



---

# RAPPORT COMPLET

## Développement Web

HTML • CSS • JavaScript

*Analyse des projets pédagogiques*

---

Muckaya Florida RAMALALAHASITERA

Classe L1B • N°331

Année universitaire 2025 – 2026

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Quiz Interactif</b>	<b>4</b>
1.1 Vue d'ensemble . . . . .	4
1.2 Structure HTML . . . . .	4
1.3 Mise en forme CSS . . . . .	4
1.4 Logique JavaScript . . . . .	4
1.4.1 Intercepter la soumission du formulaire . . . . .	4
1.4.2 Retour visuel avec animations . . . . .	5
<b>2 Application Météo</b>	<b>7</b>
2.1 Vue d'ensemble . . . . .	7
2.2 Structure HTML . . . . .	7
2.3 Mise en forme CSS . . . . .	7
2.4 Logique JavaScript . . . . .	7
2.4.1 Demander la position GPS . . . . .	7
2.4.2 Appel HTTP asynchrone avec fetch() et les Promises . . . . .	8
2.4.3 Adapter l'interface selon l'heure . . . . .	8
<b>3 To-Do List</b>	<b>10</b>
3.1 Vue d'ensemble . . . . .	10
3.2 Structure HTML . . . . .	10
3.3 Mise en forme CSS . . . . .	10
3.4 Logique JavaScript . . . . .	10
3.4.1 Modéliser une tâche comme un objet . . . . .	10
3.4.2 Construire l'interface élément par élément . . . . .	11
3.4.3 Cocher et supprimer une tâche . . . . .	11
<b>4 Pokédex</b>	<b>13</b>
4.1 Vue d'ensemble . . . . .	13
4.2 Structure HTML . . . . .	13
4.3 Mise en forme CSS . . . . .	13
4.4 Logique JavaScript . . . . .	13
4.4.1 Orchestrer plusieurs requêtes avec Promise.all() . . . . .	13
4.4.2 Détecter le bas de la page pour le scroll infini . . . . .	14
4.4.3 Recherche en temps réel . . . . .	14
<b>5 Timer Pomodoro</b>	<b>16</b>
5.1 Vue d'ensemble . . . . .	16
5.2 Structure HTML . . . . .	16
5.3 Mise en forme CSS . . . . .	16
5.4 Logique JavaScript . . . . .	16
5.4.1 Centraliser l'état en variables . . . . .	16

5.4.2	Le moteur du chronomètre : setInterval()	17
5.4.3	Pause, reprise et réinitialisation	17
<b>6</b>	<b>Gestion des Cookies</b>	<b>19</b>
6.1	Vue d'ensemble	19
6.2	Structure HTML	19
6.3	Mise en forme CSS	19
6.4	Logique JavaScript	19
6.4.1	Un gestionnaire unifié pour plusieurs boutons	19
6.4.2	Créer un cookie avec encodage	20
6.4.3	Lire et supprimer des cookies	21
	<b>Conclusion</b>	<b>22</b>

# Introduction

Ce rapport est le résultat de six semaines de travail pratique sur les technologies web fondamentales. À travers six projets concrets, j’ai appris à faire collaborer HTML, CSS et JavaScript pour construire des interfaces dynamiques et interactives.

**Pourquoi ce rapport ?** L’objectif n’est pas de lister des fonctionnalités, mais de *comprendre* pourquoi chaque outil existe, dans quelle situation l’utiliser, et quel problème concret il résout. Chaque chapitre suit une logique en trois temps :

#	Couche	Rôle
1	HTML	Structure et sémantique de la page
2	CSS	Apparence et mise en forme
3	JavaScript	Logique, interaction et dynamisme

JavaScript est la couche qui reçoit le plus d’attention dans ce rapport — c’est elle qui transforme une page statique en application vivante. Chaque bloc de code est accompagné d’explications détaillées, y compris les “pourquoi” qui ne sont pas toujours évidents au premier coup d’œil.

# Chapitre 1

## Quiz Interactif

### 1.1 Vue d’ensemble

Le quiz interactif est une application de type questionnaire à choix multiples (QCM). L'utilisateur sélectionne ses réponses et clique sur "Soumettre" : un retour visuel immédiat lui indique ce qui est juste ou faux, avec une petite animation de tremblement sur les mauvaises réponses.

### 1.2 Structure HTML

Le HTML de ce projet est volontairement simple. On utilise un formulaire `<form>` qui contient des `<input type="radio">` pour chaque choix de réponse. Chaque groupe de radios partage le même attribut `name` (par exemple `name="q1"`) — c’est ce mécanisme natif du HTML qui garantit qu’un seul choix peut être coché par question.

### 1.3 Mise en forme CSS

La partie visuelle la plus intéressante est l’animation de tremblement sur les mauvaises réponses. Elle est définie avec `@keyframes echec`, qui déplace horizontalement le bloc de quelques pixels à intervalles très rapides, créant un effet de “non” tout à fait naturel et intuitif pour l’utilisateur.

