Front-End : de 1990 à 2010

O. Hermant

16 décembre 2024

Ce TP a pour but de vous faire (re-)voir des concepts que nous utiliserons intensivement dans la programmation mobile :

- les balises et les attributs ;
- le style CSS ;
- le langage javascript ;
- le modèle de programmation client-serveur à travers une API, vu du côté client (la partie serveur sera abordée plus tard) ;
- la programmation asynchrone avec des fonctions de callback.

1 Pages web du XXe siècle : 100% HTML

Exercices :

- créer une page web vide (vous pouvez tout à fait la copier-coller des exemples de cours), et la visualiser dans un navigateur (`Open File (Ctrl + O)`)
- ajouter à votre page une division (balise `<div>` – ne pas oublier la balise fermante) et du texte à l'intérieur
- dans cette div, ajouter une liste non ordonnée (balise `<ul>`), avec quelques items (`<li>`)
- ajouter un peu de style : des bords noirs à la div précédente
  - il faut définir l'attribut `style` à la div
  - la propriété de style à définir est `border`, que vous pouvez fixer à `solid`
- faites en sorte que les éléments de la liste commence par une cercle blanc (ou tout autre style de votre choix) au lieu d'une puce,
  - comme
  - ceci.
- (facultatif) ajouter encore plus de style (bords de couleur bleue, texte aligné à droite, couleur de fond de la div, etc). Voir par exemple

  `https://www.w3schools.com/cssref/`

  pour une référence complète sur les propriétés de style.
- faites valider chacune de vos pages précédentes :

  `https://validator.w3.org`

  Ceci est important car les navigateurs ne sont pas stricts et essaient d'interpréter tout code HTML, même mal formé. Or ignorer un petit problème est le meilleur moyen qu'il devienne gros plus tard.

2 Formulaires

Les formulaires permettent d'envoyer des requêtes à un serveur. Un formulaire est inclus dans la balise suivante

```
<form action="https://kiva.mobapp.minesparis.psl.eu/api/trace" method="...">
</form>
```

qui comporte deux attributs

- `action`, qui est l'URL de la requête ;
- `method` qui est le type HTTP de la requête (`GET` ou `POST` dans le cas des formulaires).

Un formulaire contient des champs d'entrée, balise <**input**>, qui est autofermante. Nous utiliserons le type texte.

```
<input type="text" name="param" value="valeur"/>
```

Outre l'attribut `type`, chaque champ doit avoir un nom (attribut `name`). L'attribut `value` est optionnel et contient la valeur par défaut du champ d'entrée.

Chaque balise <**input**> fournit un paramètre à la requête HTTP faite au serveur lors de la soumission du formulaire. Le paramètre est composé du couple `name/value` discuté plus haut. Ceux-ci sont intégrés soit sous une forme optionnelle (i.e., dans la barre d'adresse) dans le cas d'une requête de type GET ; soit dans un dictionnaire dans le cas d'une requête de type POST. Plus de détails sur ce sujet lors des cours sur la conception du serveur.

La soumission d'un formulaire se fait avec un champ particulier : un bouton de soumission.

```
<button type="submit" value="submit">Envoyer</button>
```

Exercices :

- implémenter une formulaire (très) basique, qui soumettra une requête à l'adresse `https://kiva.mobapp.minesparis.psl.eu/api/trace`. Tester les deux méthodes `GET` et `POST`.
- ajouter au moins un champ d'entrée de type texte : lors de la soumission du nouveau formulaire, l'adresse de la requête change-t-elle (attention, question piège) ? La réponse du serveur change-t-elle ?

3 L'API KiVa

`https://kiva.mobapp.minesparis.psl.eu` est un serveur que nous apprendrons à développer au début de l'année prochaine. En attendant, nous l'exploiterons en tant que "client".

Ce serveur implémente quelques opérations de manipulation d'une table d'associations clefs-valeurs (stockée dans une base de données) qui contient par défaut trois couples :

clef	valeur
calvin	hobbes
hello	world
kiva	cool

À partir de l'adresse de base `https://kiva.mobapp.minesparis.psl.eu`, il est possible de consulter cette table par une requête de type `GET` sur la route[1] `/api/store`.

