**Séances 1 à 8 : Le Front-End Moderne en Détail avec Code**

**Séances 1 – Fondations du front-end moderne (HTML, CSS, JS)**

**Concepts Théoriques**

* **HTML5 : Structure et Sémantique** : L'épine dorsale de toute page web. On insiste sur les **balises sémantiques** (<header>, <main>, <section>, <footer>, <nav>, <article>) qui améliorent l'accessibilité et le SEO.
* **CSS3 : Flexbox, Grid, Responsive Design** :
  + **Flexbox** pour l'alignement unidimensionnel (lignes ou colonnes).
  + **Grid** pour la mise en page bidimensionnelle (lignes et colonnes), idéal pour la structure globale.
  + **Responsive Design** et les **Media Queries** pour adapter l'affichage à différentes tailles d'écran.
* **JavaScript ES6+ : Variables, Fonctions, DOM** :
  + Variables modernes (let, const), fonctions fléchées (=>), et manipulation du **DOM (Document Object Model)** pour rendre la page interactive.

**Extraits de Code (Exemples)**

**HTML Sémantique :**

HTML

<header>

<h1>Mon Site Web</h1>

<nav>

<ul><li><a href="#home">Accueil</a></li></ul>

</nav>

</header>

<main>

<section>

<h2>À Propos</h2>

<p>Contenu principal...</p>

</section>

</main>

**CSS Flexbox pour Centrer :**

CSS

.container {

display: flex;

justify-content: center; /\* Centrage horizontal \*/

align-items: center; /\* Centrage vertical \*/

height: 100vh;

}

**JS (DOM Manipulation) :**

JavaScript

// Fonction fléchée ES6+

const changeText = (elementId, newText) => {

// Manipulation du DOM

const element = document.getElementById(elementId);

if (element) {

element.textContent = newText;

}

};

document.addEventListener('DOMContentLoaded', () => {

changeText('title', 'Bienvenue sur la page JS');

});

**Séances 2 – TypeScript, Node & outils modernes**

**Concepts Théoriques**

* **Installation & Outils** : Mise en place de l'environnement (Node.js, npm, VS Code, Git). **Node.js** pour exécuter JavaScript côté serveur et **npm** pour gérer les paquets.
* **Introduction à TypeScript (TS)** : Un sur-ensemble de JavaScript qui ajoute le **typage statique**. Cela permet de détecter des erreurs plus tôt (à la compilation plutôt qu'à l'exécution) et d'améliorer la maintenabilité.
  + Types de base (string, number, boolean), interfaces, et classes.
* **Gestion des Dépendances avec npm** : Utilisation de package.json et des commandes npm install, npm start, etc.

**Extraits de Code (Exemples)**

**TypeScript (Typage Statique) :**

TypeScript

// Déclaration de variables typées

let age: number = 30;

let nom: string = "Alice";

let estActif: boolean = true;

// Interface pour définir la structure d'un objet

interface Utilisateur {

id: number;

username: string;

}

// Fonction avec typage des arguments et du retour

const saluer = (user: Utilisateur): string => {

return `Bonjour, ${user.username} (ID: ${user.id})`;

};

const user1: Utilisateur = { id: 1, username: "Bob" };

console.log(saluer(user1));

**package.json (Dépendances) :**

JSON

{

"name": "mon-projet",

"version": "1.0.0",

"scripts": {

"start": "node dist/index.js",

"build": "tsc"

},

"dependencies": {

"axios": "^1.0.0"

},

"devDependencies": {

"typescript": "^5.0.0"

}

}

**Séances 3 – Découverte d’Angular**

**Concepts Théoriques**

* **Structure du Projet** : Organisation typique d'un projet Angular (modules, composants, *assets*).
* **Composants** : Les blocs de construction d'une application Angular. Chaque composant a un template (HTML), un style (CSS), et une classe (TypeScript) qui gère la logique.
* **Templates et Data Binding** :
  + **Interpolation** ({{ variable }}) : Afficher des données dans le template.
  + **Property Binding** ([property]="data") : Lier une propriété DOM à une variable.
  + **Event Binding** ((event)="handler()") : Répondre aux événements utilisateur.
  + **Two-Way Binding** ([(ngModel)]="data") : Synchronisation bidirectionnelle des données (nécessite FormsModule).
* **Services et Injection de Dépendances (ID)** : Les Services contiennent la logique métier ou le partage de données. L'ID est le mécanisme pour fournir ces services aux composants qui en ont besoin.