### 1.4 Logique JavaScript

#### 1.4.1 Intercepter la soumission du formulaire

```
1 form.addEventListener('submit', (e) => {
2   e.preventDefault();
3   for(i = 1; i < 6; i++){
4     tableauResultats.push(
5       document.querySelectorAll(
6         'input[name="q${i}"]:checked'
7       ).value
8     );
9   }
10  verifFunc(tableauResultats);
11  tableauResultats = [];
12 })
```

Listing 1.1 – Gestion de l’événement submit

**Pourquoi `e.preventDefault()` ?** Par défaut, quand on clique sur le bouton “Soumettre” d’un formulaire HTML, le navigateur envoie une requête HTTP et recharge la page. Ce comportement remonte à l’époque où les sites web étaient entièrement générés côté serveur. Ici, on veut traiter les réponses directement dans le navigateur, sans rechargement. `preventDefault()` dit simplement au navigateur : “je gère cet événement moi-même, n’exécute pas ton comportement habituel.”

**Comment lit-on les radios ?** Le sélecteur CSS `input[name="q1"]:checked` cible l’input radio du groupe “q1” qui est actuellement coché. C’est une élégance du CSS : la pseudo-classe `:checked` filtre en temps réel l’état des éléments de formulaire, et on peut l’utiliser directement depuis JavaScript via `querySelectorAll`.

**Pourquoi réinitialiser `tableauResultats = []` à la fin ?** Si on ne vide pas le tableau après chaque soumission, les résultats de la tentative précédente s’accumulent dedans. La prochaine soumission ajouterait les nouvelles réponses à la suite des anciennes, ce qui rendrait la vérification complètement fausse.

#### 1.4.2 Retour visuel avec animations

```
1 function couleursFonction(tabValBool){
2   for(let j = 0; j < tabValBool.length; j++){
3     if(tabValBool[j] === true){
4       toutesLesQuestions[j].style.background = 'lightgreen';
5     } else {
6       toutesLesQuestions[j].style.background = '#ffb8b8';
7       toutesLesQuestions[j].classList.add('echec');
8       setTimeout(() => {
9         toutesLesQuestions[j].classList.remove('echec');
10      }, 500);
11     }
12   }
13 }
```

Listing 1.2 – Couleurs et animation par résultat

**Pourquoi un tableau de booléens ?** La fonction `verifFunc` ne colorie pas directement les questions. Elle produit d’abord un tableau de `true/false` qu’elle transmet ensuite à `couleursFonction`. Ce découpage respecte un principe de base en programmation : une fonction doit faire une seule chose à la fois. Vérifier les réponses et colorier l’interface sont deux responsabilités distinctes, et les séparer rend le code plus lisible et plus facile à déboguer.

**Le piège de l’animation CSS :** Ajouter une classe CSS déclenche l’animation `@keyframes` associée. Mais une animation ne peut se rejouer que si la classe a d’abord été retirée. Si on laisse la classe “echec” en permanence sur l’élément, l’animation ne joue qu’une seule fois et ne se redéclenchera plus jamais lors des soumissions suivantes. Le `setTimeout(() => classList.remove('echec'), 500)` retire la classe au bout de 500ms — juste après la fin de l’animation — pour qu’elle soit à nouveau disponible la prochaine fois.

**Qu’est-ce que `setTimeout` exactement ?** C’est un minuteur asynchrone : on lui donne une fonction et un délai en millisecondes, et JavaScript l’exécutera *plus tard*, sans bloquer l’exécution du reste du code. C’est ce qu’on appelle la programmation événementielle asynchrone — une caractéristique fondamentale de JavaScript qu’on retrouvera à plusieurs reprises dans ce rapport.



### Concepts clés du Quiz

**preventDefault()** pour reprendre le contrôle des formulaires HTML — **:checked** comme filtre CSS sur l'état des radios — **setTimeout()** pour exécuter du code avec un délai — **classList.add/remove()** pour manipuler les classes CSS depuis JS — **Tableau de booléens** pour séparer vérification et affichage

## Chapitre 2

# Application Météo

### 2.1 Vue d'ensemble

L'application météo localise automatiquement l'utilisateur via le GPS du navigateur, puis interroge l'API OpenWeatherMap pour afficher la météo actuelle et les prévisions. C'est un projet central parce qu'il introduit deux concepts très importants : les API web externes et la programmation asynchrone.

### 2.2 Structure HTML

Le HTML se réduit à quelques zones d'affichage (un titre, un paragraphe pour la température, un conteneur pour les icônes). Tout le contenu visible est injecté dynamiquement par JavaScript après réception des données de l'API.

### 2.3 Mise en forme CSS

CSS Grid est utilisé pour organiser la page en trois zones : les conditions actuelles en haut, les prévisions horaires au milieu, et les prévisions de la semaine en bas. La propriété `grid-template-areas` permet de nommer les zones et de les positionner de manière très lisible dans le code CSS.

