

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

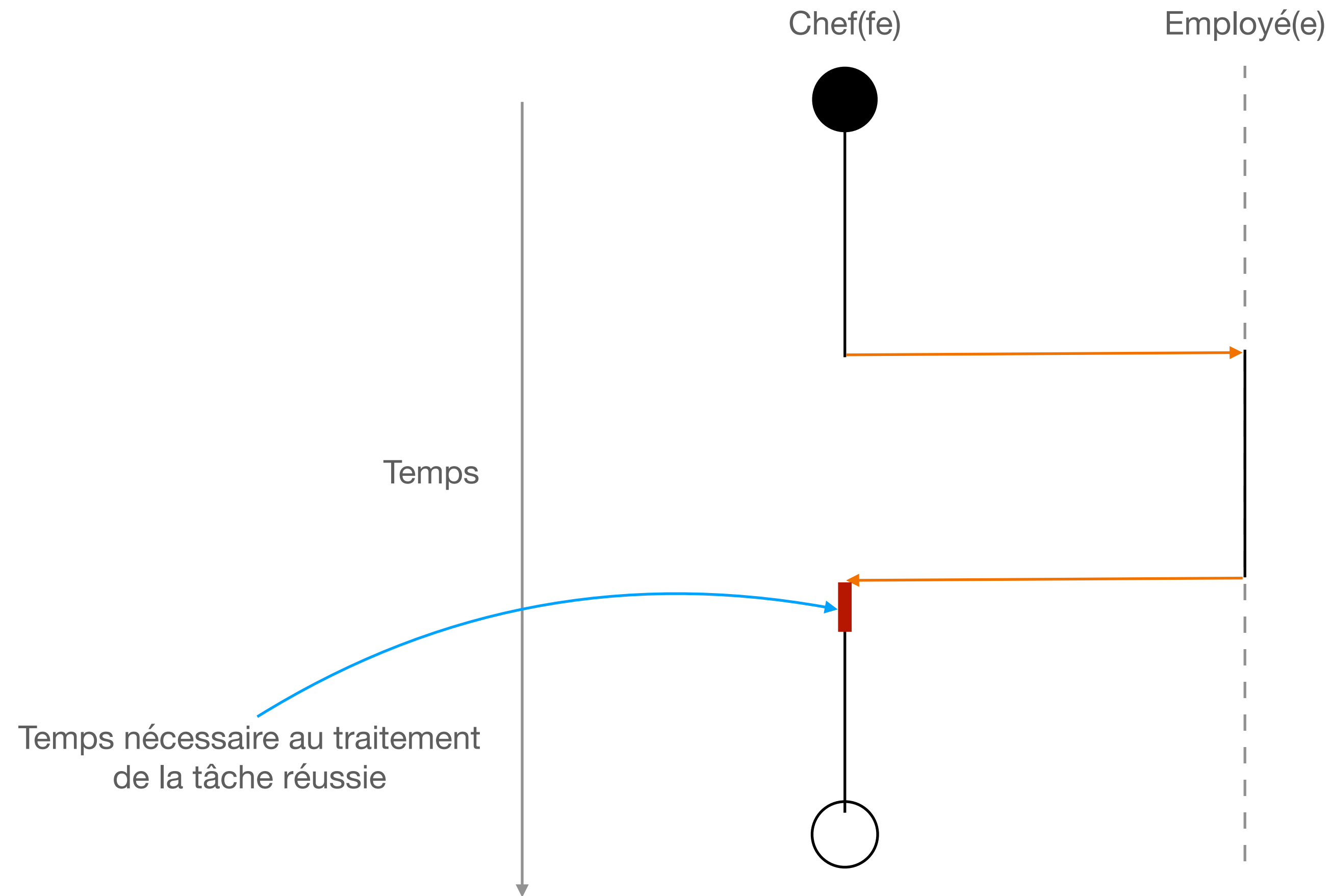
Promises - Exemple

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 boîte-aux-lettres, jusqu'à avoir reçu celui-ci. Il arrive quand il arrive.