**Extraits de Code (Exemples)**

**Composant Angular (TypeScript) :**

TypeScript

import { Component } from '@angular/core';

@Component({

selector: 'app-hello', // Balise utilisée dans d'autres templates

templateUrl: './hello.component.html',

styleUrls: ['./hello.component.css']

})

export class HelloComponent {

titre: string = 'Bienvenue sur Angular';

compteur: number = 0;

incrementer() {

this.compteur++;

}

}

**Template (HTML) avec Data Binding :**

HTML

<h2>{{ titre }}</h2> <button (click)="incrementer()">Incrémenter</button> <p [innerText]="'Compteur: ' + compteur"></p> ```

\*\*Service Angular (TypeScript) :\*\*

```typescript

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root' // Rendre le service disponible partout

})

export class TacheService {

private taches: string[] = ['Apprendre Angular', 'Faire les courses'];

getTaches(): string[] {

return this.taches;

}

ajouterTache(tache: string) {

this.taches.push(tache);

}

}

**Séances 4 – Angular avancé**

**Concepts Théoriques**

* **Routing** : Navigation entre différentes vues/composants via l'URL. Configuration des routes dans le RouterModule.
* **Formulaires** : Utilisation des **Formulaires Réactifs** (plus puissants et scalables) ou des Formulaires Pilotés par le Template.
* **HTTP** : Interagir avec des API externes en utilisant le HttpClient d'Angular, gérant les requêtes GET/POST/PUT/DELETE.
* **Gestion d’État Simple** : Techniques pour gérer l'état de l'application (partage de données) au-delà des Services, comme l'utilisation de BehaviorSubject dans un service.

**Extraits de Code (Exemples)**

**Configuration du Routing :**

TypeScript

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { DetailComponent } from './detail/detail.component';

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'detail/:id', component: DetailComponent }, // Route avec paramètre

{ path: '\*\*', redirectTo: '' } // Wildcard pour les routes inconnues

];

// ... dans le module ...

// imports: [RouterModule.forRoot(routes)]

**Formulaire Réactif :**

TypeScript

// ... dans le composant ...

import { FormGroup, FormControl, Validators } from '@angular/forms';

tacheForm = new FormGroup({

nom: new FormControl('', Validators.required),

priorite: new FormControl('basse')

});

onSubmit() {

if (this.tacheForm.valid) {

console.log(this.tacheForm.value);

}

}

**Appel HTTP (Utilisation d'un Observable) :**

TypeScript

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

// ... dans un service ...

constructor(private http: HttpClient) {}

getTaches(): Observable<any[]> {

return this.http.get<any[]>('http://localhost:3000/api/taches');

}

**Séances 5 – Introduction à React**

**Concepts Théoriques**

* **JSX (JavaScript XML)** : Une extension syntaxique qui permet d'écrire des structures d'arborescence (similaires à HTML) directement dans le code JavaScript.
* **Virtual DOM (VDOM)** : Un concept clé de React. C'est une représentation en mémoire du DOM réel. React l'utilise pour optimiser les mises à jour en ne modifiant que les parties du DOM qui ont réellement changé, rendant l'application plus rapide.
* **Composants Fonctionnels** : Des fonctions JavaScript qui acceptent des propriétés (**props**) comme argument et retournent des éléments React (JSX). C'est le standard moderne de React.
* **Hooks de Base** :
  + useState : Pour ajouter un **état** interne à un composant fonctionnel.
  + useEffect : Pour gérer les **effets secondaires** (appels API, manipulation directe du DOM, *timers*, etc.) après le rendu.

**Extraits de Code (Exemples)**

**Composant Fonctionnel avec props (JSX) :**

JavaScript

// src/components/Bienvenue.js

const Bienvenue = (props) => {

return (

// JSX: mélange de JS ({}) et de balises HTML-like

<h1>Bonjour, {props.nom} !</h1>

);

};

export default Bienvenue;

**Utilisation de useState :**

JavaScript

import React, { useState } from 'react';

const Compteur = () => {

// Déclare une variable d'état 'compteur' et sa fonction de mise à jour 'setCompteur'

const [compteur, setCompteur] = useState(0);

return (

<div>

<p>Vous avez cliqué {compteur} fois</p>

{/\* Utilisation de la fonction de mise à jour \*/}

<button onClick={() => setCompteur(compteur + 1)}>

Cliquer ici

</button>

</div>

);

};

**Utilisation de useEffect :**

JavaScript

import React, { useState, useEffect } from 'react';

const DataFetcher = () => {

const [data, setData] = useState(null);

// Exécuté après le premier rendu ET à chaque mise à jour si le tableau de dépendances est omis

// Si tableau de dépendances vide ([]), exécuté seulement après le premier rendu (simule componentDidMount)

useEffect(() => {

fetch('https://api.example.com/data')

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

.then(setData);

return () => {

// Nettoyage (simule componentWillUnmount)

console.log('Composant démonté');

};

}, []); // [] = s'exécute une seule fois

