# JSON

# What is it?

## JSON

*“Le JavaScript Object Notation (JSON) est un format standard utilisé pour représenter des données structurées de façon semblable aux objets Javascript.”*

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# What is it?

## JSON

- JSON est un format de données, fortement inspiré du JS
- Il dispose de sa propre syntaxe et est indépendant du Javascript
- La plupart des langages de programmation fournissent des librairies pour permettre de le convertir en des objets utilisables ou d'en générer pour l'exporter au sein d'un autre langage
- JSON se stock dans des chaînes de caractères

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# Quand l'utiliser ?

## JSON

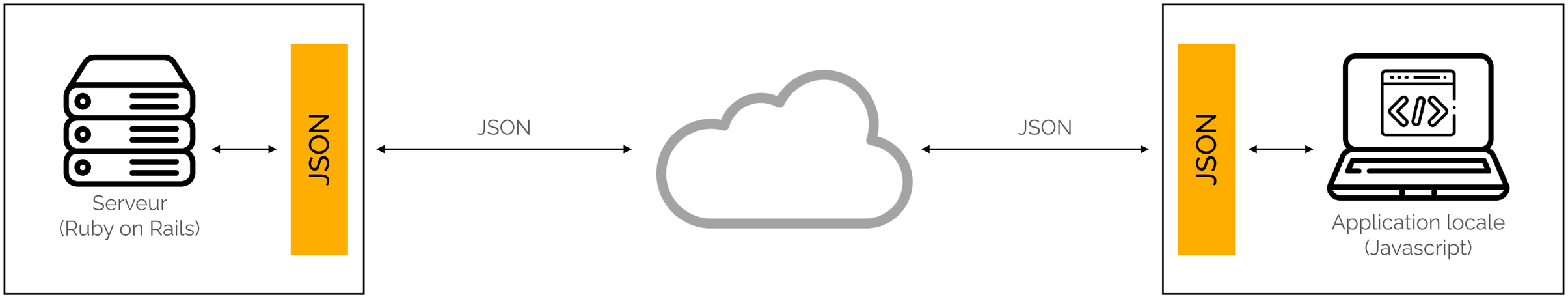
- Idéalement, lorsque deux systèmes séparés doivent s'échanger des informations (par ex. une API entre Javascript et PHP)

- Quand il faut stocker des données structurées en dehors du langage lui-même (par ex. enregistrer un tableau dans un fichier texte)

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# Usecase

## JSON



# Exemple

## JSON

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    "Molecule Man",
   "Madame Uppercut"
  ]
}
```

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# Comment l'utiliser

## JSON

- JSON supporte les types de données standards, comme:
  - Chaîne de caractères
  - Nombres
  - Null
  - Booléen

- Il y a également deux types de données structurées :
  - Tableau
  - Objet (clé/valeur, comme en js)

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# Comment l'utiliser

## JSON

Deux méthodes principales :

- `JSON.parse("...")`
  Converti une chaîne de caractères JSON en un objet javascript

- `JSON.stringify(unevariable)`
  Converti une variable Javascript en un objet JSON

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# Comment l'utiliser - JSON.parse

## JSON

```
const monJson = '{ "cours": "WebmobUI", "lieu": "HEIG-VD"}'

const monJsonparsé = JSON.parse(monJson)

console.log(monJsonparsé.cours) => "WebmobUI"

console.log(monJsonparsé.lieu) => "HEIG-VD"
```

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# Comment l'utiliser - JSON.stringify

## JSON

```
const monObjet = { cours: 'WebmobUI', lieu: 'HEIG-VD' }

console.log(monObjet.cours) => "WebmobUI"

console.log(monObjet.lieu) => "HEIG-VD"

const monJson = JSON.stringify(monObjet)

console.log(monJson) => '{ "cours": "WebmobUI", "lieu": "HEIG-VD"}'
```

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

# L’API Spotlified

# Concept

## L’API Spotlified

- L’API Spotlified tourne sur un serveur distant: https://webmob-ui-22-spotlified.herokuapp.com/

- Tous les endpoints retournent du JSON qui sera à “parser” par vos soins

- Les URLs sont dites “REST” pour plus de clarté

# Endpoints

## L'API Spotlified

3 endpoints principaux :

- Lister les artistes
- Listes les chansons d'un artiste
- Rechercher une chanson par texte libre

# Endpoints – Lister les artistes

## L'API Spotlified

URL: https://webmob-ui-22-spotlified.herokuapp.com/api/artists

Response: Tableau d'artistes

Exemple:

```
[
  { "id": 2, "name": "Alan Walker", image_url: "https://...." },
  { "id": 3, "name": "Dynoro", image_url: "https://...." }
]
```

# Endpoints – Lister les chansons d'un artiste

## L'API Spotlified

URL: https://webmob-ui-22-spotlified.herokuapp.com/api/artists/:id/songs

Response: Tableau de chanson, avec l'artiste dans la chanson

Exemple:


```
[
  { "id": 2, "title": "Faded", audio_url: "https://....", "artist": { … }},
  { "id": 3, "title": "Spectre", audio_url: "https://....", "artist": { … }}
]
```

# Endpoints – Rechercher une chanson par texte libre

## L'API Spotlified

URL: https://webmob-ui-22-spotlified.herokuapp.com/api/songs/search/:query

Response: Tableau de chanson, avec l'artiste dans la chanson

Exemple:


