**Soutenance projet 6 Sophie Bluel**

Le code que nous allons étudier est une application web qui permet aux utilisateurs de parcourir une galerie d'images. Cette application offre plusieurs fonctionnalités, notamment la visualisation des images, la filtration par catégorie, l'ajout et la suppression d'images, le téléchargement de nouvelles images et un système d'authentification pour les utilisateurs.

La soutenance portera sur la présentation et l'explication détaillée du code JavaScript développé pour la gestion d'une galerie d'images. Le code comprend plusieurs fonctions et commandes qui permettent l'affichage des œuvres, la gestion des catégories, l'ajout et la suppression d'images, ainsi que la gestion de l'authentification des utilisateurs.

**Index.js**

**. Récupération des Œuvres depuis l'API :**

La récupération les données des images à partir d'une API en utilisant la fonction **fetch**. Une fois les données récupérées, elles sont utilisées pour créer la galerie d'images en appelant la fonction **createWorks**, les catégories sont initialisées en appelant la fonction **initCategories**.

**Initialisation des Catégories :**

La fonction **initCategories** récupère les catégories à partir de l'API, puis crée des boutons de filtre pour chaque catégorie. Ces boutons permettent aux utilisateurs de filtrer les images affichées en fonction de la catégorie sélectionnée, la fonction **createCategories** pour créer et afficher les boutons de filtre correspondant aux catégories.

**Affichage des Œuvres (cartes) :**

La fonction **createWorks** génère les éléments HTML nécessaires pour afficher les images de la galerie dynamiquement pour les afficher sur la page, et parcourt chaque œuvre et crée une carte comprenant une image, un titre et d'autres éléments interactifs.

**Création des Boutons de Filtres (par Catégorie) :**

La fonction **createCategories** génère des boutons de filtre pour chaque catégorie de travail. Ces boutons sont ensuite ajoutés au DOM et associés à une fonction de gestion d'événements pour filtrer les œuvres en fonction de la catégorie sélectionnée.

la gestion de la création des boutons filtres par catégorie est réalisée par la fonction **createCategories(categoriesData, works)** :

**Sélection de l'élément conteneur :**

La fonction commence par récupérer l'élément HTML qui servira de conteneur pour les boutons filtres. Cet élément est généralement identifié par un ID unique, tel que **"filter"**.

**Sélection de l'élément conteneur :**

La fonction commence par récupérer l'élément HTML qui servira de conteneur pour les boutons filtres. Cet élément est généralement identifié par un ID unique, tel que **"filter"**.

**const myDiv = document.getElementById("filter");**

**Création des boutons filtres :**

Ensuite, la fonction parcourt les données des catégories récupérées depuis l'API et crée un bouton pour chaque catégorie.

La boucle **for** parcourt chaque élément dans le tableau **categoriesData**, qui contient les données des catégories récupérées depuis l'API.

Pour chaque catégorie, un nouvel élément bouton est créé avec **document.createElement("button")**

**Ajout des événements de clic :**

Un gestionnaire d'événements de clic est ajouté à chaque bouton créé. Lorsque l'utilisateur clique sur l'un de ces boutons, la fonction **changeState(event, works)** est déclenchée.

**Appel de la fonction de changement d'état :**

La fonction de changement d'état (**changeState**) est appelée lorsqu'un bouton filtre est cliqué. Cette fonction est responsable de filtrer les œuvres en fonction de la catégorie sélectionnée et de mettre à jour l'affichage en conséquence.

**Gestion du Changement d'État (Filtrage) :**

La fonction **changeState** est appelée lorsqu'un bouton de filtre est cliqué. Elle met à jour l'apparence des boutons pour indiquer celui qui est sélectionné, puis filtre les œuvres en fonction de la catégorie choisie en appelant la fonction **filter**.

**Filtre des Œuvres :**

La fonction **filter** prend en compte le nom du filtre sélectionné ainsi que les données sur les œuvres et filtre les œuvres correspondantes à la catégorie choisie.

**Login.js (Connexion Utilisateur) :**

Le code gère la soumission du formulaire de connexion, en envoyant les données au serveur et en traitant la réponse. S'il s'agit d'une connexion réussie, un token est stocké localement et l'utilisateur est redirigé vers une autre page. En cas d'échec de connexion, un message d'erreur est affiché.

**const form = document.getElementById("connexion-form");**

**const valid = document.getElementById("submit1");**

**const email = document.getElementById("email");**

**const password = document.getElementById("password");**

**const errorMsg = document.getElementById("errorMsg");**

Ces lignes récupèrent différents éléments HTML du formulaire de connexion à l'aide de leurs identifiants.

**form.addEventListener("submit", function (e) {**

// Actions à effectuer lors de la soumission du formulaire

**});** cette ligne ajoute un écouteur d'événements pour détecter la soumission du formulaire de connexion.

**e.preventDefault();** Cette ligne empêche le comportement par défaut du formulaire, qui serait de recharger la page lors de la soumission.

**const formInfo = new FormData(form);**

**const payload = new URLSearchParams(formInfo);** Ces lignes récupèrent les données saisies dans le formulaire et les encapsulent dans un objet **FormData**, ces données sont transformées en une chaîne de requête URL à l'aide de **URLSearchParams**.

**fetch("http://localhost:5678/api/users/login", {**

**method: "POST",**

**headers: {**

**accept: "application/json",**

**},**

**body: payload,**

**}) :** Cette ligne envoie une requête POST au serveur à l'URL spécifiée, en utilisant les données de connexion encapsulées dans le **payload**.

**.then((res) => res.json())**

**.then((data) => {**

// Actions à effectuer après avoir reçu la réponse du serveur **})**

**.catch((err) => console.log(err)); :** Ces lignes traitent la réponse du serveur. Si la requête est réussie, les données de réponse sont extraites au format JSON. Si une erreur se produit pendant la requête, elle est gérée dans le bloc **catch**.

**if (data.userId == 1) {**

**localStorage.setItem("token", data.token);**

**location.href = "index.html";**

**} else {**

**// Affichage d'un message d'erreur**

// Réinitialisation des champs email et password

// Suppression du message d'erreur après un certain délai **} :** Cette partie vérifie les données de réponse reçues du serveur. Si les informations d'identification sont correctes (par exemple, l'ID de l'utilisateur est 1), le jeton d'authentification est stocké localement et l'utilisateur est redirigé vers la page principale (**index.html**). Sinon, un message d'erreur est affiché et les champs du formulaire sont réinitialisés.

**Admin.js (Gestion de l'Authentification Utilisateur) :**

**let token = localStorage.getItem("token");** cette ligne utilise la méthode **getItem** de l'objet **localStorage** pour récupérer le jeton d'authentification stocké localement sous la clé "token".

**if (token) {**

**// Actions à effectuer si l'utilisateur est connecté } :** cette ligne vérifie si le jeton d'authentification existe dans le local storage, ce qui signifie que l'utilisateur est déjà connecté.

**document.querySelector(".bando").classList.remove("hidden");**

**document.getElementById("modifier").style.display = "flex";**

**document.getElementById("modifiers").style.display = "flex";**

**document.getElementById("filter").style.display = "none";**

**document.getElementById("login").innerText = "logout";**

* Ces lignes activent certaines fonctionnalités lorsque l'utilisateur est connecté avec succès, et rendent visible l’élément avec la classe "bando", affichent les boutons de modification, masquent les boutons de filtre et changent le texte du bouton de connexion en "logout".

**document.getElementById("login").addEventListener("click", function (event) {**

**// Actions à effectuer lors du clic sur le bouton de déconnexion });**

* + Cette ligne ajoute un écouteur d'événements pour détecter le clic sur le bouton de connexion.
  + Lorsque l'utilisateur clique sur ce bouton, les actions suivantes sont effectuées :
    - Le jeton d'authentification est supprimé du local storage à l'aide de la méthode **removeItem**.
    - Certains éléments sont masqués ou modifiés pour refléter l'état de déconnexion de l'utilisateur, tels que la classe "bando" et les boutons de modification.

Le fichier admin.js assure la gestion de l'authentification de l'utilisateur en vérifiant la présence d'un jeton d'authentification dans le local storage. Si un jeton est présent, cela signifie que l'utilisateur est connecté, et certaines fonctionnalités spécifiques sont activées en conséquence. Lorsque l'utilisateur se déconnecte, le jeton d'authentification est supprimé et les fonctionnalités associées à la connexion sont désactivées.

Haut du formulaire

La fonction **init()** vérifie si un token d'authentification est présent dans le stockage local. Si oui, elle affiche certaines fonctionnalités réservées aux utilisateurs connectés, telles que la possibilité de modifier la galerie. Si l'utilisateur se déconnecte, le token est supprimé et les fonctionnalités sont masquées.

Enfin, le code vérifie s'il existe un token d'authentification stocké localement. S'il est présent, il affiche certains éléments de l'interface utilisateur et définit des événements pour la connexion et la déconnexion de l'utilisateur.

**Modal.js :**

La récupération des éléments associés au modal, notamment le modal lui-même, le bouton d'ouverture du modal (**btn**), et le bouton de fermeture du modal (**span**). Ces éléments sont sélectionnés à l'aide de leur identifiant unique défini dans le code HTML.

