Cours 420-5C6-LI Applications web II Automne 2025 Cégep Limoilou Département d'informatique Professeur : Martin Simoneau	**Formatif 3** **Référence – CSS – LocalStorage - Context**

Objectifs

- Établir des références entre les balises *html* et les composants React
- Gérer le formatage
- Utiliser le local Storage

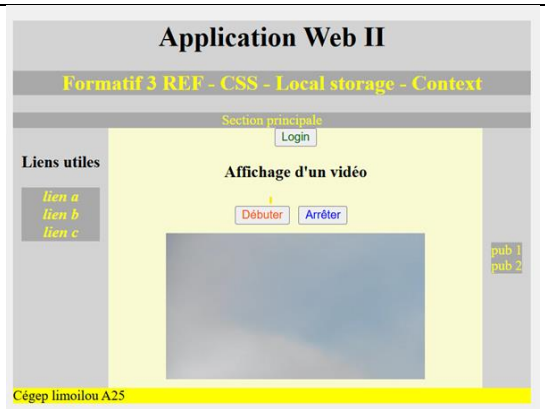
Remise:

- Le travail sera remis sur Léa à la date indiquée.

Théorie : (vous devez vous assurer de bien maitriser les éléments suivants):

1) *Gestion des références*
   a) https://fr.react.dev/reference/react/useRef
   b) https://fr.react.dev/reference/react/forwardRef
2) *Formattage :*
   a) https://legacy.reactjs.org/docs/dom-elements.html#style
   b) https://www.w3schools.com/react/react_css_styling.asp
   c) https://medium.com/@navneetskahlon/a-beginners-guide-to-css-modules-in-react-easy-styling-with-scoped-css-a47cbbb13d07#:~:text=CSS%20modules%20are%20a%20powerful%20tool%20for%20styling,allowing%20for%20easy%20reuse%20of%20styles%20across%20components.
3) *Stockage de données côté client (local storage)*
   a) https://blog.logrocket.com/storing-retrieving-javascript-objects-localstorage/
4) *Context*
   a) https://react.dev/reference/react/createContext
   b) https://react.dev/learn/scaling-up-with-reducer-and-context

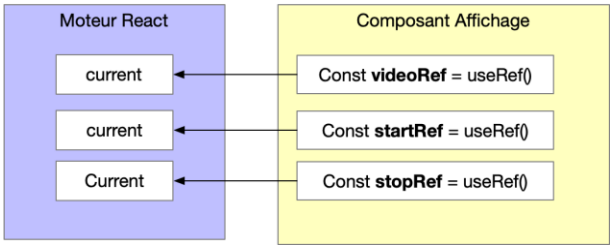
Exercice 1 Références



1) Dans la division ***Content*** vous trouverez la division ***Affichage***. Le premier fait simplement afficher le second. Dans *Affichage*, on veut que le *video* soit lancé lorsqu'on appuie sur le bouton *Débuter* et arrêté lorsqu'on appuie sur le bouton *Arrêter*. Comme les événements *onClick* sont sur les boutons et non sur le vidéo, on a besoin d'un moyen pour accéder au vidéo. On pourrait le faire en naviguant à travers les éléments *html* en partant de l'événement reçu (*event.target.nextSibling.nextSibling.nextSibling.nextSibling*), mais ça rendrait le code fragile puisqu'un petit changement dans le code *html* pourrait corrompre le comportement du code js. Nous allons étudier un mécanisme *React* de gestion de références qui permet de le faire simplement.
2) Lancez et arrêtez le *video* en utilisant une référence *React* sur l'objet *video*.
   videoRef = useRef(); //au début du composant

ref={videoRref} // dans le code JSX de la balise *video*.

a) Comme un composant React ne conserve pas ses données d'un rendu à l'autre, *React* place alors la référence de vidéo dans *videoRef.current*. Le moteur *React* garde une copie de toutes les références de tous les composants. *videoRef* est une poignée vers l'objet que *React* garde pour nous. On a accès à cet objet avec *current.*

