Rétrospective

Composants

Rétrospective

- ✓ Projet parcel vide
- ✓ Squelette HTML
- ✓ Styles CSS structurels
- ✓ Icônes
- ✓ Routeur pour les pages web
- ✓ Client pour l'API JSON
- ✓ Lecteur audio
- ✓ Local storage pour les favoris
- ✓ Détection online/offline
- ✓ Manifest PWA
- ✓ Caching
- ✓ Service worker

Projet Parcel

Rétrospective

- Il est important de se rappeler que Parcel est un “packager” qui sert à grouper plusieurs fichiers ensemble

- Il permet également de résoudre des dépendances extérieurs (par exemple, installer un package pour l’utiliser dans l’application)

- Parcel fournit un serveur web de développement pour faciliter le développement

Projet Parcel

Rétrospective

- Deux commandes principales sont utilisées pour gérer le projet

- `npm install` - Installe toutes les dépendances du projet. S'utilise typiquement à l'installation du projet ou lorsqu'une dépendance est mise à jour

- `npm run start` - Démarre le serveur web de notre projet

Markup HTML et styles CSS

Rétrospective

- Il est important d'utiliser des tags sémantiques variés pour bien différencier les éléments de l'application (privilégier <main>, <section>, <article> à)

- Pour le CSS, privilégier des classes de contrôles à appliquer à un élément, plutôt que de modifier son CSS à la main en javascript
  Par ex: la classe "active" que nous utilisons pour afficher les sections

- A noter que ces classes "active" sont manuellement définie dans le CSS que nous utilisons ! Il ne s'agit pas d'une fonction du navigateur !

Markup HTML et styles CSS

Rétrospective

- Exemple de classe de contrôle:

```css
section {
  display: none;
}

section.active {
  display: flex;
}
```

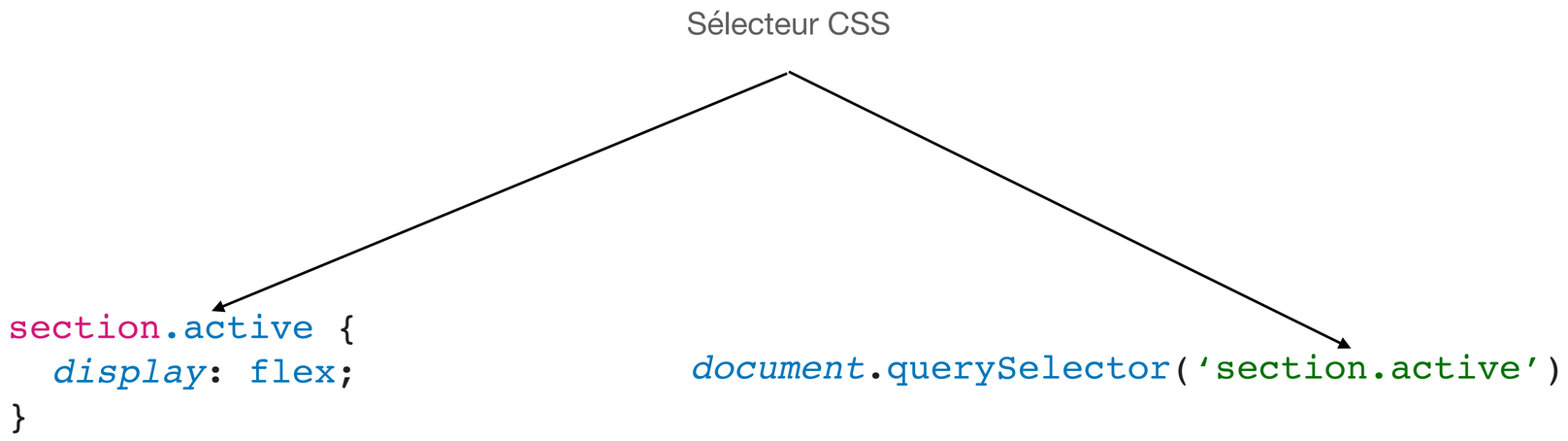
```html
<section id="ma-section">
  …
</section>
```

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

Markup HTML et styles CSS

Rétrospective

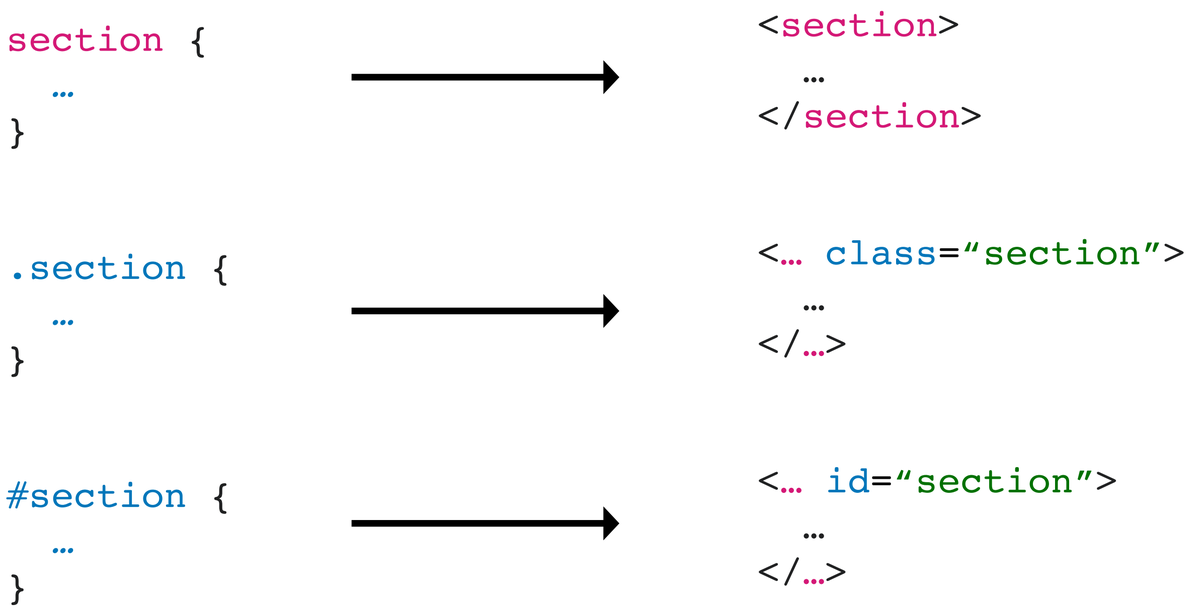
- Pour rappel, la chaine de caractères passée à querySelector est une chaine de sélecteur CSS



