

DÉVELOPPEMENT WEB FULL STACK

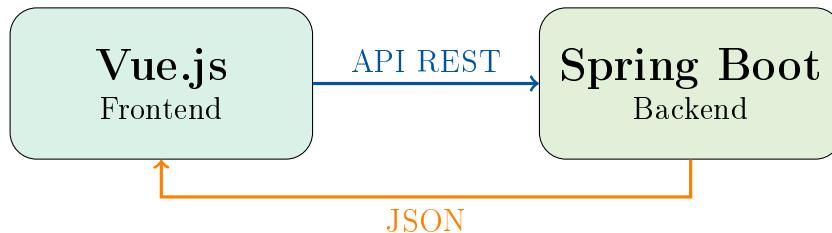
Vue.js + Spring Boot

Chapitre 1

Introduction au Full Stack

Ibrahim ALAME

Année universitaire 2024-2025



Abstract

Ce premier chapitre introduit les concepts fondamentaux du développement web Full Stack. Nous définirons ce qu'est une architecture Full Stack moderne, puis nous explorerons les principes de base de Vue.js pour le frontend et Spring Boot pour le backend. À la fin de ce chapitre, vous serez capable de créer votre premier projet Vue.js, de comprendre la structure des composants, et de mettre en place une API REST simple avec Spring Boot.

Contents

1 Introduction au développement Full Stack	4
1.1 Qu'est-ce que le développement Full Stack ?	4
1.2 L'architecture Client-Serveur	4
1.2.1 Le protocole HTTP	4
1.2.2 Le cycle requête-réponse	5
1.3 SPA vs MPA : Deux approches du web	5
1.3.1 MPA - Multi-Page Application	5
1.3.2 SPA - Single-Page Application	6
1.4 Les API REST	6
1.4.1 Qu'est-ce qu'une API REST ?	6
1.4.2 Les méthodes HTTP et le CRUD	6
1.4.3 Le format JSON	7
2 Vue.js : Principes fondamentaux	7
2.1 Présentation de Vue.js	7
2.1.1 Pourquoi choisir Vue.js ?	8
2.1.2 Vue 2 vs Vue 3	8
2.2 Installation et configuration	8
2.2.1 Prérequis	8
2.2.2 Création d'un projet Vue.js	9
2.3 Structure d'un projet Vue.js	10
2.3.1 Les fichiers importants	10
3 Les composants Vue.js	11
3.1 Qu'est-ce qu'un composant ?	11
3.2 Structure d'un Single File Component (SFC)	12
3.2.1 La syntaxe <code><script setup></code>	13
3.2.2 La section <code><template></code>	13
3.2.3 La section <code><script setup lang="ts"></code>	14
3.2.4 La section <code><style></code>	14
3.3 Les données réactives : <code>ref()</code> et <code>reactive()</code>	14
3.3.1 <code>ref()</code> pour les valeurs primitives	14
3.3.2 <code>reactive()</code> pour les objets	15
3.4 Les fonctions (méthodes)	16
3.5 Les propriétés calculées : <code>computed()</code>	17
3.6 Les props : Communication parent → enfant	19
3.6.1 Déclarer des props avec TypeScript	19
3.6.2 Passer des props depuis le parent	20
3.7 Les événements : Communication enfant → parent	21
4 API REST avec Spring Boot	23
4.1 Rappels sur Spring Boot	23
4.1.1 Création d'un projet Spring Boot	23
4.2 Structure d'un projet Spring Boot	24
4.3 Crédit à la fin	24

4.3.1	L'entité (Model)	24
4.3.2	Le Repository	25
4.3.3	Le Controller REST	26
4.4	Les annotations Spring importantes	27
4.5	Lancer et tester l'application	28
4.5.1	Lancer le serveur Spring Boot	28
4.5.2	La page Whitelabel Error	28
4.5.3	Tester les endpoints de l'API	28
4.5.4	Visualiser les données avec IntelliJ Database	29
4.6	Configuration de la base de données H2	30
4.6.1	Configuration recommandée (mode fichier avec connexions multiples)	30
4.6.2	Connexion depuis IntelliJ IDEA	31
4.6.3	Alternative : Mode mémoire simple (sans accès externe)	31
5	Exercices pratiques	31
5.1	Exercice 1 : Premier composant Vue.js	31
5.2	Exercice 2 : Liste de tâches statique	32
5.3	Exercice 3 : Composants parent-enfant	33
5.4	Exercice 4 : API Spring Boot	33
6	Résumé du chapitre	33
7	Ressources complémentaires	34

1 Introduction au développement Full Stack

1.1 Qu'est-ce que le développement Full Stack ?

Le terme **Full Stack** désigne un développeur ou une approche de développement qui couvre l'ensemble des couches d'une application web, de l'interface utilisateur jusqu'à la base de données.

Définition : Un **développeur Full Stack** est capable de travailler sur toutes les parties d'une application web : le *frontend* (côté client), le *backend* (côté serveur), et la *base de données*.

Une application web moderne est généralement composée de trois grandes couches :

1. **Le Frontend (Client)** : C'est la partie visible de l'application, celle avec laquelle l'utilisateur interagit directement. Elle s'exécute dans le navigateur web et est construite avec HTML, CSS et JavaScript. Dans ce cours, nous utiliserons **Vue.js** comme framework frontend.
2. **Le Backend (Serveur)** : C'est la partie invisible qui gère la logique métier, traite les requêtes du client, et interagit avec la base de données. Elle s'exécute sur un serveur. Nous utiliserons **Spring Boot** (Java) pour cette partie.
3. **La Base de données** : C'est le système de stockage persistant des données de l'application. Nous utiliserons **H2** (base de données en mémoire) pour le développement et **MySQL** pour la production.

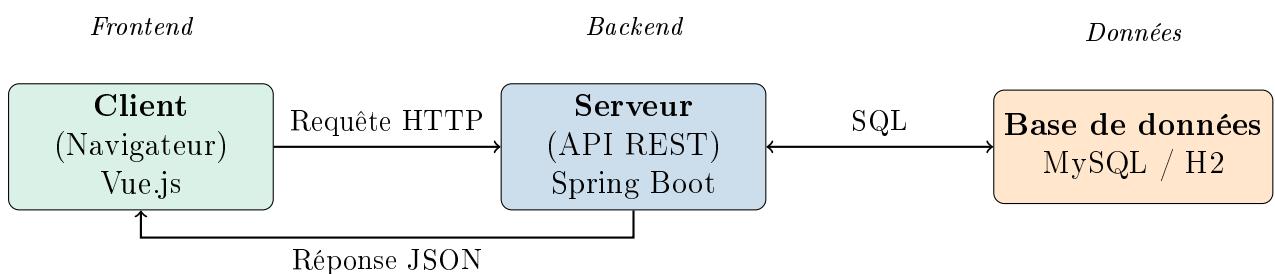


Figure 1: Architecture d'une application web Full Stack

1.2 L'architecture Client-Serveur

L'architecture **client-serveur** est le modèle fondamental du web. Elle repose sur un principe simple : le client envoie des *requêtes*, le serveur les traite et renvoie des *réponses*.

