Svelte 5

Initiation

SVELTE

Support réalisé par Pierrick Hauguel

01

Rappels

ES6

Variables et Types de Données

Les variables en JavaScript peuvent être déclarées avec let, const, ou var (moins recommandé). let permet de déclarer des variables dont la valeur peut changer, tandis que const est pour des valeurs constantes.

Les types de données incluent les types primitifs (string, number, boolean, null, undefined, symbol) et les types non primitifs (objets, tableaux).

```
let age = 25; // Variable qui peut changer
const name = "John"; // Constante
let isStudent = false; // Type booléen
let score = null; // Type null
let x; // Undefined
const person = { firstName: "Alice", lastName: "Doe" }; // Objet
let numbers = [1, 2, 3]; // Tableau
```

Maîtriser les Opérateurs

```
let x = 10;
let y = 5;
let z = x + y; // 15
x += y; // x = x + y
let isEqual = x === y; // false
let isGreaterThan = x > y; // true
let andOperation = (x > 5) && (y < 10); // true
let orOperation = (x < 5) || (y > 2); // true
```

Les opérateurs en JavaScript incluent les opérateurs arithmétiques (+, -, *, /, %), d'assignation (=, +=, -=, etc.), de comparaison (==, ===, !=, !==, >, <, >=, <=) et logiques (&&, ||, !).

Structures de Contrôle

Les structures de contrôle dirigent le flux d'exécution du programme. Les conditions utilisent if, else if, else, et switch pour exécuter différents blocs de code basés sur des conditions. Les boucles for, while, et do...while permettent de répéter l'exécution d'un bloc de code.

```
if (x > 5) {
  console.log("x est supérieur à 5");
} else {
  console.log("x est inférieur ou égal à 5");
}

for (let i = 0; i < 5; i++) {
  console.log(i);
}

let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Tout sur les Fonctions

Les fonctions peuvent être définies de plusieurs manières: déclarations de fonction, expressions de fonction, et fonctions fléchées.

```
function sayHello() {
  console.log("Hello!");
}

const sayGoodbye = function() {
  console.log("Goodbye!");
};

const add = (x, y) => x + y;

sayHello(); // Affiche "Hello!"
sayGoodbye(); // Affiche "Goodbye!"
console.log(add(5, 3)); // 8
```

Manipulation d'Objets

```
const person = {
  firstName: "John",
  lastName: "Doe",
  greet: function() {
    console.log("Hello, " + this.firstName);
  }
};

console.log(person.firstName); // John
person.lastName = "Smith";
person.greet(); // Hello, John
```

Les objets en JavaScript sont des collections de paires clé/valeur. La notation littérale permet de créer des objets, et on accède ou modifie leurs propriétés à l'aide de la notation pointée ou des crochets. Les méthodes d'objet sont des fonctions associées à des objets.

Exploiter les Tableaux

Les tableaux en JavaScript peuvent être manipulés et itérés à l'aide de méthodes telles que .map(), .filter(), .reduce(), et .forEach(). Ces méthodes permettent de traiter les éléments d'un tableau de manière efficace et concise.

```
const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x * x);
console.log(squared); // [1, 4, 9, 16, 25]

const even = numbers.filter(x => x % 2 === 0);
console.log(even); // [2, 4]
```

```
numbers.forEach(x => console.log(x));
```

Fonctions Fléchées (=>)

Les fonctions fléchées offrent une syntaxe plus concise pour écrire des fonctions en JavaScript, permettant d'écrire des expressions de fonction plus courtes et directes. Elles sont particulièrement utiles pour les fonctions anonymes et les callbacks.

```
const numbers = [1, 2, 3, 4, 5];
// Exemple avec map utilisant une fonction fléchée
const squared = numbers.map(x => x * x);
console.log(squared); // Affiche [1, 4, 9, 16, 25]

// Fonction fléchée sans argument
const sayHello = () => console.log("Hello!");
sayHello();

// Fonction fléchée avec plusieurs arguments
const add = (a, b) => a + b;
console.log(add(5, 7)); // Affiche 12

// Fonction fléchée avec corps étendu
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(2, 3)); // Affiche 6
```

Modularité avec les modules ES6

```
// file: math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// file: app.js
import { add, subtract } from './math.js';
console.log(add(2, 3)); // 5
console.log(subtract(5, 2)); // 3
```

Les modules permettent une meilleure organisation du code en séparant les fonctionnalités en différents fichiers, favorisant la réutilisation et la maintenance.

Promesses et Async/Await

```
// Function to perform a GET request using Fetch and return a promise
function fetchExample(url) {
  return fetch(url)
    .then(response => {
      // Check if the response is OK (status 200)
      if (response.ok) {
        // Return the response in JSON format
        return response.json();
      } else {
        // Throw an error with an appropriate message
        throw new Error(`Error ${response.status}: ${response.statusText}`);
      }
    })
    .then(data => {
      // Resolve the promise with the retrieved data
      return data;
    })
    .catch(error => {
      // Reject the promise with the encountered error
      throw error;
    });
}

// Example of using the fetchExample function
const url = 'https://api.example.com/data';

fetchExample(url)
  .then(data => {
    console.log('Data retrieved successfully:', data);
    // Do something with the retrieved data
  })
  .catch(error => {
    console.error('An error occurred while retrieving data:', error);
    // Handle the error appropriately
  });
```

Les promesses et la syntaxe async/await simplifient l'écriture de code asynchrone, rendant le traitement des opérations asynchrones plus lisible et facile à gérer.

Déstructuration pour une Meilleure Lisibilité

La destructuration permet d'extraire facilement des données des objets et des tableaux, rendant le code plus propre et plus lisible.

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // John
console.log(age); // 30

const numbers = [1, 2, 3];
const [first, , third] = numbers;
console.log(first); // 1
console.log(third); // 3
```

Paramètres par Défaut, Rest et Spread

Les valeurs par défaut pour les fonctions, l'opérateur Rest pour regrouper les arguments, et l'opérateur Spread pour étaler/expanser des éléments offrent plus de flexibilité dans la gestion des données.

```
function greet(name, greeting = "Hello") {
  console.log(`${greeting}, ${name}!`);
}

greet('John'); // Hello, John!
greet('John', 'Good morning'); // Good morning, John!

const parts = ['shoulders', 'knees'];
const body = ['head', ...parts, 'toes'];
console.log(body); // ["head", "shoulders", "knees", "toes"]
```

Template Literals

Les template literals permettent de construire des chaînes de caractères interpolés avec des expressions, rendant le code plus expressif et facilitant la création de chaînes multilignes.

```
const name = "John";
const greeting = `Hello, ${name}!
How are you today?`;
console.log(greeting);
```

Nouvelles Méthodes pour Objets et Tableaux

```
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.find(x => x > 3)); // 4
console.log(numbers.includes(2)); // true

const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["John", 30]
```

ES6 et les versions ultérieures ont introduit de nouvelles méthodes pour travailler avec les objets et les tableaux, facilitant la recherche, la transformation, et la vérification des données.