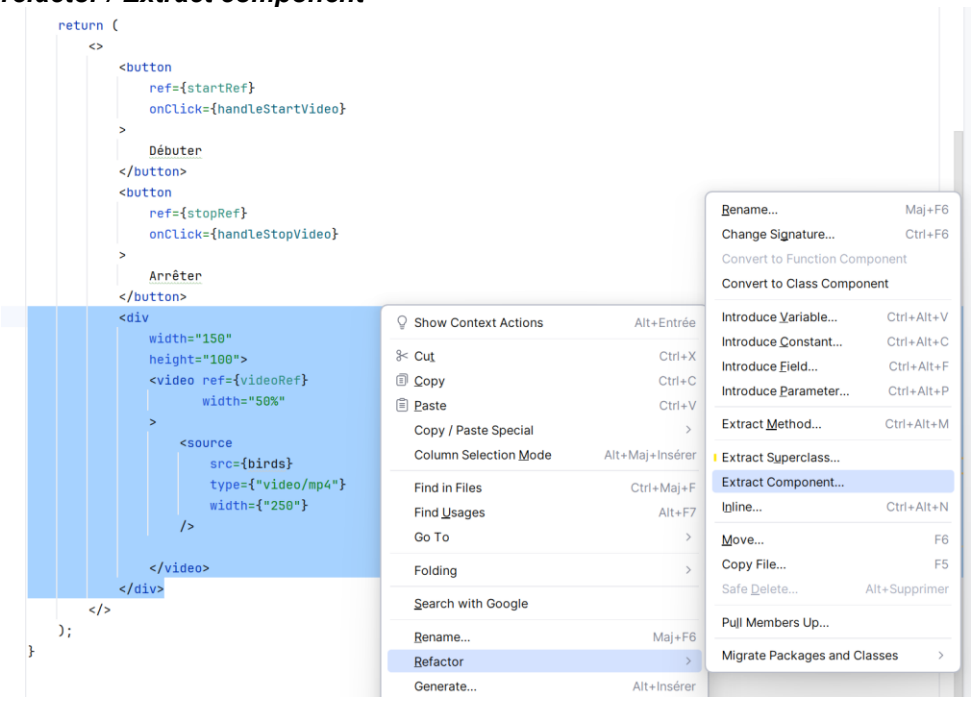


i) Grâce à *useRef*, les éléments impliqués sont gardés dans le moteur *React* et seront accessibles au prochain rendu. Une référence ressemble donc à un état, mais sa modification n'entrainera pas un nouveau rendu de l'affichage. Il sert donc à conserver des informations qui ne sont pas impliquées dans le rendu (une référence vers un élément *html*, un compteur pour les *ids*,).

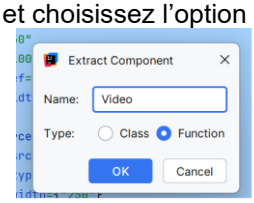
3) On veut également que lorsqu'on appuie sur le bouton *Débuter* le focus passe sur le bouton *Arrêter* et inversement, de sorte qu'en appuyant à répétition sur la barre d'espacement le code lance et arrête le vidéo en alternance.
   a) Pour y arriver, vous aurez besoin de référence sur les boutons *Débuter* et *Arrêter*.
   b) Le *button* js possède une méthode ***focus()*** qui place le focus sur lui (les entrées clavier).

Commitez Étape 1, *useRef* pour lancer et arrêter la vidée, alternance du focus

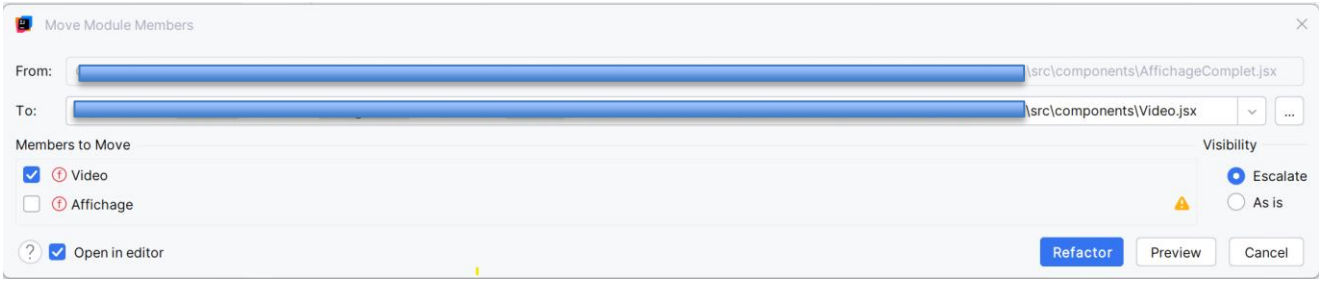
4) Créez un composant *Video*
   a) Transférez-lui la balise *video* qui était dans *Affichage*. Vous pouvez le faire facilement en utilisant un ***refactor / Extract component***