**Ouverture du modal :**

Lorsque le bouton d'ouverture du modal est cliqué (**btn.onclick**), la fonction (**btn.onclick = function () {**

**modal.style.display = "block";)** associée est déclenchée.

Cette fonction définit le style du modal pour l'afficher (**modal.style.display = "block";**). Ensuite, elle récupère tous les éléments avec la classe **iconPoubelle** à l'intérieur du modal, qui sont les icônes de poubelle associées à chaque élément, et leur ajoute un gestionnaire d'événements pour le clic.

**Gestion de la Suppression :**

Chaque fois qu'une icône de poubelle est cliquée à l'intérieur du modal, une fonction est déclenchée. Cette fonction envoie une requête **Fetch** de type **DELETE** vers l'API avec l'ID de l'œuvre associée à l'icône de poubelle. Une fois la suppression effectuée avec succès, les œuvres sont réinitialisées **(init()),** ce qui met à jour la liste des œuvres affichées sur la page.

**Fermeture du modal :**

Lorsque l'utilisateur clique sur le bouton de fermeture du modal (**span.onclick**) ou en dehors du modal (**window.onclick**), le modal est caché en modifiant le style CSS (**modal.style.display = "none";**). Cela permet de fermer le modal lorsque l'utilisateur clique sur le bouton de fermeture ou en dehors du contenu du modal.

**Autres fonctionnalités :**

la fonction **ouvre\_modal\_ajoute** est définie pour ouvrir un autre modal, celle pour ajouter une photo, elle gère l'affichage du modal et la mise en place des événements nécessaires.

la fonction **close\_modal\_ajouter** est également définie pour fermer ce modal.

la gestion du modal dans ce code permet à l'utilisateur d'ouvrir, de fermer et d'interagir avec le contenu du modal de manière fluide, tout en offrant des fonctionnalités supplémentaires telles que la suppression d'éléments spécifiques à l'intérieur du modal.

**Téléchargement des Photos :**

La fonction **telecharger** gère le téléchargement d'une photo sélectionnée par l'utilisateur. Elle utilise un objet **FileReader** pour lire le contenu de la photo et l'afficher dans un élément HTML.

**Sélection de l'élément d'entrée de fichier :**

**const input = document.getElementById("img\_input");**

La fonction commence par sélectionner l'élément HTML **<input>** avec l'ID **"img\_input"**, qui est utilisé pour télécharger les images.

**Initialisation de la variable telecharge\_image :** **var telecharge\_image = "";**

Une variable **telecharge\_image** est initialisée pour stocker l'image téléchargée, elle affiche l'image dans l'interface utilisateur.

**Création d'un objet FileReader :** **const reader = new FileReader();**

Un nouvel objet **FileReader** est créé pour lire les données de l'image sélectionnée par l'utilisateur.

**Ajout d'un écouteur d'événements load :**

**reader.addEventListener("load", () => {**

**});**

Un écouteur d'événements **load** est ajouté au lecteur de fichiers. Cet événement se déclenche lorsque la lecture du fichier est terminée avec succès.

**Lecture du contenu de l'image :**

**reader.readAsDataURL(this.files[0]);**

À l'intérieur de l'écouteur d'événements **load**, le contenu de l'image téléchargée est lu à l'aide de la méthode **readAsDataURL**. Cela convertit le contenu de l'image en une URL de données au format base64.

**Affichage de l'image téléchargée :**

**photo.style.backgroundImage = `url(${telecharge\_image} )`;**

Une fois que la lecture de l'image est terminée, l'URL de données de l'image est stockée dans la variable **telecharge\_image**. Ensuite, cette URL est utilisée pour définir l'arrière-plan de l'élément HTML qui affiche l'image téléchargée dans l'interface utilisateur.

**Masquage de l'élément de téléchargement d'image:** **document.getElementById("telechargerImage").style.display = null**;

L'élément HTML qui permet à l'utilisateur de télécharger l'image est masqué après le téléchargement réussi de l'image.

**Masquage du conteneur modal d'ajout d'image :** **document.getElementById("container\_modalAjouter").style.display = "none";**

Une fois que l'image télécharger, le conteneur modal d'ajout d'image est également masqué pour revenir à l'état initial de l'interface utilisateur.

cette fonction permet à l'utilisateur de télécharger une image en sélectionnant un fichier via un champ d'entrée de fichier, l'image téléchargée est ensuite affichée dans l'interface utilisateur, et les éléments associés sont masqués pour finaliser l'opération de téléchargement.

**Initialisation des Catégories depuis l'API :**

Une autre requête **Fetch** est effectuée pour récupérer les catégories depuis l'API. Les catégories sont ensuite utilisées pour remplir un menu déroulant sur la page.

**Fonction Supprimer :**

**suprimer** réinitialise les valeurs des éléments du formulaire et masque certains éléments lorsque le modal est fermé.

. **document.getElementById("container\_modalAjouter").style.display = null;** : Cette ligne rend visible le conteneur du modal d'ajout de photo, en définissant la valeur de **display** sur **null**, cela rétablit l'affichage par défaut spécifié dans la feuille de style CSS.

. **document.getElementById("telechargerImage").style.display = "none";** : Cette ligne masque l'élément HTML qui affiche l'image téléchargée, en définissant la valeur de **display** sur **"none"**, cet élément est masqué de manière invisible pour l'utilisateur.

**Réinitialisation du champ de saisie du titre (input\_modal) :** La ligne récupère l'élément du champ de saisie du titre par son ID (**input\_modal**) et réinitialise sa valeur en définissant **value** à **null**. Cela efface le contenu du champ de saisie après avoir ajouté une photo.

**Réinitialisation du champ de saisie de l'URL de l'image (img\_input) :** De manière similaire à l'étape précédente, cette ligne réinitialise la valeur du champ de saisie de l'URL de l'image en définissant **value** à **null**.

**Réinitialisation du champ de sélection de catégorie (categorie) :** Cette ligne réinitialise la valeur du champ de sélection de catégorie en définissant **value** à **null**. Cela efface la sélection de catégorie faite par l'utilisateur après avoir ajouté une photo.

En résumé, la fonction **suprimer** assure que toutes les données et les éléments associés sont réinitialisés à leur état initial lorsque l'utilisateur ferme le modal d'ajout de photo. Cela permet une expérience utilisateur cohérente et propre à chaque nouvelle utilisation du modal.

Haut du formulaire

### Conclusion:

Le code JavaScript développé permet de gérer efficacement une galerie d'images en offrant des fonctionnalités telles que l'affichage des œuvres, la gestion des catégories, l'ajout et la suppression d'images, ainsi que la gestion de l'authentification des utilisateurs. Les différentes fonctions et commandes sont utilisées de manière appropriée pour assurer le bon fonctionnement de l'application.

### Fonction init():

Cette fonction est appelée au chargement de la page. Elle initialise la galerie en effectuant une requête à l'API pour récupérer toutes les œuvres disponibles et leurs informations. Une fois les données récupérées, elle appelle les fonctions **createWorks()** et **initCategories()** pour créer les éléments visuels correspondants.

#### **Justification:**

* **fetch()**: Utilisé pour récupérer les données à partir de l'API. Cette méthode permet d'effectuer des requêtes HTTP asynchrones.
* **then()**: Utilisé pour chaîner les actions à effectuer une fois que la réponse de l'API est reçue. Il permet de gérer les données renvoyées par la requête.

### Fonction createWorks(works):

Cette fonction crée les éléments visuels pour afficher les œuvres dans la galerie. Elle parcourt la liste des œuvres passée en paramètre et crée des éléments HTML correspondants à chaque œuvre, tels que des figures avec des images et des légendes.

#### **Justification:**

* **createElement()**: Utilisé pour créer des éléments HTML dynamiquement.
* **setAttribute()**: Utilisé pour définir les attributs des éléments HTML.
* **appendChild()**: Utilisé pour ajouter des éléments enfants à un élément parent dans le DOM.

### Fonction initCategories(works):

Cette fonction initialise les filtres de catégories en effectuant une requête à l'API pour récupérer toutes les catégories disponibles. Une fois les données récupérées, elle crée des boutons correspondants à chaque catégorie.

#### **Justification:**

* **addEventListener()**: Utilisé pour écouter les événements, tels que les clics sur les boutons de catégories.
* **onclick()**: Utilisé pour attribuer une fonction à exécuter lorsqu'un élément est cliqué.

### Fonction changeState(event, works):

Cette fonction est appelée lorsqu'un filtre de catégorie est sélectionné. Elle met à jour l'état visuel des boutons de catégorie et filtre les œuvres en fonction de la catégorie sélectionnée.

#### **Justification:**

* **style.background**: Utilisé pour changer la couleur de fond des boutons de catégorie.
* **style.color**: Utilisé pour changer la couleur du texte des boutons de catégorie.

### Fonction filter(nameDuFiltre, works):

Cette fonction filtre les œuvres en fonction de la catégorie sélectionnée. Elle crée une nouvelle liste d'œuvres qui correspondent à la catégorie sélectionnée.

#### **Justification:**

* Utilisation d'une boucle **for-of** pour parcourir toutes les œuvres et sélectionner celles qui correspondent à la catégorie choisie.

