

Le développement Frontend

Ce cours a pour but d'introduire des notions fondamentales et générales autour du développement frontend d'applications web.

Bien que ces connaissances soient générales et non liées à des choix de technologies particulières, le contenu est orienté autour des techniques modernes du web (en 2025).

Ce contenu est bien sûr non exhaustif, et n'est là que pour guider les étudiants vers les concepts essentiels à connaître.

La documentation officielle est toujours à privilégier et il est vivement recommandé de la consulter.

L'importance de l'expérience utilisateur (UX)

L'expérience utilisateur est devenue un élément central du développement frontend moderne. Elle ne se limite pas à l'aspect visuel, mais englobe l'ensemble des interactions entre l'utilisateur et l'application.

Impact sur l'engagement des utilisateurs

- 53 % des utilisateurs abandonnent un site qui met plus de 3 secondes à charger
- 88 % des utilisateurs ne reviendront pas sur un site après une mauvaise expérience
- Une bonne UX augmente le taux de conversion de 200 % en moyenne

Principes fondamentaux de l'UX en développement

1. **Réactivité** : L'application doit répondre instantanément aux actions de l'utilisateur
2. **Cohérence** : Les interactions doivent être prévisibles et logiques

3. **Feedback** : L'utilisateur doit toujours savoir ce qui se passe (chargement, erreurs, succès)
 4. **Accessibilité** : L'application doit être utilisable par tous
-

Les enjeux du développement frontend moderne

Le développement frontend fait face à plusieurs défis majeurs :

Complexité croissante

- Applications de plus en plus sophistiquées
- Multitude de périphériques et de tailles d'écran
- Gestion d'états complexes
- Interactions en temps réel

Maintenance et évolutivité

- Code maintenable et évolutif
- Architecture scalable
- Dette technique à gérer
- Mise à jour des dépendances

Les langages de programmation

Le JavaScript

JavaScript ayant été le premier langage de programmation à être intégré dans un navigateur web, il est depuis resté le principal langage pour développer des applications web, et ce tout au long de la chaîne (pages web, serveurs, outils, etc.)

Cela a grandement facilité le développement d'applications car cela a permis d'utiliser le même langage sur une grande partie de la stack, permettant d'améliorer la qualité du code et la réutilisabilité.

TypeScript

Il est important de savoir que le JavaScript a beaucoup évolué depuis sa création. Il a été amélioré au fur et à mesure du temps pour se moderniser et améliorer la qualité de code et l'expérience développeur. Il n'est donc pas rare de trouver des exemples de code avec des anciennes pratiques qu'il ne faut plus reproduire.

Un des gros défauts de JavaScript en tant que langage de programmation est qu'il n'est pas typé : il n'est pas nécessaire de déclarer le type des variables et celui-ci peut changer au cours de l'exécution. Cela rend le langage beaucoup plus simple à apprendre et à écrire, mais le rend très peu résilient.

Afin de pallier ce manque, le langage TypeScript a été inventé pour améliorer la robustesse des programmes JavaScript. Il est à noter que TypeScript est un langage intermédiaire et n'est pas directement exécuté. Un code en TypeScript est d'abord *transpilé* en JavaScript puis exécuté.

Le fait d'écrire un programme en TypeScript permet de pré-valider que notre logique est correcte au moment du développement, mais à l'exécution, rien ne

garantit que les variables manipulées sont bien du type attendu (requêtes API, librairies externes, etc.)

Cela permet tout de même d'améliorer grandement la fiabilité des programmes, car la plupart des bugs peuvent être détectés avant même l'exécution du code, et nécessite moins de vérifier que l'application fonctionne correctement à l'exécution, à condition d'être rigoureux sur la définition des types et des variables.

Voici un exemple du même code en JavaScript et en TypeScript :

```
function calculate_average(values) {  
  return values.reduce((s, v) => s + v, 0) / values.length;  
}  
  
// Intended to be used with numbers  
const numbers = [10, 20, 30];  
console.log(calculate_average(numbers)); // ✅ Works fine: 20.0  
  
// But what if we accidentally pass strings?  
const object = { n1: 10, n2: 20 };  
console.log(calculate_average(object)); // ❌ Runtime error: unknown
```

```
function calculate_average(values: Array<number>) {  
  return values.reduce((s, v) => s + v, 0) / values.length;  
}  
  
const numbers = [10, 20, 30];  
console.log(calculate_average(numbers)); // ✅ Works fine: 20.0  
  
const object = { n1: 10, n2: 20 };  
console.log(calculate_average(object)); // ❌ Compilation error: exp
```

[Cheatsheet TypeScript](#)

Qualité de code

ESLint

Afin de réduire au maximum les erreurs de code, on va utiliser des outils de vérification automatique de qualité de code, en temps réel dans l'IDE et aussi juste avant l'exécution des tests. Ceux-ci possèdent une liste de règles qui doivent être respectées et pointeront vers les bouts de code qui "sentent mauvais".

Les retours de linting ne sont pas des erreurs de compilation : techniquement cela n'empêche pas le code de compiler et de s'exécuter, mais informe qu'il y aura certainement eu des problèmes à l'exécution.

Note sur l'IA

Les IA génératives sont aujourd'hui suffisamment performantes pour produire du code fonctionnel de grande qualité. Cependant, afin de les faire générer du code de la meilleure qualité possible, il est indispensable de leur fournir le maximum d'informations, notamment, leur typage et des règles strictes.

Plus un langage est fortement typé, meilleure sera la qualité de code produite par une IA

Le fait que le compilateur donne des erreurs précises sur les problèmes rencontrés va aider l'IA à produire un code de qualité optimale, et réduire les hallucinations.



Poussez au maximum les exigences en termes de qualité de code (typage, linting, tests, etc.), car cela ne ralentit plus la productivité aujourd'hui grâce à l'IA

Outils de développement

Afin de nous aider dans le développement d'applications, il existe une multitude d'outils qui vont nous faciliter le travail, de la création à la maintenance de celles-ci, tels que la gestion des librairies externes, compilation, serveur de développement, hot-reload, minification, etc.

Historiquement, la plupart de ces outils ont été codés en JavaScript pour garder une cohérence avec le code de l'app, mais une transition vers des langages plus rapides est en train de se faire et la plupart des outils sont en train d'être portés en Rust pour une expérience de développement (DX) plus fluide. (Turbopack, SWC, etc.)

La dépendance principale de tout projet web sera Node.js <https://nodejs.org/>.

Gestionnaires de paquets

NPM (Node Package Manager)

- Gestionnaire de paquets par défaut pour Node.js
- Permet d'installer et gérer les dépendances d'un projet
- Utilise le fichier `package.json` pour décrire le projet et ses dépendances
- Commandes essentielles :

```
npm init      # Initialiser un nouveau projet
npm install   # Installer les dépendances
npm run SCRIPT # Exécuter des scripts
```

bash

Alternatives

NPM a été à un moment critiqué pour ses faiblesses (performances, fiabilité, fonctionnalités, etc.) et de nombreuses alternatives ont été créées, telles que Yarn, pnpm, etc.

Leur fonctionnement est relativement similaire et les paquets restent compatibles avec NPM, et il est possible de passer d'un gestionnaire à un autre si besoin (nécessitant de légères modifications).

NPM a évolué pour rattraper son retard et reste généralement suffisant pour la plupart des projets.