### 2.4 Logique JavaScript

#### 2.4.1 Demander la position GPS

```
1 navigator.geolocation.getCurrentPosition(  
2     position => {  
3         let long = position.coords.longitude;  
4         let lat  = position.coords.latitude;  
5         AppelAPI(long, lat);  
6     },  
7     () => {  
8         alert("Géolocalisation refusée, veuillez l'activer");  
9     }  
10 );
```

Listing 2.1 – API de géolocalisation native

**Qu'est-ce que `navigator.geolocation` ?** C'est une API intégrée dans tous les navigateurs modernes, disponible sans rien installer ni importer. Elle permet d'interroger le système GPS ou le réseau Wi-Fi pour déterminer la position géographique. Avant de donner quoi que ce soit, le navigateur affiche une alerte à l'utilisateur pour lui demander son consentement — c'est une mesure de sécurité imposée par les standards web modernes.

**Le schéma à deux callbacks :** `getCurrentPosition` attend deux fonctions en paramètres. La première est appelée si tout se passe bien (l'utilisateur accepte et la position est

disponible). La seconde est appelée en cas d'échec (refus, GPS désactivé, délai dépassé). Ce schéma est très courant en JavaScript : on anticipe les deux issues possibles pour que l'application reste fonctionnelle dans tous les cas.

### 2.4.2 Appel HTTP asynchrone avec `fetch()` et les Promises

```

1 function AppelAPI(long, lat){
2     fetch('https://api.openweathermap.org/data/2.5/onecall
3         ?lat=${lat}&lon=${long}&units=metric&lang=fr
4         &appid=${CLEFAPI}')
5     .then(response => response.json())
6     .then((data) => {
7         resultatAPI = data;
8         temps.innerHTML =
9             resultatAPI.current.weather[0].description;
10        temperature.innerHTML =
11            `${Math.trunc(resultatAPI.current.temp)} deg`;
12        localisation.innerHTML = resultatAPI.timezone;
13    })
14 }
```

Listing 2.2 – Requête vers OpenWeatherMap

**Comprendre `fetch()` et les Promises :** Une requête réseau prend du temps — parfois plusieurs secondes selon la connexion. Si JavaScript attendait la réponse sans rien faire, toute la page serait figée pendant ce temps. Les Promises résolvent ce problème : `fetch()` retourne immédiatement un objet Promise (une “promesse de valeur future”) et le reste du code continue à s'exécuter normalement. Quand la réponse arrive enfin, le premier `.then()` se déclenche.

**Pourquoi deux `.then()` ?** La réponse HTTP brute est un flux de données binaires, pas encore un objet JavaScript utilisable. `response.json()` fait la traduction et retourne elle-même une Promise (car lire ce flux prend un instant). Le second `.then()` reçoit enfin l'objet JavaScript prêt à l'emploi. Ce chaînage de Promises est beaucoup plus lisible que l'ancienne méthode des callbacks imbriqués.

**Les template literals** (les backticks ‘...\$. . . . .’) permettent d'insérer des variables directement dans une chaîne de caractères. Sans eux, construire l'URL de l'API nécessiterait une concaténation lourde avec des + partout.

### 2.4.3 Adapter l'interface selon l'heure

```

1 if(heureActuelle >= 6 && heureActuelle < 21){
2     imIcon.src =
3         'res/jour/${resultatAPI.current.weather[0].icon}.svg';
4 } else {
5     imIcon.src =
6         'res/nuit/${resultatAPI.current.weather[0].icon}.svg';
7 }
```

Listing 2.3 – Icônes jour et nuit

**L'idée derrière ce code :** L'API retourne un code d'icône comme 01d (soleil) ou 09d (pluie). Plutôt que de gérer deux jeux de conditions complexes, on organise simplement les fichiers SVG dans deux dossiers /jour et /nuit, et on choisit le bon dossier selon l'heure locale. C'est un exemple concret d'architecture simple : bien organiser ses ressources évite

de complexifier inutilement le code.



#### Concepts clés de la Météo

**navigator.geolocation** API GPS native du navigateur — **fetch()** pour les requêtes HTTP asynchrones — **Promises** + **.then()** pour gérer les valeurs futures — **Math.trunc()** pour enlever les décimales — **Template literals** pour construire l'URL dynamiquement

# Chapitre 3

## To-Do List

### 3.1 Vue d'ensemble

L'application To-Do List permet d'ajouter des tâches, de les cocher comme terminées, et de les supprimer. La particularité de ce projet est que le HTML initial est presque vide : toute la liste est créée et gérée par JavaScript pur. C'est une introduction concrète à la génération dynamique de DOM.

### 3.2 Structure HTML

Le HTML se limite à un formulaire avec un `<input>` texte et une balise `<ul>` vide. JavaScript se charge de remplir cette liste au fur et à mesure des ajouts. Cette approche illustre bien comment fonctionnent les applications modernes à page unique (SPA).

### 3.3 Mise en forme CSS

Les pseudo-classes `:hover` et `:focus` donnent un retour visuel à chaque interaction. La propriété `text-decoration: line-through` barre le texte des tâches terminées, appliquée via une classe CSS activée par JavaScript.

### 3.4 Logique JavaScript

#### 3.4.1 Modéliser une tâche comme un objet

```
1 function rajouterTache(text){
2     const todo = {
3         text,
4         id: Date.now()
5     }
6     afficherListe(todo);
7 }
```

Listing 3.1 – Création d'un objet tâche

**Pourquoi un objet plutôt qu'une simple chaîne de caractères ?** On pourrait stocker la tâche comme juste son texte. Mais dès qu'on doit supprimer une tâche précise, on a besoin d'un identifiant unique pour la distinguer des autres. Un objet avec `text` et `id` regroupe naturellement ces deux informations de façon cohérente.