1. **Index.js**
2. **function init() {**: Définition de la fonction **init()** qui initialise la galerie d'images.
3. **fetch("http://localhost:5678/api/works")**: Utilisation de la fonction **fetch()** pour récupérer les données des œuvres depuis l'API.
4. **.then((response) => response.json())**: Chaînage d'une promesse pour convertir la réponse en format JSON.
5. **.then((works) => {**: Chaînage d'une autre promesse pour traiter les données JSON récupérées (les œuvres).
6. **createWorks(works);**: Appel de la fonction **createWorks()** avec les données des œuvres en paramètre pour afficher les œuvres dans la galerie.
7. **initCategories(works);**: Appel de la fonction **initCategories()** avec les données des œuvres en paramètre pour initialiser les filtres de catégories.
8. **});**: Fermeture des promesses chaînées.
9. **}**: Fermeture de la fonction **init()**.
10. **init();**: Appel de la fonction **init()** pour lancer le processus d'initialisation de la galerie dès que le script est chargé.

#### **Justifications :**

* La fonction **fetch()** est utilisée pour récupérer les données des œuvres depuis l'API. Cela permet d'obtenir les informations nécessaires à l'affichage de la galerie.
* Les méthodes **.then()** sont utilisées pour traiter les données renvoyées par la requête HTTP de manière asynchrone, assurant ainsi une exécution fluide du code.
* Les fonctions **createWorks()** et **initCategories()** sont appelées pour créer les éléments visuels nécessaires à l'affichage des œuvres et à l'initialisation des filtres de catégories respectivement, garantissant ainsi une expérience utilisateur complète.

**récupération des catégories par api**

**fetch("http://localhost:5678/api/categories")**: Cette ligne envoie une requête HTTP GET à l'URL spécifiée pour récupérer les données des catégories depuis l'API à l'adresse **http://localhost:5678/api/categories**.

**.then((response) => response.json())**: Cette ligne traite la réponse de la requête HTTP sous forme de flux JSON, renvoyé par le serveur. Cela convertit la réponse en un objet JavaScript utilisable.

1. **.then((categoriesData) => { createCategories(categoriesData, works); })**: Cette ligne traite les données JSON obtenues de la réponse de la requête. Une fois que les données des catégories sont disponibles, elles sont passées à la fonction **createCategories()** avec les données des œuvres (**works**). Cela permet d'initialiser et de générer dynamiquement les boutons de filtre basés sur les catégories récupérées de l'API.

#### **Justifications :**

* L'utilisation de **fetch()** permet de récupérer de manière asynchrone les données des catégories depuis l'API, ce qui évite de bloquer l'exécution du script et rend l'application plus réactive.
* La méthode **.then()** est utilisée pour gérer la réponse de la requête HTTP, en traitant d'abord la réponse en tant que flux JSON à l'aide de **.json()**.
* La fonction fléchée passée à **.then()** traite les données des catégories récupérées de l'API. Une fois les données disponibles, elles sont utilisées pour créer dynamiquement les boutons de filtre à l'aide de la fonction **createCategories()**.
* Cette approche permet une gestion asynchrone et réactive des données de catégories, garantissant que les boutons de filtre sont mis à jour en fonction des catégories disponibles dans l'API.

Haut du formulaire

**Affichage des cartes**

1. **const myDiv = document.getElementById("myGallery");**: Récupère l'élément HTML avec l'ID "myGallery", qui est l'endroit où les œuvres seront affichées dans la galerie.
2. **myDiv.innerHTML = "";**: Efface tout le contenu HTML à l'intérieur de l'élément "myGallery" avant de le remplir avec les nouvelles œuvres.
3. **const modalGallery = document.getElementById("modalGallery");**: Récupère l'élément HTML avec l'ID "modalGallery", qui est l'endroit où les œuvres seront affichées dans le modal.
4. **let html = "";**: Initialise une chaîne de caractères vide pour stocker le HTML des nouvelles œuvres à afficher.
5. **for (const work of works) {**: Boucle à travers chaque œuvre dans le tableau **works**.
6. **const itemFigure = document.createElement("figure");**: Crée un nouvel élément **<figure>** pour chaque œuvre.
7. **const itemImg = document.createElement("img");**: Crée un nouvel élément **<img>** pour afficher l'image de l'œuvre.
8. **itemImg.setAttribute("src", work.imageUrl);**: Définit l'attribut "src" de l'élément **<img>** avec l'URL de l'image de l'œuvre.
9. **const itemFigCaption = document.createElement("figcaption");**: Crée un nouvel élément **<figcaption>** pour afficher le titre de l'œuvre.
10. **itemFigCaption.textContent = work.title;**: Définit le texte de l'élément <figcaption> avec le titre de l'œuvre.
11. **itemFigure.appendChild(itemImg);**: Ajoute l'élément <img> à l'intérieur de l'élément <figure>.
12. **itemFigure.appendChild(itemFigCaption);**: Ajoute l'élément <figcaption> à l'intérieur de l'élément <figure>.
13. **myDiv.appendChild(itemFigure);**: Ajoute l'élément <figure> (contenant l'image et le titre) à l'intérieur de l'élément "myGallery".

**html += ...**: Ajoute du code HTML représentant une carte d'œuvre à la chaîne **html**. Cette carte comprend une structure HTML avec l'image, le titre et d'autres éléments tels qu'une icône de poubelle et une option pour éditer.

1. **modalGallery.innerHTML = html;**: Injecte le HTML généré dans le modal, remplaçant tout son contenu précédent par les nouvelles œuvres.

#### **Justifications :**

* La fonction **document.getElementById()** est utilisée pour récupérer les éléments DOM nécessaires à la manipulation et à l'affichage des œuvres dans la galerie et le modal.
* **createElement()** est utilisé pour créer de nouveaux éléments HTML, tels que des figures, des images et des légendes, pour chaque œuvre.
* **setAttribute()** est utilisé pour définir les attributs des éléments HTML, comme l'URL de l'image.
* **textContent** est utilisé pour définir le texte à afficher à l'intérieur des éléments HTML, tels que les titres des œuvres.
* **appendChild()** est utilisé pour ajouter les éléments nouvellement créés à la structure DOM existante.
* La concaténation de chaînes est utilisée pour générer du code HTML représentant les cartes d'œuvre.
* **innerHTML** est utilisé pour injecter le HTML généré dans les éléments DOM correspondants, mettant à jour l'affichage des œuvres dans la galerie et le modal

**création des boutons filters (par catégorie)**

**function createCategories(categoriesData, works) {**: Définit une fonction **createCategories** prenant deux paramètres : **categoriesData** qui contient les données des catégories récupérées depuis l'API, et **works** qui contient les données des œuvres. Cette fonction est responsable de la création des boutons de filtre pour chaque catégorie.

**const myDiv = document.getElementById("filter");**: Récupère l'élément HTML avec l'ID "filter". Cet élément est probablement une **<div>** dans laquelle les boutons de filtre seront ajoutés.

1. **myDiv.innerHTML = "";**: Efface le contenu HTML de l'élément "filter". Cela permet de s'assurer qu'il est vide avant d'ajouter les nouveaux boutons de filtre. Cette ligne garantit également que les boutons de filtre sont recréés à chaque fois que cette fonction est appelée, ce qui évite d'avoir des doublons de boutons.

#### **Justifications :**

* La récupération de l'élément "**filter**" à l'aide de **document.getElementById()** assure que nous ciblons l'élément approprié dans le DOM pour y ajouter les boutons de filtre.
* Effacer le contenu de cet élément avant d'ajouter de nouveaux boutons garantit qu'il n'y a pas de doublons ou de résidus d'éléments précédents qui pourraient interférer avec les nouveaux boutons de filtre.
* En réinitialisant le contenu de l'élément **"filter**" à une chaîne vide, nous nous assurons que chaque fois que cette fonction est appelée, elle commence avec un état propre, évitant ainsi les erreurs et les comportements inattendus.

**Gestion btn tous**

1. **let btnAllProjects = document.createElement("button");**: Crée un nouvel élément **<button>** dans le DOM.
2. **btnAllProjects.className = "btn btnStyle";**: Définit les classes CSS de l'élément **<button>** pour appliquer le style souhaité.
3. **btnAllProjects.textContent = "Tous";**: Définit le texte à afficher à l'intérieur du bouton, qui est "Tous" dans ce cas.
4. **btnAllProjects.addEventListener("click", (event) => { changeState(event, works); });**: Ajoute un événements de clic sur le bouton, qui appelle la fonction changeState() lorsqu'il est cliqué.

**myDiv.appendChild(btnAllProjects);**: Ajoute le bouton nouvellement créé à l'intérieur de l'élément HTML spécifié par myDiv, qui est probablement un conteneur pour les boutons de catégorie.

1. **for (const categorie of categoriesData) {**: Boucle à travers chaque catégorie dans le tableau categoriesData.
2. **let button = document.createElement("button");**: Crée un nouvel élément <button> pour chaque catégorie.
3. **button.className = "btn btnStyle";**: Définit les classes CSS de l'élément <button> pour appliquer le style souhaité, similaire au premier bouton.
4. **button.textContent = categorie.name;**: Définit le texte à afficher à l'intérieur du bouton comme le nom de la catégorie actuelle.
5. **button.onclick = function (event) { changeState(event, works); };**: Ajoute un écouteur d'événements de clic sur chaque bouton de catégorie, appelant la fonction **changeState()** avec l'événement et les œuvres en tant que paramètres.
6. **myDiv.appendChild(button);**: Ajoute chaque bouton de catégorie nouvellement créé à l'intérieur de l'élément HTML spécifié par myDiv.

