

Bienvenue

Parcours développeur front-end.
Module Svelte.

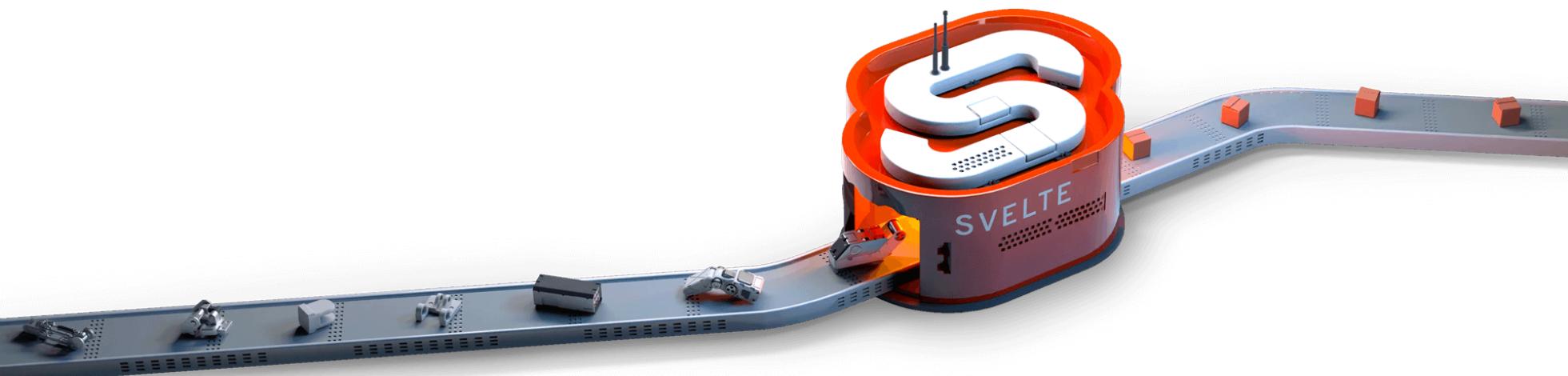
Powered by  Sliddev



Au programme de cette formation

Ce module de 40 heures vise à explorer le framework Svelte et à en comprendre les principes de fonctionnement.

Nous aborderons également les différences avec les autres grands framework du marché (React, Vue.JS, Angular, etc.), en mettant en évidence les avantages de Svelte.



Je me présente



Thomas PIERRE,
Développeur Front-End senior



Mon parcours :

■ 2013 - 2015, Responsable de communication numérique chez **Harley Davidson**



■ 2015 - 2016, License développement de project Web chez **Efficom**



■ 2015 - 2017, Intégrateur chez **PéoLéo**



■ 2017 - 2018, Développeur Front-End chez **Hobbynote**



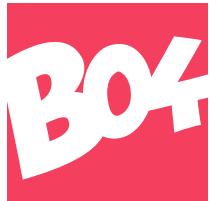
■ depuis 2018, Lead Développeur Front-End chez **Atecna**



■ depuis 2015, Gérant de **StudioB04**



StudioB04



Micro agence numérique 360.

StudioB04 est une micro agence spécialisée dans la création de sites web, et de supports de communication.

Nous accompagnons nos clients dans leur stratégie numérique et leur communication.
Nous militons pour un web plus sobre et plus accessible. Nos activités :

- Développement et maintenance de solutions web (sites vitrines, plateformes e-commerce, applications, etc.)
- Audit d'accessibilité numérique (106 critères du R.G.A.A)
- Formations et sensibilisations à l'accessibilité numérique et au numérique responsable
- Interventions en centre de formation

Au programme de cette formation

Jour 1:

Matin

1. Présentation de **Svelte**, un peu d'histoire.
2. Les principes clés de **Svelte**
3. Mise en place d'un environnement de développement
4. Les bases de la syntaxe de **Svelte**
5. Les événements
6. Les variables réactives
7. Les conditions
8. Les boucles
9. Le binding

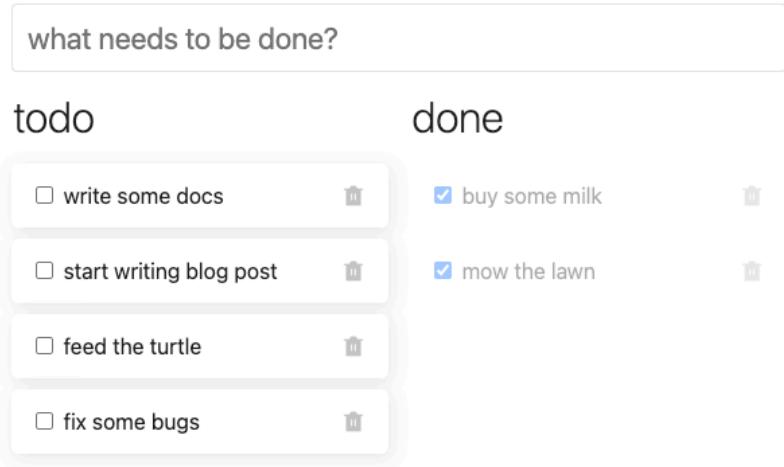
Au programme de cette formation

Jour 1:

Après midi

TP 2 : Création d'une "To-do List".

- Créer un composant **Svelte**
- Créer une fonction pour ajouter une tâche
- Créer une fonction pour déplacer une tâche en "done"
- Créer une fonction pour remettre une tâche en "todo"
- Gérer l'apparence avec la balise **style**



1. Présentation de Svelte

Un peu d'histoire.

Svelte est une librairie Javascript open-source apparue en novembre 2016 et créée par **Rich Harris**.

Rich Harris est un développeur britannique, qui a commencé sa carrière comme journaliste pour *The Guardian* et plus tard au *New York Times*.

Il a développé **Svelte** en réponse aux frameworks populaires comme **React** et **Vue** qu'il jugeait trop lourds.

Contrairement à ces derniers, **Svelte** est compilé en javascript pur et n'a besoin daucun package supplémentaire pour fonctionner.

Cela permet de produire un code final plus léger et plus performant.



Un peu d'histoire.

Rich Harris est un contributeur majeur de la communauté Javascript. Il a notamment participé à la création de :

- Ractive.js (pour TheGuardian.com)
- Rollup
- Vite
- etc.

Il travaille aujourd'hui pour **Vercel**, une entreprise spécialisée dans les outils pour les développeurs front-end, afin de continuer à travailler sur **Svelte** à plein temps.

Rich Harris est un défenseur de la simplicité dans le développement web. Il critique souvent les approches qui compliquent inutilement les outils et les workflows des développeurs.



2016, création de Svelte

Svelte est une librairie Javascript qui permet de créer simplement des composants réactifs.

La toute première version de Svelte date du 29 Novembre 2016. Elle a été écrite en **Javascript**. Ce n'est qu'à partir de la version 3 que **Svelte** utilise **TypeScript**.

La **version 5** de Svelte est sortie en Octobre 2024. Le déploiement de cette version étant encore limité, nous nous concentrerons sur **Svelte 4** dans cette formation, mais nous aborderons les nouveautés de **Svelte 5** dans un chapitre dédié, en fin de formation.

Pour accéder à la documentation de **Svelte 4**, utilisez ce lien : <https://v4.svelte.dev>



Qui utilise Svelte ?

Svelte est encore confidentiel et bien moins utilisé que **React** par exemple. Cependant il est utilisé par des entreprises majeurs comme :

- The New York Times ----- 
- IKEA ----- 
- Apple ----- 
- Spotify ----- 
- Rakuten ----- 
- Decathlon ----- 
- etc...

2. Les principes clés de Svelte

Les principes clés

Svelte propose un changement de paradigme : plutôt que de se reposer sur l'utilisation du **VirtualDOM** pour manipuler le **DOM** comme le fait **React** par exemple, ou du **ShadowDom** comme pour les **WebComponents**, Svelte va générer toute la logique lors de la compilation du code.

Cette approche apporte des bénéfices majeurs :

- Les performances sont grandement améliorées car il y a moins de code à exécuter lors du changement d'état d'un composant.
- Le poids du code une fois compilé est largement inférieur aux autres solutions du marché.
- Le code est compilé en **Javascript** natif et n'a besoin d'aucune dépendance pour fonctionner.
- Le **Javascript** dans l'**HTML**, et pas l'inverse comme avec **React** !

3. L'environnement de développement

Installation d'un environnement de développement

Pour cette formation nous allons avoir besoin de différentes choses :

- Un compte **Github** pour héberger votre code. 
- Un **I.D.E** (Environnement de développement) 
- **NodeJS et NPM.** 
- Création d'un nouveau projet et installation de **Svelte** 

ⓘ Information

Nous verrons l'installation de **Svelte** plus tard. Pour le moment nous allons utiliser le [Playground](#) du site officiel.
Pour cela vous pouvez créer un compte afin de pouvoir sauvegarder votre code.

4. Les bases de la syntaxe de Svelte

La syntaxe de Svelte

Les composants **Svelte** s'écrivent dans des fichiers `.svelte`.

Ils seront ensuite traduits par le compilateur en code **JavaScript** natif.

Afin de simplifier le développement, les composants **Svelte** contiennent de l'**HTML** classique. Pas besoin d'apprendre un nouveau langage.

Voici un exemple de composant très simple qui va retourner du texte dans un paragraphe :

MyComponent.svelte

```
1 <p>Hello World !</p>
```

Pour utiliser ce composant :

App.svelte

```
1 <script>
2   import MyComponent from './MyComponent.svelte';
3 </script>
4
5 <MyComponent /> <!-- Hello World -->
```

Essayons sur le [Playground](#) de Svelte !

Dynamiser le composant

Imaginons que l'on souhaite afficher une variable et non pas un texte en dur.

Pour cela nous allons ajouter une balise `<script>` :

MyComponent.svelte

```
1 <script>
2   const name = "John";
3 </script>
4
5 <p>Hello { name } !</p>
```

App.svelte

```
1 <script>
2   import MyComponent from './MyComponent.svelte';
3 </script>
4
5 <MyComponent /> <!-- Hello John -->
```

Astuce

Le code entre les accolades est au format JavaScript, il est donc possible d'utiliser des fonctions.

```
1 <p>Hello { name.toUpperCase() }</p>
```

Ajouter des propriétés

On pourrait avoir besoin d'utiliser le même composant avec des propriétés différentes.

Pour cela il suffit d'exporter la variable `name` pour qu'elle devienne une propriété du composant :

MyComponent.svelte

```
1 <script>
2   export let name;
3 </script>
4
5 <p>Hello { name } !</p>
```

App.svelte

```
1 <script>
2   import MyComponent from './MyComponent.svelte';
3 </script>
4
5 <MyComponent name="John"/> <!-- Hello John -->
6 <MyComponent name="Bill"/> <!-- Hello Bill -->
```

ⓘ Important

Une propriété est par définition une variable qui peut prendre plusieurs valeurs.
Par conséquent, il faut utiliser une `let` et non pas une `const` !

Ajouter des propriétés

On peut avoir envie de mettre une valeur par défaut pour chaque propriété, pour cela, on lui affecte une valeur à l'initialisation :

MyComponent.svelte

```
1 <script>
2   export let name = "World";
3 </script>
4
5 <p>Hello { name } !</p>
```

App.svelte

```
1 <script>
2   import MyComponent from './MyComponent.svelte';
3 </script>
4
5 <MyComponent /> <!-- Hello World -->
6 <MyComponent name="John"/> <!-- Hello John -->
```

① Important!

Par convention, si une propriété n'a pas de valeur par défaut, alors on va lui attribuer la valeur `undefined`.

