3IN017 - TECHNOLOGIES DU WEB

Échanges client-serveur

11 mars 2025

Gilles Chagnon

Plan

Cours précédents côté client, puis côté serveur.

Maintenant, nous allons les faire discuter entre eux. . .

Formulaires HTML et requêtes HTTP

Il faut passer à l'`action` avec `method`

Les formulaires HTML permettent d'envoyer n'importe quel type de requête HTTP *via* :

- l'attribut `method` : pour le type de requête
- l'attribut `action` : pour l'URL

Par exemple...

```
<form method="POST" action="/reponse">
(...)
</form>
```

Lors de la soumission du formulaire, une requête POST sera envoyée au serveur à l'adresse indiquée.

Soumission du formulaire

Deux possibilités pour soumettre un formulaire :

- un simple élément `button`, par exemple

```
<form method="POST" action="/reponse">
(...)
<button>OK</button>
</form>
```

- un élément `input` de type "submit" :

```
<form method="POST" action="/reponse">
(...)
<input type="submit">
</form>
```

  Cet élément peut avoir plusieurs attributs dont :
  - `value` : permet de changer son texte (sinon, texte par défaut du système d'exploitation, souvent « Envoyer »)
  - `formaction` : s'il est indiqué prend le dessus sur la valeur de l'attribut `action` de `form`
  - `formmethod` : s'il est indiqué prend le dessus sur la valeur de l'attribut `method` de `form`

Notez que dans ces deux cas, le navigateur *charge* une nouvelle page lors de la soumission du formulaire. Pour l'éviter, on peut simplement écrire

```
<button type="button">OK</button>
```

Champs de formulaire

Les valeurs des champs de formulaire sont transmises à l'aide de l'attribut `name`. Par exemple...

```
<form method="POST" action="/api">
        <label for="chp_texte">Saisissez un texte :</label>
        <input id="chp_texte" name="texte">
        <input type="submit" value="OK">
</form>
```

... envoie une requête POST avec dans le body `texte=`*`valeur_saisie_dans_le_champ`* d'identifiant `chp_texte`

Gestion des formulaires côté serveur

Côté serveur, Express peut traiter ces requêtes à condition d'utiliser le *middleware* `urlencoded` :

```
router.use(express.urlencoded({extended: true}))
```

Il suffit alors de traiter la route, le corps de la requête se trouvant dans `req.body` :

```
router.post('/echo', (req, res) =>{
// Ce code renvoie juste la contenu de sa requête au navigateur
// req.body est un JSON
        res.send(JSON.stringify(req.body));
        res.end();
})
```

Promesses

Utilité des promesses (1)

Considérons par exemple le code suivant. Il nécessite deux *callbacks* en arguments pour gérer succès ou échec :

```
function faireQqcALAncienne(successCallback, failureCallback){
        console.log("C'est fait");
        // réussir une fois sur deux
        if (Math.random() > .5) {
                successCallback("Réussite");
        } else {
                failureCallback("Échec");
        }
}

function successCallback(résultat) {
        console.log("L'opération a réussi avec le message : " + résultat);
}

function failureCallback(erreur) {
        console.error("L'opération a échoué avec le message : " + erreur);
}

faireQqcALAncienne(successCallback, failureCallback);
```

Utilité des promesses (2)

SCIENCES SORBONNE UNIVERSITÉ

On le remplace par une fonction renvoyant un objet Promise :

```
function faireQqc() {
        return new Promise((successCallback, failureCallback) => {
                console.log("C'est fait");
                // réussir une fois sur deux
                if (Math.random() > .5) {
                        successCallback("Réussite");
                } else {
                failureCallback("Échec");
                }
        })
}

function successCallback(résultat) {
  console.log("L'opération a réussi avec le message : " + résultat);
}

function failureCallback(erreur) {
  console.error("L'opération a échoué avec le message : " + erreur);
}

const promise = faireQqc();
promise.then(successCallback, failureCallback);
// ou faireQqc().then(successCallback, failureCallback);
```

