Script de Soutenance - 7 minutes

Projet : Arbres Remarquables de Paris

[0:00 - 0:45] INTRODUCTION

Bonjour à tous. Je vais vous présenter mon projet de dataviz réalisé dans le cadre de ma formation de développeuse web fullstack.

Le sujet : Les arbres remarquables de Paris. Ce sont des arbres exceptionnels par leur âge, leurs dimensions ou leur histoire, qui représentent un patrimoine naturel et culturel à préserver.

L'objectif technique : Créer un site web qui récupère des données depuis l'API OpenData de Paris et les affiche de manière interactive avec un système de recherche et des filtres.

Les technologies utilisées : HTML5 pour la structure, CSS3 pour le design, et JavaScript ES6 pour les fonctionnalités dynamiques.

Voici le résultat. [*MONTRER LE SITE*]

[0:45 - 1:30] RÉPONSE AU CAHIER DES CHARGES

Le cahier des charges demandait plusieurs fonctionnalités que j'ai toutes implémentées :

Premièrement, la récupération des données via fetch depuis l'API OpenData Paris. J'ai créé une fonction asynchrone qui récupère 20 arbres remarquables.

Deuxièmement, l'affichage dynamique des données sous forme de cartes. Chaque carte contient une photo, le nom de l'arbre, son nom usuel, son nom latin, et un lien pour télécharger la fiche PDF complète.

Troisièmement, un système de recherche multi-champs qui filtre en temps réel pendant que l'utilisateur tape. La recherche est insensible à la casse et cherche dans quatre champs différents : le nom usuel, le nom latin, le nom français et la description.

Quatrièmement, un bouton "Voir plus" qui permet d'afficher ou de masquer la description complète de chaque arbre.

Et enfin, un design entièrement responsive qui s'adapte du mobile au desktop.

Concernant la pagination : avec seulement 20 arbres, j'ai choisi de privilégier une recherche instantanée plutôt qu'une pagination qui aurait complexifié l'expérience utilisateur sans réel bénéfice.

[1:30 - 2:15] STRUCTURE HTML ET ACCESSIBILITÉ

Commençons par la structure HTML.

J'ai utilisé des balises sémantiques : un `header` pour le titre et la recherche, un `main` pour le contenu principal avec les cartes, et un `footer` pour le pied de page.

Points d'accessibilité que j'ai intégrés :

- L'attribut `lang="fr"` sur la balise HTML pour les lecteurs d'écran
- Un `aria-label` sur l'input de recherche pour décrire sa fonction
- Des attributs `alt` dynamiques sur toutes les images, générés en JavaScript avec le nom de l'arbre
- Des liens externes avec `target="_blank"` vers l'association qui labellise ces arbres

La structure est simple mais efficace : elle permet aux moteurs de recherche et aux technologies d'assistance de bien comprendre le contenu.

[2:15 - 3:30] CSS : DESIGN RESPONSIVE ET MODERNE

Passons au CSS. J'ai fait plusieurs choix techniques importants.

Premier choix : Une approche mobile-first. Par défaut, les cartes s'affichent en une colonne sur mobile. Puis, avec une media query à 1024 pixels, la mise en page passe à trois colonnes sur desktop avec un layout en grid où le header est à gauche et les cartes à droite.

Deuxième choix : L'utilisation de variables CSS. J'ai créé une palette de couleurs naturelles avec des verts forêt et des bruns écorce, stockées dans des custom properties. Cela facilite la maintenance : si je veux changer une couleur, je la modifie à un seul endroit.

Troisième choix : La combinaison de Grid et Flexbox. J'utilise CSS Grid pour la mise en page générale et l'organisation des cartes en colonnes, et Flexbox pour l'alignement des éléments à l'intérieur de chaque carte. Chaque technologie est utilisée pour ce qu'elle fait de mieux : Grid pour les layouts 2D, Flexbox pour les alignements 1D.

Quatrième choix : Des transitions CSS pour améliorer l'expérience utilisateur. Quand on survole une carte, elle se soulève légèrement avec un effet de transform et son ombre s'accentue. De même pour les boutons et la barre de recherche au focus.

J'ai également fait attention aux contrastes de couleurs pour respecter les normes d'accessibilité WCAG.

