

TD5

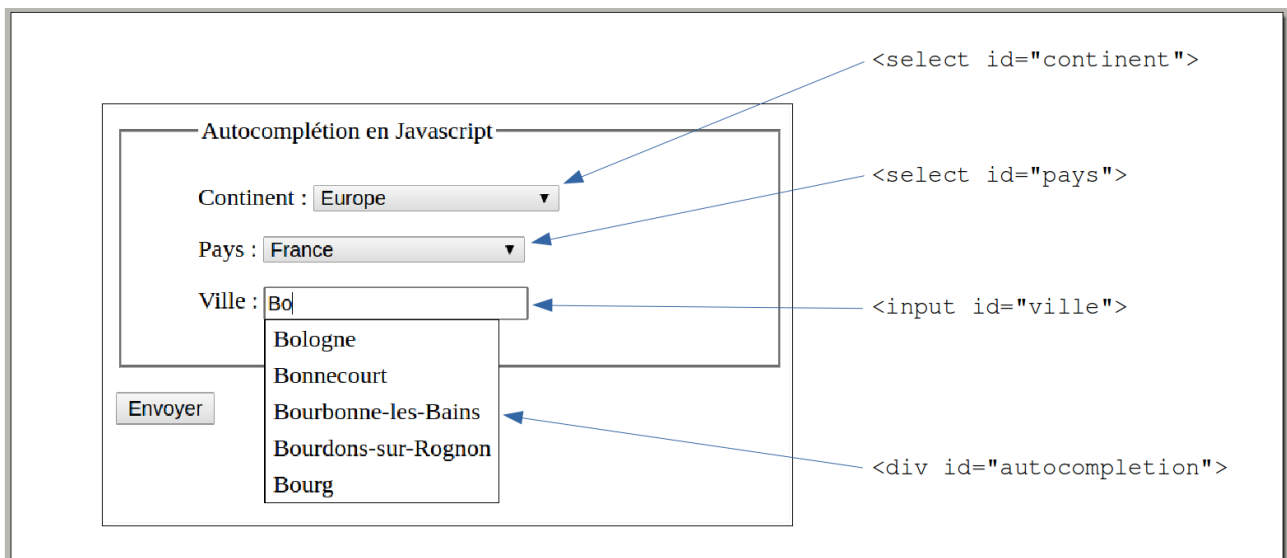
thème : asynchronisme en JavaScript, AJAX, formulaire autocomplétif

INTRODUCTION

Dans ce TD5, on reprend l'aspect asynchrone de JavaScript (abordé au TD4), qui permettra de modifier le contenu d'un élément d'autocomplétion, représenté par la balise html `<div id="autocompletion">`, à chaque modification du champ de saisie `<input id="ville">`.

Cet élément `<div id="autocompletion">` fournira alors une liste de villes dont le nom commence par les lettres insérées dans `<input id="ville">`.

Voir ci-dessous :



The screenshot shows a web form titled "Autocomplétion en Javascript". It contains three main input fields: "Continent : Europe" (a dropdown menu), "Pays : France" (a dropdown menu), and "Ville : Bo" (a text input). Below the "Ville" input, there is a list of suggestions: "Bologne", "Bonnecourt", "Bourbonne-les-Bains", "Bourdons-sur-Rognon", and "Bourg". An "Envoyer" button is located to the left of the suggestions list. Blue arrows point from HTML code snippets to the corresponding elements in the form: `<select id="continent">` points to the "Continent" dropdown, `<select id="pays">` points to the "Pays" dropdown, `<input id="ville">` points to the "Ville" text input, and `<div id="autocompletion">` points to the list of suggestions.

Le contenu cet élément `<div id="autocompletion">` est alimenté par le résultat (après traitement) d'une requête de type **AJAX** sur une base de données.

La fin du TD est consacrée à une fonctionnalité d'actualisation du `<select id="pays">` (sélecteur du pays) par changement de l'autre élément `<select id="continent">` (sélection du continent), ainsi qu'à diverses fonctionnalités complémentaires liées à la balise `<input id="ville">`.

Contrairement à la liste de villes (plus de 36000 entrées, ce qui justifie le stockage sur base de données), la liste de pays par continent est de taille raisonnable et sera gérée en local par JavaScript, par un fichier `countries.js` qui affecte une variable globale `countries`.

