

imaginons maintenant que tu es un expert des bonnes pratiques Sveltekit. Il me semble qu'il faut créer des composants pour les choses qui reviennent souvent comme les grilles pour les numéros, le numéro chance, l'affichage des résultats en bas. Comment est-ce que tu recomposerais `page.svelte` pour faire une application sveltekit bien pensée et bien structurée ?

Pour améliorer la structure de ton application SvelteKit, nous allons appliquer plusieurs principes de **composants modulaires**. L'idée est de découper ton code en composants réutilisables afin de réduire les redondances et de rendre le code plus lisible, maintenable et testable.

Objectif :

1. **Isoler les fonctionnalités répétitives** (comme les grilles de sélection des numéros, l'affichage des résultats, etc.) dans des composants.
2. **Clarifier la logique métier** dans la page principale.
3. **Créer des composants paramétrables et réutilisables** pour éviter la duplication du code.

Étape 1 : Découper en composants

Tu peux créer trois composants principaux pour les sections qui se répètent souvent :

1. **NumberGrid.svelte** : Pour la sélection des 5 numéros.
2. **LuckyNumber.svelte** : Pour la sélection du numéro chance.
3. **ResultsTable.svelte** : Pour l'affichage des résultats avec le tableau.

1. Composant **NumberGrid.svelte**

Ce composant gère la grille de sélection des numéros.

svelte

Copier le code

```
<script lang="ts"> export let selectedNumbers: number[] = []; export let toggleNumber: (num: number) => void; export let maxSelectable: number; const numbers = Array.from({ length: 49 }, (_, i) => i + 1); </script> <div class="grid
```

```
grid-cols-10 gap-2 mb-4"> {#each numbers as number} <Button variant=
{selectedNumbers.includes(number) ? 'default' : 'outline'} on:click={() =>
toggleNumber(number)} disabled={!selectedNumbers.includes(number) &&
selectedNumbers.length >= maxSelectable} class="w-full" > {number} </Button>
{/each} </div> <p class="mb-2">Numéros sélectionnés :</p> <div class="flex flex-
wrap mb-4"> {#each selectedNumbers as num} <span class="inline-block p-2 m-1
border rounded"> {num} </span> {/each} </div>
```

2. Composant **LuckyNumber.svelte**

Ce composant gère la sélection du numéro chance.

svelte

Copier le code

```
<script lang="ts"> export let selectedLuckyNumber: number | null = null; export
let selectLuckyNumber: (num: number) => void; const luckyNumbers = Array.from({
length: 10 }, (_, i) => i + 1); </script> <div class="grid grid-cols-10 gap-2 mb-
4"> {#each luckyNumbers as number} <Button variant={selectedLuckyNumber === number
? 'default' : 'outline'} on:click={() => selectLuckyNumber(number)} class="w-full"
> {number} </Button> {/each} </div> <p class="mb-2">Numéro chance sélectionné :
</p> {#if selectedLuckyNumber} <span class="inline-block p-2 m-1 border rounded
lucky-number"> {selectedLuckyNumber} </span> {/if}
```

3. Composant **ResultsTable.svelte**

Ce composant gère l'affichage des résultats.