Promsesses : avantages

- les *callbacks* ne sont appelés que si et quand c'est nécessaire
- *surtout*, la méthode `then()` renvoyant aussi une promesse, on peut les chaîner, avec chaque opération qui démarre lorsque la précédente a réussi et en utilisant le résultat de l'étape précédente. Par exemple :

```
const promesse1 = faireQqc();
const promesse2 = promesse1.then(successCallBack, failureCallBack);
// ou const promesse2 = faireQqc().then(successCallBack,
↪  failureCallBack);
```

Promesses : Chaînage

On ne pouvait enchaîner des fonctions asynchrones qu'avec des cascades de *callbacks* :

```
faireQqc(function(result) {
        faireAutreChose(result, function(newResult) {
                faireUnTroisiemeTruc(newResult, function(finalResult) {
                        console.log('Résultat final :' + finalResult);
                }, failureCallback);
        }, failureCallback);
}, failureCallback);
```

ce qui donne avec les promesses :

```
faireQqc().then(function(result) {
        return faireAutreChose(result);
})
.then(function(newResult) {
        return faireUnTroisiemeTruc(newResult);
})
.then(function(finalResult) {
        console.log('Résultat final : ' + finalResult);
})
.catch(failureCallback);
```

Promesses – Chaînage avec fonctions fléchées

SCIENCES SORBONNE UNIVERSITÉ

```
faireQqc()
.then(result) => faireAutreChose(result))
.then(newResult) => faireUnTroisiemeTruc(newResult))
.then(finalResult) => {
        console.log('Résultat final : ' + finalResult);
})
.catch(failureCallback);
```

On définit alors la fonction `failureCallback` :

```
function failureCallback(erreur) {
                console.error("L'opération a échoué avec le message : " + erreu
}
```

(de la même manière qu'avec un gestionnaire d'événement, l'objet `erreur`, de type `error` possède des propriétés et méthodes ; voir `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error`)

Promesses – Chaînage après un catch

Si on a besoin de continuer un traitement même si une opération intermédiaire a échoué :

```
new Promise((resolve, reject) => {
        console.log('Initial');
        resolve();
})
.then(() => {
        // On force une erreur dans cet exemple
        throw new Error("Oups!");
        console.log('Fais ci');
})
.catch((err) => {
        console.error(err.message);
})
.then(() => {
        console.log("Fais ceci quoi qu'il se soit passé avant");
});
```

... qui produit...

```
Initial
Oups!
Fais ceci quoi qu'il se soit passé avant
```

Axios pour simplifier la gestion des requêtes

Axios : utilité et principe

SCIENCES SORBONNE UNIVERSITÉ

Problème

Il peut être fastidieux de retenir les syntaxes de mise en œuvre des requêtes et réponses HTTP côté serveur et client.

Axios

Axios est un client HTTP :

- *isomorphique* : même syntaxe côté client que serveur
- utilisant `http` côté serveur
- utilisant `XMLHttpRequest` côté client
- basé sur les promesses

- documentation officielle : `https://axios-http.com/fr/docs/intro`
- plus sur `XMLHttpRequest` : `https://developer.mozilla.org/fr/docs/Web/API/XMLHttpRequest`

Axios – Requête GET (1)

Axios met à disposition, côté client, des méthodes permettant de facilement gérer les requêtes, et notamment le format JSON des réponses du serveur. Par exemple, si sur le serveur on a...

```
router.get('/user', (req, res) => {
        let new_user=JSON.stringify({nom: "Hugo", prenom: "Victor"});
        res.json(new_user);
});
```

...ce qui envoie au client qui envoie un GET sur l'URL `/user` le JSON :

```
{
        "nom": "Hugo",
        "prenom": "Victor"
}
```

Axios – Requête GET (2)

