Communication Client-Serveur

Qu'est-ce qu'un client et un serveur exactement ? Comment communiquent-ils ? Comment passez-vous des données entre eux ?

Comprendre ces concepts est crucial pour comprendre le code que vous êtes sur le point de commencer à écrire. Alors avant de plonger dans le code, passons un peu de temps à revoir ces concepts.

1. Clients

Un client est tout ce que vous utilisez pour interagir avec Internet. C'est le navigateur Web que vous utilisez pour lire cette page. Le navigateur Web de votre ordinateur est un client, le navigateur Web de votre téléphone est un autre client. Il existe d'autres types de clients (comme l'application Netflix sur votre téléphone ou l'application Spotify sur votre ordinateur), mais nous nous concentrerons pour l'instant sur les navigateurs Web.

Une propriété importante d'un client est qu'il s'exécute **localement** sur **votre** ordinateur (ou téléphone ou console de jeu). Lorsque vous entendez le terme « côté client », cela signifie que cela se produit sur votre ordinateur. Cette distinction deviendra importante lorsque vous vous renseignerez sur les serveurs.

HTML

L'une des tâches principales des clients consiste à analyser le contenu HTML et à l'afficher à l'utilisateur.

Par exemple, disons que vous avez ce code HTML :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Mon Premier Page Web</title>
    </head>
    <body>
        <h1>Happy Coding</h1>
        <p>Hello world!</p>
    </body>
</html>
```

Un client (alias un navigateur Web) analyse ce contenu dans une page Web bien formatée :

Vous pouvez l'essayer en enregistrant ce code HTML dans un fichier .html, puis en ouvrant ce fichier dans un navigateur Web.

Vous apprendrez comment le client obtient le contenu HTML en une seconde, mais pour l'instant, vous pouvez considérer un client comme un moteur de rendu HTML. C'est à peu près exactement ce qu'est un navigateur Web.

JavaScript

Les navigateurs Web (alias clients) peuvent également exécuter du code JavaScript. Il s'agit **du** code **côté client** qui s'exécute sur **votre** ordinateur.

Par exemple, disons que vous avez ce code HTML, qui inclut du code JavaScript :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Happy Coding</title>
    <script>
     alert("hello!");
    </script>
    </head>
    <body>
        <p>Happy coding!</p>
    </body>
</html>
```

Essayez de l'enregistrer dans un fichier .html, puis ouvrez ce fichier avec votre navigateur Web. Vous devriez voir une boîte de dialogue hello! s'ouvrir lorsque vous chargez le fichier.

Ce code JavaScript s'exécute sur le client, dans votre navigateur Web. Il s'agit d'un exemple artificiel, mais le code JavaScript est ce qui rend les pages Web interactives. Les actions telles que l'animation et la réaction aux actions de l'utilisateur, telles que le clic sur un bouton, sont généralement effectuées en exécutant du code JavaScript.

2. Requêtes et réponses

Vous savez maintenant qu'un client est un navigateur Web qui affiche du HTML et exécute JavaScript localement. D'où viennent ces HTML et JavaScript ?

Ce contenu provient généralement d'une URL. Lorsque vous saisissez une URL dans la barre d'adresse de votre navigateur Web, votre navigateur Web envoie une **demande** à cette URL et le serveur renvoie une **réponse** contenant le contenu de cette URL.

3. Les serveurs

Un serveur est un ordinateur qui répond aux demandes en **fournissant des** réponses.

Comment un serveur répond-il à une demande d'URL ?

À un niveau élevé, il existe deux types de contenu :

- Contenu statique qui ne change pas beaucoup.
- Contenu dynamique qui change au fil du temps, ou basé sur qui consulte la page.

3.1. Fichiers statiques

Les pages Web simples qui contiennent du contenu statique qui ne change pas en fonction de la personne qui les lit sont souvent stockées dans des fichiers, tout comme les fichiers sur votre ordinateur.