Le compilateur de **Svelte** affichera un avertissement dans la console si ce n'est pas fait.

```
export let name = undefined;
```

Les propriétés raccourcies

En Svelte, si le nom d'une propriété HTML et sa valeur sont identiques, on peut alors utiliser une écriture simplifiée :

```
1 <script>
2   const src = "https://picsum.photos/200";
3   const width = "200px";
4   const alt = "Lorem ipsum dolor sit amet."
5 </script>
6
7 <img src={src} width={width} alt={alt} />
```

On peut simplifier l'écriture comme ceci :

```
8 <img {src} {width} {alt} />
```

Utilisation de TypeScript

Si on le souhaite, on peut utiliser **Svelte** avec **TypeScript**. Dans ce cas, il faut le préciser grâce à l'attribut `lang` de la balise `<script/>`.

```
1 <script lang="ts">
2   export let name : string | undefined = "Bill";
3 </script>
4
5 <p>Hello {name} !</p>
```

ⓘ Important !

Bien sûr il faudra installer et configurer **TypeScript** dans votre projet.

Ajouter du style

Pour styliser un composant, c'est également très simple, il suffit d'ajouter une balise `<style>` :

```
1 <script>
2   export let name = "Bill";
3 </script>
4
5 <style>
6   p {
7     color: red;
8   }
9 </style>
10
11 <p>Hello {name} !</p>
```

```
1 <script>
2   export let name = "Bill";
3 </script>
4
5 <style>
6   @import './MyComponent.css';
7 </style>
8
9 <p>Hello {name} !</p>
```



Le style déclaré dans un composant est scopé, et ne peut pas impacter les autres éléments de la page !

Utilisation d'un préprocesseur CSS

Si on le souhaite, on peut utiliser **Svelte** avec un préprocesseur CSS comme **scss** ou du **less**.

Dans ce cas, il faut le préciser grâce à l'attribut `lang` de la balise `<style/>`.

```
1 <script>
2   export let name = "Bill";
3 </script>
4
5 <style lang="scss">
6   @import './colors.scss';
7
8   p {
9     color: $color-red;
10  }
11 </style>
12
13 <p>Hello {name} !</p>
```

Attention

Bien sûr il faudra installer et configurer **Scss** ou **Less** dans votre projet.

5. Les événements en Svelte

Les événements.

Le fonctionnement de javascript dans une page web repose sur l'utilisation d'écouteurs d'événement.
En **Svelte** on peut facilement écouter un événement avec `on:event`.

Par exemple :

```
1 <script>
2   function onClick(e) {
3     console.log('Clicked !')
4   }
5 </script>
6
7 <button on:click={e => onClick(e)}>Click me</button>
```

Ou encore

```
8 <button on:click={onClick}>Click me</button>
```

Les événements.

Tous les événements natifs peuvent être écoutés de cette façon :

```
1 <button on:click={ ... }>Click me</button>
2 <button on:focus={ ... }>Click me</button>
3 <button on:blur={ ... }>Click me</button>
4 <button on:mouseenter={ ... }>Click me</button>
5 <button on:mouseleave={ ... }>Click me</button>
6
7 <input type="text" on:change={ ... }> />
8 <input type="text" on:input={ ... }> />
9
10 <form on:submit={ ... }>
11 ...
12 </form>
```

Les événements.

Il est tout à fait possible d'avoir plusieurs écouteurs d'événement sur le même élément :

```
1 <button
2   on:click={ doSomething }
3   on:click={ doSomethingElse }
4   on:focus={ onFocus }
5   on:blur={ onBlur }
6 >
7   Click me
8 </button>
```

Les modificateurs d'événement.

Svelte propose des modificateurs d'événement qui permettent de modifier la façon dont un événement va être interprété. Cela s'avère très pratique et peut vous faire économiser quelques lignes de codes. Prenons un exemple :

```
1 <script>
2   function onSubmit(e) {
3     e.preventDefault();
4     ...
5   }
6 </script>
7
8 <form action="/login" on:submit={onSubmit}></form>
```

On peut simplifier ce code de cette façon :

```
1 <script>
2   function onSubmit(e) { ... }
3 </script>
4 <form action="/login" on:submit|preventDefault={onSubmit}></form>
```

Les modificateurs d'événement.

Voici la liste des modificateurs d'événement et leur fonction :

- `|preventDefault` ----- fait un `event.preventDefault()` avant le callback de l'écouteur
- `|stopPropagation` ----- fait un `event.stopPropagation()` avant le callback de l'écouteur
- `|stopImmediatePropagation` ----- fait un `event.stopImmediatePropagation()` avant le callback de l'écouteur
- `|once` ----- Retire l'écouteur après le premier déclenchement du callback
- `|self` ----- Ne déclenche le callback que si c'est bien cet élément qui a capté l'événement, et pas un parent.
- `|trusted` ----- Ne déclenche le callback que si `event.isTrusted` is `true`
- `|passive` ----- Améliore les performances sur le scroll pour les événements type `touch/wheel`. (automatique dans certains cas)
- `|nonpassive` ----- Met `passive = false` de façon explicite

Bien entendu, on peut ajouter plusieurs modificateurs à un événement :

```
<button on:click|preventDefault|stopImmediatePropagation|once={onClick}>Click me</button>
```

Les événements custom

On peut avoir besoin d'émettre des événements custom depuis un composant, afin qu'ils soient écoutés par leur parents.

C'est tout à fait possible avec la fonction `createEventDispatcher` de Svelte, voici un exemple :

Button.svelte

```
1 <script>
2   import { createEventDispatcher } from 'svelte';
3   const dispatch = createEventDispatcher();
4 </script>
5
6 <button on:click={() => dispatch('btnClick')}>Click me</button>
```

App.svelte

```
1 <script>
2   import Button from './Button.svelte';
3 </script>
4
5 <Button on:btnClick|preventDefault={console.log} />
```

6. La réactivité en Svelte

La réactivité en svelte.

Contrairement à d'autres frameworks comme **React**, les variables déclarées dans un composant **Svelte** sont par nature réactives :

```
1 <script>
2   let count = 0;
3 </script>
4
5 <h1>Vous avez cliqué {count} fois sur le bouton.</h1>
6
7 <button onclick={() => count++}>Click me !</button>
```

Le message "*Vous avez cliqué X fois sur le bouton.*" sera mis à jour à chaque changement de la valeur de **count**, car la variable est réactive.

Essayons sur le [Playground](#) de Svelte !

La réactivité en svelte.

En revanche, si une variable est calculée par rapport à une autre variable, alors sa valeur ne sera pas mise à jour :

```
1 <script>
2   let count = 0;
3   let double = count * 2;
4 </script>
5
6 <h1>Vous avez cliqué {count} fois sur le bouton.</h1>
7 <h2>Double = {double}.</h2>
8
9 <button onclick={() => count++}>Click me !</button>
```

Essayons sur le [Playground](#) de Svelte !

Les variables dérivées.

Pour rendre la variable `double` réactive et donc changer sa valeur dès que `count` change, il faut utiliser le `$:`. On parle alors de variables **dérivées**.

```
1 <script>
2   let count = 0;
3   $: double = count * 2;
4 </script>
5
6 <h1>Vous avez cliqué {count} fois sur le bouton.</h1>
7 <h2>Double = {double}.</h2>
8
9 <button onclick={() => count++}>Click me !</button>
```

⚠️ Attention !

Les variables dérivées demandent beaucoup de mémoire et peuvent altérer la performance globale de votre application si elles sont utilisées trop souvent.

Essayons sur le [Playground](#) de Svelte !

Les effets de bord.

Le `$:` permet également de déclencher une fonction ou une liste d'instructions en fonction du changement de valeur d'une variable ou plusieurs variables.

C'est l'équivalent du `useEffect` React.

En React :

```
1 import {useEffect} from 'React';
2
3 export default MyComponent = ({text}) => {
4   useEffect(() => {
5     console.log(text)
6   }, [text])
7
8   return <p>{text}</p>
9 }
```

En Svelte :

```
1 <script>
2   export let text = undefined;
3   $: console.log(text);
4 </script>
5
6 <p>{text}</p>
```

7. Les conditions

Les conditions dans les templates Svelte

Si l'on souhaite afficher un élément en fonction d'une condition booléenne, il faut utiliser la syntaxe :

```
{#if expression} ... {/if}
```

```
1 <script>
2   let count = 0;
3 </script>
4
5 <button on:click={count++}>Incrémenter</button>
6
7 {#if count > 10}
8   <p>Vous avez cliqué plus de 10 fois</p>
9 {/if}
```

Ici, le texte Vous avez cliqué plus de 10 fois n'apparaîtra que si `count` est supérieur à 10.

Les conditions dans les templates Svelte

Si on souhaite gérer le cas contraire, il faut utiliser la syntaxe :

```
{#if expression} ... {:else} ... {/if}
```

```
1 <script>
2   let count = 0;
3 </script>
4
5 <button on:click={count++}>Incrémenter</button>
6
7 {#if count >= 10}
8   <p>Vous avez cliqué au moins 10 fois</p>
9 {:else}
10  <p>Vous avez cliqué moins de 10 fois</p>
11 {/if}
```

Ici, le texte sera différent en fonction de la valeur de count.

Les conditions dans les templates Svelte

Comme en javascript, il est possible de gérer plusieurs cas avec la syntaxe :

```
{#if expression} ... {:else if} ... {:else} ... {/if}
```

```
1 <script>
2   let count = 0;
3 </script>
4
5 <button on:click={count++}>Incrémenter</button>
6
7 {#if count >= 10}
8   <p>Vous avez cliqué au moins 10 fois</p>
9 {:#else if count >= 5}
10  <p>Vous avez cliqué au moins 5 fois</p>
11 {:#else}
12  <p>Vous avez cliqué moins de 5 fois</p>
13 {/if}
```

Ici, le texte sera différent en fonction de la valeur de `count`.

8. Les boucles

Les boucles dans les templates Svelte

Pour iterer sur des valeurs en **Svelte**, il faut utiliser la syntaxe

```
{#each items as item} ... {/each}
```

Prenons un exemple :

```
1 <script>
2   const users = [ "John", "Bill", "Eric", "Jack", "Jane" ]
3 </script>
4
5 <ol>
6   {#each users as user}
7     <li>{user}</li>
8   {/each}
9 </ol>
```

Ici, les éléments présents dans le tableau **users** seront présentés sous forme de liste ordonnée.

Les boucles dans les templates Svelte

Si on souhaite récupérer l'index de la chaque élément de la boucle, il faut passer un deuxième paramètre :

```
{#each items as item, index} ... {/each}
```

Dans notre exemple précédent :

```
1 <script>
2   const users = [ "John", "Bill", "Eric", "Jack", "Jane" ]
3 </script>
4
5 <ul>
6   {#each users as user, index}
7     <li>{index + 1} - {user}</li>
8   {/each}
9 </ul>
```

Ici, les éléments présents dans le tableau `users` seront présentés sous forme de liste non ordonée mais commençant par un numéro.

Les boucles dans les templates Svelte

En **Svelte** il est possible très simplement de gérer le cas où l'objet itérable est vide :

```
{#each items as item, index} ... {:else} ... {/each}
```

Dans notre exemple précédent :

```
1 <script>
2   const users = []
3 </script>
4
5 <ul>
6   {#each users as user, index}
7     <li>{index + 1} - {user}</li>
8   {:else}
9     <li>Il n'y a aucun utilisateur</li>
10  {/each}
11 </ul>
```

Ici, le tableau `users` est vide. Par conséquent, le texte Il n'y a aucun utilisateur sera affiché.

Les boucles dans les templates Svelte

Il peut être parfois indispensable dans les boucles d'identifier chaque élément par une clé unique.

En **React** il existe l'attribut `key` à placer sur le premier élément de la boucle.

En **Svelte**, c'est plus simple, on précise simplement la clé entre parenthèse dans la boucle :

```
{#each items as item (key)} ... {/each}
```

```
1 <script>
2   const user = [
3     {firstname: "John", lastname: "Doe" },
4     {firstname: "Bill", lastname: "Gates" }
5   ]
6 </script>
7 <ul>
8   {#each users as user, index, (user.firstname)}
9     <li>{index + 1} - {user.firstname} {user.lastname}</li>
10  {/each}
11 </ul>
```

Pour mieux comprendre l'utilité des clés dans les boucles, essayons sur le [Playground](#) de Svelte.

