Un service web est un programme s'exécutant sur un serveur accessible depuis Internet et répondant à des demandes, appelées **requêtes**.

Par exemple, on fait des requêtes pour interagir avec un service en ligne comme un site e-commerce, un système de paiement, etc. Bref, pour tous les services que l'on utilise au quotidien !

> Mais à quoi ressemble une requête ?

C'est une question intéressante, mais aussi complexe. En effet, il a bien fallu se mettre d'accord sur le **langage** à utiliser pour communiquer avec les services. Mais voilà, un service d'envoi de mails ne fait pas du tout le même travail qu'un service nous renvoyant les pages d'un site Internet. Il semble donc évident que la manière de communiquer ne sera pas la même pour ces deux services. Des **protocoles de communication** ont donc été mis en place afin de standardiser la communication entre services faisant la même chose.

Les **requêtes** sont des données qui respectent le protocole de communication et qui sont envoyées au serveur.

Nous avons donc un protocole pour l'envoi de mail (*SMTP*), la réception de mail (*IMAP*), les requêtes liées à des ressources web (*HTTP),* aux transferts de fichiers (*FTP*), etc.

Le protocole qui va nous intéresser est le protocole nous permettant de communiquer avec un site Internet : le protocole **HTTP**. Grâce à lui, nous allons pouvoir récupérer et sauvegarder des données sur un service web, ce qui nous permettra de dynamiser le contenu de nos pages web.

## 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 !

Nous n'allons pas entrer dans le détail du format des requêtes HTTP, car de nombreuses références le font déjà très bien, mais il me semble important de vous parler de deux choses qui vous serviront souvent : les **codes HTTP** et les **méthodes HTTP**.

### Les codes HTTP

Lorsque vous faites une **requête HTTP**, cela signifie que vous demandez quelque chose au service web. Or, il est possible que le service web ne comprenne pas ce que vous demandez (par exemple si vous demandez à afficher une page qui n'existe pas). Dans ce genre de situation, il faut que vous puissiez le

savoir de manière claire et sans ambiguïté. C'est pour cette raison qu'il existe les codes HTTP. Ils vous permettront d'avoir plus d'informations sur le résultat renvoyé par le service web (succès, erreur...).

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

Lorsque vous faites une requête HTTP, vous pouvez avoir besoin de **demander une ressource** (une page HTML ou la météo par exemple). Mais pas seulement ! Vous pouvez aussi avoir besoin de **créer une ressource** (en envoyant un formulaire pour créer un nouvel utilisateur, par exemple), ou bien d'en **supprimer** une, etc.

C'est ce que l'on appelle 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.

Il s'agit surtout d'une convention. Rien n'oblige un service web à créer une ressource lors d'une requête *POST* par exemple, mais c'est généralement le cas.

## Qu'est-ce qu'une API

Je ne sais pas si vous vous souvenez, mais au début de ce cours je vous disais qu'il était possible de **récupérer** ou d'**envoyer** toutes sortes de données depuis ou vers un service web. Cela signifie que l'on n'est pas limité à la récupération de pages HTML, d'images ou de feuilles de styles CSS. En effet, on développe généralement des services web bien plus complexes, afin de répondre à un vrai besoin métier. Par exemple, un service de météo vous permettra de demander la météo d'une ville précise sur une période donnée. De la même manière, un réseau social vous permettra de rechercher des amis, de poster un contenu, de récupérer une liste des contenus, etc.

Un service web permet donc de faire toute une série de requêtes couvrant les fonctionnalités mises à disposition par le site web ou l'application. C'est ce qu'on appelle une **API** !

*API* signifie *Application Programming Interface*. Il s'agit d'une **interface** mettant à disposition des **points d'accès** vers les ressources de l'application. Ainsi, l'API d'un réseau social met par exemple à disposition une ressource *contenu* qu'il vous est possible de récupérer, de créer ou de supprimer. Cette ressource contient certains attributs comme un titre, une date et un message. Il vous est donc possible d'envoyer des requêtes au service web par rapport à cette ressource. Ainsi, vous pourrez faire une requête de type *POST* pour demander au serveur de créer une nouvelle ressource de type *contenu* avec un titre, une date et un message de votre choix.

Il est important de connaître l'API du service web avec lequel vous allez communiquer, car c'est elle qui vous informera sur les requêtes qu'il vous est possible de faire. La plupart des services web connus et accessibles pour les développeurs ont une documentation expliquant comment se servir de leur API. Sinon, si c'est la vôtre, vous ne devriez pas avoir trop de problème à savoir comment communiquer avec 😐 .

## En résumé

Aucun des deux 😃. *AJAX* signifie en fait *Asynchronous 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.

> Nous verrons en détail ce qu'*asynchrone* signifie dans la prochaine partie, retenez simplement pour l'heure que cela permet d'exécuter du code (une requête ici), sans bloquer l'exécution de la page, en attendant la réponse du service web.

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.

## Envoyez une première requête

Voyons maintenant comment on peut construire et envoyer une requête HTTP avec AJAX.

Si vous vous souvenez du chapitre précédent, vous devez avoir deviné que nous allons créer une requête HTTP avec la méthode GET, afin de récupérer des données. C'est parti !

javascript

```
1 var request = new XMLHttpRequest();
2 request.open("GET", "http://url-service-web.com/api/users");
3 request.send();
```

Ce code nous permet d'envoyer une requête HTTP de type *GET* au service web se trouvant à l'adresse `http://url-service-web.com/api/users` .

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.

## Récupérez les données au format JSON

Maintenant que nous savons comment construire et envoyer une requête HTTP, il faut récupérer et interpréter son résultat.

