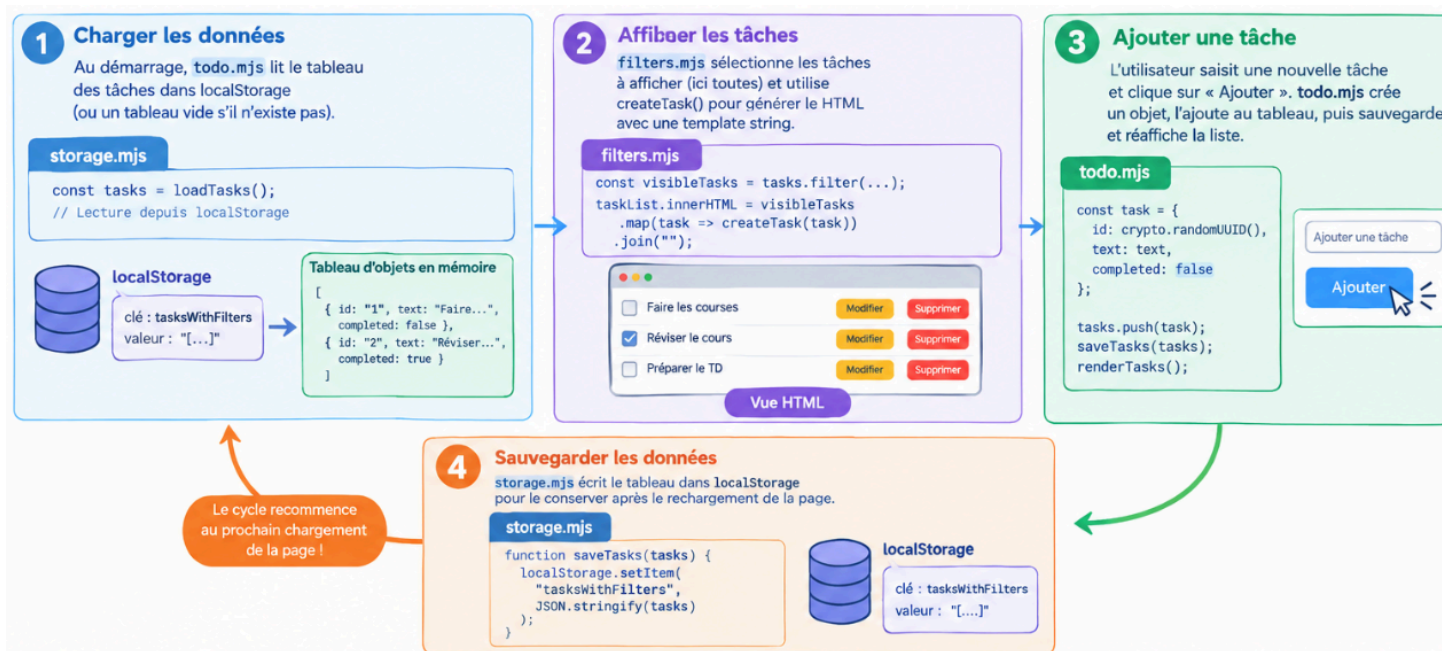


Prérequis :  Créer une liste de tâches

Développer une liste de tâches avec JavaScript et plusieurs modules

L'idée générale est de lire les données dans localStorage **1**

→ **2** les afficher → **3** Ajouter une tâche → **4** sauvegarder



 git : [code](#)

 En action

Les fichiers par types :



index.html : Structure de la page (formulaire, boutons, liste)



todo.mjs : gère les événements (ajout), coordonne les modules, Manipule le tableau tasks

tasks.mjs : génère le HTML d'une tâche avec un template String

filters.mjs : Sélectionne les tâche à afficher, utilise filter() et map()

Storage.mjs : lit et écrit dans localStorage au format JSON

🚀 en action :

<https://dupontdenis.github.io/newTODO/filter/transformationDonn%C3%A9es/index.html>

Ma liste de tâches

Nouvelle tâche

Ajouter une tâche

Ajouter

Toutes

Incomplètes

Complètes

☐ apprendre JS	Modifier	Supprimer
☒ Apprendre js	Modifier	Supprimer

Dans cette application, il faut distinguer deux opérations :

1. **Filtrer** les données : choisir, dans le tableau `tasks`, les tâches qui correspondent au filtre sélectionné par l'utilisateur. Cette opération est réalisée avec la méthode `filter()`. Elle travaille sur les objets JavaScript et ne génère pas de HTML.
2. **Générer** le HTML : transformer chaque tâche retenue en une chaîne HTML, grâce à la fonction `createTask()` et à une template string. Cette opération est réalisée avec `map()`. Les chaînes obtenues sont ensuite réunies avec `join("")` et insérées dans la page avec `innerHTML`.