Par exemple, passons en revue le processus de visite d'une page Web statique sur KevinWorkman.com/index.html:

- Vous tapez kevinworkman.com/index.html dans la barre d'adresse de votre navigateur Web et appuyez sur Entrée.
- Votre navigateur Web envoie une **demande** pour cette URL.
- Cette demande va à un serveur.
- Ce serveur recherche un fichier nommé index.html.
- Le serveur renvoie une **réponse** qui contient le contenu du fichier.
- Le client (votre navigateur Web) reçoit cette réponse et affiche le code HTML.

- Vous voyez la page Web bien formatée.

Ce chemin `URL -> request -> file -> response` est généralement suivi pour d'autres contenus statiques, comme les fichiers images `.css` et `.js`.

3.2. Contenu dynamique

Les fichiers statiques fonctionnent pour le contenu qui ne change pas souvent, mais qu'en est-il des pages Web plus avancées qui changent au fil du temps, ou des sites Web qui affichent du contenu publié par d'autres utilisateurs ? Il n'est pas possible que twitter.com soit un seul fichier statique, car le contenu de la page est différent selon les personnes que vous suivez et ce qu'elles ont publié.

Ce type de page Web nécessite un code côté serveur qui gère la demande et crée une réponse **dynamique en** fonction de l'auteur de la demande et des données stockées. Parcourons le processus de visite de Twitter :

- Vous tapez `twitter.com` dans la barre d'adresse de votre navigateur Web et appuyez sur Entrée.
- Le navigateur Web envoie une **demande** pour cette URL.
- La demande va au serveur de Twitter.
- Ce serveur exécute un code qui examine la demande, détermine de qui elle provient, récupère la liste des personnes que vous suivez et une liste de leurs tweets. Le serveur formate ces tweets en HTML.
- Le serveur renvoie une **réponse** qui contient ce code HTML.
- Le client (votre navigateur Web) reçoit cette réponse et affiche le code HTML.
    - **Remarque :** Du point de vue du client, peu importe que ce code HTML provienne d'un fichier statique ou d'un serveur dynamique. Quoi qu'il en soit, il s'agit de « juste » HTML pour le navigateur Web.
- Vous voyez la page de tweets bien formatée.

4. Rendu côté serveur

L'exemple ci-dessus a expliqué **comment** le serveur a formaté une liste de tweets en HTML. L'implémentation de l'intégralité du backend de Twitter est probablement hors de portée de ce didacticiel, commençons donc par un exemple plus petit.

Disons que vous voulez créer une page Web qui affiche l'heure Unix actuelle (le nombre de secondes depuis minuit le 1er janvier 1970). Lorsque le client demande l'URL de la page, vous pouvez utiliser la fonction `System.currentTimeMillis()` pour calculer l'heure Unix actuelle, que vous pouvez formater en HTML et envoyer en réponse.

Le terme «rendu côté serveur» est un peu trompeur, car cela signifie en réalité que le code HTML est **généré** sur le serveur. Il est toujours rendu par le navigateur sur le client.

Voici à quoi cela pourrait ressembler :

```
import java.io.IOException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/time.html")
public class UnixTimeServlet extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{
    long unixTimeSeconds = System.currentTimeMillis() / 1000;
    response.setContentType("text/html;");
    response.getWriter().println("<h1>Unix Time</h1>");
    response.getWriter().println("<p>Heure actuelle : " + unixTimeSeconds + "</p>");
    response.getWriter().println("<p>(<a href=\"\">Refresh</a>)</p>");
  }
}
```

Vous en apprendrez plus sur le fonctionnement exact de ce code dans d'autres didacticiels, mais pour l'instant, la chose importante à comprendre est que lorsque le client demande l'URL `/time.html` à notre serveur, le serveur répond avec du contenu HTML qu'il a généré à l'aide de code Java, plutôt que HTML provenant d'un fichier `.html`. Le client ne se soucie pas (ou même sait) comment le HTML a été généré. Il rend le HTML, peu importe d'où il vient.

