TP 3 - *JS*

Ce *TP* a pour objectif de réaliser un moteur de recherche des communes de France. Pour cela, nous allons nous appuyer sur l'*API* `geo.api.gouv.fr`, remarquable par sa simplicité d'utilisation.

Une partie de l'application, notamment tout ce qui relève de la mise en page et de la navigation a déjà été réalisée (fichiers `index.html` et `index.css`). Mais pour fonctionner pleinement, celle-ci requiert une bibliothèque de fonctions qui n'a malheureusement pas encore été développée ! Il vous est donc demandé d'écrire le code *JS* de cette bibliothèque au sein du fichier `index.js`. Il s'agit du seul fichier à modifier et vous n'avez pas besoin d'étudier le code *JS* du fichier `index.html` pour avancer. Sachez juste qu'il se contentera d'appeler vos fonctions au bon moment.

Pour la majeure partie du *TP*, il vous faudra trouver la documentation par vous-même. Le site *MDN Web Docs* est une très bonne ressource pour cela. Mais bien sûr, comme toujours, n'hésitez pas non plus à poser des questions ! Vous êtes encouragés à utiliser la console du navigateur pour déboguer votre code.

I. Découverte

Nous allons commencer par prendre un peu en main le moteur de recherche actuel. En l'état, il s'agit simplement d'un formulaire qui permet de faire une recherche sur un nom de commune. À son envoi, une requête `get` est faite à l'API distante et l'utilisateur est redirigé vers la page donnant le résultat de la recherche.

1. Essayez de faire quelques recherches via le formulaire. Quel format de données vous est renvoyé ?

2. Observez la structure des données renvoyées. Quelle régularité pouvez-vous y déceler ?

3. Regardez l'*URL* de la requête. Voyez-vous la relation entre celle-ci et le contenu de la réponse ? Vous pouvez trouver la documentation de l'API à l'adresse `https://geo.api.gouv.fr/decoupage-administratif/communes`.

Le moteur de recherche actuel n'est pas très orienté utilisateur : le format de données de la réponse n'est pas forcément lisible par un utilisateur lambda et on est obligé d'utiliser l'historique du navigateur pour retrouver la page du moteur de recherche. Il serait bien plus pratique de rester sur la page et d'y afficher directement le résultat dans un format plus agréable à lire : un tableau de données ! C'est le comportement attendu de l'application une fois que vous aurez assez avancé dans le développement de la bibliothèque.

II. Initialisation

Cette partie consiste à écrire le code des fonctions `initForm` et `initTable`.

1. Étudiez la structure du document *HTML*. Repérez notamment où se trouve l'élément `<form></form>`. Que représente cet élément ?

2. Écrivez la fonction `initForm` qui récupère l'élément `<form></form>` en question et le retourne.

3. À l'aide de la documentation des classes `HTMLTableElement`, `HTMLTableSectionElement` et `HTMLTableRowElement`, écrivez la fonction `initTable` qui construit un tableau de données vide. Plus précisément, cette fonction :

   - crée un tableau (élément `<table></table>`) ;
   - l'ajoute à l'intérieur de l'élément `<main></main>` ;
   - ajoute une légende au tableau (élément `<caption></caption>`), vide pour l'instant ;
   - ajoute un en-tête au tableau (élément `<thead></thead>`) contenant une ligne (élément `<tr></tr>`) composée elle-même de plusieurs cellules d'en-tête (éléments `<th></th>`) dont les intitulés respectifs sont donnés dans la constante `headers` ;
   - ajoute un corps de tableau (élément `<tbody></tbody>`) vide pour l'instant ;
   - retourne le tableau ainsi construit.

III. Manipulation d'*URL*s

Cette partie consiste à écrire le code des fonctions `getSearchParams`, `getLocalURL` et `getRemoteURL`.

1. Écrivez la fonction `getSearchParams` qui renvoie un objet `URLSearchParams` contenant les données d'un formulaire donné, en passant par un objet `FormData` intermédiaire.

2. Écrivez la fonction `getLocalURL` qui détermine l'*URL* courante du document, lui applique les paramètres donnés et la retourne sous forme d'objet `URL`. Une manière de faire est d'utiliser la variable `location`.

3. Retrouvez dans le document *HTML* l'*URL* à laquelle sont envoyées les

données du formulaire.

4. Écrivez alors la fonction `getRemoteURL` qui retourne cette *URL* sous forme d'objet *URL*, après lui avoir appliqué les paramètres donnés.

IV. Changement d'état

L'état de notre application est caractérisé par un objet de la forme `{input: …, output: …}` qui représente la dernière recherche effectuée. Cet état est donc amené à changer au fur et à mesure de vos interactions avec le moteur de recherche. Cette partie consiste à écrire le code des fonctions `computeInput` et `computeOutput` qui calculent les valeurs des champs correspondants et par extension l'état de l'application.