return data ? <p>{data.message}</p> : <p>Chargement...</p>;

};

**Séances 6 – React avancé**

**Concepts Théoriques**

* **Routing avec React Router** : Utilisation de la librairie standard pour gérer la navigation dans l'application monopage (SPA) React, avec des composants comme <BrowserRouter>, <Routes>, et <Route>.
* **Gestion d’État avec Context** : Une solution native de React pour éviter le **"Prop Drilling"** (passer des *props* à travers de nombreux composants intermédiaires). Le **Context API** permet de rendre les données accessibles à tous les composants sans les passer explicitement à chaque niveau.
* **Appels API avec fetch ou Axios** : Mise en pratique des appels HTTP, souvent gérés dans le useEffect. **Axios** est une librairie populaire qui simplifie les requêtes HTTP par rapport à l'API native fetch.

**Extraits de Code (Exemples)**

**Routing avec React Router (v6+) :**

JavaScript

import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

import Home from './Home';

import About from './About';

const App = () => (

<BrowserRouter>

<nav>

<Link to="/">Accueil</Link> | <Link to="/a-propos">À Propos</Link>

</nav>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/a-propos" element={<About />} />

</Routes>

</BrowserRouter>

);

**Context API (Création et Utilisation) :**

JavaScript

// 1. Création du Context

import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();

// 2. Fournisseur (Provider)

export const ThemeProvider = ({ children }) => {

const [theme, setTheme] = useState('light');

return (

<ThemeContext.Provider value={{ theme, setTheme }}>

{children}

</ThemeContext.Provider>

);

};

// 3. Utilisation du Context dans un composant

const ThemeToggle = () => {

// Hook useContext pour accéder à la valeur

const { theme, setTheme } = useContext(ThemeContext);

return (

<button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>

Mode actuel : {theme}

</button>

);

};

**Séances 7 – Bonnes pratiques & UI/UX**

**Concepts Théoriques**

* **UI Libraries (Material, Tailwind, Chakra)** : Utilisation de frameworks CSS/composants pour accélérer le développement et assurer un design cohérent.
  + **Tailwind CSS** est un framework "utility-first" qui permet de styliser directement dans le HTML/JSX.
* **Performance** : Techniques d'optimisation (e.g., *code splitting*, *memoization* avec React.memo ou useCallback/useMemo en React).
* **Accessibilité (A11Y)** : L'importance du standard **ARIA** (Accessible Rich Internet Applications), des balises sémantiques, et du contraste pour rendre l'application utilisable par tous.
* **Git & Gestion de Versions** : Revoir les concepts avancés (branches, merge, pull request, gestion des conflits) et la collaboration sur GitHub.

**Extraits de Code (Exemples)**

**Tailwind CSS (Utility-first) :**

JavaScript

// Style directement appliqué via des classes utilitaires prédéfinies

<button

className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded"

>

Mon Bouton Stylé

</button>

**Accessibilité (ARIA) :**

HTML

<button

aria-label="Fermer la fenêtre modale"

onClick={handleClose}

>

X

</button>

**Performance (React Memoization) :**

JavaScript

// Évite le re-rendu de ce composant enfant si ses props n'ont pas changé

const MonComposantMemo = React.memo(({ nom }) => {

console.log('Rendu du composant enfant');

return <p>Bonjour, {nom}</p>;

});

**Séances 8 – Projet final & déploiement**

**Concepts Théoriques**

* **Création de Mini-Projet Complet** : Synthèse des connaissances Angular ou React pour construire une application **CRUD (Create, Read, Update, Delete)** simple.
* **Intégration API** : Connexion à un *backend* réel ou simulé (e.g., via **JSON-Server**). Gestion du cycle de vie des données (chargement, erreur, succès).
* **Déploiement** : Mise en ligne de l'application Front-End sur des plateformes modernes et gratuites.
  + **Vercel** et **Netlify** sont optimisés pour les applications Front-End/JAMstack et gèrent automatiquement les étapes de *build*.
  + **GitHub Pages** pour des projets plus simples ou statiques.
* **Processus de Déploiement Typique** : Commit du code sur GitHub -> Liaison du dépôt à la plateforme de déploiement -> Déclenchement automatique du *build* et du déploiement (CI/CD simple).

**Extraits de Code (Exemples)**

**Mise en place de JSON Server (Simulation d'API) :**

Bash

# Installation globale ou locale

npm install -g json-server

# Créer un fichier db.json :

# { "taches": [{ "id": 1, "titre": "Projet Final" }] }

# Lancer le serveur (l'API est maintenant disponible sur http://localhost:3000/taches)

json-server --watch db.json

**Script de *build* (dans package.json avant déploiement) :**

JSON

{

"scripts": {

// Commande standard pour préparer l'application à la production

"build": "ng build --configuration=production" // (Angular)

// ou

// "build": "vite build" // (React/Vite)

}

}