   b) Puis saisissez le nom du composant et choisissez l'option *Function*



c) Le composant est alors créé dans le même fichier. Pour l'envoyer dans son propre fichier, sélectionnez la fonction *Video* et utilisez le ***refactore / Move…***



d) Le composant *Affichage* doit créer un composant *Video* là où était le vidéo.

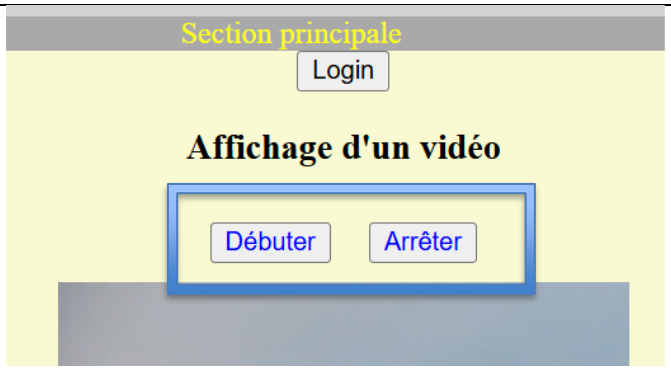
5) **Ça ne fonctionne pas !** Une référence peut être passée facilement par une propriété (du parent vers l'enfant), mais pour qu'une référence remonte vers le parent (que le composant enfant fournisse une référence à son parent) il faut utiliser la méthode React *forwardRef*.
   a) La fonction *forwardRef* reçoit en paramètre la fonction qui crée le composant. Ici, le composant *Affichage* est placé comme paramètre dans *forwardRef*. (Notez que *forwardRef* n'est plus nécessaire à partir de *React 19*, https://react.dev/blog/2024/12/05/react-19#ref-as-a-prop).
      i) Modifiez la déclaration du composant *Video* pour :
         export const Video = forwardRef(function Video(props, ref) {…}

      ii) Utilisez le composant *Video* dans *Affichage* et associez-lui *videoRef*.
         <Video ref={videoRef}/>

   b) Notez que forwardRef ne peut fournir qu'une seule référence. Si vous avez besoin de plus, consultez le hook ***useImperativeHandle***. Les *context* que nous verrons plus loin peuvent aussi partager des *ref*.

Commitez Étape 2, *useRef* et *forwardRef* pour passer la référence à un parent

Exercice 2 Formatage CSS



1) On va maintenant créer un style *inline* directement dans le code *jsx*. Pour créer un style en *js*, il suffit de créer un objet. Chaque attribut devient un élément de style (les noms des styles *css* impliquant des tirets sont remplacés par le *camel case* ex : *font-size* devient *fontSize*). *React* réserve un attribut/props nommé *style* pour l'utilisation de style :
   a) Dans le fichier *Affichage.jsx*, fabriquez un style avec un objet js qui
      i) *Color à blue*
      ii) *Margin de 10px;*
   b) Appliquez-le sur les 2 boutons directement dans le code *jsx*.
   c) On peut traiter l'objet de style *js* comme n'importe quel objet *js*. Déplacez l'objet style dans le composant *Content* et passez-le par propriété à *Affichage*. Le résultat devrait être le même.
2) Priorité des styles
   a) Dans le fichier *App*.css ajoutez un style pour avoir tous les boutons avec la couleur du texte *violet.*

```
button{
    color :violet;
}
```