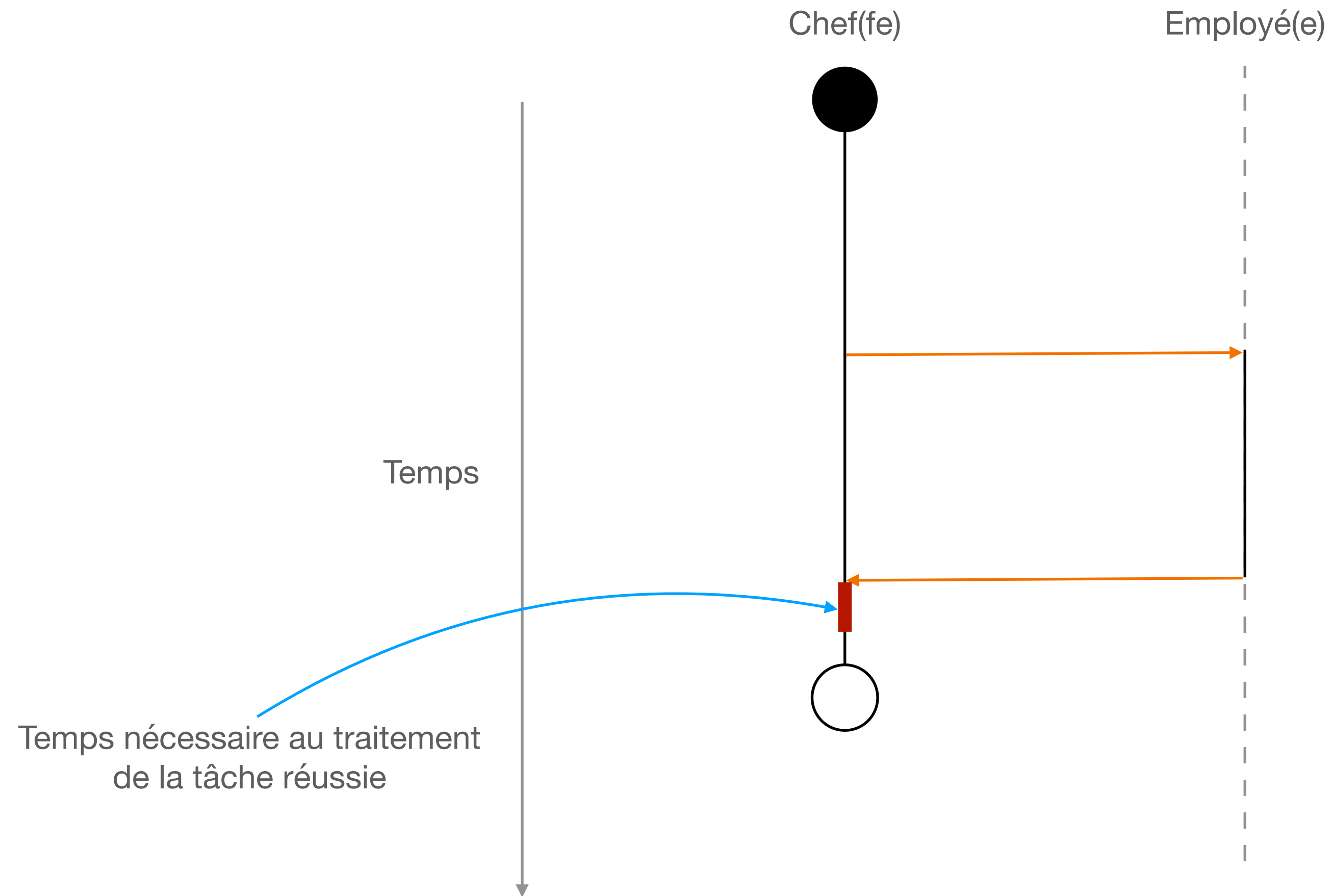
Promises - Code synchrone

API Fetch



Promises - Code asynchrone

API Fetch

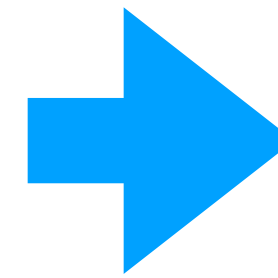


Promises - Synchrone

API Fetch

Prenons l'exemple de code suivant...

```
console.log('Hello 1')  
uneFonctionSynchroneClassiqueQuiAfficheHello2()  
console.log('Hello 3')
```



Console

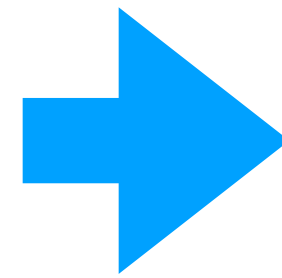
```
Hello 1  
Hello 2  
Hello 3
```

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

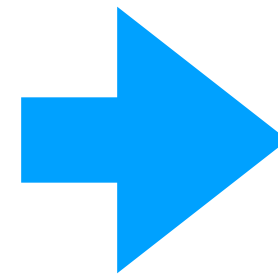
- Il existe trois types de méthodes utilisables sur une Promise et prennent toute une fonction en paramètre:
 - then - “Ensuite” - La fonction passée sera appelé lorsque tout se passe bien
 - catch - La fonction passée sera appelé lorsqu’il y a une erreur
 - finally - La fonction passée sera appelé dans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