Le package.json

Voici un exemple détaillé d'un fichier `package.json` typique pour un projet frontend moderne :

```

{
  "name": "mon-projet-frontend",
  "version": "1.0.0",
  "description": "Une application web moderne utilisant Vue.js",
  "author": "John Doe <john.doe@example.com>",
  "private": true,
  "license": "MIT",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "test": "jest",
    "lint": "eslint --ext .js,.vue src"
  },
  "dependencies": {
    "vue": "^3.3.0"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.2.0"
  },
  "engines": {
    "node": ">=16.0.0",
    "npm": ">=8.0.0"
  },
  "browserslist": ["> 1%", "last 2 versions", "not dead"]
}
```

Ce fichier va définir :

- Les informations générales (name, description, version, etc.)
 - Des contraintes de version (engine, browserslist)
 - Des scripts (start, test, etc.) qui serviront régulièrement au développement de l'application
 - Des dépendances (paquets) dont l'application a besoin, avec leur version (dev dependencies → uniquement pour le développement, inutiles pour lancer l'application en production)
-

Les outils de build

Vite

- Outil de build moderne et ultra-rapide
- À privilégier, plus simple et rapidement configurable pour les outils modernes
- Développé par l'équipe Vue.js
- Avantages :
 - Démarrage instantané
 - Rechargement à chaud très rapide
 - Configuration simple
 - Support natif des modules ES
- Particulièrement adapté aux frameworks modernes (Vue, React, Svelte)

Webpack

- Bundler le plus populaire et mature
- Nécessite du boilerplate code
- Fonctionnalités principales :
 - Regroupement des fichiers (bundling)
 - Minification du code

- Gestion des assets (images, styles, etc.)
 - Hot Module Replacement (HMR)
 - Configuration plus complexe mais très flexible
-

Les outils de test

Jest

- Framework de test JavaScript complet
- Utilisé pour les tests unitaires (tester des morceaux de codes indépendamment)
- Caractéristiques :
 - Tests isolés et parallélisés
 - Couverture de code intégrée
 - Mock et simulation faciles

```
test("addition de 2 nombres", () => {  
  expect(1 + 2).toBe(3);  
});
```

jsx

Cypress

- Outil de test end-to-end (E2E) moderne
- Permet de tester l'application comme un utilisateur réel, lance l'application dans un navigateur virtuel plutôt que d'exécuter uniquement des bouts de code
- Avantages :
 - Interface visuelle intuitive
 - Tests en temps réel dans le navigateur
 - Debugging facile
 - Capture d'écran et vidéos automatiques

jsx

```
describe('Page d'accueil', () => {  
  it('charge correctement', () => {  
    cy.visit('/')  
    cy.get('h1').should('be.visible')  
  })  
})
```

Bootstrapping

La plupart des frameworks proposent une commande pour automatiquement démarrer et configurer un nouveau projet.

Se référer à la doc officielle pour la commande à jour :

sh

```
npm create vue@latest  
npm create nuxt@latest  
npx create-next-app@latest
```

Les frameworks et bibliothèques modernes

Pourquoi utiliser un framework ?

Avantages

- **Productivité accrue**
 - Réutilisation de composants
 - Structures et patterns préétablis
 - Nombreuses fonctionnalités prêtes à l'emploi
- **Maintenance facilitée**
 - Architecture standardisée

- Code organisé et cohérent
- Documentation extensive
- **Performance optimisée**
 - Mécanismes d'optimisation intégrés
 - Gestion efficace du DOM
 - Outils de développement dédiés

Inconvénients

- Courbe d'apprentissage initiale
- Surcharge potentielle pour les petits projets
- Dépendance à l'écosystème du framework

Panorama des frameworks populaires

Vue.js

- **Caractéristiques principales**
 - Framework progressif
 - Facile à intégrer dans des projets existants
 - Syntaxe intuitive et claire (HTML et JS classiques)
 - Excellente documentation en plusieurs langues
- **Points forts**
 - Courbe d'apprentissage douce
 - Performance élevée
 - Flexibilité d'utilisation
 - Écosystème bien structuré (Pinia, Vue Router)
 - Stabilité dans l'utilisation

```
<template>  
  <div class="counter">
```

html

```

    <h2>Vue Counter</h2>
    <p>Count: {{ count }}</p>
    <button @click="count++">Increment</button>
    <button @click="count--">Decrement</button>
  </div>
</template>

<script>
  const count = ref(0);
</script>

```

React

- **Caractéristiques principales**
 - Développé et maintenu par Facebook
 - Basé sur les composants
 - Utilisation intensive du JSX
- **Points forts**
 - Énorme écosystème
 - Grande communauté
 - Nombreuses opportunités professionnelles
 - Excellent pour les grandes applications
- **Points faibles**
 - Syntaxe JSX spécifique à react
 - Evolution rapide des techniques

```

import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div className="counter">
      <h2>React Counter</h2>
      <p>Count: {count}</p>

```

tsx

```

    <button onClick={() => setCount(count + 1)}>Increment</button>
    <button onClick={() => setCount(count - 1)}>Decrement</button>
  </div>
);
}

export default Counter;

```

```

const element = <h1 className="greeting">Hello, world!</h1>;
const element = React.createElement(
  "h1",
  { className: "greeting" },
  "Hello, world!"
);

```

jsx

Svelte

- **Caractéristiques principales**
 - Approche compilée
 - Pas de Virtual DOM
 - Code plus léger
- **Points forts**
 - Performance exceptionnelle
 - Bundle final très léger
 - Réactivité native

```

<script>
  let count = 0;
</script>

<div class="counter">
  <h2>Svelte Counter</h2>
  <p>Count: {count}</p>
  <button on:click={() => count++}>Increment</button>

```

html

```
<button on:click={() => count--}>Decrement</button>
</div>
```

HTMX

- **Caractéristiques principales**

- Approche minimaliste
- Extension des capacités HTML
- Pas de reactivité
- Pas de JavaScript complexe
- Principalement pour le rendu coté serveur avec un moteur de template

- **Points forts**

- Simplicité d'utilisation
- Intégration facile
- Performances natives du navigateur
- Excellent pour les applications CRUD simples

```
<!-- HTML -->
<div class="counter" id="root">
  <h2>HTMX Counter</h2>
  <p>Count: <span id="count-value">0</span></p>
  <button
    hx-post="/increment"
    hx-trigger="click"
    hx-target="#count-value"
    hx-swap="innerHTML"
  >
    Increment
  </button>
  <button
    hx-post="/page_2"
    hx-trigger="click"
    hx-target="#root"
    hx-swap="innerHTML"
  >
```

html

```
Go to page  
</button>  
</div>
```

```
from flask import Flask  
  
app = Flask(__name__)  
count = 0  
  
@app.route('/increment', methods=['POST'])  
def increment():  
    global count  
    count += 1  
    return str(count)  
  
@app.route('/page_2', methods=['POST'])  
def page_2():  
    return f"""  
    <div>  
        <h1>Page 2</h1>  
        <p>Counter: {counter}</p>  
    </div>  
    """
```

python

Astro

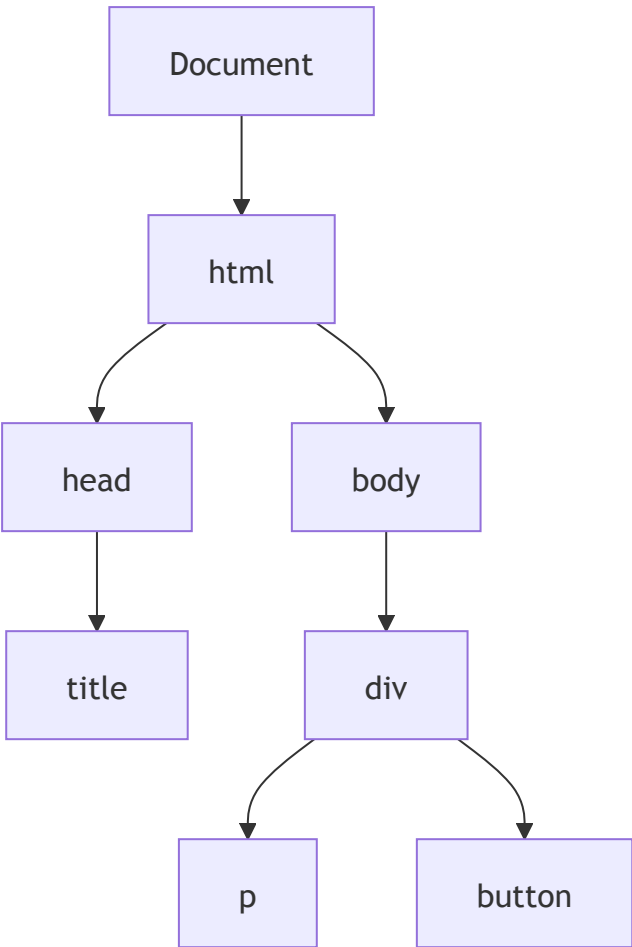
Astro est un framework permettant principalement d'exporter des sites statiques (SSG)

On peut l'utiliser avec différents framework comme React ou Vue, et il est plus efficace que Next ou Nuxt pour charger du contenu statique (Markdown, headless CMC) et rendre des sites statiques extrêmement performants. Le rendu n'est plus une SPA.