🔍 Par exemple, si le tableau contient cinq tâches, dont deux sont terminées, le filtre "Complètes" sélectionne d'abord les deux objets correspondants. Le programme génère ensuite le HTML de ces deux tâches et les affiche dans la page.

Ainsi, `filter()` détermine quelles tâches afficher, tandis que `map()` détermine comment chaque tâche sera représentée en HTML. 🚀 Ces deux opérations sont complémentaires, mais elles ont des rôles différents.

👑 Objectifs du cours

Nous allons réaliser une application qui permet de gérer une liste de tâches dans le navigateur.

L'application doit permettre de :

- ajouter une tâche ;
- afficher toutes les tâches ;
- filtrer les tâches selon leur état : toutes, complètes ou incomplètes ;
- marquer une tâche comme terminée ;
- supprimer une tâche ;

- conserver les tâches après le rechargement de la page.

Le code sera réparti dans plusieurs fichiers JavaScript.

Nous utiliserons les modules ES (import et export), les objets, les tableaux, les fonctions fléchées, les méthodes `filter()` et `map()`, les chaînes de gabarit (template strings), les événements et `localStorage`.

Organisation du projet

todo/

├─ index.html

├─ todo.mjs

├─ tasks.mjs

├─ filters.mjs

└─ storage.mjs

Chaque fichier possède une responsabilité particulière. Cette séparation permet de rendre le code plus lisible et plus facile à faire évoluer.

1. La page HTML : index.html

La page HTML définit la structure de l'application. Elle contient le formulaire d'ajout, les boutons de filtrage et la liste des tâches.

 index.html

1. `<!DOCTYPE html>`
2. `<html lang="fr">`
- 3.

```
4. <head>
5.   <meta charset="UTF-8">
6.   <meta name="viewport" content="width=device-width, initial-scale=1">
7.
8.   <link
9.
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.
      css"
10.     rel="stylesheet"
11.   >
12.
13.   <title>Ma liste de tâches</title>
14.
15.   <script type="module" src="./todo.mjs"></script>
16.</head>
17.
18.<body>
19.
20.   <div class="container">
21.     <h1 class="mt-5 mb-4">Ma liste de tâches</h1>
22.
23.     <form class="mb-3">
24.       <div class="input-group mb-3">
25.         <label class="input-group-text" for="task">
26.           Nouvelle tâche
27.         </label>
28.
29.         <input
```

```
30.         id="task"
31.         class="form-control"
32.         type="text"
33.         autocomplete="off"
34.         placeholder="Ajouter une tâche"
35.     >
36. </div>
37.
38. <button type="submit" class="btn btn-primary">
39.     Ajouter
40. </button>
41. </form>
42.
43. <button
44.     class="btn btn-primary filter-btn"
45.     data-filter="all"
46. >
47.     Toutes
48. </button>
49.
50. <button
51.     class="btn btn-outline-primary filter-btn"
52.     data-filter="incomplete"
53. >
54.     Incomplètes
55. </button>
56.
57. <button
```

```
58.     class="btn btn-outline-primary filter-btn"
59.     data-filter="complete"
60. >
61.     Complètes
62. </button>
63.
64. <ol id="tasks" class="list-group mt-3"></ol>
65. </div>
66.
67.</body>
68.</html>
```

1.1. Le rôle des éléments HTML

Le formulaire permet de saisir une nouvelle tâche. Le champ de saisie possède l'identifiant task, ce qui permet au JavaScript de retrouver cet élément.

Les trois boutons de filtrage possèdent une classe commune, filter-btn, et un attribut data-filter.

Par exemple :

```
<button class="filter-btn" data-filter="incomplete">
  Incomplètes
</button>
```

L'attribut data-filter contient la valeur qui sera utilisée par JavaScript pour déterminer le filtre choisi.

La liste est initialement vide :

```
<ol id="tasks" class="list-group"></ol>
```

Les éléments de la liste seront générés par JavaScript à partir des données des tâches.

1.2. Pourquoi utiliser `type="module"` ?

```
<script type="module" src="./todo.mjs"></script>
```

Le navigateur sait ainsi que le fichier est un module JavaScript. Il peut donc utiliser les instructions `import` et `export`.

Le fichier  `todo.mjs` sera le point d'entrée de l'application : il chargera les données, installera les gestionnaires d'événements et déclenchera l'affichage.

 Pour tester le projet, il est préférable de servir les fichiers avec un serveur de développement local, par exemple l'extension Live Server de Visual Studio Code.

2. Représenter les tâches avec des objets JavaScript

Avant de générer le HTML, nous devons définir la manière dont une tâche est représentée en mémoire.