Un service web peut choisir le format qu'il veut pour nous renvoyer des données, mais le plus courant et le plus simple est le **format *JSON***.

## Qu'est-ce que le format JSON ?

*JSON* signifie *JavaScript Object Notation*. Il s'agit d'un **format textuel** (contrairement à un format binaire plus léger mais impossible à lire à l'œil humain), se rapprochant en termes de syntaxe de celui des objets dans le langage JavaScript.

Ainsi, l'objet JavaScript suivant :

javascript

```javascript
const obj = {
    name: "Mon contenu",
    id: 1234,
    message: "Voici mon contenu",
    author: {
        name: "John"
    },
    comments: [
        {
            id: 45,
            message: "Commentaire 1"
        },
        {
            id: 46,
            message: "Commentaire 2"
        }
    ]
};
```

sera retranscrit ainsi en JSON :

json

```json
{
    "name": "Mon contenu",
    "id": 1234,
    "message": "Voici mon contenu",
    "author": {
        "name": "John"
    },
    "comments": [
        {
            "id": 45,
            "message": "Commentaire 1"
        },
        {
            id: 46,
            "message": "Commentaire 2"
        }
    ]
}
```

Vous pouvez voir comme la syntaxe est similaire entre les 2. Alors qu'en JavaScript il n'est pas nécessaire de mettre les propriétés (comme `name`, `id`, `message`, etc.), entre guillemets, cela l'est en JSON. En JavaScript, votre objet est assigné à une variable, alors qu'en JSON on ne fait que décrire une structure.

Le gros avantage de ce format lorsqu'il est utilisé avec le langage JavaScript est qu'il n'y a pas besoin de le ***parser*** comme on le ferait avec du *XML* par exemple. En effet, lorsque l'on manipule un format de données comme le *XML*, (que vous pouvez comparer à du HTML dans sa forme) il est nécessaire de le *parser*. C'est-à-dire que notre application doit le **lire** et le **comprendre** afin d'en faire ce qu'on veut. C'est une opération généralement complexe que l'on laisse à du code externe développé et éprouvé par d'autres personnes, afin de gagner du temps et d'éviter les erreurs. Mais pour le JSON c'est différent, car c'est une syntaxe qui vient des **objets en JavaScript**. Ainsi, votre navigateur sait directement le lire et le transformer en objets JavaScript. Si l'on reprend l'exemple de JSON précédent, il nous fournira donc un objet JavaScript comme celui du premier échantillon de code ci-dessus.

Le deuxième avantage de ce format est sa **légèreté** par rapport à un format comme le *XML,* qui reste bien plus verbeux, c'est-à-dire avec plus de texte. C'est un gros avantage lorsqu'on communique avec un service web, car cela permet d'optimiser les temps de réponse. En effet, un nombre plus faible d'octets a besoin de transiter dans le réseau.

## Récupérez le résultat de la requête

Avec tout ce que l'on vient d'apprendre, il n'est plus très difficile d'utiliser le résultat de la requête pour en faire ce que l'on veut.

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)` .

Voici comment procéder pour récupérer la météo actuelle sur Paris avec l'API fournie par www.prevision-meteo.ch :

javascript

```
var request = new XMLHttpRequest();
request.onreadystatechange = function() {
    if (this.readyState == XMLHttpRequest.DONE && this.status == 200) {
        var response = JSON.parse(this.responseText);
        console.log(response.current_condition.condition);
    }
};
request.open("GET", "https://www.prevision-meteo.ch/services/json/paris");
request.send();
```

Vous pouvez voir dans l'exemple ci-dessus que l'URL passée à la fonction `open()` a changé et correspond à la récupération de la météo pour Paris. Le type de requête est *GET* car nous voulons récupérer les données.

Ensuite, nous pouvons voir l'utilisation de la propriété `onreadystatechange` . Nous lui passons une fonction qui sera appelée dès que l'état de la requête changera. Nous vérifions donc d'abord si la requête est terminée et si elle s'est bien déroulée, grâce aux propriétés `readyState` et `status` de l'objet interne `this` . Si tout est bon, alors nous utilisons la fonction `JSON.parse()` afin de transformer le texte *JSON* de la réponse en objet JavaScript. Puis, nous affichons dans la console la météo actuelle, dont voici l'extrait *JSON* reçu depuis le service web à l'heure où j'écris ces lignes :

```
{
    "city_info": {
        "name": "Paris",
        "country": "France",
        "latitude": "48.8608017",
        "longitude": "2.3457999",
        "elevation": "60",
        "sunrise": "08:23",
        "sunset": "16:57"
    },
    "forecast_info": {
        "latitude": null,
        "longitude": null,
        "elevation": "55.0"
    },
    "current_condition": {
        "date": "03.12.2018",
        "hour": "00:00",
        "tmp": 15,
        "wnd_spd": 22,
        "wnd_gust": 39,
        "wnd_dir": "SO",
        "pressure": 1005.6,
        "humidity": 87,
        "condition": "Nuit nuageuse",
        "condition_key": "nuit-nuageuse",
        "icon": "https://www.prevision-meteo.ch/style/images/icon/nuit-nuageuse.png",
        "icon_big": "https://www.prevision-meteo.ch/style/images/icon/nuit-nuageuse-big.png"
    },
    "fcst_day_0": {
        "date": "03.12.2018",
        "day_short": "Lun.",
        "day_long": "Lundi",
        "tmin": 12,
        "tmax": 15,
        "condition": "Pluie modérée",
```


Extrait JSON météo Paris

Comme vous pouvez le voir, dans l'exemple de code, j'accède à l'objet `current_condition` et j'affiche la propriété `condition`, qui contient dans mon cas *"Nuit nuageuse"* (ce qui est assez fréquent à Paris 😁).