Le DOM

Le DOM (Document Object Model) est une interface de programmation qui représente la structure d'un document HTML sous forme d'arbre.

Structure du DOM



Virtual DOM vs Real DOM

Caractéristique	Real DOM	Virtual DOM
Nature	Structure réelle du document	Copie légère en mémoire

Caractéristique	Real DOM	Virtual DOM
Performances	Modifications coûteuses	Modifications rapides
Mise à jour	Mise à jour complète	Mise à jour différentielle
Utilisation	Navigateur	Frameworks modernes

Impact sur les performances

1. Real DOM

- Chaque modification déclenche un reflow
- Opérations synchrones et bloquantes
- Consommation mémoire importante

2. Virtual DOM

- Compare l'état précédent et nouveau
- Met à jour uniquement les différences
- Optimise les performances de rendu

Exemple de manipulation du DOM

```
jsx
// Manipulation directe (Real DOM)
document.getElementById("monElement").innerHTML = "Nouveau contenu";

// Écoute des événements
document.getElementById("monBouton").on("click", () => alert("clic !"));

// Avec un framework utilisant le Virtual DOM (React exemple)
useState({ contenu: "Nouveau contenu" }); // Le framework optimise le rendu
```

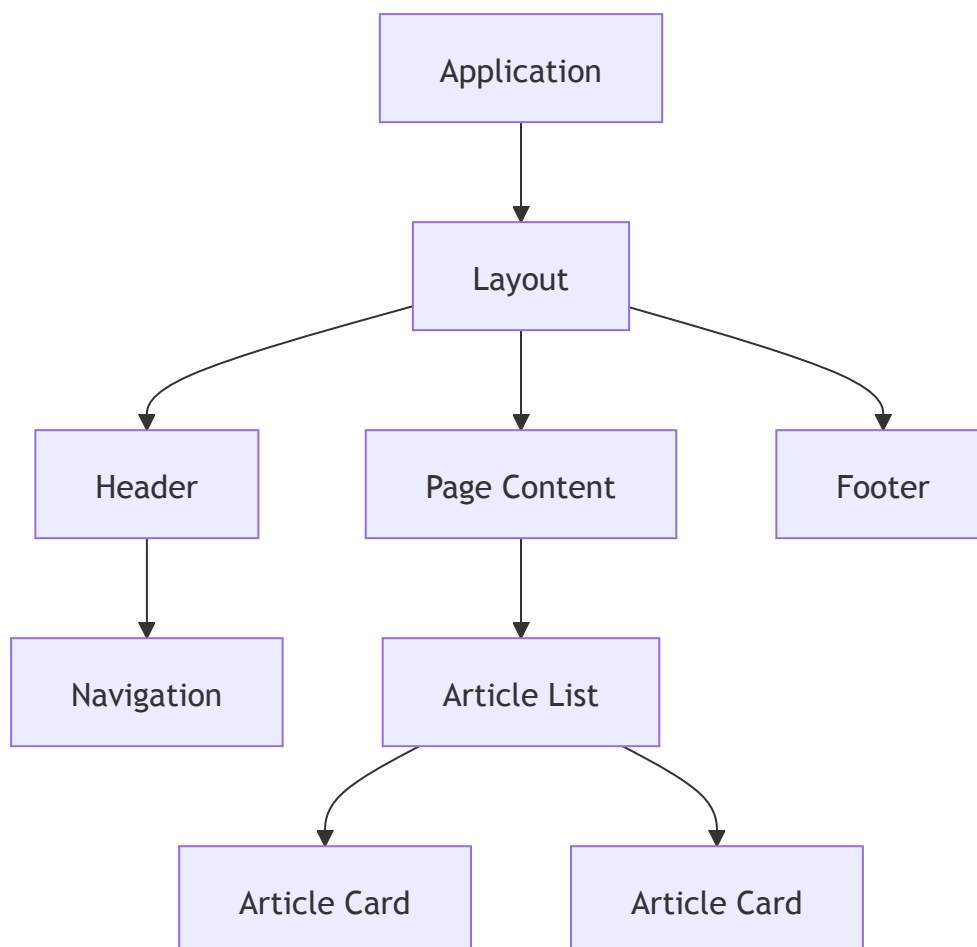
À lire

<https://vuejs.org/guide/extras/rendering-mechanism>

Architecture orientée composant

Principes de la réutilisabilité

- **Définition d'un composant**
 - Unité autonome et réutilisable
 - Encapsule HTML, CSS et logique JavaScript
 - Interface clairement définie
- **Avantages**
 - Maintenance simplifiée
 - Code DRY (Don't Repeat Yourself)
 - Tests facilités
 - Développement parallélisé



Différents types de composants

À l'échelle du framework, tous les composants sont similaires. Cependant, pour maintenir un code de qualité, nous séparons les composants selon leur usage :

App

Le composant d'application est l'unique composant principal qui va charger l'application. Il a le rôle de charger tous les modules nécessaires et c'est lui qui sera monté dans le DOM.

Layout

Le(s) composant(s) de layout servent de squelette aux pages, afin que celles-ci aient un même format (header/footer/router, etc.).

Pages/Vues

Les pages, aussi appelées vues en Vue, servent de composant principal pour chaque route de l'application, chargées par le routeur généralement depuis le layout.

Data-fetching

Certains composants ont besoin de récupérer des données (via une API par exemple), et vont afficher ces données ou bien les redistribuer aux composants qu'ils utilisent, et peuvent potentiellement avoir des effets de bord (comme modifier des données via une API).

Composants de présentation

Certains composants seront uniquement utiles pour créer l'interface, ne feront pas d'appels API, n'auront pas d'effets de bord et changeront uniquement en fonction des propriétés qui leur sont envoyées (exemples : Button, Input, Calendar, etc.).

Communication entre composants

- **Props** (données descendantes)
 - Passage de données parent vers enfant
 - Immutables dans le composant enfant
- **Contexte**
 - Données communes à toute l'application

Réactivité

Une des raisons principales pour lesquelles les frameworks ont été inventés, en plus de la composabilité, est la réactivité. Cela signifie garantir que lorsqu'une donnée change de valeur, son affichage doit être mis à jour partout où elle est utilisée.

Le but d'un framework est de garantir cette mise à jour et de faire en sorte qu'elle soit la plus rapide possible, afin de pouvoir afficher un maximum d'informations tout en restant fluide.

Chaque framework a sa façon de faire et c'est le plus compliqué à apprendre lorsque l'on passe d'un framework à un autre, même si globalement le principe reste le même : **utiliser des variables observables** (être notifié quand elles changent de valeur afin de mettre à jour l'affichage).

Voir [Panorama des frameworks populaires](#) pour comparer les différentes implémentations de réactivité des frameworks.



Il est vivement recommandé de consulter les explications sur la réactivité dans la documentation officielle du framework que vous utilisez.