1.2.1 Le protocole HTTP

Toutes les communications entre le client et le serveur web passent par le protocole **HTTP** (HyperText Transfer Protocol). Ce protocole définit :

- Les **méthodes** (ou verbes) : GET, POST, PUT, DELETE, PATCH, etc.
- Les **codes de statut** : 200 (OK), 404 (Not Found), 500 (Server Error), etc.
- Les **en-têtes** (headers) : métadonnées de la requête/réponse
- Le **corps** (body) : données transmises (souvent en JSON)

Note : Le protocole **HTTPS** est la version sécurisée de HTTP. Il chiffre les communications entre le client et le serveur. En production, vous devez *toujours* utiliser HTTPS.

1.2.2 Le cycle requête-réponse

Voici le déroulement typique d'une interaction client-serveur :

1. L'utilisateur effectue une action dans l'interface (clic sur un bouton, soumission d'un formulaire, etc.)
2. Le frontend (Vue.js) construit une requête HTTP et l'envoie au serveur
3. Le serveur (Spring Boot) reçoit la requête et l'analyse
4. Le serveur exécute la logique métier (validation, calculs, etc.)
5. Le serveur interagit avec la base de données si nécessaire
6. Le serveur construit une réponse (généralement en JSON) et la renvoie
7. Le frontend reçoit la réponse et met à jour l'interface utilisateur

1.3 SPA vs MPA : Deux approches du web

Il existe deux grandes approches pour construire des applications web : les **MPA** (Multi-Page Applications) et les **SPA** (Single-Page Applications).

1.3.1 MPA - Multi-Page Application

Dans une application multi-pages traditionnelle :

- Chaque page correspond à un fichier HTML distinct sur le serveur
- Chaque navigation déclenche une nouvelle requête HTTP
- Le serveur génère la page HTML complète et la renvoie
- Le navigateur recharge entièrement la page

C'est l'approche classique du web, utilisée par des technologies comme PHP, JSP, ou Thymeleaf avec Spring MVC.

Avantages : SEO naturel, premier chargement rapide, fonctionne sans JavaScript.

Inconvénients : Rechargements fréquents, expérience utilisateur moins fluide, plus de charge serveur.

1.3.2 SPA - Single-Page Application

Dans une application mono-page moderne :

- Une seule page HTML est chargée initialement
- La navigation se fait *côté client* via JavaScript
- Les données sont récupérées via des appels API (AJAX/Fetch)
- Seules les parties nécessaires de la page sont mises à jour

C'est l'approche adoptée par les frameworks modernes comme Vue.js, React, ou Angular.

Avantages : Expérience utilisateur fluide, interactions riches, moins de données transférées après le chargement initial.

Inconvénients : Premier chargement plus long, SEO plus complexe, nécessite JavaScript.

Notre choix : Dans ce cours, nous utiliserons l'approche **SPA** avec Vue.js pour le frontend, communiquant avec une **API REST** Spring Boot. Cette architecture est aujourd'hui le standard de l'industrie pour les applications web modernes.

1.4 Les API REST

1.4.1 Qu'est-ce qu'une API REST ?

REST (Representational State Transfer) est un style d'architecture pour concevoir des services web. Une **API REST** expose des ressources accessibles via des URLs, manipulables avec les méthodes HTTP standard.

Les principes fondamentaux de REST sont :

1. **Client-Serveur** : Séparation claire des responsabilités
2. **Sans état (Stateless)** : Chaque requête contient toutes les informations nécessaires
3. **Mise en cache** : Les réponses peuvent être mises en cache
4. **Interface uniforme** : Utilisation cohérente des méthodes HTTP
5. **Système en couches** : Le client ne sait pas s'il communique directement avec le serveur final

1.4.2 Les méthodes HTTP et le CRUD

Les opérations CRUD (Create, Read, Update, Delete) correspondent aux quatre opérations de base sur les données. Elles sont mappées sur les méthodes HTTP de la façon suivante :

CRUD	Méthode HTTP	Description	Exemple d'URL
Create	POST	Créer une ressource	POST /api/todos
Read	GET	Lire une ou plusieurs ressources	GET /api/todos GET /api/todos/1
Update	PUT PATCH	Modifier une ressource complète Modifier partiellement	PUT /api/todos/1 PATCH /api/todos/1
Delete	DELETE	Supprimer une ressource	DELETE /api/todos/1

Table 1: Correspondance CRUD - HTTP

1.4.3 Le format JSON

Les API REST modernes utilisent le format **JSON** (JavaScript Object Notation) pour échanger des données. JSON est un format textuel, léger et facilement lisible.

```

1 {
2   "id": 1,
3   "titre": "Apprendre Vue.js",
4   "description": "Suivre le cours Full Stack",
5   "fait": false,
6   "dateCreation": "2024-01-15T10:30:00",
7   "tags": ["formation", "web", "frontend"]
8 }
```

Listing 1: Exemple de données JSON

Les types de données JSON sont :

- **String** : chaîne de caractères entre guillemets doubles
- **Number** : nombre entier ou décimal
- **Boolean** : true ou false
- **Array** : tableau entre crochets []
- **Object** : objet entre accolades {}
- **null** : valeur nulle

2 Vue.js : Principes fondamentaux

2.1 Présentation de Vue.js

Vue.js est un framework JavaScript progressif pour construire des interfaces utilisateur. Créé par Evan You en 2014, Vue.js est devenu l'un des frameworks frontend les plus populaires, aux côtés de React et Angular.

2.1.1 Pourquoi choisir Vue.js ?

Vue.js présente plusieurs avantages qui en font un excellent choix pour ce cours :

- **Progressif** : Vous pouvez commencer simplement et ajouter des fonctionnalités au fur et à mesure
- **Accessible** : La courbe d'apprentissage est douce, surtout si vous connaissez HTML et JavaScript
- **Performant** : Vue.js utilise un DOM virtuel optimisé
- **Bien documenté** : La documentation officielle est excellente et disponible en français
- **Écosystème riche** : Vue Router (navigation), Pinia/Vuex (gestion d'état), etc.
- **Communauté active** : De nombreuses ressources, plugins et composants disponibles

2.1.2 Vue 2 vs Vue 3

Vue 3, sorti en septembre 2020, apporte de nombreuses améliorations :

- **Composition API** : Nouvelle façon d'organiser la logique des composants
- **Meilleures performances** : Rendu plus rapide, bundle plus léger
- **TypeScript** : Meilleur support natif
- **Fragnents** : Plusieurs éléments racine dans un composant
- **Teleport** : Rendre du contenu ailleurs dans le DOM

Note : Dans ce cours, nous utiliserons **Vue 3** avec la **Composition API** et la syntaxe `<script setup>`, combinée à **TypeScript**. C'est l'approche moderne recommandée pour les nouveaux projets Vue 3.