Promises - then/catch/finally

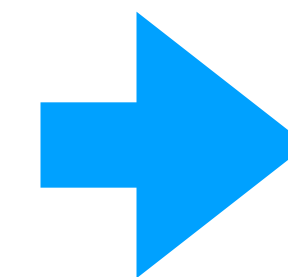
API Fetch

```
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

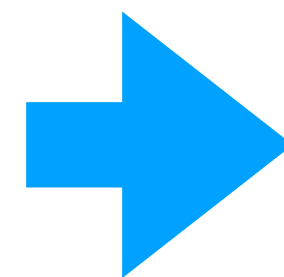
```
Hello 1  
Hello 2  
Hello 3
```

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'))
```



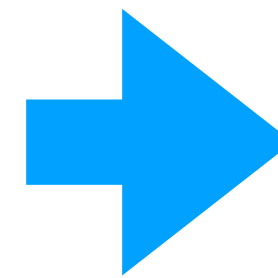
```
AfficheHello1()  
  
AfficheHello2()  
  .ensuite(() => afficheHello3())
```

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

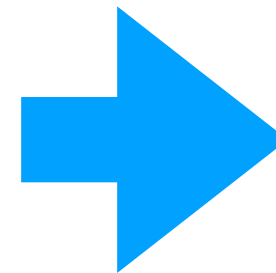
API Fetch

- 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 chargerArtistes() {

  return fetch('http.../api/artists')

    .then((response) => response.json())

}
```

```
// 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

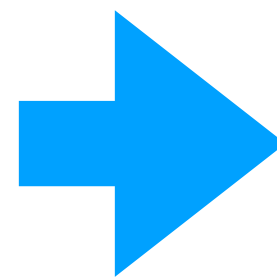
Promises - Rendre synchrone

API Fetch

```
// fichier.js
```

```
const resultat = await fetch('...')
```

Erreur ! Pas possible au root d'un fichier



```
// fichier.js
```

```
async function chargerSynchrone() {  
    const resultat = await fetch('...')  
}
```

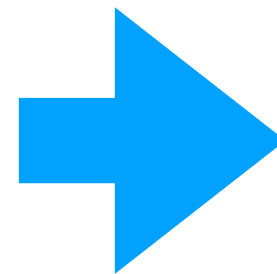
```
chargerSynchrone()
```

Possible, car englobé dans une fonction async

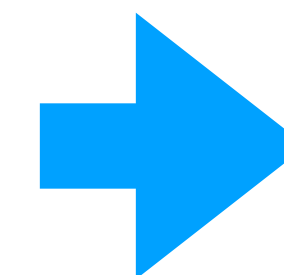
Promises - Rendre synchrone

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()
```



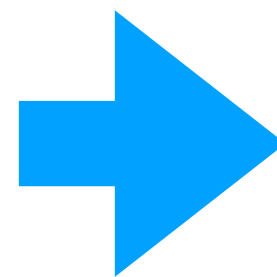
Console

```
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'))
```



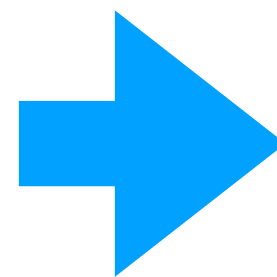
```
async function afficherHellos() {
  try {
    console.log('Hello 1')
    await unePromiseAsynchroneQuiAfficheHello2()
    console.log('Hello 3')
    console.log('Hello 4')
  }
  catch (e) {
    console.log('Erreur !', e)
  }
  finally{
    console.log('Terminé')
  }
}

afficherHellos()
```


Promises - Rendre synchrone

API Fetch

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

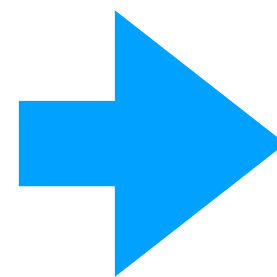