En pratique, cela signifie qu'il vous suffit de taper dans la barre d'adresse : `https://kiva.mobapp.minesparis.psl.eu/api/store`

Toute l'API (sauf `/api/trace` et `/api/info`) est authentifiée, login `"kiva"` et mot de passe `"bien"`. Les routes accessibles sont décrites dans la table 1. Voici Deux exemples pour mieux comprendre, en supposant la base de données dans son état initial :

- une requête `GET` sur `https://kiva.mobapp.minesparis.psl.eu/api/store/calvin` renverra `"hobbes"`
- une requête `GET` sur `https://kiva.mobapp.minesparis.psl.eu/api/store?flt=%25l%25` renverra `[["calvin","hobbes"],["hello","world"]]`, car le filtrage aura été effectué, en SQL, avec `where key like %l%` (l'encodage du caractère spécial `%` se fait, dans la barre d'adresse, avec `%25`).

[1] ce terme sera expliqué l'année prochaine

Requête client			Réponse serveur		
Méthode	Route	Paramètres	Contenu	Format	Type
GET	/api/store	—	Liste clefs/valeurs	JSON	200
GET	/api/store	flt	Liste clefs/valeurs filtrée sur les clefs selon la valeur de flt	JSON	200
GET	/api/store/<key>	—	Valeur associée à <key>	JSON	200
POST	/api/store	key, val	Ajoute une nouvelle paire	—	201
PUT	/api/store/<key>	val	Modifie la paire associée à <key>	—	201
PATCH	/api/store/<key>	val	Modifie la paire associée à <key>	—	201
DELETE	/api/store/<key>	—	Supprime la paire associée à <key>	—	204

Table 1: Les routes de l'API kiva

Exercices :

- essayez à la main (i.e. dans la barre d'adresse de votre navigateur) ces requêtes, ainsi que quelques autres.
- implémentez un formulaire pour la requête GET de la première ligne de la table 1 des routes de l'API.
- modifiez ce formulaire pour pouvoir effectuer la requête GET de la deuxième ligne (celle avec le paramètre flt), en ajoutant un champ d'entrée input bien choisi dans votre formulaire.
  Assignez à ce champ d'entrée une valeur par défaut qui permette de sélectionner tous les couples.
- implémentez un second formulaire pour la requête POST sur /api/store (quatrième ligne).

La soumission de formulaire ne permet pas de faire des requêtes par une méthode autre que POST ou GET et encore, la troisième requête GET de la table 1 n'est pas accessible pour l'instant non plus.

De plus, le retour est un affichage en mode texte d'informations renvoyées par le serveur au format JSON. C'est plutôt rudimentaire. Pour ces raisons, nous allons rendre les pages web *dynamiques*.

4 Pages dynamiques avec Javascript

Javascript permet d'exécuter du code du côté du client, en l'occurrence dans votre navigateur. Ce dernier l'exécute lorsqu'il le rencontre (au chargement de la page), ou en réaction à un événement. C'est cette dernière manière qui va nous intéresser, elle exploite l'attribut onClick de la balise <button>.

Exercices :

- reprenez (dans un nouveau fichier html) votre/vos formulaire de la page précédente, et supprimez les balises <form>. Nous ne soumettrons plus les requêtes au serveur par le biais des formulaires.
- faites réagir le bouton de soumission à l'événement onClick, afin de lui faire exécuter le code alert('Hello, World'). Testez.
- on veut maintenant écrire du texte dans la page plutôt que d'avoir un popup. Pour ceci, il faut :
  1. une div, avec un attribut id ("identifiant"), qui a une valeur de votre choix, mais qui doit être unique ;
  2. en JS, accéder à cette div par le DOM, en appelant la fonction querySelector() sur l'objet document – fonction à laquelle on doit fournir l'identifiant de l'élément recherché précédé d'un #. Exemple, si on veut récupérer la , il faudra la rechercher comme ceci : document.querySelector('#cestmoi').[2]

     Cette objet a, parmi d'autres, un attribut innerHTML que l'on peut modifier, par exemple par l'affectation suivante : document.querySelector('#cestmoi').innerHTML = 'pnom'. En deux lignes :

```
const maDiv = document.querySelector('#cestmoi')
maDiv.innerHTML = 'pnom'
```

Testez.