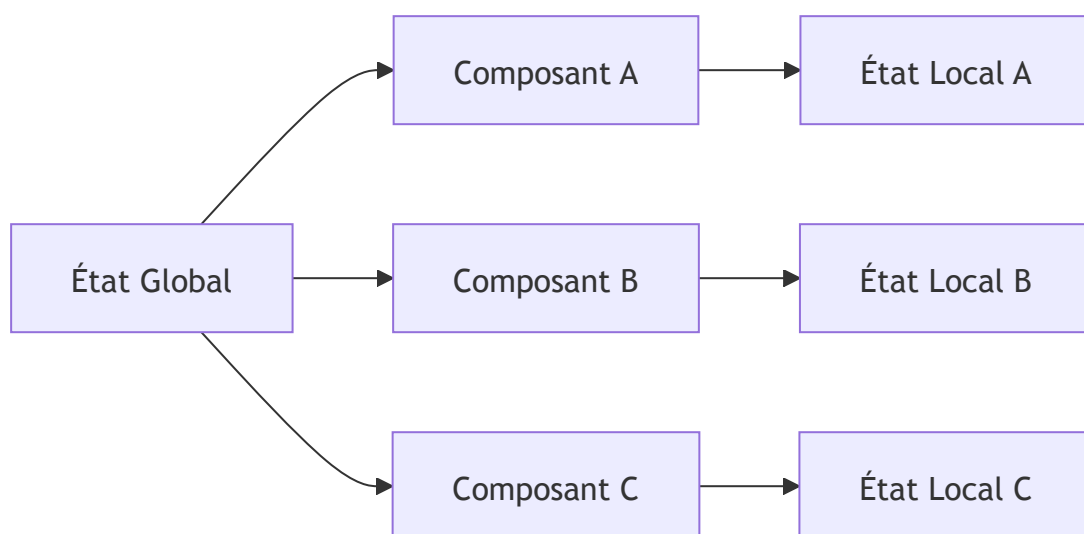
<https://vuejs.org/guide/extras/reactivity-in-depth.html>

Gestion de l'État (State Management)

Principes fondamentaux

On va toujours séparer les données d'état d'une application en deux types :

- **État local**
 - Données propres à un composant
 - Gestion simple et directe
 - Pas fait pour les données présentes à différents endroits
- **État global**
 - Données partagées entre composants
 - Passage par un store centralisé
 - Single source of truth
 - Actions et mutations contrôlées
 - État prévisible
 - Mise à jour automatique partout où la donnée est utilisée
 - Solutions dédiées (Pinia, Redux, contexte, etc.)



Communication avec le Backend

APIs REST

- Principes

- Endpoints standardisés
- Méthodes HTTP (GET, POST, PUT, DELETE)
- Réponses formatées (JSON)

L'inconvénient d'une API REST est que le client doit systématiquement requêter le serveur pour obtenir des données à jour.

GraphQL

GraphQL est un langage de requête et un environnement d'exécution pour les API, créé par Facebook en 2015. Contrairement aux API REST traditionnelles, GraphQL offre plusieurs avantages distinctifs :

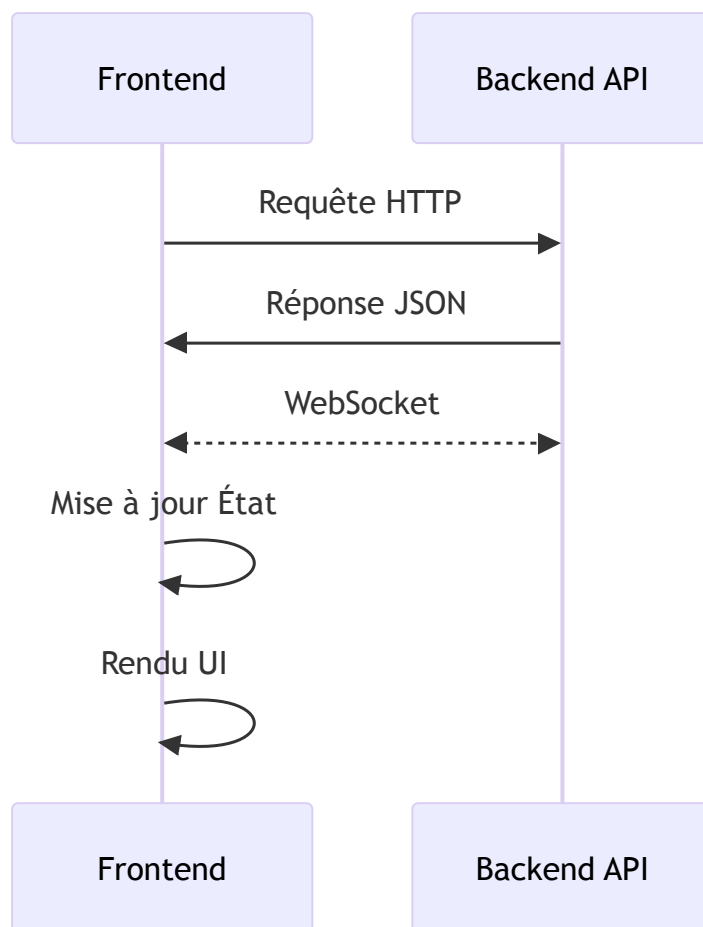
1. **Requêtes précises** : Les clients peuvent demander exactement les données dont ils ont besoin, ni plus ni moins.
2. **Structure hiérarchique** : Les données sont organisées de manière hiérarchique, reflétant naturellement les relations entre objets.
3. **Un seul endpoint** : Contrairement à REST qui utilise plusieurs endpoints, GraphQL utilise généralement un seul point d'entrée.
4. **Typage fort** : GraphQL définit un schéma typé qui sert de contrat entre le client et le serveur.
5. **Introspection** : Les clients peuvent interroger le schéma pour découvrir les capacités de l'API.

GraphQL est particulièrement utile pour les applications modernes avec des interfaces complexes et des besoins de données variables, réduisant ainsi le surchargement de données et améliorant les performances. Il offre aussi un système de souscription qui permet d'obtenir des mises à jour de données en temps (presque) réel (pas aussi performant que WebSocket).

Un des gros avantages est que, contrairement aux API REST, qui peuvent casser les clients si le format des réponses change, le retour d'une requête GraphQL sera toujours identique.

WebSocket

La technologie WebSocket permet une communication bidirectionnelle entre un client et un serveur. La connexion est ouverte au démarrage de l'application, et le client ou le serveur peut envoyer des informations à n'importe quel moment à l'autre partie. Cela permet d'avoir des données mises à jour en temps réel (jeux vidéo, plateforme de trading, chat).



Les différents types de rendu d'applications frontend

Client-Side Rendering (CSR)

- **Définition** : Le rendu est effectué entièrement côté client, dans le navigateur de l'utilisateur.
- **Fonctionnement** : Le serveur envoie un fichier HTML minimal avec des liens vers des scripts JavaScript. Une fois chargés, ces scripts créent dynamiquement le contenu de la page.
- **Caractéristiques** :
 - Charge initiale plus longue
 - Navigation ultérieure très rapide
 - Interactivité élevée
 - Peut poser des problèmes de SEO si mal implémenté
 - Consommation de ressources côté client

Page HTML retournée par le serveur en CSR

```
tsx
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="UTF-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Zero to Hundred</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

Server-Side Rendering (SSR)

- **Définition** : Le rendu HTML est effectué sur le serveur avant d'être envoyé au client.
- **Fonctionnement** : Le serveur exécute le code JavaScript, génère le HTML complet et l'envoie au client, qui peut ensuite l'"hydrater" pour le rendre interactif.
- **Caractéristiques** :
 - Premier chargement rapide
 - Meilleur pour le SEO
 - Moins de ressources utilisées côté client
 - Requiert plus de ressources serveur
 - Frameworks comme Next.js (React), Nuxt.js (Vue), SvelteKit facilitent cette approche

React a récemment introduit le concept de server component, que Next utilise maintenant nativement.

Page HTML retournée par le serveur en SSR