Markup HTML et styles CSS

Rétrospective

- Rappel sur l'utilisation des sélecteurs CSS…



Markup HTML et styles CSS

Rétrospective

Un espace entre deux sélecteurs veut dire “enfant de”

```
section.active {
  …
}
```

```
<section class=“active”>
  …
</section>
```

```
section  .active {
  …
}
```

```
<section>
  <… class=“active”>
    …
  </…>
</section>
```

```
section  div.active {
  …
}
```

```
<section>
  <div class=“active”>
    …
  </div>
</section>
```

Markup HTML et styles CSS

Rétrospective

```
section.active#section-songs {
  …
}
```

```
<section class="active" id="section-songs">
  …
</section>
```

```
section  .active#section-songs {
  …
}
```

```
<section>
  <… class="active" id="section-songs">
    …
  </…>
</section>
```

```
section  .active  #section-songs {
  …
}
```

```
<section>
  <div class="active">
    <div id="section-songs">
      …
    </div>
  </div>
</section>
```

Markup HTML et styles CSS

Rétrospective

Il est possible de cumuler les sélecteurs pour le même élément

```
section.active.en-bleu {
  …
}
```

```
<section class="active en-bleu">
  …
</section>
```

```
section.active  .en-bleu {
  …
}
```

```
<section class="active">
  <… class="en-bleu">
    …
  </…>
</section>
```

```
section.active  div.en-bleu {
  …
}
```

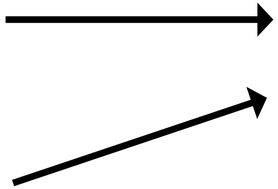
```
<section class="active">
  <div class="en-bleu">
    …
  </div>
</section>
```

Markup HTML et styles CSS

Rétrospective

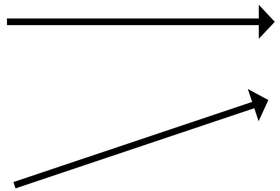
L’ordre des ids et classes n’a pas d’importance…

```
section.active.en-bleu {
  …
}

section.en-bleu.active {
  …
}
```



```
<section class=“active en-bleu”>
  …
</section>
```

```
section#section-songs.active {
  …
}

section.active#section-songs {
  …
}
```



```
<section id=“section-songs” class=“active”>
  …
</section>
```

Markup HTML et styles CSS – Icônes

Rétrospective

- Pour rappel, nous utilisons Google Material icons
- Intégré en mode CDN, via `<link rel="stylesheet" href="https://fonts.googleapis.com/css2?family=Material+Icons" />`
- https://fonts.google.com/icons
- Exemple : `<span class="material-icons">face</span>`

Javascript – Fonctions

Rétrospective

- Rappel sur les fonctions…

```
// Déclaration d'une fonction classique (v1)
function maFonction() {
  …
}

// Déclaration d'une fonction fléchée (v2)
const maFonction = () => {
  …
}

// Appel d'une fonction
maFonction()

// Référence à la fonction (mais pas appelée)
// Une référence a une fonction s'utilise typiquement pour être passée en paramètre
maFonction
```

Javascript - Fonctions

Rétrospective

```
// Déclaration d'une fonction
const maFonction = () => {
  …
}

// Ici, on passe la référence de la fonction au listener, comme une manière de lui dire
// "quand l'événement se passe, il faut appeler la fonction "maFonction"
window.addEventListener('hashchange', maFonction)
```

Javascript - Fonctions

Rétrospective

```
// HERE BE DRAGONS
// Ici, on ne passe pas la référence à la fonction, mais on l'appelle et c'est le résultat de
// maFonction() qui sera passé comme argument au listener…
window.addEventListener('hashchange', maFonction())


// Le code ci-dessus est sémantiquement équivalent à cela:
const temp = maFonction()
window.addEventListener('hashchange', temp)
```

Javascript – Fonctions fléchées

Rétrospective

- Pour simplifier, une fonction fléchée est une fonction anonyme. En gros, une fonction qui n'a pas de nom…

- Elles sont typiquement utilisées pour être passées en argument quelque part. Par exemple, lorsque l'on a pas besoin de la réutiliser, mais qu'on est obligé de passer une fonction

- Création "on the fly" ou utilisant comme valeur d'une constante/variable

Javascript - Fonctions fléchées

Rétrospective

```
// Dans ce cas, on peut utiliser une fonction fléchée. On n'aura certainement jamais besoin de rappeler
// cette fonction, mais nous n'avons pas le choix d'en passer une, car addEventListner nous l'impose

window.addEventListener('hashchange', () => {
  console.log('hello')
})
```

Javascript – Fonctions fléchées

Rétrospective

```
// HERE BE DRAGONS
// Cf. Exemple précédent sur les appels, ceci ne marche pas ! On se retrouve à nouveau dans le cas
// précédent où console.log sera directement appelé et c’est son résultat qui sera utilisé comme
// paramètre. De l’intérêt de le mettre dans une fonction fléchée

window.addEventListener('hashchange', console.log(‘hello'))
```

Import / Export

Fichiers JS

Version ‘default’ (1 export principal)

```
// player.js


const lireChanson = () => {
  // ...
}
export default lireChanson


// index.js
import lireChanson from './player.js'
```

Version plusieurs exports

```
// player.js
const lireChanson = () => {
  // ...
}
const mettreEnPause = () => {
  // ...
}
export { lireChanson, mettreEnPause }


// index.js
import { lireChanson, mettreEnPause } from './player.js'
```

Import / Export

Fichiers JS

Version mixte

```
// player.js
const lireChanson = () => {
  // ...
}
const mettreEnPause = () => {
  // ...
}
export default lireChanson
export { mettreEnPause }

// index.js
import lireChanson, { mettreEnPause } from './player.js'
```

Version mixte (import default only)

```
// player.js
const lireChanson = () => {
  // ...
}
const mettreEnPause = () => {
  // ...
}
export default lireChanson
export { mettreEnPause }

// index.js
import lireChanson from './player.js'
```

Routeur pour les pages web (changement de section)

Rétrospective

- Le premier problème que nous avons rencontré est celui de la gestion des différentes sections - Comment les afficher au bon moment?