[2]Cette manière de faire nous suffira amplement, mais on peut chercher un élément par de nombreuses manières. Voir la documentation en ligne de querySelector pour plus de détails.

- on veut maintenant incrémenter un compteur et en afficher la valeur dans la <**div**> ci-dessus. Il faudra, pour cela, déclarer une variable, ce qui est impossible dans l'attribut `onClick`.

  Vous pourrez par exemple déclarer et initialiser ce compteur dans l'entête (<**head**>) du fichier html, en mettant votre code à l'intérieur d'une balise

  ```
  <script type="text/javascript">
  </script>
  ```

  Ce code sera exécuté au chargement de la page.

- recupérez maintenant la valeur du champ d'entrée (balise <**input**>) lors du clic, puis affichez-le dans la <**div**>.

  Pour un élément du DOM (≈ une balise) de la catégorie `input`, la valeur contenue dans le champ est située dans l'attribut `value`. Ainsi, si votre champ d'entrée a comme id `"monChamp"`, sa valeur se récupère typiquement comme ceci:

  ```
  document.querySelector('#monChamp').value
  ```

- afin de limiter la taille du code écrit dans l'attribut `onClick`, mettez-le dans une fonction séparée, que vous mettrez dans la balise `<script>` ci-dessus.

- encore mieux, faites un fichier `code.js` séparé, qui vous inclurez avec l'attribut `src` de la balise `script`.

5 Table de Conversion Celsius-Fahrenheit

On rappelle les formules de conversion des degrés Celsius en degrés Fahrenheit :

$$X°\mathbf{C} = \frac{9}{5} * X + 32°\mathbf{F}$$

Affichez (dans votre page html) la table de conversion des degrés Celsius (de 0 à 100) en Fahrenheit.
Indications :

- un tableau, en HTML, se déclare comme un élément <**table**>, qui contient des <**tr**> (lignes) qui, elles-mêmes contiennent des <**td**> (cellules). Ainsi :

  ```
  <table id="fibleaunacci">
  <tr> <td>2</td> <td>3</td> <td>5</td> </tr>
  <tr> <td>8</td> <td>13</td> <td>21</td> </tr>
  </table>
  ```

  sera rendu ainsi

2	3	5
8	13	21

- évitez de *hardcoder* votre tableau (bravo aux courageuses et aux courageux), générez-le. La syntaxe la plus simple pour les boucles en JS est
  `for(let i=0; i<=100; i++){ /* du code */ }`

Exercice facultatif : faire, en plus, une calculette "à la volée" dans le style de `https://www.google.com/search?q=celsius+to+fahrenheit` :

- lancez la conversion par un clic sur un bouton "F → C" ou un bouton "C → F".

- (plus difficile) la même chose, mais sans bouton. Vous pourrez vous intéresser aux attributs/événements `onChange` et `onFocus`, voire à des événements encore plus spécifiques.

6 Requêtes HTTP en javascript

6.1 Principe

Plutôt que de visualiser la réponse du serveur à notre requête, on cherche maintenant à obtenir une page HTML qui intègre ces informations.

1. Solution 1 : côté serveur. Ce dernier inspecte la requête du client, génère et renvoie une jolie page. Bienvenu.e.s dans la programmation web du début des années 2000.
2. Solution 2 : côté client. Javascript s'occupe de faire la requête, puis de *modifier* les éléments de la page courante, grâce au DOM, aux informations renvoyées par le serveur, et à des librairies bien choisies (jQuery, Node, Angular, etc).

