JAVASCRIPT

Contenu

[I. Javascript 3](#_Toc35354367)

[A. Introduction 3](#_Toc35354368)

[B. Les variables 3](#_Toc35354369)

[C. Condition Switch cases et ternaires 5](#_Toc35354370)

[1. If, else if 5](#_Toc35354371)

[2. Switch/case 5](#_Toc35354372)

[3. Conditions ternaires 5](#_Toc35354373)

[D. Boucle While et for 5](#_Toc35354374)

[1. While 5](#_Toc35354375)

[2. For 6](#_Toc35354376)

[E. Les erreurs 7](#_Toc35354377)

[F. Les objects 7](#_Toc35354378)

[1. JSON : JavaScript Object Notation 7](#_Toc35354379)

[2. Les classes 8](#_Toc35354380)

[G. Les fonctions 10](#_Toc35354381)

[1. Ecritures 10](#_Toc35354382)

[2. Les tests 11](#_Toc35354383)

[H. La récursivité 12](#_Toc35354384)

[1. Qu’est-ce que la récursivité ? 12](#_Toc35354385)

[I. LE DOM 13](#_Toc35354386)

[1. Qu’est-ce que le DOM ? 13](#_Toc35354387)

[2. L’object document 13](#_Toc35354388)

[*3.* *Les évènements* 15](#_Toc35354389)

# Javascript

## Introduction

## Les variables

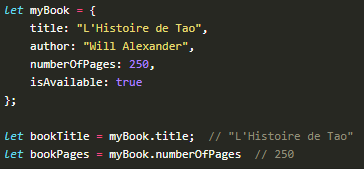
Pour déclarer une variable il suffit de mettre **let** ou **var** + leNomDeLaVariable, exemple :



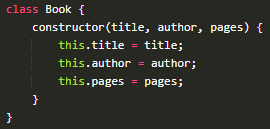
Les constantes déclarées par **const + leNomDeLaVariable** ne peuvent être modifié.

En javascript il y a 3 type de variables primitifs : string nombre et booléen.

Pour définir un objet et accéder à ses données  :



Pour créer une classe :



Instancier un objet de type Book :

Créer un tableau vide :

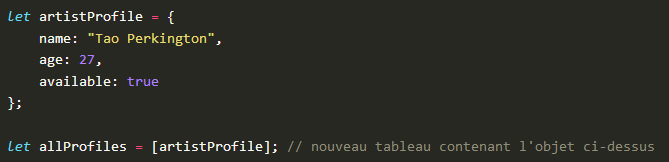
Créer un tableaux remplis :



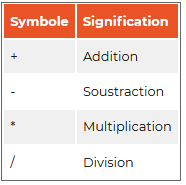




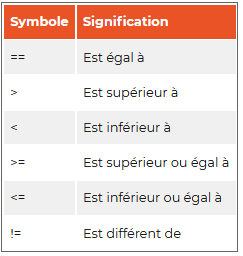
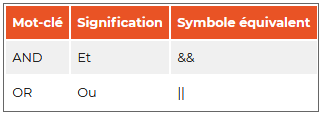
guest.lenght = longueur du tableau