## Pratiquez !

Exercez-vous maintenant avec ce que nous venons d'apprendre.

Rendez-vous dans cet éditeur CodePen pour réaliser les exercices suivants:

Dans cet exercice nous voulons récupérer la météo actuelle à Paris et l'afficher dès que nous cliquons sur le bouton *"Quelle est la météo sur Paris ?"*

- Etape 1: Nous allons commencer par la création d'une fonction `askWeather` qui va créer une requête AJAX de type `GET` vers le service web avec l'URL

`https://www.prevision-meteo.ch/services/json/paris` . Cela a pour but de demander la météo actuelle sur Paris. Toujours dans cette fonction, nous allons envoyer la requête avec la méthode `send()`

- Etape 2: Maintenant, nous voudrions récupérer la réponse du service web et l'afficher dans l'élément ayant pour ID `weather-result` . Modifiez donc notre fonction `askWeather` afin de récupérer la réponse du service web dès que tout s'est bien passé (grâce au statut de la requête et à l'état de la demande). Le service web vous retournera une réponse au format JSON, et vous aurez besoin d'afficher la propriété `condition` se trouvant dans l'objet `current_condition` . Voici à quoi ressemble la réponse JSON :

json

```
{
        "current_condition": {
                "condition": "Beau temps",
                ...
        },
        ...
}
```

- Etape 3: Enfin, nous voulons afficher la météo (et donc appeler notre fonction `askWeather` ) dès que nous cliquons sur le bouton ayant pour ID `ask-weather` .

## En résumé

Dans ce chapitre, vous avez appris :

- Ce qu'est AJAX ;
- Comment l'utiliser pour faire une requête simple à un service web ;
- Ce qu'est le format JSON ;
- En quoi ce format est adapté aux API et au JavaScript ;
- Comment récupérer les données envoyées par un service web.

*Maintenant que l'on sait faire une requête de type GET pour récupérer des données du service web, nous allons voir comment faire une requête POST pour envoyer des données au service web. Mais avant ça, voyons comment préparer nos données à être envoyées au service web.*

MARK THIS CHAPTER AS UNFINISHED

COMPRENEZ CE QU'EST UN SERVICE

VALIDEZ LES DONNÉES SAISIES PAR VOS

numéro de téléphone qu'il va le faire...

## Never trust user input!

Ce qui se traduit littéralement par : **Ne faites jamais confiance aux données saisies par vos utilisateurs !**

Il est bon de le rappeler :

> **Ne faites jamais confiance à vos utilisateurs !**

Bon, maintenant je pense qu'il est temps d'expliquer pourquoi il faut être si méfiant envers vos utilisateurs, et pourquoi c'est si important.

Certains de vos utilisateurs peuvent être **malveillants** ou ils peuvent **ne pas bien comprendre** ce que vous souhaitez qu'ils fassent. Bref, ils ne vont pas toujours entrer les données que vous attendez d'eux. Or, cela peut se révéler désastreux pour votre site web, et ce, de plusieurs façons :

- Si l'utilisateur ne comprend pas ce que vous attendez de lui, il peut entrer quelque chose qui ne vous convient pas. Votre service web ne fera pas ce qu'il faut et l'utilisateur ne comprendra pas pourquoi ça ne marche pas. Vous risquez de **perdre** cet utilisateur parce que vous n'avez pas validé les données saisies et affiché un message d'erreur indiquant que les données ne sont pas celles attendues ;
- Pire encore, l'utilisateur pourrait **attaquer** votre service web en entrant des **données malveillantes** dans un champ alors que vous attendiez simplement un nom, par exemple. Dans ce cas, il pourrait prendre le contrôle de votre service web, collecter des données utilisateurs, se faire passer pour un administrateur, et j'en passe... (il existe des tonnes de ressources sur Internet donnant des exemples de ces types d'attaques. Si cela vous intéresse, je vous encourage à faire quelques recherches) ;
- De la même manière, l'utilisateur pourrait **faire planter** votre application s'il entrait du texte dans un champ où vous attendiez un nombre.

Nous allons apprendre ici à valider les données utilisateur sur votre site web avant de les envoyer à votre service web. Cela constitue un premier rempart face aux problèmes cités plus haut, même si ce n'est pas suffisant, **il faudra toujours avoir une validation poussée des données utilisateurs sur le service web**.

> C'est important, car rien n'empêchera un utilisateur malveillant d'utiliser un logiciel pour envoyer directement les requêtes HTTP malveillantes à votre service web. De cette manière, elles ne passeront pas par la case validation côté site web...

## Validez les données suite à des événements

Afin de valider les données utilisateurs, vous pouvez vous aider des événements du *DOM*. Ainsi, vous pouvez écouter l'événement `onChange` pour vérifier la donnée, dès que l'utilisateur a fini de l'éditer. Ou bien vous pouvez écouter l'événement `onInput` pour vérifier la donnée à chaque nouveau caractère.

Par exemple, vous pouvez vérifier que ce qui est saisi commence par `Hello` avec le code suivant :

javascript

```javascript
myInput.addEventListener('input', function(e) {
    var value = e.target.value;
    if (value.startsWith('Hello ')) {
        isValid = true;
    } else {
        isValid = false;
    }
});
```

## Faites une validation plus complexe avec les Regex

Si vous n'avez jamais entendu parlé des ***Regex***, sachez qu'il s'agit d'un format spécial qui permet de *matcher* du texte, c'est-à-dire de vérifier qu'un texte corresponde à une *description* que l'on a définie. Ainsi, si l'on veut savoir si notre texte commence par la lettre `e` et est suivi d'au moins 3 chiffres, on écrira la *regex* suivante :

javascript

```javascript
function isValid(value) {
    return /^e[0-9]{3,}$/.test(value);
}
```

Cela peut paraître barbare comme notation, mais c'est très puissant et très pratique !