9. Le binding

Le binding

En Svelte, les données circulent généralement du parent à l'enfant. La directive `bind:` permet aux données de circuler dans l'autre sens, de l'enfant au parent.

Le mot clé `bind` permet de faire correspondre un attribut HTML à une variable.

Par exemple ce code peut facilement être simplifié :

```
1 <script>
2   let value = undefined;
3
4   function onInput(e) {
5     console.log(e)
6     value = e.target.value;
7   }
8 </script>
9
10 <input type="text" on:input|preventDefault={onInput}>
11 <h1>value = {value ?? 'inconnue'}</h1>
```

Le binding

Avec le mot clé `bind:` :

```
1 <script>
2   let text = undefined;
3 </script>
4
5 <input type="text" bind:value={text}>
6 <h1>value = {text ?? 'inconnue'}</h1>
```

Encore plus simple si le nom de la propriété et sa valeur sont identiques :

```
1 <script>
2   let value = undefined;
3 </script>
4
5 <input type="text" bind:value>
6 <h1>value = {value ?? 'inconnue'}</h1>
```

Le binding

Voici des exemples de mise en pratique du binding :

```
1 <script>
2   let size = "16";
3 </script>
4 <input type="range" bind:value={size} min="1" max="100" />
5 <h1 style={`font-size: ${size}px;`}>Font size : {size}px</h1>
```

```
1 <script>
2   let direction = "row";
3 </script>
4 <select bind:value={direction}>
5   <option value="row" selected={direction === "row"}>row</option>
6   <option value="column" selected={direction === "column"}>column</option>
7 </select>
8 <div style="display: flex; gap: 10px; flex-direction: {direction};">
9   <p>Lorem ipsum dolor sit amet.</p>
10  <p>Lorem ipsum dolor sit amet.</p>
11  <p>Lorem ipsum dolor sit amet.</p>
12 </div>
```

A screenshot of a code editor showing a file named `App.svelte`. The code is written in Svelte, a front-end framework. It includes a script block with several functions: `toggleList`, `addCount`, and `disableCountry`. The `addCount` function uses a for loop to increment a variable. The `disableCountry` function uses map to modify an array of objects based on a country value. Below the script is a `<main>` element containing a `for` loop that calls the `addCount` function.

```
15  });
16
17  const toggleList = () => {
18    show = !show;
19  };
20
21  const addCount = (quant) => {
22    for (let i = 0; i < quant; i++) {
23      number += 1;
24    }
25  };
26
27  const disableCountry = (country) => {
28    data = data.map((item) => {
29      if (item.country === country) {
30        item.disabled = !item.disabled;
31      }
32      return item;
33    });
34  };
35  </script>
36  <main> {#for {number} >{addCount(1)}>add
```

En conclusion

La syntaxe de Svelte se veut très simple. C'est la raison pour laquelle le framework est largement apprécié des développeurs, comme en témoigne le site [StateOfJS](#).

Bien entendu, il existe de nombreuses règles propres à Svelte que nous allons aborder dans les prochains chapitres.

Suite de la formation →

Travaux pratiques

Création d'une todo list

Création d'une todo list

what needs to be done?		
todo	done	
<input type="checkbox"/> write some docs		<input checked="" type="checkbox"/> buy some milk
<input type="checkbox"/> start writing blog post		<input checked="" type="checkbox"/> mow the lawn
<input type="checkbox"/> feed the turtle		
<input type="checkbox"/> fix some bugs		

Démonstration sur le [Playground](#)

Nous allons mettre en application les différents points que nous avons abordés ce matin.

L'objectif du TP:

- On saisit une tâche dans le champs texte, quand on tape **Entrée**, une tâche s'ajoute à la liste.
- Si on coche une case dans la liste **todo**, la tâche passe dans l'autre liste et la case est cochée.
- Si on décoche une case dans la liste **done**, la tâche repasse dans l'autre liste et la case est décochée.
- Au click sur le bouton **delete** la tâche est supprimée de la liste où elle se trouve.
- **BONUS :**Ajout d'un bouton pour éditer chaque tâche de la liste **todo**.

Temps alloué : 4 heures

Au programme de cette formation

Jour 2:

Matin

- 1. La syntaxe avancée en **Svelte**
- 2. Le cycle de vie des composants
- 3. La gestion des attributs **class** et **style**
- 4. Les balises spéciales (<svelte:head>, <svelte:window>, ...)
- 5. La directive **use**:
- 6. Les stores
- 7. Les transitions



Au programme de cette formation

Jour 2:

Après midi

1. Présentation de **Sveltekit**, le framework basé sur **Svelte**.
2. Installation et démarrage d'un nouveau projet **SvelteKit**.
3. TP 2 : Présentation du projet de formation (en binôme).

1. La syntaxe avancée de Svelte

Le mot clé @html

Lorsqu'on insère du texte via une variable dans un composant **Svelte**, il est affiché comme du texte brut. Si ce texte contient **des balises HTML**, elles ne seront pas interprétées mais affichées telles quelles :

```
1 <script>
2   let text = "<h1>Lorem Ipsum</h1>";
3 </script>
4
5 {text}
```

Ici, le composant affichera "<h1>Lorem Ipsum</h1>".

```
1 <script>
2   let text = "<h1>Lorem Ipsum</h1>";
3 </script>
4
5 {@html text}
```

Maintenant, le composant affichera **Lorem Ipsum** dans une une balise `h1`.

Les balises HTML seront donc interprétées.

ⓘ Attention !

Cela peut s'avérer dangereux si vous ne connaissez pas d'avance le contenu du code **HTML**, en effet il pourrait contenir des scripts malveillants. Dans ce cas il faudrait utiliser une fonction qui supprime les balises suspectes comme `<script/>`, `<iframe/>` ou `<embed/>`.

Le mot clé @const

On l'a vu précédemment, on peut déclarer des variables au sein de la balise script, mais on pourrait parfois avoir besoin d'en déclarer aussi directement dans le template.

Prenons un exemple :

```
1 <script>
2   const data = [312, 654, 984, 156, 364, 753, 951];
3 </script>
4
5 {#each data as number}
6   <Component number={number} double={number * 2}/>
7 {/each}
8
8 {#each data as number}
9   {@const double = number * 2}
10  <Component {number} {double}>
11 {/each}
```

Ici la variable "double" sera scopée dans la boucle {#each ... }.

Le mot clé @const

C'est particulièrement pratique pour déconstruire des objects complexes :

```
1 <script>
2   const users = [
3     { name: 'John Doe', infos: { age: 45, gender: "homme", size: 180 } },
4     { name: 'Jane Doe', infos: { age: 32, gender: "femme", size: 155 } },
5   ]
6 </script>
7
8 {#each users as user}
9   <p>{user.name}, {user.infos.gender} de {user.infos.age} ans, taile : {user.infos.size}cm.</p>
10 {/each}
```

Simplifions avec @const :

```
11 {#each users as {name, infos}}
12   {@const {age, gender, size} = user.infos }
13   <p>{name}, {gender} de {age} ans, taile : {size}cm.</p>
14 {/each}
```

Le mot clé @debug

Comme son nom l'indique, il permet de débugger votre code en affichant dans la console une valeur à un instant T, et en ajoutant un point d'arrêt dans le code. Reprenons notre exemple précédent :

```
1 <script>
2   const users = [
3     { name: 'John Doe', infos: { age: 45, gender: "homme", size: 180 } },
4     { name: 'Jane Doe', infos: { age: 32, gender: "femme", size: 155 } },
5   ]
6 </script>
7
8 {#each users as user}
9   {@const {age, gender, size} = user.infos }
10  {@debug age, gender, size} <!-- Ces valeurs seront loguées à chaque changement -->
11  <p>{user.name}, {gender} de {age} ans, taille : {size}cm.</p>
12 {/each}
```

⚠️ Attention !

Le @debug ne doit servir qu'en phase de développement, et doit être retiré lors du passage en prod !

2. Le cycle de vie des composants

Le cycle de vie des composants

Comme dans tous les frameworks **Javascript**, les composant **Svelte** passent par plusieurs états :

1. **L'initialisation** : c'est le moment où un composant est créé et "monté" dans le **DOM**.
2. **La mise à jour** : chaque fois qu'une variable réactive ou une propriété change, **Svelte** met à jour le **DOM**.
3. **La destruction** : lorsque le composant est retiré du **DOM**.

Il est possible d'intervenir à chaque étape de ce cycle grâce à des hooks.

L'initialisation

C'est le moment où un composant est créé et monté dans le DOM.

Parfois on doit attendre que le composant soit initialisé pour réaliser une action. Pour cela il faut utiliser le hook `onMount` :

```
1 <script>
2   import { onMount } from 'svelte';
3
4   let offsetHeight;
5   console.log(offsetHeight); // undefined
6
7   onMount(() => {
8     console.log(offsetHeight); // 48
9   });
10 </script>
11
12 <h1 bind:offsetHeight>Titre du composant</h1>
```

Dans cet exemple, on veut connaître la hauteur du h1, mais cette hauteur dépend du contexte.

Par conséquent on ne connaîtra sa valeur réelle qu'après l'initialisation du composant.

La mise à jour

Chaque fois qu'une variable réactive ou une propriété change, **Svelte** met à jour le DOM.

```
1 <script>
2   import { afterUpdate } from 'svelte';
3
4   let count = 0;
5
6   afterUpdate(() => {
7     console.log(`Mise à jour avec count = ${count}`);
8   });
9 </script>
10
11 <button on:click={() => count++}>
12   Cliquez-moi ({count})
13 </button>
```

Dans cet exemple, on crée un bouton qui incrémente un compteur.

Dès que la variable **count** est modifiée, le composant se met à jour grâce à la réactivité.

ⓘ Information

La fonction **afterUpdate** sera lancée juste après le changement d'état.

Il existe aussi une fonction **beforeUpdate** qui s'exécute juste avant le changement d'état.

La destruction

Lorsque le composant est retiré du DOM. Le hook `onDestroy` permet de nettoyer les ressources (supprimer des écouteurs d'événements, annuler des abonnements, etc.).

```
1  <!-- Component.svelte -->
2  <script>
3    import { onDestroy } from 'svelte';
4
5    onDestroy(() => {
6      console.log('Composant détruit');
7    });
8  </script>
9
10 <h1>Hello World</h1>
11
12 <!-- App.svelte -->
13 {#if condition}
14   <Component/>
15 {/if}
```

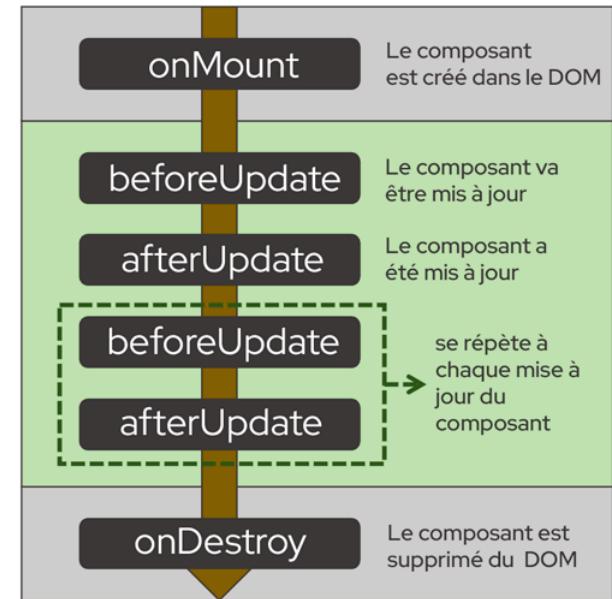
Dans cet exemple, le composant **Component** sera présent dans le **DOM** seulement si la condition est vraie.

Chaque fois qu'elle passera de vraie à fausse, le message sera logué dans la console.