Les calculs

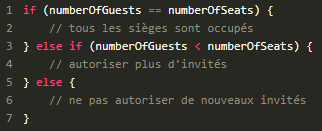
 

Les symboles de comparaison et conditions multiples

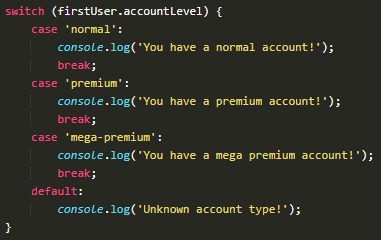
 

## Condition Switch cases et ternaires

### If, else if



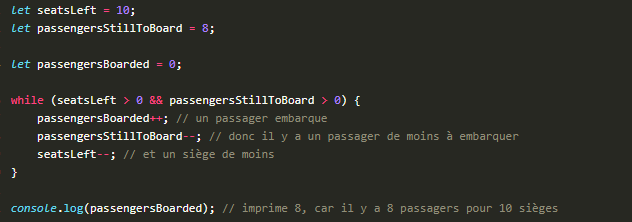
### Switch/case



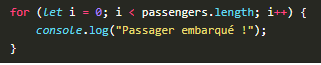
### Conditions ternaires

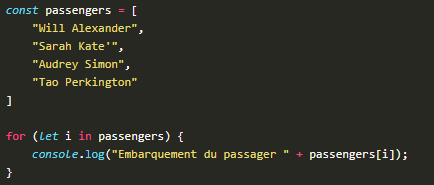
## Boucle While et for

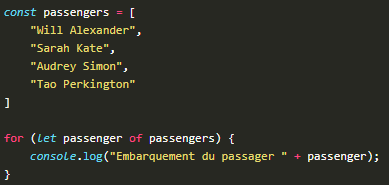
### While

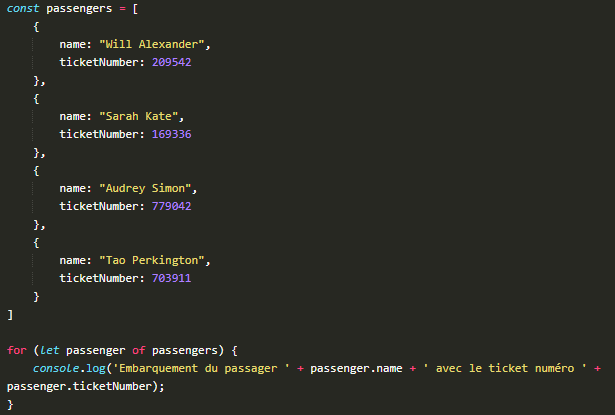


### For

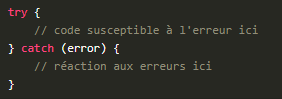








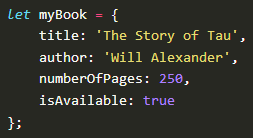
## Les erreurs



## Les objects

### JSON : JavaScript Object Notation

Les objets JavaScript sont écrits en **JSON** (**J**ava**S**cript **O**bject **N**otation). Ce sont des séries de paires clés/valeurs séparées par des virgules, entre des accolades. Les objets peuvent être enregistrés dans une variable :

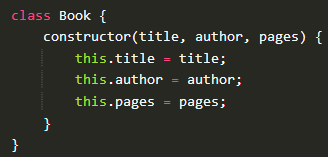


Pour accéder aux données d’un Object :



### Les classes

Création d’une classe et de son constructeur :

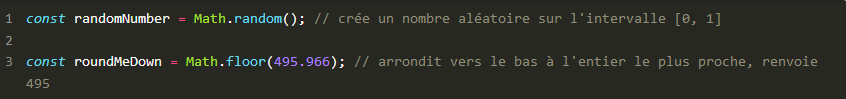


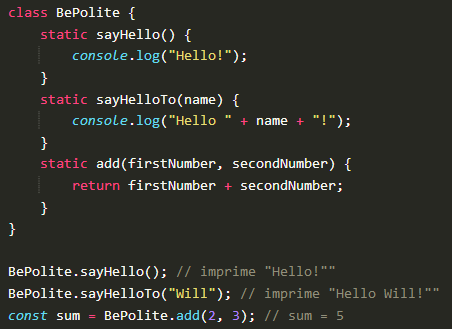
Créer une instance de la classe : 

Creer une méthode de la classe :



Il existe aussi des méthode staticques :

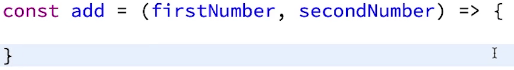


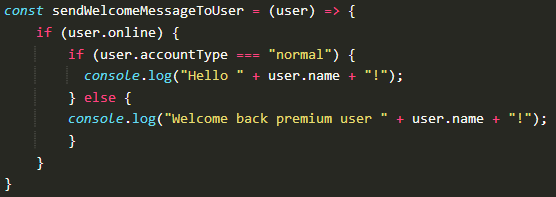


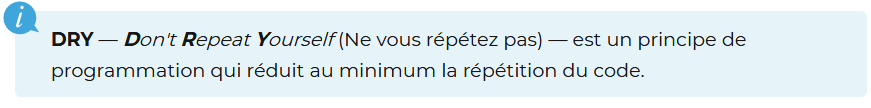
## Les fonctions

### Ecritures

Il existe deux méthodes d’écritures :

1. 

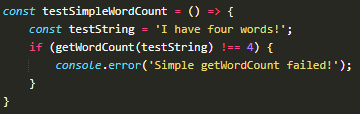


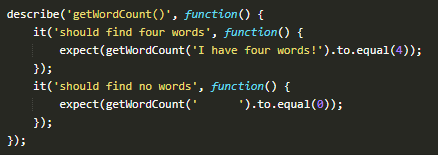


### Les tests

Ils existent trois types de tests :

* les tests unitaires :





* les tests d’intégrations :

Les tests d'intégration vérifient les multiples fonctions ou classes pour s'assurer qu'elles travaillent ensemble comme elles sont censées le faire. L'image ci-dessus montre ce qui se passe quand les unités individuelles fonctionnent correctement (les deux tiroirs s'ouvrent correctement séparément), mais que leur intégration dans le système qui les entoure cause un problème de fonctionnement.

* Les tests fonctionnels (E2E)

Les tests fonctionnels, aussi appelés de bout en bout (E2E), vérifient des scénarios complets en contexte. Par exemple, un utilisateur se connecte à votre application, ouvre ses notifications et les marque toutes comme lues. Ces tests vérifient aussi les ressources externes que votre projet peut utiliser, par exemple un système de paiement tiers.

Il y a trois techniques de débogages différentes :

* affichage de console — parfait pour des erreurs petites, simples et isolées, mais pas idéal dans des projets plus grands et plus complexes ;
* utilisation d'outils pour développeur — qu'il s'agisse des outils intégrés dans la plupart des navigateurs ou du débogueur de votre environnement de développement intégré, c'est à coup sûr la façon fiable de déboguer du code ;
* débogage avec un canard en plastique — quand les choses vont terriblement mal.

## La récursivité

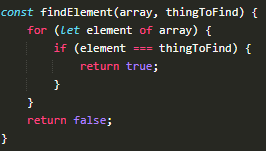
### Qu’est-ce que la récursivité ?

Une fonction récursive est une fonction qui s'appelle elle-même d'une façon ou d'une autre.