**La magie de `Date.now()` :** Cette méthode retourne le nombre de millisecondes écoulées depuis le 1er janvier 1970 (l'"epoch Unix"). Deux tâches créées à des moments différents auront toujours des valeurs différentes — même si elles sont créées à une milliseconde d'intervalle. C'est une astuce pratique pour générer des identifiants uniques sans avoir besoin d'une infrastructure complexe.

**La notation raccourcie `{text}` :** En JavaScript moderne (ES6+), quand le nom de la

propriété et le nom de la variable sont identiques, on peut écrire `{text}` au lieu de `{text: text}`. C'est du sucre syntaxique qui rend le code plus concis et agréable à lire.

### 3.4.2 Construire l'interface élément par élément

```
1 function afficherListe(todo){
2     const item = document.createElement('li');
3     item.setAttribute('data-key', todo.id);
4
5     const input = document.createElement('input');
6     input.setAttribute('type', 'checkbox');
7     input.addEventListener('click', tacheFaite);
8     item.appendChild(input);
9
10    const txt = document.createElement('span');
11    txt.innerText = todo.text;
12    item.appendChild(txt);
13
14    const btn = document.createElement('button');
15    btn.addEventListener('click', supprimerTache);
16    item.appendChild(btn);
17
18    liste.appendChild(item);
19    toutesLesTaches.push(item);
20 }
```

Listing 3.2 – Génération du DOM pour une tâche

**Le processus de construction du DOM :** `createElement()` crée un élément HTML en mémoire — il n'est pas encore visible dans la page. `appendChild()` l'insère comme enfant d'un autre élément. C'est un peu comme construire avec des LEGO : on assemble d'abord les pièces, puis on pose l'ensemble dans la page.

**L'attribut data-key :** Les attributs `data-*` sont des attributs personnalisés HTML5 qui permettent de stocker des données directement sur un élément. Ici, on “attache” l'ID de la tâche à son élément `<li>`. Quand l'utilisateur clique sur “Supprimer”, on peut lire cet attribut pour savoir exactement quelle tâche doit être retirée.

**Pourquoi attacher les événements ici et pas ailleurs ?** Les éléments n'existent que depuis cet instant précis. On ne peut écouter les événements d'un élément que s'il existe déjà. Chaque élément reçoit donc ses propres écouteurs au moment exact de sa création.

### 3.4.3 Cocher et supprimer une tâche

```
1 function tacheFaite(e){
2     e.target.parentNode.classList.toggle('finDeTache');
3 }
4
5 function supprimerTache(e){
6     toutesLesTaches.forEach(el => {
7         if(e.target.parentNode.getAttribute('data-key')
8             === el.getAttribute('data-key')){
9             el.remove();
10        }
11    })
12    toutesLesTaches = toutesLesTaches.filter(li =>
13        li.dataset.key !== e.target.parentNode.dataset.key
14    );
15 }
```

15 }

## Listing 3.3 – Toggle et suppression

**L'élégance de `toggle()`** : La méthode `classList.toggle('finDeTache')` fait exactement ce que son nom suggère : si la classe est présente, elle la retire ; si elle est absente, elle l'ajoute. Un seul appel suffit pour basculer entre l'état “en cours” et l'état “terminée”, sans avoir à vérifier l'état actuel soi-même.

**Deux suppressions nécessaires** : Supprimer une tâche exige deux opérations distinctes. D'abord `el.remove()` retire l'élément du DOM (la page ne l'affiche plus). Ensuite `filter()` retire la référence de notre tableau `toutesLesTaches`. Si on oublie la deuxième étape, le tableau contiendrait des références vers des éléments qui n'existent plus en mémoire — ce sont des fuites mémoire potentielles, et les itérations futures sur ce tableau produiraient des comportements imprévisibles.



## Concepts clés de la To-Do List

**`createElement`** + **`appendChild`** pour construire le DOM dynamiquement — **`data-`**\* pour lier des métadonnées aux éléments HTML — **`Date.now()`** comme générateur d'identifiants uniques — **`classList.toggle()`** pour alterner un état — **`filter()`** pour retirer proprement un élément d'un tableau

## Chapitre 4

# Pokédex

### 4.1 Vue d'ensemble

Le Pokédex affiche des cartes Pokémon en interrogeant l'API publique PokeAPI. C'est le projet le plus ambitieux techniquement : il combine des appels API multiples et parallèles, un scroll infini, et une recherche en temps réel. C'est ici que les Promises deviennent vraiment indispensables.

### 4.2 Structure HTML

La structure de base est une grille vide. Chaque carte Pokémon (image, nom, type, statistiques) est entièrement générée par JavaScript après réception des données de l'API. Le formulaire de recherche est en position `sticky` pour rester visible lors du défilement.

