Promesses et structures de contrôle async/await en javascript

Thomas Schatz et Benoit Favre

généré le 8 février 2022

Exercice 1 : Saisie

La fonction javascript `prompt()` demande de manière synchrone à l'utilisateur d'entrer une chaîne de caractères et la renvoie (c'est une fonction bloquante, rien d'autre ne peut se passer sur la page tant qu'on a pas répondu). Dans cette exercice on va écrire une page permettant de récupérer des entrées auprès de l'utilisateur de manière asynchrone (non bloquante).

Le code javascript ci-dessous demande à l'utilisateur d'entrer cinq nombres et affiche l'opération et son résultat.

```
function sum_5_numbers() {
  let j;
  let sum = 0;
  let sum_display = "";

  const div = document.querySelector("#calcul");

  for(let i = 0; i < 5; i++) {
    j = parseInt(prompt("Entrez un nombre: "));
    if (sum_display != "") {
     sum_display += "+";
    }
    sum_display = sum_display + j;
    div.innerText = sum_display;
    sum += j;
  }
  sum_display += "=" + sum;
  div.innerText = sum_display;
}
```

1. Réécrivez ce code de manière asynchrone, en utilisant une balise `input` de type `text` et un bouton `next` à la place de `prompt`. Pour cette première version, on ajoutera un moniteur pour les évènements de type `'click'` sur le bouton `next` qui appelera une fonction de rappel `update_sum` appropriée. Faites en sorte que le processus soit réinitialisé si l'utilisateur entre un sixième nombre.
2. Modifiez votre code pour qu'il utiliser des promesses et les structures de contrôle `async` et `await`. Plus spécifiquement, vous commencerez par écrire une fonction `get_next_user_number` qui renverra une promesse qui se résoudra dès que l'utilisateur soumettra un nombre en cliquant sur `next`. Vous Appelerez ensuite cette fonction avec le mot-clé `await` dans le corps d'une boucle infinie située dans une fonction définie avec le mot clé `async`. Le corps de cette boucle alternera entre récupérer un nombre avec `get_next_user_number` et mettre à jour l'affichage avec une version de `update_sum` modifiée de manière appropriée.

3. Que fait le code ci-dessous ? Intégrez ce code avec le votre en créant une fonction `main` qui appellera à la fois `blink_text('Magic!');` et `sum_5_numbers();`. Vérifiez que les deux composants de votre page web fonctionnent en parallèle sans se bloquer l'un l'autre, malgré la présence d'une boucle infinie dans chaque.

```
async function blink_text (text) {

  let d = document.createElement("div");
  d.innerText = text;
  d.style.display = 'block';
  document.querySelector('body').appendChild(d);
  let div_visible = true;

  while (true) {
    await one_second();
    if (div_visible === true) {
      div_visible = false;
      d.style.display = 'none';
    } else {
      div_visible = true;
      d.style.display = 'block'
    }
  }

  function one_second() {
    let p = new Promise( (accept_callback, reject_callback) => {
      setInterval(accept_callback, 1000);
    });
    return p;
  }
}
```

Exercice 2 : Paris

On va écrire un programme de paris utilisant la programmation par évènement javascript.

La page web contiendra deux boutons intitulés « pilule rouge » et « pilule bleue » d'identifiant respectif `red` et `blue` et une div d'identifiant `gains` qui affichera les gain de l'utilisateur. On considèrera qu'un caramel (« toffee » en anglais) est en jeu à chaque pari.

Dans l'archive .zip jointe au TD vous avez un exemple de solution qui utilise des fonctions de rappels. Dans cette solution, au chargement de la page, l'ordinateur génère un nombre aléatoire `p` avec `Math.random()` qui reste fixé par la suite. Ensuite, une fonction `initiate_new_bet` est appelée. Elle génère un second nombre aléatoire avec `Math.random()` et la place dans une variable `n` qui est visible depuis la fonction `initiate_new_bet`. Si `n` est inférieur à `p` la pilule rouge est gagnante pour cet essai, sinon c'est la bleue.

Un moniteur d'évènement `'click'` est ajouté avec la méthode `addEventListener` aux éléments `red` et `blue` pour réagir correctement lorsque l'utilisateur clique sur l'un des boutons. Si l'utilisateur a choisi la réponse gagnante, la fonction `gain_toffee` est appelée sinon la fonction `lose_toffee` est appelée. Les fonctions `lose_toffee` et `gain_toffee` terminent en ré-appelant `initiate_new_bet` pour lancer un nouveau pari. Remarquez que les moniteurs d'évènement utilisent la variable `n` afin d'être mis à jour automatiquement lorsque le contenu de cette variable change (quand on appelle `initiate_new_bet`).

Cette approche par fonctions de rappel n'est pas forcément très facile à comprendre. Dans cet exercice nous allons chercher une solution plus simple et élégante en utilisant des promesses et les structures de contrôle `async` et `await`. Plus spécifiquement, vous utiliserez une fonction `get_pill_color_from_user` qui renverra la couleur du bouton sur lequel l'utilisateur a cliqué sous la forme d'une promesse et qui sera appelée depuis une fonction asynchrone en utilisant le mot clé `await`. Cette fonction asynchrone contiendra une boucle infinie dont le corps fera séquentiellement un nouveau tirage aléatoire pour `n` un appel à `get_pill_color_from_user` et un appel à une fonction `resolve_gains` qui s'occupera de déterminer si l'utilisateur a gagné ou perdu et mettra jour l'affichage en conséquence.

Exercice 3 : Assemblage de composants asynchrones

Rassemblez l'afficheur de somme et le texte clignotant `Magic!\verb` de l'exercice 1 et le logiciel de paris de l'exercice 2 sur une même page en modifiant le moins possible leur code et vérifiez que les différents services fonctionnent en parallèles sans interférer les uns avec les autres.