```
html
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" href="/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Zero to Hundred</title>
  </head>
  <body>
    <div id="app">
      <h1>Articles</h1>
      <div>
        <h2>Article 1</h2>
        <h2>Article 2</h2>
        <h2>Article 3</h2>
      </div>
    </div>
  </body>
</html>
```

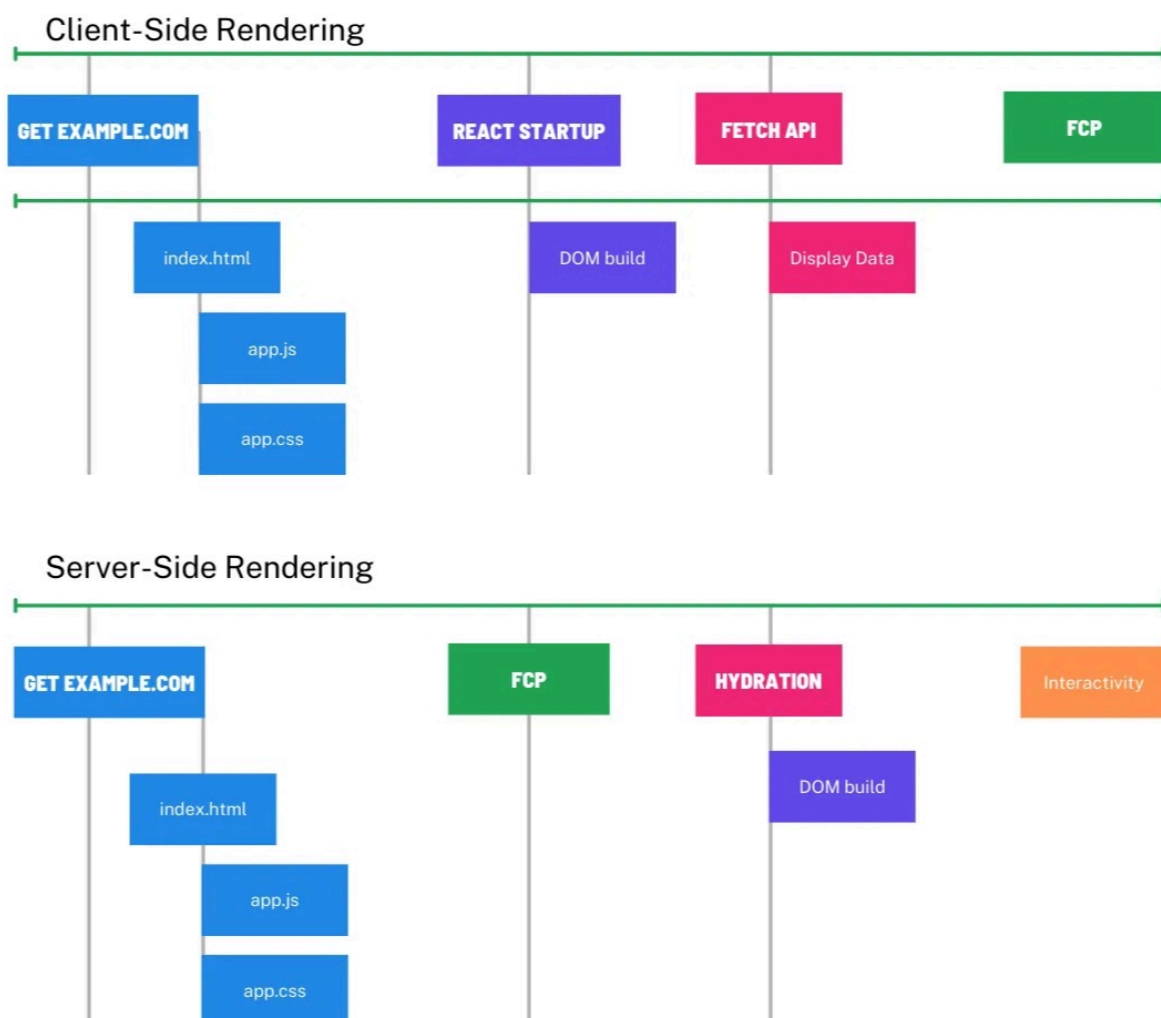
```
</div>
<script type="module" src="/src/main.ts"></script>
</body>
</html>
```



Hydration

L'hydration fait référence au processus par lequel une application JavaScript "reprend vie" dans le navigateur après qu'un HTML initial a été généré et envoyé par le serveur. Plus précisément, c'est le processus qui permet de transformer une page HTML statique (rendue côté serveur) en une application interactive côté client, comme si elle était CSR.

Time to First Contentful Paint



Static Site Generation (SSG)

- **Définition** : Les pages HTML sont pré-générées lors de la phase de build.
- **Fonctionnement** : Toutes les pages sont créées en amont et servies comme des fichiers statiques.
- **Caractéristiques** :
 - Performances optimales
 - Sécurité maximale
 - Excellent pour le SEO
 - Idéal pour les contenus qui ne changent pas fréquemment
 - Outils comme Astro, Gatsby, VuePress, Nuxt Content

Incremental Static Regeneration (ISR)

- **Définition** : Une évolution du SSG qui permet de régénérer des pages spécifiques à intervalles réguliers ou sur demande.
- **Caractéristiques** :
 - Combine les avantages du SSG avec une certaine fraîcheur des données
 - Disponible dans des frameworks comme Next.js

Hybride

Il est possible de combiner toutes ces techniques : on peut pré-rendre le contenu au build (articles de blog) tout en gardant un site dynamique qui récupère certaines données (commentaires) à l'exécution (*islands architecture*).

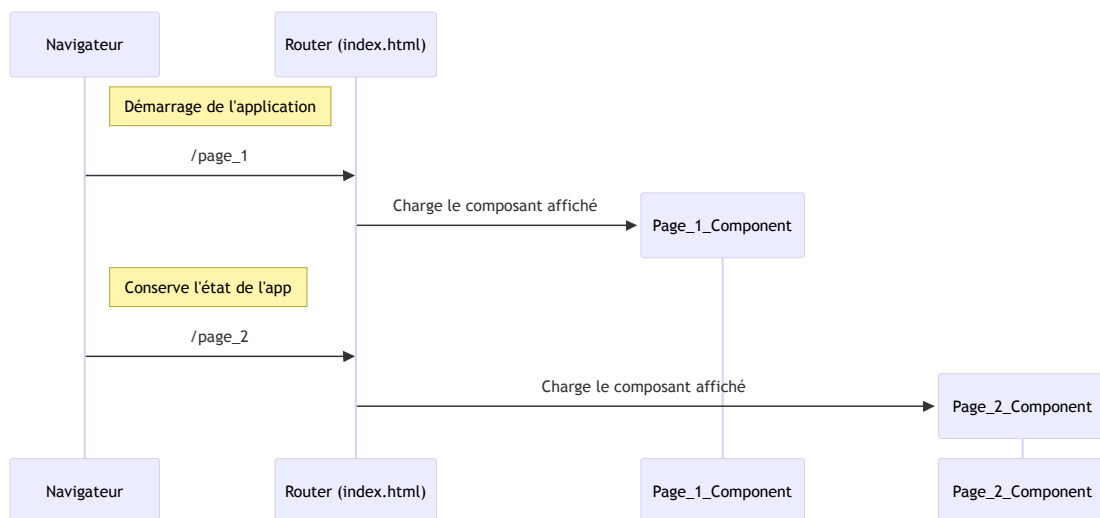
Problèmes rencontrés en SSR

- **Différences entre environnements** : Le code doit s'exécuter à la fois côté serveur (Node.js) et côté client (navigateur), avec des API différentes.
- **Répartition de l'état** : L'application tourne sur deux machines en même temps (client et serveur), certaines données sont accessibles d'un côté et pas de l'autre, synchronisation des deux.
- **Accès aux API navigateur** : Les API spécifiques au navigateur (`window` , `document` , `localStorage` , `web3`) ne sont pas disponibles côté serveur, et `fetch` est différent.
- **Gestion des imports conditionnels** : Nécessité de charger différents modules selon l'environnement d'exécution.
- **Charge serveur accrue** : Génération de HTML pour chaque requête, augmentant la charge CPU et mémoire.
- **Techniques en constante évolution** : Les façons de développer un site en SSR évoluent très vite et il est facile d'être perdu dans les mises à jour des frameworks.

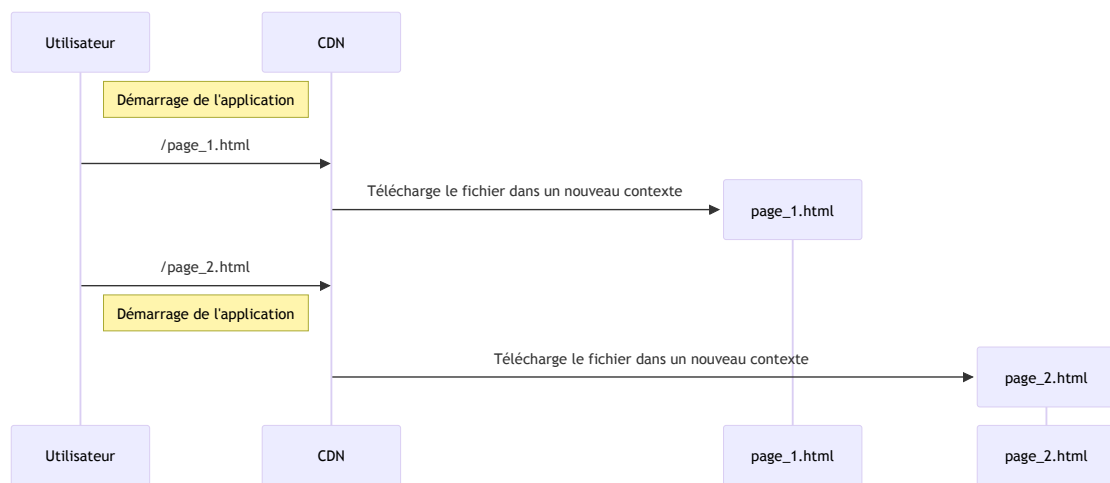
Routage côté client

Single Page Application (SPA)

- **Définition**
 - Application web monopage
 - Navigation sans rechargement
 - Expérience fluide type "application native"
- **Fonctionnement**
 - Chargement initial de l'application
 - Mise à jour dynamique du contenu
 - Gestion de l'historique du navigateur



Contre-exemple



SSG

Les applications à rendu statique n'auront généralement pas besoin de routeur (comme ce site).

Solutions d'hébergement d'applications frontend

Pour les applications statiques

Une application statique est une application dont les pages sont construites au moment du build et ne changent pas tant qu'il n'est pas reconstruit.

Un serveur statique est un serveur qui va uniquement renvoyer les fichiers présents dans un dossier, souvent de type CDN, avec du cache. Il n'aura pas de logique propre et ne changera pas le contenu du site en lui-même.

Il est optimisé pour retourner le contenu le plus rapidement possible, et est souvent réparti un peu partout dans le monde pour que la ressource soit au plus proche des utilisateurs.

Ces hébergements sont très peu coûteux et souvent gratuits.

On les utilisera pour les applications **CSR** et **SSG**, qui, lors de la compilation, fournissent en sortie un ensemble de fichiers suffisants pour l'exécution de l'application.

Pour le SSG, toutes les données seront déjà incluses dans ces fichiers, tandis que pour le CSR elles seront récupérées à l'exécution par le client.