Une tâche est un objet contenant trois propriétés :

```
const task = {  
  id: "123",  
  text: "Apprendre JavaScript",  
  completed: false  
};
```

- id identifie la tâche ;
- text contient son texte ;
- completed indique si elle est terminée.

La propriété **completed** est un booléen : elle vaut true ou false.

L'application gère un tableau de ces objets :

```
1. let tasks = [  
2.   {  
3.     id: "123",  
4.     text: "Apprendre JavaScript",  
5.     completed: false  
6.   },  
7.   {  
8.     id: "456",  
9.     text: "Réviser les modules",  
10.    completed: true  
11.  }  
12.];
```

Cet exemple illustre la structure des données. Dans notre application, le tableau sera chargé depuis **localStorage** au démarrage ; nous n'aurons donc pas besoin d'écrire ces données directement dans le code.

👑 **Principe important : les données et leur affichage sont deux choses différentes.**

Le [tableau] tasks représente les tâches. Le HTML représente leur affichage dans la page. Si une tâche est ajoutée ou supprimée, nous modifions d'abord les données, puis nous actualisons l'affichage.

3. Générer le HTML d'une tâche : tasks.mjs

Le fichier  tasks.mjs contient la fonction chargée de construire le HTML correspondant à une tâche.

🚀 Nous utilisons une chaîne de **gabarit**, délimitée par des accents graves, plutôt que de construire chaque élément avec document.createElement().

3.1. Le code de tasks.mjs

 tasks.mjs

```

1. export function createTask(task) {
2.   return `
3.     <li
4.       class="list-group-item d-flex justify-content-between
        align-items-center"
5.       data-id="${task.id}"
6.     >
7.       <div class="d-flex align-items-center gap-2 flex-grow-1 me-3">
```

```
8.      <input
9.          type="checkbox"
10.         class="form-check-input"
11.         data-action="toggle"
12.         ${task.completed ? "checked" : ""}
13.     >
14.
15.     <span class="task-text ${task.completed ?
    "text-decoration-line-through" : ""}">${escapeHTML(task.text)}</span>
16. </div>
17.
18. <div class="d-flex gap-2">
19.     <button
20.         type="button"
21.         class="btn btn-warning btn-sm"
22.         data-action="edit"
23.     >
24.         Modifier
25.     </button>
26.
27.     <button
28.         type="button"
29.         class="btn btn-danger btn-sm"
30.         data-action="delete"
31.     >
32.         Supprimer
33.     </button>
34. </div>
```

```
35.     </li>
36. `;
37.}
38.
39.function escapeHTML(text) {
40.    return String(text).replace(/([<>"])/g, character => {
41.        const entities = {
42.            "&": "&amp;",
43.            "<": "&lt;",
44.            ">": "&gt;",
45.            "\"": "&quot;",
46.            "'": "&#39;";
47.        };
48.
49.        return entities[character];
50.    });
51.}
```

3 Ajouter une tâche

L'utilisateur saisit une nouvelle tâche et clique sur « Ajouter ». **todo.mjs** crée un objet, l'ajoute au tableau, puis sauvegarde et réaffiche la liste.

todo.mjs

```
const task = {
  id: crypto.randomUUID(),
  text: text,
  completed: false
};

tasks.push(task);
saveTasks(tasks);
renderTasks();
```

Ajouter une tâche

Ajouter

3.2. Comprendre la chaîne de gabarit

La fonction reçoit un objet task en paramètre :

```
createTask(task);
```

Elle retourne une chaîne HTML. Les expressions entre **`${...}`** sont évaluées pour chaque tâche.

🚧 Par exemple : `${task.text}`

permettrait d'insérer le texte de la tâche. Dans notre code, nous utilisons :

🚧 `${escapeHTML(task.text)}`

Cette fonction **échappe** les caractères qui pourraient être interprétés comme du HTML. C'est une précaution importante puisque le texte vient d'une saisie utilisateur.

3.3. Afficher une tâche terminée

Cette expression conditionnelle :

```
${task.completed ? "checked" : ""}
```

ajoute l'attribut checked à la case à cocher lorsque la tâche est terminée.

De même :

```
${task.completed ? "text-decoration-line-through" : ""}
```

ajoute une classe Bootstrap qui barre le texte d'une tâche terminée.

3.4. Associer le HTML à une tâche

L'attribut suivant est généré pour chaque tâche :

```
data-id="${task.id}"
```

Il permet de retrouver l'identifiant de la tâche à partir de son élément HTML.



Les attributs

- data-action="toggle",
- data-action="edit" et
- data-action="delete" permettent de distinguer les différentes actions.

La fonction createTask() ne gère aucun événement et ne modifie pas le tableau. Elle se contente de produire une chaîne HTML.

Enfin, le mot-clé export permet à un autre module d'importer cette fonction.