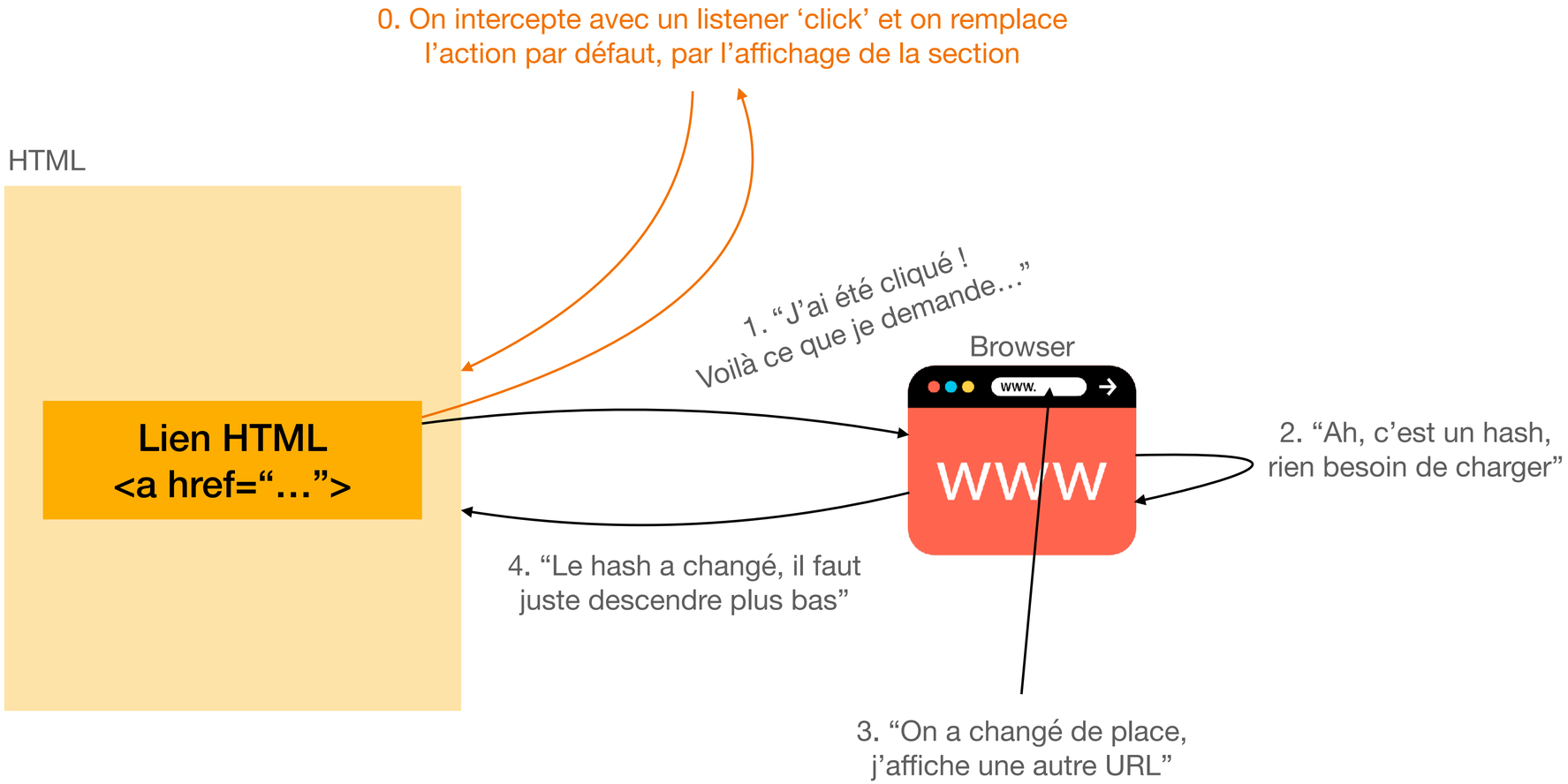
- Deux grandes écoles :

  - Mettre des “eventListener” sur des boutons et cacher/afficher (“v1”)

  - Détecter les changements d’URL et réagir en fonction (“v2”)