Pour les applications SSR

Les pages d'une application à rendu côté serveur seront rendues à chaque requête utilisateur.

Les applications à rendu côté serveur nécessiteront plus de travail de la part du backend qu'un simple envoi de fichier. Lorsque le client demande des pages, le

serveur devra récolter les données demandées, et construire lui-même le HTML.

Il faudra alors passer par une plateforme spécialement conçue pour ça, ou un serveur classique.

Plateformes d'hébergement principales

Les plateformes PaaS (Platform as a Service) fourniront un service très simple tout clé en main, en général une simple commande suffit pour obtenir une application déployée et entièrement fonctionnelle. Le cloud sera lui plus complexe à mettre en place, mais avec plus de contrôle de la stack. Enfin, le bare-metal est une solution qui peut offrir de très bonnes performances avec des coûts très faibles.

- **Vercel** : créateurs de Next.js, permet de déployer très facilement la plupart des applications web full-stack sous réserve de respecter certains critères (serverless)
- **Netlify** : Équivalent de Vercel
- **Render** : Permet de déployer tout type d'applications assez simplement
- **AWS** : Tout type de déploiement (complexe)
- **Bare Metal** : Machine à configurer soi-même, très économique

Styles, CSS et UI

Historiquement, il a existé de nombreux outils pour faciliter l'intégration du design dans une page web.

La technologie principale pour construire le style d'une page web est le CSS. Cependant, il devient rapidement fastidieux de n'utiliser que celui-ci lorsqu'une application grandit. C'est pourquoi de nombreux outils ont été créés pour faciliter l'expérience développeur et maintenir un code propre et maintenable.

Historiquement, on peut retrouver des outils comme **Bootstrap** ou **Material Design**. Aujourd'hui, la norme s'est tournée vers **Tailwind**, qui a été largement adopté et qui est très robuste. Beaucoup de librairies l'utilisent comme base, et elle permet de modifier rapidement le style d'une page de manière cohérente, globale et indépendante.

Un développeur qui maîtrisera Tailwind pourra rapidement s'adapter à un projet existant qui l'utilise, et ce, peu importe les autres technologies utilisées.

UX : Style VS UI

L'expérience utilisateur (UX) sera déterminée par deux facteurs : le style (CSS) et l'interface utilisateur (UI).

Il est important de distinguer le style de l'UI : le style détermine à quoi l'application ressemble, et l'UI détermine comment elle réagit.

Il existe donc trois types de librairies différentes :

- Les librairies de style pur (Tailwind)
- Les librairies d'UI pur (HeadlessUI, Radix)
- Les librairies de style + UI (Material Design, shadcn, daisyUI)

Ces librairies pourront être spécifiques à un framework (react, vue), ou bien utilisables peu importe le framework utilisé.

Design System

Lorsqu'une application grandit, il est important de s'assurer que son design soit cohérent à la charte graphique de l'entreprise. Les designers vont alors créer un design system, qui sera utilisé par les développeurs pour construire l'ensemble des composants d'interface.

Les librairies de styles et de composants pourront être personnalisées pour respecter le design system (typographie, couleurs, espacements, etc).

Libraries

Style

- [Tailwind CSS](#)

UI

- [DaisyUI](#)
- [shadcn/ui](#) (existe pour React et Vue)

Headless

- [HeadlessUI](#)
- [Radix](#) (React)
- [Reka UI](#) (Vue)

Icones

- [Heroicons](#)
- [Lucide](#)

SEO pour les applications web

Les enjeux du référencement

Le référencement (SEO - Search Engine Optimization) est crucial pour la visibilité d'une application web. Les applications modernes, particulièrement les Single Page Applications (SPA), font face à des défis spécifiques en matière de SEO :

1. Problématiques des SPA traditionnelles

- Contenu généré dynamiquement par JavaScript
- Les robots des moteurs de recherche peuvent avoir du mal à indexer le contenu
 - Comment le robot détermine-t-il que tout le contenu a été chargé et est prêt à être indexé ?
- Temps de chargement initial potentiellement long

2. Impact sur la visibilité

- Classement dans les résultats de recherche
- Accessibilité du contenu pour les moteurs de recherche
- Expérience utilisateur et métriques Web Vitals

Solutions techniques

1. Métadonnées dynamiques