4. Filtrer et afficher les tâches : filters.mjs

Le fichier  filters.mjs s'occupe de deux responsabilités :

- mémoriser le filtre actif et réagir aux clics sur les boutons ;
- sélectionner et afficher les tâches correspondantes.

Il importe la fonction `createTask()` depuis `tasks.mjs`.

4.1. Le code de `filters.mjs`



```

1. import { createTask } from "./tasks.mjs";
2.
3. let currentFilter = "all";
4.
5. export function applyFilter(tasks, taskList) {
6.   const visibleTasks = tasks.filter(task => {
7.     return (
8.       currentFilter === "all" ||
9.       (currentFilter === "complete" && task.completed) ||
10.      (currentFilter === "incomplete" && !task.completed)
11.    );
12.  });
13.
14.  taskList.innerHTML = visibleTasks
15.    .map(task => createTask(task))
16.    .join("");
17.}
18.
19. export function setupFilters(taskList, getTasks) {
20.  const buttons = document.querySelectorAll(".filter-btn");

```

```
21.  
22. buttons.forEach(button => {  
23.     button.addEventListener("click", () => {  
24.         currentFilter = button.dataset.filter;  
25.  
26.         buttons.forEach(item => {  
27.             item.classList.remove("btn-primary");  
28.             item.classList.add("btn-outline-primary");  
29.         });  
30.  
31.         button.classList.remove("btn-outline-primary");  
32.         button.classList.add("btn-primary");  
33.  
34.         applyFilter(getTasks(), taskList);  
35.     });  
36. });  
37.}
```

4.2. La variable `currentFilter`

```
let currentFilter = "all";
```

Elle mémorise le filtre actuellement choisi.

Les valeurs possibles sont :

- "all" : toutes les tâches ;
- "incomplete" : les tâches dont `completed` vaut `false` ;
- "complete" : les tâches dont `completed` vaut `true`.

Cette variable est déclarée en dehors des fonctions : elle conserve donc sa valeur entre les appels à `applyFilter()`.

4.3. La méthode `filter()`

```
const visibleTasks = tasks.filter(task => {
  return (
    currentFilter === "all" ||
    (currentFilter === "complete" && task.completed) ||
    (currentFilter === "incomplete" && !task.completed)
  );
});
```

La méthode `filter()` parcourt le tableau et conserve les éléments pour lesquels la fonction de test renvoie `true`.

Si `currentFilter` vaut `"complete"`, seule la condition `task.completed` permet de conserver une tâche.

Si le filtre vaut `"incomplete"`, c'est la condition `!task.completed` qui s'applique.

🕵️ Le tableau initial `tasks` n'est pas modifié. **`filter()`** produit un nouveau tableau contenant uniquement les tâches retenues.

4.4. 👑 Les méthodes `map()` et `join()`

```
taskList.innerHTML = visibleTasks
  .map(task => createTask(task))
  .join("");
```

Trois opérations se succèdent :

1. **map()** transforme chaque objet tâche en chaîne HTML grâce à `createTask(task)`.
2. **join("")** rassemble toutes les chaînes en une seule, sans séparateur.
3. **innerHTML** insère le résultat dans la liste de la page.

 Par exemple, si `visibleTasks` contient trois tâches,

1. `map()` produit trois chaînes HTML.
2. `join("")` les concatène,
3. puis le navigateur affiche les trois éléments.

 Cette approche montre clairement la différence entre filtrer les données et générer leur représentation HTML.

4.5. Installer les événements des boutons

```
export function setupFilters(taskList, getTasks) {  
  const buttons = document.querySelectorAll(".filter-btn");
```

`querySelectorAll()` récupère les trois boutons qui possèdent la classe `filter-btn`.

Pour chaque bouton, nous installons un gestionnaire de clic :

```
    button.addEventListener("click", () => {  
      currentFilter = button.dataset.filter;  
      // ...  
    });
```

La propriété `dataset.filter` permet de lire la valeur de l'attribut `data-filter`.

L'appel suivant relance le filtrage :

```
applyFilter(getTasks(), taskList);
```

Le paramètre **getTasks** est une fonction qui renvoie le tableau courant. Cela permet au module de filtrage de récupérer les données sans avoir à les posséder lui-même.

Le module  **filters.mjs** ne sauvegarde rien : il gère seulement le choix du filtre et l'affichage.

5. Sauvegarder les données : storage.mjs

Le navigateur propose une API appelée `localStorage`. Elle permet de conserver des données sous forme de chaînes de caractères, même après la fermeture ou le rechargement de la page.

Notre tableau contient des objets JavaScript. Nous devons donc convertir ce tableau en JSON pour le sauvegarder, puis effectuer l'opération inverse pour le recharger.