```
[
  { "id": 2, "title": "Faded", audio_url: "https://....", "artist": { … }},
  { "id": 3, "title": "Spectre", audio_url: "https://....", "artist": { … }}
]
```

# API fetch

# API Fetch

## API Fetch

- L'API fetch permet de charger du contenu depuis une URL
- Elle est dite asynchrone
- Asynchrone signifie que le code suivant l'appel à fetch continuera son exécution sans n'avoir encore de résultat
- L'API fetch est basé sur les Promises
- Il est possible de synchroniser partiellement des promises en utilisant les mots-clés `async`/`await`

# API Fetch

## API Fetch

- L'API fetch retourne une promise
- Par défaut, la fonction passée à then prend un argument : Response
- Response est la réponse reçue par le navigateur, sous forme d'objet, avec différentes méthodes et attributs
- Par ex:
  `response.ok` => true|false
  `response.json()` => retourne le body de la réponse, déjà parsé par JSON

# Promises

## API Fetch

- Les Promises sont des promesses. Leurs appels vous promettent d'obtenir une réponse en retour, dans un avenir proche
- Une promesse peut être soit réalisée (resolved) ou rejetée (rejected)
- Vous avez la possibilité de définir les actions à effectuer, selon sa réalisation ou son rejet
- Vous utilisez les promesses dans la vie de tous les jours…

# Promises – Exemple

## API Fetch

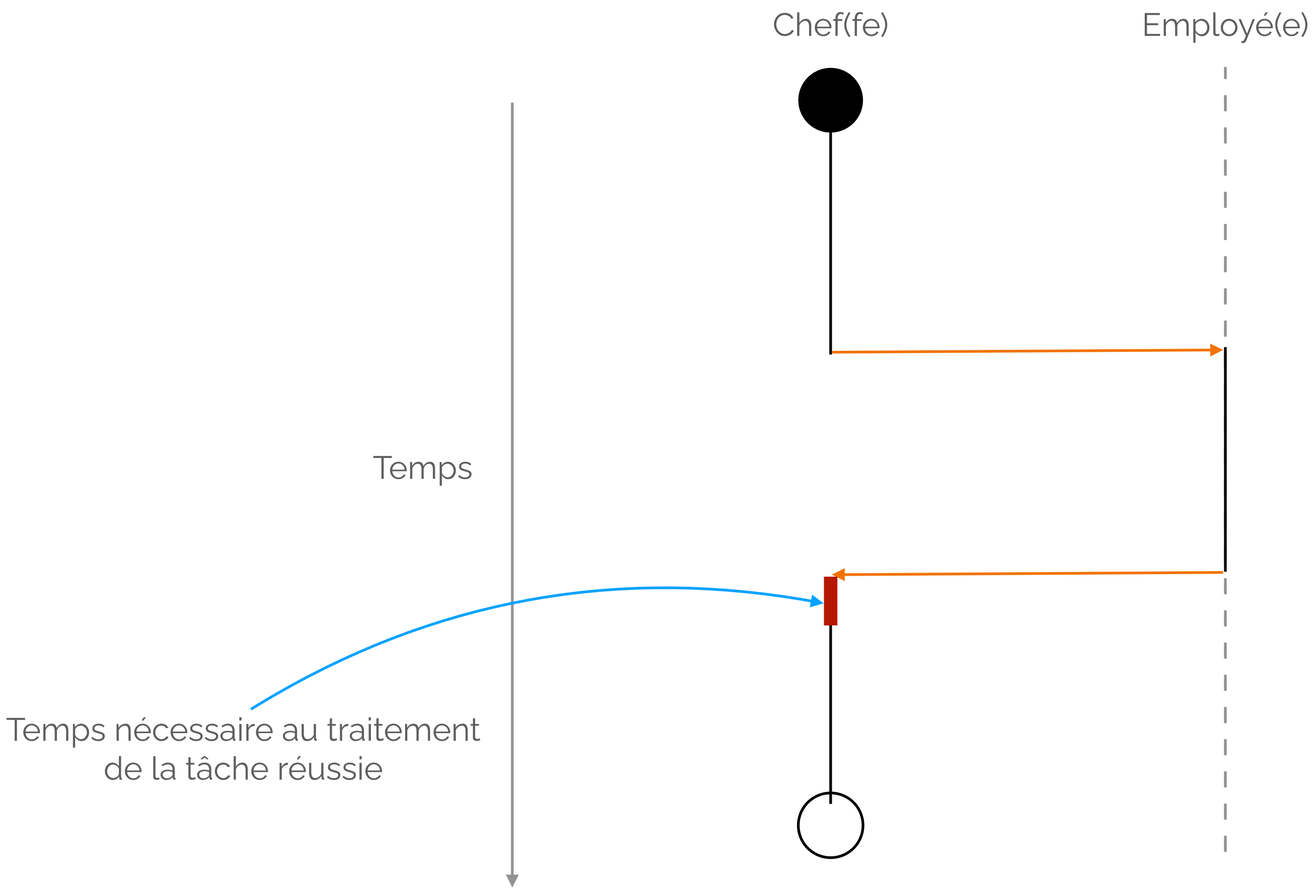
- Supposons que vous êtes chef(fe) de service et que vous donnez une tâche à l'un(e) de vos employé(e)s

- Vous retournez ensuite à votre bureau, continuer votre travail jusqu'à ce que la personne vienne toquer à votre porte, dossier en main, pour vous signaler que la tâche est terminée

- Même constat lorsque vous commandez un colis, vous ne restez pas devant votre boite-aux-lettres, jusqu'à avoir reçu celui-ci. Il arrive quand il arrive.

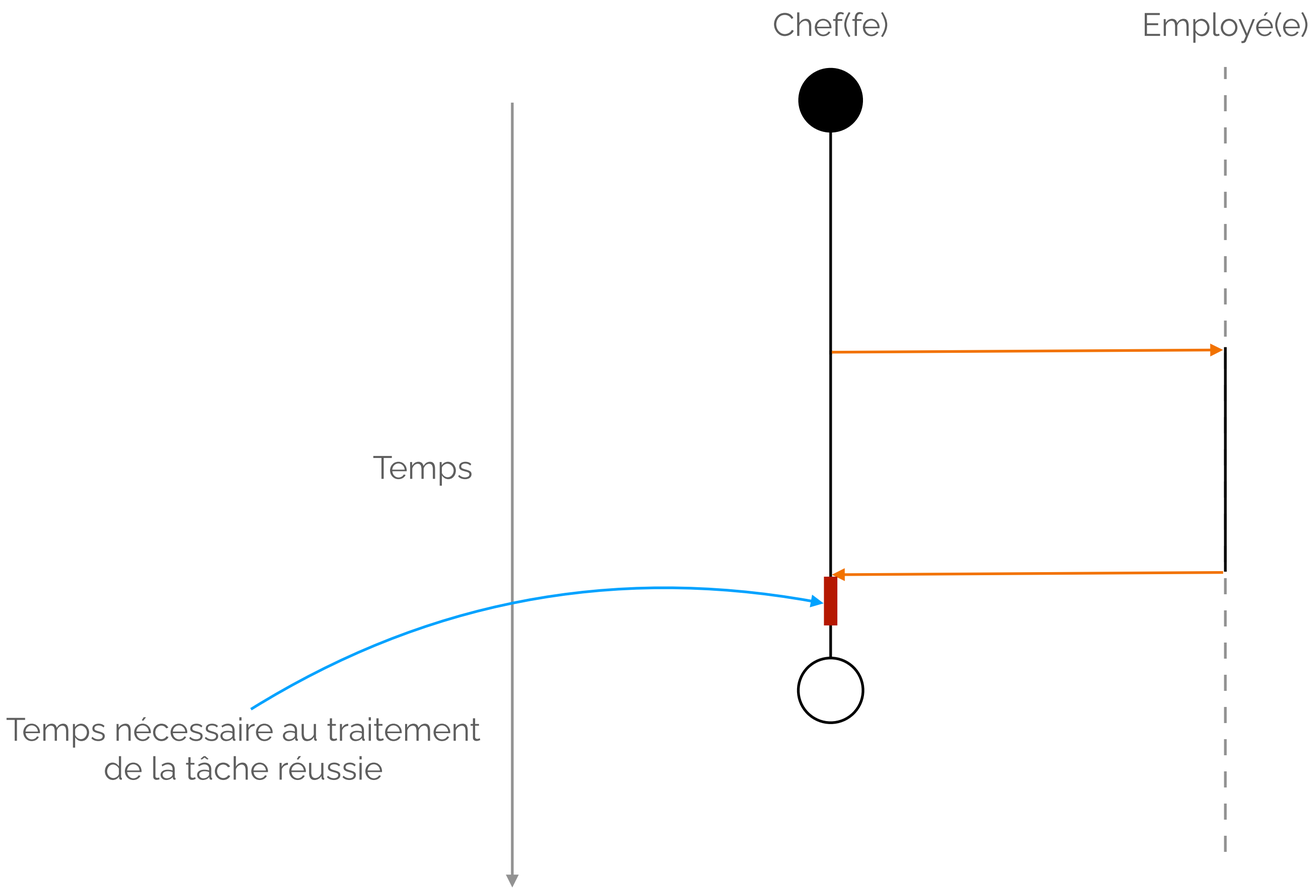
# Promises – Code synchrone

## API Fetch



# Promises – Code asynchrone

## API Fetch



# Promises – Synchrone

## API Fetch

Prenons l'exemple de code suivant...

