Guide de révision en développement web

HTML, CSS et JavaScript

Support d'auto-apprentissage

Table des matières

1 Introduction

Ce document est un guide de révision en auto-apprentissage conçu pour les étudiants qui apprennent le développement web. Il couvre le HTML, le CSS et le JavaScript, des concepts fondamentaux jusqu'à des notions plus avancées. Chaque sous-section contient une courte explication théorique et un exemple de code concret afin de relier l'idée à une pratique réelle. Le document se termine par un mini-projet combinant les trois technologies.

2 HTML – HyperText Markup Language

2.1 Qu'est-ce que le HTML ?

Le HTML (HyperText Markup Language) est un langage de balisage utilisé pour structurer les pages web. Il décrit le contenu et son sens (titres, paragraphes, liens, images, formulaires). Le HTML n'implémente pas la logique ; son rôle est de fournir une structure claire que les navigateurs peuvent afficher et que les technologies d'assistance peuvent interpréter.

Exemple (une petite page structurée) :

```
<h1>My Profile</h1>
<p>Hello! My name is Renan and I am learning web development.</p>
```

2.2 Structure de base d'un document HTML

Un document HTML suit une structure standard. `<!DOCTYPE html>` déclare le HTML5. La balise `<head>` contient les métadonnées (titre, encodage, liens vers le CSS). La balise `<body>` contient tout le contenu visible.

Exemple (document minimal complet) :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>My First Page</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

2.3 Éléments de texte

Les titres (`h1`–`h6`) créent une hiérarchie. Les paragraphes (`p`) contiennent du texte normal. Utilisez `strong` et `em` pour l'emphase sémantique (le sens), et pas uniquement pour le style visuel.

Exemple (titres + emphase) :

```
<h2>Course Goals</h2>
<p>Learn <strong>HTML</strong> structure and <em>good practices</em
   >.</p>
```

2.4 Liens et images

Les liens (`a`) permettent aux utilisateurs de naviguer grâce à l'attribut `href`. Les images (`img`) utilisent `src` pour charger un fichier et `alt` pour le décrire (accessibilité et SEO).

Exemple (lien + image) :

```
<a href="https://example.com">Visit Example</a>
<img src="profile.png" alt="Profile picture of the user" />
```

2.5 Listes

Les listes aident à organiser l'information. Les listes non ordonnées (`ul`) utilisent des puces. Les listes ordonnées (`ol`) utilisent des numéros. Chaque élément est placé dans une balise `li`.

Exemple (liste non ordonnée) :

```
<ul>
  <li>HTML fundamentals</li>
  <li>CSS layout</li>
  <li>JavaScript basics</li>
</ul>
```

2.6 Formulaires et champs de saisie

Les formulaires collectent des données utilisateur. Une balise `form` regroupe les champs. Les champs peuvent être validés via des attributs HTML comme `required`, et les labels améliorent l'ergonomie et l'accessibilité.

Exemple (formulaire simple) :

```
<form>
  <label for="email">Email:</label>
  <input id="email" type="email" required />
```

```
  <button type="submit">Submit</button>
</form>
```

2.7 HTML sémantique

Les balises sémantiques décrivent le rôle des sections (`header`, `nav`, `main`, `section`, `footer`). Cela améliore la lisibilité, l'accessibilité et le SEO.

Exemple (structure sémantique) :

```
<header>
  <h1>My Website</h1>
</header>
<main>
  <section>
    <h2>About</h2>
    <p>This page uses semantic HTML.</p>
  </section>
</main>
<footer>
  <p>   2026</p>
</footer>
```

2.8 Bonnes pratiques HTML

Utilisez une hiérarchie de titres claire, des éléments sémantiques, un texte significatif et des attributs d'accessibilité (comme `alt`). Gardez le HTML centré sur la structure ; évitez de mélanger beaucoup de styles directement dans le HTML.

Exemple (bonne pratique : label + input + required) :

```
<label for="name">Full name</label>
<input id="name" type="text" required />
```

3 CSS – Cascading Style Sheets

3.1 Qu'est-ce que le CSS ?

Le CSS contrôle la présentation visuelle du HTML : couleurs, typographie, espacements, mise en page et responsivité. Séparer le HTML (structure) du CSS (style) améliore la maintenabilité et le travail en équipe.