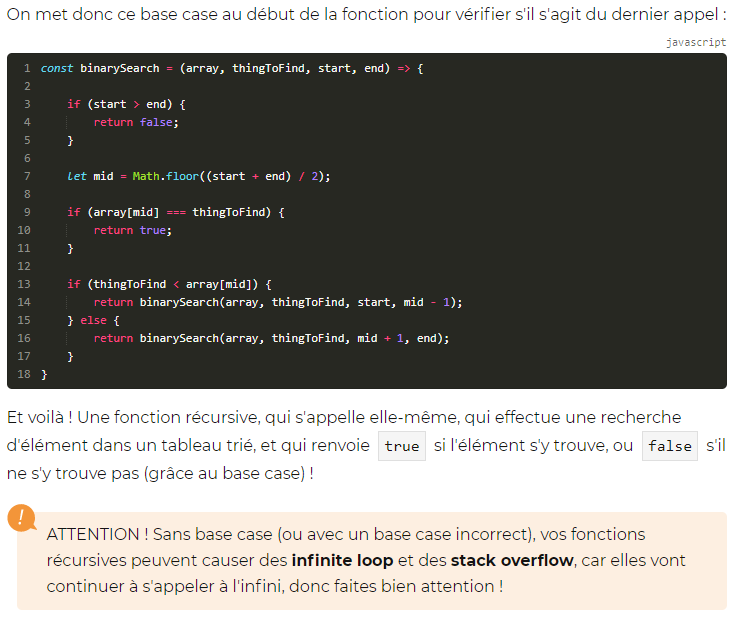
Prenons l'exemple de la recherche binaire.

Le but de l'exercice : chercher un élément dans un array trié pour savoir s'il s'y trouve.

Une approche basique (et plutôt lente) serait la suivante :



Ce code est trop long



## LE DOM

### Qu’est-ce que le DOM ?

Le DOM, qui signifie Document Object Model (c'est-à-dire "modèle d'objet de document", en français), est une interface de programmation qui est une représentation du HTML d'une page web et qui permet d'accéder aux éléments de cette page web et de les modifier avec le langage JavaScript.

Avec une interface de programmation nous permettant de parcourir le DOM, nous allons pouvoir interagir avec lui. Ces interactions comprennent :

* La modification du contenu d'un élément précis ;
* La modification du style d'un élément ;
* La création ou la suppression d'éléments ;
* L'interaction avec les utilisateurs, afin de repérer des clics sur un élément ou encore de récupérer leur nom dans un formulaire ;
* Etc.

### L’object document

Cet objet, auquel vous avez directement accès dans votre code JavaScript, est le point de départ du DOM. Il représente votre page (votre document) entière.

C'est donc lui qui contient les fonctions dont vous aurez besoin pour retrouver les éléments que vous cherchez.

Obtenir un élément









**document.querySelector("#myId p.article > a") :**   fera une recherche dans l'élément ayant pour id  #myId , les éléments de type  <p>  qui ont pour classe  article , afin de récupérer le lien (  <a>  ) qui est un enfant direct (pas des enfants de ses enfants).



Ici, le premier élément <input name="identifiant"/> dans un <div class="panneau-utilisateur principal"> dans le document est retourné .

Les recherches depuis un élément

Il n'y a pas qu'avec  document  que vous pouvez rechercher des éléments. Comme nous l'avons vu au début du chapitre, chaque élément est un objet JavaScript avec ses propriétés et ses fonctions. Et parmi ces dernières, il en existe pour parcourir les enfants et le parent de chaque élément !

* **element.parentElement  :** cette propriété nous retourne l'élément parent de celui-ci ;
* **element.parentNode  :** cette propriété nous retourne l'élément parent qui peut être un texte de celui-ci ;
* **element.children  :** cette propriété nous retourne la liste des enfants de cet élément ;
* **element.childNodes :** cette propriétés nous retournes un tableau de tous les enfants de l’élément.
* **element.firstChild  et element.lastChild :** ces propriétés nous retournes le premier ou le derniers élément de celui-ci. Peut correspondre à un nœud textuel. Contrairement à la propriété suivante.
* **element.firstElementChild  et element.lastElementChild :** ces propriétés nous retournes le premier ou le derniers élément de celui-ci. NE Peut correspondre à un nœud textuel.
* **element.nextSibling**  /  **element.previousSibling  :** ces propriétés nous permettent de naviguer vers l'élément suivant / précédent de même niveau que notre élément. Peut correspondre à un nœud textuel. Contrairement à la propriété suivante.
* **element.nextElementSibling**  /  **element.previousElementSibling  :** ces propriétés nous permettent de naviguer vers l'élément suivant / précédent de même niveau que notre élément. Ne Peut correspondre à un nœud textuel.
* **element.nodeType et element.nodeName :** Vérifie le type et le nom d’un nœud voir tableau ci-dessous.

| **Numéro** | **Type de nœud** |
| --- | --- |
| **1** | **Nœud élément** |
| **2** | **Nœud attribut** |
| **3** | **Nœud texte** |
| 4 | Nœud pour passage CDATA (relatif au XML) |
| 5 | Nœud pour référence d'entité |
| 6 | Nœud pour entité |
| 7 | Nœud pour instruction de traitement |
| **8** | **Nœud pour commentaire** |
| 9 | Nœud document |
| 10 | Nœud type de document |
| 11 | Nœud de fragment de document |
| 12 | Nœud pour notation |

Modifiez ou obtenir le contenu d’un élément

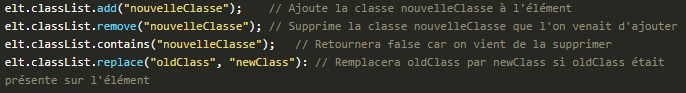
Element.innerHTML : permet de modifier ou d’obtenir le contenu HTML d’un élément.