### 4.3 Mise en forme CSS

La pseudo-classe `:focus-within` est utilisée sur le conteneur du champ de recherche : elle s'active quand un élément enfant (ici l'input) prend le focus. Cela permet d'animer le label ou la bordure du conteneur entier sans écrire une seule ligne de JavaScript.

### 4.4 Logique JavaScript

#### 4.4.1 Orchestrer plusieurs requêtes avec `Promise.all()`

```
1 function fetchPokemonBase() {
2   const promises = [];
3
4   fetch('https://pokeapi.co/api/v2/pokemon?limit=${limite}')
5     .then(response => response.json())
6     .then((allPoke) => {
7       allPoke.results.forEach((pokemon) => {
8         promises.push(
9           fetchPokemonComplet(pokemon)
10            .catch(e => console.error(e))
11         );
12       });
13     })
14     .then(() => {
15       Promise.all(promises).then(() => {
16         tableauFin = allPokemon
17           .sort((a, b) => a.id - b.id)
18           .slice(0, PokeNombreDebut);
19         createCard(tableauFin);
20         chargement.style.display = 'none';
21       });
22     })
23 }
```

## Listing 4.1 – Chargement parallèle des Pokémon

**Le problème qu'on cherche à résoudre :** Charger 151 Pokémon implique 151 requêtes API distinctes. Si on les envoyait une par une en attendant chaque réponse, l'opération prendrait plusieurs minutes. On veut les lancer toutes en même temps (en parallèle), mais savoir quand elles sont toutes terminées pour afficher les résultats.

**Comment fonctionne ce code :** On lance toutes les requêtes d'un coup dans une boucle `forEach`. Chaque appel `fetchPokemonComplet()` retourne une Promise, qu'on accumule dans le tableau `promises`. Ensuite, `Promise.all(promises)` crée une nouvelle Promise qui ne se résout que quand *toutes* les Promises du tableau sont terminées. C'est comme un chef d'orchestre qui attend que chaque musicien ait fini sa partie avant de baisser la baguette.

**Pourquoi `.sort()` est nécessaire ?** Les requêtes réseau ne reviennent pas dans l'ordre de leur envoi. Un Pokémon numéro 50 peut arriver avant le numéro 3 si son serveur répond plus vite. Sans tri, les cartes seraient dans un ordre aléatoire à chaque chargement. `.sort((a, b) => a.id - b.id)` trie par ID numérique : si le résultat est négatif, `a` vient avant `b` ; si positif, c'est l'inverse.

#### 4.4.2 Détecter le bas de la page pour le scroll infini

```
1 window.addEventListener('scroll', () => {
2     const {scrollTop, scrollHeight, clientHeight} =
3         document.documentElement;
4
5     if(clientHeight + scrollTop >= scrollHeight - 20){
6         addPoke(6);
7     }
8 })
```

## Listing 4.2 – Détection du scroll infini

**La géométrie du scroll :** Trois mesures permettent de savoir si on est en bas de la page. `clientHeight` est la hauteur visible de la fenêtre. `scrollTop` est la distance déjà défilée depuis le haut. `scrollHeight` est la hauteur totale du document, y compris la partie non visible. Quand la somme des deux premières est égale (ou proche) à la troisième, on est arrivés en bas.

**La marge de 20px :** On ne compare pas avec une égalité stricte mais avec un seuil de 20 pixels. Les calculs de scroll peuvent être imprécis selon le navigateur et la résolution d'écran, et une égalité stricte ne se déclencherait presque jamais. Cette petite marge est une bonne pratique pour fiabiliser la détection.

**Le destructuring ES6 :** `const {scrollTop, scrollHeight, clientHeight} = document.documen` extrait trois propriétés en une seule ligne. Sans destructuring, on écrirait trois lignes séparées avec des répétitions. Cette syntaxe est très répandue en JavaScript moderne.

#### 4.4.3 Recherche en temps réel

```
1 function recherche(){
2     let filter      = searchInput.value.toUpperCase();
```

```
3   let allLi      = document.querySelectorAll('li');
4   let allTitles = document.querySelectorAll("li > h5");
5
6   for(i = 0; i < allLi.length; i++){
7       let titleValue = allTitles[i].innerText;
8       allLi[i].style.display =
9           titleValue.toUpperCase().indexOf(filter) > -1
10          ? "flex" : "none";
11   }
12 }
```

Listing 4.3 – Filtrage des cartes par nom

**Insensibilité à la casse :** On convertit les deux chaînes en majuscules avec `toUpperCase()` avant de les comparer. Ainsi, chercher “pikachu”, “PIKACHU” ou “Pikachu” donne le même résultat. Sans cette normalisation, la recherche serait frustrante pour l'utilisateur.

**Pourquoi `indexOf()` ?** On ne cherche pas une correspondance exacte mais si le texte *contient* la saisie. `indexOf(filter)` retourne la position de la saisie dans le nom du Pokémon, ou -1 si elle est absente. La condition `> -1` signifie donc “si trouvé quelque part”. L'opérateur ternaire `condition ? valeurSiVrai : valeurSiFaux` est une forme compacte de `if/else` bien adaptée aux expressions simples.