```
async function afficherArtiste() {  
  const response = await fetch('http.../api/artists')  
  const artistes = response.json()  
  const artist = artistes[0]  
  console.log(artist)  
}  
afficherArtiste()
```

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

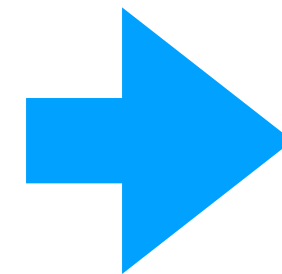
API Fetch

- 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 `` dans la liste des artistes
- Il suffit ensuite de dupliquer cet élément vide autant de fois que nécessaire pour afficher la liste complète

Deux écoles

Templating

- 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, ...

Templating manuel

Templating

- Nous allons utiliser le templating manuel pour la première partie du cours (Spotlified)
- Moteur de template en partie 2, avec les frameworks javascript

Templating manuel

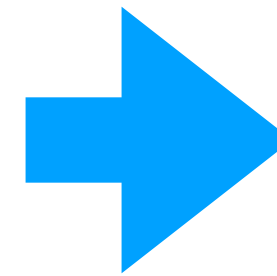
Templating

- 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...

Example

Templating

```
<div class="artist-list">
  <a href="#">
    
    <div class="artist-list-item-title">Ava Max</div>
  </a>
  <a href="#">
    
    <div class="artist-list-item-title">Ed Sheeran</div>
  </a>
  ...
</div>
```



```
<div class="artist-list">
</div>
<template id="artist-list-item-template">
  <a href="#">
    <img src="" />
    <div class="artist-list-item-title"></div>
  </a>
</template>
```

Exemple

Templating

```
const artistList = document.querySelector('.artist-list')

const artistListItemTemplate = document.querySelector('#artist-list-item-template')

function afficherArtiste(artiste) {

  const newArtist = artistListItemTemplate.content.cloneNode(true) // true pour cloner également les enfants du node

  newArtist.querySelector('a').href = '#artists-' + artiste.id

  newArtist.querySelector('img').src = artiste.image_url

  newArtist.querySelector('.artist-list-item-title').innerText = artiste.name

  artistList.append(newArtist)
}

function afficherArtistes(artistes) {

  artistList.replaceChildren() // Remplace les enfants par rien, donc supprime tous les enfants

  for(const artiste of artistes){

    afficherArtiste(artiste)