L'Importance de l'Immutabilité

```
// Mauvaise pratique : modification directe d'un objet
let livre = { titre: "Programmation JavaScript" };
livre.titre = "Programmation Avancée"; // Modification directe

// Bonne pratique : immutabilité
const livreImmutable = Object.freeze({ titre: "Programmation JavaScript" });
const nouveauLivre = { ...livreImmutable, titre: "Programmation Avancée" };
```

L'immutabilité est un principe fondamental qui consiste à ne pas modifier directement les données. Au lieu de cela, toute modification produit une nouvelle instance des données. Cela aide à éviter les effets de bord et rend le code plus prévisible et facile à débugger.

Principes de la Programmation Fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui traite les calculs comme l'évaluation de fonctions mathématiques et évite les données changeantes ou mutables. Les principes clés incluent l'utilisation de fonctions pures et la composition de fonctions pour construire des logiciels.

```
// Fonction pure
const ajouter = (x, y) => x + y;

// Composition de fonctions
const multiplierParDeux = x => x * 2;
const ajouterEtMultiplier = (x, y) => multiplierParDeux(ajouter(x, y));

console.log(ajouterEtMultiplier(2, 3)); // 10
```

Stratégies de Gestion des Erreurs

La gestion des erreurs est cruciale pour développer des applications robustes. L'utilisation de blocs try / catch permet de gérer les exceptions de manière élégante, tandis que les promesses offrent un mécanisme pour traiter les erreurs dans les opérations asynchrones.

```
// Utilisation de try / catch
try {
  const resultat = operationRisquee();
  console.log(resultat);
} catch (erreur) {
  console.error("Une erreur s'est produite:", erreur);
}

// Gestion des erreurs dans les promesses
fetch("https://api.exemple.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(erreur => console.error("Erreur lors de la récupération des données:"
```

Maîtriser la Gestion des Paquets

```
// package.json avec un script personnalisé
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "start": "webpack --mode development && node server.js",
    "build": "webpack --mode production"
  },
  "dependencies": {
    "react": "^17.0.1"
  },
  "devDependencies": {
    "webpack": "^5.22.0",
    "babel-loader": "^8.2.2"
  }
}
```

NPM et Yarn sont des gestionnaires de paquets pour JavaScript qui simplifient l'installation, la mise à jour et la gestion des dépendances de projets. Ils permettent également de définir et d'exécuter des scripts personnalisés pour automatiser les tâches de développement courantes.

Linting et Formatage avec ESLint et Prettier

```
// .eslintrc.json
{
  "extends": "eslint:recommended",
  "rules": {
    "no-unused-vars": "warn",
    "eqeqeq": ["error", "always"]
  }
}
```

```
// .prettierrc
{
  "semi": false,
  "singleQuote": true
}
```

ESLint et Prettier sont des outils essentiels pour maintenir la qualité du code JavaScript. ESLint analyse le code pour détecter les erreurs et les problèmes de style, tandis que Prettier reformate automatiquement le code selon des règles définies, assurant une cohérence stylistique.

Nouveautés Javascript

ECMAScript 6 (ES6) - 2015
Apport majeur :

Classes : Introduction de la syntaxe de classe pour la programmation orientée objet, qui est plus propre et facile à utiliser.
Modules : Standardisation du support des modules ES6 pour une meilleure gestion du code en facilitant l'import et l'export de composants.
Promesses : Pour la gestion asynchrone, facilitant l'écriture de code asynchrone et la gestion des opérations asynchrones.

Nouveautés Javascript

Autres apports :

let et const : Nouveaux mots-clés pour les déclarations de variables, offrant des portées de bloc, réduisant ainsi les erreurs communes dues aux variables globales.
Fonctions fléchées : Syntaxe plus concise pour l'écriture de fonctions, et ne crée pas de nouveau contexte this.
Paramètres par défaut, Rest et Spread : Amélioration de la gestion des paramètres de fonctions.
Destructuration : Permet une affectation plus facile et plus claire des données à partir d'arrays ou d'objets.
Templates littéraux : Intégration facile des variables dans les chaînes de caractères avec les templates littéraux.

ECMAScript 7 (ES7) - 2016

Opérateur d'exponentiation (**) : Pour calculer la puissance d'un nombre.
Méthode Array.prototype.includes : Vérifie si un tableau inclut un élément donné, renvoyant true ou false.

ECMAScript 8 (ES8) - 2017

async et await : Simplification de l'écriture de fonctions asynchrones pour rendre le code asynchrone aussi facile à lire et à écrire que le code synchrone.
Méthodes de l'objet : Object.values(), Object.entries(), et Object.getOwnPropertyDescriptors() pour une meilleure manipulation des objets.

ECMAScript 9 (ES9) - 2018

Opérateur de décomposition pour objets : Étend les fonctionnalités de décomposition aux objets.
Promesses finally() : Méthode ajoutée aux promesses pour exécuter du code une fois que la promesse est réglée, indépendamment du résultat.

ECMAScript 10 (ES10) - 2019

Array.prototype.{flat, flatMap} : Méthodes pour aplatir des tableaux imbriqués et appliquer une fonction, puis aplatir le résultat.
Object.fromEntries() : Transforme une liste de paires clé-valeur en un objet.
Chaînes de caractères et modifications de Array.sort : Améliorations mineures pour la manipulation de chaînes de caractères et la méthode sort.

ECMAScript 11 (ES11) - 2020

BigInt : Introduit un type pour représenter des entiers très grands.
Promise.allSettled : Une nouvelle méthode de promesse qui renvoie un tableau de résultats après que toutes les promesses données se soient résolues ou rejetées.
Dynamique import() : Importations de modules sur demande pour améliorer la performance du chargement de modules.

Introduction à Vite

Vite est un outil de build moderne conçu pour les projets JavaScript, TypeScript, et les frameworks tels que Vue, React, et Svelte. Il offre un démarrage rapide de serveur de développement et des rechargements à chaud ultra-rapides en tirant parti de l'importation de modules ES natifs dans les navigateurs et en pré-bundling des dépendances avec esbuild.

```
# Installation de Vite pour un nouveau projet React
npm create vite@latest mon-projet-react -- --template react

# Démarrage du projet
cd mon-projet-react
npm install
npm run dev
```

Configuration de Vite

```
// vite.config.js
import reactRefresh from '@vitejs/plugin-react-refresh';

export default {
  plugins: [reactRefresh()],
  resolve: {
    alias: {
      '@': '/src',
    },
  },
  build: {
    outDir: 'build',
  },
};
```

Vite est hautement configurable via son fichier vite.config.js. Vous pouvez ajuster la résolution des modules, les plugins, les options CSS, et bien plus, permettant une personnalisation profonde pour répondre aux besoins spécifiques de votre projet.

Optimisations et Déploiement avec Vite

Vite offre des fonctionnalités d'optimisation prêtes à l'emploi pour le déploiement de production, telles que la minification, le découpage de code (code splitting), et le préchargement des modules. Ces optimisations améliorent la performance de chargement de l'application en production.

```
# Commande pour construire un projet pour la production
npm run build

# Analyse du bundle pour optimisation
npm run preview
```

02

TypeScript

TypeScript, c'est quoi ?

TypeScript est un sur-ensemble de JavaScript développé par Microsoft, ayant pour but d'ajouter un système de typage statique. Cela nous permettra d'écrire un code plus sécurisé, plus facilement maintenable, et de détecter des erreurs avant même l'exécution du code.