b) Remarquez que seul le bouton *Login* est en violet. C'est parce que les styles dans le code ont priorité sur ceux des fichiers *css*. Commentez le style sur le premier bouton dans Affichage et vous aurez :

c) Maintenant, appliquez le même format, mais avec la couleur green dans *index.css*. Vous obtiendrez :

d) Pourquoi est-ce que les styles dans *index.css* sont prioritaires aux styles dans *app.css* ? Simplement parce qu'ils sont chargés après dans *main.jsx*.

```
import App from './App.jsx'
import './index.css'
```

e) Inversez les 2 imports et vous verrez les deux boutons passez au violet.

Commitez Étape 5-8, Formatage en ligne et priorité d'application des styles

3) Module de style

a) Nous allons créer un module *css* pour le composant *Affichage*. Les styles de classes définis dans un module ne sont valide que pour le module, il n'interfère pas avec le reste du code.

i) Créez un dossier *affichage* dans components et déplacez-y *Affichge.jsx*.

ii) Dans le dossier affichage créez un fichier ***Affichage.module.css*** (le nom est important, attention à ne pas le modifier). Dans ce fichier CSS ajoutez le style suivant :

```
.orangered{
        color:orangered;
}
```

iii) Notez qu'on utilise un style de classes. Le mécanisme de module n'isole que les styles de classes.

b) Vous devez importer le module de style dans le composant *Afficahge.jsx* et lui donner un nom (ici styles)

```
import styles from "./Affichage.module.css";
```

c) Appliquez le style *orangered* par programmation directement dans le code *jsx* de *Affichage.jsx* en ajoutant l'attribut suivant au premier bouton.

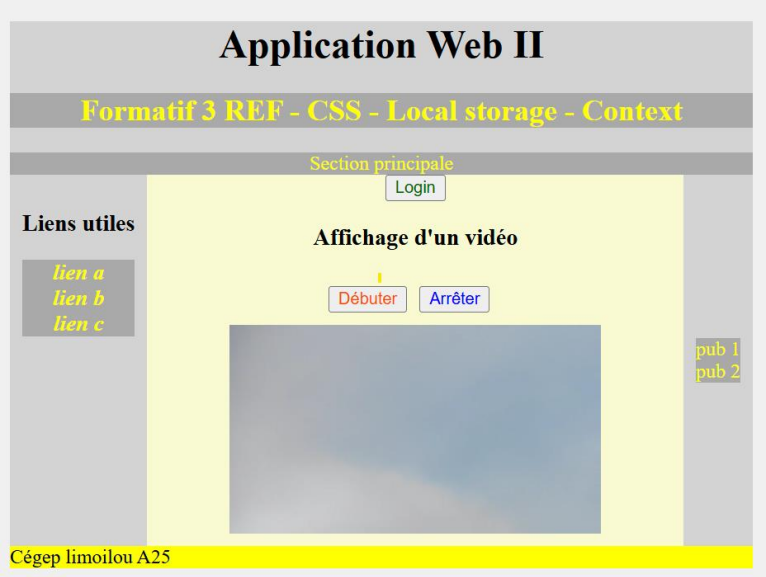
i) className={styles.orangered}

d) Remarquez que même si les styles sont définis dans le composant *Affichage*, ils affectent l'ensemble de la page. Par contre, les styles de classes définies dans un module seront isolés par *React*. Pour le constater, ouvrez l'outil de développement du navigateur et observez les styles appliqués au bouton. Vous trouverez quelque chose ressemblant à : *_orangered_4rm90_1*.