Nous allons explorer la seconde solution. Elle est par nature **asynchrone**, c'est à dire que l'utilisateur clique sur un bouton, puis il vacque à ses occupations sans attendre la réponse du serveur. Ainsi, le code JS exécuté lors du clic devra-t-il prévoir l'exécution d'une fonction *après que* le serveur aura répondu à la requête, et en attendant il rendra la main à l'utilisateur immédiatement.

Enregistrer une telle fonction de rappel (*callback*) au moment de l'envoi de la requête est typique de la programmation web AJAX (**A**synchronous **J**avascript **A**nd **X**ML).

6.2 Préliminaires

- Nous suggérons de mettre toutes vos fonctions dans l'en-tête, à l'intérieur d'une balise `<script>` (ou dans un fichier `.js`), et de ne mettre que le minimum de code JS dans la propriété `onClick` des boutons, voire ne rien y mettre du tout (cf. le cours).
- Écrire une fonction `function update(data)` qui prend des données (sous forme textuelle) et les écrit dans une `<div>`. Affecter cette fonction au clic sur un bouton, et la tester.

6.3 Mise à jour de la page par des requêtes au serveur

6.3.1 Requêtes avec axios

Nous allons utiliser la librairie axios, qu'il faut commencer par inclure dans l'en-tête de votre page (balise `<head>`) :

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Nous utiliserons uniquement l'appel à la fonction `axios`, qui prend un unique paramètre, un objet. Par exemple, l'appel suivant

```
axios({
    method: 'GET',
    baseURL: 'https://kiva.mobapp.minesparis.psl.eu/api/',
    url: '/store',
    params: {flt: '%l%'},
    headers: {Authorization: `Basic ${btoa('kiva:bien')}`},
})
```

configure en 5 points, puis soumet, une requête répondant au cahier des charges du serveur (voir la table 1). La requête est 1) de type `GET` ; 2) envoyée au serveur kiva ; 3) sur la route `/store` ; 4) avec un seul paramètre "get" (paramètre passé dans l'adresse de la requête, après le point d'interrogation `?`) qui filtrera tous les enregistrements de la base dont la clef contient un "l". Enfin, le serveur attend aussi un identifiant ("kiva") et un mot de passe ("bien"), qui sont 5) fournis en en-tête.

Comme vous avez pu le constater, les champs de l'objet argument de l'appel à `axios` configurent la requête. Les champs qui nous seront utiles sont :

- `method` : la méthode de la requête HTTP. Une chaîne de caractères (`'GET'`, `'POST'`...) correspondant à la première colonne de la table 1.
- `baseURL` : une chaîne de caractères, qui correspond à "l'adresse de base de l'API" (sa partie invariable).

- `url` : une chaîne de caractères, qui correspond aux différentes routes de l'API (sa partie variable, seconde colonne de la table 1).
- `params` : utilisé seulement pour les requêtes de type `GET`. Un objet de type dictionnaire `clef : valeur`. Il peut être vide ou absent si la route ou la requête ne nécessite pas de paramètre.
- `data` : utilisé pour tous les types de requêtes, *sauf* `GET`. Un objet de type dictionnaire similaire à l'objet `params` ci-dessus.
- `headers` : un objet, de type dictionnaire, qui contient les paramètres d'en-tête de la requête HTTP. En particulier, le champ `Authorization` permet de s'identifier pour accéder à certaines routes. Selon le type d'identification utilisé, le champ `Authorization` aura pour valeur :
  - `"Basic ..."`, où "`...`" doit être remplacé par un encodage en base 64 de la combinaison `login+":"+mdp`, pour nous ce sera `"kiva:bien"`.
  - `"Bearer ..."`, où "`...`" doit être remplacé par un token d'identification (voir la section 7).

Notez qu'on pourrait choisir la valeur `'https://kiva.mobapp.minesparis.psl.eu/'` pour le champ `baseURL`, au prix de devoir fixer le champ `url` à `'/api/store'`. Ce découpage entre "adresse de base" et "route" dépend en grande partie de vous et votre logique applicative plutôt d'un impératif physique.

Pour plus d'information, voir la documentation de axios, et en particulier `https://axios-http.com/fr/docs/req_config`.