SCIENCES SORBONNE UNIVERSITÉ

alors dans le composant React

```
// import du module
import axios from 'axios';
import {useState} from 'react';

// port du serveur; par défaut axios écoute le port 80
axios.defaults.baseURL = 'http://localhost:8000';

function App() {
        const [personnes, setPersonnes] = useState([]);

        axios.get('/user')
        .then(res => {
                // les données reçues sont converties dans l'objet res.data
                const personne = res.data;
                // on agglutine personne dans le tableau personnes, qu'on
                ↪  réaffecte ensuite à l'état
                setPersonnes([...personnes, personne]);
        })
        .catch(res) => {
                // traitement de l'erreur
        });

        return (
            <div className="App">
                    ...
```

Axios – Gestion des erreurs

SCIENCES SORBONNE UNIVERSITÉ

En complétant l'exemple précédent, par exemple :

```
function App() {
        const [personnes, setPersonnes] = useState([]);
        const [error, setError] = useState(null);

        axios.get('/user')
        .then(res => {
                const personne = res.data;
                setPersonnes([...personnes, personne])
        })
        .catch(e) => {
                setError(e);
        });

  return (
    <div className="App">
      {(error)?<p>{error.message}</p>:<p>OK</p>}
    </div>
  );
}
```

Axios – Requête POST (1)

SCIENCES SORBONNE UNIVERSITÉ

De même, Axios permet de simplifier la gestion des requêtes POST et si côté serveur on a...

```
router.post('/adduser', (req, res) =>{
        // affichage du corps de la requête dans la console du serveur
        console.log(req.body);
        // renvoi de la requête au format JSON vers le client
        res.json(JSON.stringify(req.body));
})
```

Axios – Requête POST (2)

SCIENCES SORBONNE UNIVERSITÉ

... on a en regard côté client...

```
function App() {
        const [nom, setNom] = useState("");

        const handleSubmit = (e)  => {
                e.preventDefault();

                axios.post('/adduser', { "chaine": nom })
                .then(res => { console.log(res.data) })
        }

        const changeNom = (e) => { setNom(e.target.value); }

        return (
        <div className="App">
                <form onSubmit={handleSubmit}>
                        <label htmlFor="chp_nom">Nom</label>
                        <input id="chp_nom" name="chp_nom"
                        ↪  onChange={changeNom}/>
                        <input type="submit" />
                </form>
        </div>
        );
}
```

Axios – Requête POST (3)

Le code précédent fonctionne ainsi :

1. l'utilisateur saisit une valeur dans le champ de texte
2. il valide en actionnant le bouton
3. React intercepte la soumission du formulaire en lançant le gestionnaire `handleSubmit`
4. celui-ci empêche le comportement par défaut de l'événement (chargement d'une nouvelle page) par `e.preventDefault();`
5. le client envoie au serveur une requête POST avec dans le body le contenu `{ chaine: 'chaîne saisie' }`
6. le serveur reçoit la requête, affiche ce body dans sa console et le renvoie au format JSON
7. le client reçoit cette réponse et l'affiche dans sa console : `{"chaine":"chaîne saisie"}`.

Il n'y aucun réel traitement ici mais n'importe quelle manipulation des données est possible.

Axios – Côté serveur, Instance de base

SCIENCES SORBONNE UNIVERSITÉ

Requêtes souvent adressées à un seul serveur

- soucis de mise à jour si changement de serveur ou de port
- code redondant

On définit alors une instance de base. Par exemple dans un fichier api.js, côté serveur :

```
import axios from 'axios';

export default axios.create({
// c'est aussi l'endroit où spécifier le port éventuel
        baseURL: "http://monserveurdapi.com/api"
});
```

Il suffit alors de l'importer avant de l'utiliser :

```
import API from './api';
(...)
API.post('/users/adduser',...)
.get('/messages/getmessage', ...);
```

Axios – Côté client, async et await (1)

SCIENCES SORBONNE UNIVERSITÉ

Principe d'async et await :

- async devant une déclaration de fonction signifie que cette fonction renvoie une promesse. Les deux syntaxes suivantes sont équivalentes :

```
async function f(){
        return("resultat");
}

function f(){
        return Promise.resolve("resultat");
}
```

  On peut alors écrire `f().then(alert)` qui affiche "résultat" dans une boîte d'alerte (ne pas oublier que l'argument de `then` est un *callback* qui est appelé avec comme argument ce qui est retourné par la promesse, ici la chaîne de caractères "resultat").
- `await` s'utilise dans une fonction déclarée avec `async`. Si `maPromesse` est une promesse, et si on écrit `let a = await maPromesse;` alors JavaScript va attendre que la promesse renvoie un résultat avant de continuer l'exécution de la fonction.