5.1. Le code de storage.mjs

 storage.mjs

```
1. const STORAGE_KEY = "tasksWithFilters";  
2.  
3. export function loadTasks() {  
4.   const storedTasks = localStorage.getItem(STORAGE_KEY);  
5.  
6.   if (storedTasks === null) {  
7.     return [];
```

```
8.   }
9.
10.  try {
11.    const tasks = JSON.parse(storedTasks);
12.
13.    if (!Array.isArray(tasks)) {
14.      return [];
15.    }
16.
17.    return tasks.filter(task =>
18.      task !== null &&
19.      typeof task === "object" &&
20.      typeof task.id === "string" &&
21.      typeof task.text === "string" &&
22.      typeof task.completed === "boolean"
23.    );
24.  } catch {
25.    return [];
26.  }
27.}
28.
29.export function saveTasks(tasks) {
30.  localStorage.setItem(STORAGE_KEY, JSON.stringify(tasks));
31.}
```

5.2. Sauvegarder avec `JSON.stringify()`

```
localStorage.setItem(  
  STORAGE_KEY,  
  JSON.stringify(tasks)  
);
```

`JSON.stringify()` transforme le tableau d'objets en une chaîne JSON.

`localStorage.setItem()` enregistre cette chaîne sous la clé `tasksWithFilters`.

🚀 Le nom de la clé permet de distinguer ces données d'autres données éventuellement stockées par une autre version de l'application.

5.3. Recharger avec `JSON.parse()`

```
const storedTasks = localStorage.getItem(STORAGE_KEY);
```

Cette instruction récupère la chaîne enregistrée. Si aucune donnée n'existe encore, `getItem()` renvoie `null` et la fonction retourne un tableau vide.

Sinon :

```
const tasks = JSON.parse(storedTasks);
```

reconstitue les objets JavaScript.

🕵 Le bloc `try...catch` évite que des données JSON incorrectes interrompent le démarrage de l'application. La fonction vérifie également que les données obtenues sont bien un tableau et que les tâches ont les propriétés attendues.

 Le module exporte deux fonctions : `loadTasks()` et `saveTasks()`. Les autres fichiers n'ont pas besoin de connaître les détails du stockage.

6. Coordonner l'application : `todo.mjs`

Le fichier  `todo.mjs` constitue le point d'entrée de l'application. Il importe les fonctions nécessaires, récupère les éléments HTML, charge les données et installe les événements liés aux actions de l'utilisateur.

6.1. Le code complet

 `todo.mjs`

1. `import { createTask } from "./tasks.mjs";`
2. `import { applyFilter, setupFilters } from "./filters.mjs";`
3. `import { loadTasks, saveTasks } from "./storage.mjs";`
- 4.
5. `const taskList = document.querySelector("#tasks");`
6. `const form = document.querySelector("form");`
7. `const taskInput = document.querySelector("#task");`
8. `const submitButton = form.querySelector("button[type='submit']");`
- 9.
10. `let tasks = loadTasks();`
- 11.
12. `function renderTasks() {`
13. `taskList.innerHTML = tasks.map((task) => createTask(task)).join("");`
14. `applyFilter(tasks, taskList);`

```
15.}
16.
17.// Initialiser les filtres
18.setupFilters(taskList, () => tasks);
19.
20.// Ajouter une tâche
21.form.addEventListener("submit", (event) => {
22.  event.preventDefault();
23.
24.  const text = taskInput.value.trim();
25.
26.  if (text === "") {
27.    return;
28.  }
29.
30.  const task = {
31.    id: crypto.randomUUID(),
32.    text: text,
33.    completed: false,
34.  };
35.
36.  tasks.push(task);
37.
38.  saveTasks(tasks);
39.  renderTasks();
40.
41.  taskInput.value = "";
42.  taskInput.focus();
```

```
43.});  
44.  
45.// Gérer les actions dans la liste  
46.taskList.addEventListener("click", (event) => {  
47.  const button = event.target.closest("button");  
48.  
49.  if (!button) {  
50.    return;  
51.  }  
52.  
53.  const li = button.closest("li");  
54.  
55.  if (!li) {  
56.    return;  
57.  }  
58.  
59.  const task = tasks.find((item) => item.id === li.dataset.id);  
60.  
61.  if (!task) {  
62.    return;  
63.  }  
64.  
65.  // Supprimer une tâche  
66.  if (button.matches('[data-action="delete"]')) {  
67.    tasks = tasks.filter((item) => item.id !== task.id);  
68.  
69.    saveTasks(tasks);  
70.    renderTasks();
```

```
71. return;
72. }
73.
74. // Commencer la modification
75. if (button.matches('[data-action="edit"]')) {
76.   const span = li.querySelector("span");
77.   const input = document.createElement("input");
78.
79.   input.type = "text";
80.   input.className = "form-control";
81.   input.value = task.text;
82.
83.   span.replaceWith(input);
84.
85.   button.dataset.action = "validate";
86.   button.textContent = "Valider";
87.   button.classList.replace("btn-warning", "btn-success");
88.
89.   input.focus();
90.   return;
91. }
92.
93. // Valider la modification
94. if (button.matches('[data-action="validate"]')) {
95.   const input = li.querySelector("input.form-control");
96.   const text = input.value.trim();
97.
98.   if (text === "") {
```

```
99.   input.focus();
100.   return;
101. }
102.
103.   task.text = text;
104.
105.   saveTasks(tasks);
106.   renderTasks();
107. }
108. });
109.
110. // Changer l'état d'une tâche
111. taskList.addEventListener("change", (event) => {
112.   if (!event.target.matches('[data-action="toggle"]')) {
113.     return;
114.   }
115.
116.   const li = event.target.closest("li");
117.
118.   if (!li) {
119.     return;
120.   }
121.
122.   const task = tasks.find((item) => item.id === li.dataset.id);
123.
124.   if (!task) {
125.     return;
126.   }
```

```
127.  
128.   task.completed = event.target.checked;  
129.  
130.   saveTasks(tasks);  
131.   renderTasks();  
132. });  
133.  
134. // Initialisation  
135. renderTasks();
```