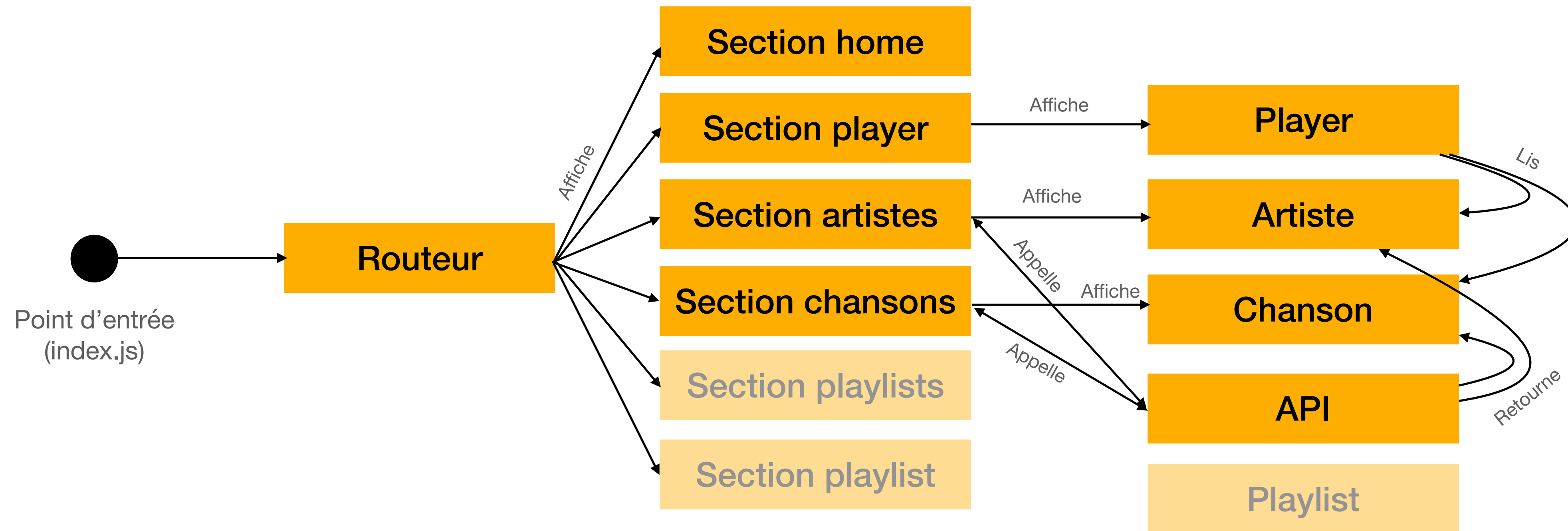
  }

}
```

Put it together

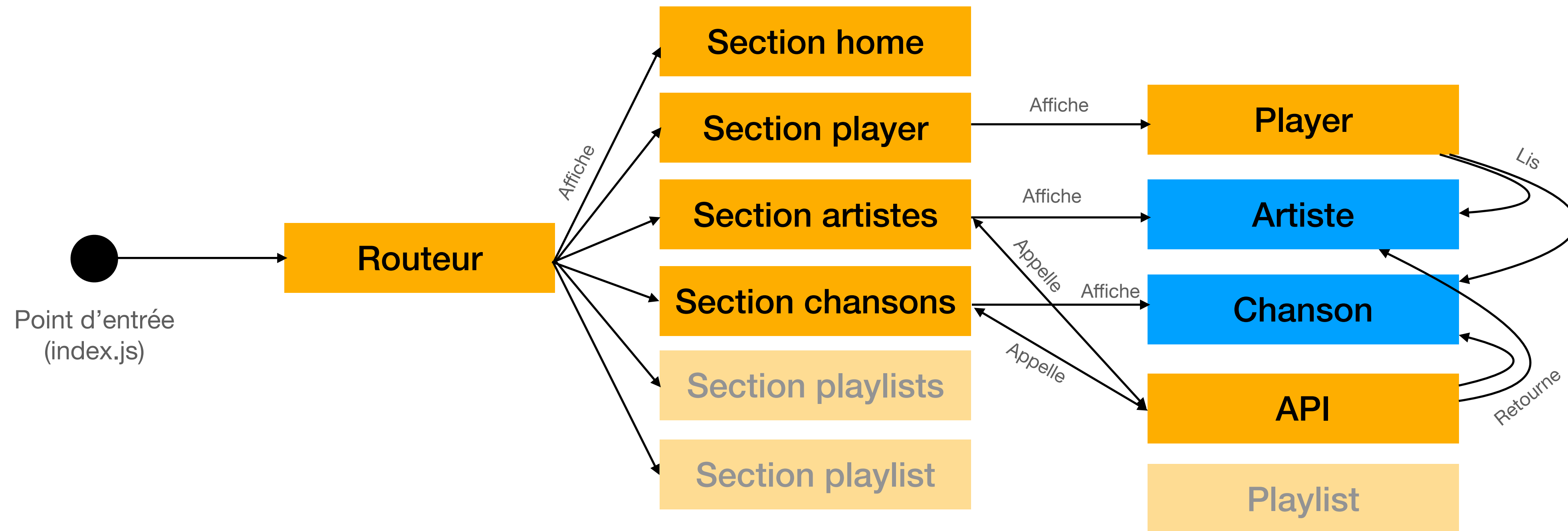
Vue globale

Put it together



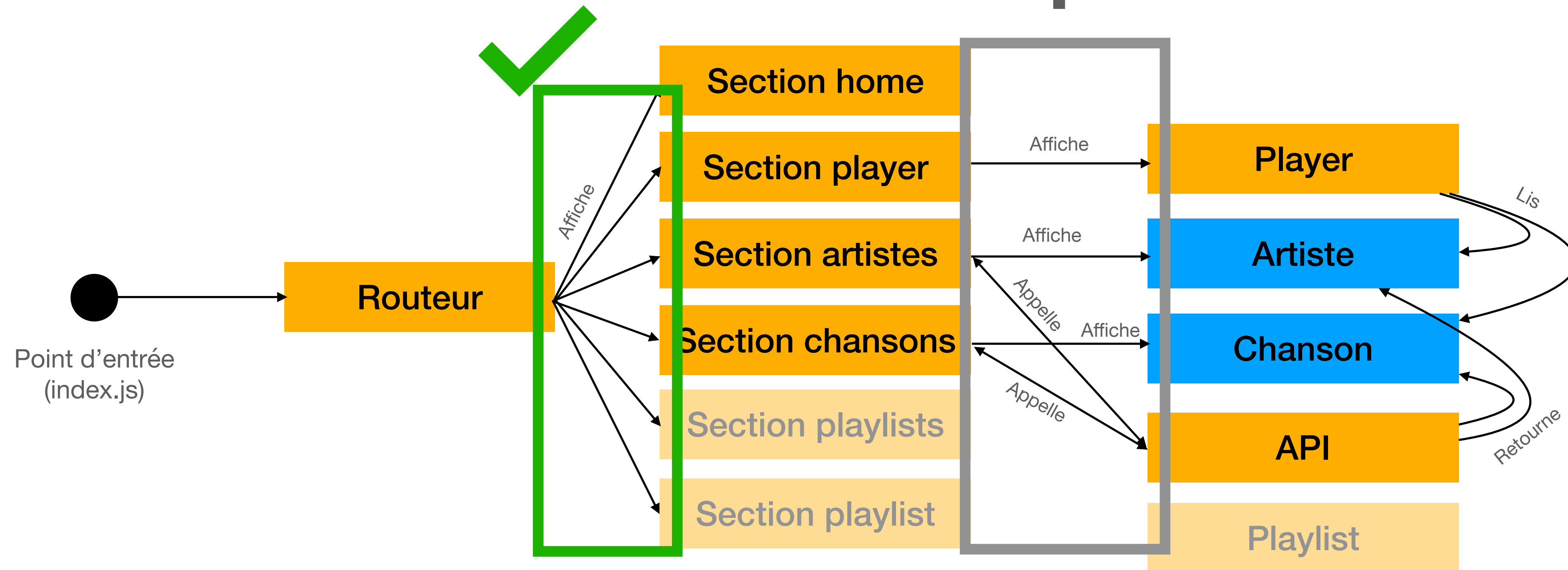
Vue globale

Put it together



Vue globale

Put it together



Routeur

Put it together