Axios – Côté client, async et await (2)

SCIENCES SORBONNE UNIVERSITÉ

async et await permettent de ne pas recourir explicitement à then et simplifient l'écriture avec Axios. Par exemple...

```
async deleleUser (user_id) {
        try {
                let reponse  = await axios.delete("/users/"+user_id);
        } catch (erreur) {
                console.error(erreur.message);
                return;
        }
        console.log(reponse);
}
```

Axios – deux didacticiels avec React

- https://www.digitalocean.com/community/tutorials/react-axios-react-fr
- https://educrak.com/chapitre-content/64bea6c68842f-axios-pour-les-requetes

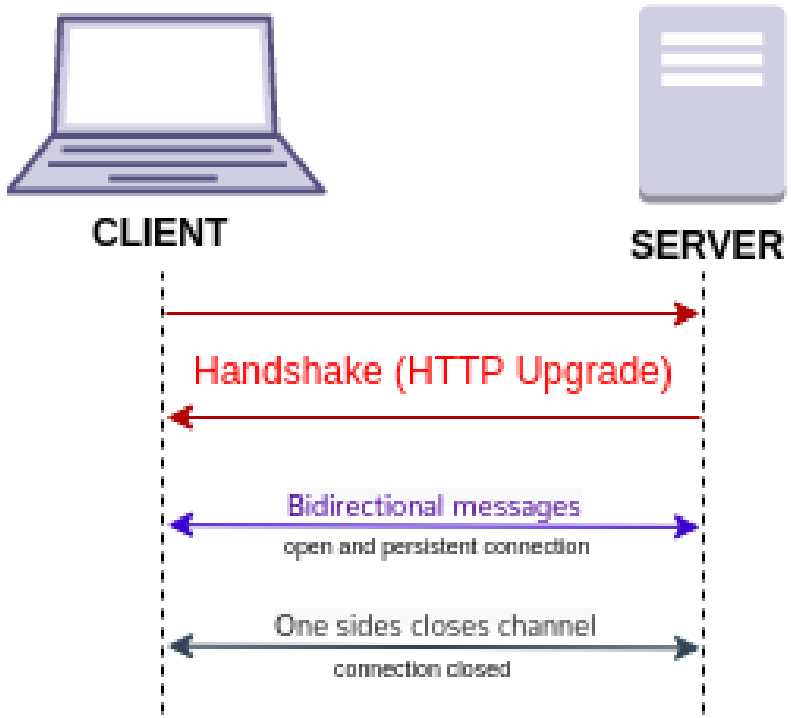
Échanges asynchrones avec WebSocket

Websocket : principe

Websocket c'est :

- un protocole normalisé en 2011
- une interface de programmation, *living standard* maintenu par le W3C

pour assurer des communications asynchrones bidirectionnelles client↔serveur.
Websocket ne fait pas appel à http.



Websocket – bibliothèques

Plusieurs bibliothèques pour se simplifier la vie, par exemple...

- socket.io, sans dépendance et capable en plus de Websocket de gérer les connexions *via* cookie ou répartition de charge. Attention, couteau multi-fonction qui recourt à Websocket si possible mais a d'autres possibilités pour marcher sur les vieux navigateurs
- ws pour une implémentation plus « pure » de Websocket
- react-use-websocket pour l'intégration avec React
- ...

Mais il est possible d'utiliser l'API standard (`https://developer.mozilla.org/fr/docs/Web/API/WebSocket`).

Websocket – côté client (1)

Il faut d'abord créer un nouvel objet `WebSocket`. Attention, ce n'est pas le protocole http, mais ws

```
let socket = new WebSocket("ws://monserveurws.com");
```

Pour une connexion sécurisée, il faut utiliser wss (comme http/https) si le serveur le supporte.