Vous mettrez en place un contexte web dans votre `public_html` avec les fichiers :

- `completion.css`,
- `loading.gif` qui sert plus tard,
- `scripts.js` à compléter,
- `countries.js` qui sert plus tard,
- la base de `requeteVille.php`,
- `Conf.php`, `Model.php`, `resultat.php`,
- `completion.html`

Le fichier `scripts.js` est à construire complètement.

EXERCICE 1 – Ébauche du fichier `scripts.js`

1. Dans le fichier `scripts.js`, codez la fonction `afficheVilles` qui prend en paramètre un tableau de villes comme

```
["Bordeaux", "Toulouse", "Montpellier", "Nice"]
```

et remplit la `<div id="autocompletion">` avec un paragraphe par nom de villes comme ci-dessous

```
<div id="autocompletion">
  <p>Bordeaux</p>
  <p>Toulouse</p>
  <p>Montpellier</p>
  <p>Nice</p>
</div>
```

Votre code utilisera obligatoirement la méthode `appendChild` pour chaque `<p>` à créer. Insérez votre fichier `scripts.js` dans le `html`, puis testez votre fonction avec le tableau ci-dessus en le déclarant dans une variable `tableau` et en lançant `afficheVilles(tableau)` dans la console;

```
let tableau = ["Bordeaux", "Toulouse", "Montpellier", "Nice"];
afficheVilles(tableau);
```

Faites un deuxième appel de la fonction avec le même tableau.

Vous devez constater qu'il peut être malin de commencer par vider le contenu de la balise `<div id="autocompletion">`. C'est ce que nous allons faire dans la question suivante.

2. Créez une fonction `videVille` qui vide l'élément `<div id="autocompletion">`. Cette fonction sera appelée par `afficheVilles`. Vous utiliserez deux approches différentes :

- une méthode qui utilise `removeChild` ;
- une méthode plus basique qui affecte tout simplement au `innerHTML` de la balise `<div id="autocompletion">` la valeur `" "`.

Remarque : on ne dit pas que les méthodes `appendChild` et `removeChild` sont meilleures qu'un "bricolage" du `innerHTML`, par contre elles sont plus dans la logique objet, et seront plus simples à utiliser si l'arborescence à ajouter/modifier se complique. Notre arborescence de `<p>` reste ici simple.

EXERCICE 2 – La page de requête `requeteVille.php`

Cette page côté serveur est déjà codée pour vous permettre de lancer une requête de type **SELECT** sur la base de données. Elle incorpore un fichier `Model.php` qui incorpore lui-même un fichier `Conf.php`.

Ce fichier `Model.php` vous propose une méthode `static selectByName` qui permettra de récupérer les 5 premières villes dont le nom commence comme la chaîne de caractères passée en paramètre à cette méthode (voir le code).

Les deux classes `Conf` et `Model` ont été abordées au S3 et vous n'avez pas à y toucher (sauf si vous voulez changer les paramètres de connexion pour utiliser votre propre base de données, avec le fichier `cities.sql` du dossier `src/sql`).

Vous n'interviendrez que sur quelques lignes du fichier `requeteVille.php`.

Ce fichier sera exécuté au moyen d'url du type `requeteVille.php?ville=Bo`

Le paramètre `ville` permettra d'utiliser `selectByName($name)`, avec la bonne valeur pour le paramètre `$name`. Par exemple, l'url

`http://webinfo.iutmontp.univ-montp2.fr/~monlogin/JS/td5-moi/src/php/requeteVille.php?ville=Bo`

permettra de lancer, par la fonction `selectByName`, la requête SQL suivante :

```
SELECT * FROM cities WHERE name LIKE 'Bo%' LIMIT 5
```

1. Complétez la page `requeteVille.php` pour qu'elle suive les étapes suivantes :

- extraire la ville passée en GET dans l'url ;
- appeler la fonction `selectByName` déjà codée et stocker le résultat dans une variable `$tab` ;
- produire un `echo json_encode` de cette variable. L'affichage produit se fera donc en format JSON facilement exploitable par JavaScript.