#### Concepts clés du Pokédex

**Promise.all()** pour synchroniser plusieurs requêtes parallèles — **.sort()** avec comparateur pour trier un tableau d'objets — **Scroll event** et calcul géométrique pour le scroll infini — **Destructuring ES6** pour extraire plusieurs propriétés — **indexOf()** pour la recherche partielle insensible à la casse

## Chapitre 5

# Timer Pomodoro

### 5.1 Vue d'ensemble

Le timer Pomodoro est basé sur la technique de gestion du temps du même nom : 25 minutes de travail concentré suivies de 5 minutes de pause, en cycles répétitifs. Ce projet introduit la gestion précise du temps avec `setInterval`, et la notion d'état applicatif géré par des variables booléennes.

### 5.2 Structure HTML

Le HTML se résume à deux affichages numériques (travail et repos), un compteur de cycles, et trois boutons : Démarrer, Pause/Reprendre, Réinitialiser. La police `Digital-7` importée via `@font-face` simule l'apparence d'un affichage LCD.

### 5.3 Mise en forme CSS

L'effet LCD est obtenu en combinant la police monospace `Digital-7` avec une couleur verte sur fond sombre. Des animations `@keyframes` font pulser les chiffres lorsque le timer approche de zéro, créant un sentiment d'urgence bien reconnaissable.

### 5.4 Logique JavaScript

#### 5.4.1 Centraliser l'état en variables

```
1 const T_INIT = 1500; // 25 * 60 secondes
2 const T_REP  = 300;  //  5 * 60 secondes
3
4 let tempsInitial    = T_INIT;
5 let tempsDeRepos    = T_REP;
6 let pause           = false;
7 let nbDeCycles      = 0;
8 let ChronoEnMarche = false;
9 let timer;
```

Listing 5.1 – Variables d'état global

**Travailler en secondes :** On stocke le temps en secondes entières plutôt qu'au format MM :SS. Cela simplifie tout : décrémenter d'une unité, calculer les minutes restantes, vérifier si on est à zéro. La conversion en format lisible se fait uniquement au moment de l'affichage.

**Les constantes comme source de vérité unique :** En définissant `T_INIT = 1500` et `T_REP = 300` comme constantes, on centralise la configuration. Si on veut modifier la durée à 30 minutes, on change une seule ligne. Sans ces constantes, il faudrait rechercher le nombre "1500" partout dans le code à chaque modification.

**La variable timer :** C'est peut-être la variable la plus importante ici. `setInterval()` retourne un numéro d'identification de l'intervalle créé. Sans le stocker dans `timer`, on ne pourrait jamais l'arrêter. C'est comme récupérer le ticket d'un vestiaire : sans lui, impossible de reprendre son manteau.

#### 5.4.2 Le moteur du chronomètre : `setInterval()`

```

1 btnGo.addEventListener('click', () => {
2     if(!ChronoEnMarche){
3         ChronoEnMarche = true;
4         timer = setInterval(() => {
5             if(!pause && tempsInitial > 0){
6                 tempsInitial--;
7                 affichageTravail.innerHTML =
8                     `${Math.trunc(tempsInitial / 60)} : ` +
9                     `${tempsInitial % 60 < 10
10                        ? '0' + (tempsInitial % 60)
11                        : tempsInitial % 60}`;
12             }
13         }, 1000);
14     }
15 })

```

Listing 5.2 – Démarrage du timer et affichage

**Comment fonctionne `setInterval` ?** On lui passe une fonction et un délai en millisecondes. Il appelle cette fonction de manière répétée, à intervalles réguliers, jusqu'à ce qu'on l'arrête. Avec 1000ms, la fonction est appelée toutes les secondes — c'est le "tick" de notre horloge.

**La garde `if(!ChronoEnMarche)` :** Si l'utilisateur clique plusieurs fois sur "Démarrer", sans cette condition chaque clic créerait un nouvel `setInterval` supplémentaire. Avec deux intervalles actifs, le timer se décrémente deux fois par seconde. Avec dix clics, dix fois plus vite. La variable booléenne `ChronoEnMarche` agit comme un verrou pour éviter cela.

**Convertir les secondes en MM :SS :** La division entière `Math.trunc(tempsInitial / 60)` donne les minutes. L'opérateur modulo `%` donne le reste de la division — ici, les secondes restantes après avoir retiré les minutes complètes. Le ternaire ajoute un zéro devant les secondes inférieures à 10 (:05 au lieu de :5).

#### 5.4.3 Pause, reprise et réinitialisation

```

1 btnPause.addEventListener('click', () => {
2     if(ChronoEnMarche){
3         btnPause.innerHTML = pause ? "Pause" : "Reprendre";
4         pause = !pause;
5     }
6 })
7
8 btnReset.addEventListener('click', () => {
9     clearInterval(timer);
10    ChronoEnMarche = false;
11    pause = false;
12    tempsInitial = T_INIT;
13    tempsDeRepos = T_REP;
14    nbDeCycles = 0;

```

```
15     btnPause.innerText = "Pause";  
16     affichageTravail.innerText = '25 : 00';  
17 })
```

Listing 5.3 – Contrôle complet du timer

**La pause ne stoppe pas l'intervalle :** Une approche naïve serait de faire `clearInterval(timer)` pour la pause et de recréer un intervalle pour la reprise. Mais cela complexifie la gestion et risque de créer des doublons. La solution ici est plus élégante : l'intervalle continue de tourner toutes les secondes, mais la condition `if(!pause)` l'empêche de faire quoi que ce soit. L'horloge tourne, mais elle ne décompte plus.