L'annotation `@WebServlet` indique au serveur pour quelle URL ce code doit gérer les demandes. Cet exemple utilise `/time.html`, mais vous auriez pu choisir n'importe quoi. Les URL comme `/time` sans extension de fichier ou même les URL comme `/time/*` avec des caractères génériques fonctionnent bien !

5. Rendu côté client

Le rendu côté serveur vous permet de créer des pages Web dynamiques, mais il nécessite également que vous sortiez la page entière en une seule fois. Pour un petit exemple, ce n'est peut-être pas très grave, mais pensez à une page plus compliquée comme Twitter : et si vous vouliez charger plus de tweets lorsque l'utilisateur faisait défiler jusqu'en bas de son fil ?

Si vous vous fiez au rendu côté serveur, vous devrez inclure une tonne de tweets dans le code HTML que votre serveur renvoie au client. Cela pourrait ralentir votre page, et la plupart des utilisateurs ne verront probablement même pas la plupart des tweets de toute façon !

C'est là qu'une technologie appelée AJAX est utile. AJAX est une façon élégante de dire que vous pouvez utiliser JavaScript pour demander du contenu à un serveur. Avec AJAX, le flux de requête ressemble à ceci :

- L'utilisateur tape une URL dans la barre d'adresse de son navigateur Web.
- Le navigateur Web envoie une requête à un serveur.
- Le serveur répond avec un **contenu initial** . Ce contenu est généralement principalement statique, par exemple la barre de navigation en haut. Il comprend également du code JavaScript responsable de la construction des parties dynamiques de la page.
- Ce code JavaScript fait **une autre demande** pour plus de contenu.
- Cette requête va au serveur exactement comme n'importe quelle autre requête. Le serveur répond avec le contenu demandé. Il s'agit généralement du contenu dynamique et peut se présenter sous de nombreux formats différents : HTML, XML, JSON, texte brut, etc.
- Cette réponse revient au code JavaScript, et le code JavaScript analyse le contenu pour créer le reste de l'interface utilisateur à l'aide de fonctions JavaScript.
- L'utilisateur voit d'abord le contenu initial, peut-être une barre de chargement pendant que le contenu dynamique est demandé, puis le contenu dynamique est rempli.

Regardons à nouveau notre exemple de temps Unix, cette fois en utilisant l'approche de rendu côté client.

5.1. Points de terminaison du serveur

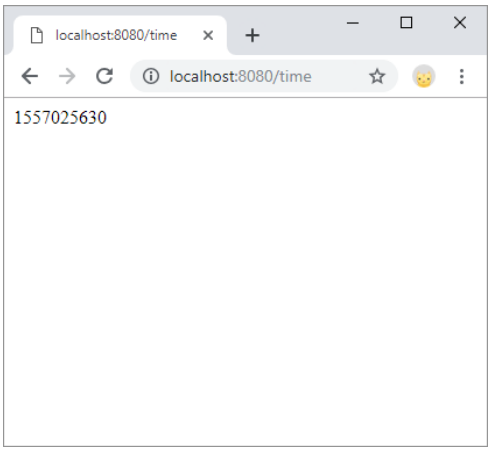
Tout d'abord, créez un **point de terminaison** (qui est un mot sophistiqué pour une URL qui répond avec des données) sur votre serveur. Dans cet exemple, le point de terminaison doit renvoyer l'heure Unix actuelle.

```
import java.io.IOException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
@WebServlet("/time")
public class UnixTimeServlet extends HttpServlet {

  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{
    long unixTimeSeconds = System.currentTimeMillis() / 1000;

    response.setContentType("text/html;");
    response.getWriter().println(unixTimeSeconds);
  }
}
```

C'est à peu près exactement le même code que vous avez vu ci-dessus avec le rendu côté serveur. La seule différence est qu'au lieu de sortir du contenu HTML, ce code répond avec une seule valeur : l'heure Unix actuelle.