2. Testez ensuite le bon fonctionnement de la page en appelant des url du type `requeteVille.php?ville=Mo` ou `requeteVille.php?ville=Tou`. Vous devez voir dans le navigateur un affichage brut du résultat de la requête SQL.

```
[{"id": "388", "name": "Toussieux", "postal_code": "01600", "region_id": "22", "insee_id": "01423"}, {"id": "1147", "name": "Toulis-et-Attencourt", "postal_code": "02250", "region_id": "7", "insee_id": "02745"}, {"id": "1520", "name": "Toulon-sur-Allier", "postal_code": "03400", "region_id": "23", "insee_id": "03286"}, {"id": "2073", "name": "Toudon", "postal_code": "06830", "region_id": "25", "insee_id": "06141"}, {"id": "2074", "name": "Toulon-sur-Allier", "postal_code": "03400", "region_id": "23", "insee_id": "03286"}]
```

résultat de la requête `requeteVille.php?ville=Tou`

EXERCICE 3 – Requête asynchrone

Un peu de technique

Comme au TD4, nous allons utiliser un objet `XMLHttpRequest` qui permet de lancer des requêtes HTTP de manière asynchrone, c'est-à-dire sans bloquer la page web courante.

L'ensemble des technologies autour des pages web asynchrones s'appelle **AJAX** (Asynchronous Javascript And Xml).

Voici le squelette d'une requête **AJAX** :

```
function requeteAJAX(stringVille) {
    let url = "php/requeteVille.php?ville=" + stringVille;
    let requete = new XMLHttpRequest();
    requete.open("GET", url, true);
    requete.addEventListener("load", function () {
        console.log(requete);
    });
    requete.send(null);
}
```

La fonction `requeteAJAX` :

- gère un paramètre `stringVille` qui est une chaîne de caractères (ce sera celle qu'on écrira dans la balise `<input id="ville">`);
- crée une url pour `requeteVille.php`, construite à partir du paramètre `stringVille`;
- crée un objet `XMLHttpRequest` nommé `requete`;
- ouvre cette requête avec la méthode `open` qui donne le type de requête HTTP à effectuer (ici `GET`), l'URL de la page demandée et le troisième argument (`true`) signifie que la requête doit être asynchrone.
- met cet objet `requete` en écoute de l'événement `load`, ce qui signifie que l'objet `requete` attendra la fin du chargement des données commandées à la base de données, pour lancer la fonction déclarée de manière anonyme et dont la modeste mission est ici d'afficher l'objet `requete` dans la console).
- lance la requête par la méthode `send`. Le paramètre `null` est lié au fait que la méthode est `GET`. Si c'était `POST`, on aurait comme paramètre une chaîne de caractères annonçant les paires `nom=valeur`, c'est-à-dire ici `ville=...`

Le principe d'une requête asynchrone est qu'elle ne bloque pas l'exécution du JavaScript le temps que le serveur renvoie sa réponse.

La fonction anonyme, qui ne fait qu'afficher l'objet `requete` dans la console, est appelée fonction **callback**. Elle sera appelée lorsque le serveur aura retourné ses informations.

Elle a pour mission **le traitement de la réponse du serveur**.

Bien entendu, au final, cette fonction **callback** aura pour mission de remplir le contenu de la balise `<div id="autocompletion">`.

C'est ce que nous allons structurer, en plusieurs étapes. Comme nous allons construire plusieurs versions de la fonction `callback`, nous allons plutôt utiliser ce code plus générique :

```
function requeteAJAX(stringVille,callback) {
    let url = "php/requeteVille.php?ville=" + stringVille;
    let requete = new XMLHttpRequest();
    requete.open("GET", url, true);
    requete.addEventListener("load", function () {
        callback(requete);
    });
    requete.send(null);
}
```

Dans cette version de `'requeteAJAX'`, on passera en deuxième paramètre le nom de la fonction qu'on aura choisie pour jouer le rôle du callback. Ainsi, pour avoir l'équivalent du premier code, on pourrait avoir définir une fonction `callback_1` de la façon suivante :

```
function callback_1(req) {  
    console.log(req);  
}
```

Et on pourrait utiliser par exemple un appel `requeteAJAX("Bo", callback_1);`

écriture de quelques versions du callback

1. Complétez votre fichier `scripts.js` avec le code des deux cadres précédents. Placez la ligne d'insertion du script dans le head de la page.