#### **Justifications :**

* **document.createElement()** est utilisé pour créer de nouveaux éléments HTML (boutons) dans le DOM, permettant la manipulation et l'ajout dynamiques d'éléments.
* L'attribution des classes CSS et du texte à chaque bouton permet de styliser et d'afficher correctement les boutons de catégorie dans l'interface utilisateur.
* L'ajout d'écouteurs d'événements de clic permet d'interagir avec les boutons de catégorie, ce qui déclenchera les actions appropriées lorsque l'utilisateur clique sur un bouton spécifique.
* L'ajout des boutons nouvellement créés à l'intérieur de l'élément HTML spécifié assure qu'ils sont correctement intégrés à l'interface utilisateur et visibles par l'utilisateur.

**Fonction changement de couleurs**

**const myButtons = document.getElementById("filter").children;**: Récupère tous les enfants de l'élément HTML avec l'ID "filter", qui sont probablement les boutons de filtre, et les stocke dans la variable **myButtons**.

1. **for (const myButton of myButtons) {**: Boucle à travers chaque bouton de filtre.
2. **myButton.style.background = "white";**: Définit la couleur de fond de chaque bouton de filtre sur blanc.
3. **myButton.style.color = "#1d6154";**: Définit la couleur du texte de chaque bouton de filtre sur une nuance de vert.
4. **const clickedElement = event.target;**: Récupère l'élément sur lequel l'événement de clic a été déclenché.
5. **clickedElement.style.background = "#1d6154";**: Définit la couleur de fond de l'élément cliqué sur une nuance de vert pour indiquer qu'il est activé.
6. **clickedElement.style.color = "white";**: Définit la couleur du texte de l'élément cliqué sur blanc pour améliorer la lisibilité.
7. **if (clickedElement.textContent == "Tous") {**: Vérifie si l'élément cliqué est le bouton "Tous".
8. **console.log(works);**: Affiche les données des œuvres dans la console du navigateur.
9. **createWorks(works);**: Affiche toutes les œuvres si le bouton "Tous" est cliqué en appelant la fonction **createWorks()**.
10. **filter(event.target.textContent, works);**: Filtre les œuvres en fonction du texte de l'élément cliqué (nom de la catégorie) en appelant la fonction **filter()**.

#### **Justifications :**

* La récupération des boutons de filtre permet de les cibler pour les manipulations de style et de comportement.
* La boucle **for-of** parcourt chaque bouton de filtre et définit son style pour réinitialiser leur apparence.
* L'élément cliqué est identifié à l'aide de **event.target**, ce qui permet de modifier son style pour le mettre en surbrillance et indiquer qu'il est actif.
* La condition **if...else** permet de différencier le comportement lorsque le bouton "Tous" est cliqué ou lorsqu'un autre bouton de filtre est cliqué.
* L'affichage des données des œuvres dans la console est utile pour le débogage et la vérification du bon fonctionnement de l'application.
* L'appel des fonctions **createWorks()** et **filter()** permet d'actualiser l'affichage des œuvres en fonction du bouton de filtre cliqué.

**Filtre des autres boutons**

**console.log(nameDuFiltre, works);**: Affiche dans la console le nom du filtre et les données des œuvres qui vont être filtrées. Cela permet de vérifier quel filtre est appliqué et quelles données d'œuvres sont concernées, ce qui facilite le débogage et la compréhension du processus de filtrage.

**var filterWork = [];**: Initialise un tableau vide **filterWork** qui va stocker les œuvres filtrées qui correspondent au filtre spécifié. Cela prépare l'espace de stockage pour les données filtrées.

1. **for (const work of works) {**: Boucle à travers chaque œuvre dans le tableau **works**.
2. **if (nameDuFiltre == work.category.name) {**: Vérifie si le nom de la catégorie de l'œuvre correspond au nom du filtre spécifié. Si c'est le cas, l'œuvre est éligible pour être incluse dans le filtrage.
3. **filterWork.push(work);**: Ajoute l'œuvre à **filterWork** si elle correspond au filtre spécifié.
4. **createWorks(filterWork);**: Appelle la fonction **createWorks()** avec le tableau d'œuvres filtrées en tant qu'argument. Cela met à jour l'affichage des œuvres dans la galerie en ne montrant que les œuvres qui correspondent au filtre spécifié.

#### **Justifications :**

* L'affichage des données du filtre et des œuvres dans la console facilite le suivi du processus de filtrage et aide au débogage en fournissant des informations utiles sur les données traitées.
* L'utilisation d'un tableau vide **filterWork** pour stocker les œuvres filtrées garantit que seules les œuvres qui correspondent au filtre spécifié seront incluses dans le résultat final.
* La boucle **for-of** parcourt chaque œuvre et vérifie si elle correspond au filtre spécifié en comparant le nom de la catégorie de chaque œuvre avec le nom du filtre.
* L'ajout des œuvres éligibles à **filterWork** assure que seules les œuvres correspondant au filtre seront incluses dans le tableau filtré.
* L'appel de la fonction **createWorks()** avec le tableau d'œuvres filtrées actualise l'affichage dans la galerie en affichant uniquement les œuvres qui correspondent au filtre spécifié.

1. **Modal.js**

**var modal = document.getElementById("myModal");**: Récupère l'élément HTML avec l'ID "myModal", qui représente la fenêtre modale dans le DOM. Cette ligne permet d'accéder à la fenêtre modale pour pouvoir la manipuler dans le script JavaScript.

**var btn = document.getElementById("modifier");**: Récupère le bouton qui sert à ouvrir la fenêtre modale. Cet élément est identifié par son ID "modifier". Cette ligne permet de cibler le bouton pour lui ajouter un gestionnaire d'événements qui ouvrira la fenêtre modale lorsque l'utilisateur clique dessus.

**var span = document.getElementsByClassName("close")[0];**: Récupère le bouton ou l'élément utilisé pour fermer la fenêtre modale. Il est généralement identifié par la classe "close". Cette ligne permet d'accéder à cet élément pour lui ajouter un gestionnaire d'événements qui fermera la fenêtre modale lorsque l'utilisateur clique dessus.

1. L'événement "**click"** est ajouté au bouton d'ouverture du modal (**btn**). Lorsque l'utilisateur clique sur ce bouton, la fonction anonyme est déclenchée. À l'intérieur de cette fonction, la propriété **style.display** de la fenêtre modale (**modal**) est réglée sur "block", ce qui la rend visible à l'utilisateur.
2. Ensuite, une sélection de tous les éléments avec la classe **"iconPoubelle**" est effectuée à l'aide de **document.querySelectorAll()**. Ces éléments représentent les icônes de poubelle à l'intérieur de la fenêtre modale, qui sont utilisées pour supprimer des éléments. Pour chaque élément sélectionné, un gestionnaire d'événements "click" est ajouté. Lorsque l'utilisateur clique sur l'une de ces icônes, une fonction est déclenchée, extrayant l'ID de l'élément cliqué (**e.target.id**), probablement pour identifier l'élément à supprimer ou effectuer d'autres actions en fonction de cet ID.

#### **Justifications :**

* La récupération de la fenêtre modale, du bouton d'ouverture et du bouton de fermeture permet de les cibler et de les manipuler dans le script JavaScript.
* L'ajout d'un gestionnaire d'événements "click" sur le bouton d'ouverture et sur le bouton de fermeture permet de définir le comportement de la fenêtre modale lorsque l'utilisateur interagit avec ces éléments.
* La modification de la propriété **style.display** de la fenêtre modale pour la rendre visible (**"block"**) lors de l'ouverture et invisible lors de la fermeture (**"none"**) est une pratique courante pour afficher ou masquer une fenêtre modale.
* L'ajout de gestionnaires d'événements aux éléments enfants de la fenêtre modale, tels que les icônes de poubelle, permet d'ajouter des fonctionnalités supplémentaires à la fenêtre modale, telles que la suppression d'éléments.

**Delete**

**fetch("http://localhost:5678/api/works/" + id, {**

**headers: {**

**Authorization: "Bearer " + token,**

**},**

**method: "DELETE",**

**}).then((response) => response.json());**

- Cette ligne envoie une requête HTTP DELETE à l'URL spécifiée, qui est composée de l'URL de base de l'API "http://localhost:5678/api/works/" concaténée avec l'ID de l'élément à supprimer.

- Les en-têtes de la requête sont configurés pour inclure le jeton d'authentification (Bearer token), qui permet d'authentifier l'utilisateur et d'autoriser la suppression de l'élément.

- La méthode de la requête est définie sur **"DELETE"** pour indiquer qu'il s'agit d'une action de suppression.

- La promesse renvoyée par la méthode `**fetch()`** est traitée à l'aide de `.then()`, où la réponse est transformée en JSON à l'aide de `response.json()` pour récupérer d'éventuelles données de réponse.

**init();**

- Après la suppression réussie de l'élément, la fonction `init()` est appelée pour réinitialiser ou mettre à jour l'affichage, probablement pour refléter les changements après la suppression de l'élément.

**span.onclick = function () { modal.style.display = "none"; };**

- Lorsque l'utilisateur clique sur l'élément HTML `<span>`, cette fonction est déclenchée. Elle modifie la propriété `display` de l'élément modal pour le masquer, en le rendant invisible à l'utilisateur.