```
// Affichage d'une section
function displaySection() {
  // Comme nos hash et nos ids de section sont les mêmes, hash = sectionid.
  // S'il n'y a pas de hash (par ex, on est sur "localhost:8080/"), le défaut devient '#home-section'
  const sectionId = window.location.hash || '#home-section'

  // Supprime/Ajoute la classe active sur la section
  document.querySelector('section.active')?.classList.remove('active')
  document.querySelector(sectionId)?.classList.add('active')

  // Supprime/Ajoute la classe active sur le lien
  document.querySelector('nav a.active')?.classList.remove('active')
  document.querySelector('nav a[href="' + sectionId + '"]')?.classList.add('active')

  // ... ???
  if(sectionId == '#artistes-section')
    loadArtistesSection() // Par exemple ???
}
```


Routeur - Liens enfants (vue artiste)

Put it together

- Comment gérer les liens enfants, type “#artists-12” ?
- Il nous faut différencier :
 - La liste des artistes “#artists”
 - La vue d’un artiste, selon son id “#artists-12”

Routeur - Liens enfants (vue artiste)

Put it together

“#artists-12”



Est-ce qu'il y a un tiret ?

Routeur - Liens enfants (vue artiste)

Put it together

- Pléthore de manières de faire... Les deux principales seraient:
 - Chercher le tiret dans la chaîne de caractère et la découper avec des expressions régulières ou des fonctions de recherche...
 - Se dire que la chaîne “#artists-12” n’est en fait qu’une liste d’éléments séparés par des tirets... et les convertir en tableau

Routeur - Liens enfants (vue artiste)

Put it together

- La v2 semble plus simple...
- La fonction “split” vient à la rescousse !
- Elle permet de découper une chaîne en un tableau de sous-chaînes de caractères, selon un caractère donné
- Exemple:
`'une-chaîne-de-tirets'.split('-') ==> ['une', 'chaîne', 'de', 'tirets']`

Routeur - Fonction split

Put it together

- Exemple précédent
`'une-chaine-de-tirets'.split('-') ==> ['une', 'chaine', 'de', 'tirets']`
- Que se passe-t-il si pas de tirets ?
`'uneChaineSansTirets'.split('-') ==> ['uneChaineSansTirets']`
- Que se passe-t-il avec une chaine vide ?
`''.split('-') ==> ['']`

Routeur - Fonction split

Put it together

- Avec nos artistes :

- Avec tirets

```
const hashSplité = '#artists-12'.split('-') ==> ['#artists', '12']
```

- Que se passe-t-il si pas de tirets ?

```
'#artists'.split('-') ==> ['#artists']
```

Routeur - Fonction split

Put it together

- Exemple d'implémentation :

```
const hashSplité = window.location.hash.split( '-' )
```

```
// avec '#artists-12' ==> [ '#artists', '12' ]
```

```
// avec '#artists' ==> [ '#artists' ]
```

Routeur - Fonction split

Put it together

- Que se passe-t-il quand on essaie d'accéder à un élément d'un tableau qui n'existe pas ?

Exemple: La cellule 397 du tableau ['a', 'b', 'c'] ?

```
const tableau = [ 'a', 'b', 'c' ]
```

```
console.log(tableau[397]) // ==> undefined
```


Routeur - Fonction split

Put it together

Par exemple...