Mais notez qu'il s'agit toujours d'une URL normale. Vous pouvez même le visiter dans un navigateur Web, exactement comme n'importe quelle autre URL :



Ce n'est pas une page Web très intéressante, mais c'est parce qu'elle n'est pas destinée à être vue par les humains ! Il existe donc JavaScript peut le demander et ensuite utiliser la réponse pour construire le HTML de la page.

5.2. Fetching

Vous savez maintenant comment créer un point de terminaison côté serveur qui gère une demande d'URL, exécute du code Java et renvoie des données dans la réponse.

L'autre moitié est le code JavaScript qui demande les données au serveur et les utilise pour créer le code HTML de la page.

client.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Unix Time</title>
    <script>
      function requestUnixTime() {
        const resultContainer = document.getElementById('result');
        resultContainer.innerText = 'Loading...';

        fetch('/time') // request data from the server
        .then(response => response.text()) // convert the response to raw text
        .then((unixTime) => {
          // build the HTML of the page
          resultContainer.innerText = unixTime;
        });
      }
    </script>
  </head>
  <body onload="requestUnixTime();">
    <h1>Unix Time</h1>
    <p>Current Unix time: <span id="result"></span></p>
    <p>(<a href="">Refresh</a>)</p>
  </body>
</html>
```

Ce fichier contient du HTML **statique** et du JavaScript. Lorsqu'un utilisateur accède à l' URL `user.html`, le client envoie une requête et le serveur renvoie ce contenu en réponse. Le client affiche alors la page et appelle la fonction `requestUnixTime()`.

La fonction `requestUnixTime()` utilise l'API Fetch pour envoyer une requête à l'URL `/time`. Comme dans l'exemple ci-dessus, le serveur gère cette URL en renvoyant une réponse contenant l'heure Unix actuelle. Le

code JavaScript utilise cette réponse pour créer l'interface utilisateur, dans ce cas en affichant l'heure Unix actuelle dans la page.

6. Envoi de données au serveur

Jusqu'à présent, chaque exemple impliquait un client **demandant du** contenu à un serveur et le serveur **répondant** avec le contenu demandé.

Mais il est également possible pour un client d'envoyer des données au serveur. Il existe plusieurs façons de procéder, selon le type de données que vous souhaitez envoyer.

6.1. Paramètres de requête

Les paramètres de requête sont des propriétés `key=value` ajoutées à la fin d'une URL après un point d'interrogation `?`. Par exemple, regardons cette URL :

- **https://www.youtube.com/watch?v=PBsUD40nPkI**

Cette URL pointe vers https://www.youtube.com/watch et inclut un paramètre `v` avec une valeur de `PBsUD40nPkI`. Le serveur YouTube qui gère cette demande peut utiliser ce paramètre pour charger la bonne vidéo et la renvoyer dans le cadre de la réponse.

Vous pouvez passer plusieurs paires `key=value`, séparées par une esperluette `&`.

- **https://www.youtube.com/watch?v=PBsUD40nPkI&t=42**

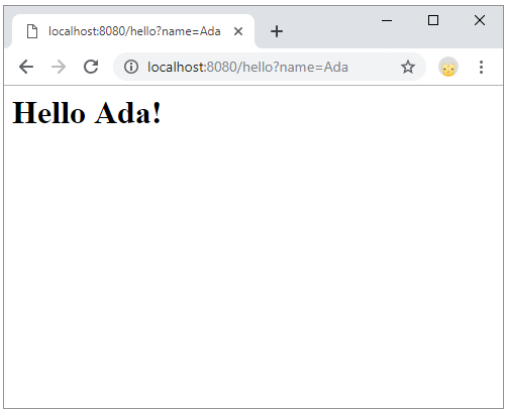
Cette URL pointe à nouveau vers https://www.youtube.com/watch et inclut deux paramètres : `v` avec une valeur de `PBsUD40nPkI` et `t` avec une valeur de `42`. Le serveur YouTube qui gère cette demande utilise ces paramètres pour charger une vidéo et passe à 42 secondes.

Voici un exemple de ce à quoi cela ressemblerait côté serveur :

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet {

  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{
    String name = request.getParameter("name");
    response.setContentType("text/html;");
    response.getWriter().println("<h1>Hello " + name + "!</h1>");
  }
}
```

