TP n° 4 : Services Web et AJAX

Dans ce TP, on apprendra à réaliser des services Web en node.js, et l'on utilisera le paradigme AJAX pour la communication asynchrone entre le navigateur (javascript coté client) et le serveur Web qui fournit ces services.

I) Services Web et Json

Un service Web est une fonctionnalité proposée par un serveur et utilisable par d'autres programmes via le protocole HTTP, à la différence des pages Web habituellement destinées aux humains. Le programme transfère au serveur les paramètres d'*input* du service via HTTP et le serveur envoie sa réponse de la même manière.

Les programmes utilisent pour cela des formats standards de représentation des données, compréhensibles par d'autres programmes. Le format textuel de données XML a été introduit dans ce but ; aujourd'hui ce format est de plus en plus remplacé par le format JSON (JavaScript Object Notation), plus léger à transférer.

JSON est lui aussi un format textuel, très proche de la déclaration directe d'objets en JavaScript : un objet défini par `{name: "nom", age: 24}` deviendra, une fois transformé en JSON, la chaîne de caractères `'{"name": "nom", "age": 24}'` (les guillemets autour des noms de propriétés sont optionnels en Javascript, mais imposés en JSON). Un tableau `[42, "abc"]` sera transformé en `'[42, "abc"]'`, etc.

Les objets Javascript (à l'exception de certains objets présentant une structure cyclique) sont donc facilement sérialisés, c.-à-d. représentés de manière portable dans le format JSON, pour pouvoir être transférés et ensuite reconstruits par le programme qui les reçoit.

JavaScript possède les fonctions `JSON.stringify(`*Object*`)` pour transformer un objet en sa représentation JSON, et `JSON.parse(`*string*`)` pour transformer une chaîne JSON en un objet JavaScript. Cependant la plupart des fonctions qui opèrent sur le format JSON effectuent une conversion automatique ; il sera donc peu fréquent de devoir utiliser explicitement ces fonctions.

1. **Service Web dictionnaire** Le but de cet exercice est de mettre au point un service Web de dictionnaire permettant de :
   - rechercher dans le dictionnaire les mots commençant par un préfixe donné ;
   - ajouter des mots au dictionnaire ;
   - renvoyer l'ensemble des mots du dictionnaire.

   a. Commencez par définir dans un fichier `Dico.js` un constructeur Javascript `Dico()` pour la création d'un objet dictionnaire. Les mots d'un dictionnaire seront stockés dans un simple tableau trié de chaînes. Un dictionnaire devra être muni des méthodes suivantes (*c.f.* les méthodes de `Array.prototype`, invocables sur tout tableau) :
   - `search(word)` renvoyant un booléen indiquant si le dictionnaire contient ou non `word` (par une boucle, ou mieux, en examinant la valeur de retour de `findIndex` invoquée avec le prédicat approprié).
   - `insert(word)` insérant un mot dans le dictionnaire s'il ne s'y trouve pas déjà (`search`, `push`, `sort`).
   - `words()` renvoyant le contenu courant du dictionnaire.
   - `prefixSearch(query)` renvoyant un tableau contenant tous les mots du dictionnaire dont `query` est un préfixe (par une boucle ou par `filter` avec le prédicat approprié, en se servant aussi de la méthode `String.prototype.search` et d'expressions régulières).

   Terminez votre fichier par l'instruction suivante :

   ```
   module.exports = new Dico();
   ```

   Elle vous permettra, dans le second fichier à écrire, d'importer l'objet construit par `new` via l'instruction suivante :

```
const dico = require('./Dico');
```

2. Écrire à présent un serveur Node, ayant le comportement spécifié ci-dessous. Commencez par récupérer l'objet `dico` créer dans `Dico.js`, et remplissez manuellement ce dictionnaire par quelques mots, dont certains avec un prefixe commun.

(i) Le serveur doit pouvoir recevoir et émettre des données JSON. Il sera créé de la manière suivante [1]

```
const express = require("express");
const server = new express();
server.use(express.json());
server.use(express.urlencoded({extended:false}));
```

où `express.json()` est un middleware invoquant `JSON.parse` sur le corps d'une requête HTML, puis transmettant la requête au middleware suivant.