2.2 Installation et configuration

2.2.1 Prérequis

Avant de commencer, assurez-vous d'avoir installé :

- **Node.js** (version 18 ou supérieure) : <https://nodejs.org>
- **npm** (inclus avec Node.js) ou **yarn**
- Un éditeur de code (**VS Code** recommandé avec l'extension Volar)

Pour vérifier votre installation :

```
1 node --version      # Affiche v18.x.x ou supérieur
2 npm --version       # Affiche 9.x.x ou supérieur
```

Listing 2: Vérification de l'installation

2.2.2 Création d'un projet Vue.js

Il existe deux méthodes principales pour créer un projet Vue.js :

Méthode 1 : Vite (recommandé) Vite est un outil de build moderne, très rapide, créé par Evan You (le créateur de Vue.js).

```

1 # Creer un nouveau projet avec TypeScript
2 npm create vite@latest mon-projet --template vue-ts
3
4 # Se deplacer dans le dossier
5 cd mon-projet
6
7 # Installer les dependances
8 npm install
9
10 # Lancer le serveur de developpement
11 npm run dev

```

Listing 3: Création d'un projet avec Vite et TypeScript

Le serveur de développement démarre sur <http://localhost:5173>. Vite offre un rechargeement à chaud (Hot Module Replacement) quasi instantané.

TypeScript : Le template `vue-ts` configure automatiquement TypeScript. Vous bénéficiez ainsi de la vérification de types, de l'autocomplétion améliorée et d'une meilleure maintenabilité du code.

Méthode 2 : Vue CLI Vue CLI est l'outil officiel historique, toujours maintenu.

```

1 # Installer Vue CLI globalement (une seule fois)
2 npm install -g @vue/cli
3
4 # Creer un nouveau projet (selectionner TypeScript dans les
   options)
5 vue create mon-projet
6
7 # Se deplacer dans le dossier
8 cd mon-projet
9
10 # Lancer le serveur de developpement
11 npm run serve

```

Listing 4: Création d'un projet avec Vue CLI

Recommandation : Pour les nouveaux projets Vue 3, privilégiez **Vite** pour sa rapidité et sa simplicité. Vue CLI reste utile pour des projets nécessitant une configuration avancée.

2.3 Structure d'un projet Vue.js

Après la création d'un projet avec Vite et TypeScript, vous obtenez la structure suivante :

```

1 mon-projet/
2 |-- node_modules/          # Dépendances npm (ne pas modifier)
3 |-- public/                 # Fichiers statiques publics
4 |   |-- favicon.ico
5 |-- src/                   # Code source de l'application
6 |   |-- assets/             # Images, styles, etc.
7 |   |-- components/         # Composants réutilisables
8 |   |   |-- HelloWorld.vue
9 |   |   |-- App.vue          # Composant racine
10 |   |   |-- main.ts          # Point d'entrée de l'application
11 |   |   |-- vite-env.d.ts    # Types pour Vite
12 |-- index.html              # Page HTML principale
13 |-- package.json            # Configuration npm et dépendances
14 |-- tsconfig.json           # Configuration TypeScript
15 |-- vite.config.ts          # Configuration de Vite

```

Listing 5: Structure d'un projet Vue.js avec TypeScript

2.3.1 Les fichiers importants

index.html C'est le fichier HTML de base. Il contient un élément `<div id="app">` où Vue.js montera l'application.

```

1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8" />
5     <link rel="icon" type="image/svg+xml" href="/vite.svg" />
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1.0" />
8     <title>Mon Application Vue</title>
9   </head>
10  <body>
11    <div id="app"></div>
12    <script type="module" src="/src/main.ts"></script>
13  </body>
14 </html>

```

Listing 6: index.html

main.ts C'est le point d'entrée TypeScript. Il crée l'instance Vue et la monte sur l'élément `#app`.

```

1 import { createApp } from 'vue'
2 import App from './App.vue'
3

```

```

4 // Creer l'application Vue et la monter sur #app
5 createApp(App).mount('#app')

```

Listing 7: src/main.ts

tsconfig.json Ce fichier configure TypeScript pour le projet.

```

1 {
2   "compilerOptions": {
3     "target": "ES2020",
4     "module": "ESNext",
5     "strict": true,
6     "jsx": "preserve",
7     "moduleResolution": "bundler",
8     "allowImportingTsExtensions": true,
9     "noEmit": true
10   },
11   "include": ["src/**/*.{ts,tsx, vue}"]
12 }

```

Listing 8: tsconfig.json (extrait)

App.vue C'est le composant racine de l'application. Tous les autres composants seront imbriqués à l'intérieur.

3 Les composants Vue.js

3.1 Qu'est-ce qu'un composant ?

Les **composants** sont la pierre angulaire de Vue.js. Un composant est une unité réutilisable et autonome qui encapsule :

- Le **template** (HTML) : la structure visuelle
- Le **script** (JavaScript) : la logique et les données
- Le **style** (CSS) : l'apparence

Définition : Un **composant** Vue.js est un bloc de construction réutilisable qui possède sa propre logique, son propre template et son propre style. Les composants peuvent être imbriqués pour former des interfaces complexes.

Cette approche présente plusieurs avantages :

- **Réutilisabilité** : Un composant peut être utilisé plusieurs fois
- **Maintenabilité** : Le code est organisé en petites unités
- **Testabilité** : Chaque composant peut être testé indépendamment
- **Collaboration** : Différents développeurs peuvent travailler sur différents composants