```
console.log('Hello 1')

uneFonctionSynchroneClassiqueQuiAfficheHello2()

console.log('Hello 3')
```

Console

```
Hello 1
Hello 2
Hello 3
```

# Promises – Asynchrone

## API Fetch

Que se passe-t-il avec une méthode asynchrone ?

```
console.log('Hello 1')
unePromiseAsynchroneQuiAfficheHello2()
console.log('Hello 3')
```

Console

```
Hello 1
Hello 2
Hello 3
```

OU

Console

```
Hello 1
Hello 3
Hello 2
```

Un des deux… On ne sait pas…

Comment s'assurer que "Hello 3" ne sera affiché qu'après "Hello 2" ?

# Promises – then/catch/finally

## API Fetch

- Il existe trois types de méthodes utilisables sur une Promise et prennent toute une fonction en paramètre:
  - then - “Ensuite” - La fonction passée sera appelé lorsque tout se passe bien
  - catch - La fonction passée sera appelé lorsqu’il y a une erreur
  - finally - La fonction passée sera appelé dans les deux cas

# Promises – then/catch/finally

## API Fetch

```
console.log('Hello 1')

unePromiseAsynchroneQuiAfficheHello2()

console.log('Hello 3')
```

Extraire le code devant s'exécuter après, dans une fonction, au sein d'un "then"

```
console.log('Hello 1')

unePromiseAsynchroneQuiAfficheHello2()
   .then(() => console.log('Hello 3'))
```

Console

```
Hello 1

Hello 2

Hello 3
```

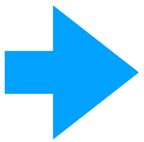
# Promises – then/catch/finally

## API Fetch

- Les promises permettent de structurer le code de manière claire, en donnant des instructions précises, selon le déroulement des événements

- Elles permettent surtout de ne pas bloquer l'exécution de la page (par exemple pendant le chargement de la liste des artistes) et d'avoir une expérience plus fluide

```
console.log('Hello 1')

unePromiseAsynchroneQuiAfficheHello2()

   .then(() => console.log('Hello 3'))
```



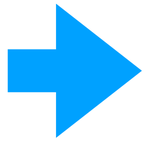
```
AfficheHello1()

AfficheHello2()

   .ensuite(() => afficheHello3())
```

# Promises – Chaînage

## API Fetch

- Les promises sont construites sur le concept du chaînage
- Il est possible de mettre plusieurs then/catch/finally à la suite

```
console.log('Hello 1')
unePromiseAsynchroneQuiAfficheHello2()
   .then(() => console.log('Hello 3'))
   .then(() => console.log('Hello 4'))
   .catch(() => console.log('Erreur !'))
   .finally(() => console.log('Terminé'))
```



```
AfficheHello1()
AfficheHello2Asynchrone()
   .ensuite(() => afficheHello3())
   .ensuite(() => afficheHello4())
   .siErreur(() => afficheErreur())
   .quoiQuIlArrive(() => afficheTerminé())
```

# Promises – Chaînage

## API Fetch

- Le chaînage permet également de transformer l'information au fur et à mesure et la passer à l'étape suivante

- Chaque étape prend en argument la valeur de retour de la fonction précédente

# Promises – Chaînage

## API Fetch

- Supposons que l'on souhaite charger les artistes et afficher le premier artiste…

```
fetch('http…/api/artists') // Va charger les artistes sur le serveur et retourne la réponse par défaut
   .then((response) => {
      const artistes = response.json() // On prend la réponse de base et on la converti en JSON.
                                       // Cela retournera un tableau d'artistes
      const artist = artistes[0] // On prend le premier élément du tableau artistes
      console.log(artist) // On affiche le premier artiste
   })
```

# Promises – Chaînage

## API Fetch

- Le code suivant serait équivalent !

```
fetch('http…/api/artists') // Va charger les artistes sur le serveur et retourne la réponse par défaut
   .then((response) => response.json()) // On prend la réponse de base et on la converti en JSON.
                                        // Cela retournera un tableau d'artistes
   .then((artists) => artists[0]) // On prend le tableau retourné par la méthode précédente et
                                  // on retourne le premier artiste
   .then((artist) => console.log(artist)) // On affiche le premier artiste retourné par la méthode précédente
```

# Promises – Chaînage

## API Fetch

- Et avec les autres méthodes ?

```
afficherRondDeChargement()
fetch('http…/api/artists')
   .then((response) => response.json())
   .then((artists) => artists[0])
   .then((artist) => console.log(artist))
   .catch(() => alert('Il y a eu un problème avec le serveur !')) // On attrape l'erreur du fetch et on affiche un message
   .finally(() => cacherRondDeChargement()) // Succès ou erreur, on cache le rond de chargement, car la promise est terminée
```

# Promises – Langage fonctionnel

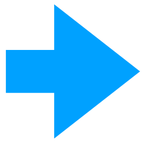
## API Fetch

- Quel intérêt d'utiliser autant de fonctions ?
- Javascript est un langage "Fonctionnel" -> Basé sur les fonctions et est hautement performant dans leur gestion
- Il n'est pas obligatoire d'utiliser des fonctions fléchées dans les then. N'importe quelle référence vers une fonction est acceptée
- Plus l'on sépare le code en fonctions, plus celui-ci sera clair et concis… et plus notre ami Déméter sera content.

# Promises – Langage fonctionnel

## API Fetch

```
AfficheHello1()

AfficheHello2()
   .ensuite(() => afficheHello3())
   .ensuite(() => afficheHello4())
   .siErreur(() => afficheErreur())
   .quoiQuIlArrive(() => afficheTerminé())
```



```
AfficheHello1()

AfficheHello2()
   .ensuite(afficheHello3)
   .ensuite(afficheHello4)
   .siErreur(afficheErreur)
   .quoiQuIlArrive(afficheTerminé)
```

# Promises – Langage fonctionnel

## API Fetch