```
<head>
  <!-- Titre de la page (très important pour le SEO) -->
  <title>Titre de ma page | Mon Site Web</title>

  <!-- Description (apparaît dans les résultats de recherche) -->
```

html

```
<meta name="description" content="Description détaillée de la page

<!-- Mots-clés (moins important aujourd'hui, mais toujours utilisé
<meta name="keywords" content="mot-clé1, mot-clé2, mot-clé3">

<!-- Auteur -->
<meta name="author" content="Nom de l'auteur">

<!-- Contrôle du comportement des robots -->
<meta name="robots" content="index, follow">

<!-- Image de preview sur les réseaux sociaux -->
<meta name="twitter:image" content="https://www.monsite.com/image.
</html>
```

```
export default {
  metaInfo: {
    title: "Mon titre de page",
    meta: [
      { name: "description", content: "Description de ma page" },
      { property: "og:title", content: "Titre pour les réseaux socia
    ],
  },
};
```

jsx

2. Plan du site (Sitemap)

- Génération automatique du sitemap.xml
- Mise à jour dynamique des URLs
- Soumission aux moteurs de recherche

3. Optimisations techniques

- URLs propres et significatives
- Structure HTML sémantique
- Optimisation des images et ressources

- Temps de chargement optimisé

Server-Side Rendering (SSR)

Le SSR est une solution majeure pour améliorer le SEO des applications modernes.

Il permettra, contrairement au CSR, de retourner instantanément toutes les données nécessaires à l'indexation. Les frameworks comme Next.js proposent des outils efficaces pour une génération automatique des métadonnées, images de couverture, sitemap, etc.

Les tests pour le frontend

Les tests sont une partie importante du développement frontend. Ils permettent de s'assurer que le code fonctionne comme prévu et de détecter les erreurs avant qu'elles ne soient visibles par les utilisateurs.

Contrairement au backend où les tests garantissent une bonne gestion des données et des permissions des utilisateurs, les tests frontend n'assurent pas la sécurité des données, mais bien du bon fonctionnement de l'application et d'une bonne UX.

Tests unitaires

Les tests unitaires sont des tests qui vérifient le bon fonctionnement d'une partie isolée de code. En frontend on testera typiquement les composants ou les composables/hooks.

Les parties du code à tester doivent être impérativement **isolées** du reste de l'application, c'est à dire qu'on ne doit pas exécuter du code qui n'est pas dans le code à tester (en dehors des librairies). Pour cela on peut utiliser des **mocks** ou des **stubs** qui vont remplacer les parties du code tierces.

Dans chaque unité testée, il faudra prendre en compte les différents cas de figure (embranchements) que le code peut prendre. Des outils de **coverage** permettent de savoir les parties du code testées ou non.

Tests d'intégration (End to End)

Les tests d'intégration sont des tests qui vérifient le bon fonctionnement de l'application dans son ensemble. Pour les lancer, on démarrera un serveur similaire à un serveur de production, l'application sera connectée au backend,

et on simulera l'utilisation d'un utilisateur via navigateur qui sera contrôlé par le test, et on vérifiera que l'application se comporte comme prévu.

Il ne sera pas nécessaire et possible de tester tous les cas d'usage possible en test d'intégration, mais on essaiera au moins de couvrir les cas d'usage habituels et critiques.

La sécurité sur le frontend

Bien que les données doivent principalement être sécurisé coté backend, le frontend a lui aussi un rôle à jouer en matière de sécurité.

Le risque principal pour une application frontend est qu'un acteur malveillant puisse injecter du code JavaScript dans la page web pour modifier son comportement et en extraire des données notamment d'authentification.

Ces informations sont très sommaires et données à titre d'exemple. Il est recommandé de lire des articles plus complets sur le sujet pour bien protéger une application.

XSS

Les attaques de cross-site scripting (XSS) sont des attaques de type injection de code. Elles permettent d'injecter du code JavaScript dans une page web et d'exécuter du code malveillant.

Pour s'en protéger, l'application doit filtrer les entrées des utilisateurs et ne pas exécuter du code HTML/JavaScript provenant d'une origine non fiable.

Le serveur web doit aussi définir les origines des ressources externes autorisées.

CORS

Le CORS est un mécanisme de sécurité qui permet de limiter les risques de XSS, en limitant les origines qui peuvent accéder à une ressource.

Fonctionnement

Le serveur doit définir quelles origines sont autorisées à accéder à ses ressources.

Lorsqu'une page web essaie d'accéder à une ressource (appel API par exemple), le navigateur vérifiera si l'origine de la page est autorisée à accéder à la ressource.

```
Access-Control-Allow-Origin: my-app.com
```

Le CORS n'empêche pas d'accéder à la ressource en soi, mais les règles du navigateur vont interdire l'accès à la ressource si l'origine n'est pas autorisée.

En développement

Lorsqu'on développe localement une application, on aura souvent le frontend et le backend sur des origines (port) différentes, il est fastidieux de définir correctement les origines autorisées, et on activera souvent le CORS pour toutes les origines.