[3:30 - 5:30] JAVASCRIPT : LES PARTIES TECHNIQUES

Maintenant, la partie JavaScript, qui est le cœur du projet.

La récupération des données

J'ai créé une fonction asynchrone `getData()` qui utilise l'API Fetch.

```javascript
async function getData() {
  try {
    const response = await fetch("URL_API");
    const apiData = await response.json();
    allTrees = apiData.results;
    displayTrees(allTrees);
  } catch(error) {
    console.log(error);
  }
}
```

Pourquoi asynchrone ? Parce que l'appel à l'API prend du temps. Avec `async/await`, mon code attend la réponse sans bloquer le reste de l'application. C'est la syntaxe moderne pour gérer les promesses, plus lisible que les `.then()`.

Comment fonctionne l'API ? Mon navigateur envoie une requête HTTP GET à l'API OpenData Paris, qui me renvoie un objet JSON contenant un tableau `results` avec les 20 arbres. Chaque arbre est un objet avec des propriétés comme `arbres_libellefrancais`, `com_nom_usuel`, `com_url_photo1`, etc.

J'ai ajouté un `try/catch` pour gérer les erreurs : si l'API ne répond pas, l'erreur est capturée et affichée dans la console.

Important : je stocke toutes les données dans une variable globale `allTrees`. Pourquoi ? Pour éviter de rappeler l'API à chaque recherche. Les données sont chargées une seule fois au démarrage, puis je filtre côté client.

L'affichage dynamique

La fonction `displayTrees()` crée les cartes HTML dynamiquement en JavaScript.

Elle commence par vider le conteneur avec `innerHTML = ""` pour repartir de zéro. Ensuite, elle vérifie s'il y a des arbres à afficher. Si le tableau est vide, elle affiche un message "Aucun arbre trouvé".

Puis, pour chaque arbre, elle crée une série d'éléments HTML :

- Un `div` avec la classe `arbre-card`
- Une image avec `createElement("img")`, en assignant la source et un texte alternatif
- Des titres h2, h3, h4 pour les différents noms
- Un paragraphe pour la description, caché par défaut avec `style.display = "none"`
- Et un bouton "Voir plus"

Tous ces éléments sont ajoutés à la carte avec `appendChild()`, et la carte est ajoutée au conteneur principal.

C'est ici que je démontre ma maîtrise de la manipulation du DOM : créer des balises HTML via JavaScript, leur ajouter du contenu, et les insérer dans la page.

Le bouton "Voir plus/Voir moins"

Pour chaque carte, j'ajoute un écouteur d'événement sur le bouton :

```javascript
buttonCache.addEventListener("click", function() {
  if (description.style.display === "none") {
    description.style.display = "block";
    buttonCache.innerHTML = "Voir moins";
  } else {
    description.style.display = "none";
    buttonCache.innerHTML = "Voir plus";
  }
});
```

Le principe : je vérifie l'état actuel de la description. Si elle est cachée, je l'affiche et change le texte du bouton en "Voir moins". Sinon, je fais l'inverse. C'est un toggle, un système de bascule.

Cela démontre ma capacité à manipuler les événements utilisateur et à modifier le CSS via JavaScript.

Le système de recherche

C'est la partie la plus complexe. La fonction `searchTrees()` prend en paramètre le texte tapé par l'utilisateur.

Première étape : normaliser la saisie avec `toLowerCase()` et `trim()` pour rendre la recherche insensible à la casse et supprimer les espaces inutiles.

Deuxième étape : si la recherche est vide, j'affiche tous les arbres.

Troisième étape : sinon, je filtre. J'utilise deux boucles imbriquées :

- La première parcourt tous les arbres
- La deuxième parcourt les quatre champs de recherche : nom usuel, nom latin, nom français, description

Pour chaque arbre, je vérifie si le texte recherché est présent dans au moins un des quatre champs. Avant de chercher, je vérifie que la valeur existe et que c'est bien une chaîne de caractères avec `typeof value === 'string'`. Cette vérification évite les erreurs si une donnée est manquante.

Si je trouve une correspondance, j'ajoute l'arbre au tableau `filteredTrees` et je sors de la boucle interne avec `break` pour optimiser la performance.

Enfin, j'appelle `displayTrees()` avec uniquement les arbres filtrés.