**window.onclick = function (event) {**

**if (event.target == modal) {**

**modal.style.display = "none";**

**}**

**};**

- Lorsque l'utilisateur clique n'importe où sur la fenêtre, cette fonction est déclenchée. Elle vérifie si l'élément cliqué est égal à l'élément modal. Si tel est le cas, cela signifie que l'utilisateur a cliqué à l'extérieur de la fenêtre modale, donc la fenêtre modale est cachée en modifiant sa propriété `display` à "none".

#### **Justifications :**

* La requête DELETE est utilisée pour supprimer l'élément identifié par son ID à partir de l'API, avec une authentification à l'aide d'un jeton d'accès **(Bearer token)** pour s'assurer que l'utilisateur a les autorisations nécessaires pour effectuer cette action.
* Après la suppression réussie de l'élément, la fonction **init()** est appelée pour mettre à jour l'affichage, assurant ainsi que les changements sont reflétés à l'écran.
* Les gestionnaires d'événements pour la fermeture du modal lors du clic sur l'icône "x" et lors du clic en dehors de la fenêtre modale assurent une expérience utilisateur intuitive en permettant à l'utilisateur de fermer le modal de différentes manières.

Haut du formulaire

**Ouverture de la page modal ajouter photo**

**let modalAjouter = null;** cette ligne déclare une variable `**modalAjouter**` et **l'initialise à `null**`. Cette variable sera utilisée pour stocker une référence à l'élément modal lorsque celui-ci est ouvert.

**function ouvre\_modal\_ajoute(e) {**

**e.preventDefault();**  cette fonction est définie pour ouvrir le modal lorsqu'un événement est déclenché (probablement un clic sur un bouton ou un lien).

- La méthode `**preventDefault()`** est appelée sur l'objet événement `e` pour empêcher le comportement par défaut de l'événement, ce qui est souvent nécessaire lors de l'ouverture de modaux pour éviter que la page ne soit rechargée ou que le lien ne soit suivi.

**const model\_page = document.getElementById("modalAjouter"); model\_page.style.display = null; model\_page.removeAttribute("aria-hidden"); modalAjouter = model\_page;** cette séquence de lignes récupère l'élément modal à l'aide de son identifiant "**modalAjouter**" et le stocke dans la variable `**model\_page**`.

- La propriété `display` de l'élément modal est définie sur `**null**`, ce qui signifie que la valeur de style pour la propriété `display` est réinitialisée à sa valeur par défaut, qui dépend du style CSS de l'élément.

- L'attribut `**aria-hidden**` de l'élément modal est supprimé pour indiquer qu'il est désormais visible et accessible aux lecteurs d'écran.

- La référence à l'élément modal est ensuite assignée à la variable `**modalAjouter`** pour un accès ultérieur.

**modalAjouter.querySelector("modal-javascipt");** cette ligne semble être incorrecte ou inachevée. Il semble qu'elle tente de sélectionner un élément à l'intérieur du modal avec un sélecteur CSS, mais le sélecteur utilisé est "**modal-javascipt",** qui semble être une erreur de frappe ou une classe manquante.

**document.getElementById("ajouter").removeEventListener("click", handleAjouterClick);** cela supprime un gestionnaire d'événements associé à un élément avec l'identifiant "**ajouter"** pour l'événement **"click**". Le gestionnaire d'événements **`handleAjouterClick**` est supprimé afin d'éviter que plusieurs gestionnaires ne soient attachés à cet événement si la fonction `**ouvre\_modal\_ajoute`** est appelée plusieurs fois.

**document.getElementById("ajouter").addEventListener("click", handleAjouterClick);** cette ligne attache un nouveau gestionnaire d'événements pour **l'événement "click"** à l'élément avec l'identifiant "ajouter". Lorsque cet élément est cliqué, la fonction **`handleAjouterClick`** sera appelée pour gérer l'événement.

#### **Justifications :**

* La variable **modalAjouter** est utilisée pour stocker une référence à l'élément modal afin de pouvoir y accéder et le manipuler ultérieurement.
* La fonction **ouvre\_modal\_ajoute** est définie pour ouvrir le modal et effectuer les actions nécessaires lors de son ouverture, comme la suppression des gestionnaires d'événements existants et l'ajout d'un nouveau gestionnaire d'événements pour gérer le clic sur le bouton "ajouter".
* Les actions effectuées dans cette fonction garantissent que le modal est correctement affiché et que les interactions de l'utilisateur sont correctement gérées.

Haut du formulaire

**Ajoutez une fonction distincte pour gérer l'événement "ajouter"**

**function handleAjouterClick(event) { :** cette ligne définit une fonction nommée `**handleAjouterClick`,** qui prend un paramètre `**event**`. Cette fonction est destinée à être utilisée comme gestionnaire d'événements pour traiter les clics sur le bouton d'ajout.

. **event.preventDefault();** cette ligne appelle la méthode `**preventDefault()`** sur l'objet événement `**event`.** Cela empêche le comportement par défaut de l'événement, qui dans ce cas est généralement de soumettre un formulaire ou de suivre un lien, évitant ainsi que le comportement par défaut ne se produise.

**document.getElementById("img\_input").click();** cette ligne sélectionne l'élément de formulaire avec **l'ID "img\_input**", qui est probablement un input de type "file" pour **télécharger une image.**

En appelant la méthode `**click()`** sur cet élément, elle déclenche programmationnement un clic sur cet élément, ce qui ouvre la boîte de dialogue de sélection de fichier pour que l'utilisateur puisse choisir une image à télécharger.

**Justifications :**

- L'appel à `**event.preventDefault()`** est important pour éviter que le formulaire ne soit soumis ou que le comportement par défaut du lien ne soit déclenché, car dans ce cas, le comportement attendu est d'ouvrir la boîte de dialogue de sélection de fichier lors du clic sur le bouton.

- En appelant `**document.getElementById("img\_input").click()`,** l'action attendue est déclenchée de manière programmatique, permettant ainsi à l'utilisateur de télécharger une image en simulant un clic sur l'élément de formulaire approprié. Cela permet d'améliorer l'expérience utilisateur en automatisant le processus de sélection de fichier.

**Affichaque de categorie null**

**const category = document.getElementById("categorie");**

**category.value = null;** - Cette ligne récupère l'élément du formulaire avec **l'ID "categorie**" (probablement une liste déroulante ou un menu de sélection) et stocke une référence dans la variable `category`.

- Ensuite, elle définit la valeur de cet élément à `null`, probablement pour réinitialiser sa valeur lors de la création ou de la réinitialisation du formulaire.

**document.getElementById("btn\_valider").addEventListener("mouseover", mouseOver); -** Cette ligne attache un gestionnaire d'événements à l'élément de bouton avec l'ID "**btn\_valider**". L'événement est "**mouseover**", ce qui signifie qu'il sera déclenché lorsque le curseur de la souris passe sur le bouton.

- Le gestionnaire d'événements associé est la fonction `**mouseOver**`, qui sera appelée lorsque l'événement "mouseover" se produira sur le bouton.

**function mouseOver() { const mytitle = document.getElementById("input\_modal");**

**const mycat = document.getElementById("categorie"); const myImage = document.getElementById("img\_input")**

**if (mytitle.value != "" && mycat.value != "" && myImage.value != "") {**

**document.getElementById("btn\_valider").style.background = "#1d6154";**

**} else {**

**document.getElementById("btn\_valider").style.background = "#A7A7A7";**

**}**

**}**

- Cette fonction `**mouseOver**` est appelée lorsqu'un événement "**mouseover"** est déclenché sur le bouton de validation.

- À l'intérieur de cette fonction, les valeurs des champs de titre, de catégorie et d'image sont récupérées.

- Ensuite, elle vérifie si les champs de titre, de catégorie et d'image ne sont pas vides. Si c'est le cas, elle change la couleur de fond du bouton de validation en vert (`#1d6154`), sinon, elle le met en gris clair (`#A7A7A7`).

4.

**document.getElementById("btn\_valider").addEventListener("click", function (event) {**

**event.preventDefault();**

- Cette ligne ajoute un gestionnaire d'événements pour l'événement "click" sur le bouton de validation. Lorsque le bouton est cliqué, la fonction anonyme passée comme deuxième argument sera exécutée.

- La première instruction de cette fonction est `**event.preventDefault()**`, qui empêche le comportement par défaut du bouton, ce qui est souvent nécessaire dans les formulaires pour éviter que la page ne se recharge lors de la soumission.

**const mytitle = document.getElementById("input\_modal"); const mycat = document.getElementById("categorie"); const myImage = document.getElementById("img\_input");**

- Ces lignes récupèrent les valeurs des champs de titre, de catégorie et d'image et les stockent dans les variables `**mytitle`, `mycat` et `myImage**` respectivement.

**const errTile = document.getElementById("errorTitle");**

**const errCat = document.getElementById("errorCategory");**

**const errImg = document.getElementById("errorImg");**

**errImg.style.display = "none";**

**errTile.style.display = "none";**

**errCat.style.display = "none";**

- Ces lignes récupèrent les éléments d'affichage des erreurs associées aux champs de titre, de catégorie et d'image et les stockent dans les variables correspondantes.

- Ensuite, elles masquent ces éléments en définissant leur style `display` sur "none", afin qu'ils ne soient pas visibles initialement.