**L'inversion booléenne `pause = !pause` :** C'est l'idiome JavaScript pour inverser un booléen. Pas besoin de `if/else`, une seule instruction suffit.

**Le reset est une remise à zéro complète :** On arrête définitivement l'intervalle avec `clearInterval(timer)`, puis on remet toutes les variables à leur valeur initiale. Utiliser les constantes `T_INIT` et `T_REP` garantit qu'on repart exactement aux valeurs de départ, sans risque d'erreur de frappe.



#### Concepts clés du Pomodoro

**setInterval()** pour exécuter une fonction de façon répétée — **clearInterval()** seule façon d'arrêter un intervalle — **Variable booléen** comme verrou anti-double-démarrage — **Modulo %** pour extraire les secondes depuis un total — **Constantes** pour centraliser la configuration de l'appli

# Chapitre 6

## Gestion des Cookies

### 6.1 Vue d'ensemble

Ce projet explore les cookies navigateur : de petits fichiers texte que le navigateur stocke côté client pour conserver des informations entre les sessions. On peut créer un cookie, le lire et le supprimer. C'est une introduction à la persistance des données sans base de données.

### 6.2 Structure HTML

Deux boutons, trois champs (nom, valeur, date d'expiration) et une zone d'affichage. L'attribut `data-cookie` sur chaque bouton indique l'action à effectuer, ce qui permet à un seul gestionnaire JavaScript de gérer les deux boutons à la fois.

### 6.3 Mise en forme CSS

Des animations `@keyframes` `idle` et `bounce` donnent vie aux boutons quand ils sont inactifs, les faisant subtilement rebondir pour attirer l'attention. Cet effet “idle” rappelle les interfaces de jeux vidéo qui animent les éléments en attente d'interaction.

### 6.4 Logique JavaScript

#### 6.4.1 Un gestionnaire unifié pour plusieurs boutons

```
1 btns.forEach(btn => { btn.addEventListener('click', btnAction); })
2
3 function btnAction(e) {
4   let nvObj = {};
5   input.forEach(input => {
6     let attrName = input.getAttribute('name');
7     let attrValeur = attrName !== "cookieExpire"
8       ? input.value : input.valueAsDate;
9     nvObj[attrName] = attrValeur;
10  })
11  let description = e.target.getAttribute('data-cookie');
12  if(description === 'creer')
13    creerCookie(nvObj.cookieName, nvObj.cookieValue,
14              nvObj.cookieExpire);
15  else if(description === 'toutAfficher')
16    listeCookies();
17 }
```

Listing 6.1 – Gestionnaire centralisé par attribut `data-*`

**L'approche `data-*` pour le routage :** Plutôt que d'attacher une fonction différente à chaque bouton, on utilise un seul gestionnaire `btnAction`. L'attribut `data-cookie` sur chaque bouton indique au gestionnaire quelle action exécuter. C'est un schéma de “rou-

tage” simplifié : l’attribut joue le rôle d’instruction.

**Collecter tous les inputs dans un objet :** La boucle `forEach` sur les inputs construit un objet `nvObj` en lisant le `name` de chaque input comme clé. Cette approche est extensible : ajouter un nouveau champ au formulaire ne nécessite pas de modifier ce code de collecte.

**Traitement spécial pour la date :** Les inputs de type `date` ont deux propriétés : `.value` retourne une chaîne de caractères au format “YYYY-MM-DD”, tandis que `.valueAsDate` retourne un objet `Date` JavaScript. Pour manipuler la date (appeler `.toUTCString()` notamment), on a besoin de l’objet, pas de la chaîne.

#### 6.4.2 Créer un cookie avec encodage

```

1 function creerCookie(name, value, exp) {
2     let cookies = document.cookie.split(';');
3     cookies.forEach(cookie => {
4         cookie = cookie.trim();
5         let formatCookie = cookie.split('=');
6         if(formatCookie[0] === encodeURIComponent(name))
7             dejaFait = true;
8     })
9     if(dejaFait){
10         infoTxt.innerHTML = "Nom d j utilis ";
11         dejaFait = false; return;
12     }
13     if(name.length === 0){
14         infoTxt.innerHTML = "Nom requis."; return;
15     }
16
17     document.cookie =
18         `${encodeURIComponent(name)}= ` +
19         `${encodeURIComponent(value)}; ` +
20         `expires=${exp.toUTCString()} `;
21
22     let info = document.createElement('li');
23     info.innerHTML = `Cookie "${name}" cr avec succ s.`;
24     affichage.appendChild(info);
25     setTimeout(() => { info.remove(); }, 1500);
26 }

```

Listing 6.2 – Vérification et création d’un cookie

**La particularité de `document.cookie` :** C’est une propriété très spéciale. En lecture, elle retourne tous les cookies concaténés en une seule chaîne (“nom1=val1 ; nom2=val2”). En écriture, elle n’écrase pas tous les cookies — elle ne crée ou ne modifie qu’un seul cookie à la fois. C’est une interface API déguisée en propriété simple.