Les événements : J'ai ajouté deux écouteurs sur la barre de recherche.

- L'événement `input` pour une recherche en temps réel pendant que l'utilisateur tape
- L'événement `keypress` pour valider avec la touche Entrée, pour ceux qui préfèrent cette approche traditionnelle

---

[5:30 - 6:15] COMPÉTENCES DÉMONTRÉES

Ce projet démontre toutes les compétences du cahier des charges :

Pour le HTML et CSS :

- Je sais connecter mon HTML avec mon JavaScript
- Je sais utiliser le CSS avec variables, transitions et animations
- Je sais quand utiliser Grid et Flexbox selon le besoin
- Je sais rendre ma page responsive avec des media queries

Pour le JavaScript :

- Je sais créer des balises HTML via JavaScript avec `createElement()`
- Je sais ajouter des informations dans ces balises avec `innerHTML`, `src`, `alt`
- Je sais afficher les données récupérées par l'API
- Je comprends comment fonctionne une API REST et le format JSON
- Je sais manipuler un événement avec `addEventListener()`
- Je sais changer le CSS via JavaScript avec `style.display`
- Je sais fetcher des données avec `fetch()`
- Je comprends ce qu'est une fonction asynchrone et quand l'utiliser

Compétences avancées :

- Je gère les erreurs avec `try/catch`
- Mon code est organisé en fonctions claires et réutilisables
- Je gère les cas limites comme l'absence de résultats

[6:15 - 6:50] POINTS D'AMÉLIORATION

Maintenant, parlons des améliorations possibles, parce qu'un bon développeur identifie toujours ce qui peut être optimisé.

Premièrement, la gestion d'erreurs pourrait être plus visible pour l'utilisateur. Actuellement, si l'API ne répond pas, l'erreur apparaît juste dans la console. Je pourrais afficher un message à l'écran : "Erreur de chargement, veuillez réessayer".

Deuxièmement, je pourrais ajouter un loader pendant le chargement des données, pour que l'utilisateur sache que quelque chose se passe.

Troisièmement, je pourrais mieux gérer les données manquantes. Par exemple, si un arbre n'a pas de description, afficher "Aucune description disponible" au lieu de "undefined".

Quatrièmement, si j'augmentais le nombre d'arbres récupérés, je devrais implémenter une vraie pagination avec les paramètres `limit` et `offset` de l'API, et des boutons "Suivant/Précédent".

Enfin, je pourrais ajouter des fonctionnalités supplémentaires comme une carte interactive pour localiser les arbres, ou des filtres par arrondissement ou par espèce.

[6:50 - 7:00] CONCLUSION

Pour conclure, ce projet m'a permis de mettre en pratique l'ensemble du cycle de développement d'une application web moderne : de la conception à la réalisation, en passant par la récupération de données réelles via une API.

J'ai appris à structurer mon code de manière claire, à gérer l'asynchrone, et à créer une interface utilisateur interactive et responsive.

C'est un projet que je peux montrer fièrement dans mon portfolio, et qui démontre ma capacité à créer des applications web fonctionnelles et accessibles.

Merci de votre attention. Je suis prête à répondre à vos questions.

📝 NOTES POUR LA PRÉSENTATION

À FAIRE pendant la présentation :

- Montrer le site dès l'intro
- Faire une démo de recherche en tapant "chêne"
- Cliquer sur "Voir plus" pour montrer le toggle
- Redimensionner la fenêtre pour montrer le responsive
- Ouvrir la console si on vous pose des questions techniques

TIMING :

- Introduction : 45 secondes
- Cahier des charges : 45 secondes
- HTML/Accessibilité : 45 secondes
- CSS : 1 min 15
- JavaScript : 2 minutes (la partie la plus importante)
- Compétences : 45 secondes
- Améliorations : 35 secondes
- Conclusion : 10 secondes

QUESTIONS POSSIBLES :

- "Pourquoi pas de pagination ?" → "Avec 20 arbres, la recherche en temps réel est plus adaptée"
- "Comment gérez-vous les erreurs ?" → "Try/catch + message console, amélioration possible avec message utilisateur"
- "Pourquoi stocker les données ?" → "Éviter de rappeler l'API, meilleure performance"
- "Grid ou Flexbox ?" → "Grid pour layout 2D, Flexbox pour alignement 1D"