Element.textContent: permet de modifier ou d’obtenir le contenu d’un élément mais ne prendra pas en compte les balises HTML.

Modifier les classes d’un élément

Element.classList: *permet de modifier la ou les classes d’un élément.*

* Add (<string>, [<string>, ...] ) : ajoute la ou les classes spécifiées ;
* remove (<string>, [<string>, ...] ) : supprime la ou les classes spécifiées ;
* contains (<string> ) : vérifie si la classe spécifiée est contenue par cet élément ;
* replace (<old>, <new> ) : remplace l'ancienne classe par la nouvelle classe.



Changer le style d’un élément



Récupérer le CSS d’un élément

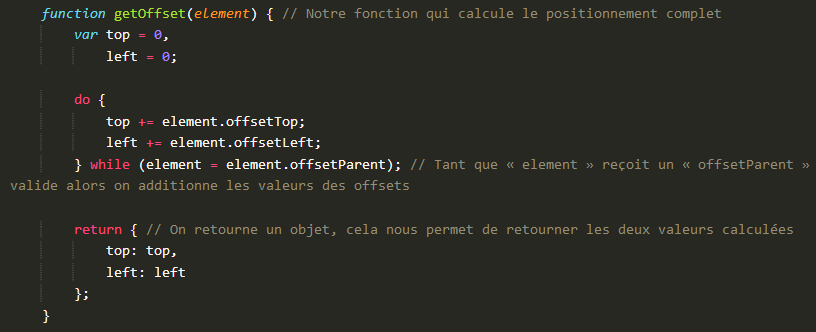
On utilise la fonction getComputedStyle(élément).style



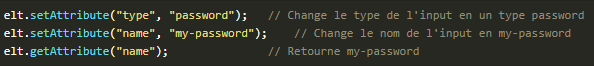
| **Nom de l'attribut** | **Contient…** |
| --- | --- |
| offsetWidth | Contient la largeur complète (width + padding + border) de l'élément. |
| offsetHeight | Contient la hauteur complète (height + padding + border) de l'élément. |
| offsetLeft | Surtout utile pour les éléments en position absolue. Contient la position de l'élément par rapport au bord gauche de son élément parent. |
| offsetTop | Surtout utile pour les éléments en position absolue. Contient la position de l'élément par rapport au bord supérieur de son élément parent. |
| offsetParent | Utile que pour un élément en position absolue ou relative ! Contient l'objet de l'élément parent par rapport auquel est positionné l'élément actuel. |

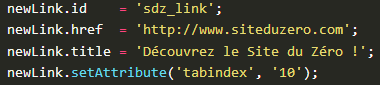


Pour récupérer les offset parent il faut utiliser element.offsetParent



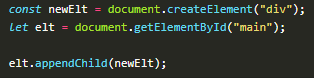
Modifier un attribut





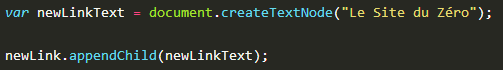
Créer de nouveaux éléments



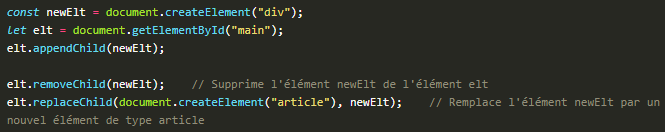
Ajoutez des éléments enfants

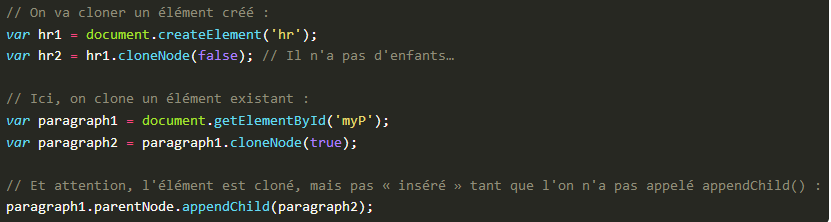
Supprimez un attribut



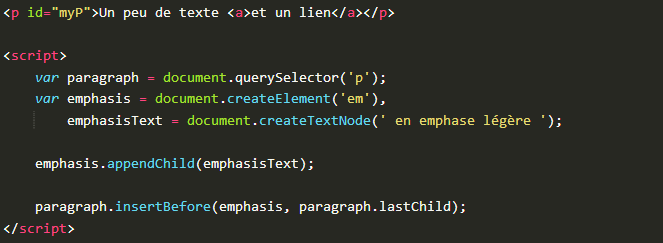
Ajoutez des nœuds Textuels

Supprimez cloner et remplacez des éléments





Insérer des éléments avec insertBefore()