Résumé

Les différentes étapes du cycle de vie d'un composant :

Hook	Moment de déclenchement	Exemple d'usage
onMount	Lorsque le composant est monté dans le DOM.	Charger des données ou configurer le composant
beforeUpdate	Avant chaque mise à jour du DOM (après un changement d'état ou de props).	Préparer des données pour le rendu
afterUpdate	Après chaque mise à jour du DOM.	Animer des éléments ou interagir avec le DOM mis à jour
onDestroy	Juste avant que le composant ne soit retiré du DOM.	Retirer des écouteurs d'événement



3. La gestion des attributs class et style

L'attribut class

En Svelte, les classes s'écrivent exactement comme en HTML classique.

```
1  <p class="paragraph">
2      Lorem ipsum dolor sit <span>amet</span>.
3  </p>
4
5  <style>
6      .paragraph {
7          color: #332233;
8          font-weight: normal;
9      }
10     .paragraph span {
11         color: #FF55FF;
12     }
13 </style>
```

Les selecteurs non utilisés

En revanche si vous essayez d'appliquer du style à un élément qui n'est pas présent dans le composant, un message vous avertira :

```
1 <p class="paragraph">
2   Lorem ipsum dolor sit <span>amet</span>.
3 </p>
4
5 <style>
6   .paragraph {
7     color: #332233;
8     font-weight: normal;
9   }
10  .paragraph div {
11    color: #FF55FF;
12  }
13  .otherClass {
14    font-weight: bold;
15  }
16 </style>
```

--> Unused CSS selector ".paragraph div"

--> Unused CSS selector ".otherClass"

Le mot clé :global

Dans certains cas, on peut avoir besoin de styliser un élément qui n'est pas présent dans le composant.

Imaginons par exemple une div qui contient un composant <Button/>, On veut que le background de la div soit bleu si elle contient un <Bouton/>.

Dans ce cas il faut utiliser le mot clé `:global` :

```
1  <button>click me !</button>
2  <style>
3      :global(div) {
4          background-color: orange;      /* Toutes les divs de la page seront impactées ! */
5      }
6      :global(div:has(button)) {
7          background-color: blue;       /* Seules les divs qui contiennent le button seront impactées ! */
8      }
9  </style>
```



Le `:global` doit être évité dans la mesure du possible car il affecte toute la page dans laquelle le composant est instancié

Les classes conditionnelles

Il arrive fréquemment que l'on souhaite ajouter ou retirer une classe sous certaines conditions. On pourrait tout à fait le faire cette façon :

```
1 <script>
2   let isBold = true;
3 </script>
4
5 <button on:click={() => isBold = !isBold}>Toggle class</button>
6
7 <div class={isBold ? "bold" : null}>
8   Lorem ipsum dolor sit amet consectetur.
9 </div>
```

Vous vous en doutez ce n'est pas idéal, ce n'est pas très lisible si on a une condition complexe ou plusieurs cas à gérer. **Svelte** propose une solution plus claire et plus simple !

Les classes conditionnelles

Il est possible d'ajouter simplement une classe sur un élément de façon conditionnelle avec :

```
<element class:my-class={condition} />
```

```
1 <script>
2   let isBold = true;
3 </script>
4
5 <button on:click={() => isBold = !isBold}>Toggle class</button>
6
7 <div class:bold={isBold}>
8   Lorem ipsum dolor sit amet consectetur.
9 </div>
```

Les classes conditionnelles

Comme on l'a vu précédemment pour les attributs, si le nom de la classe et la condition sont identiques, on peut simplifier l'écriture :

```
1 <script>
2   let bold = true;
3 </script>
4
5 <button on:click={() => bold = !bold}>Toggle class</button>
6
7 <div class:bold>
8   Lorem ipsum dolor sit amet consectetur.
9 </div>
```

Bien sûr, il est tout à fait possible d'avoir plusieurs classes conditionnelles sur un élément, et même de les mixer avec des classes classiques :

```
1 <div class:bold class:light={!bold} class="text-xl text-primary">
2   Lorem ipsum dolor sit amet consectetur.
3 </div>
```

L'attribut style

Il peut arriver qu'on doive utiliser la balise `<style/>` pour styliser un composant. Comme pour les classes, Svelte permet d'utiliser la syntaxe HTML standard :

En React :

```
1 <p style={{color: red, fontSize: 20}}>
2   Lorem ipsum dolor sit amet.
3 </p>
```

En Svelte :

```
1 <p style="color: red; font-size: 20px;">
2   Lorem ipsum dolor sit amet.
3 </p>
```

L'attribut style

Il existe une autre façon d'écrire le style sur un élément qui ressemble aux classes conditionnelles avec :

```
<element style:property={value} />
```

```
1 <p style:color="red" style:font-size="20px">
2   Lorem ipsum dolor sit amet.
3 </p>
```

Encore une fois, ce code peut être raccourci si la variable et la règle CSS à appliquer sont identiques :

```
1 <script>
2   let color = "red";
3   let fontSize = "20px";
4 </script>
5
6 <!-- Cela fonctionne que pour les règles qui ne contiennent pas de tirets ! 😞 -->
7 <p style:color style:font-size={fontSize}>
8   Lorem ipsum dolor sit amet.
9 </p>
```

L'attribut style

L'intérêt de cette méthode est que cela permet d'utiliser une variable **Javascript** pour styliser le composant, ce qui n'est pas possible dans la balise `<style/>`. Par exemple :

```
1 <script>
2   let size = 16;
3 </script>
4
5 <input type="range" min="3" max="50" bind:value={size} />
6
7 <p style:font-size={`${size}px`}>Ce texte fait {size}px</p>
8
9 <style>
10  p {
11    font-size: 16px; /* Impossible de récupérer la variable "size" ici !! */
12  }
13 </style>
```

4. Les balises spéciales

La balise <svelte:head/>

Cette balise permet d'insérer des éléments dans la balise `head` du document.

Component.svelte

```
1  <script>
2    export let title = undefined
3    export let description = undefined
4  </script>
5
6  <svelte:head>
7    <title>{title}</title>
8    <meta name="description" content={description} />
9  </svelte:head></p>
```

App.svelte

```
1  <Component
2    title="Formation Svelte"
3    description="Module de formation Svelte"
4  />
```

Les balises `title` et `meta:description` seront ajoutées au sein de la balise `head`, ou remplaceront les balises existantes.

⚠️ Attention !

Cet élément ne peut apparaître qu'au niveau supérieur de votre composant et ne doit jamais être inclus dans un bloc ou un autre élément.

La balise <svelte:window/>

Cette balise permet d'ajouter des écouteurs d'événements ou de binder des variables à l'objet `window` sans avoir à se soucier de les supprimer lorsque le composant est détruit, ni de vérifier l'existence de `window` lors du rendu côté serveur, Exemple :

```
1 <script>
2   let windowHeight;
3 </script>
4
5 <svelte:window bind:innerWidth={windowWidth} on:resize={console.log}>
6
7 <h1>Largeur de la fenêtre : {windowWidth}px</h1>
```

⚠️ Attention !

Cet élément ne peut apparaître qu'au niveau supérieur de votre composant et ne doit jamais être inclus dans un bloc ou un autre élément.

💡 À savoir

Sur le même principe il existe aussi les balises `svelte:body` et `svelte:document`

La balise <svelte:element/>

Cette balise permet de rendre un élément avec un type spécifié dynamiquement avec `this`.

Si la valeur de `this` est nulle ou indéfinie, l'élément et ses enfants ne seront pas rendus.

```
1  <script>
2    let tag = "ul";
3    const changeList = () => (tag = tag === 'ul' ? 'ol' : 'ul');
4  </script>
5
6  <svelte:element this={tag}>
7    <li>Lorem ipsum</li>
8    <li>Lorem ipsum</li>
9    <li>Lorem ipsum</li>
10 </svelte:element>
11
12 <button on:click={changeList}>Changer le type de liste</button>
13 <button on:click={() => (tag = null)}>retirer la liste</button>
```

ⓘ Attention

Il n'est pas possible de binder des valeurs sur un élément de type `svelte:element` car sa vraie nature n'est pas connue.

La balise <svelte:component/>

Cette balise permet de rendre un composant de manière dynamique, en utilisant le constructeur de composant spécifié par la propriété `this`. Lorsque cette propriété change, le composant est détruit puis recréé.

Si la valeur de `this` est nulle ou indéfinie, l'élément et ses enfants ne seront pas rendus.

```
1  <script>
2    import ComponentA from './ComponentA.svelte';
3    import ComponentB from './ComponentB.svelte';
4
5    let currentComponent = ComponentA;
6  </script>
7
8  <button on:click={() => currentComponent = ComponentA}>Afficher le composant A</button>
9  <button on:click={() => currentComponent = ComponentB}>Afficher le composant B</button>
10
11 <svelte:component this={currentComponent} />
```

La balise <slot/>

Cet élément permet d'afficher les enfants d'un composant.

C'est l'équivalent de la propriété `children` en React :

Widget.svelte :

```
1 <div>
2   <slot>Aucun contenu</slot>
3 </div>
```

App.svelte :

```
1 <Widget />
2
3 <Widget>
4   <p>Lorem ipsum dolor sit amet.</p>
5 </Widget>
```

Le composant `<Widget />` possède un **slot**. Il est donc possible de passer des éléments enfants au composant.

Voilà ce qui sera rendu :

```
1 <div>
2   Aucun contenu
3 </div>
4
5 <div>
6   <p>Lorem ipsum dolor sit amet.</p>
7 </div>
```

La balise <slot/>

Un composant Svelte peut avoir plusieurs **slots**. Dans ce cas il convient de les nommer par un attribut `name` :

Widget.svelte :

```
1 <div>
2   <slot name="header" />
3   <p>Lorem ipsum dolor sit amet.</p>
4   <slot name="footer" />
5 </div>
```

App.svelte :

```
1 <Widget>
2   <h1 slot="header">Hello</h1>
3   <div slot="footer">
4     <p>All rights reserved.</p>
5     <p>Copyright (c) 2024 StudioB04</p>
6   </div>
7 </Widget>
```

Le composant `<Widget />` possède 2 slots : **header** et **footer**.

Il est donc possible de l'utiliser en lui passant des éléments enfants dans des balises `slot`.

Voilà ce qui sera rendu :

```
1 <div>
2   <h1>Hello</h1>
3   <p>Lorem ipsum dolor sit amet.</p>
4   <div>
5     <p>All rights reserved.</p>
6     <p>Copyright (c) 2024 StudioB04</p>
7   </div>
8 </div>
```

La balise <svelte:fragment/>

Cette balise vous permet de placer des éléments dans un `slot` sans qu'il soit wrappé dans un container. C'est l'équivalent de `<></>` ou `<React.Fragment>` en React.

Dans l'exemple précédent, si je ne veux pas que les 2 paragraphes du footer soit dans une `div` :

```
1 <Widget>
2   <h1 slot="header">Hello</h1>
3   <svelte:fragment slot="footer">
4     <p>All rights reserved.</p>
5     <p>Copyright (c) 2024 StudioB04</p>
6   </svelte:fragment>
7 </Widget>
```

Voilà ce qui sera rendu :

```
1 <div>
2   <h1>Hello</h1>
3   <p>Lorem ipsum dolor sit amet.</p>
4   <p>All rights reserved.</p>
5   <p>Copyright (c) 2024 StudioB04</p>
6 </div>
```

La balise <svelte:self/>

Cette balise permet à un composant de s'appeler lui même de façon recursive.

Widget.svelte :

```
1  <script>
2    /** @type {number} */
3    export let count;
4  </script>
5
6  {#if count > 0}
7    <p>{count}...</p>
8    <svelte:self count={count - 1} />
9  {:else}
10   <p>Décolage !!!! 🚀 </p>
11 {/if}
```

App.svelte :

```
1  <Test count={10}>
```

Ici le composant `Widget` va s'appeler lui même avec une propriété `count` qui vaudra `count - 1` si `count > 0`.