6.3.2 Traitement de la réponse avec les fonctions de callback

La requête de la section précédente est certes soumise, mais rien n'y est fait pour traiter la réponse renvoyée par le serveur. Celle-ci est pour l'instant perdue. Il faut indiquer explicitement quelle est la fonction qui se chargera de traiter la réponse. Ce type de fonction est dit de *callback*.

L'appel à `axios` construit un objet de type *promesse*, auquel on peut fournir une fonction de callback avec la construction `.then`. La promesse se chargera d'appeler le callback ainsi enregistré en lui fournissant un objet de type "résultat de la requête" (contenant la réponse du serveur), lorsque cette requête aura abouti.

Cet objet-résultat a un certain nombre de champs, dont :

- `status` : un entier contenant le code de retour (`200`, `201`, `400`, `401`, `404`...)
- `data` : un objet correspondant aux données renvoyées. Axios s'occupe automatiquement de construire l'objet correspondant au format brut JSON reçu dans la réponse du serveur.[3]

Considérons, par exemple, la requête suivante :

```
axios({
    method: 'GET',
    baseURL: 'https://kiva.mobapp.minesparis.psl.eu/api/',
    url: '/store',
    params: {flt: '%l%'},
    headers: {Authorization: `Basic ${btoa('kiva:bien')}`},
})
.then(res => { alert(`status˜: ${res.status}`); update(res.data); })
.catch(err => { alert(err.message) })
```

Cette requête aura pour effet, en cas de succès, de :

- afficher un popup d'alerte annonçant le code de retour de la requête ;
- mettre à jour la <**div**> de la section 6.2 ci-dessus avec les données reçues (i.e., la liste de toutes les clefs-valeurs), par un appel à la fonction `update` que vous y avez programmé.

En cas d'échec, un popup contenant le message d'erreur s'affichera, grâce au second callback, enregistré dans **`catch`**.

[3] d'autres formats sont possibles, mais le serveur, y compris celui de vos projets, renverront quasi-systématiquement des informations au format JSON.

6.4 Exercices

- implémentez les requêtes correspondant aux routes de l'API de la table 1.
  - en utilisant des champs <input> pour fournir aux requêtes des paramètres, sur les routes qui en demandent.
  - Formatter correctement (en HTML) les informations reçues. Par exemple si le serveur renvoie une liste, afficher cette dernière dans un tableau HTML. Puis ajouter du style.
- Pour la requête `GET` avec paramètre `flt`, faire en sorte que, si le champ <input> correspondant est vide, la requête soumise soit faite *sans* le paramètre flt (plutôt qu'un paramètre vide).
- factoriser les constantes, par exemple l'adresse de base du serveur.
- nettoyez les couples clefs-valeurs impertinentes (et seulement ceux-ci) que vos camarades ont ajoutés depuis le début du TP.
- programmez une mise à jour régulière et automatique de la liste clefs-valeurs (avec la fonction javascript `setInterval()`)
- Ajoutez, à côté de chacune des entrées de cette liste (dans un tableau), un bouton qui permet de supprimer l'entrée en question.

7 Pour la fin : authentification par token (facultatif)

S'il vous reste encore du temps, passez à de l'authentification par token, qui est une manière standard de s'authentifier et qui sera utilisée lors du projet :

1. On fait une première requête `GET` sur la route `/api/login`.
2. le serveur renvoie un *token* (jeton), qui est une chaîne de caractères. On passe ensuite ce token dans l'en-tête de toutes les requêtes qui suivront. C'est à dire que, au lieu de configurer les en-têtes des requêtes par `"Authorization: Basic ..."`, il faut ajouter l'en-tête `"Authorization: Bearer "` (plus le token reçu).

L'authentification par login et mot de passe n'a ainsi lieu que lors de la première requête, jusqu'à expiration du token.

8 Ressources supplémentaires

Le site `https://www.w3schools.com` contient suffisamment de ressources sur tous les thèmes abordés dans ce TP et dans ce cours. Il en existe bien entendu une multitude d'autres, telles que openclassrooms, la documentation des API des librairies, etc.