Nouveauté : ajoutez l'attribut `defer` à `<script>` pour que le chargement du JS ne bloque pas la construction du DOM. Ceci revient à placer l'insertion du fichier avant la fin du body. La ligne d'insertion sera donc :

```
<script type="text/javascript" src="js/scripts.js" defer></script>
```

2. Rechargez la page `src/completion.html` et lancez dans la console la commande

```
requeteAJAX("Bo", callback_1);
```

Vous devez voir dans la console un descriptif complet de l'objet `requete`, avec notamment son attribut `responseText`.

3. Lancez d'autres commandes similaires en changeant le premier argument.
4. Écrivez une fonction `callback_2` qui, au lieu d'afficher dans la console l'objet XHR, comme le faisait `callback_1`, affichera un `JSON.parse` de son attribut `responseText`. Quel est l'effet de `JSON.parse` ?

5. Testez ce `callback_2` en console avec la commande

```
requeteAJAX("Bo", callback_2);
```

Vous devriez obtenir un résultat comme ci-dessous



```
>> requeteAJAX("Bo", callback_2);  
← undefined  
▼ (5) [...]  
  ▶ 0: Object { id: "45", name: "Boisse", postal_code: "01120", ... }  
  ▶ 1: Object { id: "46", name: "Boisse", postal_code: "01190", ... }  
  ▶ 2: Object { id: "47", name: "Bolozon", postal_code: "01450", ... }  
  ▶ 3: Object { id: "48", name: "Bouligneux", postal_code: "01330", ... }  
  ▶ 4: Object { id: "49", name: "Bourg-en-Bresse", postal_code: "01000", ... }  
    length: 5  
  ▶ <prototype>: Array []
```

6. Créez une fonction `callback_3` qui transforme encore le résultat précédent pour créer un tableau contenant l'attribut `name` de chacun des objets. Ainsi, quand on lance la commande `requeteAJAX("Bo", callback_3)` on doit obtenir dans la console :

```
>> requeteAJAX("Bo",callback_3);
< undefined
  ► Array(5) [ "Boisse", "Boissey", "Bolozon", "Bouligneux", "Bourg-en-Bresse" ]
```

7. Créez enfin le callback final `callback_4` qui produit le même tableau que `callback_3`, et qui affiche son contenu (par l'intermédiaire de la fonction `afficheVilles`) dans la balise `<div id="autocompletion">`.
8. Testez votre fonction `callback_4` en appelant `requeteAJAX` à partir de la console avec des chaînes de caractères diverses en premier paramètre.
9. Puisque `callback_4` est satisfaisante, c'est elle que nous adoptons. Créez maintenant, toujours dans le fichier `scripts.js`, une fonction `maRequeteAJAX` qui prend en paramètre une chaîne de caractères.

Grâce à cette fonction, l'instruction `maRequeteAJAX("Toul")` sera exactement équivalente à l'instruction `requeteAJAX("Toul", callback_4)`.

Exercice 4 – premiers gestionnaires d'événements

1. Munissez le champ `<input id="ville">` d'un écouteur d'événement, associé à l'événement `input` (qui est lancé à chaque modification du contenu d'un `<input>`). La fonction appelée sera déclarée de façon anonyme, et son action sera d'appeler la fonction `maRequeteAJAX`, avec comme paramètre la valeur de la balise `<input id="ville">`. Ainsi, chaque modification de ce champ met à jour le contenu de la balise `<div id="autocompletion">`. On y est presque...
2. Munissez la balise `<div id="autocompletion">` d'un écouteur d'événement, associé à l'événement `click`. Le clic sur un des paragraphes enfants de la balise aura le comportement intuitif attendu :
 - remplir `<input id="ville">` avec le contenu du paragraphe cliqué ;
 - vider `<div id="autocompletion">`.

Pour cela vous utiliserez `event.target` qui permet de savoir quel paragraphe est la cible de l'événement `click`.

A ce stade votre champ d'autocomplétion est opérationnel.

EXERCICE 5 – Les deux sélecteurs

Les deux sélecteurs `<select id="continent">` et `<select id="pays">` vont fonctionner indépendamment du champ d'autocomplétion. Le sélecteur de continents sera chargé dès le début, et le contenu du sélecteur de pays devra s'actualiser au changement de la valeur du sélecteur de continents. La liste des pays et des continents auxquels ils appartiennent se trouve dans le fichier `countries.js`.