Voilà ce qui sera rendu :

```
1  <p>10...</p>
2  <p>9...</p>
3  <p>8...</p>
4  ...
5  <p>3...</p>
6  <p>2...</p>
7  <p>1...</p>
8  <p>Décolage !!!! 🚀 </p>
```

5. La directive :use

La directive :use

La directive `:use` en Svelte est utilisée pour appliquer des actions à des éléments DOM.

Les actions sont des fonctions JavaScript qui permettent d'ajouter des comportements spécifiques à un élément, comme gérer des événements, manipuler des classes, ou interagir avec des API DOM.

highlight.js

```
1  export function highlight(element) {  
2      element.style.backgroundColor = 'yellow';  
3  
4      return {  
5          destroy() {  
6              element.style.backgroundColor = '';  
7          }  
8      };  
9  }
```

Component.svelte

```
1  <script>  
2      import { highlight } from './highlight.js';  
3  </script>  
4  
5  <p use:highlight>  
6      Ce texte sera surligné.  
7  </p>
```

⚠️ Important

Il faut penser à annuler l'effet d'un `:use` quand le composant est détruit. C'est pourquoi il y a une fonction `destroy()` retournée.

La directive :use

On peut transmettre des arguments à l'action. Reprenons l'exemple précédent :

highlight.js

```
1  export function highlight(element, color) {  
2      element.style.backgroundColor = color;  
3  
4      return {  
5          update(newColor) {  
6              element.style.backgroundColor = newColor;  
7          },  
8          destroy() {  
9              element.style.backgroundColor = '';  
10         }  
11     };  
12 }
```

Component.svelte

```
1  <script>  
2      import { highlight } from './highlight.js';  
3      let color = 'yellow';  
4  </script>  
5  
6  <label for="color">Entrez une couleur</label>  
7  <input id="color" bind:value={color} />  
8  
9  <p use:highlight={color}>  
10     Ce texte change de couleur.  
11  </p>
```

Important

Le premier argument fait toujours référence à l'élément DOM sur lequel la directive `:use` est appliquée. Les paramètres suivants sont des arguments de la fonction.

La directive :use

Voici un exemple d'utilisation de `:use` : Créer une action autofocus.

Cette action place le focus sur un élément dès qu'il est monté dans le DOM.

```
1 // autofocus.js
2 export const autofocus = element => {
3   setTimeout(() => element.focus(), 100);
4 }
```

Astuce 1

Ici on ajoute un délai de 100ms pour laisser au navigateur le temps d'ajouter l'élément au DOM avant de placer le focus sur lui.

Astuce 2

Il n'est pas nécessaire de nettoyer l'action avec `destroy()` puisque le champs va disparaître et donc perdre le focus !

```
1 <script>
2   import { autofocus } from './autofocus.js';
3   let showForm = false;
4 </script>
5
6 <button on:click={() => (showForm = !showForm)}>
7   Toggle Form
8 </button>
9
10 {#if showForm}
11   <div class="modal">
12     <input type="text" use:autofocus>
13   </div>
14 {/if}
```

6. Les stores

Les stores

Lorsqu'on utilise un framework comme **Svelte**, ou **React**, il est important de comprendre que les composants sont indépendants les uns des autres et qu'ils sont agnostiques.

C'est à dire qu'ils ne communiquent pas entre eux (sauf avec les événements) et qu'ils n'ont pas connaissance de leur environnement.

Parfois il peut être utile de partager des informations entre plusieurs composants. On pourrait y arriver en passant des données en cascade à tous les composants via des props mais ce serait très fastidieux !

Par exemple, dans certains composants, on peut avoir besoin de savoir si un utilisateur est connecté ou pas.

"En Svelte, les stores sont une façon simple de partager des données entre différents composants."

Les stores

Un store est un objet qui contient une valeur réactive. Lorsqu'on modifie la valeur du store, tous les composants qui l'utilisent sont automatiquement mis à jour.

Svelte propose un module `svelte/store` qui exporte 3 fonctions :

- **Writable** (modifiable) : pour stocker une donnée qui peut être modifiée à tout moment
- **Readable** (lecture seule) : Utilisée pour les données qu'on ne veut pas modifier directement.
- **Derived** (dérivé) : Calculée à partir d'autres stores.

Voici un exemple de store "Writable" :

`store.js`

```
1 import { writable } from 'svelte/store';
2
3 export const count = writable(0); // Store modifiable dont la valeur initiale est 0.
```

Les stores "Writable"

Utilisons ce store dans un composant **Svelte**.

Buttons.svelte :

```
1 <script>
2   import { count } from './store.js';
3 </script>
4
5 <button on:click={() => $count++}>Incrémenter</button>
6 <button on:click={() => $count--}>Décrémenter</button>
```

Ici nous avons un composant qui permet de modifier la valeur du store **count**.

Astuce

Notez l'utilisation du symbole `$` lorsque qu'on fait référence à un store !

Display.svelte :

```
1 <script>
2   import { count } from './store.js';
3 </script>
4
5 <p>Compteur est : {$count}</p>
```

Ici nous avons un composant qui permet d'afficher la valeur du store **count**.

Astuce

La valeur du store sera automatiquement mise à jour dans tous les composants qui l'utilisent.

Les stores "Readable"

Un store "Readable" est utile lorsque vous voulez qu'un composant puisse lire des données sans pouvoir les modifier directement. Cela permet par exemple de partager une donnée à toute l'application.

Imaginons une fonction qui retourne la date :

```
1 import { readable } from 'svelte/store';
2
3 export const currentDate = readable(new Date(), (set) => {
4     const interval = setInterval(() => {
5         set(new Date()); // Met à jour la valeur du store toutes les secondes
6     }, 1000);
7
8     return () => {
9         clearInterval(interval); // Nettoie l'intervalle quand le store n'est plus utilisé
10    };
11});
```



La fonction `set` en deuxième paramètre est utilisée pour mettre à jour la valeur du store.

Les stores "Readable"

Il est désormais possible de connaître la date depuis n'importe quel composant de l'application sans avoir à la recalculer :

```
1 <script>
2   import { currentDate } from './date.store.js'
3 </script>
4
5 <p>La date est {$currentDate.toTimeString()}</p>
```



Le texte du composant sera mis à jour automatiquement toutes les 1000ms puisque la valeur du store change !

Les stores "Derived"

Un store "Derived" est un store basé sur un ou plusieurs autres stores, voici un exemple :

```
1 // store.js
2 import { derived, writable } from 'svelte/store';
3
4 export const price = writable(100);
5 export const quantity = writable(2);
6
7 export const total = derived([price, quantity], ([price, quantity]) => $price * $quantity);
```

💡 Explication

Ici nous avons 2 stores "Writable" : *price* et *quantity*. Le store *total* est calculé par rapport à la valeur courante des autres stores. Le premier argument de la fonction **derived** est un tableau des stores dont il va dépendre. Le deuxième argument est une fonction qui va réaliser un traitement avec les valeurs des stores comme paramètres.

Les stores "Derived"

Et voici un exemple concret de l'utilisation de ce store :

```
1 <script>
2   import { price, quantity, total } from "./store.js"
3 </script>
4
5 <label for="price">Price</label>
6 <input id="price" type="number" bind:value={$price}>
7
8 <label for="quantity">Quantity</label>
9 <input id="quantity" type="number" bind:value={$quantity}>
10
11 <h2>Total : {$total}</h2>
```

💡 Explication

La valeur des champs number est bindée sur la valeur des stores **price** et **quantity**.

En modifiant leur valeur, la valeur de **total** est automatiquement recalculée pour tous les composants qui l'utilisent

En résumé

Les **stores** sont très importants pour le fonctionnement d'une application **Svelte**. Ils permettent de faire communiquer entre eux les différentes briques de votre application.

Les **stores** peuvent être utilisés dans des composants **Svelte**, mais peuvent également être consommés par des utilitaires **JavaScript** via la fonction `get` :

```
1 import { currentDate } from './date.store.js'  
2 import { get } from 'svelte/store';  
3  
4 const value = get(currentDate);
```

Retrouvez toute la documentation sur les **stores** sur le [site officiel de Svelte](#).

7. Les transitions

Les transitions

Svelte propose des outils qui permettent d'animer très simplement l'apparition ou la disparition d'un élément du DOM. Le module `svelte/transition` exporte 7 fonctions d'animations :

`fade` , `blur` , `fly` , `slide` , `scale` , `draw` et `crossfade` .

Voici un exemple :

```
1 <script>
2   import { fade } from "svelte/transition";
3   let visible = false
4 </script>
5
6 <button on:click={() => (visible = !visible)}>
7   Display block
8 </button>
9
10 {#if visible}
11   <p transition:fade>Lorem ipsum dolor sit amet.</p>
12 {/if}
```

Le bloc de texte apparaîtra et disparaîtra en fondue lorsque **visible** sera modifié

Les transitions

Les transitions peuvent prendre des paramètres comme `delay` ou `duration` pour modifier la façon dont l'animation va se jouer.

```
1 <script>
2   import { fade } from "svelte/transition";
3   let visible = false
4 </script>
5
6 <button on:click={() => (visible = !visible)}>
7   Display block
8 </button>
9
10 {#if visible}
11   <p transition:fade={{ delay: 500, duration: 1000 }}>
12     Lorem ipsum dolor sit amet.
13   </p>
14 {/if}
```

Ici l'animation sera retardée de 500 millisecondes et durera 1 seconde.

Information

Chaque transition a des propriétés qui lui sont propres et acceptent certains paramètres.

fade

La transition `fade` joue sur l'opacité de l'élément à afficher. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number, default 0) ----- durée de l'animation en ms.
- **easing** (easing function, default "linear") ----- progression de l'animation.

```
1 <script>
2   import { fade } from 'svelte/transition';
3 </script>
4
5 {#if condition}
6   <div transition:fade={{ delay: 250, duration: 300, ease: cubic }}>
7     fades in and out
8   </div>
9 {/if}
```

blur

La transition `blur` joue sur l'opacité et le flou de l'élément à afficher. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number, default 0) ----- durée de l'animation en ms.
- **easing** ([easing function](#), default "linear") ----- progression de l'animation.
- **opacity** (number, default 0) ----- Opacité de départ de l'animation.
- **amount** (number | string, default 5) ----- Profondeur du flou (en unité css)

```
1 <script>
2   import { blur } from 'svelte/transition';
3 </script>
4
5 {#if condition}
6   <div transition:blur={{ delay: 250, duration: 300, opacity: 0.5, amount: "4rem" }}>
7     blur in and out
8   </div>
9 {/if}
```

fly

La transition `fly` joue sur l'opacité et la position de l'élément à afficher. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number, default 0) ----- durée de l'animation en ms.
- **easing** ([easing function](#), default "linear") ----- progression de l'animation.
- **opacity** (number, default 0) ----- Opacité de départ de l'animation.
- **x** (number | string, default 0) ----- Position de départ de l'animation en X
- **y** (number | string, default 0) ----- Position de départ de l'animation en Y

```
1 <script>
2   import { fly } from 'svelte/transition';
3 </script>
4 {#if condition}
5   <div transition:fly={{ delay: 250, duration: 300, opacity: 0.5, x: 0, y: -100 }}>
6     fly in and out
7   </div>
8 {/#if}
```

slide

La transition `slide` fait glisser l'élément horizontalement ou verticalement. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number, default 0) ----- durée de l'animation en ms.
- **easing** ([easing function](#), default "linear") ----- progression de l'animation.
- **axis** (x | y, default y) ----- Axe de l'animation

```
1 <script>
2   import { slide } from 'svelte/transition';
3 </script>
4
5 {#if condition}
6   <div transition:slide={{ delay: 250, duration: 300, axis: "x" }}>
7     slide in and out
8   </div>
9 {/if}
```

scale

La transition `scale` joue sur l'opacité et l'échelle de l'élément à afficher. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number, default 0) ----- durée de l'animation en ms.
- **easing** ([easing function](#), default "linear") ----- progression de l'animation.
- **start** (number, default 0) ----- échelle au démarrage de l'animation.
- **opacity** (number, default 0) ----- opacité au démarrage de l'animation.

```
1 <script>
2   import { scale } from 'svelte/transition';
3 </script>
4
5 {#if condition}
6   <div transition:scale={{ delay: 0, duration: 300, start: 0.5, opacity:0 }}>
7     scale in and out
8   </div>
9 {/if}
```

draw

La transition `draw` permet d'animer le tracé d'un élément SVG. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number | function, default 800) ----- durée de l'animation en ms.
- **speed** (number, default undefined) ----- vitesse de l'animation.
- **easing** (easing function, default "linear") ----- progression de l'animation.

```
1 <script>
2   import { draw } from 'svelte/transition';
3 </script>
4
5 <svg viewBox="0 0 5 5" xmlns="http://www.w3.org/2000/svg">
6   {#if condition}
7     <path transition:draw={{ duration: 3000, easing: 'elasticInOut' }}>
8       d="M2 1 h1 v1 h1 v1 h-1 v1 h-1 v-1 h-1 v-1 h1 z"
9       fill="none" stroke="cornflowerblue" stroke-width="0.1px" />
10  {/#if}
11 </svg>
```

crossfade

La transition `crossfade` génère une paire de transitions appelées envoi et réception. Lorsqu'un élément est "envoyé", il recherche un élément correspondant "reçu" et génère une transition qui transforme l'élément à la position de son homologue et l'efface. Voici la liste des paramètres autorisés :

- **delay** (number, default 0) ----- délai avant démarrage de l'animation en ms.
- **duration** (number | function, default 800) ----- durée de l'animation en ms.
- **easing** ([easing function](#), default "linear") ----- progression de l'animation.
- **fallback** (function) ----- Une transition de secours.