svelte

Copier le code

```
<script lang="ts"> export let drawResults: any[] = []; export let selectedNumbers:
number[] = []; export let selectedLuckyNumber: number | null = null; export let
sortColumn: string; export let sortOrder: string; export let sortBy: (column:
string) => void; </script> <Table> <TableHeader> <TableRow> <th class="font-bold
sortable px-4 py-2 text-left" on:click={() => sortBy('Date')}> Date {#if
sortColumn === 'Date'} <span class="sort-arrow">{sortOrder === 'asc' ? '▲' : '▼'}
</span> {/if} </th> <th class="font-bold px-4 py-2 text-left">Vos numéros</th> <th
class="font-bold px-4 py-2 text-left">Combinaison gagnante</th> <th class="font-
```

```

bold sortable px-4 py-2 text-left" on:click={() => sortBy('Gain')}}> Gain {#if
sortColumn === 'Gain'} <span class="sort-arrow">{sortOrder === 'asc' ? '▲' : '▼'}
</span> {/if} </th> </TableRow> </TableHeader> <TableBody> {#each drawResults as
result} <TableRow> <TableCell>{result.date}</TableCell> <TableCell> {#each
selectedNumbers as num} <span class="inline-block p-2 m-1 border rounded
{result.matchingNumbers.includes(num) ? 'highlight' : ''}"> {num} </span> {/each}
<span class="inline-block p-2 m-1 border rounded lucky-number
{result.matchingLuckyNumber ? 'highlight' : ''}"> {selectedLuckyNumber} </span>
</TableCell> <TableCell> {#each result.draw as num} <span class="inline-block p-2
m-1 border rounded">{num}</span> {/each} <span class="inline-block p-2 m-1 border
rounded lucky-number">{result.chance}</span> </TableCell> <TableCell> {result.gain
> 0 ? `${result.gain.toLocaleString('fr-FR', { minimumFractionDigits: 2,
maximumFractionDigits: 2 })}€` : '0€'} </TableCell> </TableRow> {/each}
</TableBody> </Table>

```

Étape 2 : Réorganiser le code dans +page.svelte

Voici maintenant comment utiliser ces composants dans ta page principale +page.svelte :

svelte

Copier le code

```

<script lang="ts"> import { onMount } from 'svelte'; import NumberGrid from
'$lib/components/NumberGrid.svelte'; import LuckyNumber from
'$lib/components/LuckyNumber.svelte'; import ResultsTable from
'$lib/components/ResultsTable.svelte'; import { Button } from
'$lib/components/ui/button'; import Card from
'$lib/components/ui/card/card.svelte'; let selectedNumbers: number[] = []; let
selectedLuckyNumber: number | null = null; let totalSpent = 0; let totalWon = 0;
let totalGamesWon = 0; let netResult = 0; let message = ''; let drawResults: any[]
= []; let isLoading = false; let sortColumn = 'Date'; let sortOrder: 'asc' |
'desc' = 'desc'; let gamesPlayedMessage = ""; // Message pour les grilles jouées
function toggleNumber(number: number) { if (selectedNumbers.includes(number)) {
selectedNumbers = selectedNumbers.filter((n) => n !== number); } else { if
(selectedNumbers.length < 5) { selectedNumbers = [...selectedNumbers, number]; } }
function selectLuckyNumber(number: number) { selectedLuckyNumber = number; }
function resetSelection() { selectedNumbers = []; selectedLuckyNumber = null;
totalSpent = 0; totalWon = 0; totalGamesWon = 0; netResult = 0; message = '';
drawResults = []; sortColumn = 'Date'; sortOrder = 'asc'; } function
calculateResults() { // Ta logique de calcul } function sortBy(column: string) {