**const data = new FormData(document.getElementById("modalAjouter"));**

- Cette ligne crée un nouvel objet `FormData` en utilisant le formulaire avec **l'ID "modalAjouter**". Cet objet représente les données du formulaire sous forme de paires clé-valeur.

**Justifications :**

- La réinitialisation de la valeur de la catégorie à `**null**` assure que le champ de catégorie est vide lors de l'initialisation du formulaire ou lors de sa réinitialisation.

- Les gestionnaires d'événements **`mouseover**` et `click` sur le bouton de validation **(`btn\_valider`)** améliorent l'interaction utilisateur en ajustant visuellement le bouton en fonction de l'état de remplissage du formulaire.

- La récupération des valeurs des champs de formulaire permet de vérifier leur contenu pour valider le formulaire avant de le soumettre.

- La gestion des éléments d'affichage des erreurs assure qu'ils sont masqués au départ et ne sont affichés que si nécessaire lors de la validation du formulaire.

**fetch works message d'erreur quand un des éléments est vide (image, titre, categorie)**

**if (myImage.value == "") {**

**errImg.style.display = "block";**

**}**

- Cette condition vérifie si le champ d'image est vide. Si c'est le cas, elle affiche l'élément d'affichage des erreurs associé à l'image (`errImg`) en définissant son style `display` sur "block". Cela signifie que le message d'erreur sera affiché à l'utilisateur.

**if (mytitle.value == "" && mycat.value != "") {**

**errTile.style.display = "block";**

**}**

- Cette condition vérifie si le champ de titre est vide et si le champ de catégorie n'est pas vide. Si c'est le cas, elle affiche l'élément d'affichage des erreurs associé au titre (`errTile`) en définissant son style `display` sur "block".

- Cette condition vérifie si le champ de titre n'est pas vide et si le champ de catégorie est vide. Si c'est le cas, elle affiche l'élément d'affichage des erreurs associé à la catégorie (`errCat`) en définissant son style `display` sur "block".

**if (mytitle.value == "" && mycat.value == "") {**

**errTile.style.display = "block";**

**errCat.style.display = "block";**

**}**

Cette condition vérifie si à la fois le champ de titre et le champ de catégorie sont vides. Si c'est le cas, elle affiche à la fois les éléments d'affichage des erreurs associés au titre et à la catégorie **(`errTile` et `errCat`)** en définissant leur style `display` sur "block".

#### **Justifications :**

* La validation des champs assure que les données soumises par l'utilisateur sont valides avant d'être envoyées au serveur, améliorant ainsi l'expérience utilisateur.
* L'affichage des messages d'erreur permet à l'utilisateur de comprendre ce qui doit être corrigé dans le formulaire pour qu'il soit soumis avec succès.
* La définition de la propriété **display** sur "block" pour les éléments d'affichage des erreurs assure qu'ils deviennent visibles pour l'utilisateur lorsque des erreurs sont détectées dans le formulaire.

**Quand aucun un element n’est vide**

**if (mytitle.value != "" && mycat.value != "" && myImage.value != "") { :** cette condition vérifie si les champs de titre **(`mytitle`),** de catégorie **(`mycat`),** et d'image **(`myImage`)** sont tous remplis. Si c'est le cas, cela signifie que toutes les données requises ont été saisies dans le formulaire.

**fetch("http://localhost:5678/api/works", {**

**headers: {**

**Authorization: "Bearer " + token,**

**},**

**method: "POST",**

**body: data,**

**}).then((response) => response.json());** cette ligne effectue une requête POST vers l'URL spécifiée pour créer une nouvelle ressource (dans ce cas, un nouvel élément de travail). Les données du formulaire sont incluses dans le corps de la requête.

La clé d'autorisation (`Authorization`) est incluse dans les en-têtes pour l'authentification, généralement avec un jeton d'accès (`token`).

**suprimer();** cette ligne appelle une fonction nommée **suprimer()**, qui semble être une faute de frappe pour la fonction **supprimer()**. Cette fonction pourrait supprimer des éléments ou effectuer d'autres actions liées à la création de la nouvelle ressource.

**init();** cette ligne appelle la fonction `init()`, qui semble être utilisée pour initialiser ou rafraîchir une partie de l'interface utilisateur après l'ajout d'un nouvel élément de travail.

**close\_modal\_ajouter();** cette ligne appelle la fonction **close\_modal\_ajouter()**, qui est probablement utilisée pour fermer le modal d'ajout après que l'opération d'ajout soit terminée avec succès.

**modal.style.display = "none";** cette ligne définit le style `display` du modal sur "none", le cachant de la vue de l'utilisateur.

**document.getElementById("ajouterModal").addEventListener("click", ouvre\_modal\_ajoute);** cette ligne ajoute un écouteur d'événements au clic sur un élément **avec l'ID "ajouterModal".** Lorsque cet élément est cliqué, il appelle la fonction `**ouvre\_modal\_ajoute()`,** qui semble ouvrir un modal d'ajout.

**Justifications :**

- La condition initiale garantit que le formulaire n'est soumis que si toutes les données requises sont présentes, assurant ainsi l'intégrité des données envoyées au serveur.

- La requête POST envoie les données du formulaire au serveur pour être traitées, suivant ainsi le principe de l'architecture RESTful.

- L'appel aux fonctions **`supprimer()`, `init()`, et `close\_modal\_ajouter()`** semble être destiné à effectuer des actions supplémentaires après la soumission réussie du formulaire, telles que la mise à jour de l'interface utilisateur et la gestion des modaux.

- L'ajout d'un écouteur d'événements garantit que la fonction `**ouvre\_modal\_ajoute()`** est appelée lorsque l'utilisateur clique sur l'élément associé, permettant ainsi d'ouvrir le modal d'ajout.

- Cette ligne ajoute un écouteur d'événements au clic sur un élément avec l'ID **"ajouterModal".** Lorsque cet élément est cliqué, il appelle la fonction `**ouvre\_modal\_ajoute()`,** qui semble ouvrir un modal d'ajout.

**Justifications :**

- La condition initiale garantit que le formulaire n'est soumis que si toutes les données requises sont présentes, assurant ainsi l'intégrité des données envoyées au serveur.

- La requête POST envoie les données du formulaire au serveur pour être traitées, suivant ainsi le principe de l'architecture RESTful.

- L'appel aux fonctions `**supprimer()`, `init()`,** et `**close\_modal\_ajouter()`** semble être destiné à effectuer des actions supplémentaires après la soumission réussie du formulaire, telles que la mise à jour de l'interface utilisateur et la gestion des modaux.

- L'ajout d'un écouteur d'événements garantit que la fonction `**ouvre\_modal\_ajoute()`** est appelée lorsque l'utilisateur clique sur l'élément associé, permettant ainsi d'ouvrir le modal d'ajout.

**Fermer modal ajout photo**

**function fermer\_modal\_ajoute(e) {**

Cette ligne déclare une fonction nommée **fermer\_modal\_ajoute** qui prend un événement **e** en paramètre. Cette fonction est appelée lorsque l'utilisateur souhaite fermer le modal d'ajout.

**e.preventDefault(); :** cette ligne utilise la méthode **preventDefault()** pour empêcher le comportement par défaut de l'événement. Dans ce cas, cela empêche le navigateur de recharger la page ou de suivre un lien lorsqu'un événement de clic se produit.

**close\_modal\_ajouter();:** cette ligne appelle la fonction **close\_modal\_ajouter()**, qui est responsable de la fermeture effective du modal d'ajout.

**function close\_modal\_ajouter() { :** cette ligne déclare une fonction nommée **close\_modal\_ajouter()**. Cette fonction est chargée de cacher le modal d'ajout et de modifier un attribut pour indiquer que le modal est masqué.

**const modalAjouter = document.getElementById("modalAjouter");:** cette ligne récupère l'élément HTML du modal d'ajout en utilisant son ID "modalAjouter" et le stocke dans une variable appelée **modalAjouter**.

**modalAjouter.style.display = "none";:** cette ligne définit le style CSS de l'élément **modalAjouter** pour le cacher en utilisant **display: none**. Cela rend le modal invisible sur la page.

**modalAjouter.setAttribute("aria-hidden", "true");:** cette ligne ajoute un attribut **aria-hidden** à l'élément **modalAjouter** avec la valeur "true". Cet attribut est utilisé pour indiquer aux technologies d'assistance que le modal est masqué de manière à ce qu'elles ne le présentent **pas à l'utilisateur.**

**document.getElementById("modal\_close").addEventListener("click", fermer\_modal\_ajoute);:** cette ligne ajoute un écouteur d'événements de clic à l'élément avec l'ID "modal\_close". Lorsque cet élément est cliqué, la fonction **fermer\_modal\_ajoute** est appelée, ce qui permet de fermer le modal d'ajout.

**Justifications :**

* La fonction **fermer\_modal\_ajoute** est définie pour gérer l'action de fermeture du modal d'ajout de manière contrôlée.
* L'utilisation de **e.preventDefault()** assure que l'événement de clic sur le bouton de fermeture du modal n'entraîne pas de comportement par défaut, ce qui peut éviter des interactions non désirées avec la page.
* La fonction **close\_modal\_ajouter** encapsule la logique de fermeture du modal, améliorant ainsi la lisibilité du code et facilitant sa réutilisation.
* L'ajout de l'attribut **aria-hidden** assure une meilleure accessibilité du modal pour les utilisateurs ayant des besoins spécifiques en matière d'accessibilité.
* L'écouteur d'événements de clic sur le bouton de fermeture du modal permet une fermeture cohérente et prévisible du modal pour l'utilisateur.