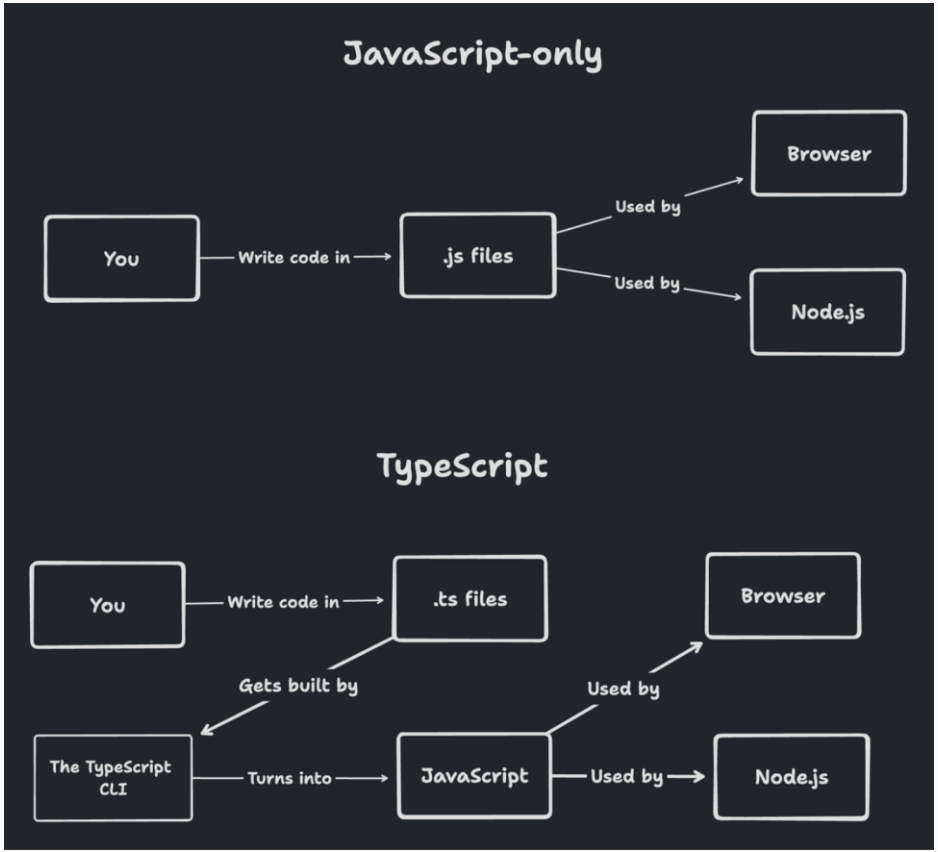
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

Les fichiers TypeScript auront une extension en .ts ou .tsx

Les avantages de TypeScript

1. Syntaxe proche du JavaScript
2. Permet de détecter des erreurs potentielles, même sans déclarer de type
3. Facilite la documentation du code
4. Permet à l'IDE de gérer les suggestion de code plus facilement
5. Meilleure maintenabilité
6. L'inférence de type

Le fonctionnement de TypeScript



La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.
Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;

person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string
```

Les types primitifs

TypeScript nous fournit 3 types primitifs, et 2 types utilitaires:

1. number
2. string
3. boolean
4. unknown (quand on ne sait pas quel type nous allons recevoir, on y reviendra après)
5. any (à éviter)

La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.
Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;
function sayHello(name : string) : string {
    return "Hello " + name;
}
person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string

sayHello(isMajor) // string attendu en paramètre, mais on passe un boolean
```

L'inférence de type

C'est sympa de pouvoir déclarer des types primitifs sur les variables lorsqu'on les déclare, mais ça va vite devenir très répétitif non ?
C'est là que l'inférence de type entre en scène: TS va être capable de « déduire » le type de certaines variables et signatures de fonction via leur valeurs d 'origine. Pas besoin de tout typer explicitement !

```
let person = "Jean";
let age = 30;
let isMajor = true;
function sayHello(name : string)  {
    return "Hello " + name;
}
person = age; //Même erreur d'auparavant
isMajor = isMajor + person; //Idem

sayHello(isMajor) // Toujours pareil
```

L'inférence de type

Attention, TS n'est pas omniscient, et ne sera pas toujours capable d'inférer les types correctement.
Si il ne sait pas, il utilisera le type any, qui ne procède à aucune vérification !
Par exemple :

```
const myArray = []; //Type implicite: any[]
myArray.push(1) //Pas d'erreur car non strict
myArray.push('Hello') //Pas d'erreur car non strict, mais dangereux
```

Types non conformes à la réalité

TypeScript nous permet de détecter des erreurs dans le code, mais peut parfois autoriser des opérations qui généreraient des erreurs à l'exécution.

```
const names = ["Charles", "Pierrick"]
console.log(names[2].toLowerCase())
//TS ne signale pas d'erreur, mais la dernière ligne va générer une erreur à l'exécution
```

Type et Interface

TypeScript nous permet de créer nos propres types, grâce aux mots clés « type » et « interface ».

```
type Age = number;
interface Person {
    name: string;
    age: Age;
}
let driver: Person = {
    name: 'James May',
    age: 56
};
driver.age = 57; // OK
driver.age = '57'; // Error
```

Type ou Interface ?

Type et interface peuvent sembler interchangeables, car ils le sont majoritairement. Normalement, type est plutôt utilisé pour définir les alias, et interface pour typer des objets complets. À notre niveau, nous pouvons nous contenter d'utiliser type au général, car ils nous permettent de faire des « union types » qui nous seront utiles pour la suite.

```ts
type item = {
    name: string,
    price: number,
}

type weapon = item & {
    damage: number,
}
```

```ts
interface item {
    name: string;
    price: number;
}

interface weapon extends item {
    damage: number;
}
```

Union types et Narrowing

Avec le mot clé type, nous pouvons créer des « union types », c'est-à-dire des variables qui peuvent accepter plusieurs types, par exemple:

```
type StringOrNumber = string | number;

let myAge: StringOrNumber = 25;
myAge = '25'; // OK
```

Pour savoir quel est le type réel d'une variable avec une union, on peut utiliser l'inférence de TS :

```
const doSometing = (value: StringOrNumber) => {
    if (typeof value === 'string') {
        //TypeScript infère que c'est une string
        return value.toUpperCase();
    }
    //TypeScript infère que c'est un number
    return value;
}
```

Narrowing sur des objets ?

Pour faire du narrowing sur des types complexes, nous pouvons utiliser le mot clé « in » pour déterminer ou non l'existence d'une propriété propre à un des types.

```
type Fish = {
    swim : () => void;
}
type Bird = {
    fly : () => void;
}

function move(animal: Fish | Bird){
    if("swim" in animal){
        return animal.swim();
    }
    return animal.fly();
}
```

Les génériques

Les génériques sont une fonctionnalité de TS qui nous permettent de créer des types génériques réutilisables et de réduire le code à écrire. Prenons l'exemple ci-dessous:

```
type StringCollection = {
    name: string;
    items : string[];
}
type NumberCollection = {
    name: string;
    items : number[];
}
type BooleanCollection = {
    name: string;
    items : boolean[];
}
```

Les génériques

On peut créer un type générique Collection, qui prendra en « paramètre » un type T qui sera assigné à la propriété items. Nos types précédents seront donc déclarées par le générique.

```
type Collection<T> = {
    name: string;
    items : T[];
}
type StringCollection = Collection<string>;
type NumberCollection = Collection<number>;
type BooleanCollection = Collection<boolean>;
```

Les génériques

Il existe des génériques utilitaires, comme <Partial>, <Omit>, <Record>... que nous utiliserons peut être lors de la formation.