```

```

if (sortColumn === column) { sortOrder = sortOrder === 'asc' ? 'desc' : 'asc'; }
else { sortColumn = column; sortOrder = 'asc'; } // Tri des résultats } </script>
<div class="container max-w-5xl mx-auto space-y-4"> <Card class="p-6"> <h1
class="text-2xl font-bold">Simulez vos gains au Loto&nbsp;?</h1> <p class="text-
gray-600"> Imaginez que vous jouez les mêmes numéros depuis le 2 novembre 2019...
</p> </Card> <Card class="p-6"> <NumberGrid selectedNumbers={selectedNumbers}
toggleNumber={toggleNumber} maxSelectable={5} /> </Card> <Card class="p-6">
<LuckyNumber selectedLuckyNumber={selectedLuckyNumber} selectLuckyNumber=
{selectLuckyNumber} /> </Card> <div class="flex justify-center space-x-4"> <Button
on:click={calculateResults} disabled={selectedNumbers.length !== 5 ||
selectedLuckyNumber === null || isLoading} class="px-4 py-2" > {#if isLoading}
Veuillez patienter {:else} Calculer les résultats {/if} </Button> <Button
variant="outline" on:click={resetSelection} class="px-4 py-
2">Réinitialiser</Button> <Button variant="outline" on:click={randomNumbers}
class="px-4 py-2">Numéros aléatoires</Button> <Button variant="outline" on:click=
{randomNumbersUntilWin} class="px-4 py-2">Jouer jusqu'à gagner</Button> </div>
{#if gamesPlayedMessage} <div class="mt-4"> <div class="flex items-center justify-
start p-4 bg-red-100 text-red-800 border-l-4 border-red-500 shadow-lg rounded-lg">
<p class="font-bold text-lg">{gamesPlayedMessage}</p> </div> </div> {/if} {#if
message} <div class="p-4 mt-4 text-white font-bold" class:bg-green-500={netResult
>= 0} class:bg-red-500={netResult < 0}> <p>{message}</p> </div> {/if} {#if
drawResults.length > 0} <ResultsTable drawResults={drawResults} selectedNumbers=
{selectedNumbers} selectedLuckyNumber={selectedLuckyNumber} sortColumn=
{sortColumn} sortOrder={sortOrder} sortBy={sortBy} /> {/if} </div>

```

Avantages de cette approche :

1. **Réutilisation** : Les composants comme `NumberGrid`, `LuckyNumber`, et `ResultsTable` peuvent être réutilisés ailleurs si besoin.
2. **Lisibilité** : La page principale est beaucoup plus claire et concise, avec chaque section bien isolée dans un composant.
3. **Maintenance** : Il est plus facile de maintenir et de tester chaque composant indépendamment.

En suivant cette structure, ton application sera plus modulaire et évolutive !

40

Vous avez dit :

merci c'est génial. Je voudrais tester comme ça mais sans abîmer mon projet qui fonctionne bien actuellement, tu me conseilles de faire comment ? Est-ce qu'il n'y aurait pas aussi une bonne pratique concernant les fonctions insérées directement dans page.svelte ? et aussi le style inséré dans page.svelte ? je crois que comme j'utilise tailwind css, je ne suis pas censé faire de balise style ?

ChatGPT a dit :

ChatGPT

Je suis ravi que ça te plaise ! Passons maintenant à la question de **comment tester cette refactorisation sans compromettre ton projet actuel** et discutons des **bonnes pratiques concernant la gestion des fonctions et du style dans SvelteKit**, surtout si tu utilises Tailwind CSS.

1. Tester sans affecter le projet existant :

Il existe plusieurs façons de tester les modifications sans "casser" ton projet actuel.

A. Créer une branche Git pour le développement :

La manière la plus courante (et recommandée) est de créer une **nouvelle branche** dans ton dépôt Git. Cela te permet de tester tes modifications sans affecter la version stable.

1. Créer une nouvelle branche :

bash

Copier le code

```
git checkout -b refactor-components
```

2. **Faire des modifications dans cette branche** : Modifie les fichiers, ajoute les nouveaux composants, et fais tes tests.
3. **Tester l'application localement** pour vérifier que tout fonctionne comme prévu.
4. **Fusionner les modifications** dans la branche principale une fois que tu es satisfait :

bash

Copier le code

```
git checkout main git merge refactor-components
```

B. Utiliser un projet de test isolé :

Si tu veux vraiment éviter tout risque pour le projet actuel, tu peux créer un **nouveau dossier** et cloner ton projet pour travailler dessus séparément.

1. Copie du projet actuel :

- Fais une copie complète du projet dans un nouveau dossier :

bash

Copier le code

```
cp -r mon-projet mon-projet-refactor
```

2. Modifie et teste les nouveaux composants dans ce projet séparé.

3. Intégration progressive :

- Si tout fonctionne correctement dans la version refactorisée, tu peux ensuite transférer les composants dans ton projet principal.