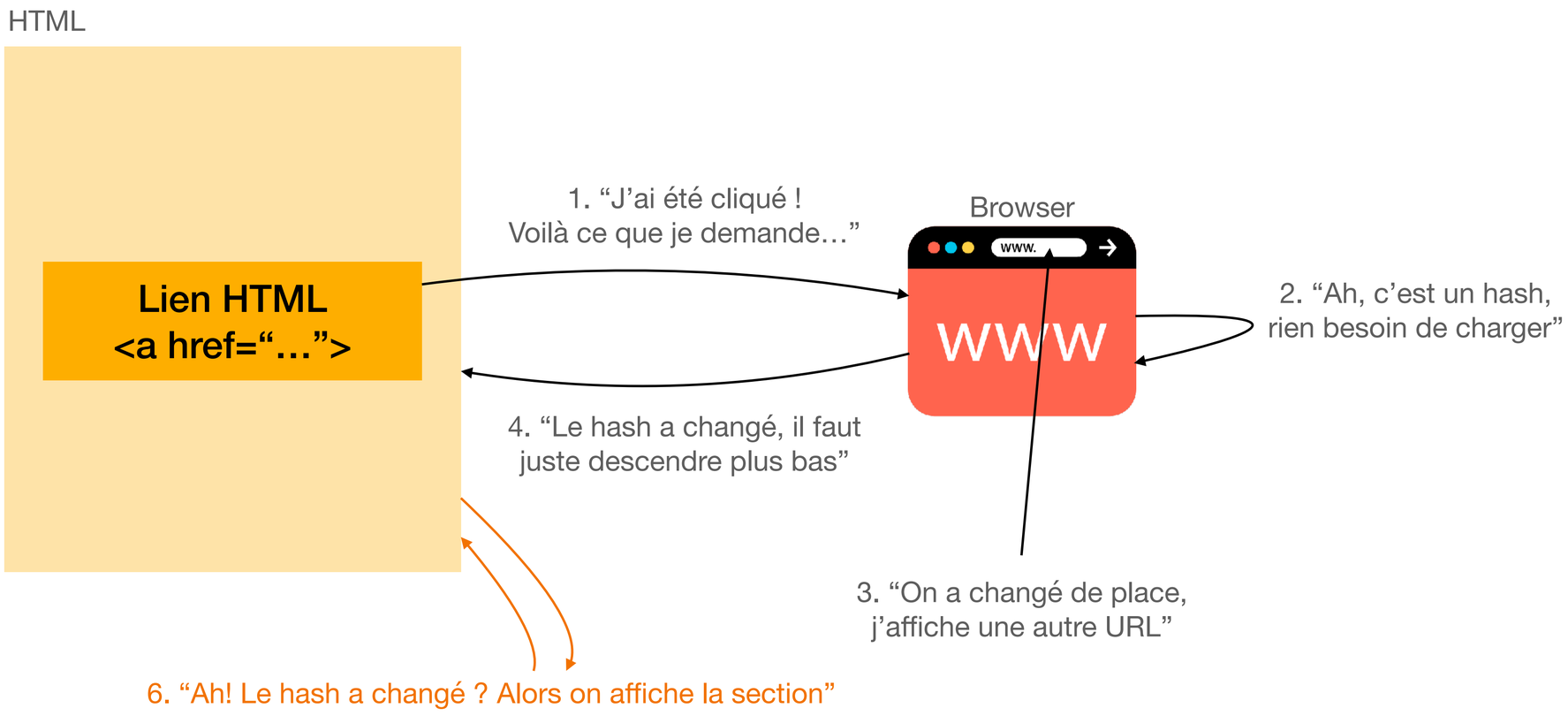
Routeur pour les pages web (changement de section) V1

Rétrospective



Routeur pour les pages web (changement de section) V2

Rétrospective



Routeur pour les pages web (changement de section)

Rétrospective

- Nous nous sommes concentrés sur la v2 - pourquoi ?
  - Le code est moins invasif. Avec un seul eventListener, on peut gérer tous les cas, plutôt que de devoir linker indépendamment chaque élément du menu
  - Cela permet d'obtenir facilement des URLs différentes par section -> Parfait pour les PWA, c'est SEO compliant
  - On laisse la fonctionnalité des liens par défaut… Un clique droite "Ouvrir dans un nouvel onglet" fonctionne. Ce n'est pas le cas avec un click listener.

Routeur pour les pages web (changement de section)

Rétrospective

- Comment récupérer l'info ?

- A chaque changement d'URL, le browser émet un événement. Soit "`popstate`" pour tout changement d'URL, soit "`hashchange`" pour un changement dans le hash uniquement

- Comme nous travaillons uniquement avec des hash pour des raisons pratique, "`hashchange`" fait l'affaire

Routeur pour les pages web (changement de section)

Rétrospective

```
const routeur = () => {
  …
}

window.addEventListener('hashchange', routeur)
```

Ou aussi….

```
window.addEventListener('hashchange', () => { … })
```

Routeur pour les pages web (changement de section)

Rétrospective

- NB: hashchange n'est dispatché que quand l'URL change. Arriver sur une page web avec une URL n'est pas un changement. C'est l'état de base.

- L'idée est donc d'appeler une fois cette fonction au chargement pour que logique s'applique correctement à l'url en cours

Routeur pour les pages web (changement de section)

Rétrospective

```
const routeur = () => {
  …
}


// On link la fonction "routeur" à l'événement hashchange pour être
averti d'un changement de hash dans l'url
window.addEventListener('hashchange', routeur)


// Affichage au chargement pour traiter l'url en cours (exemple: on
ouvre un lien dans un nouvel onglet)
displaySection()
```

Routeur pour les pages web (changement de section)

Rétrospective

- Nous somme maintenant informé d'un changement de hash, c'est à dire lorsque l'utilisateur va cliquer sur un lien du type:

```
<a href="#home">
  …
</a>
```

Routeur pour les pages web (changement de section)

Rétrospective

- Reste à trouver la section en cours d'affichage, s'il y en a une, et la cacher

- Il faut ensuite trouver la section à afficher et l'afficher

- Pour des raisons pratiques, nous avons essayer d'utiliser des id de section qui correspondent aux hash.

```
<a href="#home">
  …
</a>
```

```
<section id="home-section">
  …
</section>
```

Routeur pour les pages web (changement de section)

Rétrospective

- On trouve la section avec la classe “active” et on lui enlève cette classe, pour la cacher

- On trouve la section qui correspond à l’url et on y ajoute la classe active pour l’afficher

Routeur pour les pages web (changement de section)

Rétrospective

- Il convient de faire pareil pour les liens du menu (le classe active la colorie en bleu)

- On gère cela séparément, car dans certains cas, un même élément du menu est utilisé avec deux sections (typiquement “Musique” est actif pour la liste d’artistes ET la liste des chansons)

Routeur pour les pages web (changement de section)