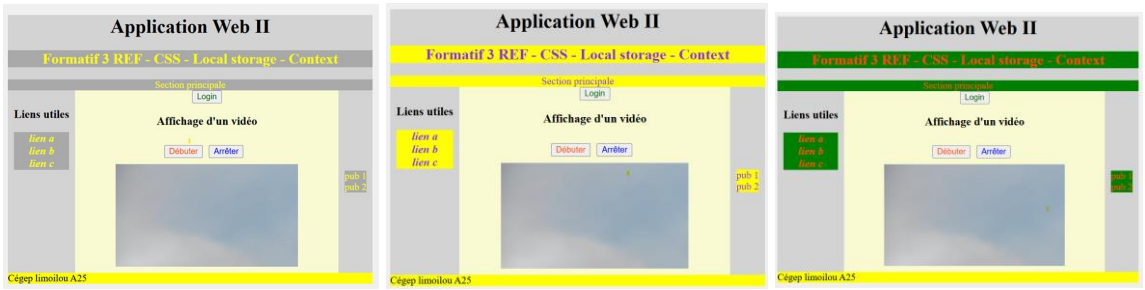
```
<h3>Affichage d'un vidéo</h3>
<button class="_orangered_4rm90_1" style>Débuter</button> == $0
<button style="color: blue; margin: 10px;">Arrêter</button>
```

Exercice 3 Context

1) Le projet original contient déjà plusieurs composants simples. Si vous les lancez, vous devriez voir un site comme :



2) Nous allons utiliser un *context* pour changer facilement le style (il y a déjà 3 thèmes dans le fichier *app.css*) :



3) Le thème modifie

   i) le titre dans l'entête (*Header*)

   ii) Les *pubs* dans la zone gauche (*Pubs*)

   iii) Les liens dans la zone droite (*Aside*)

4) En utilisant un état avec les props, il faudrait le mettre dans l'application et le faire cascader partout à travers tous les composants. Un *context* peut être utilisé partout sans passage explicite par les *props*. Ça simplifie le code grandement. Vous devez donc :

   i) Créez un *context* dans le fichier ***ThemeContext.jsx*** avec la méthode *createContext*(). Choisissez l'une des classes de thème comme valeur par défaut (valeur utilisée uniquement s'il n'y a pas de provider, voir plus loin).

   ii) dans *App.jsx* :

      (1) Importez *ThemeContext*

      (2) Ajoutez une balise *ThemeContext.Provider* autour de la division principale avec une valeur initiale à votre gout (thème de votre choix).

```
<ThemeContext.Provider value={"theme-gray"}>…
```

iii) Dans *Pub.jsx*, *Header.jsx* et Link.jsx :

(1) Importez *ThemeContext*

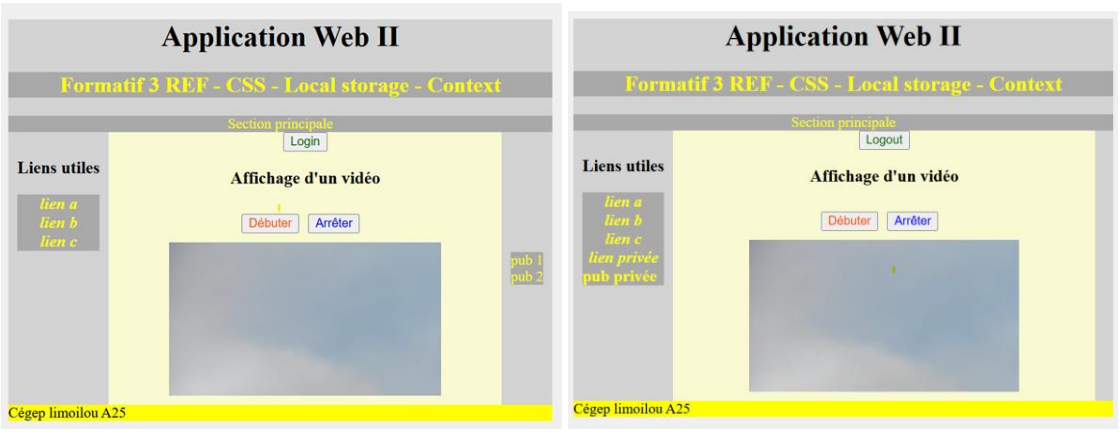
(2) Avec ***useContext***() récupérez la valeur actuelle et utilisez-la comme classe pour les éléments à styliser (avec l'attribut *className*).

iv) Remarquez que si vous ne mettez pas de provider dans *App.jsx*, c'est la valeur par *defaut* utilisée avec *createContext*() qui sera utilisée. Un composant peut donc utiliser un *context* même sans *provider*. Ça facilite l'utilisation des composants qui utilisent des *context*, puisqu'ils fonctionneront même si l'on ne leur fournit pas le *context* en question. Pour le vérifier, commentez temporairement la balise *ThemeContextProvider* dans *App.jsx*.

v) Assurez-vous qu'en modifiant le thème du Provider, le thème affiché change conséquemment.