> Pour en savoir plus, **un peu de documentation** ne fait pas de mal 😉.

## Découvrez les contraintes HTML5

Depuis *HTML* version 5, il est possible d'ajouter de la validation **directement** dans le code HTML, sans avoir besoin d'écrire la moindre ligne de JavaScript.

Pour cela, différents **attributs** sont ajoutés et permettent d'**empêcher la soumission d'un formulaire** si toutes les validations ne sont pas respectées.

### L'attribut type pour les inputs

Pour valider les informations saisies dans une balise `input`, il est possible d'utiliser l'attribut `type`.

L'attribut `type` de la balise `input` ne prend pas seulement comme valeurs `text` et `password`. Cela peut aussi être `email`, `tel`, `URL`, `date` et bien d'autres.

Lorsque vous ajoutez un élément `input` avec un attribut `type="email"`, le navigateur empêchera la soumission du formulaire si ce n'est pas une adresse email correcte.

### Les attributs de validation simples

En fonction du `type` de l'`input`, vous pouvez utiliser différents attributs pour **perfectionner** votre validation :

- `min` / `max` : fonctionne avec des champs de type ***nombre*** ou ***date***. Cela permet de définir une valeur minimum et une valeur maximum autorisées ;
- `required` : fonctionne avec à peu près **tous les types de champs**. Cela rend obligatoire le remplissage de ce champ ;
- `step` : fonctionne avec les **dates** ou les **nombres**. Cela permet de définir une valeur d'incrément lorsque vous changez la valeur du champ via les flèches ;
- `minlength` / `maxlength` : fonctionne avec les **champs textuels** ( `text`, `url`, `tel`, `email` ...). Cela permet de définir un nombre de caractères minimum et maximum autorisé.

### Les patterns

Nous avons vu qu'il était possible d'avoir une validation complexe grâce aux Regex en JavaScript. Eh bien c'est aussi possible directement en HTML5 avec l'attribut `pattern`. Il suffit de définir une Regex dans cet attribut, et vous obligez la valeur du champ correspondant à la respecter.

Par exemple, si on prend le code suivant :

```html
<input type="text" pattern="[0-9]{,3}" />
```

il empêchera un utilisateur d'entrer autre chose que des chiffres, et limitera leur nombre à 3 chiffres.

> Pour en savoir plus, vous pouvez aller faire un tour sur **MDN**.

## Pratiquez !

Voici quelques exercices pour vous faire les dents sur la validation 😎. Rendez-vous sur cet éditeur CodePen et réalisez les étapes suivantes:

1. Nous souhaitons dans un premier temps valider le champ `Code` du formulaire. A chaque lettre saisie dans le champ ayant pour ID `code` nous voulons vérifier que la valeur du champ commence bien par `CODE-` grâce à une *Regex* que voici : `/^CODE-/`. Si la valeur commence bien par `CODE-` alors nous affichons dans l'élément ayant pour ID `code-validation` : `Code valide`, sinon nous affichons dans cet élément `Code invalide`.
2. Maintenant que nous savons si notre code est valide ou non, nous voudrions griser (grâce à l'attribut `disabled`) le bouton de soumission (L'input de type `submit` ayant pour ID `submit-btn`) quand le code est invalide, et le dégriser quand le code est valide.
   Cela signifie que nous allons devoir ajouter un attribut `disabled="true"` au bouton de soumission quand le code est invalide. Et supprimer cet attribut `disabled` quand le code est valide (Rappelez-vous du cours sur la modification du DOM pour définir et supprimer des attributs).
3. Finalement, nous avons un champ `Email` et nous voudrions le rendre obligatoire et obliger l'utilisateur à entrer une adresse email valide. Il faudra aussi empêcher le formulaire de se soumettre s'il n'est pas valide.
   Mais nous voudrions faire tout ça uniquement avec le HTML5, sans utiliser de code JavaScript.
   *Vous pouvez changer le* `type` *du champ email plutôt que d'utiliser une Regex via* `pattern` 

## En résumé

Dans ce chapitre, vous avez appris :

- Qu'il ne faut jamais faire confiance à l'utilisateur ;
- À valider des données lors d'événements ;
- À utiliser les Regex pour valider les données ;
- À utiliser les attributs fournis par HTML pour valider les données.

*Maintenant qu'on peut protéger notre site web de nos utilisateurs, et leur afficher de beaux messages d'erreur quand les données saisies ne sont pas celles attendues, nous allons pouvoir sauvegarder nos données sur notre service web !*

MARK THIS CHAPTER AS UNFINISHED

RÉCUPÉREZ DES DONNÉES D'UN SERVICE WEB

SAUVEGARDEZ DES DONNÉES SUR LE SERVICE WEB

Eh bien, c'est aussi possible d'en envoyer au service web en les ajoutant à notre requête !

Cependant, cela ne se fait pas avec toutes les méthodes (qu'on appelle aussi des *verbs*) HTTP. En effet, la méthode **GET** est seulement faite pour récupérer des données, alors que des méthodes comme **POST** et **PUT** sont faites pour en envoyer et en recevoir.

Le fonctionnement d'un verb à l'autre est très similaire. Avec les verbs **POST** et **PUT**, nous allons simplement ajouter des données dans le corps de notre requête.