**La flèche de retour**

**document.getElementById("left").addEventListener("click", fermer\_modal\_ajoute);**

* **document.getElementById("left")** : cette partie de la ligne de code récupère l'élément du DOM avec **l'ID "left**". Cet élément représente probablement un élément interactif, tel qu'une flèche ou un bouton, qui permet de fermer le modal.
* **.addEventListener("click", fermer\_modal\_ajoute)** : cette méthode ajoute un écouteur d'événements à l'élément récupéré. L'écouteur est configuré pour détecter l'événement de clic sur cet élément. Lorsque l'événement de clic se produit, la fonction **fermer\_modal\_ajoute** est appelée.

**Justification :**

* Cette ligne de code assure une expérience utilisateur cohérente et intuitive en permettant à l'utilisateur de fermer le modal lorsqu'il clique sur l'élément identifié par l**'ID "left**". Cela améliore l'utilisabilité de l'interface utilisateur en offrant une autre option de fermeture du modal, en plus des boutons de fermeture habituels.
* L'ajout de cet événement de clic sur l'élément **"left**" assure une accessibilité accrue pour l'utilisateur, car il offre une alternative supplémentaire pour fermer le modal, ce qui peut être utile pour les utilisateurs ayant des besoins spécifiques en matière d'accessibilité ou des préférences d'interaction différentes.
* La fonction **fermer\_modal\_ajoute** qui est appelée lors du clic sur l'élément "**left"** assure une gestion cohérente de la fermeture du modal, garantissant ainsi une expérience utilisateur fluide et prévisible.

**Télécharger les photos**

Cette portion de code concerne le téléchargement d'une image lorsque l'utilisateur sélectionne un fichier à partir de l'élément d'entrée de type fichier (**input**) dans le formulaire.

1. **const input = document.getElementById("img\_input");** : Cette ligne récupère l'élément HTML **<input>** avec l'ID "img\_input" à partir du document HTML et le stocke dans la variable **input**. Cet élément est utilisé pour sélectionner des fichiers.
2. **var telecharge\_image = "";** : Déclare une variable **telecharge\_image** initialisée à une chaîne vide. Cette variable stockera les données de l'image téléchargée.
3. **const reader = new FileReader();** : Crée un nouvel objet FileReader qui permet à l'application web de lire le contenu des fichiers (ou des blobs).
4. **reader.addEventListener("load", () => { ... })** : Ajoute un écouteur d'événements pour l'événement "load" du FileReader, qui est déclenché lorsque la lecture du fichier est terminée. Lorsque cet événement se produit, la fonction fléchée associée est exécutée.
5. **telecharge\_image = reader.result;** : La propriété **result** de l'objet FileReader contient les données binaires du fichier sous forme d'URL de données. Ces données sont attribuées à la variable **telecharge\_image**.
6. **const photo = document.getElementById("telecharger\_image");** : Récupère l'élément HTML avec l'ID "telecharger\_image" et le stocke dans la variable **photo**. Cet élément est probablement un conteneur où l'image téléchargée sera affichée.
7. **document.getElementById("telechargerImage").style.display = null;** : Modifie la propriété CSS **display** de l'élément avec l'ID "telechargerImage" pour afficher cet élément. Cela peut être utilisé pour afficher un élément qui était précédemment caché.
8. **photo.style.backgroundImage =** url(${telecharge\_image} )**;** : Définit la propriété **backgroundImage** de l'élément **photo** avec l'URL de données de l'image téléchargée, ce qui affiche l'image téléchargée comme arrière-plan de cet élément.
9. **document.getElementById("container\_modalAjouter").style.display = "none";** : Cache l'élément avec l'ID "container\_modalAjouter" en modifiant sa propriété CSS **display** pour la définir sur "none", ce qui le rend invisible.
10. **reader.readAsDataURL(this.files[0]);** : Démarre la lecture du contenu du fichier sélectionné par l'utilisateur à partir de l'élément **<input>** de type fichier. Le contenu du fichier sera lu sous forme d'URL de données.
11. **document.getElementById("img\_input").addEventListener("change", telecharger);** : Ajoute un écouteur d'événements sur l'élément avec l'ID "img\_input" pour détecter tout changement (comme la sélection d'un fichier). Lorsque l'événement "change" se produit, la fonction **telecharger** est appelée pour traiter le fichier sélectionné.

**Justification :**

Ce code permet à l'utilisateur de sélectionner un fichier image à partir de son système de fichiers local via l'élément **<input>** de type fichier. Lorsque l'utilisateur sélectionne un fichier, l'image est affichée dans un conteneur spécifique sur la page web. Ce processus améliore l'expérience utilisateur en permettant le téléchargement et l'affichage d'images sur la page sans avoir à recharger la page.

**Api**

Ce code est une requête HTTP qui récupère des données de catégories à partir d'une API située à l'adresse "<http://localhost:5678/api/categories>". Une fois les données récupérées, elles sont utilisées pour créer des options dans un élément de liste déroulante (**<select>**).

1. **fetch("http://localhost:5678/api/categories")** : Cette ligne envoie une requête GET à l'URL spécifiée pour récupérer les données des catégories à partir de l'API.
2. **.then((response) => response.json())** : Une fois la réponse reçue du serveur, cette ligne convertit la réponse en format JSON. La méthode **json()** renvoie une promesse qui résout avec les données JSON de la réponse.
3. **.then((data) => { ... })** : Une fois que les données JSON sont extraites de la réponse, cette ligne définit une fonction qui sera exécutée lorsque la promesse précédente est résolue. Les données JSON sont passées en tant que paramètre de cette fonction.
4. **const categorieData = data;** : Les données JSON récupérées sont stockées dans une variable locale appelée **categorieData** pour faciliter leur manipulation ultérieure.
5. **const mySelect = document.getElementById("categorie");** : Récupère l'élément HTML avec l'ID "categorie", qui est probablement un élément **<select>** dans le document HTML, et le stocke dans la variable **mySelect**.
6. **for (i = 0; i < categorieData.length; i++) { ... }** : Cette boucle itère à travers toutes les catégories extraites des données JSON.
7. **const catName = categorieData[i].name;** : Pour chaque catégorie, cette ligne extrait le nom de la catégorie à partir de l'objet de catégorie et le stocke dans la variable **catName**.
8. **const optionItem = document.createElement("option");** : Crée un nouvel élément d'option (**<option>**) qui représente une option dans la liste déroulante.
9. **optionItem.textContent = catName;** : Définit le texte de contenu de l'option nouvellement créée comme étant le nom de la catégorie extrait précédemment.
10. **optionItem.value = categorieData[i].id;** : Définit la valeur de l'attribut **value** de l'option nouvellement créée comme étant l'identifiant de la catégorie, généralement utilisé pour identifier la catégorie sélectionnée lors de la soumission du formulaire.
11. **mySelect.appendChild(optionItem);** : Ajoute l'option nouvellement créée à l'élément de liste déroulante (**<select>**) spécifié.

**Justification :**

Ce code est utilisé pour dynamiquement peupler une liste déroulante HTML avec les catégories disponibles récupérées à partir d'une API. Cela permet à l'utilisateur de sélectionner une catégorie lorsqu'il interagit avec le formulaire sur la page web. L'utilisation de JavaScript pour effectuer cette opération côté client améliore l'expérience utilisateur en évitant le rechargement de la page pour obtenir les données des catégories à chaque fois que le formulaire est affiché

**Fontion supprime**

**document.getElementById("container\_modalAjouter").style.display = null;:** cette ligne modifie le style CSS de l'élément HTML ayant l'ID "container\_modalAjouter" en définissant sa propriété **display** à **null**. Cela signifie que le style **display** sera réinitialisé à sa valeur par défaut définie dans la feuille de style ou dans le balisage HTML lui-même.

**document.getElementById("telechargerImage").style.display = "none";:** cette ligne modifie le style CSS de l'élément HTML ayant l'ID "telechargerImage" en définissant sa propriété **display** à "none". Cela rend l'élément invisible et le retire de la mise en page, mais l'espace qu'il occupe est toujours réservé dans le flux du document.

**const input\_titre\_ajout = document.getElementById("input\_modal");**

**input\_titre\_ajout.value = null;**

Ces lignes récupèrent l'élément de saisie de titre du formulaire avec **l'ID "input\_modal",** puis elles réinitialisent la valeur de cet élément en la définissant sur **null**. Cela efface le texte saisi dans le champ de saisie du titre.

**const input\_photo\_url = document.getElementById("img\_input");**

**input\_photo\_url.value = null;**

Ces lignes récupèrent l'élément de saisie de l'URL de l'image du formulaire avec **l'ID "img\_input**", puis elles réinitialisent la valeur de cet élément en la définissant sur **null**. Cela efface le contenu de la zone de saisie de l'URL de l'image.

**const category = document.getElementById("categorie");**

**category.value = null;**

Ces lignes récupèrent l'élément de sélection de catégorie du formulaire avec **l'ID "categorie**", puis elles réinitialisent la valeur de cet élément en la définissant sur **null**. Cela réinitialise la sélection de catégorie à sa valeur par défaut, ce qui signifie qu'aucune catégorie n'est sélectionnée après l'exécution de cette fonction.