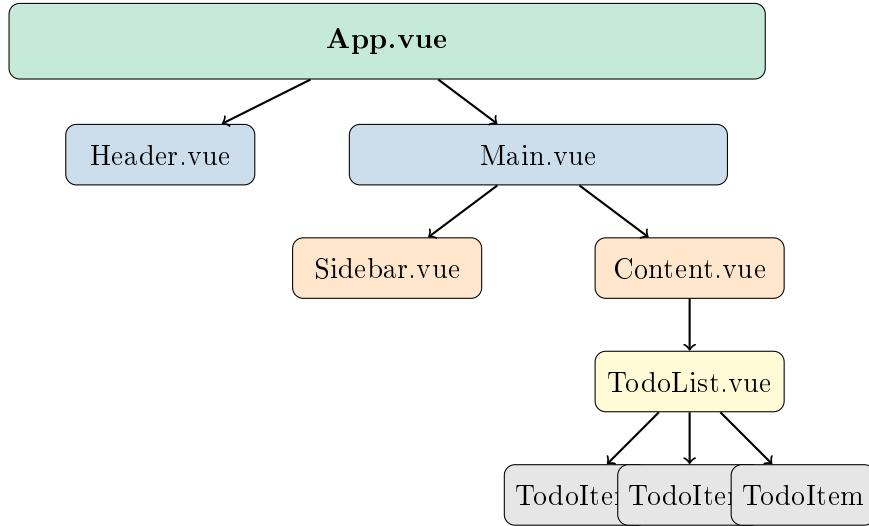


Figure 2: Hiérarchie de composants dans une application Vue.js

3.2 Structure d'un Single File Component (SFC)

Vue.js utilise des fichiers .vue appelés **Single File Components** (SFC). Ces fichiers combinent template, script et style dans un seul fichier.

```

1 <script setup lang="ts">
2 import { ref } from 'vue'
3
4 // Données réactives
5 const titre = ref<string>('Bienvenue')
6 const message = ref<string>('Ceci est mon premier composant
    Vue.js !')
7
8 // Méthodes
9 function changerMessage(): void {
10     message.value = 'Vous avez cliqué sur le bouton !'
11 }
12 </script>
13
14 <template>
15     <!-- Structure HTML du composant -->
16     <div class="mon-composant">
17         <h1>{{ titre }}</h1>
18         <p>{{ message }}</p>
19         <button @click="changerMessage">Cliquez-moi</button>
20     </div>
21 </template>
22
23 <style scoped>
24 /* Styles CSS du composant */
25 .mon-composant {
26     padding: 20px;
  
```

```

27     background-color: #f5f5f5;
28     border-radius: 8px;
29   }
30
31 h1 {
32   color: #42b883;
33 }
34
35 button {
36   background-color: #42b883;
37   color: white;
38   border: none;
39   padding: 10px 20px;
40   border-radius: 4px;
41   cursor: pointer;
42 }
43
44 button:hover {
45   background-color: #369970;
46 }
47 </style>

```

Listing 9: Structure d'un composant Vue.js avec Composition API (MonComposant.vue)

3.2.1 La syntaxe `<script setup>`

La syntaxe `<script setup>` est une simplification syntaxique de la Composition API. Elle offre plusieurs avantages :

- **Moins de boilerplate** : Pas besoin d'exporter explicitement les variables et fonctions
- **Meilleures performances** : Code plus optimisé à la compilation
- **Meilleur support TypeScript** : Inférence de types améliorée
- **Accès direct** : Toutes les variables et fonctions déclarées sont automatiquement disponibles dans le template

L'attribut `lang="ts"` indique que nous utilisons TypeScript.

3.2.2 La section `<template>`

Le template contient le HTML du composant. Vue.js étend HTML avec des fonctionnalités spéciales :

- **Interpolation** : `{{ variable }}` pour afficher des données
- **Directives** : `v-if`, `v-for`, `v-bind`, `v-on`, etc.
- **Binding d'événements** : `@click`, `@input`, `@submit`, etc.
- **Binding d'attributs** : `:class`, `:style`, `:href`, etc.

3.2.3 La section <script setup lang="ts">

Avec la Composition API et TypeScript, on utilise des fonctions importées depuis Vue :

- **ref()** : Créer une donnée réactive primitive
- **reactive()** : Créer un objet réactif
- **computed()** : Créer une propriété calculée
- **watch() / watchEffect()** : Observer les changements
- **onMounted(), onUnmounted()** : Hooks de cycle de vie
- **defineProps()** : Déclarer les props
- **defineEmits()** : Déclarer les événements émis

3.2.4 La section <style>

Le style contient le CSS du composant. L'attribut **scoped** limite les styles au composant actuel, évitant les conflits avec d'autres composants.

Attention : Sans l'attribut **scoped**, les styles s'appliqueront globalement à toute l'application. Utilisez **scoped** par défaut pour éviter les effets de bord.

3.3 Les données réactives : ref() et reactive()

Avec la Composition API, on utilise **ref()** et **reactive()** pour créer des données réactives. Lorsqu'une donnée change, Vue.js met automatiquement à jour le DOM.

3.3.1 ref() pour les valeurs primitives

ref() est utilisé pour les valeurs primitives (string, number, boolean) et peut aussi contenir des objets.

```

1 <script setup lang="ts">
2 import { ref } from 'vue'
3
4 // Types primaires
5 const compteur = ref<number>(0)
6 const nom = ref<string>('Vue.js')
7 const actif = ref<boolean>(true)
8
9 // Accéder/modifier la valeur dans le script : utiliser .value
10 function incrementer(): void {
11   compteur.value++
12 }
13
14 // Dans le template, .value n'est PAS nécessaire
15 // {{ compteur }} fonctionne directement

```

```
16 </script>
```

Listing 10: Utilisation de ref()

Important : Dans le <script>, vous devez utiliser `.value` pour accéder ou modifier la valeur d'une ref. Dans le <template>, Vue.js déballe automatiquement la ref, donc `.value` n'est pas nécessaire.

3.3.2 reactive() pour les objets

`reactive()` est utilisé pour les objets et tableaux. Il rend l'objet entier réactif.

```
1 <script setup lang="ts">
2 import { reactive } from 'vue'
3
4 // Definition d'une interface TypeScript
5 interface Utilisateur {
6   prenom: string
7   age: number
8   adresse: {
9     ville: string
10    codePostal: string
11  }
12}
13
14 // Objet réactif
15 const utilisateur = reactive<Utilisateur>({
16   prenom: 'Jean',
17   age: 25,
18   adresse: {
19     ville: 'Paris',
20     codePostal: '75001'
21   }
22 })
23
24 // Tableau réactif
25 interface Tache {
26   id: number
27   titre: string
28   fait: boolean
29 }
30
31 const taches = reactive<Tache[]>([
32   { id: 1, titre: 'Apprendre Vue', fait: false },
33   { id: 2, titre: 'Créer un projet', fait: false }
34 ])
35
36 // Modifier directement (pas de .value nécessaire)
37 function anniversaire(): void {
```

```

38     utilisateur.age++
39 }
40
41 function ajouterTache(titre: string): void {
42     taches.push({ id: Date.now(), titre, fait: false })
43 }
44 </script>

```

Listing 11: Utilisation de reactive()

Quand utiliser ref() vs reactive() ?