💡 Exemple

Voici un exemple sur le [Playground](#) de Svelte

Les transitions

Jusqu'à présent, nous avons utilisé le mot clé `transition:` pour réaliser une animation identique à l'apparition et à la disparition de l'élément.

Il est tout à fait possible d'avoir une transition différente, pour cela il faut utiliser les mot clés `in:` et `out:`

```
1 <script>
2   import { fade, slide } from 'svelte/transition';
3 </script>
4
5 {#if condition}
6   <div in:slide={{ duration: 500, y: -150 }} out:fade={{ duration: 300 }}>
7     Lorem ipsum dolor sit amet.
8   </div>
9 {/if}
```

Les callback de transition

Svelte permet de connaître précisément le moment où l'animation démarre ou s'arrête grâce à 4 callbacks :

- `introstart` ----- démarrage de l'animation d'apparition
- `introend` ----- fin de l'animation d'apparition
- `outrostart` ----- démarrage de l'animation de disparition
- `outroend` ----- fin de l'animation de disparition

```
1  {#if condition}
2    <div in:fade out:blur on:introstart={ ... } on:outroend={ ... }>
3      Lorem ipsum, dolor sit amet !!
4    </div>
5  {/if}
```

```
15    });
16
17    const toggleList = () => {
18      show = !show;
19    };
20
21    const addCount = (quant) => {
22      for (let i = 0; i < quant; i++) {
23        number += 1;
24      }
25    };
26
27    const disableCountry = (country) => {
28      data = data.map((item) => {
29        if (item.country == country) {
30          item.disabled = !item.disabled;
31        }
32        return item;
33      });
34    };
35  
```

En conclusion

Svelte propose tous les outils nécessaires pour rendre vos interfaces interactives et fonctionnelles, mais attention toutefois à ne pas abuser des animations car les performances de votre application pourraient en pâtir.

Nous avons abordé les principales fonctionnalités de **Svelte** mais il reste plein de choses à découvrir.

Si vous souhaitez aller plus loin, il existe un [Tutoriel interactif](#) qui vous permettra de valider tous les points que l'on a abordés.

Suite de la formation →

SvelteKit

Le framework basé sur Svelte.

C'est quoi SvelteKit ?

- **SvelteKit** est un meta framework basé sur **Svelte**, permettant de développer rapidement des applications web robustes et performantes.
- **SvelteKit** permet de créer des applications complexes grâce à la puissance et la simplicité de **Svelte**, tout en conservant sa légèreté.
- **SvelteKit** est capable de générer les pages côté serveur (**SSR**) pour des performances et un référencement optimal.
- **SvelteKit** propose des outils pour réaliser un site web complet de A à Z (routeur, optimisation du build, préchargement des pages, optimisation des images, etc.)

Comparaison

SvelteKit est l'équivalent de **Next** pour **React**, ou **Nuxt** pour **VueJS**.

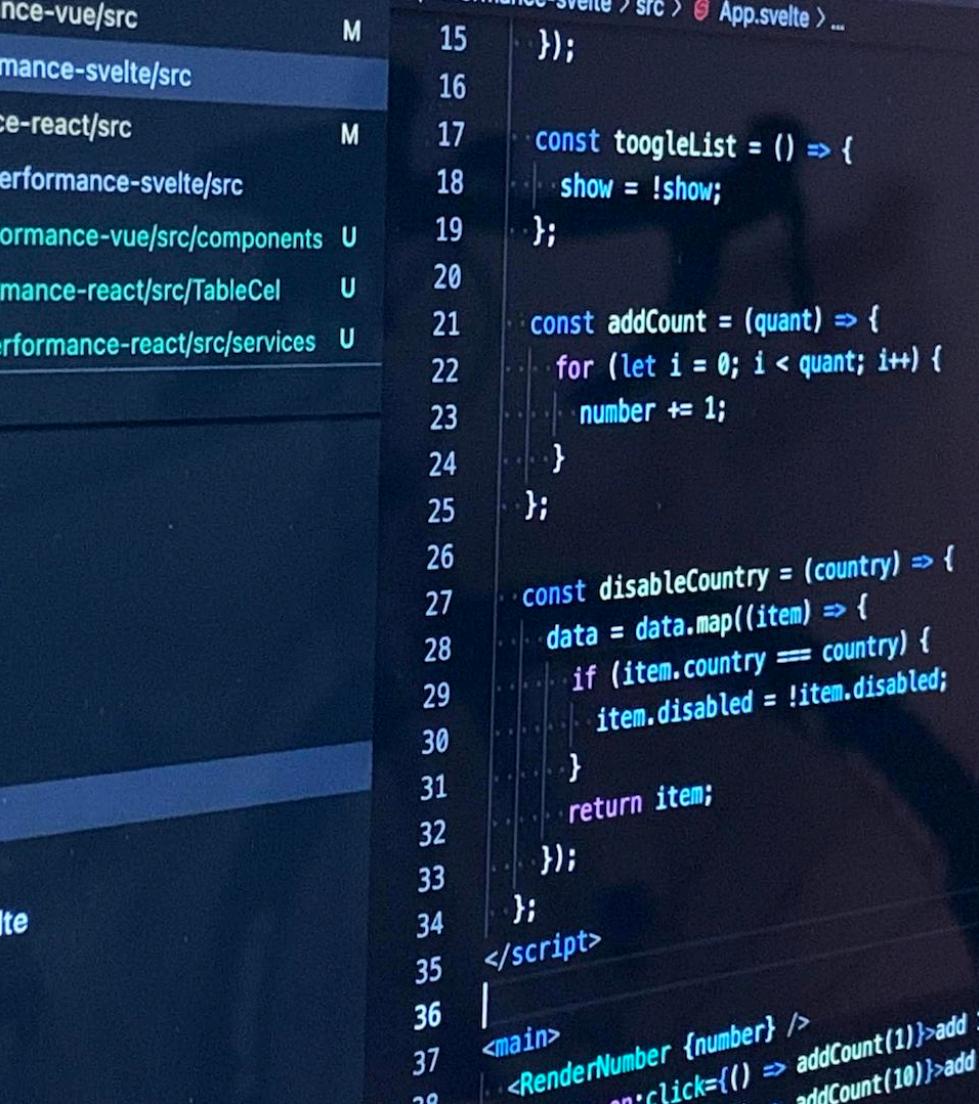
Installons SvelteKit !

Pour créer un nouveau projet, rien de plus simple !

```
1 npx sv create formationSvelte
2 cd formationSvelte
3 npm install
4 npm run dev
```

Important

SvelteKit embarque Vite pour faciliter le développement.



A screenshot of a code editor showing a file named `App.svelte`. The code is written in Svelte, a superset of JavaScript. The editor interface includes a sidebar with project navigation, a top bar with tabs, and a status bar at the bottom. The code itself is a component definition with various functions and components like `<script>`, `<main>`, and `<RenderNumber>`.

```
15 });
16
17 const toggleList = () => {
18   show = !show;
19 };
20
21 const addCount = (quant) => {
22   for (let i = 0; i < quant; i++) {
23     number += 1;
24   }
25 };
26
27 const disableCountry = (country) => {
28   data = data.map((item) => {
29     if (item.country == country) {
30       item.disabled = !item.disabled;
31     }
32     return item;
33   });
34 };
35 </script>
36
37 <main>
38   <RenderNumber {number} />
39   <button on:click={() => addCount(1)}>add 1</button>
40   <button on:click={() => addCount(10)}>add 10</button>
41   <button on:click={() => addCount(100)}>add 100</button>
42 </main>
```

Organisation des fichiers

Si l'installation s'est déroulée comme prévu, vous devriez avoir des dossiers et des fichiers dans votre répertoire `/formationSvelte`. Voici les plus importants :



svelte.config.js

Ce fichier permet de configurer votre projet SvelteKit, et notamment les adapters.

Les adapters sont des utilitaires qui permettent de builder votre projet en fonction de la plateforme sur laquelle il sera déployé (Cloudflare, Netlify, Node, Vercel, static, etc.)

```
1 import adapter from '@sveltejs/adapter-auto';
2
3 /** @type {import('@sveltejs/kit').Config} */
4 const config = {
5   kit: {
6     adapter: adapter()
7   }
8 };
9
10 export default config;
```

vite.config.js

Ce fichier permet de gérer la configuration de Vite.

Vous pouvez y créer des alias de dossiers, changer le répertoire de dev ou de build, ajouter des plugins, etc.

```
1 import { sveltekit } from '@sveltejs/kit/vite';
2 import { defineConfig } from 'vite';
3
4 export default defineConfig({
5   plugins: [sveltekit()]
6 });
```

app.html

C'est le point d'entrée de votre application. Toutes les pages seront basées sur ce fichier de base.

```
1  <!doctype html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <link rel="icon" href="%sveltekit.assets%/favicon.png" />
6      <meta name="viewport" content="width=device-width, initial-scale=1" />
7      %sveltekit.head%
8    </head>
9    <body data-sveltekit-preload-data="hover">
10      <div style="display: contents">%sveltekit.body%</div>
11    </body>
12  </html>
```

💡 Remarque

Notez les blocs

%sveltekit.head% - qui contiendra tout ce qui sera injecté via la balise `svelte:head`

%sveltekit.body% - qui contiendra vos composants

%sveltekit.assets% qui fait référence à votre dossier `/public`.

Le dossier /static

C'est le dossier qui contient tous les assets de votre projet, accessibles depuis le front.

Vous pouvez y stocker des images, des vidéos, des fichiers CSS, etc. Ils seront accessibles depuis vos composants à la racine de votre projet.

Exemple :

```
1   └── static
2       └── images
3           └── photo.jpg
```

```
1 <div>
2     
3 </div>
```

Le dossier /src

C'est le dossier qui contient toutes les sources de votre projet et notamment les routes. Vous êtes libres d'y créer autant de sous dossiers que vous voulez, et créer des alias dans le fichier `svelte.config.js` si besoin. Exemple :



```
1  /** @type {import('@sveltejs/kit').Config} */
2  const config = {
3    ...
4    kit: {
5      alias: { '@utils': './src/utils/' }
6    }
7  };
```

```
1  <script>
2    import { formatDate } from "@utils/date";
3  </script>
```

Créons un dossier /components

Dans le dossier `/src`, nous allons créer un sous-dossier **components**, qui contiendra tous les composants de notre application, et créer l'alias correspondant :

```
1  /** @type {import('@sveltejs/kit').Config} */
2  const config = {
3    ...
4    kit: {
5      alias: {
6        /* vous pouvez nommer l'alias comme vous le souhaitez ! */
7        '@components': './src/components'
8      }
9    }
10};
```

Les routes

Les routes avec SvelteKit

De quoi parle-t-on ?