Le socket contient 4 événements :

`open`	connexion établie
`close`	connexion fermée
`error`	erreur WebSocket
`message`	donnée reçue

Pour envoyer des données,

Websocket – côté client – Exemple

SCIENCES SORBONNE UNIVERSITÉ

```
let socket = new WebSocket("ws://monserveurws.com");

socket.addEventListener("open", (e) => {
        console.log("[open] Connexion ouverte");
        console.log("Envoi au serveur");
        socket.send("Je m'appelle Henri");
});

socket.addEventListener("message", (evt) => {
  console.log("[message] Données reçues du serveur: "+evt.data);
});


socket.addEventListener("close", (evt) => {
        if (evt.wasClean) {
                console.log("[close] Connexion fermée proprement, code="
                +evt.code+" raison="+evt.reason});
         } else {
                // par exemple, processus serveur arrêté ou panne réseau
                // evt.code est généralement 1006 dans ce cas
                console.log('[close] Connexion fermée');
        }
});

socket.addEventListener("error", (err) => { console.error(err);});
```

Websocket – Côté serveur avec ws (1)

Il faut au préalable installer ws : `npm install ws` puis créer un fichier (par exemple wsserver.js) à lancer avec node : `node wsserver.js`.

```
// Import des modules nécessaires
const WebSocketServer = require('ws');

// Création d'un nouveau serveur Websocket
const wss = new WebSocketServer.Server({ port: 8080 })

// Quand arrive une connexion avec Websocket (après l'exécution de let
↪  socket = new WebSocket("ws://monserveurws.com"); côté client)
wss.on("connection", ws => {
        console.log("Nouveau client connecté");
        // instructions à exécuter quand la connexion a été établie
});
console.log("Le serveur WebSocket fonctionne sur le port 8080");
```

Websocket – Côté serveur avec ws (2)

SCIENCES SORBONNE UNIVERSITÉ

Exemple d'instructions

```
wss.on("connection", ws => {
        console.log("Nouveau client connecté");

        // envoi d'un message au client
        ws.send('Bienvenue, la connexion est active!');

        // à la réception d'un message du client
        ws.on("message", data => {
                console.log("En provenance du client: "+data)
        });

        // Gestion de la déconnexion du client
        ws.on("close", () => {
                console.log("Fin de la connexion du client");
        });
        // si erreur de connexion du client
        ws.onerror = function () {
                console.log("Une erreur est survenue")
        }
});
```

API Notification

Principe

- une API standard : `https://developer.mozilla.org/fr/docs/Web/API/Notification`
- utilise les notifications du système d'exploitation
- fonctionne uniquement en contexte sécurisé (https)
- il faut demander la permission à l'utilisateur
- 3 états possibles, accessibles *via* la propriété en lecture seule `Notification.permission` :
  - default : permission pas encore demandée
  - granted : notifications autorisées
  - denied : notifications refusées

Fonctionnement

- On demande la permission avec une promesse :

```
Notification.requestPermission().then((result) => {console.log(result);});
```

- on crée une nouvelle notification avec `Notification` :

```
const img = '/mawebapp/img/icon-128.png';
const texte = 'Voici une notification';
const notif = new Notification('Mise à jour :', { body: texte,
↪  icon: img });
```

- 4 événements : error, click, close, show ; accessibles avec :
  - les raccourcis onerror, onclick, onclose, onshow ; par exemple,

```
notif.onclose= (e) => {console.log(notification fermée)}
```

  - ou un gestionnaire d'événement :

```
notif.addEventListener("click", (...))
```

Conclusion

Nous avons passé en revue plusieurs manières pour un serveur d'échanger avec le client :

- en utilisant du « bon vieil » HTML pour gérer les envois de requêtes et les données
- avec une bibliothèque, Axios, pour simplifier la gestion des requêtes
- avec une API, Websocket, qui ne repose pas sur le protocole http
- en utilisant le système de notification du système d'exploitation

En fonction de vos besoins et de la nature du site/de l'application Web que vous créerez, vous choisirez une manière ou une autre. . .