- Utilisez `ref()` pour les valeurs primitives et quand vous avez besoin de réassigner la valeur entière
- Utilisez `reactive()` pour les objets complexes dont vous modifiez les propriétés
- En cas de doute, `ref()` fonctionne dans tous les cas

3.4 Les fonctions (méthodes)

Avec la Composition API, les méthodes sont simplement des fonctions déclarées dans le `<script setup>`. Elles sont automatiquement disponibles dans le template.

```

1 <script setup lang="ts">
2 import { ref } from 'vue'
3
4 const compteur = ref<number>(0)
5
6 // Fonctions simples
7 function incrementer(): void {
8     compteur.value++
9 }
10
11 function decrementer(): void {
12     compteur.value--
13 }
14
15 function reinitialiser(): void {
16     compteur.value = 0
17 }
18
19 // Fonction avec parametre
20 function afficherMessage(texte: string): void {
21     alert(texte)
22 }
23
24 // Fonction avec retour
25 function doubler(n: number): number {

```

```

26     return n * 2
27 }
28 </script>

```

Listing 12: Déclaration de fonctions

Dans le template, on appelle les fonctions avec `@event` :

```

1 <template>
2   <div>
3     <p>Compteur : {{ compteur }}</p>
4     <p>Double : {{ doubler(compteur) }}</p>
5     <button @click="incrementer">+</button>
6     <button @click="decrementer">-</button>
7     <button @click="reinitialiser">Reset</button>
8     <button @click="afficherMessage('Bonjour !')">Dire
9       bonjour</button>
10    </div>
11 </template>

```

Listing 13: Appel de fonctions dans le template

3.5 Les propriétés calculées : computed()

Les **computed** sont des propriétés dont la valeur est calculée à partir d'autres données. Elles sont mises en cache et ne sont recalculées que lorsque leurs dépendances changent.

```

1 <script setup lang="ts">
2 import { ref, reactive, computed } from 'vue'
3
4 const prenom = ref<string>('Jean')
5 const nom = ref<string>('Dupont')
6
7 interface Tache {
8   id: number
9   titre: string
10  fait: boolean
11 }
12
13 const taches = reactive<Tache[]>([
14   { id: 1, titre: 'Tache 1', fait: true },
15   { id: 2, titre: 'Tache 2', fait: false },
16   { id: 3, titre: 'Tache 3', fait: false }
17 ])
18
19 // Computed simple (lecture seule)
20 const nomComplet = computed<string>(() => {
21   return `${prenom.value} ${nom.value}`
22 })
23
24 // Filtrer les taches non terminees

```

```

25 const tachesRestantes = computed<Tache[]>(() => {
26   return taches.filter(t => !t.fait)
27 })
28
29 // Compter les taches restantes
30 const nombreTachesRestantes = computed<number>(() => {
31   return tachesRestantes.value.length
32 })
33
34 // Message dynamique
35 const messageProgression = computed<string>(() => {
36   if (nombreTachesRestantes.value === 0) {
37     return 'Toutes les taches sont terminees !'
38   }
39   return `Il reste ${nombreTachesRestantes.value} tache(s) à
40   faire.`
41 })
42 // Computed avec getter et setter
43 const nomCompletEditable = computed({
44   get(): string {
45     return `${prenom.value} ${nom.value}`
46   },
47   set(valeur: string): void {
48     const parts = valeur.split(' ')
49     prenom.value = parts[0] || ''
50     nom.value = parts[1] || ''
51   }
52 })
53 </script>

```

Listing 14: Utilisation de computed()

```

1 <template>
2   <div>
3     <p>Nom complet : {{ nomComplet }}</p>
4     <p>{{ messageProgression }}</p>
5
6     <ul>
7       <li v-for="tache in tachesRestantes" :key="tache.id">
8         {{ tache.titre }}
9       </li>
10      </ul>
11
12      <!-- Computed avec setter -->
13      <input v-model="nomCompletEditable" />
14    </div>
15 </template>

```

Listing 15: Utilisation des computed dans le template

Quand utiliser computed() vs une fonction ?

- Utilisez `computed()` quand le résultat dépend de données réactives et doit être mis en cache
- Utilisez une fonction quand vous avez besoin de passer des paramètres ou d'exécuter une action avec effets de bord

3.6 Les props : Communication parent → enfant

Les **props** permettent de passer des données d'un composant parent vers un composant enfant. Avec `<script setup>`, on utilise `defineProps()`.

3.6.1 Déclarer des props avec TypeScript

```

1 <script setup lang="ts">
2 // Methode 1 : Interface TypeScript (recommandee)
3 interface Props {
4   titre: string
5   fait?: boolean           // ? = optionnel
6   priorite?: number
7 }
8
9 // defineProps avec valeurs par defaut via withDefaults
10 const props = withDefaults(defineProps<Props>(), {
11   fait: false,
12   priorite: 1
13 })
14
15 // Acceder aux props
16 console.log(props.titre)
17 console.log(props.fait)
18 </script>
19
20 <template>
21   <div class="todo-item" :class="{ termine: fait }">
22     <span>{{ titre }}</span>
23     <span class="priorite">P{{ priorite }}</span>
24   </div>
25 </template>

```

Listing 16: Déclaration de props (TodoItem.vue)

```

1 <script setup lang="ts">
2 // Methode 2 : Validation runtime (similaire a Options API)
3 const props = defineProps({
4   titre: {
5     type: String,
6     required: true

```

```

7   },
8   fait: {
9     type: Boolean,
10    default: false
11  },
12  priorite: {
13    type: Number,
14    default: 1,
15    validator: (value: number) => value >= 1 && value <= 5
16  }
17 })
18 </script>

```

Listing 17: Alternative : validation runtime

3.6.2 Passer des props depuis le parent

```

1 <script setup lang="ts">
2 import { reactive } from 'vue'
3 import TodoItem from './TodoItem.vue',
4
5 interface Tache {
6   id: number
7   titre: string
8   fait: boolean
9   priorite: number
10 }
11
12 const taches = reactive<Tache[]>([
13   { id: 1, titre: 'Faire les courses', fait: false, priorite: 2
14     },
15   { id: 2, titre: 'Reviser le cours', fait: true, priorite: 1 },
16   { id: 3, titre: 'Appeler maman', fait: false, priorite: 3 }
17 ])
18 </script>
19
20 <template>
21   <div class="todo-list">
22     <h2>Ma liste de tâches</h2>
23
24     <!-- Valeurs statiques -->
25     <TodoItem titre="Apprendre Vue.js" :fait="true" />
26
27     <!-- Valeurs dynamiques avec v-bind () -->
28     <TodoItem
29       v-for="tache in taches"
30       :key="tache.id"
31       :titre="tache.titre"
32       :fait="tache.fait"
33     </TodoItem>
34   </div>
35 </template>

```

```

32         :priorite="tache.priorite"
33     />
34   </div>
35 </template>
```

Listing 18: Passage de props (TodoList.vue)

Règle importante : Les props sont en **lecture seule**. Un composant enfant ne doit jamais modifier directement une prop. Si vous avez besoin de modifier une valeur, utilisez des événements pour communiquer avec le parent.