Les routes désignent les chemins par lesquels on accède à tel ou tel écran de votre application ou page de votre site web.

Imaginons que notre site contienne une page de contact, et que nous voulions y accéder via l'URL `/contact`. Dans ce cas, il suffit de créer un sous-dossier `/contact` dans le dossier `/src/routes` et d'y ajouter un fichier `+page.svelte`.

⚠️ Important !

Le préfixe `+` est important, il indique à **SvelteKit** que ce fichier est lié au routage.

Chaque sous dossier du dossier `/src/route` correspond à une URL.

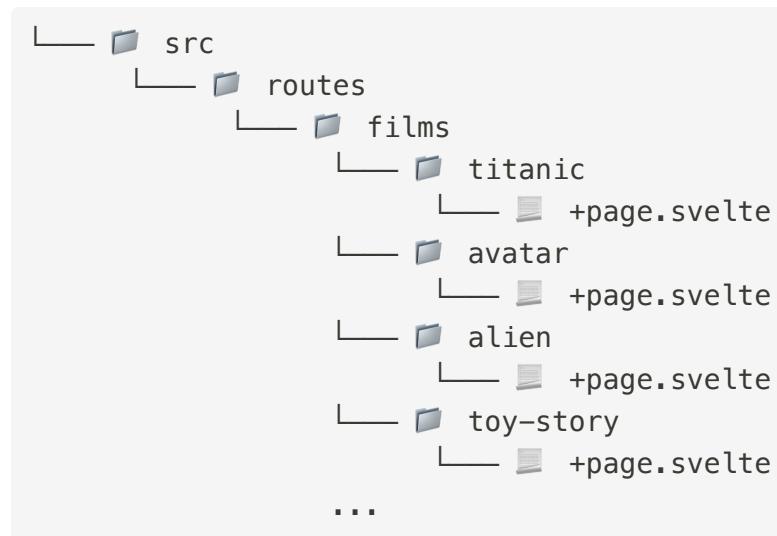
Ainsi le fichier `/src/route/contact/newsletter/+page.svelte` sera accessible via l'URL `/contact/newsletter`.

Le fichier `+page.svelte` à la racine du dossier `/src/routes` correspond à la page d'accueil de votre projet.

Les routes dynamiques

Il est possible de gérer des routes dynamiques, c'est à dire, qui contiennent un "slug".

Imaginons que vous concevezz un site sur le cinéma. Vous souhaitez créer des pages pour des films, vous pourriez tout à fait créer un dossier et un fichier `+page.svelte` par film mais cela serait un peu compliqué à gérer.



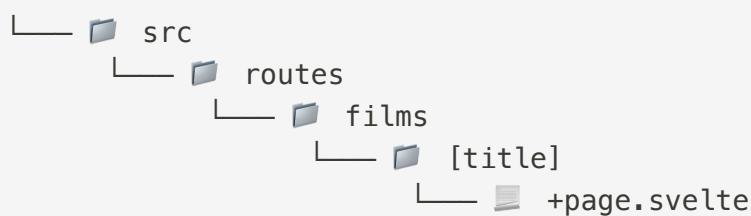
Chaque page de film est bien accessible par une URL propre, par exemple `/films/avatar`.

⚠ Oui mais c'est pas top ! 😞

Cette solution est trop compliquée à maintenir, trop lourde à gérer, trop de code dupliqué, etc.

Les routes dynamiques

Dans ce cas précis, nous allons utiliser une route dynamique. Pour ce faire, le dossier doit être entre crochets :



Maintenant, tous les chemins derrière l'URL `/films/XXXX` feront référence au bon template.

💡 Beaucoup mieux! 😊

Cette solution est plus simple à maintenir, un seul fichier à maintenir.

Toutes ces routes feront appel au même fichier `+page.svelte`:

- `/films/titanic`
- `/films/avatar`
- `/films/alien`
- `/films/toy-story`
- `/films/...`

Les contrôleurs

Chaque route peut avoir un contrôleur spécifique pour gérer la donnée à afficher. Si on reprend notre exemple de tout à l'heure, lorsque je visite l'URL `/films/avatar` je souhaite afficher les données du film Avatar.

Pour cela il faudra faire un appel à une base de donnée ou une API. C'est le rôle du contrôleur.

Les contrôleurs doivent être nommés `+page.js` (ou `+page.ts` si vous utilisez typescript), et être dans le dossier de la route associée :



Les contrôleurs

Voici un exemple de contrôleur :

```
1  /** /src/films/+page.js */
2  export async function load(route) {
3      const data = await fetch(`https://cinema-api.com/${route.params.title}`);
4      return data;
5  }
```

```
1  <!-- /src/films/+page.svelte -->
2  <script>
3      export let data = undefined;
4  </script>
5
6  <h1>{data.title}</h1>
7  <p>{data.content}</p>
```

Le contrôleur va instancier le composant `+page.svelte` avec la propriété **data** qui contiendra la réponse de l'API.

⚠️ Attention !

Notez que le script est asynchrone, ce qui va provoquer un délai dans l'affichage de la page.

Les contrôleurs

On a parfois besoin de passer des paramètres dans les URL. Imaginons maintenant que l'on veuille créer une page qui liste les films par année de sortie et par avis.

On pourrait avoir une URL qui ressemblerait à ça : `/films/list?year=2024&rating=5`. Dans le contrôleur on peut récupérer les valeurs de **year** et **rating** avec `urlSearchParams`

```
1  /** /src/films/list/+page.js */
2  export async function load(route) {
3      const data = await fetch(`https://cinema-api.com/list`, {
4          method: "GET",
5          body: JSON.stringify({
6              year: route.url.searchParams.get('year');
7              rating: route.url.searchParams.get('rating');
8          }),
9      });
10
11      return data;
12  }
```

Les layouts

Jusqu'à présent, nous avons traité les pages comme des composants entièrement indépendants. Lors d'une navigation, le composant existant `+page.svelte` sera détruit, et un nouveau le remplacera.

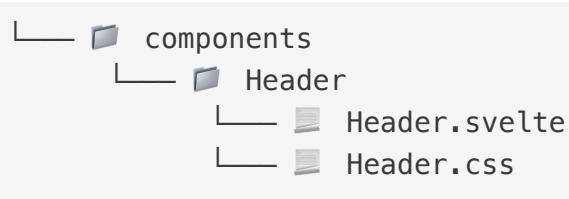
Cependant, certains éléments doivent être visibles sur chaque page, comme une **navigation**, un **header** ou un **footer** par exemple. Au lieu de les répéter dans chaque `+page.svelte`, et qu'il soient détruits à chaque changement d'url, on va utiliser les **layouts**.

Les layouts s'écrivent `+layout.svelte` et se placent dans un sous dossier de `/src/routes`. Toutes les pages qui sont contenues dans ce dossier ou dans un de ses sous-dossiers hériteront de ce layout.



Les layouts

Pour l'exemple, nous allons créer un composant `Header` qui sera ajouté au layout global.



`Header.svelte` :

```
1 <script>
2   export let title = undefined;
3 </script>
4
5 <header>
6   <h1>{title}</h1>
7 </header>
8
9 <style>
10  @import './Header.css';
11 </style>
```

`Header.css` :

```
1 header {
2   padding: 10px;
3   background-color: #eeeeee;
4   border-block-end: 1px solid #333333;
5   margin-block-end: 20px;
6 }
7
8 header h1 {
9   margin: 0;
10 }
```

Les layouts

Dorénavant, il suffit de créer un fichier `+layout.svelte` à la racine du dossier `/src/route` pour y ajouter le header :

`+layout.js` :

```
1  export function load(route) {  
2      return { title: route?.params?.slug };  
3  }
```

`+layout.svelte` :

```
1  <script>  
2      import Header from "../components/Header/Header.svelte";  
3      export let data = undefined  
4  
5      const title = data?.title;  
6  </script>  
7  
8  <Header {title}>  
9  
10 <slot />
```

Gestion des erreurs

SvelteKit propose une solution très simple pour la gestion des pages d'erreur. Il suffit d'ajouter un fichier `+error.svelte` dans le dossier `/src/routes`.

Il est possible de créer une page `+error.svelte` différente pour chaque sous-dossier du dossier `/src/routes`

Les pages d'erreur héritent elles aussi des layouts.

`src/routes/+error.svelte`

```
1 <script>
2   import { page } from '$app/state';
3 </script>
4
5 <h1>{page.error.message}</h1>
6
7 <p>Une erreur s'est produite.</p>
8
9 <a href="/">Retour à l'accueil</a>
```

💡 Information

`$app/state` est un store qui contient des informations sur l'état de page courante.

```
nce-vue/src
mance-svelte/src
ce-react/src
erformance-svelte/src
ormance-vue/src/components U
mance-react/src/TableCel U
erformance-react/src/services U
  M
  15    });
  16
  17    const toggleList = () => {
  18      show = !show;
  19    };
  20
  21    const addCount = (quant) => {
  22      for (let i = 0; i < quant; i++) {
  23        number += 1;
  24      }
  25    };
  26
  27    const disableCountry = (country) => {
  28      data = data.map((item) => {
  29        if (item.country == country) {
  30          item.disabled = !item.disabled;
  31        }
  32        return item;
  33      });
  34    };
  35  </script>
  36
  <main> <button>{number}</button> <button>+count(1)>add</button>
```

En conclusion

SvelteKit est une boîte à outils qui va vous permettre de réaliser simplement des applications complexes et dynamiques.

Nous ne l'avons pas abordé dans ce chapitre mais **SvelteKit** propose de nombreuses options pour générer du **SSR** ou du **CSR**, une **SPA**, des **Service Workers**, etc...

Je vous encourage à jeter un coup d'oeil à la [documentation officielle de SvelteKit](#) pour en apprendre plus.

⚠️ Attention

Depuis une mise à jour récente et le passage à la version 5, la documentation est axée sur la nouvelle syntaxe de Svelte v5.

[Suite de la formation →](#)

Le projet de formation

Création d'un portail sur le cinéma.

Le projet de formation

Pour ce projet en binôme, vous devrez coder un site web basé sur **SvelteKit** et sur l'API **TMDB**.

- Votre projet doit être basé sur la version **4 ou 5** de **SvelteKit**.
- Votre site doit proposer au moins un **champs de recherche**, une page de **résultat de recherche**, une page **film** avec un maximum d'informations, et une page d'erreur.
- Vos pages devront partager **au moins un layout et des composants communs**.
- Vous êtes libres de proposer le design de votre choix (ou de prendre un template libre de droit).
- Votre site doit avoir une déclinaison **clair et sombre** activable depuis un bouton.
- Vous êtes libre d'utiliser toutes les librairies que vous souhaitez mais **vous devrez le justifier**.
- Votre projet doit être disponible sur un **repo Github public**, et documenté (**README.md**).

Voici un exemple : [MyMovieSvelte](#)

Le projet de formation

Avant de commencer, vous allez devoir :

- Créer un compte sur www.themoviedb.org pour obtenir une clé API.
- Créer un nouveau dépôt public sur [github](https://github.com) pour y sauvegarder votre projet.
- Initier un nouveau projet **SvelteKit** et le mettre sur le dépôt. (configurez le comme vous le souhaitez)
- Stocker votre clé API et les informations sensibles dans un fichier `.env` (pensez à me la fournir lors de la remise de votre projet !)