Ce code serveur gère les requêtes adressées à l'URL `/hello` et obtient la valeur d'un paramètre `name`, qu'il utilise pour répondre avec du contenu HTML. Cet exemple génère du HTML directement, mais vous pouvez également utiliser des paramètres de requête avec des points de terminaison.



6.2. Chemin de l'URL

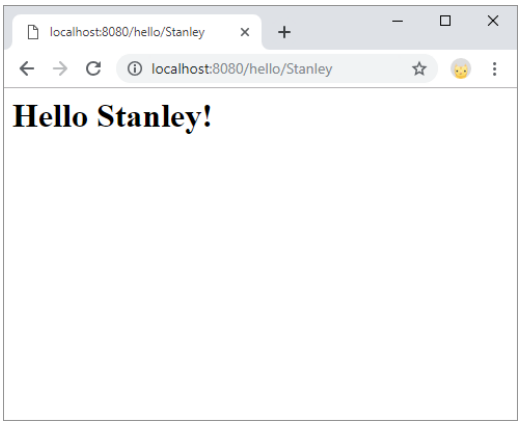
Les paramètres de requête ne sont pas considérés comme faisant partie de l'URL : ils viennent *après* l'URL. Mais il est également possible de transmettre des données dans l'URL elle-même.

Tout d'abord, vous créez un gestionnaire sur le serveur pour toute URL commençant par un certain chemin. Par exemple, vous pouvez créer un gestionnaire pour n'importe quelle URL qui commence par `/hello/`, donc `/hello/Ada`, `/hello/Stanley`, et `/hello/Grace` qui déclenchent toutes le même code. Ensuite, ce code pourrait analyser l'URL pour déterminer quel nom a été transmis. Cela ressemblerait à ceci :

```
import java.io.IOException;
```

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello/*")
public class HelloServlet extends HttpServlet {

  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{

    // name is whatever comes after /hello/
    String name = request.getRequestURI().substring("/hello/".length());