3.7 Les événements : Communication enfant → parent

Pour communiquer de l'enfant vers le parent, on utilise les **événements personnalisés** avec `defineEmits()`.

```

1 <script setup lang="ts">
2 // Declaration des props
3 interface Props {
4   titre: string
5   fait?: boolean
6 }
7
8 const props = withDefaults(defineProps<Props>(), {
9   fait: false
10 })
11
12 // Declaration des evenements avec leurs types
13 const emit = defineEmits<{
14   (e: 'toggle'): void
15   (e: 'supprimer'): void
16   (e: 'modifier', nouveauTitre: string): void
17 }>()
18
19 // Fonctions qui emettent les evenements
20 function onToggle(): void {
21   emit('toggle')
22 }
23
24 function onSupprimer(): void {
25   emit('supprimer')
26 }
27
28 function onModifier(titre: string): void {
29   emit('modifier', titre)
30 }
31 </script>
32
33 <template>
```

```

34   <div class="todo-item" :class="{ fait: fait }">
35     <input
36       type="checkbox"
37       :checked="fait"
38       @change="onToggle"
39     />
40     <span>{{ titre }}</span>
41     <button @click="onSupprimer">X</button>
42   </div>
43 </template>

```

Listing 19: Émission d'événements (TodoItem.vue)

```

1 <script setup lang="ts">
2 import { reactive } from 'vue'
3 import TodoItem from './TodoItem.vue'
4
5 interface Tache {
6   id: number
7   titre: string
8   fait: boolean
9 }
10
11 const taches = reactive<Tache[]>([
12   { id: 1, titre: 'Apprendre Vue', fait: false },
13   { id: 2, titre: 'Créer un projet', fait: true }
14 ])
15
16 function toggleTache(id: number): void {
17   const tache = taches.find(t => t.id === id)
18   if (tache) {
19     tache.fait = !tache.fait
20   }
21 }
22
23 function supprimerTache(id: number): void {
24   const index = taches.findIndex(t => t.id === id)
25   if (index !== -1) {
26     taches.splice(index, 1)
27   }
28 }
29 </script>
30
31 <template>
32   <div class="todo-list">
33     <TodoItem
34       v-for="tache in taches"
35       :key="tache.id"
36       :titre="tache.titre"
37       :fait="tache.fait"

```

```

38      @toggle="toggleTache(tache.id)"
39      @supprimer="supprimerTache(tache.id)"
40    />
41  </div>
42 </template>
```

Listing 20: Écoute des événements (TodoList.vue)

Avantage TypeScript : Avec la syntaxe typée de `defineEmits`, vous bénéficiez de l'autocomplétion et de la vérification de types pour les événements et leurs paramètres.

4 API REST avec Spring Boot

4.1 Rappels sur Spring Boot

Spring Boot est un framework Java qui simplifie la création d'applications Spring. Il offre :

- Une **configuration automatique** intelligente
- Un **serveur embarqué** (Tomcat par défaut)
- Une gestion simplifiée des **dépendances**
- Des **starters** pour démarrer rapidement

4.1.1 Création d'un projet Spring Boot

La méthode la plus simple est d'utiliser **Spring Initializr** :

1. Aller sur <https://start.spring.io>
2. Configurer le projet :
 - Project : Maven
 - Language : Java
 - Spring Boot : 3.x (dernière version stable)
 - Group : com.example
 - Artifact : todoapi
 - Packaging : Jar
 - Java : 17 ou 21
3. Ajouter les dépendances :
 - Spring Web
 - Spring Data JPA
 - H2 Database
4. Générer et télécharger le projet

4.2 Structure d'un projet Spring Boot

```

1 todoapi/
2 | -- src/
3 |   | -- main/
4 |   |   | -- java/
5 |   |   |   | -- com/example/todoapi/
6 |   |   |   |   | -- TodoapiApplication.java      # Point d'entree
7 |   |   |   |   | -- controller/                 # Controllers
8 |   |   |   |   | -- REST
9 |   |   |   |   |   | -- model/                  # Entites JPA
10 |   |   |   |   |   | -- repository/            # Acces aux
11 |   |   |   |   |   | -- service/                # Logique metier
12 |   |   |   |   |   | -- resources/              # Configuration
13 |   |   |   |   | -- application.properties     # Tests
14 |   |   |   | -- test/                         # Dependances
15 | -- pom.xml
16 Maven

```

Listing 21: Structure d'un projet Spring Boot

4.3 Création d'une API REST simple

4.3.1 L'entité (Model)

Une entité représente une table dans la base de données.

```

1 package com.example.todoapi.model;
2
3 import jakarta.persistence.*;
4
5 @Entity
6 @Table(name = "todos")
7 public class Todo {
8
9     @Id
10    @GeneratedValue(strategy = GenerationType.IDENTITY)
11    private Long id;
12
13    @Column(nullable = false)
14    private String titre;
15
16    private String description;
17
18    @Column(nullable = false)
19    private boolean fait = false;
20
21    // Constructeurs
22    public Todo() {}

```

```

23
24     public Todo(String titre, String description) {
25         this.titre = titre;
26         this.description = description;
27     }
28
29     // Getters et Setters
30     public Long getId() { return id; }
31     public void setId(Long id) { this.id = id; }
32
33     public String getTitre() { return titre; }
34     public void setTitre(String titre) { this.titre = titre; }
35
36     public String getDescription() { return description; }
37     public void setDescription(String description) {
38         this.description = description;
39     }
40
41     public boolean isFait() { return fait; }
42     public void setFait(boolean fait) { this.fait = fait; }
43 }
```

Listing 22: model/Todo.java

4.3.2 Le Repository

Le repository fournit les méthodes d'accès aux données.

```

1 package com.example.todoapi.repository;
2
3 import com.example.todoapi.model.Todo;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6 import java.util.List;
7
8 @Repository
9 public interface TodoRepository extends JpaRepository<Todo,
10                                         Long> {
11
12     // Spring Data JPA genere automatiquement l'implementation
13     // de nombreuses methodes : findAll(), findById(), save(),
14     // delete()...
15
16     // Methodes personnalisées (optionnel)
17     List<Todo> findByFait(boolean fait);
18     List<Todo> findByTitreContaining(String titre);
19 }
```