```
const hashSplité = window.location.hash.split('-')

// si le premier élément est artiste, on est dans la gestion des artistes...

if(hashSplité[0] == '#artists') {

    // est-ce que le deuxième élément retourne quelque chose ? Et donc n'est pas undefined ? Oui?
    // Alors il y a un id et on affiche cet artiste

    if(hashSplité[1]) {

        afficherChansonsArtiste(hashSplité[1]) // la fonction prend en argument l'id de l'artiste à afficher

    }

    else {

        afficherArtistes()

    }

}
```

Routeur - Fonction split

Put it together

Par exemple... avec un switch

```
const hashSplité = window.location.hash.split('-')

// si le premier élément est artiste, on est dans la gestion des artistes...

switch(hashSplité[0]) {

  case '#artists':

    // est-ce que le deuxième élément retourne quelque chose ? Et donc n'est pas undefined ? Oui?
    // Alors il y a un id et on affiche cet artiste

    if(hashSplité[1]) {

      afficherChansonsArtiste(hashSplité[1])

    }

    else {

      afficherArtistes()

    }

    break;

  case '#player':

    ...

    break;

}
```

GO !

Player

Concept

Player

- Le player doit être capable de lire une chanson
- Avancer dans la chanson
- Mettre à jour le bouton play/pause, selon son état
- Afficher son titre + les détails de l'artiste
- Utiliser les boutons précédent/suivant, selon le contexte dans lequel la chanson en cours a été lue

Tag audio

Player

- Le tag audio prend sa source grâce à l'attribut “src”
- Il dispose de plusieurs méthodes de contrôles
- Il émet plusieurs événements pour gérer son état

Tag audio - Play/pause Player

```
const player = document.querySelector( '#audio-player' )
```

```
player.src = 'http:...hello.mp3'
```

```
player.paused // retourne true ou false
```

```
player.play( )
```

```
player.pause( )
```

```
// avancer dans la chanson, quand on déplace le slider, par ex  
player.currentTime = 1234
```

Tag audio - Play/pause Player

```
const player = document.querySelector( '#audio-player' )

...

function togglePlayPause() {

    if(player.paused)

        player.play( )

    else

        player.pause( )

}

document.querySelector( '#player-control-play' ).addEventListener( 'click', togglePlayPause)
```


Tag audio - Avancer Player

```
const player = document.querySelector( '#audio-player' )  
  
...  
  
function avancerPlayer(event) {  
    player.currentTime = event.currentTarget.value  
}  
  
document.querySelector( '#player-progress-bar' ).addEventListener( 'change', avancerPlayer)
```

Tag audio - Événements

Player

```
const player = document.querySelector( '#audio-player' )

// Appelé quand la valeur de player.paused à changé (true/false)
player.addEventListener( 'play', changerIcône)

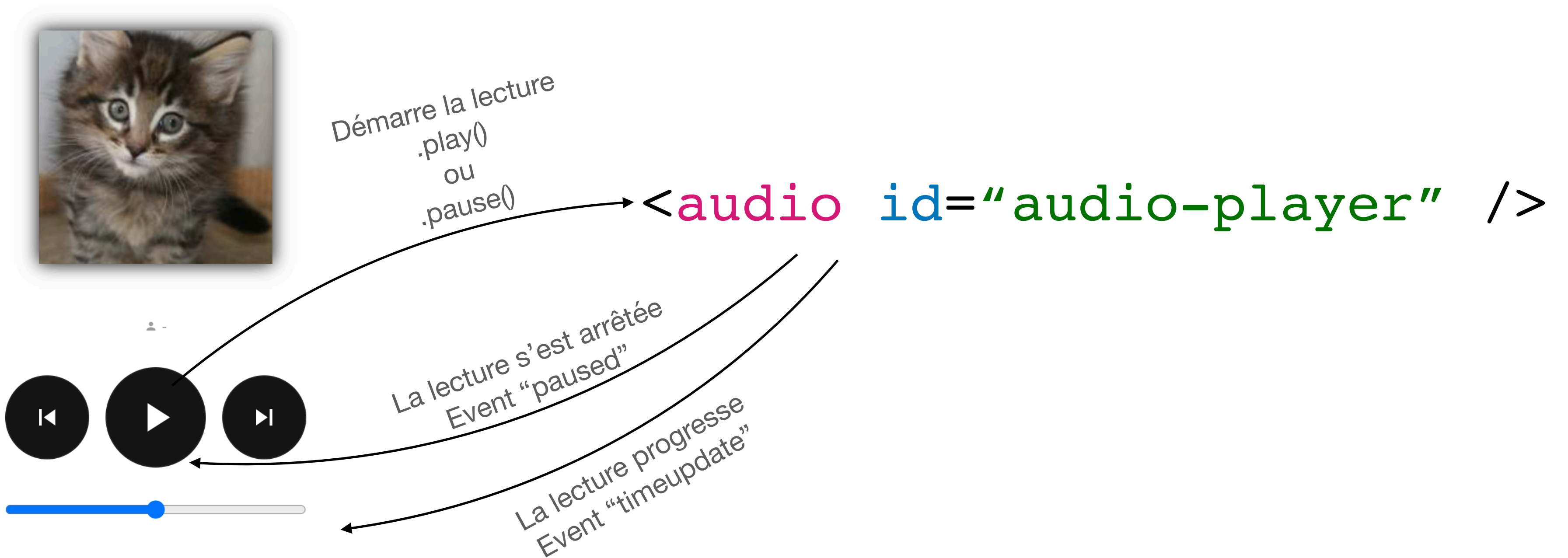
// Appelé quand la chanson est à la fin
player.addEventListener( 'ended', chansonSuivante)