Commitez **Étape-1 *Context* statique pour gérer un thème**

5) On désire maintenant gérer un pseudo-login (uniquement l'aspect visuel). Le fait d'être connecté sur un site peut changer considérablement l'interface.



6) Lorsque l'utilisateur est connecté, on va apporter les changements suivants :

i) Un ***Lien privé*** apparaît

ii) La section des publicités disparaît

iii) Le bouton *login* laisse la place au bouton ***logout***

7) Comme pour la gestion des thèmes, utiliser un état nous obligerait à le transmettre à travers toute une série de composants. Encore une fois, nous allons utiliser un *context* pour éviter cela.

i) Créez un context ***LogStateContext*** et un fichier jsx du même nom. La valeur du context sera un simple *boolean* indiquant si l'usager est *logged*. Évidemment, à l'ouverture de l'application, il devrait être déconnecté, donc avec une valeur *false*.

ii) Le *context* n'est pas un état en lui-même, mais il peut servir à en diffuser un efficacement dans une large partie de l'application. C'est un moyen plus global pour partager une valeur, un état ou une référence. Il faudra donc créer un état *isLogged* dans *App.jsx*.

iii) Au lieu de diffuser l'état aux sous-composants par les *props*, on va le diffuser avec le *context*. Entourez donc la division principale de l'application par une balise *<LogStateContext.Provider>* avec l'attribut value contenant un objet avec l'état et le modificateur d'état. Notez qu'il faut 2 accolades un pour passer de *jsx* à *js* et l'autre pour créer l'objet :

```
<LogStateContext.Provider value={{isLoged, setIsLoged}}>
```

iv) Modifiez les composants suivants pour refléter correctement l'état *isLogged* retrouvé avec *useContext*();

(1) *Aside* : ajout d'un lien privé si connecté

(2) *Pubs* : la section n'est pas affichée lorsque l'utilisateur n'est pas connecté.

v) Dans la section Content, ajoutez 2 nouveaux composants *Login* et *Logout* dans leur fichier respectif. Chacun ne contient qu'un seul bouton (respectivement *login* et *logout*).

(1) Login : n'est affiché que lorsque l'utilisateur n'est pas connecté et lorsqu'on appuie dessus, il utilise le setter contenu dans *LogStateContext* pour indiquer que l'utilisateur est connecté.

(2) Logout : fait l'inverse.

vi) Assurez-vous que tout fonctionne correctement.

Commitez **Étape-2 Context avec état pour gérer le login/logout**

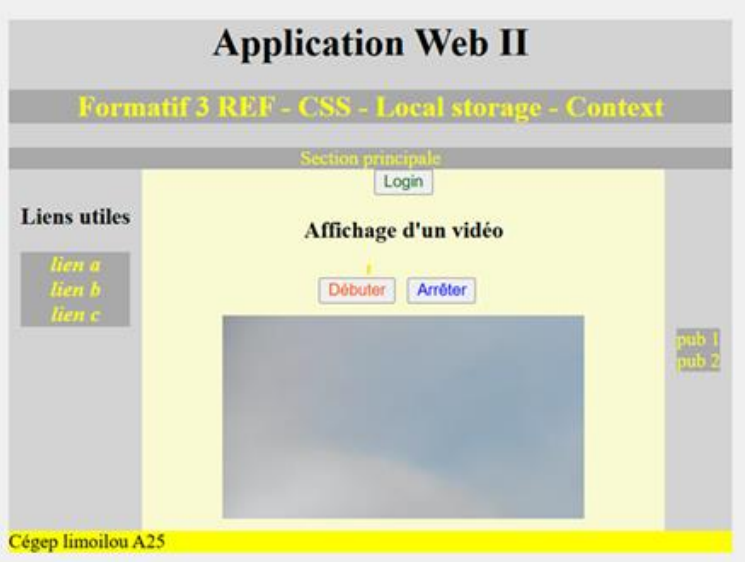
8) On a défini le *context* provider à partir de *App.jsx*. Il est possible de le définir n'importe où. C'est le *context provider* le plus proche qui sera utilisé. Un *context provider* intermédiaire peut même réutiliser/bonifier le *context provider* défini plus haut. On va maintenant créer un sous-thème pour afficher les liens de la section Liens utiles avec une police de taille 34px.