Vous pouvez également les retrouver sur la documentation TypeScript, juste ici: https://www.typescriptlang.org/docs/handbook/utility-types.html

Pour aller plus loin

Compte Twitter et Youtube et Matt Pocock
https://twitter.com/mattpocockuk
https://www.youtube.com/@mattpocockuk

03

Introduction à Svelte

Introduction à Svelte

- Svelte est un framework moderne de développement web qui se distingue des autres frameworks populaires comme React et Vue. Contrairement à ces derniers, Svelte n'utilise pas de runtime pour gérer le DOM. À la place, il compile le code des composants en JavaScript, HTML, et CSS purs au moment du build. Cette approche élimine la surcharge liée au runtime, offrant des performances optimales et une expérience utilisateur plus fluide.

Fonctionnement de Svelte

- Svelte convertit le code de vos composants en instructions JavaScript optimisées qui manipulent directement le DOM. Contrairement aux frameworks qui utilisent un DOM virtuel pour gérer les changements (comme React), Svelte effectue cette optimisation dès la compilation. Cela signifie que les composants Svelte s'exécutent avec moins de frais généraux, car ils sont déjà optimisés pour le navigateur lors du déploiement. Ce processus réduit la taille de l'application et améliore les performances globales.

Pourquoi choisir Svelte ?

Svelte présente de nombreux avantages par rapport aux frameworks traditionnels :

- Pas de runtime : l'absence de runtime réduit la surcharge côté client et améliore les performances.
- Syntaxe intuitive : la syntaxe de Svelte est simple et proche de JavaScript natif, ce qui facilite son apprentissage.
- Réactivité intégrée : Svelte propose une réactivité native sans avoir besoin de hooks ou d'autres mécanismes complexes.
- Cas d'utilisation variés : Svelte est particulièrement adapté pour des applications interactives, des tableaux de bord, des PWA, et bien plus encore.

Comparaison avec React et Vue

Svelte se différencie de frameworks comme React et Vue par son modèle sans runtime :

- React utilise un Virtual DOM pour gérer les changements d'état et appliquer des mises à jour efficaces. Cependant, cela nécessite un runtime pour surveiller les modifications.
- Vue fonctionne de manière similaire à React avec un système de directives et de Virtual DOM.
- Svelte, quant à lui, compile le code en instructions JavaScript au moment du build, sans Virtual DOM ni runtime, rendant chaque composant plus léger et plus performant.

Structure d’un Projet Svelte

Dans Svelte, chaque composant est encapsulé dans un fichier .svelte, ce qui simplifie la gestion du code en centralisant la logique, le style et la structure HTML dans un même fichier. Par défaut, Svelte crée un fichier App.svelte comme point d’entrée principal de l'application, ainsi qu’un fichier main.js pour monter le composant dans le DOM. Un projet standard inclut également :

src/ : le dossier principal contenant tous les composants et la logique de l’application.
public/ : le dossier des ressources statiques telles que les images et les icônes.
package.json : fichier de configuration avec les dépendances et les scripts de build et de développement.

04

Installation de Svelte

Installation de Svelte avec create-svelte

Pour démarrer un nouveau projet Svelte, utilisez l'outil create-svelte, un générateur de projet qui configure automatiquement la structure du projet et les dépendances de base. Tapez la commande suivante pour installer create-svelte et générer un nouveau projet Svelte :

```
npm install -g create-svelte
npx create-svelte my-app
```

Structure d'un Projet Svelte Généré

Après avoir exécuté create-svelte, vous obtiendrez un projet structuré de la manière suivante :

```
my-app/
├── public/
├── src/
│   ├── App.svelte
│   ├── main.js
└── package.json
```

05

Les Composants

Introduction aux Composants dans Svelte

- Les composants sont au cœur de Svelte.
- En Svelte 5, les composants sont des fonctions au lieu de classes.
- Des concepts comme les props, les événements et les slots évoluent pour devenir plus flexibles et intuitifs.

```
<script>
  let message = "Bonjour, Svelte 5 !";
</script>

<p>{message}</p>
```

Déclaration des props avec $props

$props remplace export let pour déclarer des propriétés. Permet la déstructuration et les types en TypeScript.

```
<script>
  let { name, age } = $props({ name: "Anonyme", age: 25 });
</script>

<p>Nom : {name}, Âge : {age}</p>
```

Props avancées avec renaming

- Les noms de props peuvent être modifiés lors de la déstructuration.
- Idéal pour gérer des mots-clés réservés ou améliorer la lisibilité.

```
<script>
  let { class: className, ...others } = $props();
</script>

<div class={className} {...others}>
  Contenu ici
</div>
```

Transmission des props (spread)

Utilisez le spread (...) pour transmettre des props à un sous-composant ou un élément HTML.

```
<script>
  let props = { title: "Salut", content: "Voici un contenu" };
</script>

<Child {...props} />
```

```
<script>
  let { title, content } = $props();
</script>

<h1>{title}</h1>
<p>{content}</p>
```

Composants imbriqués

- Les composants peuvent être inclus les uns dans les autres pour construire des interfaces complexes.

```
<script>
  let items = ["Item 1", "Item 2", "Item 3"];
</script>

<ul>
  {#each items as item}
    <ListItem text={item} />
  {/each}
</ul>
```

```
<script>
  let { text } = $props();
</script>

<li>{text}</li>
```

Snippets pour les composants

- Les snippets remplacent les slots pour une composition plus puissante et flexible.
- Ils permettent de transmettre du contenu réutilisable sous forme de fonction.

```
<script>
  let items = [{ label: "Option 1" }, { label: "Option 2" }];
</script>

<Dropdown>
  {#snippet item(option)}
    <li>{option.label}</li>
  {/snippet}

  {@render item(items[0])}
</Dropdown>
```

```
<script>
  let { item } = $props();
</script>

<ul>
  {@render item({ label: "Par défaut" })}
</ul>
```

Typage des composants en TypeScript

- $props peut inclure des types pour une validation stricte des propriétés. Les snippets peuvent également être typés pour plus de sécurité.

```
<script lang="ts">
  import type { Snippet } from "svelte";

  let { data }: { data: { id: number; text: string }[] } = $props();

  let { row }: { row: Snippet<[any]> } = $props();
</script>

<table>
  {#each data as item}
    {@render row(item)}
  {/each}
</table>
```

Composants comme fonctions

- Les composants sont des fonctions pouvant être instanciées avec mount ou hydrate. Remplace la syntaxe de classe dans les versions précédentes.

```
import { mount } from "svelte";
import App from "./App.svelte";

const app = mount(App, {
  target: document.getElementById("app"),
  props: { message: "Bonjour !" },
});
```