**Justification :**

Cette fonction est appelée pour effacer les données saisies dans le formulaire de la modal d'ajout lorsque l'utilisateur ferme cette modal. Cela garantit que le formulaire est vide et prêt à être rempli lors de la prochaine ouverture de la modal, améliorant ainsi l'expérience utilisateur en offrant un formulaire propre et clair à chaque interaction.

**Haut du formulaire**

**admin.js**

**let token = localStorage.getItem("token");** cette ligne récupère la valeur du token stocké dans le stockage local du navigateur sous la clé "token" et l'assigne à la variable **token**.

**if (token) {** cette ligne commence une structure conditionnelle qui vérifie si la variable **token** contient une valeur (c'est-à-dire si un token est présent dans le stockage local).

**document.querySelector(".bando").classList.remove("hidden");** cette ligne sélectionne l'élément ayant la classe CSS "bando" et lui retire la classe "hidden", ce qui permet d'afficher cet élément. Cela peut être utilisé pour afficher une bannière ou un message spécifique **lorsqu'un utilisateur est connecté.**

**document.getElementById("modifier").style.display = "flex";** cette ligne modifie le style CSS de l'élément HTML avec **l'ID "modifier**" pour le rendre visible en le définissant sur "flex". Cela peut être utilisé pour afficher un bouton de modification spécifique lorsque l'utilisateur est connecté.

**document.getElementById("modifiers").style.display = "flex";** cette ligne modifie le style CSS de l'élément HTML avec l'ID "modifiers" pour le rendre visible en le définissant sur "flex". Cela peut être utilisé pour afficher un autre bouton de modification spécifique lorsque l'utilisateur est connecté.

**document.getElementById("filter").style.display = "none";** cette ligne modifie le style CSS de l'élément HTML avec l'ID "filter" pour le rendre invisible en le définissant sur "none". Cela peut être utilisé pour cacher des éléments de filtrage ou de navigation spécifiques lorsque l'utilisateur est connecté.

**document.getElementById("login").innerText = "logout";** cette ligne modifie le texte à l'intérieur de l'élément HTML avec **l'ID "login**" pour le remplacer par "**logout**", ce qui peut être utilisé pour changer le texte du bouton de connexion en bouton de déconnexion lorsque l'utilisateur est connecté.

**document.getElementById("login").addEventListener("click", function (event) { :** cette ligne ajoute un écouteur d'événements au clic sur l'élément HTML avec l'ID "login", ce qui déclenche une fonction lorsque l'utilisateur clique sur ce bouton.

**event.preventDefault();** cette ligne empêche le comportement par défaut de l'événement, ce qui évite que la page ne se recharge lorsqu'on clique sur le bouton.

**localStorage.removeItem("token");** cette ligne supprime le token du stockage local lorsque l'utilisateur se déconnecte.

**document.getElementById("login").innerText = "login";** cette ligne rétablit le texte du bouton de connexion à "login" après que l'utilisateur s'est déconnecté.

**document.querySelector(".bando").classList.add("hidden");** cette ligne ajoute la classe **"hidden"** à l'élément ayant la classe **CSS "bando**", ce qui permet de le cacher à nouveau lorsque l'utilisateur se déconnecte.

**document.getElementById("modifier").style.display = "none";** cette ligne masque l'élément avec **l'ID "modifier"** en définissant son style sur "none" après la déconnexion de l'utilisateur.

**document.getElementById("modifiers").style.display = "none";** cette ligne masque l'élément avec l'ID "modifiers" en définissant son style sur "none" après la déconnexion de l'utilisateur.

**document.getElementById("filter").style.display = "flex";** cette ligne affiche à nouveau l'élément avec **l'ID "filter**" en définissant son style sur "flex" après que l'utilisateur se soit déconnecté.

**Justification :**

Ce bloc de code est utilisé pour contrôler l'affichage des éléments et le comportement des boutons en fonction de l'état de connexion de l'utilisateur. Lorsqu'un token est présent dans le stockage local, cela signifie que l'utilisateur est connecté, donc le code ajuste l'interface utilisateur en conséquence en affichant certains éléments (comme une bannière, des boutons de modification, etc.) et en masquant d'autres (comme les boutons de filtrage). Lorsque l'utilisateur se déconnecte, le code rétablit l'interface utilisateur à son état initial en cachant les éléments associés à la connexion et en affichant ceux associés à la déconnexion. Cela améliore l'expérience utilisateur en offrant une interface utilisateur cohérente et contextuellement appropriée.

**Haut du formulaire**

**login.js**

**console.log("link good!");** cette ligne affiche un message "link good!" dans la console du navigateur. Cela peut être utilisé pour déboguer et vérifier si le script est correctement lié à la page HTML.

**const form = document.getElementById("connexion-form");** cette ligne sélectionne l'élément HTML avec l'ID "connexion-form" et l'assigne à la variable **form**. Cet élément représente le formulaire de connexion de l'utilisateur.

**const valid = document.getElementById("submit1");** cette ligne sélectionne l'élément HTML avec l'ID "submit1" et l'assigne à la variable **valid**. Cet élément est généralement un bouton "Valider" ou "Se connecter" du formulaire.

**const email = document.getElementById("email");** cette ligne sélectionne l'élément HTML avec l'ID "email" et l'assigne à la variable **email**. Cet élément représente le champ de saisie de l'adresse e-mail dans le formulaire.

**const password = document.getElementById("password");** cette ligne sélectionne l'élément HTML avec l'ID "password" et l'assigne à la variable **password**. Cet élément représente le champ de saisie du mot de passe dans le formulaire.

**const errorMsg = document.getElementById("errorMsg");** cette ligne sélectionne l'élément HTML avec l'ID "errorMsg" et l'assigne à la variable **errorMsg**. Cet élément est généralement utilisé pour afficher les messages d'erreur lors de la tentative de connexion.

**form.addEventListener("submit", function (e) { :** cette ligne ajoute un écouteur d'événements sur l'événement "submit" du formulaire. Lorsque le formulaire est soumis, la fonction anonyme associée est exécutée.

**e.preventDefault();** cette ligne empêche le comportement par défaut du formulaire lors de la soumission, ce qui évite que la page ne se recharge.

**const formInfo = new FormData(form);**

**const payload = new URLSearchParams(formInfo);**

Ces lignes créent un objet **FormData** à partir des données du formulaire (**formInfo**) et le convertissent en une chaîne de requête de type URL (**payload**). Cette chaîne de requête est utilisée comme corps de la requête dans la requête **fetch**.

**fetch("http://localhost:5678/api/users/login", {**

**method: "POST",**

**headers: {**

**accept: "application/json",**

**},**

**body: payload,**

**})**

Cette ligne envoie une requête POST à l'URL "<http://localhost:5678/api/users/login>" avec les options spécifiées : méthode POST, en-tête "accept" indiquant que le client peut accepter les données au format JSON, et le corps de la requête contenant les données du formulaire encodées en URL.

**.then((res) => res.json()) :** cette ligne analyse la réponse de la requête en tant que JSON lorsque la promesse est résolue.

**.then((data) => {**

**console.log(data);** cette ligne affiche les données de réponse dans la console du navigateur. Cela peut être utilisé pour vérifier les données renvoyées par le serveur, telles que les détails de l'utilisateur ou les messages d'erreur.

**if (data.userId == 1) { :** cette ligne vérifie si l'identifiant de l'utilisateur dans les données de réponse est égal à 1, ce qui indique généralement une connexion réussie pour un utilisateur spécifique.

**localStorage.setItem("token", data.token);** cette ligne stocke le token d'authentification renvoyé par le serveur dans le stockage local du navigateur. Ce token est généralement utilisé pour authentifier les requêtes ultérieures à **l'API.**

**location.href = "index.html";** cette ligne redirige l'utilisateur vers la page "index.html" après une connexion réussie.

**errorMsg.innerText =** " Erreur dans l’identifiant ou le mot de passe";

**document.getElementById("email").value = null;**

**document.getElementById("password").value = null;**

Ces lignes affichent un message d'erreur spécifique dans l'élément avec l'ID "errorMsg" et réinitialisent les champs d'e-mail et de mot de passe du formulaire.

**function msgdelet() {**

**error.innerText = "";**

**}**

**setTimeout(msgdelet, 50000);**

Ces lignes définissent une fonction **msgdelet()** qui efface le contenu de l'élément d'erreur après un délai de 50000 millisecondes (50 secondes).

**.catch((err) => console.log(err));** cette ligne capture les erreurs potentielles survenues lors de l'envoi de la requête ou de l'analyse de la réponse et les affiche dans la console du navigateur.

**Justification :**

Ce bloc de code gère le processus d'envoi des informations de connexion au serveur via une requête POST. Il récupère les données du formulaire, les envoie au serveur, traite la réponse et agit en conséquence. Par exemple, s'il reçoit une réponse indiquant que l'identification est réussie, il stocke le token d'authentification dans le stockage local et redirige l'utilisateur vers une autre page. En cas d'échec de l'identification, il affiche un message d'erreur et efface les champs d'entrée du formulaire. En résumé, ce code améliore l'expérience utilisateur en gérant la connexion de manière interactive et informative.

Haut du formulaire