6.2. Charger les modules et récupérer les éléments HTML

```
import { applyFilter, setupFilters } from "./filters.mjs";  
import { loadTasks, saveTasks } from "./storage.mjs";
```

La première instruction importe les fonctions responsables du filtrage et de l'affichage.

La seconde importe celles qui chargent et sauvegardent les données.

Les instructions suivantes récupèrent les éléments de la page :

```
const taskList = document.querySelector("#tasks");  
const form = document.querySelector("form");  
const taskInput = document.querySelector("#task");
```

Ces variables permettent d'accéder à la liste, au formulaire et au champ de saisie.

6.3. Charger les tâches au démarrage

```
let tasks = loadTasks();
```

Le tableau est initialisé à partir des données conservées dans localStorage.

Le mot-clé let est nécessaire parce que la variable tasks sera réaffectée lors d'une suppression.

La fonction d'affichage est définie ainsi :

```
function renderTasks() {  
  applyFilter(tasks, taskList);  
}
```

Chaque appel à renderTasks() demande au module de filtrage de réafficher les données actuelles en respectant le filtre choisi.

Nous installons ensuite les événements des boutons de filtrage :

```
setupFilters(taskList, () => tasks);
```

L'expression () => tasks est une fonction fléchée. Lorsqu'elle est appelée, elle renvoie le tableau courant.

Enfin, renderTasks() affiche les tâches chargées au démarrage.

6.4. Ajouter une tâche

Le formulaire réagit à l'événement submit :

```
form.addEventListener("submit", event => {  
  event.preventDefault();
```

event.preventDefault() empêche le comportement habituel du formulaire, qui serait de recharger la page.

Nous récupérons ensuite le texte saisi :

```
const text = taskInput.value.trim();
```

La méthode `trim()` supprime les espaces au début et à la fin. Si le texte est vide, aucune tâche n'est ajoutée.

L'objet est créé avec un identifiant unique :

```
const task = {  
  id: crypto.randomUUID(),  
  text: text,  
  completed: false  
};
```

`crypto.randomUUID()` génère un identifiant aléatoire. La nouvelle tâche est ensuite ajoutée au tableau :

```
tasks.push(task);
```

Enfin, les données sont sauvegardées et l'affichage est actualisé :

```
saveTasks(tasks);  
renderTasks();
```

Le champ de saisie est vidé et retrouve le focus grâce à :

```
taskInput.value = "";  
taskInput.focus();
```

6.5. Marquer une tâche comme terminée

L'événement `change` est utilisé pour réagir au changement d'état de la case à cocher :

```
taskList.addEventListener("change", event => {
  if (!event.target.matches('[data-action="toggle"]')) {
    return;
  }
});
```

Le gestionnaire est installé sur la liste entière, et non sur chaque case à cocher individuellement. Il vérifie que la cible de l'événement est bien une case portant `data-action="toggle"`.

Nous retrouvons ensuite l'élément `li` qui contient la case :

```
const li = event.target.closest("li");
```

Puis l'objet correspondant dans le tableau :

```
const task = tasks.find(item => item.id === li.dataset.id);
```

La méthode `find()` renvoie le premier objet dont l'identifiant correspond à celui de l'élément HTML.