06

Les évènements

Introduction aux événements dans Svelte 5

Svelte 5 modernise la gestion des événements en remplaçant on: par des attributs d'événements standards (onclick, oninput, etc.). Les événements sont désormais des propriétés comme les autres, ce qui simplifie leur utilisation et leur compréhension.

```
<script>
  let count = $state(0);

  function increment() {
    count++;
  }
</script>

<button onclick={increment}>
  Clics : {count}
</button>
```

Syntaxe moderne des événements

En Svelte 5, les gestionnaires d'événements peuvent être déclarés avec un nom direct (onclick) ou une fonction assignée. La syntaxe classique on:event est encore acceptée mais est dépréciée.

```
<script>
  let count = $state(0);

  function handleClick() {
    count++;
  }
</script>

<!-- Syntaxe moderne -->
<button onclick={handleClick}>Incrémenter</button>

<!-- Syntaxe dépréciée -->
<button on:click={handleClick}>Incrémenter</button>
```

Gestionnaires nommés

Pour une meilleure lisibilité, préférez des noms explicites pour vos gestionnaires d'événements. Les événements sont des fonctions normales pouvant être réutilisées.

```
<script>
  function increment() {
    console.log("Incrément !");
  }

  function logMessage(message) {
    console.log(message);
  }
</script>

<button onclick={increment}>Cliquez ici</button>
<button onclick={() => logMessage("Clic détecté !")}>
  Log Message
</button>
```

Transmission des événements dans les composants

En Svelte 5, les composants acceptent des propriétés pour transmettre des gestionnaires d'événements, remplaçant createEventDispatcher.

```
<script>
  function handleInflate() {
    console.log("Gonflé !");
  }
</script>

<Balloon inflate={handleInflate} />
```

```
<script>
  let { inflate } = $props();
</script>

<button onclick={inflate}>Gonfler</button>
```

Réexpédition des événements

- Les événements peuvent être réexpédiés via des props, ou via des objets d’attributs pour une flexibilité maximale.

```
<script>
  let { onclick, ...props } = $props();
</script>

<button {...props} onclick={onclick}>
  Cliquez-moi
</button>
```

06

La réactivité

Introduction aux runes

Svelte 5 introduit les "runes", une nouvelle API pour contrôler la réactivité dans vos composants. Ces primitives permettent une réactivité fine-grainée et universelle, que ce soit dans les composants .svelte ou dans des modules .js/.ts.

$state – État réactif

Utilisez $state pour déclarer des états réactifs. Ces états peuvent être utilisés dans des composants ou des classes. Les objets et tableaux créés avec $state sont profondément réactifs.

```
<script>
  let numbers = $state([1, 2, 3]);

  function ajouter() {
    numbers.push(numbers.length + 1); // Réactivité profonde
  }
</script>

<button on:click={ajouter}>Ajouter</button>
<p>{numbers.join(' + ')} = {numbers.reduce((a, b) => a + b, 0)}</p>
```

$state.frozen – État immuable

$state.frozen crée des états immuables. Les objets ou tableaux ne peuvent être modifiés directement, mais doivent être réassignés. Améliore les performances pour les données volumineuses.

```
<script>
  let numbers = $state.frozen([1, 2, 3]);

  function ajouter() {
    numbers = [...numbers, numbers.length + 1]; // Nouvelle assignation
  }
</script>

<button on:click={ajouter}>Ajouter</button>
<p>{numbers.join(' + ')} = {numbers.reduce((a, b) => a + b, 0)}</p>
```

$derived – État dérivé

$derived permet de créer des états réactifs dérivés à partir d'autres états. Les expressions doivent être sans effets de bord.

```
<script>
  let count = $state(0);
  let double = $derived(count * 2);
</script>

<button on:click={() => count++}>
  {count} doublé est {double}
</button>
```

$effect – Effets réactifs

$effect exécute un code chaque fois que des dépendances changent ou qu'un composant est monté. Il peut inclure une fonction de nettoyage pour libérer les ressources.

```
<script>
  let count = $state(0);

  $effect(() => {
    console.log("Le compteur est à :", count);
    return () => console.log("Nettoyage"); // Appelé avant le prochain effet
  });
</script>

<button on:click={() => count++}>Incrémenter</button>
```

$effect.pre – Effets avant le rendu DOM

$effect.pre exécute du code avant que le DOM soit mis à jour. Idéal pour des cas comme le défilement automatique.

```
<script>
  let messages = $state([]);
  let div;

  $effect.pre(() => {
    if (div && div.scrollHeight > div.offsetHeight) {
      div.scrollTo(0, div.scrollHeight);
    }
  });

  function ajouterMessage() {
    messages.push(`Message ${messages.length + 1}`);
  }
</script>

<div bind:this={div}>
  {#each messages as message}
    <p>{message}</p>
  {/each}
</div>
<button on:click={ajouterMessage}>Ajouter un message</button>
```

$effect.active et $effect.root

$effect.active vérifie si le code est exécuté dans un effet ou un modèle.
$effect.root permet de contrôler manuellement la portée d'un effet.

```
<script>
  $effect(() => {
    console.log("Dans un effet :", $effect.active()); // true
  });

  console.log("Hors effet :", $effect.active()); // false
</script>
<p>Dans le modèle : {$effect.active()}</p> <!-- true -->
```

$derived.by – Dérivations complexes

Pour des calculs plus complexes, $derived.by permet d'utiliser une fonction pour produire un état dérivé. Utile lorsque l'expression ne tient pas dans une seule ligne.

```
<script>
  let numbers = $state([1, 2, 3]);
  let total = $derived.by(() => numbers.reduce((acc, n) => acc + n, 0));
</script>

<button on:click={() => numbers.push(numbers.length + 1)}>
  Ajouter
</button>
<p>Somme : {total}</p>
```

$inspect – Debugging réactif

$inspect fonctionne comme un console.log, mais suit les changements réactifs des états surveillés.Disponible uniquement en mode développement.

```
<script>
  let count = $state(0);

  $inspect(count).with((type, value) => {
    console.log(`Type : ${type}, Valeur : ${value}`);
  });
</script>

<button on:click={() => count++}>Incrémenter</button>
```

Comparaison avec le mode classique (non-runes)

En mode non-runes, la réactivité repose sur des let réactifs et des directives $:. Avec les runes, la réactivité est explicite, claire et plus puissante.

```
<script>
  let count = 0;
  $: double = count * 2;
</script>
```

```
<script>
  let count = $state(0);
  let double = $derived(count * 2);
</script>
```

07

Les évènements

Écoute des événements

Dans Vue.js, vous pouvez écouter les événements DOM en utilisant v-on ou la directive raccourcie **@**. Cela permet d'exécuter des méthodes spécifiques lorsque des événements se produisent.

