no-cache'});
// on récupère le dernier événement et on force sa conversion en nombre
var lastId = req.headers['last-event-id']*1;
if (lastId) {
  for (var i = lastId; i < eventId; i++) {
    response.write('data: ' + JSON.stringify(history[eventId]) + '\
nid: ' +
                                                  eventId + '\n\n');
  }
}
// Enfin, on met en cache la réponse
connections.push(response);
/** Réception d'une requête web classique */
```

```
connections.forEach(function (response) {
  history[++eventId] = { agent: request.headers['user-agent'],
                         time: + new Date };
 response.write('data: ' + JSON.stringify(history[eventId]) + 'nid: ' +
                                                  eventId + '\n\n');
});
```

Le code client ressemble à celui-ci :

```
var es = new EventSource('/eventsource');
es.onmessage = function (event) {
  var data = JSON.parse(event.data);
  log.innerHTML += '<li><strong>' + data.agent +
                   '</strong><br> connecté le <em>' +
                    (new Date(data.time)) + '</em></li>';
};
```

C'est donc une application très simple mais, en coulisse, tout le travail s'effectue à l'aide d'événements "push" produits en temps réel par le serveur.

Implémentations disponibles

EventSource est assez bien reconnu. Chrome, Safari, Firefox et Opera le gèrent très bien – à l'heure où ce livre est écrit, on ne sait pas très bien s'il en sera de même pour IE10. Cependant, EventSource se repliant vers le polling, il est très facile de simuler cette API pour créer une rustine en JavaScript et en Ajax (vous trouverez quelques exemples à l'URL **https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills** sous la rubrique "EventSource").

J'ai également remarqué que si l'on créait un EventSource pendant ou immédiatement après le chargement d'une page, certains navigateurs continuaient à afficher leur icône de chargement, ce qui pourrait laisser croire que le chargement de la page n'est pas terminé alors que ce n'est pas le cas. Je ne sais pas si c'est un bogue dans l'implémentation ou une fonctionnalité pour garder les développeurs en éveil, mais il suffit d'attendre que le document ait fini son chargement et d'envelopper le code d'initialisation de l'objet EventSource dans un appel à `setTimeout(init, 10)` pour éviter ce problème.

Résumé

Ce chapitre vous a présenté un terme à la mode : temps réel. Cela dit, l'ajout du temps réel à un site web permet de le rendre vraiment attractif – maintenant que vous savez qu'il est très simple à mettre en œuvre en JavaScript, je suis sûr que vous ne pourrez plus résister. Il reste bien sûr un peu de configuration à effectuer sur le serveur mais, une fois cela mis en place, vous pourrez capter l'attention de tous ceux qui ont tendance à zapper de site en site. Passons maintenant à un sujet épous-tou-flant.