Rétrospective

- Exemple simplifié

```
const displaySection = (id) => {
  document.querySelector('section.active')?.classList.remove('active')
  document.querySelector(`${id}-section`)?.classList.add('active')
}
```

Le “?” veut dire “si ce qu’il y a avant retourne quelque chose, on fait ce qu’il y a après”

Routeur pour les pages web (changement de section)

Rétrospective

- Cela fonctionne bien pour afficher/masquer les sections, mais que faire quand il y a une logique particulière sur une section ? Par exemple, charger des informations

- On peut rajouter une condition, selon la valeur de l'URL

Routeur pour les pages web (changement de section)

Rétrospective

- Exemple simplifié

```
const routeur = () => {
  const hash = window.location.hash

  displaySection(hash)

  if(section == '#artists') {
    faireQuelqueChose()
  }
}
```

Routeur pour les pages web (changement de section)

Rétrospective

- Exemple simplifié, s'il y en a plusieurs, un switch est plus propre…

```
const routeur = () => {
  const hash = window.location.hash

  displaySection(hash)

  switch(hash) {
    case '#artists':
      faireQuelqueChose()
    break;
    case '#player':
      faireAutreChose()
    break;
  }
}
```

Routeur pour les pages web (changement de section)

Rétrospective

- Cela fonctionne bien, mais que faire quand nous avons besoin d'une info présente dans l'URL ? Par exemple l'id d'un artiste ?

- On structure généralement les URLs en mode REST:

  artists —> Liste d'artistes

  artists/12 —> Infos de l'artiste numéro 12

Routeur pour les pages web (changement de section)

Rétrospective

- Comme nous utilisons des hashs, nous pouvons remplacer les / par des tirets, par exemple:

- #artists —> Liste d'artistes

  #artists-12 —> Infos de l'artiste numéro 12

Routeur pour les pages web (changement de section)

Rétrospective

- On peut alors découper l'url en utilisant la fonction split()

- Cela nous donne un tableau qui est structurellement équivalent à notre URL

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

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

Routeur pour les pages web (changement de section)

Rétrospective

- Exemple simplifié

```
const routeur = () => {
  const hash = window.location.hash.split('-') || ‘#home'
  const hashs = hash.split('-')

  displaySection(hashs[0])

  switch(hashs[0]) {
    case ‘#artists’:
      if(hashs[1]) // S’il y a un deuxième élément, comme un id
        faireQuelqueChoseAvec(hashs[1])
      else
        faireQuelqueChose()
    break;
    case ‘#player’:
      faireAutreChose()
    break;
  }
}
```

Routeur pour les pages web (changement de section)

Rétrospective

- On sait maintenant gérer des URLs simples et des URLs complexes, avec des paramètres, par exemple

- Plus qu'à rendre cela interactif…!

Client pour l'API Json

Rétrospective

- Notre prochain problème: Charger des données distantes et les transformer en quelque chose d'utilisable en javascript

- Il faut pour cela se référer à l'API Spotlified (voir slides)

- C'est une API JSON. Le contenu doit donc être chargé, puis converti

Client pour l'API Json

Rétrospective

- Problem solved!

```
async function loadJson(url) {
  const response = await fetch(url)
  const parsedJson = await response.json()
  return parsedJson
}
```

Client pour l'API Json – variante promise

Rétrospective

- Problem solved!

```
const loadJson = (url) => fetch(url).then((response) => response.json())
```

Afficher les données

Rétrospective

- Prochain problème: Il nous faut maintenant un moyen de représenter les données chargées dans une liste

- Chaque liste a un markup différent. La liste des artistes n'a pas le même que la liste des chansons

Afficher les données

Rétrospective

- L'idée est de créer une coquille vide d'un élément et de lui passer simplement les informations à afficher via des chaines de caractères (des attributs!).

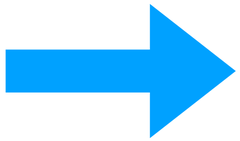
- Pour cela, nous utilisons un Custom HTML Element qui est spécialement prévu pour ce usecase et permet de ranger le HTML dans un genre de "black box"

- La section "Templating" des slides "Cours 2", pages 56-62 et 78-81 couvrent assez bien ce sujet

Afficher les données

Rétrospective

```
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>
```

Afficher les données

Rétrospective

- Pour aller plus loin (et respecter le concept de “chacun son job”), on peut également laisser le HTML dans le HMTL, plutôt que de le stocker dans le fichier javascript sous forme de chaine

- Il existe pour cela le tag <template> qui permet d’y insérer du HTML sans que celui-ci soit affiché

- On peut ensuite clone cet élément “squelette” depuis le javascript

Afficher les données

Rétrospective

```
const newContent = document.querySelector('#artist-list-item-template')
const newElement = newContent.content.cloneNode(true) // true pour cloner également les enfants du node
newElement.querySelector(‘img’).src = this.getAttribute('cover')
newElement.querySelector(‘div’).innerText = this.getAttribute('name')
```

Nous sommes ici dans la partie qui copie l’élément vide et le rempli. Notion très importante : Nous appelons ici querySelector non pas sur tout le document, mais uniquement sur la copie que l’on vient de créer.