```
// api.js
function chargerArtistes() {
   return fetch('http…/api/artists')
            .then((response) => response.json())
}
```

```
// section_artistes.js
import { chargerArtistes } from 'api.js'

function afficherArtistes(artistes) {
   for(const artiste of artistes){
      …
   }
}

function afficherSectionArtistes() {
   chargerArtistes().then(afficherArtistes)
}
```

# Promises – Langage fonctionnel

## API Fetch

Faisons plaisir à Déméter…

```
// api.js


function fetchJson(url) {
   return fetch(url)
            .then((response) => response.json())
}
function chargerArtistes() {
   return fetchJson('http…/api/artists')
}
```

```
// section_artistes.js
import { chargerArtistes } from 'api.js'
function afficherArtiste(artiste) {
   …
}
function afficherArtistes(artistes) {
   artistes.forEach(afficherArtiste)
   // ou
   for(const artiste of artistes){
      afficherArtiste(artiste)
   }
}
function afficherSectionArtistes() {
   chargerArtistes().then(afficherArtistes)
}
```

# Promises – Rendre synchrone

## API Fetch

- Il est possible de rendre des promises (presque) synchrones en utilisant les mots-clés async/await

- Le mot clé await mis devant l'appel à une promise la rend synchrone et permet de l'utiliser comme une fonction classique

```
const résultat = await mapromise()
```

- "Presque", car await ne peut être utilisé seul. Il doit obligatoirement être utilisé au sein d'une fonction async (donc au sein d'une promise)

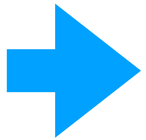
- Cette action consiste simplement à remonter la promise d'un niveau

# Promises – Rendre synchrone

## API Fetch

```
// fichier.js

const résultat = await fetch('…')
```

Erreur ! Pas possible au root d'un fichier



```
// fichier.js

async function chargerSyncrone() {

   const résultat = await fetch('…')

}

chargerSyncrone()
```

Possible, car englobé dans une fonction async

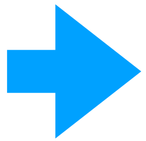
# Promises – Rendre synchrone

## API Fetch

```
console.log('Hello 1')
unePromiseAsynchroneQuiAfficheHello2()
console.log('Hello 3')
```

```
async function afficherHellos() {
   console.log('Hello 1')
   await unePromiseAsynchroneQuiAfficheHello2()
   console.log('Hello 3')
}
afficherHellos()
```

Console



```
Hello 1
Hello 2
Hello 3
```

# Promises – Rendre synchrone

## API Fetch

```
console.log('Hello 1')
unePromiseAsynchroneQuiAfficheHello2()
   .then(() => console.log('Hello 3’))
   .then(() => console.log('Hello 4’))
   .catch((e) => console.log(‘Erreur !’, e))
   .finally(() => console.log('Terminé'))
```

```
async function afficherHellos() {
   try {
      console.log('Hello 1')
      await unePromiseAsynchroneQuiAfficheHello2()
      console.log('Hello 3’)
      console.log('Hello 4’)
   }
   catch (e) {
      console.log(‘Erreur !’, e)
   }
   finally{
      console.log('Terminé')
   }
}
afficherHellos()
```

# Promises – Rendre synchrone

## API Fetch

```
fetch('http…/api/artists')
   .then((response) => {
      const artistes = response.json()
      const artist = artistes[0]
      console.log(artist)
   })
```

```
async function afficherArtiste() {
   const response = await fetch('http…/api/artists')
   const artistes = await response.json()
   const artist = artistes[0]
   console.log(artist)
}
afficherArtiste()
```

# Promises – Rendre synchrone

## API Fetch

```
afficherRondDeChargement()
fetch('http…/api/artists')
   .then((response) => response.json())
   .then((artists) => artists[0])
   .then((artist) => console.log(artist))
   .catch((e) => alert('Il y a eu un problème!'))
   .finally(() => cacherRondDeChargement())
```

```
async function afficherArtiste() {
   afficherRondDeChargement()
   try {
      const response = await fetch('http…/api/artists')
      const artistes = response.json()
      const artist = artistes[0]
      console.log(artist)
   }
   catch (e) {
      alert('Il y a eu un problème!')
   }
   finally {
      cacherRondDeChargement()
   }
}
afficherArtiste()
```

# Promises – Async

## API Fetch

- Au fond, à quoi sert le mot clé async ?
- Il converti une fonction en promise… Promiseception…

```
function afficherArtiste() {
   …
}
afficherArtiste().then(…)
```

Erreur ! C'est une fonction classique