(ii) Dans les routes en `GET` d'URL à paramètres (`?word=abc`), `req.query` est un objet dont les noms de propriétés sont les noms des paramètres, et les valeurs de ces propriétés les valeurs des paramètres sous forme de chaînes (`{word='abc'}`).

(iii) Dans les routes en `POST` dont le corps contient une chaîne JSON (`'{"word" : "abc"}'`), `req.body` est une référence vers l'objet résultant du parsing de cette chaîne (`{word='abc'}`).

(iv) La réponse du serveur doit être émise via `res.json(`*réponse*`)` [2].

Voici le comportement attendu du serveur :

a. Une requête en `GET` sur `/dictionary` renvoie, au format JSON, l'ensemble des mots du dictionnaire.

b. Une requête en `GET` sur `/dictionary/search` avec un paramètre `word` renvoie, au format JSON, les mots du dictionnaire dont la valeur de `word` est un préfixe.

c. Une requête en `POST` sur `/dictionary`, dont le corps contient une chaîne JSON de la forme `'{"word":` *chaîne*`}'` insère *chaîne* dans le dictionnaire si elle ne s'y trouve pas déjà. La réponse du serveur sera dans ce cas un simple message de reussite ou d'erreur.

Pour les requêtes en `GET`, il est très facile de simuler l'usage du service à l'aide d'un navigateur (`http://localhost:8080/dictionary/search?word=ab`). On peut aussi simuler l'usage du service par l'envoi explicite de requêtes depuis le terminal à l'aide du la commande `curl` :

- `curl -X GET localhost:8080/dictionary/search?word=abc`
- `curl -X POST -H 'Content-Type:application/json' -d '{"word": "abc"}' localhost:8080/dictionary`

1. Ce middleware est utilisable à condition que votre version d'`express` soit suffisamment récente. C'est le cas si la commande `npm view express version` renvoie un numéro de version au moins égal à `4.16.0`. Dans le cas contraire, il faudra commencer par installer dans votre répértoire de travail le module `body-express` avec `npm`, puis initialiser le serveur par :

```
const express = require("express");
const bodyParser = require("body-parser");
const server = new express();
server.use(bodyParser.json());
server.use(bodyParser.urlencoded(extended:false));
```

2. L'usage de `send` aurait le même effet que celui de `json` pour ce seul énoncé, mais en règle générale, l'usage de la méthode `json` offre des possiblités de formattage des données JSON émises que ne permet pas `send`.

II) Appel aux services avec AJAX

Un programme qui a besoin d'utiliser les service fournis par un serveur doit effectuer des requêtes de `GET` et/ou `POST` correspondant aux services demandés. Nous avons vu que, côté client, Javascript permet de modifier le DOM à la volée, afin d'obtenir des pages réactives et dynamiques.

Il se trouve que Javascript permet aussi d'effectuer des requêtes HTTP afin d'interagir avec des services, et ce, sans recharger la page. AJAX[3] est le nom donné à l'ensemble des outils javascript permettant de réaliser cela. Le concept d'AJAX est d'effectuer ces requêtes de manière asynchrone, afin de ne pas bloquer la page en attendant que le serveur réponde (ou non) à ces requêtes.

Les outils nécessaires sont évidemment disponibles dans le langage de base, mais leur interface étant complexe, on préfèrera utiliser la surcouche ajax de jQuery. Dans jQuery, la méthode `$.ajax` permet d'effectuer des requêtes HTTP. Cependant, les deux méthodes suivantes, qui sont des surcouches de `$.ajax`, sont beaucoup plus simples d'utilisation :

- `$.get(url, params, callback)`

```
$.get("http://localhost:8080/dictionary/search", {word : "abc"},
    function(data) {
        console.log(data); // tableau des mots de prefixe "abc"
    });
```

- `$.post(url, params, callback)`

```
$.post("http://localhost:8080/dictionary", {word : "abc"},
      function(data) {
          console.log(data); // tableau de tous les mots apres ajout.
      });
```

Le premier paramètre est une URL (relative ou absolue). Le second `params` représente les paramètres de la requête, il s'agit soit d'une chaîne, soit d'un objet "pur" (un ensemble de couples de noms de propriétés et de valeurs). Le troisième `callback` est une fonction de la forme `function (data) .....` Cette fonction sera appelée avec en argument `data` les données renvoyées par le serveur sous la forme d'un objet pur lorsque le serveur aura renvoyé une réponse.