```
Access-Control-Allow-Origin: *
```

```
// express.js
app.use(
  cors({
    origin: "*",
  })
);
```

js

Exemple d'attaque

Scenario 1

Imaginons une application web avec authentification par cookie.

- Les utilisateurs peuvent commenter des articles

- Le contenu des commentaires est directement affiché sur la page web comme du HTML et peut donc contenir du code JavaScript
- Un utilisateur malveillant peut injecter du code dans un commentaire qui va:
 - Récupérer les cookies de l'utilisateur
 - Les envoyer à un serveur malveillant

S'être protégé de ce genre d'attaque signifie:

- Ne pas interpréter les entrées des utilisateurs en tant que code potentiellement exécutable
- Avoir défini sur le serveur web les origines des ressources externes autorisées

Scenario 2

Imaginons une application web malveillante qu'un utilisateur ouvre par mégarde

- L'application fait appel à une application tierce non protégée sur lequel l'utilisateur est authentifié personnellement, et va récupérer des données de cet utilisateur sur cette application

S'être protégé de ce genre d'attaque signifie:

- Le serveur tiers définit les origines autorisées dans ses headers CORS
- Le navigateur détectera que le site malveillant n'est pas autorisé à accéder à la ressource et refusera l'accès

Notions supplémentaires

Autres notions à approfondir

- Service workers et multithreading
- Local Storage, IndexedDB
- Authentication
- Lazy loading et prefetch
- WebAssembly
- Edge/Fluid Computing
- Web Components
- PWA

Vue.js

Application de démonstration

- [Démo Vue.js \(CSR\)](#) (*Code source*)
-

Lien vers la documentation officielle

- [Vue.js](#)

Outils

- [Vite](#) (compilation)
- [Vue Router](#) (routing)
- [Pinia](#) (store management)
- [Vitest](#) (tests unitaires)
- [Vue Test Utils](#) (tests unitaires pour composants Vue)
- [Cypress](#) (tests e2e)
- [Playwright](#) (tests e2e)

UI

- [shadcn-vue](#)

Nuxt

Application de démonstration

[Démon Nuxt \(SSR\)](#) ([Code source](#))

Lien vers la documentation officielle

- [Nuxt.js](#)

Outils

- [Pinia](#) (store management)
- [Nuxt Test Utils](#) (tests unitaires et e2e pour app Nuxt)
- [Nuxt Security](#) (Sécurisation d'app facile)

UI

- [shadcn-vue](#)
- [Nuxt UI](#)

Exercices Vue.js

Pour travailler sur ces exercices, vous devez forker le projet de démonstration et créer une nouvelle branche.

<https://github.com/opac-teach/vue-demo>

Page Memecoin

- Ajouter une page `/exercices`
- Dans cette page, ajouter une section qui affiche une liste de memecoins
 - Récupérer la liste de memecoins depuis l'adresse
 - `https://nuxt-demo-blush.vercel.app/api/get-memecoins`
 - Les afficher dans la page
- Dans cette page, ajouter un formulaire pour créer un memecoin
 - Entrées du formulaire :
 - `name`, 4-16 caractères, obligatoire
 - `symbol`, 2-4 caractères, obligatoire, uniquement des majuscules
 - `description`, 0-1000 caractères, pas obligatoire
 - `logoUrl`, 0-200 caractères, pas obligatoire, doit être une URL valide
 - Envoyer sous forme de requête POST à cette adresse
 - `https://nuxt-demo-blush.vercel.app/api/create-memecoin`
 - Gérer la validation des entrées dans l'interface
 - Empêcher l'envoi du formulaire si les conditions ne sont pas respectées
 - Afficher des messages d'erreurs de validation
 - Afficher le résultat, et gérer le cas des erreurs API
 - Rafraîchir la liste des memecoins affichée
 - Bonus : Récupérer et stocker la liste des memecoins dans un store Pinia

Authentification

- Ajouter une page d'authentification
 - Créer un formulaire qui demande juste un mot de passe
 - `password` non vide
 - Envoyer ce mot de passe en POST à
 - `https://nuxt-demo-blush.vercel.app/api/login`
 - (le bon mot de passe est `admin123`)
 - Stocker le token JWT retourné dans un store Pinia
 - Stocker aussi son `userId` retourné
 - Injecter le token JWT dans les prochaines requêtes à l'API
 - Bonus: stocker le token dans le `localStorage` pour qu'il soit chargé si on relance l'app
 - Changer l'URL de la requête pour créer un memecoin vers
 - `https://nuxt-demo-blush.vercel.app/api/create-memecoin-protected`
 - Vérifier que la création fonctionne bien et que l'owner du memecoin créé est bien le nôtre
 - Faire évoluer l'UI en fonction de l'état connecté
 - Si on est connecté :
 - Afficher un bouton pour se déconnecter dans la navbar
 - Si l'utilisateur affiche la page de login, ne pas l'afficher et rediriger ailleurs
 - Si on est déconnecté :
 - Afficher un bouton pour se connecter dans la navbar
 - Ne pas afficher le formulaire de création de memecoin, mais un bouton pour se connecter à la place

Tests

- Créer des tests unitaires pour :

- Le(s) store(s) Pinia
- Le composant qui affiche la liste des memecoins
- Le formulaire de création de memecoin
- Créer un test d'intégration (e2e) qui va créer un memecoin et vérifier qu'il apparaît bien dans la liste

Rendu

- Commit/Push votre travail sur votre fork, et créez une pull request sur le projet d'origine pour valider votre travail

Exercices Nuxt

Pour travailler sur ces exercices, vous devez forker le projet de démonstration et créer une nouvelle branche.

<https://github.com/opac-teach/nuxt-demo>

Nous allons maintenant refaire la même chose que pour l'exercice [VueJS CSR](#), mais en utilisant le framework Nuxt en SSR.

Vous pouvez copier coller une partie du travail que vous avez déjà fait, mais certaines choses doivent être changées:

- Adapter les changements pour Nuxt
- Faire attention à ce que les requêtes ne soient pas lancées plusieurs fois (un serveur et un client)
- Pour l'authentification, utiliser l'url suivante:
 - <https://nuxt-demo-blush.vercel.app/api/login-cookie>
 - Ce endpoint stocke le token dans le cookie, ne plus le stocker dans le localStorage
 - Vérifier que les appels à [/create-memecoin-protected](#) fonctionnent encore, changer les requêtes si besoin
 - Remplacer la route [get-memecoins](#) par [get-memecoins-protected](#)
- Gérer l'état connecté avec des middlewares (redirection)

Par exemple, si on est connecté et qu'on essaye d'accéder à la page de login, on doit être redirigé vers la page d'accueil. Si on accède à la page de memecoins sans être connecté, on est redirigé vers la page de login.

- Ajouter une route API pour récupérer un memecoin par son id
- Ajouter une page pour afficher un memecoin par son id

- Définir les balises méta pour le SEO sur les pages de memecoins avec les infos pertinentes

[Projet] Plateforme de memecoin

Maintenant que vous maîtrisez les concepts de base, vous allez créer la partie frontend d'une plateforme de memecoins !

Description du projet

L'application est une plateforme de memecoins, où les utilisateurs peuvent créer et acheter/vendre des memecoins.

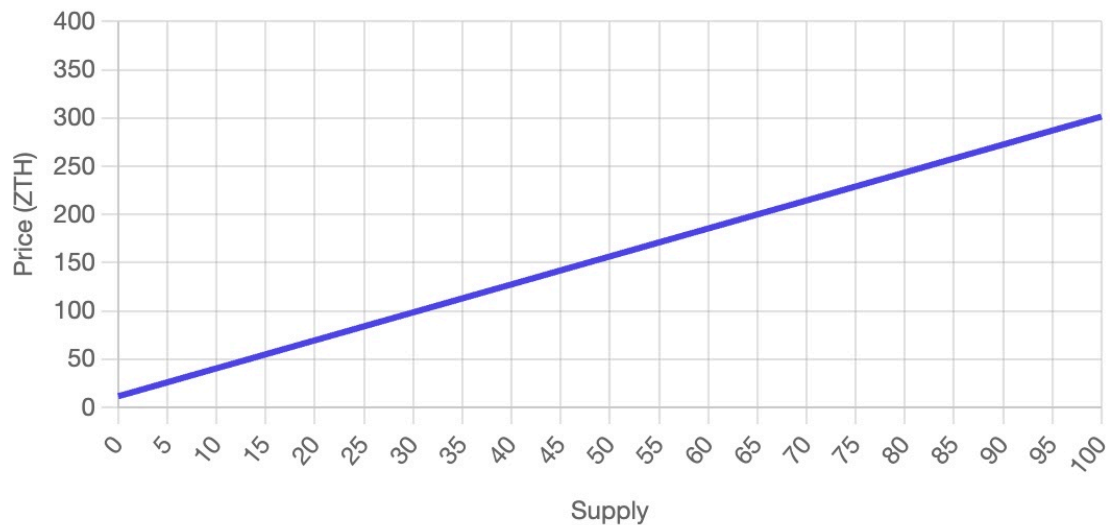
Les utilisateurs peuvent gagner du ZTH (la monnaie native de la plateforme) en créant des memecoins, en achetant stratégiquement des coins tendance, et en les vendant au moment optimal. Le solde ZTH d'un utilisateur sert de score sur le classement de la plateforme.

Bien que le site web et ses fonctionnalités ressembleront à un projet web3, tout se déroulera off-chain et n'utilisera aucune blockchain.

Tokenomics

- Chaque utilisateur reçoit 100 ZTH à l'inscription.
- Créer un memecoin coûte 1 ZTH
- Le trading des memecoin se base sur un mécanisme de *bonding curve*
 - L'échange ne se fait pas entre utilisateurs mais via une reserve de liquidité en ZTH
 - Le prix est directement lié à la quantité de token existant
 - Lorsqu'un utilisateur achète (mint) un token (avec du ZTH), le prix de celui-ci augmente et les ZTH dépensés sont placés dans la reserve
 - Lorsqu'un utilisateur vend (burn) un token, son prix diminue et il reçoit des ZTH venant de la reserve
 - Le montant de ZTH dans la reserve est toujours égale au prix de vente de la totalité des tokens existant
- Toute cette logique est déjà gérée par le backend fourni

Bonding Curve Preview




Initial Price

11.5 ZTH

Price at 100 Supply

301.5 ZTH

 alt text

Formule de prix

Pour une bonding curve linéaire, le prix P d'un token est directement proportionnel à la quantité de tokens en circulation S (supply):

$$P = a * S + b$$

Avec a (slope) et b (starting price) des constantes.

Pour calculer le prix d'achat ou de vente, il faut donc calculer l'intégrale de la fonction de prix entre la quantité de tokens en circulation actuelle et la quantité de tokens en circulation après l'achat ou la vente.

Pour acheter X tokens, le cout C revient à :

$$C = a * ((X+S)^2 - S^2) / 2 + X * b$$

Backend

Un backend déjà fonctionnel vous sera fourni.

La spécification API est disponible à cette adresse:

<https://zero-to-hundred-backend.onrender.com/api/swagger>

Travail demandé

Vous pourrez au choix utiliser VueJS ou Nuxt.

Le site web devra avoir un look fun et original, dans l'esprit crypto et degen

Vous pouvez vous inspirer de sites tels que pump.fun memecoin.org
[deployyyyyr](https://deployyyyyr.com)

Pour ce qui est modèle de données et interaction, votre unique référence sera la spécification OpenAPI swagger du backend, que vous trouverez à cette adresse:

Vous devrez implémenter le maximum de fonctionnalités possibles au vu de ce que est proposé par le backend.

Bonus

- (bonne) Utilisation de Nuxt
- SEO, sitemap et metadata
- Bonne UX
- Utilisation stricte de Typescript
- Bonne couverture de test

Implementation de référence

<https://zero-to-hundred-frontend.onrender.com>