08

Les cycles de vie

Montage du composant avec $effect

$effect remplace onMount pour exécuter du code au moment où le composant est monté. Contrairement à onMount, $effect peut suivre des dépendances réactives.

```
<script>
  let count = $state(0);

  $effect(() => {
    console.log("Composant monté. Compteur : ", count);
  });
</script>

<button on:click={() => count++}>Incrémenter</button>
```

Effets avec nettoyage

$effect peut inclure une fonction de nettoyage, similaire à onDestroy, qui s'exécute avant que l'effet soit relancé ou que le composant soit détruit.

```
<script>
  let count = $state(0);

  $effect(() => {
    console.log("Effet déclenché :", count);

    return () => {
      console.log("Nettoyage pour :", count);
    };
  });
</script>

<button on:click={() => count++}>Incrémenter</button>
```

Effets avant la mise à jour du DOM

$effect.pre remplace beforeUpdate pour exécuter du code avant la mise à jour du DOM.
Idéal pour mesurer ou sauvegarder des états avant que le DOM ne change.

```
<script>
  let messages = $state([]);
  let viewport;

  $effect.pre(() => {
    if (viewport && viewport.scrollTop === viewport.scrollHeight) {
      console.log("Scroll en bas !");
    }
  });

  function ajouterMessage() {
    messages.push(`Message ${messages.length + 1}`);
  }
</script>

<div bind:this={viewport}>
  {#each messages as message}
    <p>{message}</p>
  {/each}
</div>
<button on:click={ajouterMessage}>Ajouter un message</button>
```

Comparaison avant/après

- Les runes unifient et simplifient les cycles de vie en remplaçant les anciennes API.

```
<script>
  import { onMount, beforeUpdate, onDestroy } from 'svelte';

  onMount(() => {
    console.log("Composant monté");
  });

  beforeUpdate(() => {
    console.log("Avant mise à jour");
  });

  onDestroy(() => {
    console.log("Composant détruit");
  });
</script>
```

```
<script>
  $effect(() => {
    console.log("Composant monté");

    return () => console.log("Composant détruit");
  });

  $effect.pre(() => {
    console.log("Avant mise à jour");
  });
</script>
```

09

Les formulaires

Introduction aux formulaires dans Svelte 5

Les formulaires dans Svelte 5 utilisent la puissance des runes pour une gestion réactive des données. Des fonctionnalités comme le bind et les événements restent au cœur des formulaires, avec des optimisations pour les interactions utilisateur.

```
<script>
  let name = $state("");
</script>

<input bind:value={name} placeholder="Votre nom" />
<p>Bonjour, {name} !</p>
```

Liaison avec bind

bind:value synchronise automatiquement les valeurs des champs de formulaire avec des variables réactives. Compatible avec $state pour une réactivité fine-grainée.

```
<script>
  let email = $state("");
</script>

<input type="email" bind:value={email} placeholder="Votre email" />
<p>Email saisi : {email}</p>
```

Gestion des cases à cocher

Utilisez bind:checked pour synchroniser l'état des cases à cocher avec des variables réactives.

```
<script>
  let consent = $state(false);
</script>

<label>
  <input type="checkbox" bind:checked={consent} />
  J'accepte les termes et conditions
</label>

<p>{consent ? "Consentement donné" : "Consentement non donné"}</p>
```

Champs de formulaire multiples avec $state

$state peut être utilisé pour gérer plusieurs champs de formulaire en tant qu'objet.

```
<script>
  let form = $state({ name: "", age: "" });
</script>

<input type="text" bind:value={form.name} placeholder="Nom" />
<input type="number" bind:value={form.age} placeholder="Âge" />
<p>Nom : {form.name}, Âge : {form.age}</p>
```

Formulaires et validation simple

- Les formulaires peuvent inclure une logique de validation simple directement dans le composant.

```
<script>
  let email = $state("");
  let isValid = $derived(() => email.includes("@"));
</script>

<input type="email" bind:value={email} placeholder="Email" />
<p>{isValid ? "Email valide" : "Email invalide"}</p>
```

Soumission des formulaires

Capturez les soumissions de formulaire avec onsubmit pour éviter le rechargement de la page et gérer les données.

```
<script>
  let formData = $state({ username: "", password: "" });

  function handleSubmit(event) {
    event.preventDefault();
    console.log("Données soumises :", formData);
  }
</script>

<form onsubmit={handleSubmit}>
  <input type="text" bind:value={formData.username} placeholder="Nom d'utilisateur" />
  <input type="password" bind:value={formData.password} placeholder="Mot de passe" />
  <button type="submit">Se connecter</button>
</form>
```

Gestion des fichiers avec bind:files

bind:files permet de gérer les sélections de fichiers pour des téléchargements ou des prévisualisations.

```
<script>
  let files = $state(null);

  function handleFiles() {
    console.log(files?.[0]?.name || "Aucun fichier sélectionné");
  }
</script>

<input type="file" bind:files={files} multiple />
<button on:click={handleFiles}>Afficher les fichiers</button>
```

Radio boutons

Utilisez bind:group pour synchroniser un groupe de radio boutons avec une seule variable.

```
<script>
  let choice = $state("Option 1");
</script>

<label><input type="radio" bind:group={choice} value="Option 1" /> Option 1</label>
<label><input type="radio" bind:group={choice} value="Option 2" /> Option 2</label>
<label><input type="radio" bind:group={choice} value="Option 3" /> Option 3</label>

<p>Choix sélectionné : {choice}</p>
```

10 Syntaxes de template

Introduction aux listes

Svelte utilise {#each} pour parcourir et afficher des listes. Les listes peuvent être réactives grâce à $state, ce qui permet des mises à jour automatiques de l'affichage.

```
<script>
  let items = $state(["Article 1", "Article 2", "Article 3"]);
</script>

<ul>
  {#each items as item}
    <li>{item}</li>
  {/each}
</ul>

<button on:click={() => items.push(`Article ${items.length + 1}`)}>
  Ajouter un article
</button>
```

Utilisation de {#each} avec des indices

Vous pouvez récupérer l'index de chaque élément dans la liste avec , index.

```
<script>
  let items = $state(["Tâche 1", "Tâche 2", "Tâche 3"]);
</script>

<ul>
  {#each items as item, index}
    <li>{index + 1}. {item}</li>
  {/each}
</ul>
```

Utilisation de {#key} pour des listes réactives

{#key} force Svelte à recréer un élément lorsqu'une clé change. Idéal pour les animations ou des mises à jour précises.

```
{#key value}
  <Component />
{/key}
```

Rendu conditionnel avec {¤if}

Utilisez {#if} pour afficher du contenu en fonction d'une condition. Ajoutez {#else} pour gérer le cas contraire.

```
<script>
  let isLoggedIn = $state(false);
</script>

<p>{#if isLoggedIn}Bienvenue, utilisateur !{#else}Veuillez vous connecter.{/if}</p>

<button on:click={() => isLoggedIn = !isLoggedIn}>
  {isLoggedIn ? "Se déconnecter" : "Se connecter"}
</button>
```

Chaînage des conditions avec {¤if} et {:else if}

Combinez plusieurs conditions en utilisant {#if}, {:else if} et {#else}.

```
<script>
  let userType = $state("guest");
</script>

<p>
  {#if userType === "admin"}Bienvenue, administrateur !
  {:else if userType === "member"}Bienvenue, membre !
  {:else}Bienvenue, invité !{/if}
</p>

<button on:click={() => userType = "admin"}>Passer admin</button>
<button on:click={() => userType = "member"}>Passer membre</button>
<button on:click={() => userType = "guest"}>Passer invité</button>
```

Utilisation de {¤await} pour les promesses

Utilisez {#await} pour gérer le rendu pendant une opération asynchrone.

```
<script>
  let promise = fetch("/api/data").then(res => res.json());
</script>

{#await promise}
  <p>Chargement...</p>
{:then data}
  <p>Données : {data.content}</p>
{:catch error}
  <p>Erreur : {error.message}</p>
{/await}
```