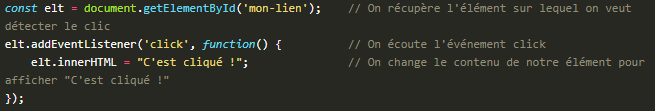
### Les évènements

La liste des évènements  est disponible [ICI](https://developer.mozilla.org/fr/docs/Web/Events)

| **Nom de l'événement** | **Action pour le déclencher** |
| --- | --- |
| click | Cliquer (appuyer puis relâcher) sur l'élément |
| dblclick | Double-cliquer sur l'élément |
| mouseover | Faire entrer le curseur sur l'élément |
| mouseout | Faire sortir le curseur de l'élément |
| mousedown | Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément |
| mouseup | Relâcher le bouton gauche de la souris sur l'élément |
| mousemove | Faire déplacer le curseur sur l'élément |
| keydown | Appuyer (sans relâcher) sur une touche de clavier sur l'élément |
| keyup | Relâcher une touche de clavier sur l'élément |
| keypress | Frapper (appuyer puis relâcher) une touche de clavier sur l'élément |
| focus | « Cibler » l'élément |
| blur | Annuler le « ciblage » de l'élément |
| change | Changer la valeur d'un élément spécifique aux formulaires (input,checkbox, etc.) |
| input | Taper un caractère dans un champ de texte ([son support n'est pas complet sur tous les navigateurs](https://caniuse.com/#feat=input-event)) |
| select | Sélectionner le contenu d'un champ de texte (input,textarea, etc.) |

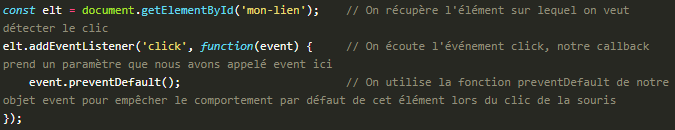
Réagir à un évènement

addEventListener (Event,callback)  : prend en paramètres le nom de l'événement à écouter et la fonction à appeler dès que l'événement est exécuté



La fonction preventDefault()

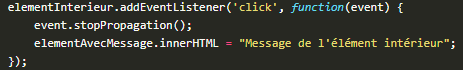
En appelant cette fonction dans votre callback, vous demandez au gestionnaire des événements de ne pas exécuter le comportement par défaut de votre élément (qui est la redirection vers une autre page pour un lien). Et cela marche aussi pour d'autres types d'éléments que le lien. Cette même fonction exécutée pendant un événement  **onsubmit** sur un formulaire empêchera le formulaire de s'envoyer au serveur, par exemple.



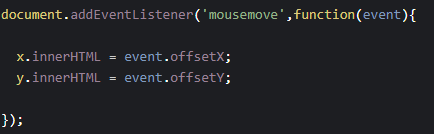
La fonction stopPropagation()

il nous permet d'empêcher la propagation de l'événement vers son parent. En effet, lorsqu'un événement est déclenché, il est d'abord reçu par l'élément cible, mais il est ensuite remonté vers les éléments parents qui sont aussi dans la cible.

Avec stopPropagation() , vous pouvez ainsi empêcher que d'autres éléments reçoivent l'événement.

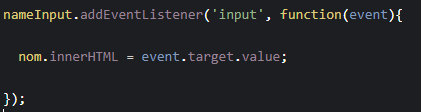






L’évènement **change** permet d’effectuer quelque chose lorsque l’élément change.

L’évènement **input** lui effectuera la modification dès lors que vous taper un caractère dans un formulaire par exemple.



Les propriétés formulaire :



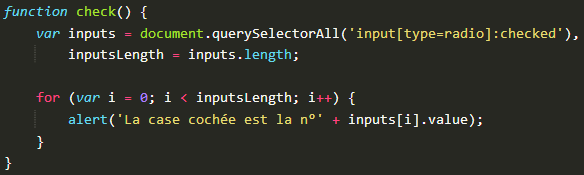


Disabled désactive un champ de texte.



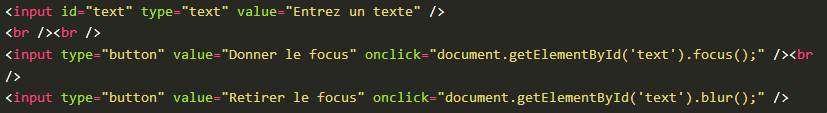


Type représente le type d’input et checked return true si l’élément radio est check



Ici querySelectorALL renvoie sous forme de tableau toutes les balises input de type radio qui sont check.





Blur retire le focus alors que focus donne le focus.

## Le protocole HTTP

### Qu'est-ce que le protocole HTTP ?

HTTP signifie HyperText Transfer Protocol. C'est un protocole qui permet de communiquer avec un site Internet. Il va permettre de charger des pages HTML, des styles CSS, des polices de caractères, des images, etc. Mais ce n'est pas tout, le protocole HTTP nous permet aussi d'envoyer des formulaires et de récupérer et d'envoyer toutes sortes de données depuis ou vers un serveur implémentant ce protocole !

### Les codes HTTP

Ces codes sont représentés par des nombres ayant chacun une signification particulière. Ils sont répartis dans plusieurs catégories :

* Les codes de **100** à **199** sont des codes d'information, plutôt techniques et peu utilisés en pratique ;
* Les codes de **200** à **299** sont des codes de succès. Ils veulent dire que votre requête s'est bien déroulée et vous apporteront une information sur le type de résultat que vous recevez ;
* Les codes de **300** à **399** sont les codes de redirection. Ils indiquent généralement que le contenu que vous recherchez a bougé ou n'est pas accessible directement ;
* Les codes de **400** à **499** sont des codes d'erreur liés à l'utilisation du service web (ressource inexistante, authentification requise, pas les bonnes permissions, requête mal construite, etc). Ce sont des erreurs que l'on retrouve fréquemment dans la pratique et pour lesquelles il est bon d'avoir une stratégie de résolution ;
* Les codes de**500** à **599** sont des codes d'erreur venant du service web (plantage du service, service ne répondant plus, manque de mémoire, etc.). En tant qu'utilisateur du service, vous ne pouvez pas y faire grand-chose, mais de la même manière que pour les erreurs 4xx, il est bon d'avoir une stratégie de résolution.