L'état de la tâche est mis à jour :

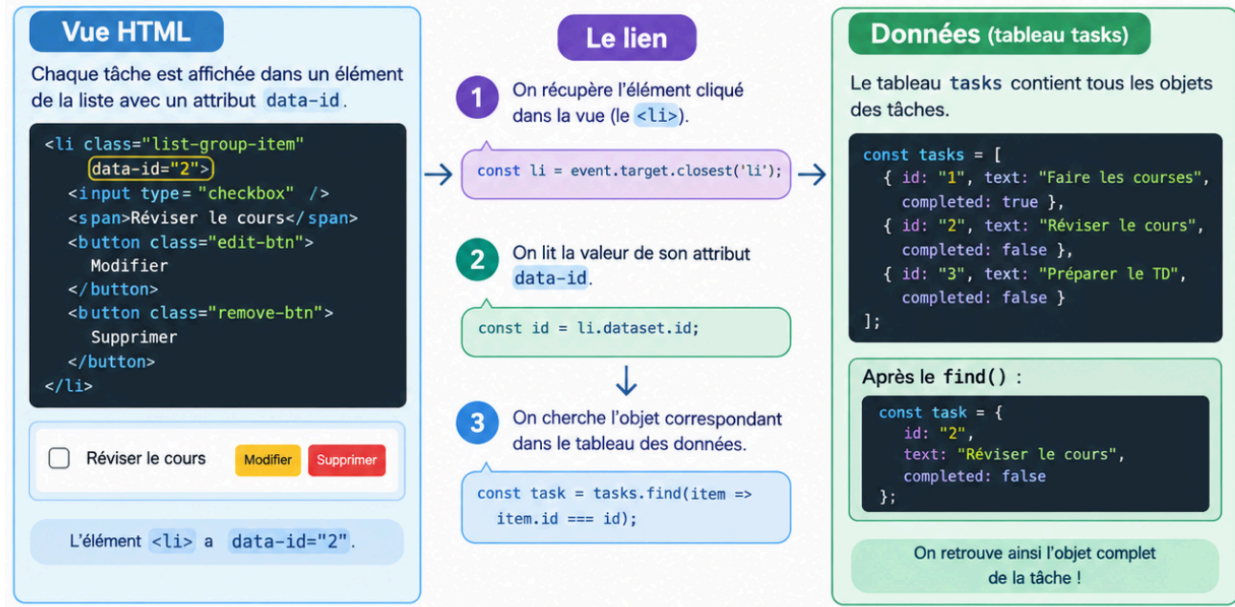
```
task.completed = event.target.checked;
```

La propriété `checked` indique si la case est cochée. Nous sauvegardons les données et réaffichons la liste pour actualiser le style et respecter le filtre actif.

En résumé

👑 L'idée essentielle : le navigateur affiche des **éléments** HTML, mais l'application manipule des **objets** JavaScript. L'identifiant `id`, recopié dans l'attribut HTML `data-id`, permet de retrouver l'objet correspondant à l'élément affiché et donc

de modifier les bonnes données.



Les trois étapes du rapprochement entre la vue HTML et les données JavaScript.

1. Retrouver l'élément HTML concerné. Lorsqu'on coche une case, `closest("li")` permet de retrouver l'élément de liste qui contient cette case.
2. Lire son identifiant. L'attribut `data-id` de cet élément fournit l'identifiant de la tâche affichée.
3. Retrouver l'objet correspondant. La méthode **find()** parcourt le tableau `tasks` et recherche l'objet dont la propriété `id` correspond à cet identifiant.

6.6. Supprimer une tâche

Le gestionnaire de clic commence par vérifier si le clic provient du bouton de suppression :

```
const button = event.target.closest('[data-action="delete"]');

if (!button) {
  return;
}
```

Cette vérification permet d'ignorer les autres clics dans la liste.

Nous retrouvons l'identifiant de la tâche puis nous construisons un nouveau tableau sans celle-ci :

```
tasks = tasks.filter(item => item.id !== li.dataset.id);
```

La méthode `filter()` **conserve** toutes les tâches dont l'identifiant est différent de celui de la tâche supprimée.

Il ne reste qu'à sauvegarder et actualiser l'affichage :

```
saveTasks(tasks);
renderTasks();
```

6.7. Pourquoi utiliser la délégation d'événements ?

Le code installe les gestionnaires sur `taskList`, et non directement sur chaque case ou chaque bouton.

C'est important parce que les éléments de la liste sont recréés à chaque affichage par `innerHTML`. Les gestionnaires attachés directement aux anciens éléments disparaîtraient avec eux.

Avec la délégation d'événements, le gestionnaire reste installé sur la liste et peut traiter les événements provenant des nouveaux éléments.

Limite de cette version : le bouton « Modifier » est présent dans le HTML généré, mais son comportement n'est pas encore programmé. Il faudra ajouter un gestionnaire pour permettre de modifier le texte d'une tâche.