Cela va viser ainsi les éléments qu’il contient uniquement et nous permettre de remplacer leur contenu

Afficher les données

Rétrospective

- Reste ensuite à appliquer ce que nous venons de voir pour chaque élément à afficher

- Typiquement, parcourir le tableau grâce à un forEach et créer un custom Element pour chaque objet du tableau

Afficher les données

Rétrospective

- Nous savons maintenant:
    - Charger des données
    - Les afficher
    - Appeler une logique particulière, basée sur une URL

Afficher les données

Rétrospective

- Prochain problème: interagir avec les données affichées

- Nous savons afficher des données et générer des liens HTML pour afficher d'autres données

- Qu'en est-il de l'interaction ? Par exemple, lire une chanson. Nous avons besoin de plus d'infos que cette simple chanson. Les passer par url semble donc compliqué

Interagir avec les données affichées

Rétrospective

- Chaque élément d'une liste, par exemple, doit pouvoir effectuer une action particulière

- Nous savons que lorsque nous remplissons les informations d'un élément, nous avons accès à tous ses tags HTML

- Nous pouvons nous en servir pour lier des event listener

Interagir avec les données affichées

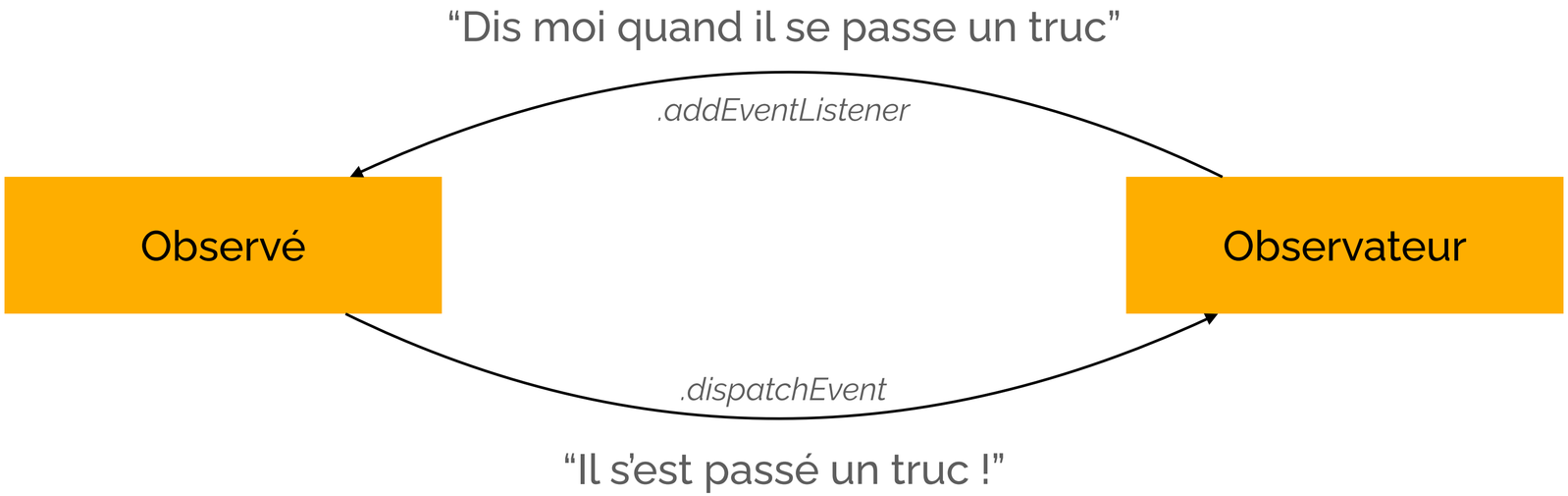
Rétrospective

- On aimerait maintenant être capable d'écouter des événements sur le custom HTML élément que nous avons créé

- Pour garder le concept de l'encapsulation (la black box!), il faudrait une manière d'exposer à l'élément parent (le custom élément) ce qui se passe dans ses enfants, sans que tout le monde soit au courant de la structure interne

Interagir avec les données affichées

Rétrospective



Interagir avec les données affichées

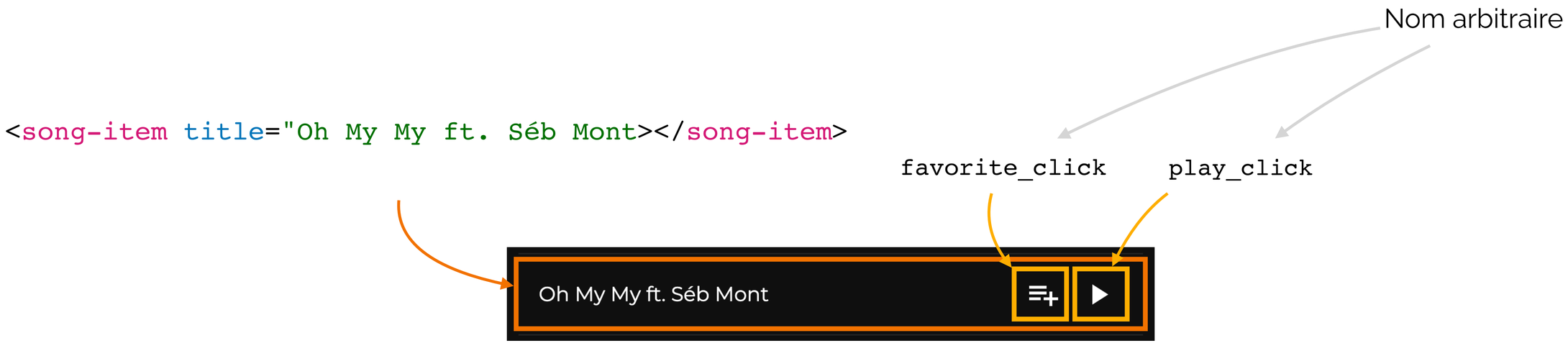
Rétrospective

- A l'image des custom éléments, il existe des custom events !

- Ils permettent de générer ses propres événements et les transmettre à qui est intéressé

Concept

Events, listeners & custom elements



Concept

Events, listeners & custom elements

```
// Déclaration d’un custom event
const playEvent = new CustomEvent(‘play_click')


class SongItem extends HTMLElement {
   connectedCallback() {
      this.innerHTML = …


      const playButton = this.querySelector(‘.play-button’)


      playButton.addEventListener(‘click’, (e) => {
        e.preventDefault()
        this.dispatchEvent(playEvent)
        // ou
        // this.dispatchEvent(new CustomEvent(‘play_click'))
      }
   }
}
customElements.define("song-item", SongItem)
```

Concept

Events, listeners & custom elements

Interagir avec les données affichées

Rétrospective

- On peut alors rajouter des events listener

```
const newSongItem = document.createElement('song-item')
// …
newSongItem.addEventListener('play_click', () => console.log('ça play'))
// …
songList.append(newSongItem)
```

- On sait maintenant interagir avec !