## Envoyez des données avec une requête POST

Afin d'envoyer des données à un service web avec la méthode **POST** via AJAX, nous allons devoir passer par la méthode `send()` en lui passant en paramètres les données à envoyer.

javascript

```
var request = new XMLHttpRequest();
request.open("POST", "http://url-service-web.com/api/users");
request.setRequestHeader("Content-Type", "application/json");
request.send(JSON.stringify(jsonBody));
```

Comme vous pouvez le voir, nous avons passé le contenu à envoyer au service web à notre fonction `send()`. 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");`

> *PUT* fonctionne exactement de la même manière que *POST*.

## Pratiquez !

Voici un petit exercice, à réaliser sur cet éditeur CodePen.

Dans cet exercice nous voulons pouvoir entrer du texte dans le formulaire et l'envoyer vers un service web. Ce service web va simplement nous renvoyer notre contenu en plus d'autres informations et nous allons afficher le contenu renvoyé par le serveur.

1. Nous allons commencer par créer une fonction appelée `send` et qui va créer notre objet AJAX.

Nous souhaitons créer une requête de type `POST` vers l'adresse suivante : `https://mockbin.com/request`, et y envoyer un contenu JSON ayant une propriété `value` qui contiendra la valeur du champ de saisie de la page (avec l'ID `value`). Par exemple : `{value: document.getElementById("value").value}`.

Nous souhaitons aussi, lorsque la requête s'est bien envoyée, afficher le résulat renvoyé par le service web. Pour ce faire, nous allons afficher ce qui se trouve dans `postData.text` de la réponse dans le contenu HTML de l'élément ayant pour ID `result`.

2. Maintenant nous voulons envoyer notre requête, et donc appeler notre fonction `send` dès que nous soumettons le formulaire ayant pour ID `form`.

   *N'oubliez pas d'annuler le comportement par défaut de la soumission du formulaire, sinon votre page va se recharger*

## En résumé

Dans ce chapitre, vous avez appris :

- Que l'envoi de données en HTTP se fait via certains verbs ;
- À envoyer des données en HTTP.

*Vous savez comment envoyer des requêtes à un service web pour lui demander des ressources ou en créer de nouvelles. Dans la partie suivante, nous aborderons l'asynchronisme en JavaScript afin d'exécuter du code en "parallèle".*

Que pensez-vous de ce cours ? Donnez-nous votre avis **ici**.

MARK THIS CHAPTER AS UNFINISHED

VALIDEZ LES DONNÉES SAISIES PAR VOS UTILISATEURS

QUIZ: COMMUNIQUEZ AVEC UN SERVICE WEB

### Teacher

Fabien Henon

## JavaScript est synchrone et mono-thread

*Qu'est-ce que ça veut dire ?*

Eh bien, tout simplement 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. Il n'y a pas d'autre code qui pourra être exécuté en parallèle. Il ne peut faire qu'une seule chose à la fois.

Mais dans le titre du chapitre tu parles de faire de l'asynchrone en JavaScript, et là tu nous dis que ce n'est plus possible ?

Eh bien, en fait il est possible et même très facile de faire de l'asynchrone en JavaScript, mais l'exécution restera synchrone...

😉

Bon, essayons de clarifier tout ça maintenant...

Si du code **synchrone** est du code qui s'exécute ligne après ligne en attendant la **fin de l'exécution** de la ligne précédente, alors on peut facilement en déduire que du code **asynchrone** va s'exécuter ligne après ligne, mais la ligne suivante **n'attendra pas** que la ligne asynchrone ait fini son exécution. Prenons cet exemple :

javascript

```
let productId = 1;
let productPrice = getProductPriceAsync(productId);
doSomething(productPrice);
```

En admettant que la fonction `getProductPriceAsync()` soit asynchrone, alors la ligne suivante sera exécutée avant la fin de l'exécution de la fonction asynchrone, mais il ne sera pas encore possible d'utiliser la valeur de `productPrice` (nous verrons dans le chapitre suivant comment nous pouvons nous en servir).

Mais comment peut-on faire ça avec un langage synchrone ?