    response.setContentType("text/html;");
    response.getWriter().println("<h1>Hello " + name + "!</h1>");
  }
}
```

Ce code utilise une URL générique pour correspondre à n'importe quelle URL commençant par `/hello/`, puis utilise la `substring()` fonction pour obtenir le nom de l'URL qui vient après la partie `/hello/`.



6.3. Requêtes POST

Vous pouvez utiliser des paramètres de requête ou des chemins d'URL pour transmettre de petites chaînes à votre serveur, mais il y a quelques considérations importantes avec ces approches :

- Les URL apparaissent dans votre historique et dans les journaux du serveur. Ce n'est peut-être pas un gros problème pour les URL comme `/hello/Ada` ou `/watch?v=PBsUD40nPkI`, mais ce n'est pas un bon moyen de transmettre des données comme des mots de passe ou des informations de carte de crédit.
- Il y a une limite à la longueur d'une URL. La limite exacte dépend du navigateur que vous utilisez, mais elle est généralement d'environ 2000 caractères . Cela peut sembler suffisant pour des paramètres uniques, mais que se passe-t-il si vous souhaitez laisser les utilisateurs écrire des articles de blog ?
- Une URL est une chaîne de caractères. Que faire si vous souhaitez envoyer des données binaires au serveur, comme télécharger un fichier image ?

Vous pouvez utiliser des requêtes `POST` pour résoudre ces problèmes.

Toutes les requêtes que vous avez vues jusqu'à présent sont des requêtes `GET` : un client demande à **obtenir du** contenu à partir d'un serveur. Lorsque vous visitez `google.com` dans votre navigateur, vous faites une requête `GET` à `google.com`. C'est aussi pourquoi tout le code du serveur a utilisé la fonction `doGet()` jusqu'à présent.

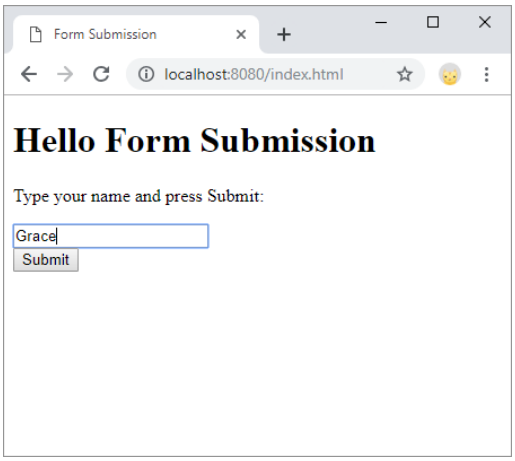
Par comparaison, une `POST` demande se produit lorsqu'un client demande d' **envoyer** (publier) du contenu à un serveur. L'un des exemples les plus courants est de remplir un formulaire. Lorsque vous remplissez un formulaire et appuyez sur le bouton `Submit`, votre navigateur fait une requête `POST` au serveur, contenant les données du formulaire.

Voici un exemple de fichier HTML contenant un formulaire :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Form Submission</title>
  </head>
  <body>
    <h1>Hello Form Submission</h1>
    <p>Type your name and press Submit:</p>

    <form method="POST" action="/hello">
      <input name="name" />
      <br/>
      <button>Submit</button>
    </form>

  </body>
```

```
</html>
```

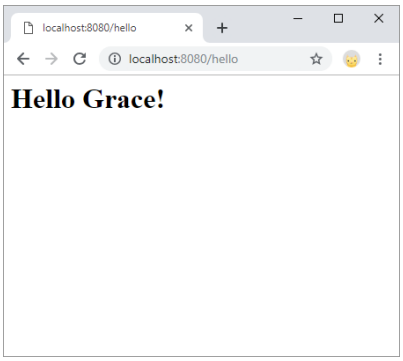
Notez les attributs method et action de l'élément form. Cela indique au navigateur de faire une requête POST à l'URL /hello contenant les données du formulaire lorsque l'utilisateur appuie sur le bouton Submit.



Le côté serveur ressemblerait à ceci :

```
import java.io.IOException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet {

  @Override
     public void doPost(HttpServletRequest request, HttpServletResponse response) throws
IOException {
    // the name parameter comes from the form
    String name = request.getParameter("name");

    response.setContentType("text/html;");
    response.getWriter().println("<h1>Hello " + name + "!</h1>");
  }
}
```

La seule particularité de ce code est qu'il utilise la doPost()fonction, qui gère les requêtes POST. Un projet plus réaliste effectuerait généralement un certain traitement des données, comme le stockage dans une base de données, puis la redirection vers une URL différente, mais cet exemple génère du HTML comme réponse.



7. Serveurs locaux

Lors du développement local, vous effectuerez généralement le déploiement sur un serveur **local** , ce qui signifie que vous exécuterez **à** la **fois** le serveur et le client sur votre ordinateur. Une fois que vous êtes prêt à publier votre code, vous le déploierez sur un serveur **distant** afin que d'autres personnes puissent interagir avec lui.

Je le mentionne maintenant parce qu'une façon courante de parler de code est de savoir s'il s'exécute « sur le serveur » ou « sur le client ». Cette distinction est plus évidente lorsqu'on parle d'un serveur distant, car dans ce cas le client et le serveur sont deux ordinateurs différents. Lorsque vous déployez localement, le serveur et le client se trouvent être le même ordinateur, mais vous devez toujours penser en termes de côté serveur et côté client.

Si cela vous aide, essayez d'imaginer le serveur comme un ordinateur complètement séparé, même si vous déployez localement.