@html pour insérer du contenu HTML brut

Utilisez **@**html pour rendre du contenu HTML brut. Attention aux attaques XSS.

```
<script>
  let htmlContent = "<strong>Texte important</strong>";
</script>

<p>@html : <span>@html {htmlContent}</span></p>
```

11

Les Snippets

Introduction aux snippets dans Svelte 5

- Les **snippets** sont une nouveauté puissante de Svelte 5.
- Ils remplacent et étendent les capacités des slots en permettant de transmettre et de réutiliser du contenu via des fonctions.
- Idéal pour des composants dynamiques et réutilisables.

```
<script>
  let { item } = $props();
</script>

<ul>
  {@render item({ text: "Option par défaut" })}
</ul>
```

```
<script>
  let options = [{ text: "Option 1" }, { text: "Option 2" }];
</script>

<Dropdown>
  {#snippet item(option)}
    <li>{option.text}</li>
  {/snippet}

  {@render item(options[0])}
</Dropdown>
```

Passer plusieurs snippets

Un composant peut accepter plusieurs snippets pour différents types de contenu.

```
<script>
  let { header, row } = $props();
</script>

<table>
  <thead>{@render header()}</thead>
  <tbody>
    {#each [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }] as item}
      {@render row(item)}
    {/each}
  </tbody>
</table>
```

```
<MyTable>
  {#snippet header()}
    <tr><th>ID</th><th>Nom</th></tr>
  {/snippet}

  {#snippet row(item)}
    <tr><td>{item.id}</td><td>{item.name}</td></tr>
  {/snippet}
</MyTable>
```

12

use:

Introduction à use dans Svelte 5

```
// src/lib/useHover.js
export function useHover(node) {
  function onMouseEnter() {
    node.style.backgroundColor = "yellow";
  }
  function onMouseLeave() {
    node.style.backgroundColor = "";
  }
  node.addEventListener("mouseenter", onMouseEnter);
  node.addEventListener("mouseleave", onMouseLeave);

  return {
    destroy() {
      node.removeEventListener("mouseenter", onMouseEnter);
      node.removeEventListener("mouseleave", onMouseLeave);
    },
  };
}
```

- use permet d'attacher des actions à des éléments HTML pour ajouter des comportements. Les actions sont des fonctions qui reçoivent un élément DOM et retournent éventuellement une fonction de nettoyage.

```
<script>
  import { useHover } from '$lib/useHover';
</script>

<div use:useHover>
  Passez votre souris ici
</div>
```

Actions avec paramètres

- Les actions peuvent accepter des paramètres pour personnaliser leur comportement.
- Les paramètres sont réactifs si vous utilisez un store ou une rune.

```
export function useHighlight(node, color) {
  function onMouseEnter() {
    node.style.backgroundColor = color;
  }
  function onMouseLeave() {
    node.style.backgroundColor = "";
  }
  node.addEventListener("mouseenter", onMouseEnter);
  node.addEventListener("mouseleave", onMouseLeave);

  return {
    update(newColor) {
      color = newColor;
    },
    destroy() {
      node.removeEventListener("mouseenter", onMouseEnter);
      node.removeEventListener("mouseleave", onMouseLeave);
    },
  };
}
```

```
<script>
  import { useHighlight } from '$lib/useHighlight';
  let color = "cyan";
</script>

<div use:useHighlight={color}>
  Passez votre souris ici
</div>
<input bind:value={color} placeholder="Choisissez une couleur" />
```

Introduction à use dans Svelte 5

- use permet d’attacher des actions à des éléments HTML pour ajouter des comportements. Les actions sont des fonctions qui reçoivent un élément DOM et retournent éventuellement une fonction de nettoyage.

13

Les stores

Introduction aux stores dans Svelte 5

Les stores restent un moyen essentiel pour gérer un état global ou partagé dans une application. Avec Svelte 5, les stores coexistent avec les runes, mais les runes offrent une alternative plus simple pour certains cas. Les stores incluent writable, readable et derived, ainsi que des stores personnalisés.

```
<script>
  import { writable } from 'svelte/store';

  const count = writable(0);

  function increment() {
    count.update(n => n + 1);
  }
</script>

<button on:click={increment}>Incrémenter</button>
<p>Compteur : {$count}</p>
```

writable – Créer un store modifiable

writable est le type de store de base permettant de lire et écrire une valeur réactive. Utilisez update pour appliquer des transformations ou set pour une nouvelle valeur.

```
<script>
  import { writable } from 'svelte/store';

  const name = writable("Anonyme");

  function changeName(newName) {
    name.set(newName);
  }
</script>

<input bind:value={$name} placeholder="Entrez un nom" />
<p>Bonjour, {$name} !</p>
<button on:click={() => changeName("Svelte User")}>Changer le nom</button>
```

readable – Créer un store en lecture seule

readable crée un store dont la valeur est déterminée par une logique externe. Utile pour des valeurs dérivées de sources externes comme des APIs ou des événements.

```
<script>
  import { readable } from 'svelte/store';

  const time = readable(new Date(), set => {
    const interval = setInterval(() => {
      set(new Date());
    }, 1000);

    return () => clearInterval(interval); // Nettoyage
  });
</script>

<p>Heure actuelle : {$time.toLocaleTimeString()}</p>
```

derived – Créer un store dérivé

derived permet de créer des stores dépendants d'autres stores. La valeur du store dérivé change automatiquement en fonction des dépendances.

```
<script>
  import { writable, derived } from 'svelte/store';

  const count = writable(0);
  const double = derived(count, $count => $count * 2);
</script>

<button on:click={() => count.update(n => n + 1)}>Incrémenter</button>
<p>Compteur : {$count}</p>
<p>Double : {$double}</p>
```

Stores personnalisés

Créez des stores personnalisés en combinant writable, readable ou derived avec une logique personnalisée.

```
<script>
  import { writable } from 'svelte/store';

  function createCounter() {
    const { subscribe, set, update } = writable(0);

    return {
      subscribe,
      increment: () => update(n => n + 1),
      decrement: () => update(n => n - 1),
      reset: () => set(0),
    };
  }

  const counter = createCounter();
</script>

<button on:click={counter.increment}>+</button>
<button on:click={counter.decrement}>-</button>
<button on:click={counter.reset}>Réinitialiser</button>
<p>Compteur : {$counter}</p>
```

Stores et réactivité dans les composants

Les stores peuvent être utilisés dans les composants grâce au préfixe $ pour les rendre réactifs.

```
<script>
  import { writable } from 'svelte/store';

  const isLoggedIn = writable(false);

  function toggleLogin() {
    isLoggedIn.update(status => !status);
  }
</script>

<p>{ $isLoggedIn ? "Connecté" : "Déconnecté" }</p>
<button on:click={toggleLogin}>
  {$isLoggedIn ? "Se déconnecter" : "Se connecter"}
</button>
```

Stores combinés avec les runes

Bien que $state et $derived des runes remplacent certains cas d'utilisation des stores, vous pouvez toujours les combiner. Les stores restent utiles pour des états globaux.

```
<script>
  import { writable } from 'svelte/store';

  let localCount = $state(0);
  const globalCount = writable(0);

  function incrementBoth() {
    localCount++;
    globalCount.update(n => n + 1);
  }
</script>

<p>Local : {localCount}</p>
<p>Global : {$globalCount}</p>
<button on:click={incrementBoth}>Incrémenter les deux</button>
```

Stores et contexte global

Partagez des stores entre composants en utilisant setContext et getContext.

```
<script>
  import { setContext } from 'svelte';
  import { writable } from 'svelte/store';

  const theme = writable("light");
  setContext("theme", theme);
</script>

<Child />
```

```
<script>
  import { getContext } from 'svelte';

  const theme = getContext("theme");
</script>

<p>Thème actuel : {$theme}</p>
```

14

Le routage dans Sveltekit

Introduction au routage dans SvelteKit

SvelteKit utilise un système de fichiers pour le routage. Les fichiers .svelte dans le dossier src/routes deviennent automatiquement des routes. Les URLs sont directement dérivées des noms de fichiers et dossiers.