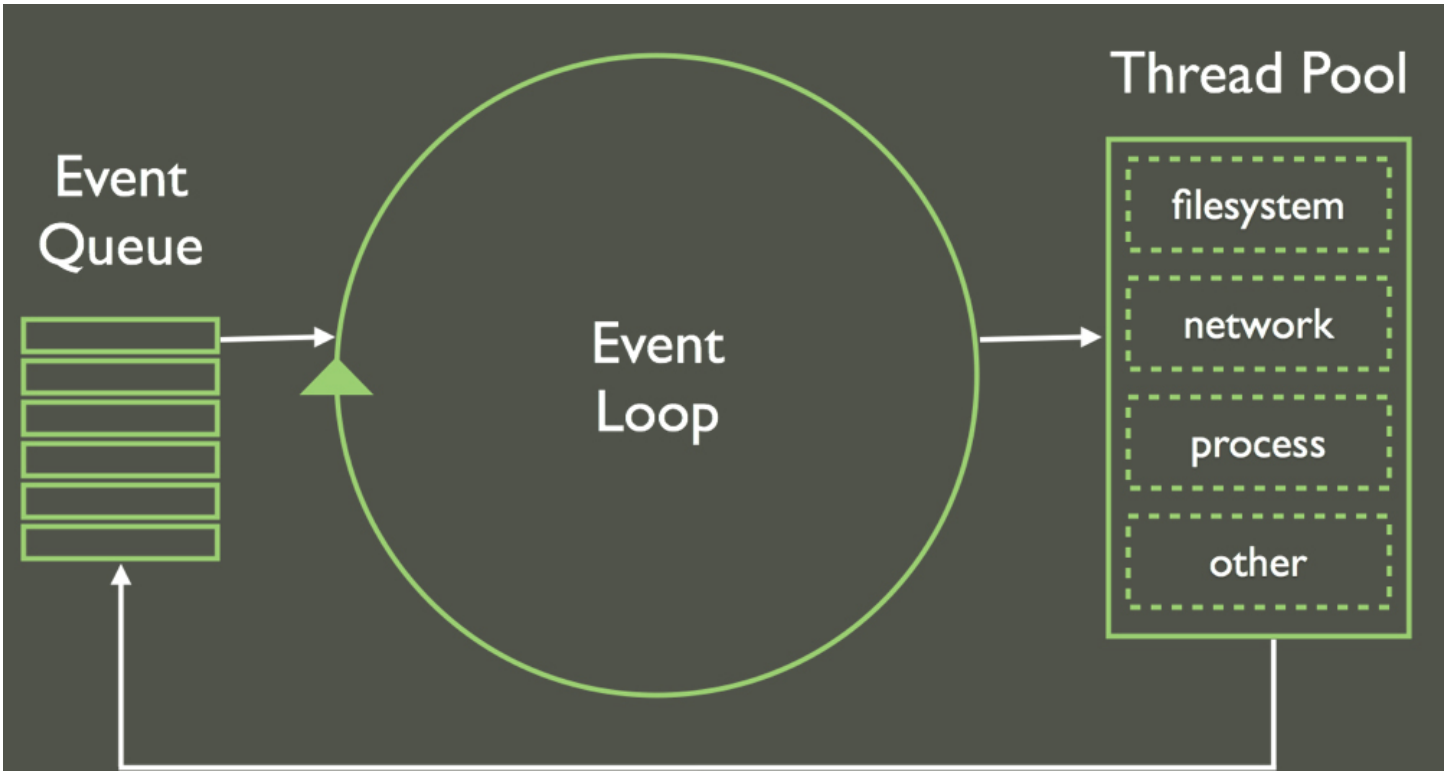
Avec ce qu'on appelle l'*event loop* !

## L'event loop

En JavaScript, chaque ligne de code est exécutée de façon synchrone, mais il est possible de demander à exécuter du code de manière asynchrone. Et lorsque l'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***. Tout le cœur du langage fonctionne autour de ça.

Ainsi, le code n'est pas réellement exécuté en parallèle car il est mis en file d'attente, mais il ne bloque pas l'exécution du code depuis lequel il a été appelé.




Event Loop

## Jouez avec l'event loop

Maintenant que vous comprenez un peu mieux ce qu'est l'event loop, voyons concrètement comment demander à exécuter du code de manière asynchrone.

### La fonction setTimeout

`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.

javascript

```
setTimeout(function() {
    console.log("I'm here!")
}, 5000);

```

```
5 console.log("Where are you?");
```

Dans l'exemple ci-dessus, le texte `Where are you?` s'affichera avant `I'm here!`, qui ne sera affiché qu'au bout de 5 secondes.

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.

Pour en savoir plus sur la fonction setTimeout, vous trouverez la documentation ici.

### Les autres méthodes

Il existe d'autres méthodes un peu moins répandues, voire très peu utilisées :

- `setInterval` (doc) : 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` (doc). 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...

## Le cas de l'I/O

L'I/O correspond aux événements liés à l'***input***(les flux d'entrée) et l'***output***(les flux de sortie). Cela correspond notamment à la lecture/écriture des fichiers, aux requêtes HTTP, etc.

Vous avez dû remarquer que lorsque l'on exécutait la fonction `send()` lors d'une requête HTTP, celle-ci ne bloquait pas l'exécution du code. On n'attend pas que la requête soit envoyée et une réponse reçue avant d'exécuter le reste du code. C'est donc une fonction asynchrone, et tout ce qui touche à l'I/O peut être exécuté de manière asynchrone. Et c'est tant mieux, car leur exécution peut prendre du temps 😇.

## En résumé

Dans ce chapitre, vous avez appris :

- Que le JavaScript est synchrone et mono-thread ;
- Ce qu'est l'event loop ;

C'est la méthode la plus "vieille" mais toujours utilisée par beaucoup de modules JavaScript (nous verrons ce que sont les modules dans la dernière partie de ce cours). Une ***callback*** est simplement une fonction que vous définissez. 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**. Ainsi, le code que nous mettons dans notre fonction *callback* sera exécuté de manière asynchrone. Cela ne vous rappelle pas quelque chose ? Les **événements** ! Les événements sont un exemple typique de fonction asynchrone à laquelle on passe une fonction *callback*.

javascript

```javascript
element.addEventListener('click', function(e) {
    // Do something here ...
});
```

Dans l'exemple ci-dessus, la fonction qui est envoyée à `addEventListener` est une callback. Elle n'est pas appelée tout de suite, elle est appelée plus tard, dès que l'utilisateur clique sur l'élément. Ça ne bloque donc pas l'exécution du code et c'est donc asynchrone 😁.

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

Par exemple, lorsque nous définissions une fonction dans la propriété `onreadystatechange`, nous étions en train de lui définir une callback. De la même manière, la fonction que nous passons en paramètre à `setTimeout` est une callback.