i) Dans *App.css*, créez une classe css *theme-big* qui définit une police à 34 px (*font-size*)

ii) Dans le composant *Aside*,

(1) Récupérer le *ThemeContext.*

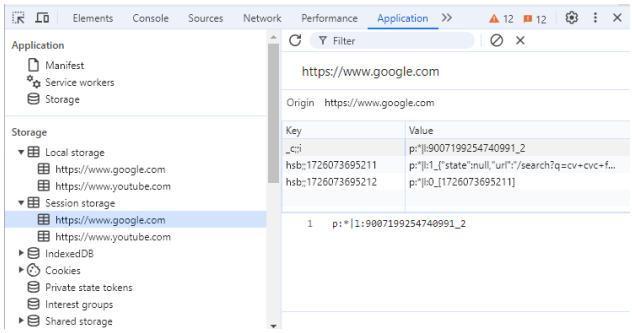
(2) Ajoutez un *context* provider autour de la balise *aside*. La nouvelle valeur est la concaténation des thèmes définis par le provider parent avec la chaine « *theme-big* ».



Commitez **Étape-3 Context partiellement redéfini par un sous composant.**

Exercice 4 Local Storage

9) Le navigateur nous offre 2 outils pour garder des données d'une application client au-delà d'une simple session : ***localStorage*** et ***sessionStorage***. Vous pouvez consulter les données emmagasinées par une application dans l'onglet ***Application*** des outils de développeur du navigateur.



a) *Les 2 types de stockages sont des maps qui associent des clés avec des valeurs. On y retrouve les méthodes suivantes :*
   i) ***setItem()*** *: Ajoute des données à un objet Web Storage à l'aide de ses deux arguments, une clé et une valeur :* `localStorage.setItem("key", "value")`
   ii) ***getItem()*** *: Renvoie la valeur du nom de clé qui lui est passé :* `localStorage.getItem("key")`
   iii) ***removeItem()****: Supprime une clé qui lui est passée avec sa valeur correspondante :* `localStorage.removeItem("key")`
   iv) ***clear()*** *: efface toutes les paires clé-valeur dans le stockage associé (doit être utilisé avec prudence :* `localStorage.clear())`
   v) ***key()****: Renvoie la clé à l'index spécifié dans le stockage :* `localStorage.key(0)`
   vi) ***length*** *: renvoie le nombre total de paires clé-valeur stockées dans le stockage associé :* `localStorage.length`

b) On peut y stocker un peu n'importe quoi, des tableaux, des objets, des données simples, mais pas de méthode ou procédure. Pour y stocker un objet, il faut utiliser la conversion *JSON*.

```
i) localStorage.setItem("userObj", JSON.stringify(myObject));
ii)let newObject = localStorage.getItem("myObject");
console.log(JSON.parse(newObject));
```

10) Utilisez l'outil ***localStorage*** pour emmagasiner le compteur de référence dans le composant Affichage.
   a) Pour l'enregistrer, il faut le faire lorsque l'application se ferme. On reçoit alors l'événement *beforeunload* de l'objet *js window*.
      window.addEventListener("beforeunload", (ev) => {…}
   b) Pour le récupérer, il faut le faire lorsque l'application est chargée. On reçoit alors l'événement *load* de l'objet *js window*.
      window.addEventListener("load", (ev) => {…}

11) Essayez le même code en remplaçant *localStorage* par ***sessionStorage***. Tant qu'une fenêtre de navigateur est ouverte sur le site, les données vont perdurer, mais elles seront effacées lorsque toutes les fenêtres du navigateur sur le site seront fermées.

FIN