1. Écrivez la fonction `computeInput` qui récupère la valeur du champ de recherche nommé `"nom"` à partir des paramètres donnés et la retourne.

2. Écrivez la fonction `computeOutput` qui soumet le formulaire manuellement. Plus précisément, cette fonction :

   - envoie une requête à l'API à l'adresse donnée par votre fonction `getRemoteURL` en utilisant la fonction `fetch` ;

   - récupère, analyse puis retourne la réponse sous forme de tableau *JS*.

   N'oubliez pas les mots-clés `await` ! Il sont nécessaires lorsque l'on appelle une fonction asynchrone qui demande généralement du temps pour être exécutée (comme effectuer une requête ou analyser une réponse).

V. Débogage

À ce point du *TP*, nous avons lu le contenu du formulaire, envoyé ces données à l'API et récupéré le résultat de la recherche. L'application a donc changé d'état et il faut maintenant refléter ce changement sur le document lui-même, c'est-à-dire tout simplement afficher le résultat de la recherche. Mais avant cela, vous allez pouvoir un peu plus expérimenter la console du navigateur en y affichant le résultat de manière structurée.

1. Écrivez la fonction `logState` qui :

   - ouvre dans la console un groupe réduit étiquetté par le champ `input` de l'état donné ;

   - affiche dans la console le champ `output` de l'état donné sous forme de tableau avec pour noms de colonnes uniquement `"nom"`, `"codeDepartement"` et `"codeRegion"` ;

- ferme le groupe.

VI. Mise à jour du tableau de données

Nous allons afficher le résultat dans le tableau que vous avez créé au début du *TP*. Cette partie consiste à écrire le code des fonctions `renderCaption` et `renderBody` .

1. À l'aide de la documentation de la classe `HTMLTableElement`, écrivez la fonction `renderCaption` qui met à jour la légende du tableau donné pour qu'elle indique combien de correspondances ont été trouvées et qu'elle rappelle quelle recherche a été effectuée.

2. À l'aide de la documentation des classes `HTMLTableElement`, `HTMLTableSectionElement` et `HTMLTableRowElement`, écrivez la fonction `renderBody` qui met à jour le corps du tableau donné avec le résultat donné. L'ordre des colonnes est donné par la constante `keys`.

VII. Mise à jour du formulaire

Vous avez peut-être remarqué que si vous naviguez dans l'historique du navigateur ou bien si vous rechargez la page, alors le contenu du champ de recherche devient incohérent par rapport au tableau affiché ou bien se vide. Pour pallier ce comportement, il est possible de mettre à jour nous-mêmes le champ de recherche à chaque changement d'état de l'application. Cette partie consiste à écrire le code de la fonction `renderField` et à améliorer celui de la fonction `initForm`.

1. À l'aide de la documentation de la classe `HTMLFormElement`, écrivez la fonction `renderField` qui met à jour le contenu du champ de recherche avec la recherche effectuée.

2. À l'aide de vos fonctions `computeInput` et `renderField`, améliorez la fonction `initForm` pour qu'elle auto-complète le champ de recherche avec la valeur du paramètre `"nom"` de l'*URL* initiale du document (uniquement si un tel paramètre existe bien sûr). Une manière de faire est d'utiliser la variable `location`.

VIII. Tri du tableau

Votre application devrait être pleinement fonctionnelle désormais ! Vous pourriez vous arrêtez là, mais on va rajouter une fonctionnalité bonus : le tri. Le principe est de pouvoir ordonner le tableau de données selon les valeurs d'une certaine colonne rien qu'en cliquant sur sa cellule d'en-tête. Cette partie consiste à améliorer le code de la fonction `initTable` et à écrire celui de la

fonction `sortState`.

1. Améliorez la fonction `initTable` pour qu'elle ajoute un attribut `tabindex="0"` à chaque élément `<th></th>`.

2. À l'aide la fonction `….sort()`, écrivez la fonction `sortState` qui ordonne le résultat. Plus précisément :

   - si le numéro de colonne donné est positif, alors l'ordre doit être croissant selon les valeurs de la colonne correspondante ;

   - sinon l'ordre doit être décroissant selon les valeurs de la colonne correspondant au complément à 1 du numéro de colonne donné ;

   - si le type de données (indiqué par la constante `types`) de la colonne est `"number"`, le tri doit être fait selon l'ordre naturel des nombres ;

   - sinon si le type de données de la colonne est `"string"`, le tri doit être fait selon l'ordre lexicographique du français.