Listing 23: repository/TodoRepository.java

4.3.3 Le Controller REST

Le controller définit les endpoints de l'API.

```

1 package com.example.todoapi.controller;
2
3 import com.example.todoapi.model.Todo;
4 import com.example.todoapi.repository.TodoRepository;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.*;
9
10 import java.util.List;
11 import java.util.Optional;
12
13 @RestController
14 @RequestMapping("/api/todos")
15 @CrossOrigin(origins = "http://localhost:5173") // Autoriser
    Vue.js
16 public class TodoController {
17
18     @Autowired
19     private TodoRepository todoRepository;
20
21     // GET /api/todos - Recuperer tous les todos
22     @GetMapping
23     public List<Todo> getAllTodos() {
24         return todoRepository.findAll();
25     }
26
27     // GET /api/todos/{id} - Recuperer un todo par son ID
28     @GetMapping("/{id}")
29     public ResponseEntity<Todo> getTodoById(@PathVariable Long
        id) {
30         Optional<Todo> todo = todoRepository.findById(id);
31         return todo.map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
32     }
33
34     // POST /api/todos - Creer un nouveau todo
35     @PostMapping
36     @ResponseStatus(HttpStatus.CREATED)
37     public Todo createTodo(@RequestBody Todo todo) {
38         return todoRepository.save(todo);
39     }
40
41     // PUT /api/todos/{id} - Mettre a jour un todo
42     @PutMapping("/{id}")
43     public ResponseEntity<Todo> updateTodo(

```

```

45     @PathVariable Long id,
46     @RequestBody Todo todoDetails) {
47
48     return todoRepository.findById(id)
49         .map(todo -> {
50             todo.setTitre(todoDetails.getTitre());
51             todo.setDescription(todoDetails.getDescription());
52             todo.setFait(todoDetails.isFait());
53             return
54                 ResponseEntity.ok(todoRepository.save(todo));
55         })
56         .orElse(ResponseEntity.notFound().build());
57     }
58
59     // DELETE /api/todos/{id} - Supprimer un todo
60     @DeleteMapping("/{id}")
61     public ResponseEntity<Void> deleteTodo(@PathVariable Long
62         id) {
63         if (todoRepository.existsById(id)) {
64             todoRepository.deleteById(id);
65             return ResponseEntity.noContent().build();
66         }
67     }

```

Listing 24: controller/TodoController.java

4.4 Les annotations Spring importantes

Annotation	Description
@RestController	Indique que la classe est un controller REST (combine @Controller et @ResponseBody)
@RequestMapping	Définit l'URL de base pour tous les endpoints du controller
@GetMapping	Mappe une requête GET sur une méthode
@PostMapping	Mappe une requête POST sur une méthode
@PutMapping	Mappe une requête PUT sur une méthode
@DeleteMapping	Mappe une requête DELETE sur une méthode
@PathVariable	Extrait une variable de l'URL (ex: /todos/{id})
@RequestBody	Déserialise le corps JSON de la requête en objet Java
@CrossOrigin	Autorise les requêtes cross-origin (CORS)
@Autowired	Injection de dépendances automatique

Table 2: Principales annotations Spring pour les API REST

4.5 Lancer et tester l'application

4.5.1 Lancer le serveur Spring Boot

Pour lancer l'application, exécutez la commande suivante depuis le répertoire du projet :

```

1 # Avec Maven
2 ./mvnw spring-boot:run
3
4 # Ou avec le JAR
5 java -jar target/todoapi-0.0.1-SNAPSHOT.jar

```

Listing 25: Lancer Spring Boot

Le serveur démarre sur <http://localhost:8080>.

4.5.2 La page Whitelabel Error

Si vous accédez à <http://localhost:8080/> dans votre navigateur, vous verrez :

Whitelabel Error Page

This application has no explicit mapping for /error,
so you are seeing this as a fallback.

There was an unexpected error (type=Not Found, status=404).

C'est normal ! Cette page signifie que l'application fonctionne, mais qu'aucun endpoint n'est défini sur la route /. Notre API expose des endpoints sur /api/..., pas sur la racine.

4.5.3 Tester les endpoints de l'API

Pour tester votre API, vous devez accéder aux URLs correctes :

- <http://localhost:8080/api/hello> → Retourne "Bonjour depuis Spring Boot!"
- <http://localhost:8080/api/todos> → Retourne la liste des todos (vide au début)

Avec le navigateur : Les requêtes GET peuvent être testées directement dans le navigateur en entrant l'URL.

Avec curl:

```

1 # GET - Recuperer tous les todos
2 curl http://localhost:8080/api/todos
3
4 # GET - Message de bienvenue
5 curl http://localhost:8080/api/hello
6
7 # POST - Creer un todo
8 curl -X POST http://localhost:8080/api/todos \
9     -H "Content-Type: application/json" \

```

```

10      -d '{"titre": "Apprendre Spring Boot", "description":'
           "Chapitre 1"}'

11 # GET - Recuperer le todo cree
12 curl http://localhost:8080/api/todos/1

14
15 # PUT - Modifier un todo
16 curl -X PUT http://localhost:8080/api/todos/1 \
17   -H "Content-Type: application/json" \
18   -d '{"titre": "Apprendre Spring Boot", "fait": true}'

19
20 # DELETE - Supprimer un todo
21 curl -X DELETE http://localhost:8080/api/todos/1

```

Listing 26: Tester l'API avec curl

Avec Postman : Postman est un outil graphique très pratique pour tester les APIs REST. Il permet de construire facilement des requêtes HTTP et de visualiser les réponses.

Astuce : Pour avoir une page d'accueil, vous pouvez ajouter un endpoint sur la racine :

```

1 @GetMapping("/")
2 public String home() {
3     return "Bienvenue sur l'API Todo!";
4 }

```

4.5.4 Visualiser les données avec IntelliJ Database

Pour voir les données dans votre base H2 (avec la configuration AUTO_SERVER=TRUE) :

1. Lancez d'abord l'application Spring Boot
2. View → Tool Windows → Database
3. Cliquer sur + → Data Source → H2
4. Configurer la connexion :
 - Connection type : Remote
 - URL : jdbc:h2:file:./data/tododb;AUTO_SERVER=TRUE
 - User : sa
 - Password : (laisser vide)
5. Cliquer sur Test Connection
6. Si le test réussit, cliquer sur OK

Erreur "Database may be already in use" ?

Cette erreur signifie que vous n'avez pas AUTO_SERVER=TRUE dans votre URL. Vérifiez que votre `application.properties` contient bien :

```
spring.datasource.url=jdbc:h2:file:./data/tododb;AUTO_SERVER=TRUE
```

Et utilisez exactement la même URL dans IntelliJ.

Vous pouvez maintenant :

- Voir la structure des tables dans l'arborescence
- Double-cliquer sur une table pour voir son contenu
- Exécuter des requêtes SQL avec la console intégrée

```
1 -- Voir tous les todos
2 SELECT * FROM TODOS;
3
4 -- Compter les todos terminés
5 SELECT COUNT(*) FROM TODOS WHERE FAIT = TRUE;
```

4.6 Configuration de la base de données H2

H2 est une base de données légère, parfaite pour le développement.