2. Gestion des fonctions dans SvelteKit :

Effectivement, une bonne pratique est d'isoler les **fonctions** dans des modules ou services séparés, plutôt que de les laisser directement dans ton fichier `+page.svelte`. Cela rend ton code plus modulaire et maintenable.

A. Créer un fichier utilitaire :

Tu peux créer un fichier `src/lib/utils/lotoUtils.ts` où tu vas déplacer les fonctions comme `calculateResults`, `randomNumbers`, `sortDrawResults`, etc.

Exemple :

ts

Copier le code

```
// src/lib/utils/lotoUtils.ts export function calculateResults( selectedNumbers: number[], selectedLuckyNumber: number | null, lotoData: any[], ticketPrice: number ) { // Ta logique de calcul ici } export function randomNumbers(): number[] { const randomNumbers = []; while (randomNumbers.length < 5) { const random = Math.floor(Math.random() * 49) + 1; if (!randomNumbers.includes(random)) { randomNumbers.push(random); } } return randomNumbers; } export function
```

```
sortDrawResults(drawResults: any[], column: string, order: string) { // Tri des résultats }
```

Dans ton fichier `+page.svelte`, tu peux ensuite **importer ces fonctions** :

ts

Copier le code

```
import { calculateResults, randomNumbers, sortDrawResults } from  
'$lib/utils/lotoUtils';
```

3. Bonne pratique pour les styles dans SvelteKit avec Tailwind CSS :

Quand tu utilises **Tailwind CSS**, tu n'as généralement pas besoin d'ajouter des balises

`<style>` directement dans ton composant, car Tailwind offre une approche "utility-first" qui te permet de styliser les éléments directement avec des classes CSS.

A. Utiliser les classes Tailwind :

Dans ton fichier `+page.svelte`, tu peux retirer complètement les balises `<style>` et remplacer les styles personnalisés par des classes Tailwind.

Exemple de transformation :

- Au lieu de définir des styles personnalisés pour `.inline-block`, tu peux directement utiliser les classes Tailwind telles que : html Copier le code ``
- Pour le style conditionnel de `.highlight`, tu peux utiliser Tailwind avec les classes conditionnelles de Svelte : html Copier le code ``

B. Centraliser le style dans `app.css` (ou Tailwind config) :

Si tu as des styles globaux ou des personnalisations de Tailwind, tu peux les centraliser dans le fichier `app.css` ou via la configuration de Tailwind dans `tailwind.config.js`.

Par exemple, si tu veux personnaliser certains styles, tu pourrais ajouter un style dans

`app.css` :

CSS

Copier le code

```
/* src/app.css */ @layer base { .lucky-number { border-color: #d97706; /* Tailwind compliant */ } .highlight { background-color: #34d399; color: white; } }
```

Ou tu pourrais personnaliser Tailwind directement dans le fichier de configuration `tailwind.config.js` pour ajouter des couleurs, des tailles, etc.

Exemple d'ajout dans `tailwind.config.js` :

js

Copier le code

```
module.exports = { theme: { extend: { colors: { 'highlight-green': '#34d399', 'highlight-orange': '#d97706', }, }, }, }, }, }
```

Conclusion et recommandation :

1. **Crée une branche Git** ou une copie de ton projet pour tester les nouvelles modifications sans affecter la version actuelle.
2. **Modularise tes fonctions** en les extrayant dans des fichiers utilitaires pour une meilleure maintenabilité.
3. **Évite d'utiliser des balises `<style>`** dans les composants si tu utilises **Tailwind CSS** et privilégie l'utilisation des classes utilitaires.
4. **Ajoute les styles globaux ou spécifiques dans `app.css`** ou personnalise Tailwind via `tailwind.config.js`.

Cela te permettra d'avoir une application bien structurée, maintenable, et évolutive tout en respectant les bonnes pratiques de développement avec SvelteKit et Tailwind CSS.