Utiliser le player

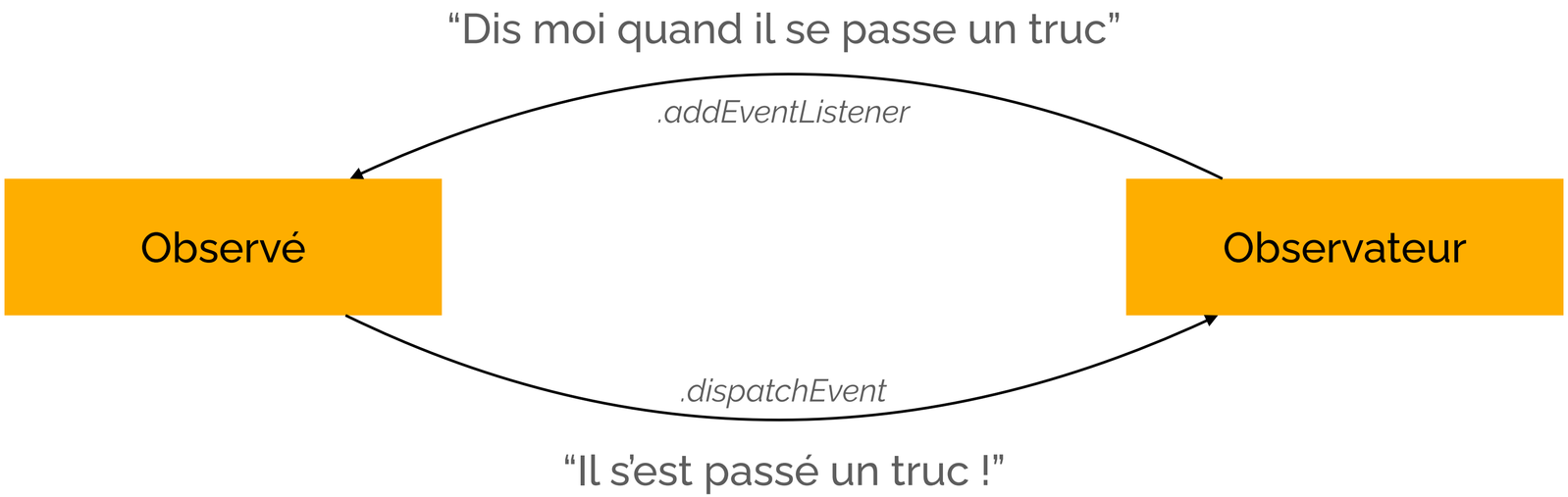
Rétrospective

- Prochain problème: Lire une chanson
- Il y a pour ça plusieurs slides sur le player. Cours 4, pages 1-14
- Cela va surtout consister en des ajouts d'event listeners sur des éléments…
- Il faudra également passer les informations de l'élément cliqué, grâce au listener vu au slide précédent