Les callbacks sont faciles à comprendre et à utiliser, mais elles souffrent d'un gros problème de lisibilité du code, via ce qu'on appelle le *callback hell*. En effet, on se retrouve régulièrement dans des situations où on va imbriquer plusieurs couches de callbacks , rendant le code difficile à lire et pouvant générer des erreurs.

javascript

```javascript
elt.addEventListener('click', function(e) {
    mysql.connect(function(err) {
        mysql.query(sql, function(err, result) {
            fs.readFile(filePath, function(err, data) {
                mysql.query(sql, function(err, result) {
                    // etc ...
                });
            });
        });
    });
});
```

Ce code, qui n'est pas facile à lire, pourrait pourtant correspondre à un cas d'utilisation concret des callbacks : dès que l'utilisateur clique sur un élément, on ouvre une connexion *MySQL*, puis on récupère des données depuis la base de données, on lit un contenu dans un fichier et on fait une nouvelle requête *MySQL*, etc.

C'est bien beau de gérer du code asynchrone, mais rien ne vous garantit que tout se soit bien passé. Il nous faut donc un mécanisme pour savoir si une erreur est survenue !

### Gestion des erreurs

Pour gérer les erreurs avec les callbacks, la méthode la plus utilisée est de prendre **2 paramètres** dans notre callback. Le 2e paramètre est notre donnée et le 1er est l'erreur. Si elle n'est pas ***null*** ou ***undefined***, elle contiendra un message d'erreur indiquant qu'une erreur est intervenue.

Si on reprend l'exemple ci-dessus, on voit par exemple que la lecture d'un fichier avec le module `fs` peut nous retourner une erreur :

javascript

```javascript
fs.readFile(filePath, function(err, data) {
    if (err) {
        throw err;
    }
    // Do something with data
});
```

## Promises

Les ***promises***, ou *promesses* en français, sont un peu plus complexes mais bien plus puissantes et faciles à lire que les callbacks.

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.

Voyons avec un exemple concret pour mieux comprendre :

javascript

```javascript
functionThatReturnsAPromise()
    .then(function(data) {
        // Do somthing with data
    })
    .catch(function(err) {
        // Do something with error
    });
```

Dans l'exemple ci-dessus, la fonction `functionThatReturnsAPromise` nous renvoie une `Promise`. On peut donc utiliser sa fonction `then()` en lui passant une fonction qui sera exécutée dès qu'un résultat sera reçu (avec le résultat en question passé à notre fonction). On peut aussi utiliser sa fonction `catch()` en lui passant une fonction qui sera exécutée si une erreur est survenue (avec l'erreur en question passée à notre fonction).

Le gros avantage est que l'on peut aussi **chaîner** les `Promises`. Ainsi, la valeur que l'on retourne dans la fonction que l'on passe à `then()` est transformée en une nouvelle `Promise` résolue, que l'on peut utiliser avec une nouvelle fonction `then()`. Si notre fonction retourne par contre une exception, alors une nouvelle `Promise` rejetée est créée et on peut l'intercepter avec la fonction `catch()`. Mais si la fonction que l'on a passée à `catch()` retourne une nouvelle valeur, alors on a à nouveau une `Promise` résolue que l'on peut utiliser avec une fonction `then()`, etc.

Voici un exemple qui vous montre comment on peut profiter des `Promise` pour chaîner notre code asynchrone :

javascript

```javascript
returnAPromiseWithNumber2()
    .then(function(data) { // Data is 2
        return data + 1;
    })
    .then(function(data) { // Data is 3
        throw new Error('error');
    })
    .then(function(data) {
        // Not executed
    })
    .catch(function(err) {
        return 5;
    })
    .then(function(data) { // Data is 5
        // Do something
    });
```

Dans l'exemple ci-dessus, la fonction `returnAPromiseWithNumber2` nous renvoie une `Promise` qui va être résolue avec le nombre `2`. La première fonction `then()` va récupérer cette valeur. Puis, dans cette fonction on retourne `2 + 1`, ce qui crée une nouvelle `Promise` qui est immédiatement résolue avec `3`. Puis, dans le `then()` suivant, nous retournons une erreur. De ce fait, le `then()` qui suit ne sera pas appelé et c'est le `catch()` suivant qui va être appelé avec l'erreur en question. Lui-même retourne une nouvelle valeur qui est transformée en `Promise` qui est immédiatement résolue avec la valeur `5`. Le dernier `then()` va être exécuté avec cette valeur.

## Gestion des erreurs

Nous avons déjà vu comment se gèrent les erreurs avec les `Promises` . Une erreur correspond à une **exception** qui a été lancée, et il est possible de l'intercepter en appelant la fonction `catch()` de la `Promise` .

## 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.

javascript

```
async function fonctionAsynchrone1() {/* code asynchrone */}
async function fonctionAsynchrone2() {/* code asynchrone */}

async function fonctionAsynchrone3() {
 const value1 = await fonctionAsynchrone1();
 const value2 = await fonctionAsynchrone2();
 return value1 + value2;
}
```

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.

> `async` / `await` utilisent les Promises en arrière-plan, il est donc possible d'utiliser les 2 en même temps.

### Gestion des erreurs

async / await utilisant les Promises, la levée d'une erreur se fait aussi par une **exception**.

Pour intercepter cette erreur, par contre, il suffit d'exécuter notre code asynchrone dans un bloc `try {} catch (e) {}` , l'erreur étant envoyée dans le `catch` .

## Pratiquez !

Rendez-vous sur cet éditeur CodePen pour réaliser l'exercice suivant.