Voici les codes HTTP les plus fréquents auxquels vous serez probablement confrontés :

* **200 :** indique que tout s'est bien passé ;
* **201 :** indique que tout s'est bien passé et qu'une nouvelle ressource a bien été créée ;
* **204 :** indique que tout s'est bien passé mais qu'aucun résultat n'est renvoyé ;
* **400 :** indique qu'une requête est erronée ;
* **401 :** indique que l'utilisateur n'est pas authentifié, alors que c'est nécessaire ;
* **403 :** indique que l'utilisateur n'a pas le droit d'accéder à cette ressource ;
* **404 :** indique que la ressource demandée n'existe pas ;
* **500 :** indique que le serveur a subi une erreur interne.

### Les méthodes HTTP

Elles permettent d'identifier le type de requête que vous souhaitez faire. Voici les principales méthodes :

* **GET :** permet de récupérer des ressources, comme par exemple le temps actuel sur un service de météo ;
* **POST :** permet de créer ou modifier une ressource, comme la création d'un nouvel utilisateur sur votre application ;
* **PUT :** permet de modifier une ressource, comme le nom de l'utilisateur que vous venez de créer avec POST ;
* **DELETE :** Permet de supprimer une ressource, comme un commentaire dans un fil de discussion.

## AJAX

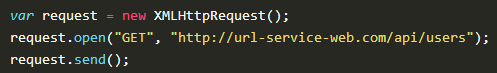
### Qu’est-ce que c’est ?

AJAX (AJAX signifie en fait Asynchrones JavaScript And XML) Il s'agit d'un ensemble d'objets et de fonctions mis à disposition par le langage JavaScript, afin d'exécuter des requêtes HTTP de manière asynchrone.

AJAX va nous permettre d'exécuter des requêtes HTTP sans avoir besoin de recharger la page du navigateur. Cela a plusieurs avantages :

* Avoir un site plus réactif car on n'a pas besoin de recharger toute la page dès qu'on a besoin de mettre à jour une partie du contenu ;
* Améliorer l'expérience utilisateur avec du nouveau contenu qui se charge au fur et à mesure qu'on le découvre, par exemple.

### Envoyé une requête



Voici ce que fait le code, ligne par ligne :

* **Ligne 1 :** on crée un nouvel objet de type  XMLHttpRequest  qui correspond à notre objet AJAX. C'est grâce à lui qu'on va créer et envoyer notre requête ;
* **Ligne 2 :** on demande à ouvrir une connexion vers un service web. C'est ici que l'on précise quelle méthode HTTP on souhaite, ainsi que l'URL du service web ;
* **Ligne 3 :** on envoie finalement la requête au service web.

### Récupérer les données en format JSON

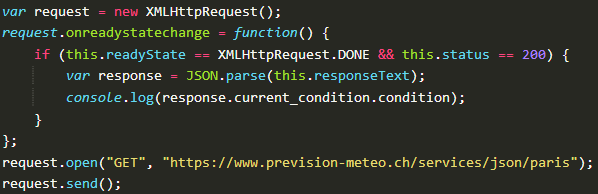
Pour cela, nous allons devoir utiliser la propriété  **onreadystatechange**  en lui passant une fonction. Cette fonction sera appelée à chaque fois que l'état de la requête évolue.

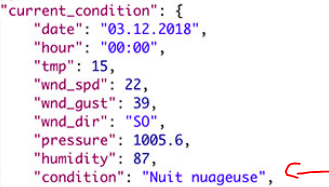
Voici les différents états possibles :

* **UNSENT  (code 0) :** l'objet est prêt mais la méthode  open()  n'a pas encore été appelée ;
* **OPENED  (code 1) :**  open()  a été appelé ;
* **HEADERS\_RECEIVED  (code 2) :**  send()  a été appelé, les headers et  status  sont disponibles au sein de l'objet ;
* **LOADING  (code 3) :** réception en cours, les données reçues sont partielles ;
* **DONE  (code 4) :** requête terminée.

C'est  DONE  qui va nous intéresser car c'est à ce moment-là que la requête est terminée et que nous venons de recevoir le résultat du service web. Pour récupérer l'état actuel de la requête, la fonction que l'on passe à  onreadystatechange  contiendra un objet  this  directement accessible dans la fonction, et qui nous permettra d'accéder aux propriétés suivantes :

* **readyState  :** qui contient l'état de la requête ;
* **status  :** qui contient le code de statut de la requête (souvenez-vous, 2xx quand ça s'est bien passé, 3xx pour les redirections, 4xx pour les erreurs...) ;
* **responseText  :** qui contient la réponse du service web au format texte. Ainsi, si le texte que l'on attend est au format JSON, il va falloir le transformer en objet JavaScript avec la fonction  **JSON.parse(texteJSON)** .



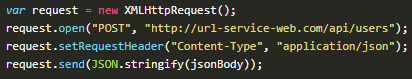


### Vérifié les données





### Envoyé les données



Étant donné que l'on souhaite envoyer du JSON à notre service web, nous avons d'abord besoin de transformer notre objet JavaScript en JSON (qui, rappelons-le, est un format textuel, c'est-à-dire que c'est simplement du texte, contrairement à un objet JavaScript qui est une structure complexe du langage).