**Pourquoi `encodeURIComponent()` est obligatoire :** Les cookies ne tolèrent pas certains caractères dans leur nom ou valeur : les espaces, les accents, le signe = (qui sépare nom et valeur), le point-virgule (qui sépare les cookies). `encodeURIComponent()` convertit ces caractères en séquences sûres comme `%20` (espace) ou `%3D` (signe égal). Sans cet encodage, le cookie serait corrompu.

**La vérification d’unicité compare les noms encodés :** Les noms sont stockés dans le navigateur sous leur forme encodée. Pour comparer correctement, on encode aussi le

nom saisi avant la comparaison. Comparer “café” avec “caf%C3%A9” ne donnerait jamais de correspondance.

### 6.4.3 Lire et supprimer des cookies

```

1 function listeCookies() {
2     let cookies = document.cookie.split(';');
3     if(cookies.join('') === ''){
4         infoTxt.innerText = 'Aucun cookie'; return;
5     }
6     affichage.innerText = "";
7
8     cookies.forEach(cookie => {
9         cookie = cookie.trim();
10        let formatCookie = cookie.split('=');
11        let item = document.createElement('li');
12        item.innerText =
13            'Nom : ${decodeURIComponent(formatCookie[0])} | ' +
14            'Valeur : ${decodeURIComponent(formatCookie[1])}';
15        affichage.appendChild(item);
16
17        item.addEventListener('click', () => {
18            document.cookie =
19                `${formatCookie[0]}=; expires=${new Date(0)}';
20            item.innerText = 'Cookie supprimé.';
21            setTimeout(() => { item.remove(); }, 1000);
22        })
23    })
24 }
```

Listing 6.3 – Affichage et suppression interactive

**Supprimer un cookie : le paradoxe de l’expiration** : On ne peut pas “effacer” un cookie directement. La seule façon de le supprimer est de le réécrire avec une date d’expiration située dans le passé. `new Date(0)` crée un objet `Date` correspondant au 1er janvier 1970 à 00 :00 :00 UTC. Le navigateur constate que le cookie est “expiré depuis plus de 50 ans” et le supprime automatiquement.

**`decodeURIComponent()` pour l’affichage** : C’est le symétrique exact de `encodeURIComponent()`. Il retranscrit les séquences `%XX` en caractères lisibles. Sans lui, l’utilisateur verrait “caf%C3%A9” au lieu de “café” dans la liste.

**Le message temporaire avec `setTimeout`** : Après suppression, on change le texte de l’élément en “Cookie supprimé.” et on programme sa disparition après 1 seconde. Ce schéma de notification temporaire est très courant dans les interfaces modernes (les “toast notifications”).



#### Concepts clés des Cookies

**`document.cookie`** propriété en lecture et écriture avec sémantiques différentes — **`encodeURIComponent()`** et **`decodeURIComponent()`** pour les caractères spéciaux — **Suppression = réécriture avec date passée** — **`data-*`** pour centraliser les gestionnaires d’événements — **`setTimeout()`** pour les notifications temporaires

## Conclusion

Au fil de ces six projets, une progression claire s’est dessinée — à la fois dans la complexité des concepts abordés et dans la maturité de l’approche.

En **HTML**, chaque projet a confirmé que la sémantique n’est pas décorative : les attributs `data-*`, les types d’inputs, la structure des formulaires sont des points d’appui indispensables pour JavaScript. Un HTML bien conçu rend le JavaScript plus simple et plus lisible.

En **CSS**, les pseudo-classes (`:hover`, `:focus`, `:checked`, `:focus-within`), les animations `@keyframes` et CSS Grid ont montré qu’une grande partie de l’interactivité peut être réalisée sans une seule ligne de JavaScript. CSS est souvent sous-estimé.

En **JavaScript**, la progression a été la plus marquante :

Projet	Concept central	Nouveauté
Quiz	Formulaires, DOM	<code>preventDefault</code> , <code>classList</code>
Météo	API externe	<code>fetch</code> , Promises
To-Do	DOM dynamique	<code>createElement</code> , <code>filter</code>
Pokédex	Parallélisme	<code>Promise.all</code> , scroll infini
Pomodoro	Temporalité	<code>setInterval</code> , états booléens
Cookies	Persistance	<code>document.cookie</code> , encodage

La méthode acquise — structurer d’abord, styliser ensuite, logique en dernier — est plus qu’une habitude de travail : c’est une discipline qui pousse à séparer les responsabilités, à rendre le code modulaire, et à anticiper les cas d’erreur avant qu’ils ne surviennent.

La prochaine étape naturelle sera l’exploration du `localStorage` pour une persistance plus puissante que les cookies, l’apprentissage de `async/await` comme syntaxe moderne des Promises, et la découverte de frameworks comme React ou Vue.js — qui reposent entièrement sur les concepts pratiqués ici.