4.6.1 Configuration recommandée (mode fichier avec connexions multiples)

Cette configuration permet d'accéder à la base depuis Spring Boot ET depuis IntelliJ en même temps :

```
1 # H2 en mode fichier avec AUTO_SERVER (connexions multiples)
2 spring.datasource.url=jdbc:h2:file:./data/tododb;AUTO_SERVER=TRUE
3 spring.datasource.driverClassName=org.h2.Driver
4 spring.datasource.username=sa
5 spring.datasource.password=
6
7 # Configuration JPA
8 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
9 spring.jpa.hibernate.ddl-auto=update
10 spring.jpa.show-sql=true
```

Listing 27: `application.properties`

Important : Le paramètre AUTO_SERVER=TRUE est essentiel ! Sans lui, H2 verrouille le fichier et une seule application peut y accéder à la fois. Avec ce paramètre, H2 démarre automatiquement un serveur TCP en arrière-plan.

4.6.2 Connexion depuis IntelliJ IDEA

Une fois l'application Spring Boot démarrée :

1. Ouvrir l'onglet **Database** (View → Tool Windows → Database)
2. Cliquer sur + → Data Source → **H2**
3. Configurer :
 - Connection type : **Remote**
 - URL : `jdbc:h2:file:./data/tododb;AUTO_SERVER=TRUE`
 - User : `sa`
 - Password : (laisser vide)
4. Cliquer sur **Test Connection** puis **OK**

Astuce : L'application Spring Boot doit être lancée **avant** de tester la connexion dans IntelliJ, car c'est elle qui démarre le serveur H2.

4.6.3 Alternative : Mode mémoire simple (sans accès externe)

Si vous n'avez pas besoin d'accéder à la base depuis IntelliJ, utilisez simplement :

```

1 # H2 en mémoire (données perdues à chaque redémarrage)
2 spring.datasource.url=jdbc:h2:mem:tododb
3 spring.datasource.driverClassName=org.h2.Driver
4 spring.datasource.username=sa
5 spring.datasource.password=
6
7 # Configuration JPA
8 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
9 spring.jpa.hibernate.ddl-auto=create-drop
10 spring.jpa.show-sql=true

```

Listing 28: application.properties - Mode mémoire simple

Dans ce cas, vous pouvez voir les requêtes SQL dans la console Spring Boot grâce à `show-sql=true`.

5 Exercices pratiques

5.1 Exercice 1 : Premier composant Vue.js

Objectif : Créer un composant compteur avec les fonctionnalités suivantes :

1. Afficher un compteur initialisé à 0
2. Un bouton pour incrémenter le compteur

3. Un bouton pour décrémenter le compteur
4. Un bouton pour réinitialiser le compteur à 0
5. Afficher "Compteur positif" ou "Compteur négatif" selon la valeur (utiliser `computed`)

Indications :

```

1 <script setup lang="ts">
2 import { ref, computed } from 'vue'
3
4 const compteur = ref<number>(0)
5
6 // A compléter : fonctions et computed
7 </script>
```

Listing 29: Structure de base

5.2 Exercice 2 : Liste de tâches statique

Objectif : Créer une liste de tâches simple avec :

1. Une interface TypeScript `Tache` avec : id, titre, fait
2. Un tableau réactif de tâches avec `reactive`
3. Affichage de chaque tâche avec `v-for`
4. Une checkbox pour marquer une tâche comme terminée
5. Style différent pour les tâches terminées (texte barré)
6. Compteur de tâches restantes (propriété calculée)

Indications :

```

1 <script setup lang="ts">
2 import { reactive, computed } from 'vue'
3
4 interface Tache {
5   id: number
6   titre: string
7   fait: boolean
8 }
9
10 const taches = reactive<Tache[]>([
11   // Ajouter des taches de test
12 ])
13
14 // A compléter : computed et fonctions
15 </script>
```

Listing 30: Structure de base

5.3 Exercice 3 : Composants parent-enfant

Objectif : Séparer la liste de tâches en deux composants :

1. `TodoList.vue` : Composant parent qui gère la liste
2. `TodoItem.vue` : Composant enfant pour afficher une tâche
3. Utiliser `defineProps` pour passer les données au composant enfant
4. Utiliser `defineEmits` pour notifier le parent des changements

5.4 Exercice 4 : API Spring Boot

Objectif : Créer une API REST pour gérer des produits :

1. Entité `Product` avec : id, nom, prix, quantité
2. Repository `ProductRepository`
3. Contrôleur avec les endpoints CRUD :
 - `GET /api/products` : Liste de tous les produits
 - `GET /api/products/{id}` : Un produit par ID
 - `POST /api/products` : Créer un produit
 - `PUT /api/products/{id}` : Modifier un produit
 - `DELETE /api/products/{id}` : Supprimer un produit
4. Tester avec Postman ou curl

6 Résumé du chapitre

Dans ce premier chapitre, nous avons posé les bases du développement Full Stack :

- **Architecture Full Stack** : Compréhension des trois couches (frontend, backend, base de données) et de leur communication
- **API REST** : Les principes REST, les méthodes HTTP (GET, POST, PUT, DELETE), et le format JSON
- **Vue.js avec Composition API et TypeScript** :
 - Création de projet avec Vite et le template `vue-ts`
 - Structure des Single File Components (`.vue`)
 - Syntaxe `<script setup lang="ts">`
 - Données réactives avec `ref()` et `reactive()`
 - Propriétés calculées avec `computed()`

- Props typées avec `defineProps<T>()`
- Événements typés avec `defineEmits<T>()`
- **Spring Boot :**
 - Structure d'un projet
 - Entités JPA
 - Repositories Spring Data
 - Controllers REST

Concept	Syntaxe Composition API + TypeScript
Données réactives (primitive)	<code>const x = ref<Type>(valeur)</code>
Données réactives (objet)	<code>const x = reactive<Type>({...})</code>
Propriété calculée	<code>const x = computed<Type>(() => ...)</code>
Props	<code>defineProps<Interface>()</code>
Événements	<code>defineEmits<{(e: 'nom'): void}>()</code>

Table 3: Récapitulatif de la syntaxe Composition API avec TypeScript

Dans le prochain chapitre, nous verrons comment faire communiquer Vue.js et Spring Boot, en utilisant Axios pour les requêtes HTTP et Vue Router pour la navigation.

7 Ressources complémentaires

- Documentation Vue.js : <https://vuejs.org/guide/>
- Documentation Spring Boot : <https://spring.io/projects/spring-boot>
- Spring Initializr : <https://start.spring.io>
- Vite : <https://vitejs.dev>
- MDN Web Docs (HTTP) : <https://developer.mozilla.org/fr/docs/Web/HTTP>