Exemple (style de base) :

```
body {
  font-family: Arial, sans-serif;
  line-height: 1.5;
}
```

3.2 Comment fonctionne le CSS

Le CSS utilise des règles : un sélecteur cible des éléments ; des déclarations appliquent des propriétés et des valeurs. La cascade détermine quelle règle l'emporte lorsque plusieurs règles s'appliquent.

Exemple (sélecteur + déclarations) :

```
p {
  color: #333;
  font-size: 16px;
}
```

3.3 Sélecteurs

Les sélecteurs ciblent les éléments. Les classes sont réutilisables ; les IDs sont uniques. La spécificité influence la règle appliquée.

Exemple (élément, classe et id) :

```
h1 { margin-bottom: 8px; }
.card { padding: 16px; border: 1px solid #ddd; }
#main-title { text-transform: uppercase; }
```

3.4 Modèle de boîte (Box Model)

Chaque élément est une boîte : contenu, padding, bordure, marge. Le modèle de boîte est essentiel pour l'espacement et la mise en page. Utiliser `box-sizing: border-box;` rend souvent le dimensionnement plus prévisible.

Exemple (box model + border-box) :

```
* { box-sizing: border-box; }

.box {
  width: 300px;
  padding: 16px;
  border: 2px solid #000;
  margin: 12px;
```

```
}
```

3.5 Flexbox

Flexbox est un système de mise en page unidimensionnel (ligne ou colonne). Il est utile pour l'alignement et la répartition de l'espace.

Exemple (centrer du contenu avec Flexbox) :

```
.container {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 200px;
}
```

3.6 Grid Layout

CSS Grid est un système de mise en page bidimensionnel (lignes et colonnes). Il est puissant pour la structure de page et les tableaux de bord.

Exemple (grille 3 colonnes) :

```
.grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 12px;
}
```

3.7 Design responsive

Le design responsive adapte la mise en page à différentes tailles d'écran. Les media queries appliquent des règles selon la largeur.

Exemple (règle mobile) :

```
@media (max-width: 768px) {
  .grid {
    grid-template-columns: 1fr;
  }
}
```

3.8 Animations et transitions

Les transitions rendent les changements d'état plus fluides (hover, focus). Les animations (keyframes) créent des effets en plusieurs étapes.

Exemple (transition au survol) :

```
button {
  transition: transform 0.2s ease;
}
button:hover {
  transform: scale(1.05);
}
```

3.9 Bonnes pratiques CSS

Privilégiez des classes réutilisables, une nomenclature cohérente et des feuilles de style externes. Utilisez des unités responsives (comme `rem` et les pourcentages) et évitez des sélecteurs trop spécifiques, difficiles à surcharger.

Exemple (classe utilitaire réutilisable) :

```
.text-center { text-align: center; }
.mt-16 { margin-top: 16px; }
```

4 JavaScript

4.1 Qu'est-ce que JavaScript ?

JavaScript est un langage de programmation qui ajoute de l'interactivité et de la logique aux pages web. Il peut mettre à jour le DOM, gérer des événements, valider des formulaires et communiquer avec des serveurs via des API.

Exemple (affichage simple) :

```
console.log("JavaScript is running!");
```

4.2 Variables et types de données

Les variables stockent des valeurs. Préférez `const` par défaut, et utilisez `let` lorsque la réaffectation est nécessaire. Les types courants incluent string, number, boolean, array et object.

Exemple (variables + types) :

```
const name = "Renan";
let age = 20;
const skills = ["HTML", "CSS", "JS"];
const user = { name: "Renan", age: 20 };
```

4.3 Conditions et boucles

Les conditions aident votre programme à prendre des décisions. Les boucles répètent des actions, utiles pour traiter des listes de valeurs.

Exemple (if + boucle) :

```
const score = 75;

if (score >= 60) {
  console.log("Passed");
} else {
  console.log("Failed");
}

for (let i = 1; i <= 3; i++) {
  console.log("Step " + i);
}
```

4.4 Fonctions

Les fonctions regroupent une logique réutilisable. Les fonctions fléchées (arrow functions) sont une syntaxe moderne et concise, très utilisée en développement web.