Pages d'accueil et autres routes

src/routes/index.svelte est la page d'accueil (/). D'autres fichiers .svelte dans src/routes correspondent à d'autres URLs.

```
src/routes/
├── +page.svelte          // Page principale "/"
├── about/
│   └── +page.svelte      // Page "/about"
├── blog/
│   ├── +page.svelte      // Page "/blog"
│   └── [slug]/
│       └── +page.svelte  // Route dynamique "/blog/:slug"
```

Routes dynamiques

Les crochets [param] dans les noms de dossiers permettent de créer des routes dynamiques.
Les paramètres sont accessibles via params.

```
// src/routes/blog/[slug]/+page.js
import type { PageLoad } from './$types';

export const load: PageLoad = ({ params }) => {
    if (params.slug === 'hello-world') {
        return {
            title: 'Hello world!',
            content: 'Bienvenue sur notre blog...'
        };
    }
    throw error(404, 'Not found');
};
```

```
<!-- src/routes/blog/[slug]/+page.svelte -->
<script lang="ts">
    import type { PageData } from './$types';

    let { data }: { data: PageData } = $props();
</script>

<h1>{data.title}</h1>
<p>{data.content}</p>
```

Chargement des données côté serveur

Les fichiers +page.server.js (ou +layout.server.js) permettent de charger des données uniquement sur le serveur. Ces données sont accessibles au moment du rendu initial et ne sont pas exposées au client. Utile pour des opérations sensibles (requêtes à une base de données, variables d'environnement, etc.).

```
// src/routes/blog/[slug]/+page.server.js
import { error } from '@sveltejs/kit';
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ params }) => {
    const post = await fetchPostFromDatabase(params.slug);

    if (!post) {
        throw error(404, 'Article introuvable');
    }

    return { post };
};
```

```
<script lang="ts">
    import type { PageData } from './$types';

    let { data }: { data: PageData } = $props();
</script>

<h1>{data.post.title}</h1>
<p>{data.post.content}</p>
```

Layouts partagés

Les layouts permettent de partager des éléments (comme une navigation) entre plusieurs pages. Utilisez un fichier +layout.svelte pour créer un layout.

```
src/routes/
├── +layout.svelte    // Layout global
├── +page.svelte      // Route "/"
└── about/
    └── +page.svelte  // Route "/about"
```

```svelte
<!-- src/routes/+layout.svelte -->
<script>
    let { children } = $props();
</script>

<nav>
    <a href="/">Accueil</a>
    <a href="/about">À propos</a>
</nav>

{@render children()}
```

Layouts imbriqués

- Les layouts imbriqués sont utilisés pour des sections spécifiques d’un site.
- Chaque layout hérite du layout parent.

```
src/routes/
├── +layout.svelte          // Layout global
└── dashboard/
    ├── +layout.svelte      // Layout pour "/dashboard"
    ├── stats/
    │   └── +page.svelte    // Page "/dashboard/stats"
    └── profile/
        └── +page.svelte    // Page "/dashboard/profile"
```

```
<!-- src/routes/dashboard/+layout.svelte -->
<script>
    let { children } = $props();
</script>

<section>
    <nav>
        <a href="/dashboard/stats">Statistiques</a>
        <a href="/dashboard/profile">Profil</a>
    </nav>

    {@render children()}
</section>
```

Redirections

- Les redirections sont gérées dans les fichiers de chargement avec la méthode redirect.

```
// src/routes/private/+page.server.js
import { redirect } from '@sveltejs/kit';

export function load({ session }) {
  if (!session.user) {
    throw redirect(302, '/login');
  }
}
```

Erreurs personnalisées

- Les erreurs sont gérées avec des fichiers +error.svelte. Personnalisez les messages d’erreur en fonction des codes d’état.

```
<!-- src/routes/+error.svelte -->
<script>
  let { error } = $props();
</script>

<h1>Erreur {error.status}</h1>
<p>{error.message}</p>
```

15 API routes dans SvelteKit

Introduction aux routes API dans SvelteKit

- Les routes API dans SvelteKit permettent de gérer des requêtes HTTP côté serveur. Chaque fichier dans src/routes/api devient une route API accessible à partir de son chemin. Les fichiers +server.js (ou .ts) définissent la logique des endpoints.

```
src/routes/
└── api/
    ├── hello/
    │   └── +server.js  // Route "/api/hello"
    └── posts/
        ├── +server.js  // Route "/api/posts"
        └── [id]/
            └── +server.js  // Route "/api/posts/:id"
```

Création d'une route GET

- Utilisez une fonction GET dans un fichier +server.js pour gérer les requêtes GET.

```
// src/routes/api/hello/+server.js
export function GET() {
  return new Response(JSON.stringify({ message: "Bonjour, API SvelteKit !" }), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

Routes avec paramètres dynamiques

- Utilisez des dossiers [param] pour créer des routes dynamiques. Les paramètres sont accessibles via params.

```
// src/routes/api/posts/[id]/+server.js
export function GET({ params }) {
  const { id } = params;
  return new Response(JSON.stringify({ postId: id }), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

Création d'une route POST

- Gérez les requêtes POST avec une fonction POST dans +server.js. Accédez aux données envoyées via request.json().

```
// src/routes/api/posts/+server.js
export async function POST({ request }) {
  const body = await request.json();
  return new Response(JSON.stringify({ success: true, data: body }), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

Gestion des méthodes PUT et DELETE

- Déclarez PUT et DELETE pour mettre à jour ou supprimer des ressources.

```
// src/routes/api/posts/[id]/+server.js
export async function PUT({ params, request }) {
  const { id } = params;
  const body = await request.json();

  return new Response(JSON.stringify({
    message: `Post ${id} mis à jour`,
    data: body,
  }), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

Utilisation des cookies et headers

- Lisez et modifiez les cookies ou les en-têtes via l'objet request.

```
// src/routes/api/user/+server.js
export function GET({ request }) {
  const cookies = request.headers.get('cookie');
  return new Response(JSON.stringify({ cookies }), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

```
// src/routes/api/user/+server.js
export function POST() {
  return new Response("Cookie défini", {
    headers: {
      'Set-Cookie': 'sessionId=12345; HttpOnly; Path=/;',
    },
  });
}
```