// Appelé quand la valeur de player.duration a changé (ex: une nouvelle chanson
// est chargée, on met à jour la durée)
player.addEventListener( 'durationchange', mettreAJourValeurMaxSlider)

// Appelé lorsque la chanson progresse (mettre à jour le slider)
player.addEventListener( 'timeupdate', mettreAJourValeurSlider)
```

Tag audio

Player



Tag audio - Next/prev Player

- Le tag audio ne gère pas la notion de précédent/suivant
- Ce sera à vous de garder une copie du tableau de chansons dans lequel se trouve la chanson en cours et passer à la précédente/suivante, selon sa position dans le tableau

Fonctions du player

Player

- Nous allons évidemment cacher toutes ces méthodes dans un seul fichier et offrir une interface au reste de l'application pour discuter avec le player
- Cela évite de copier/coller des `querySelector('#audio-player')` partout dans le code
- Cela permet aussi de gérer la liste de lecture en cours pour les boutons précédent/suivant

Fonctions du player

Player

- Une section autre que celle du player doit pouvoir appeler une fonction de lecture du style:

`lireChanson(laChanson, leTableauDeChansons)`

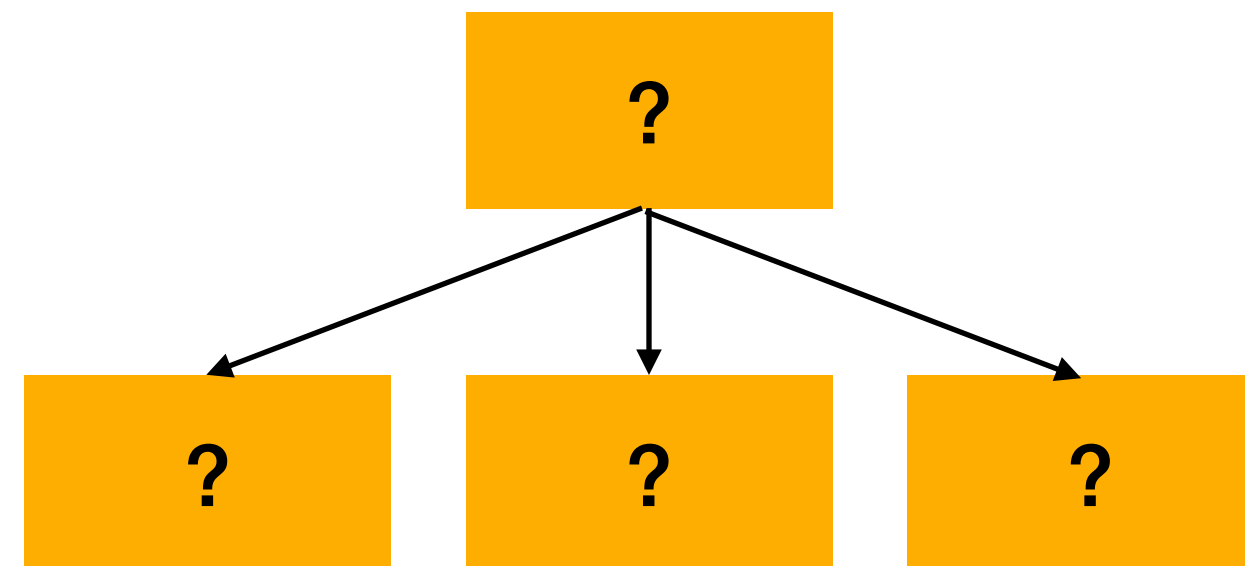
- Le reste doit être encapsulé dans un fichier player.js, par exemple
- Au chargement de l'application, tous les événements doivent être liés au tag audio pour mettre à jour l'interface

Structure

Player



Comment structurer le code ?



GO !