Pour faire cette transformation, nous utilisons la fonction **JSON.stringify(json)** . Toujours parce que l'on souhaite envoyer du JSON à notre service web, il faut le prévenir qu'il va recevoir du JSON. Cela se fait grâce à des headers, qui sont des en-têtes envoyés en même temps que la requête pour donner plus d'informations sur celle-ci. Le header en question est Content-Type , avec la valeur application/json . Ainsi, il faut ajouter cette ligne : **request.setRequestHeader("Content-Type", "application/json")**;

## Asynchrone

### Prsentation

Javascript est synchrone et mono-thread c’est à dire : qu'il n'y a qu'un seul fil d'exécution du code source. Cela signifie que lorsque vous écrivez du code, chaque ligne sera exécutée l'une après l'autre en attendant la fin de l'exécution de la ligne précédente.

Pour effectuer du code asynchrone en javascript nous utilisons l’event loop. On demande à exécuter une fonction de façon asynchrone, la fonction en question est placée dans une sorte de file d'attente qui va exécuter toutes les fonctions qu'elle contient les unes après les autres. C'est ce qu'on appelle l'event loop.

### Les fonctions ( setTimeout ,setInterval, ect.. )

**setTimeout**  est la fonction **la plus répandue** lorsque l'on veut exécuter du code asynchrone sans bloquer le fil d'exécution en cours. Cette fonction prend 2 paramètres :

* La **fonction à exécuter** de manière asynchrone (qui sera donc ajoutée à la file d'attente de l'event loop) ;
* Le **délai**, en millisecondes, avant d'exécuter cette fonction.

La fonction **setTimeout** nous retourne une valeur permettant d'identifier le code asynchrone que l'on veut exécuter. Il est possible de passer cet identifiant en paramètre à la fonction **clearTimeout** si vous souhaitez annuler l'exécution asynchrone de la fonction avant qu'elle ne soit exécutée.

**setInterval** : elle fonctionne exactement comme  setTimeout , à ceci près qu'elle exécute la fonction passée en paramètre **en boucle** à une**fréquence déterminée** par le temps en millisecondes passé en second paramètre. Il suffira de passer la valeur de retour de  setInterval  à  clearInterval  pour **stopper** l'exécution en boucle de la fonction ;

**setImmediate** : Cette fonction prend en seul paramètre la fonction à exécuter de façon synchrone.  La fonction en question sera placée dans la**file d'attente** de l'event loop, mais va **passer devant** toutes les autres fonctions, sauf certaines spécifiques au Javascript : les événements (les mêmes qu'on a vus au premier chapitre, et qui sont donc exécutés de façon asynchrone  😉), le rendu, et l'I/O. Il existe aussi  nextTick , qui permet, là, de court-circuiter tout le monde. À utiliser avec précaution, donc...

### Les callbacks

Les callbacks sont la base de l'asynchrone en JavaScript et sont très utilisées.

Le principe de la callback est de la passer en paramètre d'une fonction asynchrone. Une fois que la fonction asynchrone a fini sa tâche, elle va appeler notre fonction callback en lui passant un résultat

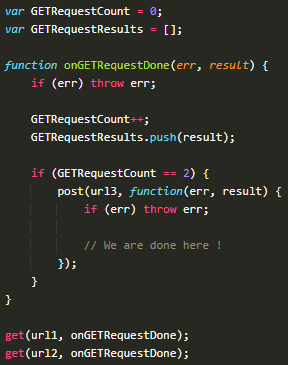
Exemple avec les évènements :



En parallèle

Voyons ensemble comment faire nos 2 requêtes en parallèle, suivies d'une requête en séquence avec les callbacks. Et vous verrez, ça peut vite devenir complexe !

Pour cet exemple, nous partons du principe que nous avons accès à 2 fonctions ( **get** et **post** ) qui font respectivement une requête **GET** et une requête **POST** quand on leur passe en paramètre l'URL de la requête, et une callback à exécuter quand on a le résultat (avec une variable d'erreur en premier paramètre).

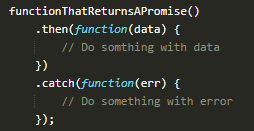


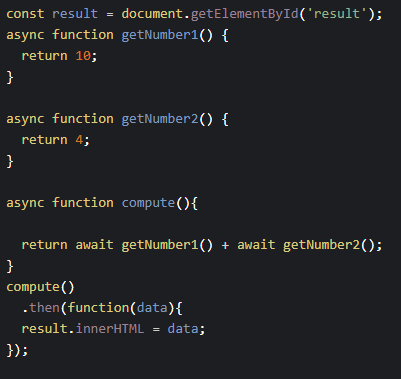
### Les promises

Lorsque l'on exécute du code asynchrone, celui-ci va immédiatement nous retourner une "promesse" qu'un résultat nous sera envoyé prochainement.

Cette promesse est en fait un objet Promise qui peut être **resolve** avec un résultat, ou **reject** avec une erreur.

Lorsque l'on récupère une Promise , on peut utiliser sa fonction **then()** pour exécuter du code dès que la promesse est résolue, et sa fonction **catch()** pour exécuter du code dès qu'une erreur est survenue.

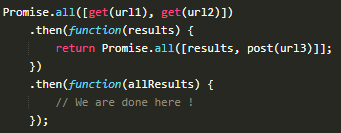




Dans un premier temps nous avons créer une fonction asynchrone (avec async) qui s'appelle compute et qui va récupérer les résultats des 2 fonctions asynchrones getNumber1() et getNumber2() (avec await) et renvoyer la somme des 2 valeurs récupérées.