7. Comprendre le cycle de fonctionnement

Lorsqu'une action est effectuée, l'application suit généralement ce cycle :

1. L'utilisateur déclenche un événement.
2. `todo.mjs` met à jour le tableau `tasks`.
3. `storage.mjs` sauvegarde le tableau.
4. `filters.mjs` sélectionne les tâches visibles.
5. `tasks.mjs` produit le HTML de chaque tâche.
6. Le navigateur affiche le résultat.

 Par exemple, lors de l'ajout d'une tâche :

```
tasks.push(task);  
saveTasks(tasks);  
renderTasks();
```

Ces trois instructions résument le principe central de l'application : modifier les données, les sauvegarder, puis actualiser l'affichage.

8. Pourquoi séparer le code en plusieurs fichiers ?

La séparation des modules évite de réunir toutes les responsabilités dans un seul fichier.

Fichier	Responsabilité
index.html	Définir la structure de la page
todo.mjs	Gérer les événements et coordonner les opérations
tasks.mjs	Générer le HTML d'une tâche
filters.mjs	Filtrer et afficher les tâches
storage.mjs	Charger et sauvegarder les données

Cette organisation facilite les modifications. Par exemple, pour changer la présentation d'une tâche, il suffit principalement de modifier `tasks.mjs`. Pour changer la manière de sauvegarder les données, on peut intervenir dans `storage.mjs` sans réécrire les filtres.

🚀 Les modules communiquent au moyen des fonctions qu'ils exportent et importent.

👑 L'idée essentielle est de ne pas confondre les données, leur représentation HTML, leur stockage et les actions de l'utilisateur. Chaque module prend en charge une partie de ce travail, et `todo.mjs` coordonne l'ensemble.

🛟 Help : J'aime dans le code suivre les événements

API Django : multi... Suivez votre colis j... Tache : transformati... Créer une liste de tâ... Accueil - Espace ens... Afficher dans le DO... Card

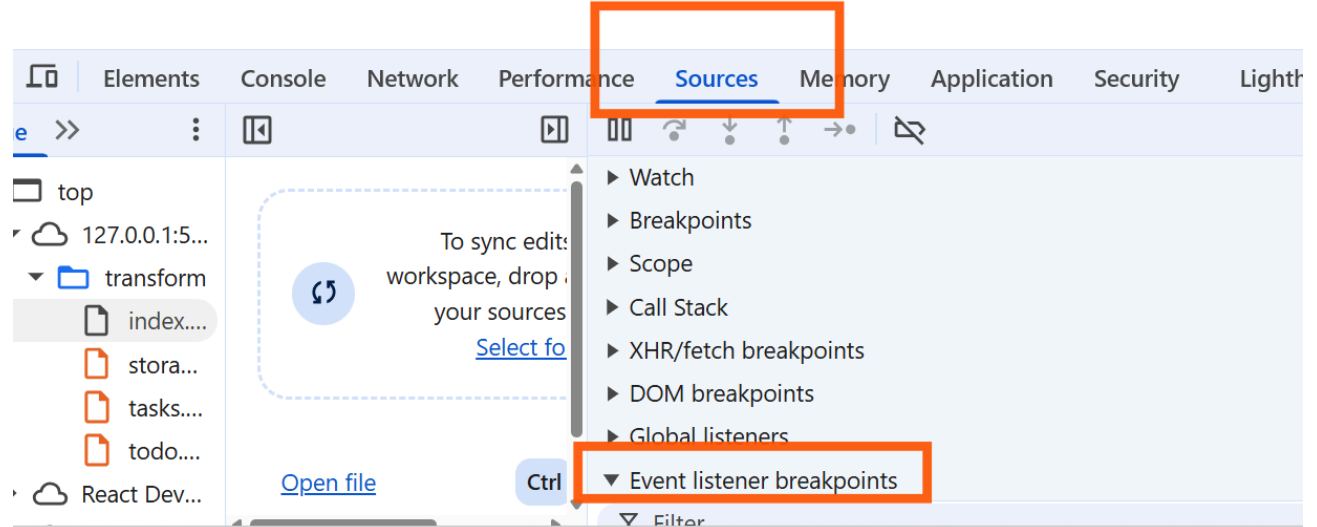
Ma liste de tâches

Nouvelle tâche

Ajouter

crer

fuir



The screenshot shows the Chrome DevTools interface. The 'Sources' panel is active, and the 'Event listener breakpoints' option is highlighted in the left sidebar. The main pane displays a list of event listeners for the 'transform' component, including 'index...', 'stora...', 'tasks...', and 'todo...'. The 'Event listener breakpoints' option is highlighted with a red box, and the 'Event listener breakpoints' option is also highlighted with a red box.