Dans cet exercice j'ai créé 2 fonctions asynchrones (avec le mot clé `async` ) `getNumber1()` et `getNumber2()`

1. Dans un premier temps nous allons 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.
2. Maintenant nous allons appeler 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 résumé

Dans ce chapitre, vous avez appris :

- Ce qu'est une callback ;
- Ce que sont les `Promises`
- Comment utiliser `async` et `await`

*Nous connaissons maintenant 3 techniques pour faire du code asynchrone et pouvoir utiliser sa valeur ; voyons maintenant comment ça peut nous servir dans le cas de plusieurs requêtes HTTP !*

MARK THIS CHAPTER AS UNFINISHED

COMPRENEZ COMMENT FONCTIONNE L'ASYNCHRONE EN JS

PARALLÉLISEZ PLUSIEURS REQUÊTES HTTP

## Teacher

**Fabien Henon**

Diplômé d'un master d'expert en informatique à l'école Epitech, et d'un master spécialisé en création d'entreprise et entrepreneuriat.

## Enchaînez des requêtes avec les callbacks

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).*

javascript

```
var GETRequestCount = 0;
var GETRequestResults = [];

function onGETRequestDone(err, result) {
    if (err) throw err;

    GETRequestCount++;
    GETRequestResults.push(result);

    if (GETRequestCount == 2) {
        post(url3, function(err, result) {
            if (err) throw err;

            // We are done here !
        });
    }
}

get(url1, onGETRequestDone);
get(url2, onGETRequestDone);
```

Comme vous pouvez le voir, le code est assez particulier à lire. Il y a d'autres façons d'écrire ce code, mais ça reste une des façons les plus simples et rapides à écrire.

Afin d'exécuter 2 requêtes `GET` en même temps, nous pouvons appeler 2 fois la fonction `get()`. Étant donné que cette fonction est **asynchrone**, elle ne bloquera pas l'exécution du code. Ainsi l'autre fonction `get()` sera aussi appelée alors que la première ne sera pas encore terminée. C'est comme ça qu'on peut avoir 2 requêtes en parallèle.

Par contre, nous voulons exécuter une requête `POST` une fois que les **2 requêtes** `GET` **sont terminées**, et pas avant ! Pour ce faire, nous devons savoir si les requêtes `GET` sont terminées. C'est pour ça que la variable `GETRequestCount` est créée. On va l'**incrémenter** dans la fonction *callback* que l'on a envoyée aux appels à `get()`, et si on atteint 2 (le nombre de requêtes `GET` qu'on a faites), alors on va exécuter la requête `POST`.

`GETRequestResults` sert à conserver les réponses des requêtes `GET`, car on ne les a pas toutes les 2 en même temps.

## Enchaînez des requêtes avec les Promises

Grâce à la fonction `Promise.all`, voyons comment exécuter nos requêtes en parallèle et en séquence avec les `Promises`.

*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. Ces fonctions retourneront une* `Promise` *avec le résultat de la requête.*

javascript

```
Promise.all([get(url1), get(url2)])
    .then(function(results) {
        return Promise.all([results, post(url3)]];
    })
    .then(function(allResults) {
        // We are done here !
    });
```

Ici, nous utilisons la fonction `Promise.all` qui prend en paramètre une liste de `Promise` (cela peut aussi être de simples valeurs qui sont alors transformées en `Promise` résolues), et qui permet de toutes les exécuter en parallèle et de retourner une nouvelle `Promise` qui sera résolue quand toutes les `Promise` seront résolues.

Ainsi, la fonction `then()` recevra les résultats de toutes les `Promise` sous forme d'un tableau.

Afin d'exécuter notre requête `POST` une fois que les requêtes `GET` sont terminées, nous l'exécutons donc dans la fonction `then()`.

Notez que dans la fonction `then()`, nous faisons encore une fois appel à la fonction `Promise.all` en lui passant les résultats des requêtes `GET` et notre requête `POST`. Étant donné que `Promise.all` considère les simples valeurs comme des `Promise` résolues, cela nous permet, dans le prochain `then()`, de récupérer une liste qui contient les résultats des requêtes `GET` et le résultat de la requête `POST` : `allResults = [ [ getResult1, getResult2 ], postResult ]`.

## Enchaînez des requêtes avec async/await

Finalement, voyons comment exécuter le même code mais avec `async` / `await`.

*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. Ces fonctions sont asynchrones (avec le mot clé* `async` *).*

javascript

```
async function requests() {
    var getResults = await Promise.all([get(url1), get(url2)]);
    var postResult = await post(url3);
    return [getResults, postResult];
}

requests().then(function(allResults) {
    // We are done here !
});
```

Nous utilisons aussi la fonction `Promise.all` dans ce code, car c'est comme ça que l'on peut exécuter des fonctions asynchrones en parallèle (rappelez-vous que `async` correspond en arrière-plan à une `Promise` ).

Par contre, ici, nous utilisons `await` devant `Promise.all` afin d'attendre la fin de l'exécution des 2 requêtes `GET` , puis nous utilisons `await` devant la requête `POST` afin d'attendre son résultat. Puis nous renvoyons un tableau avec tous les résultats.

Lorsque nous appelons la fonction `requests()` , ici, nous utilisons `then()` pour récupérer tous les résultats (mais vous auriez aussi pu utiliser `await` au sein d'une autre fonction avec le mot clé `async` ).

## En résumé

Dans ce chapitre, vous avez appris :

- À enchaîner des requêtes avec la méthode des callbacks ;
- À enchaîner des requêtes avec la méthode des `Promises`
- À enchaîner des requêtes avec la méthode `async` / `await`

Vous connaissez maintenant 3 techniques pour exécuter du code asynchrone et vous savez les mettre en situation. À vous maintenant de vous faire votre propre avis sur celle que vous préférez utiliser. 😀

*Il est temps maintenant de passer à la prochaine partie où nous verrons comment coder en JavaScript dans un environnement plus efficace.* 🤓

Que pensez-vous de ce cours ? Donnez-nous votre avis **ici**.