Exemple (fonction + fonction fléchée) :

```
function sum(a, b) {
  return a + b;
}

const multiply = (a, b) => a * b;
```

4.5 Tableaux et objets

Les tableaux stockent des collections ordonnées ; les objets stockent des données structurées en paires clé-valeur. Ils sont essentiels dans les applications réelles, car la plupart des données sont stockées sous ces formats.

Exemple (méthodes de tableau + accès objet) :

```
const students = ["Anna", "Bob", "Chris"];
students.push("Diana");
const upper = students.map(s => s.toUpperCase());

const profile = { name: "Anna", age: 22 };
console.log(profile.name);
```

4.6 Manipulation du DOM

Le DOM représente la structure de la page. JavaScript peut sélectionner des éléments et les mettre à jour dynamiquement.

Exemple (sélection + modification de texte) :

```
const title = document.querySelector("h1");
title.textContent = "Updated Title";
```

4.7 Événements

Les événements sont des actions utilisateur (click, input, submit). Les écouteurs d'événements permettent à votre code de réagir à ces actions.

Exemple (événement click) :

```
const btn = document.querySelector("#myButton");

btn.addEventListener("click", () => {
  alert("Button clicked!");
});
```

4.8 JavaScript asynchrone

Le code asynchrone permet d'exécuter des tâches (comme des requêtes réseau) sans bloquer la page. Les Promises et `async/await` sont les moyens standards pour gérer la logique asynchrone.

Exemple (bases de async/await) :

```
async function loadData() {
  const response = await fetch("https://jsonplaceholder.typicode.
     com/posts");
  const data = await response.json();
  console.log(data);
}
```

```
loadData();
```

4.9 Utilisation d'une fausse API

Les fausses API simulent des serveurs réels. JSONPlaceholder est populaire pour l'apprentissage. Vous pouvez vous entraîner à récupérer (GET) et créer (POST) des données.

Exemple (GET + affichage des titres) :

```
async function loadPosts() {
  const res = await fetch("https://jsonplaceholder.typicode.com/
     posts?_limit=5");
  const posts = await res.json();

  const list = document.querySelector("#postList");
  list.innerHTML = posts.map(p => `<li>${p.title}</li>`).join("");
}

loadPosts();
```

4.10 Bonnes pratiques JavaScript

Écrivez un code lisible : utilisez des noms explicites, de petites fonctions et un formatage cohérent. Gérez les erreurs dans le code asynchrone pour éviter des échecs silencieux.

Exemple (try/catch avec fetch) :

```
async function safeLoad() {
  try {
    const res = await fetch("https://jsonplaceholder.typicode.com/
       posts/1");
    if (!res.ok) throw new Error("Request failed");
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error("Error:", err.message);
  }
}
safeLoad();
```

5 Mini-projet final

5.1 Titre du projet : Tableau de bord utilisateur simple

Objectif : Créer une petite application web en utilisant HTML, CSS et JavaScript.

5.2 Exigences

— **HTML :** Utiliser une structure sémantique avec un en-tête, une section principale et un pied de page.

— **CSS :** Créer une mise en page responsive (Flexbox ou Grid). Styliser un composant de type carte et des boutons.

— **JavaScript :** Gérer un clic sur un bouton pour récupérer des données depuis une fausse API et les afficher dynamiquement.

5.3 Exemple de départ (structure minimale)

HTML :

```
<header><h1>User Dashboard</h1></header>
<main>
  <button id="loadBtn">Load Posts</button>
  <ul id="postList"></ul>
</main>
<footer><p>Made for revision</p></footer>
```

CSS :

```
main { max-width: 700px; margin: 0 auto; padding: 16px; }
button { padding: 10px 14px; border: 1px solid #333; cursor:
   pointer; }
```

JavaScript :

```
document.querySelector("#loadBtn").addEventListener("click", async
   () => {
  const res = await fetch("https://jsonplaceholder.typicode.com/
     posts?_limit=5");
  const posts = await res.json();
  document.querySelector("#postList").innerHTML =
    posts.map(p => `<li>${p.title}</li>`).join("");
});
```

5.4 Idées bonus

— Ajouter un message de chargement pendant la récupération des données
— Afficher un message d'erreur si la requête échoue
— Ajouter un champ de recherche pour filtrer les posts par titre

Exercice – Créer une application Pokédex avec une API REST (PokeAPI)

Des captures d'écran de l'application vous sont fournies (référence de l'interface). Votre objectif est de **reproduire le comportement de l'application** en consommant la **PokeAPI** et en manipulant le DOM avec JavaScript.