Le sélecteur de continents

1. Insérez, au niveau du `head` de `completion.html`, le fichier `countries.js` qui permet d'accéder à la variable `countries`. Attention d'insérer ce fichier avant le précédent. Réutilisez l'attribut `defer`.
2. Créez, dans `scripts.js`, une fonction `chargerSelecteurContinents` basée sur `appendChild` et qui permet de structurer le sélecteur de continents en lui ajoutant des enfants `<option>...</option>`. Chacun de ces enfants aura pour `innerHTML` l'une des clés qu'on obtient par la méthode `Object.keys` appliquée à `countries`.

Vous aurez donc à utiliser le contenu de `Object.keys(countries)`.

Vous ajouterez un enfant de la forme

```
<option selected disabled>choisissez un continent</option>.
```

Il faudra pour cela agir sur les attributs `selected` et `disabled` de l'élément créé (les mettre à la valeur `true`). Testez cette fonction dans la console et vérifiez que le sélecteur de continents se remplit bien.

3. Faites en sorte que ce sélecteur se remplisse au chargement de la page. Pour cela, votre fonction `chargerSelecteurContinents` sera associée à l'événement `DOMContentLoaded` dans un écouteur d'événement de l'objet `document`.

Le sélecteur de pays

4. Lors d'un changement de valeur du sélecteur de continents, le sélecteur de pays doit proposer les pays du continent sélectionné.

Créez, dans `scripts.js`, une fonction `chargerSelecteurPays` qui permet de construire les fils de la balise `<select id="pays">`:

- récupérer la valeur du sélecteur de continents ;
- récupérer, dans `countries`, le tableau correspondant à cette valeur ;
- créer, pour chaque entrée du tableau, une ligne du sélecteur de pays au moyen de la méthode `appendChild` (assez similaire au sélecteur de continents).
- ne pas oublier la ligne `<option selected disabled>choisissez un pays</option>`

5. Testez votre fonction dans la console. Vous ferez bien attention au fait que le sélecteur de pays doit être réinitialisé à chaque fois.
6. Munissez le sélecteur de continents d'un écouteur d'événement pour que chaque changement de ce sélecteur lance la fonction `chargerSelecteurPays`.

EXERCICE 6 – Compléments divers

Détail css

Vous corrigerez un petit détail : Il y a un petit carré gris qui apparaît quand la balise d'autocomplétion `<div id="autocompletion">` est vide. C'est sa border de largeur 1px. Faites en sorte de corriger, au niveau du JavaScript, l'attribut `style.borderWidth` de cette `div` en fonction de son contenu, pour ne pas avoir ce défaut du petit carré gris.

Limitation de l'autocomplétion

Modifiez légèrement la fonction associée à l'événement `input` pour que l'auto-complétion n'opère que si le contenu du champ `Ville` contient au moins deux caractères.

Signal de chargement

Lorsqu'un chargement est en cours, nous pouvons le signaler à l'utilisateur pour qu'il patiente le temps nécessaire. Dans notre cas, nous afficherons le GIF de chargement `loading.gif` fourni dans l'archive `.zip` pendant le délai de réponse du serveur.

1. Modifiez `requeteAJAX` pour que la fonction prenne en paramètres supplémentaires deux fonctions `startLoadingAction` et `endLoadingAction`.
 - La première fonction sera exécutée dès le lancement de la requête.
 - La deuxième sera exécutée dès la réception de la réponse. On aura donc des appels de la forme

```
requeteAJAX("Bo", callback, action_debut, action_fin)
```

2. Dans la fonction `'maRequeteAJAX'`, modifiez l'appel à `'requeteAJAX'` pour ajouter deux nouveaux paramètres : ce seront deux fonctions déclarées en fonctions anonymes :

- la première, qui jouera le rôle de `startLoadingAction`, rendra visible le GIF de chargement ;
 - la deuxième, qui jouera le rôle de `endLoadingAction`, lui redonnera une visibilité `hidden`.
3. Pour que le comportement soit visible, truquez en ajoutant une temporisation de 1 seconde dans `requeteVille.php`

Note : l'instruction PHP : `sleep(1);`

Exercice 7 – Utilisation des touches ↓ , ↑ et ↵

Les touches haut et bas servent habituellement à se déplacer dans la liste des suggestions, et la touche **ENTER** à valider l'élément courant. L'utilisation de ces touches entraîne une mise à jour du champ texte où s'inscrit le nom de la ville. Programmez ces comportements.