Autocomplétion Le but de cet exercice est d'utiliser le service de dictionnaire précédent afin de proposer l'auto-complétion d'un champ de texte sur votre page. Il vous faudra créer un répertoire `public` dans lequel vous placerez une page web ainsi qu'un script lié à cette page (il faudra aussi lier la page au script de jQuery, comme dans le TP 2). L'ajout de ce répertoire dans le domaine du serveur peut se faire par :

```
server.use(express.static('public'));
```

L'envoi d'une page web de ce répertoire peut se faire sur le modèle suivant :

```
res.sendFile("dico.html", {root: 'public'});
```

3. Ajoutez au serveur une route `GET "/"` renvoyant une page web contenant seulement un champ de texte nommé "query" (il est inutile de l'inclure à un formulaire), un bouton `Ajouter` ainsi qu'un `div` ayant pour identifiant "results".

3. Acronyme de Asynchronous Javascript And XML. Contrairement à ce que son nom suggère, AJAX permet également de travailler avec JSON.

4. Complétez le script lié à la page de manière à obtenir le comportement suivant : à chaque fois que l'utilisateur modifie le contenu du champ de texte (`.on('input', ...)`), une requête `GET` sur est envoyée sur `/dictionary/search` avec comme paramètre le contenu courant du champ texte (`.val()`). Le résultat est alors placé dans le `div` "results", sous la forme d'une liste html contenant chaque mot de la réponse).
5. Ajoutez un bouton "Ajouter" qui ajoute le contenu du champ de texte au dictionnaire (requête `POST` sur `/dictionary`). Le `div` "results" devra bien sûr être mis à jour.
6. À l'aide de CSS si nécessaire, faites en sorte que la liste des suggestions s'affiche juste en dessous du champ de texte, comme un menu déroulant.
7. Faites en sorte que lorsque l'utilisateur clique sur un mot suggéré, celui-ci soit affiché dans le champ de texte et que le menu disparaisse [4].

III) Prefix tree (bonus)

Améliorez votre structure de dictionnaire pour qu'elle supporte efficacement l'operation de recherche de préfixe. Un type abstrait adapté à ce genre de recherche est l'arbre de préfixes (*prefix tree* ou *trie* en anglais).

Un arbre de préfixes est simplement un arbre enraciné, dont chaque sommet peut être marqué ou non, et dont chaque arête est indexée par une lettre de l'alphabet. Il possède de plus la propriété qu'il existe un chemin étiqueté $(a_1, \ldots, a_p)$ de la racine à un sommet marqué si et seulement si le dictionnaire contient le mot $a_1 a_2 \ldots a_p$.

8. Après avoir pris soin de spécifier le type abstrait de données pour représenter un sommet, créez un constructeur Vertex pour créer des sommets de l'arbre. (Indice : on pourra utiliser un tableau à 26 éléments pour stocker les éventuels sommets fils, et utiliser la méthode `charCodeAt()` de `String` pour obtenir le code ASCII d'un caractère).
9. Créez un constructeur `Trie()` pour créer des arbres de préfixes. Implementez les méthodes `insert`, `search`, `list`, et `tt prefixSearch`.
10. Insérez une liste de mots, et assurez-vous que vos méthodes fonctionnent correctement.

4. Rappel : on peut construire un objet jQuery représentant un élément du DOM en fournissant explicitement son contenu HTML à la fonction `$( )` sous forme de chaîne de caractères. Par ailleurs, lorsque l'on fournit un handler à la méthode `click` d'un tel objet, `this` dans ce handler désigne l'objet lui-même.