```
async function afficherArtiste() {
   …
}
afficherArtiste().then(…)
```

Possible ! afficherArtiste est devenu une promise…

# Promises – Async/await

## API Fetch

then ou async/await ?

# Templating

# Concept

## Templating

- Le templating permet de définir un squelette de base à utiliser pour des éléments dynamiques
- Exemple: Un élément `<div>` dans la liste des artistes
- Il suffit ensuite de dupliquer cet élément vide autant de fois que nécessaire pour afficher la liste complète

# Plusieurs écoles

## Templating

- Construire le HTML à la volée - utiliser .innerText ou (similaire) et interpoler le contenu

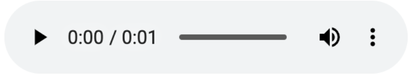
- Utiliser le templating manuel - Garder un élément vide, le cloner, modifier son DOM et l'insérer dans l'élément parent

- Utiliser un moteur de template - Un markup spécifique, interprété par une librairie qui gère le remplacement des zones à éditer et va l'intégrer dans l'élément parent Exemple: Handlebars, JSX, ...

- Se servir de HTML5...

# Shadow DOM

## Templating

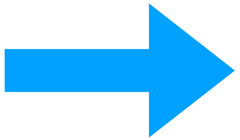
- “Black box”
- Encapsulation et abstraction du DOM contenu dans un élément
- Principalement utilisé par les browsers pour abstraire des éléments complexes
- Exemple: `<audio />` 


https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

# Shadow DOM

## Templating

```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
    <audio src="http://...."></audio> == $0
  </body>
</html>
```



```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
    ▼<audio src="http://...."> == $0
      #shadow-root (user-agent)
        ▼<div pseudo="-webkit-media-controls" class="phase-pre-ready state-no-source"> flex
          ▼<div pseudo="-webkit-media-controls-overlay-enclosure">
            ▼<input pseudo="-internal-media-controls-overlay-cast-button" type="button" aria-label="play on remote device" style="display: none;">
              ▶#shadow-root (user-agent)
              </input>
            </div>
          ▼<div pseudo="-webkit-media-controls-enclosure"> flex
            ▼<div pseudo="-webkit-media-controls-panel" style="display: none;">
              ▶<input type="button" pseudo="-webkit-media-controls-play-button" aria-label="play" class="pause" disabled style="display: none;">…</input>
                <div aria-label="elapsed time: 0:00" pseudo="-webkit-media-controls-current-time-display" style="display: none;">0:00</div>
                <div aria-label="total time: / 0:00" pseudo="-webkit-media-controls-time-remaining-display" style="display: none;">/ 0:00</div>
              ▶<input type="range" step="any" pseudo="-webkit-media-controls-timeline" max="NaN" min="0" aria-label="audio time scrubber 0:00 / 0:00" aria-valuetext=
              "elapsed time: 0:00" disabled>…</input>
              ▶<div pseudo="-webkit-media-controls-volume-control-container" class="closed" style="display: none;">…</div>
              ▶<input type="button" pseudo="-webkit-media-controls-fullscreen-button" aria-label="enter full screen" style="display: none;">…</input>
              ▶<input type="button" aria-label="show more media controls" title="more options" pseudo="-internal-media-controls-overflow-button" style>…</input>
              </div>
            </div>
            <div role="menu" aria-label="Options" pseudo="-internal-media-controls-text-track-list" style="display: none;"></div>
            <div role="menu" aria-label="Options" pseudo="-internal-media-controls-playback-speed-list" style="display: none;"></div>
          ▼<div pseudo="-internal-media-controls-overflow-menu-list" role="menu" class="closed" style="display: none;">
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label=" Play " class="animated-1" style="display: no
            ne;">…</label>
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label="enter full screen Full screen " style="displa
            y: none;">…</label>
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label="download media Download " style="display: non
            e;">…</label>
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label=" Mute " style="display: none;">…</label>
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label="play on remote device Cast " style="display:
            none;">…</label>
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label="show closed captions menu Captions " style="d
            isplay: none;">…</label>
            ▶<label pseudo="-internal-media-controls-overflow-menu-list-item" role="menuitem" tabindex="0" aria-label="show playback speed menu Playback speed "
            class="animated-0" style>…</label> flex
            </div>
          </div>
      </audio>
  </body>
</html>
```

https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

# Custom elements

## Templating

*Here's the magic !*

```
<my-super-custom-element />
```

# Extended custom elements

## Templating

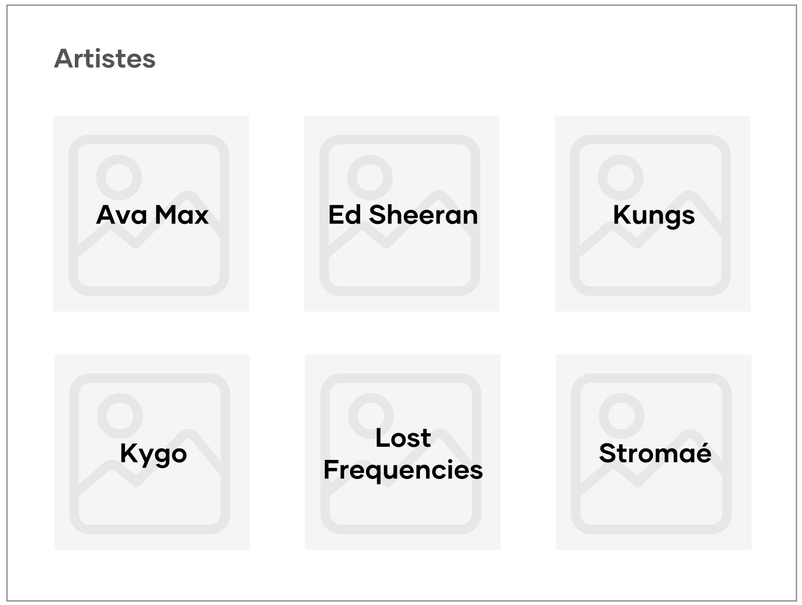
```
<mega-button />
```

```
const MegaButton = customElements.define(
  'mega-button',
   HTMLButtonElement,
  {  extends: ‘button' }
);
```

https://developer.mozilla.org/fr/docs/Web/Web_Components/Using_custom_elements

# Custom elements

## Templating




?