Vous pouvez gagner des points en proposant des choses en plus comme :

- Des pages ou des fonctionnalités supplémentaires (page "acteur", page de recherche avec filtres, etc.)
- Gérer le **SSR**.
- Optimiser le référencement avec les balises `meta` (via la balise `svelte:head`)
- Héberger votre projet sur [Netlify](https://Netlify.com). (voir la documentation sur les adapters)
- etc...

Svelte 5

La plus grosse mise à jour de Svelte à ce jour.

Svelte version 5

Svelte 5 est sorti le 22 Octobre 2024 après 18 mois de développement, grâce au travail de dizaines de contributeurs.

Il s'agit de la mise à jour la plus importante de l'histoire du projet. Svelte 5 est une réécriture complète. L'objectif de cette version, est de créer des applications plus rapides, plus légères et plus fiables.

Malgré tout cela, Svelte 5 reste presque entièrement rétrocompatible avec les versions précédentes. Pour la majorité des utilisateurs, la montée de version sera totalement transparente.

Svelte 5 introduit un nouveau concept : **les Runes**.

Les Runes sont l'équivalent des Hooks de React.

Information

La version 5 de Svelte n'est pas encore totalement implantée en entreprise. Il faudra compter plusieurs mois, voire des années pour que le déploiement soit complet. En revanche tous les nouveaux projets se feront sur la **version 5 de Svelte**.

Une nouvelle syntaxe

Avant de commencer il est important de noter que la nouvelle syntaxe de **Svelte 5** est optionnelle : les composants écrits avec l'ancienne syntaxe continueront de fonctionner avec **Svelte 5** (pour l'instant).

⚠️ Attention !

En revanche, si dans un composant, vous utilisez de la syntaxe de **Svelte 5**, alors il devient incompatible avec **Svelte 4** et doit être complètement réécrit avec la nouvelle syntaxe !

\$props

Les propriétés des composants

\$props

La rune `$props` de **Svelte 5** permet de récupérer toutes les propriétés passées à un composant.

Avec **Svelte 4**, pour déclarer une propriété, il suffisait d'exporter une variable.

Avec **Svelte 5**, il faut utiliser la rune `$props` qui contient toutes les propriétés passées au composants, sans avoir besoin de les déclarer au préalable.

On aura alors tendance à utiliser la déstructuration pour récupérer les props qui nous intéressent.

```
1 <script>
2   export let title = undefined;
3 </script>
4
5 <h1>{title}</h1>
```

```
1 <script>
2   let props = $props();
3 </script>
4
5 <h1>{props.title}</h1>
```

```
1 <script>
2   let { title } = $props();
3 </script>
4
5 <h1>{title}</h1>
```

\$props

Il est toujours possible de passer une valeur par défaut.

```
1 <script>
2   let { title = "Hello World" } = $props();
3 </script>
4
5 <h1>{title}</h1>
```

Il est également possible de renommer une propriété lors de la déstructuration. Cela peut s'avérer utile si la propriété est un mot réservé ou si elle est déjà utilisée.

```
1 <script>
2   // "super" est un mot réservé en JavaScript
3   let { super: title = "Hello World" } = $props();
4 </script>
5
6 <h1>{title}</h1>
```

Pour récupérer les autres propriétés d'un coup, on utilise la syntaxe "..."

```
1 <script>
2   let { title = "Hello World", ...otherProps } = $props();
3 </script>
4
5 <h1>{title}</h1>
```

\$state

La réactivité

\$state

La rune `$state` de **Svelte 5** est une nouveauté majeure qui améliore considérablement la gestion de l'état en permettant de déclarer, manipuler et observer des états globaux de manière plus fluide et intuitive.

C'est l'équivalent du `useState` en React.

Avec Svelte 4:

```
1 <script>
2   let count = 0;
3 </script>
4
5 <button onclick={() => count++}>clicks: {count}</button>
```

Avec Svelte 5:

```
1 <script>
2   let count = $state(0);
3 </script>
4
5 <button onclick={() => count++}>clicks: {count}</button>
```

\$derived

Les variables dérivées

\$derived

La rune `$derived` de Svelte 5 permet de déclarer une variable dérivée. C'est à dire une variable dont la valeur dépend d'une autre variable.

Avec Svelte 4:

```
1 <script>
2   let count = 0;
3   $double = count * 2;
4 </script>
5
6 <button onclick={() => count++}>clicks: {count}</button>
7 <p>double = {double}</p>
```

Avec Svelte 5:

```
1 <script>
2   let count = $state(0);
3   let double = $derived(count * 2);
4 </script>
5
6 <button onclick={() => count++}>clicks: {count}</button>
7 <p>double = {double}</p>
```

\$effect

Les effets de bord

\$effect

La rune `$effect` de Svelte 5 permet d'exécuter des instructions quand la valeur d'une variable évolue.
C'est l'équivalent de `useEffect` en React.

Avec Svelte 4:

```
1 <script>
2   let count = 0;
3   $: console.log('la valeur de count à changé : ', count)
4 </script>
5
6 <button onclick={() => count++}>clicks: {count}</button>
```

Avec Svelte 5:

```
1 <script>
2   let count = $state(0);
3   $effect(() => {
4     console.log('la valeur de count à changé : ', count)
5   })
6 </script>
7
8 <button onclick={() => count++}>clicks: {count}</button>
```

\$bindable

La liaison de valeurs

\$bindable

En **Svelte**, les propriétés circulent du parent vers l'enfant. La rune `$bindable` de **Svelte 5** permet de faire remonter les données de l'enfant vers le parent.

Le composant enfant:
Input.svelte

```
1 <script>
2   let { value = $bindable('default value'), ...props } = $props();
3 </script>
4
5 <input bind:value={value} {...props} />
```

Le composant parent:
App.svelte

```
1 <script>
2   import Input from './Input.svelte';
3   let message = $state('hello');
4 </script>
5
6 <Input bind:value={message} />
```

Ici, la valeur de message du composant parent sera impactée par la valeur du champs du composant enfant.

\$inspect

Pour debugger facilement

\$inspect

La rune `$inspect` de Svelte 5 est utile au développement. Elle permet de loguer dans la console tout changement d'état d'une variable, y compris les changements profonds dans les objets.

```
1 <script>
2   let count = $state(0);
3   let message = $state('hello');
4
5   // will console.log when 'count' or 'message' change
6   $inspect(count, message);
7 </script>
8
9 <button onclick={() => count++}>Increment</button>
10 <input bind:value={message} />
```

Chaque fois que la valeur de `count` ou `message` changera, leur valeur sera loguée dans la console.

💡 Pratique !

la rune `$inspect` ne fonctionne qu'en phase de développement, elle sera ignorée lors du build.

Les évènements

onevent

Les évènements

La version 5 de **Svelte** introduit une nouvelle manière d'écouter les événements.

Avec **Svelte 4** :

```
1 <script>
2   let count = 0;
3 </script>
4
5 <button on:click={() => { count++ }}>
6   {count}
7 </button>
```

Avec **Svelte 5** :

```
1 <script>
2   let count = 0;
3 </script>
4
5 <button onclick={() => { count++ }}>
6   {count}
7 </button>
```

Cela permet notamment d'utiliser l'écriture courte des propriétés si la fonction s'appelle **onclick**

```
1 <script>
2   import { onclick } from '$libs/utils/events';
3   let count = $state(0);
4 </script>
5
6 <button {onclick}>{count}</button>
```

Les évènements customs

Avec Svelte 4, nous avons vu la méthode `createEventDispatcher` qui permet de créer un événement custom :

Button.svelte

```
1 <script>
2   import { createEventDispatcher } from 'svelte';
3   const dispatch = createEventDispatcher();
4 </script>
5
6 <button on:click={() => dispatch("myEvent", "Hello")}>Say Hello !</button>
```

App.svelte

```
1 <script>
2   import Button from "./Button.svelte";
3 </script>
4
5 <Button on:myEvent={console.log} />
```

Les évènements customs

Cette méthode devient obsolète avec **Svelte 5**. Les événements se comportent exactement comme de simples propriétés.

Button.svelte

```
1 <script>
2   const { myEvent } = $props();
3 </script>
4
5 <button onclick={() => myEvent("Hello")}>Say Hello !</button>
```

App.svelte

```
1 <script>
2   import Button from "./Button.svelte";
3 </script>
4
5 <Button myEvent={console.log} />
```

À quoi ça sert ?

Pourquoi changer de syntaxe ?

À quoi ça sert ?

On pourrait penser que tous les changements de syntaxe que l'on a vus n'apportent pas grand chose puisqu'il s'agit simplement d'une façon différente de faire la même chose.

En réalité ils apportent une plus grande clarté au code, et une simplification des applications. En effet il est moins souvent nécessaire d'avoir recours à des stores ou des événements custom pour gérer des états complexes.

Mais la grande force des **Runes**, c'est qu'elles permettent de dissocier les composants de leur logique.

Important

Les fichiers qui vont contenir de la logique **Svelte** basée sur les **Runes** devront avoir une extension en `.svelte.js` (ou `.svelte.ts` si vous utilisez TypeScript).

Voici un exemple

Prenons en exemple ce composant `Counter.svelte` :

```
1  <script>
2    let { title } = $props();
3
4    let count = $state(0);
5    let double = $derived(count * 2);
6
7    $effect(() => console.log('counter was updated to : ', count));
8
9    const increment = () => { count +=1 };
10   </script>
11
12  <h1>{title}</h1>
13  <button onclick={increment}>{count}</button>
```

Voici un exemple

On pourra alors créer un fichier, que l'on appellera `counter.svelte.js` et qui contiendra la logique de notre composant :

```
1  export function createCounter() {
2      let count = $state(0);
3      let double = $derived(count * 2);
4      const increment = () => { count ++ };
5
6      $effect(() => console.log('counter was updated to : ', count));
7
8      return {
9          increment,
10         get count () {
11             return count;
12         },
13         get double () {
14             return double;
15         }
16     }
17 }
```

ⓘ Explication

le fichier exporte une fonction qui va initier le composant. Un même fichier pourra donc être utilisé pour plusieurs composants.

ⓘ Important

Notez que nous devons retourner les variables `count` et `double` sous form de getter pour conserver la réactivité !

Voici un exemple

Le composant `Counter.svelte` pourra ainsi être réécrit comme ceci :

```
1 <script>
2   import { createCounter } from "./counter.svelte.js";
3   const counter = createCounter();
4 </script>
5
6 <h1>{counter.title}</h1>
7 <button onclick={counter.increment}>{counter.count}</button>
```

Notez que pour conserver la réactivité, `counter` ne doit pas être déstructuré !

```
const { title, count } = createCounter();
```

❗ La réactivité est perdue

```
const counter = createCounter();
```

✓ La réactivité est conservée

```
nce-vue/src
mance-svelte/src M
ce-react/src
erformance-svelte/src
ormance-vue/src/components U
mance-react/src/TableCel U
rformance-react/src/services U
ite
te

15    });
16
17    const toggleList = () => {
18        show = !show;
19    };
20
21    const addCount = (quant) => {
22        for (let i = 0; i < quant; i++) {
23            number += 1;
24        }
25    };
26
27    const disableCountry = (country) => {
28        data = data.map((item) => {
29            if (item.country == country) {
30                item.disabled = !item.disabled;
31            }
32            return item;
33        });
34    };
35    </script>
36
|<main> <button>{number}</button><button>+count(1)</button><button>add</button>
```

En conclusion

Svelte 5 et les Runes apportent une nouvelle façon de concevoir le code, en s'approchant de la logique de **React** et de ses **Hooks**.

Bien que la migration d'un projet **Svelte 4** vers **Svelte 5** puisse se faire sans trop de difficultés, cela peut s'avérer complexe pour des gros projets.

Il existe un package `svelte/legacy` qui met à disposition des fonctions utiles pendant la phase de migration.

Fin de la formation →

Merci !

Fin de la formation.
Module Svelte.

Pour en apprendre plus sur **Svelte** et **SvelteKit**, il existe une [documentation](#),
ainsi qu'un [tutorial complet](#) disponibles sur le site officiel de **Svelte**.



Powered by  Sliddev

Merci !