Contraintes

— Utiliser **JavaScript** (aucun framework).
— Utiliser `fetch` avec `async/await`.
— Gérer les états de chargement et les erreurs de l'API.

API à utiliser

URL de base :
— `https://pokeapi.co/api/v2`
Endpoint requis :
— **GET** `/pokemon/{identifier}`
Où `identifier` peut être le **nom** du Pokémon (en minuscules) ou son **ID**.

Fonctionnalités requises (basées sur les captures d'écran)

1. **Onglet Recherche**
   — Un champ de saisie pour rechercher un Pokémon par **nom ou ID**.
   — Un bouton pour lancer la recherche (la touche Entrée est autorisée).
   — Afficher une carte Pokémon contenant au minimum : **ID, nom, image, types, taille et poids**.
   — Afficher un message d'erreur si le Pokémon n'est pas trouvé.
2. **Onglet Aléatoire**
   — Un bouton qui charge un **Pokémon aléatoire** (IDs de 1 à 898).
   — Afficher la même carte Pokémon que dans l'onglet Recherche.
   — Afficher un indicateur de chargement pendant la requête.

Pokemon Search

Search | Random

Search by name or ID (e.g., Pikachu, 25)

Search

Try: Pikachu, Charizard, 1-898

Pokemon Search

Search | Random

25

Search

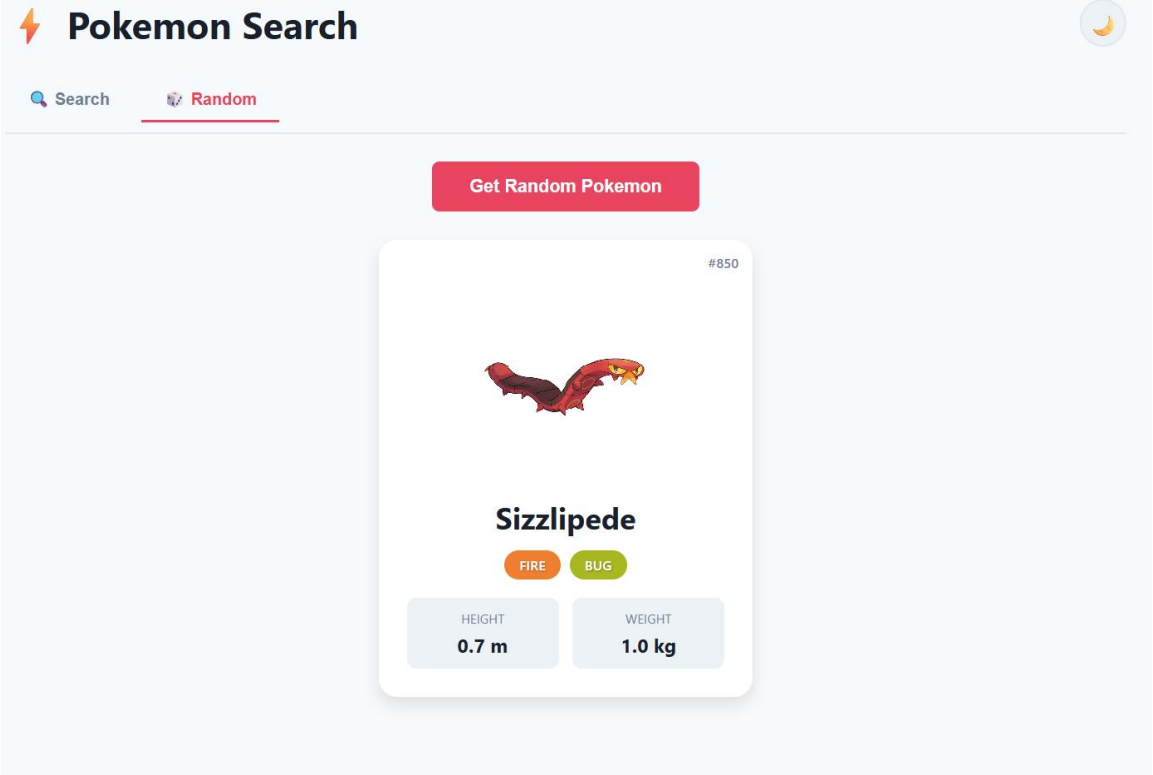
Try: Pikachu, Charizard, 1-898

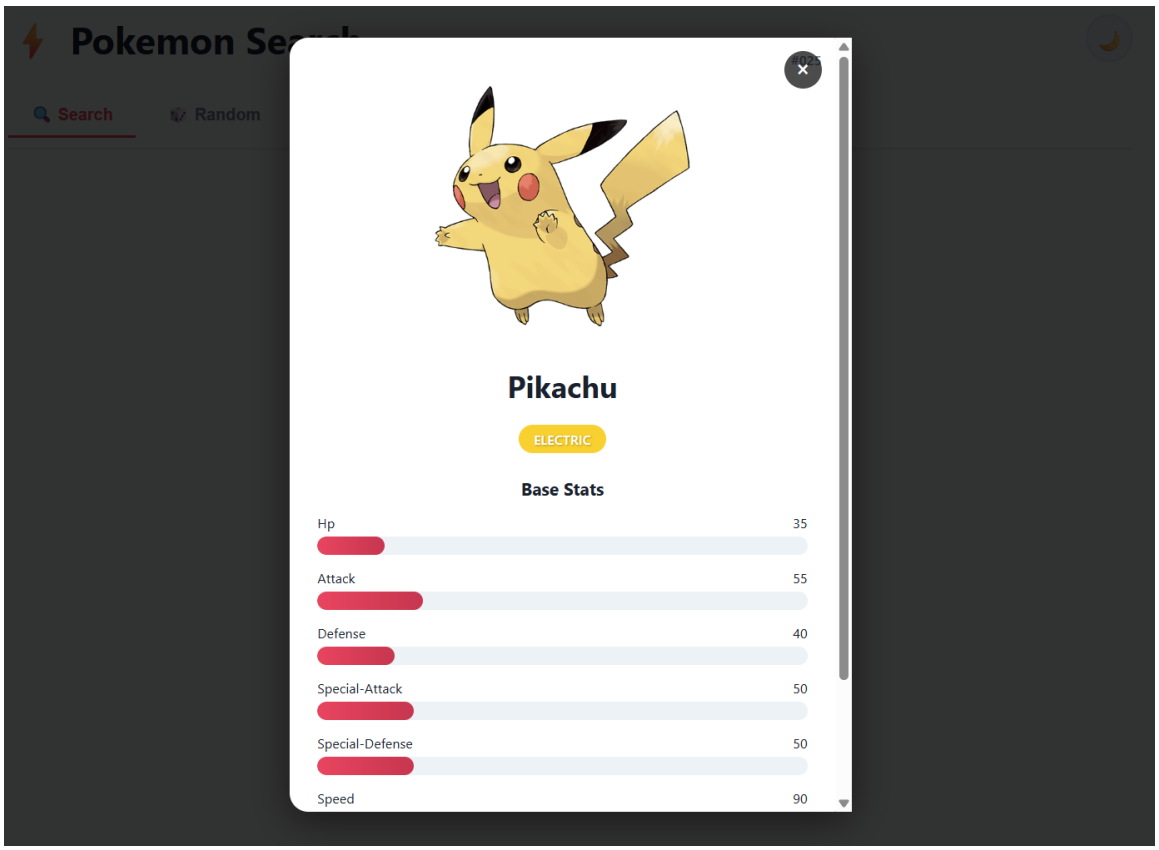
#025

Pikachu

ELECTRIC

HEIGHT	WEIGHT
0.4 m	6.0 kg


Pokemon Search
Search
Random
Get Random Pokemon
#850
Sizzlipede
FIRE
BUG
HEIGHT
0.7 m
WEIGHT
1.0 kg


Search
Random
Pikachu
ELECTRIC
Base Stats
Hp 35
Attack 55
Defense 40
Special-Attack 50
Special-Defense 50
Speed 90