```
<artist-cover>
  ...
</artist-cover>
```

# Attributes

## Templating

```
<artist-item
  id=“1"
  name="Ed Sheeran"
  song-count="32"
  thumbnail-url="https://…"
>
</artist-item>
```

https://developer.mozilla.org/fr/docs/Web/HTML/Element/Input/search

# Templating manuel

## Templating

- Nous allons d'abord utiliser le templating manuel pour la première partie du cours (Spotlified)

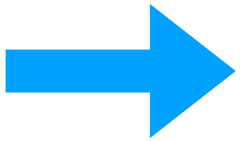
- Comment transformer en CustomElements ?

- Moteur de template en partie 2, avec les frameworks javascript

# Custom Elements

## Templating

```
class BlogPost extends HTMLElement {

   connectedCallback() {

    this.innerHTML = `

      <img src="${this.getAttribute("cover")}" />

      <div>${this.getAttribute("title")}</div>

    `

   }

}

customElements.define("blog-post", BlogPost)
```



```
<blog-post
  title="Voyager"
  cover="https://…"
>
</blog-post>
```

# Custom Elements

## Templating

- La méthode connectedCallback est appelée une fois que l'élément est inséré dans le DOM

- Idéal pour ajouter à ce moment là !

# Custom Elements

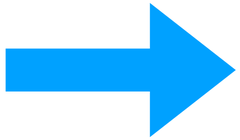
## Templating

- Par défaut, un élément n'est pas mis à jour automatiquement
- Il faut déclarer les attributs qui impactent le rendering et qui mérite de re-render l'élément
- Cela se fait en deux étapes :
  - Déclarer la liste des attributs via “observedAttributes"
  - Ecouter le callback “attributeChangedCallback”

# Custom Elements

## Templating

```
class BlogPost extends HTMLElement {

   static observedAttributes = ['cover', 'title']

   connectedCallback() {

    this.innerHTML = `

      <img src="${this.getAttribute("cover")}" />

      <div>${this.getAttribute("title")}</div>

    `

   }

   attributeChangedCallback() {

    this.innerHTML = `

      <img src="${this.getAttribute("cover")}" />

      <div>${this.getAttribute("title")}</div>

    `

   }

}

customElements.define("blog-post", BlogPost)
```



```
<blog-post
  title="Voyager"
  cover="https://…"
>
</blog-post>
```

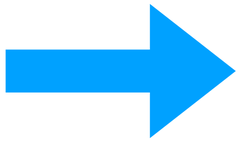
# Custom Elements

## Templating

# Custom Elements

## Templating

```
class BlogPost extends HTMLElement {
   static observedAttributes = ['cover', 'title']
   connectedCallback() {
      this.render()
   }
   attributeChangedCallback() {
      this.render()
   }
   render() {
    this.innerHTML = `
      <img src="${this.getAttribute("cover")}" />
      <div>${this.getAttribute("title")}</div>
     `
   }
}
customElements.define("blog-post", BlogPost)
```



```
<blog-post
  title="Voyager"
  cover="https://…"
>
</blog-post>
```

# Goal

## Templating

- Être capable de charger la liste des artistes
- Les afficher à l'emplacement prévu, grâce à une boucle et avec le bon contenu
- Adapter votre méthode pour utiliser les custom elements

# Architecture de code

# Modularité et responsabilité

## Architecture de code

- Une architecture est dite “modulaire” quand elle est séparée en plusieurs modules, avec des responsabilités précises

- Chaque module a une implémentation privée qui lui est propre et une implémentation publique pour interagir avec les autres modules

- Chaque module a des voisins directs, avec qui il a une forte interaction, et d’autres voisins indirects avec lesquels il échange par le biais d’autres modules

- Cela s’applique aussi bien à une vue globale qu’à une vue détaillée

# Modularité et responsabilité – Exemple

## Architecture de code


- Le CSS est responsable de la mise en page
- Il offre le langage CSS comme implémentation publique (on lui dit quoi faire)
- Il a comme voisin direct le HTML, car il met en page des éléments du langageHTML

- Le HTML est responsable de la structure de la page
- Il offre le langage HTML comme implémentation publique
- Il a comme voisin direct le CSS, car il lui fournit les éléments à mettre en page
- Le JS comme autre voisin direct, car il interagit avec les éléments HTML

- Le CSS est responsable de la dynamique de la page
- Il offre le langage JS comme implémentation publique
- Il a comme voisin direct le HTML, car il utilise les éléments HTML pour les rendre dynamique
- Le CSS n'est pas un voisin direct, car le JS va d'abord utiliser un élément HTML pour y ajouter des styles

# Loi de Déméter – Principe de connaissance minimale

## Architecture de code

*« Ne parlez qu'à vos amis immédiats ».*

# Loi de Déméter – Principe de connaissance minimale

## Architecture de code

- La loi de Déméter vise à limiter les connaissances de chaque module
- Le but est de diminuer les dépendances et donc la complexité
- Cela permet une plus grande flexibilité et une notion d'agilité

# Loi de Déméter – Principe de connaissance minimale

## Architecture de code


- Un client qui aurait besoin d'un conseille au sein d'une entreprise, appelle la réception qui va l'aiguiller vers la personne adaptée à sa demande

- Le client ne se soucie pas de la liste des employés, leur présence, changements, etc…

# Loi de Déméter – Dans la pratique

## Architecture de code

On souhaite afficher une <section />