Utiliser le player

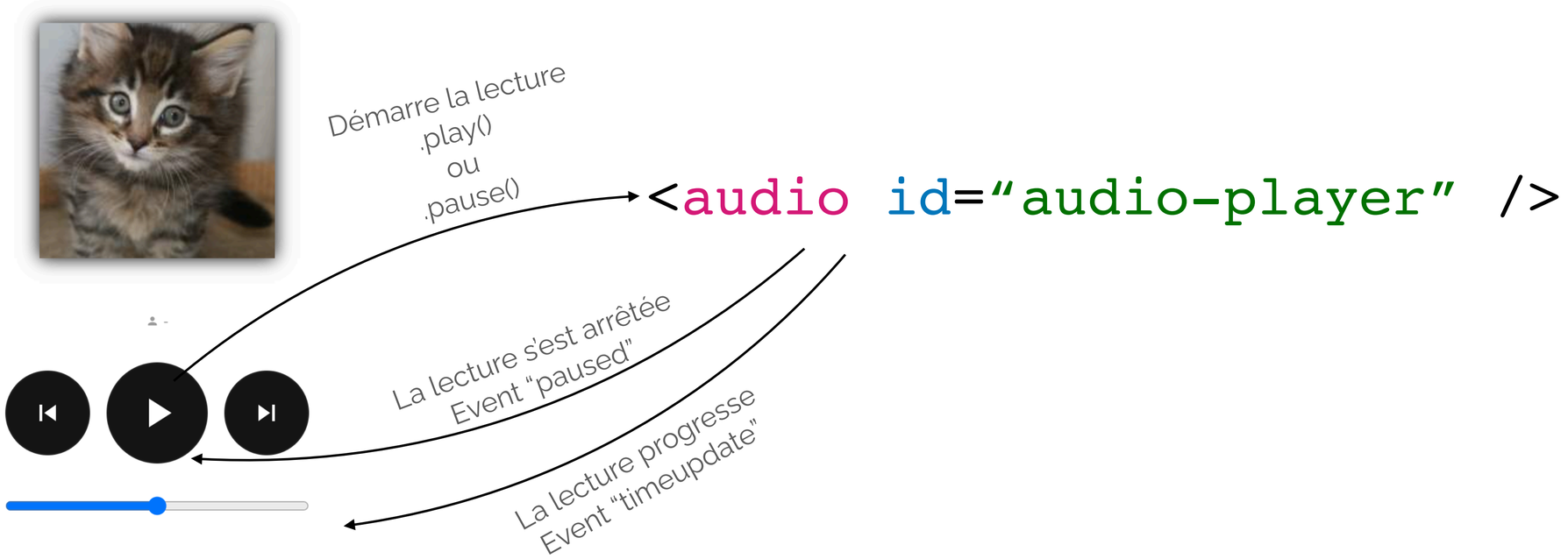
Rétrospective

- Il est important de se rappeler du concept d'observateur/observé



Utiliser le player

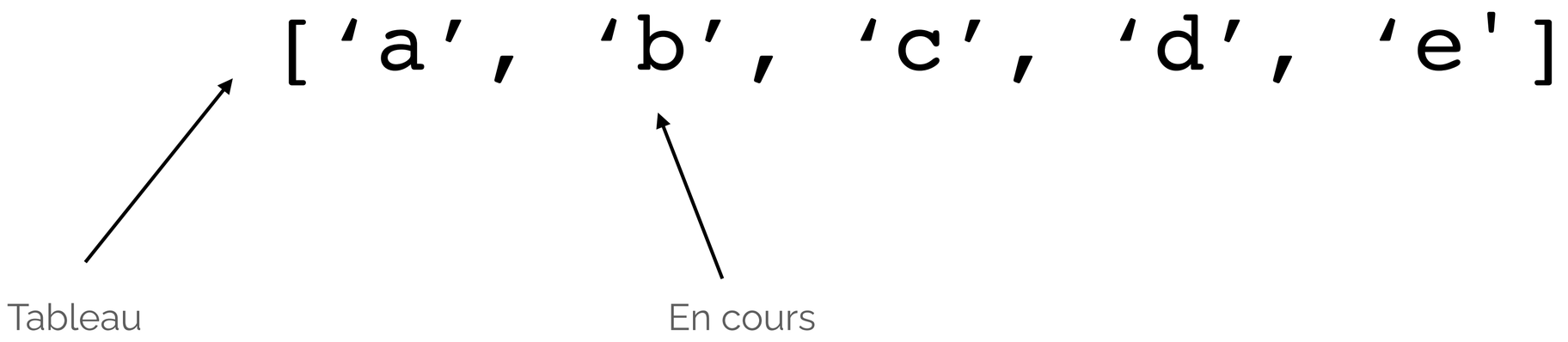
Rétrospective



Utiliser le player

Rétrospective

- Pour la gestion de précédent/suivant, le concept est simple : garder la position d'un élément, au sein d'un tableau d'autres éléments



Utiliser le player

Rétrospective

- Nous avons pour cela créé deux variables que nous modifions via une méthode “playSong”

- La méthode reçoit alors deux infos:
  - La chanson à lire
  - Parmis le tableau transmis

Utiliser le player

Rétrospective

- Lors de son appel, elle s'occupe également de remplacer du contenu HTML d'après les informations reçues (via des modifications d'attributs ou autres variables

Fonction de recherche

Rétrospective

- Prochain problème: Rechercher des chansons

- D'un point de vue abstrait, rechercher des chansons est sensiblement équivalent à charger les chansons d'un artiste:

    - On aimerait une liste de chansons

    - Il y a un critère discriminant - Des chansons qui correspondent au numéro de l'artiste ou qui correspondent à un texte arbitraire

Fonction de recherche

Rétrospective

- Le code devrait donc être sensiblement équivalent…

- On sait maintenant passer des informations dans les URLs et appliquer une méthode particulière, selon ces infos

- Pourquoi ne pas créer une url de recherche ‘#search’ et y passer en paramètre le texte recherché ? ‘#search-marecherche’

Fonction de recherche

Rétrospective

- Il faut alors utiliser le bon endpoint de l'api pour charger des chansons selon un texte et non un id

- Un moyen de récupérer ces informations sous forme de tableau de chansons, car… nous savons afficher un tableau de chanson !

- Ne reste plus qu'à trouver un moyen de rediriger vers cette url de recherche…

Fonction de recherche

Rétrospective

- L'idée est alors d'écouter les changement sur le champ de recherche et à chaque modification de la valeur, rediriger sur l'url '#search-lavaleur'

```
const searchInput = document.querySelector('#search-input')
searchInput.addEventListener('input', () => {
  window.location.hash = `#search-${encodeURIComponent(searchInput.value)}`
})
```

- Plus qu'à rajouter une condition dans notre hashchange et charger les bonnes chansons

Favoris

Rétrospective

- Prochain problème: Gérer les favoris

- Les favoris ne sont pas gérés par le serveur, mais en local

- Il nous faut donc un genre de tableau, dans lequel stocker des chansons choisies et pouvoir afficher ce tableau

- Egalement un moyen d'ajouter/supprimer des éléments dans ce tableau et vérifier si un élément donné y est présent

Favoris

Rétrospective

- Bonne nouvelle, on sait gérer:
    - Afficher un tableau de chansons
    - Interagir avec les éléments d'une liste
    - Afficher une section, avec une logique particulière
    - Reste juste à trouver un tableau que l'on peut éditer et enregistrer…

Favoris

Rétrospective

- On peut pour cela utiliser JsonStorage

- Il nous faut alors une instance centralisée de celui-ci pour le lire et l'éditer

- localStorage requiert un id pour chaque élément du tableau. Le plus simple est donc d'utiliser l'id de la chanson à ajouter… celui-ci étant unique

Favoris

Rétrospective

- Comme localStorage ne sait pas gérer des objets autres que des chaines, nous avons créé des fonctions d'aide, capable de stocker tout type de contenu grâce à JSON !

```
const setItem = (id, value) => localStorage.setItem(id, JSON.stringify(value))

const getItem = (id) => JSON.parse(localStorage.getItem(id))

const getItems = () => Object.keys(localStorage).map(getItem)

const removeItem = (id) => localStorage.removeItem(id)
```

https://developer.mozilla.org/fr/docs/Web/API/Window/localStorage

Favoris

Rétrospective

- Ajouter un listener pour le clique sur le bouton “Favori” d’un élément de la liste. Il va ajouter la chanson au tableau des favoris ou l’enlever, selon son état

- Réutiliser l’affichage des chansons, en ne changeant que leur provenance

- Voir solution dans le projet corrigé, ainsi que les slides “Favoris”, Cours 5

Aperçu global

Structure