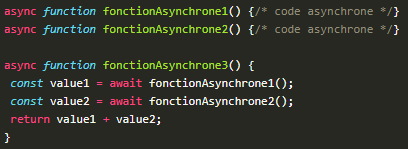
Nous avons appelé notre fonction compute() et utiliser sa valeur de retour comme une Promise pour finalement afficher le résultat de la promesse dans le contenu HTML de l'élément ayant pour ID result.

En parallèle

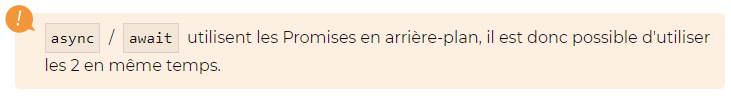


### Async/await

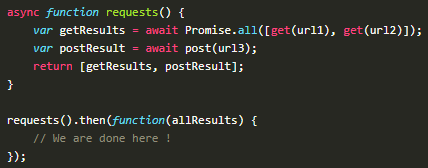
**async** et **await** sont 2 nouveaux mots clés qui permettent de gérer le code asynchrone de manière beaucoup plus intuitive, en bloquant l'exécution d'un code asynchrone jusqu'à ce qu'il retourne un résultat.



Dans cet exemple, nous avons un total de 3 fonction asynchrones : **fonctionAsynchrone1** , **fonctionAsynchrone2** , **fonctionAsynchrone3** . Quand on utilise **async** et **await** , une fonction asynchrone doit avoir le mot clé **async** avant la fonction. Ensuite, dans le code, nous pouvons faire appel à des fonctions asynchrones et attendre leur résultat grâce au mot clé **await** que l'on met devant l'appel de la fonction.



En parallèle



## Optimiser le code

### Linter

Le linter est un programme qui va analyser notre code et détecter les erreurs de syntaxe, les variables non utilisées, les variables qui n'existent pas, la mauvaise organisation du code, le non-respect des bonnes pratiques d'écriture de code...

Le JavaScript étant un langage non compilé, vous ne verrez les erreurs de syntaxe de votre code que lors de l'exécution du code concerné. C'est-à-dire que si une de vos fonctions utilise une variable qui n'est pas définie, vous ne le saurez que lorsque votre fonction sera exécutée. Autant dire que pour certaines fonctions peu utilisées, vous pouvez très facilement laisser des erreurs "d'inattention" passer en production. 😫

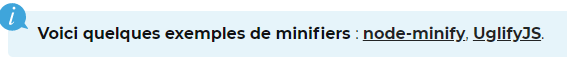
Un linter pourra donc vous aider à régler ces erreurs avant que vos utilisateurs n'en subissent les conséquences (les tests sont aussi un très bon moyen complémentaire au linter pour éviter les erreurs, mais ce n'est pas l'objet de ce cours). Il pourra aussi vous aider à optimiser un peu votre code et vous sera d'une grande aide pour définir des bonnes pratiques d'écriture de code lorsque vous travaillerez en équipe !



### Minifier

Comme vous le savez, votre code JavaScript est chargé par votre navigateur. Cela signifie que votre navigateur va demander à votre serveur votre code JavaScript afin de l'exécuter. Il est alors évident que moins votre code sera lourd (en termes de taille de fichier), plus votre navigateur pourra le récupérer et le charger rapidement.

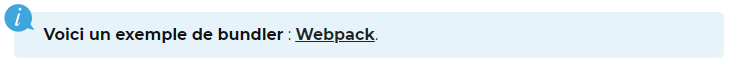
Un minifier est donc un programme responsable de la minification de votre code. C'est-à-dire qu'il va essayer de rendre votre code le plus léger possible en retirant les espaces et retours à la ligne inutiles, en renommant vos variables avec des noms plus courts, en supprimant le code non utilisé, en supprimant les commentaires, en optimisant certains bouts de code pour les réécrire avec une syntaxe plus légère, etc.



### Bundler

De la même manière qu'il est important d'avoir un code le plus léger possible, il est aussi important de réduire au maximum le nombre de fichiers qui composent votre code !

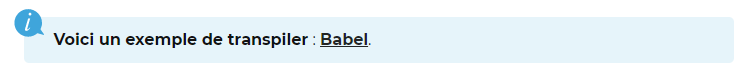
C'est donc le rôle du bundler. Il va se charger de packager votre code pour qu'il tienne dans un seul fichier.



### Transpiler

Vous devez vous en douter, le langage JavaScript évolue. À l'heure actuelle, nous en sommes à la version ECMAScript 2018 (ES9), mais déjà la version ECMAScript 2015 (ES6) n'est plus supportée dans son intégralité tous les navigateurs. Et c'est bien dommage, car les nouvelles versions peuvent offrir des choses intéressantes ( async / await par exemple).

Mais alors comment coder avec la dernière version de JavaScript tout en étant compatible avec tous les navigateurs ? Avec un transpiler !



### Utiliser NODE.JS

#### Initialisez un projet



#### Installer un module



--save-dev signifie que l'on souhaite que NPM sauvegarde cette dépendance dans le fichier **package.json** en tant que dépendance de développement. Il existe aussi --save qui ajoute la dépendance en tant que dépendance de production.

#### Installer un projet avec package.json



Cette commande va installer toutes les dépendances de votre projet qui ont été ajoutées dans le fichier **package.json**. Ce fichier est très utile pour mémoriser et installer tous les modules dont dépend votre projet