```css
section {
  /* display: flex */
  display: none;
}
```

```html
<section id="ma-section">
  …
</section>
```

```js
const section = document.querySelector('#ma-section')
section.style.display = 'flex'
```

Pas très “Déméter”… Le JS va directement modifier le CSS.
Que se passe-t-il si cette section devient un block et plus un flex ?

# Loi de Déméter - Dans la pratique

## Architecture de code

On souhaite afficher une <section />


```css
section {
  display: none;
}
section.active {
  display: flex;
}
```

```html
<section id="ma-section">
  …
</section>
```

```js
const section = document.querySelector('#ma-section')
section.classList.add('active')
```

Mieux ! Le JS ne fait qu'ajouter un attribut HTML (une classe) et il ne s'occupe pas du mode d'affichage.

C'est le CSS qui va gérer le changement -> Les responsabilités sont respectées !

# Loi de Déméter – Dans la pratique 2

## Architecture de code

Taille du texte en CSS


- S'occupe d'afficher le player
- La taille exacte du texte ne l'intéresse pas !

- S'occupe de gérer la taille du texte pour le monde
- Le player ne l'intéresse pas !

# Loi de Déméter – Changement de section

## Architecture de code

Plutôt **V1** ou **V2** ?

# Structure de l’application

## Architecture de code

Ou d’autres dans l’application ?

# Schémas UML

## Architecture de code

- Les schémas UML (**U**nified **M**odeling **L**anguage) sont des schémas standardisés pour représenter une application

- Il en existe plus d'une vingtaine…

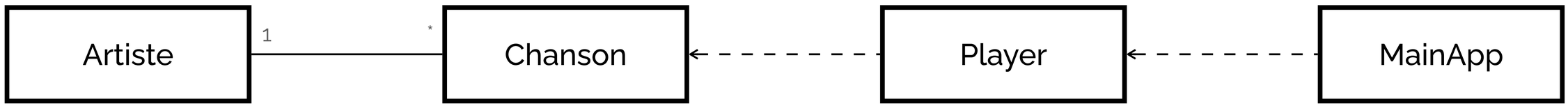
- Nous allons utiliser les diagrammes de classe

https://fr.wikipedia.org/wiki/Diagramme_de_classes

# Schémas UML – Diagramme de classe

## Architecture de code

- Chaque classe (ou modèle) est représenté par un rectangle avec le nom de la classe
- Optionnellement, la liste de ses attributs et de ses fonctions
- Les éléments sont connectés ensemble par plusieurs types de flèches, représentant le type d'interaction (héritage, composition, …)




https://fr.wikipedia.org/wiki/Diagramme_de_classes

# Spotlified

## Architecture de code

Quels sont les éléments importants de l'application ?

# Templating bis

# Templating manuel

## Templating

- Où stocker les éléments vides ?
- HTML nous offre un tag prêt à l'emploi : `<template></template>`
- Il s'agit en gros d'une div cachée… avec 1-2 détails en plus

https://blog.webdevsimplified.com/2020-06/template-tag/

# Exemple

## Templating

```
<div class=“blog-post-list“>
  <a href=““>
    <img src="http://placekitten.com/200/300" />
    <div>Ava Max</div>
  </a>
  …
</div>
```

```
<div class=“blog-post-list“>
</div>
<template id=“blog-post-template”>
  <a href=““>
    <img src="" />
    <div></div>
  </a>
</template>
```

https://blog.webdevsimplified.com/2020-06/template-tag/

# Exemple – Before

## Templating

```
Const blogPostTemplate = document.querySelector(‘#blog-post-template’)

class BlogPost extends HTMLElement {

   connectedCallback() {

     this.innerHTML = `

        <img src=“${this.getAttribute(“cover”)}” />

        <div>${this.getAttribute(“title”)}</div>

       `

   }

}

customElements.define("blog-post", BlogPost)
```

```
<blog-post
  title="Voyager"
  cover="https://…"
>
</blog-post>
```

# Exemple - After

## Templating

```
Const blogPostTemplate = document.querySelector(‘#blog-post-template’)

class BlogPost extends HTMLElement {

   connectedCallback() {

     this.replaceChildren(blogPostTemplate.content.cloneNode(true))

     this.querySelector(‘img’).src = this.getAttribute(“cover”)

     this.querySelector(‘div’).innerText = this.getAttribute(“title”)

   }

}

customElements.define("blog-post", BlogPost)
```

```
<blog-post
  title="Voyager"
  cover="https://…"
>
</blog-post>
```