STATE MANAGEMENT ANGULAR 19

Dr: khalil lakhdhar

ELITE TECH CONSULTING khalillakhdhar@gmail.com

Contents

Angular 19 – ToDo Store avec mockapi.io et Signals

Objectif

Créer un TodoStore connecté à mockapi.io avec Angular 19 en utilisant les signaux, computed et un store réactif moderne.

Préparation

1. Crée un projet sur https://mockapi.io

2. Ajoute une ressource 'todos' avec les champs : title (string) et completed (boolean)

3. Note l'URL de l'API, par exemple : https://<your_project>.mockapi.io/todos

1. todo.model.ts

```
export interface Todo {
  id: string;
  title: string;
  completed: boolean;
}
```

2. todo.store.ts

```
import { Injectable, signal, computed } from '@angular/core';
import { Todo } from './todo.model';
import { HttpClient } from '@angular/common/http';

@Injectable({ providedIn: 'root' })
export class TodoStore {
  private apiUrl = 'https://<your_project>.mockapi.io/todos';

  private _todos = signal<Todo[]>([]);
  todos = this._todos;
  completedCount = computed(() => this._todos().filter(t => t.completed).length);

  constructor(private http: HttpClient) {
```

```
  this.loadTodos();
 }

 loadTodos() {
  this.http.get<Todo[]>(this.apiUrl).subscribe(todos => this._todos.set(todos));
 }

 add(title: string) {
  const newTodo = { title, completed: false };
  this.http.post<Todo>(this.apiUrl, newTodo).subscribe(todo => {
   this._todos.update(todos => [...todos, todo]);
  });
 }

 toggle(todo: Todo) {
  const updated = { ...todo, completed: !todo.completed };
  this.http.put<Todo>(`${this.apiUrl}/${todo.id}`, updated).subscribe(t => {
   this._todos.update(todos =>
    todos.map(td => (td.id === t.id ? t : td))
   );
  });
 }

 remove(id: string) {
  this.http.delete(`${this.apiUrl}/${id}`).subscribe(() => {
   this._todos.update(todos => todos.filter(todo => todo.id !== id));
  });
 }
}
```

3. todo.component.ts

```
import { Component } from '@angular/core';
import { TodoStore } from './todo.store';

@Component({
 selector: 'app-todo',
 templateUrl: './todo.component.html',
})
export class TodoComponent {
 newTitle = '';

 constructor(public store: TodoStore) {}

 add() {
  if (this.newTitle.trim()) {
   this.store.add(this.newTitle.trim());
   this.newTitle = '';
  }
 }
}
```

4. todo.component.html

```
<h2>📝 Ma ToDo List (mockapi.io)</h2>

<input [(ngModel)]="newTitle" placeholder="Nouvelle tâche" />
<button (click)="add()">Ajouter</button>

<ul>
 <li *ngFor="let todo of store.todos()">
  <input
   type="checkbox"
   [checked]="todo.completed"
   (change)="store.toggle(todo)"
  />
  {{ todo.title }}
  <button (click)="store.remove(todo.id)">❌</button>
 </li>
</ul>
<p>Total : {{ store.todos().length }} | Terminées : {{ store.completedCount() }}</p>
```

Exemple 2 AuthStore Global avec Signals

Objectif

Créer un AuthStore global en Angular 19 avec Signals, pour gérer l'authentification et le profil utilisateur, sans localStorage.

1. Modèle – auth.model.ts

```
export interface User {
  id: string;
  username: string;
  email: string;
}

export interface AuthResponse {
  token: string;
  user: User;
}
```

2. Store – auth.store.ts

```
import { Injectable, signal, computed, effect } from '@angular/core';
import { User, AuthResponse } from './auth.model';
import { HttpClient } from '@angular/common/http';
import { toObservable } from '@angular/core/rxjs-interop';

@Injectable({ providedIn: 'root' })
export class AuthStore {
  private _token = signal<string | null>(null);
  private _user = signal<User | null>(null);
  private _loading = signal(false);
  private _error = signal<string | null>(null);

  user = this._user;
  token = this._token;
  error = this._error;
```

```
  loading = this._loading;

  isAuthenticated = computed(() => this._token() !== null);

  constructor(private http: HttpClient) {
    effect(() => {
      console.log("Authentifié ?", this.isAuthenticated());
      if (this._user()) {
        console.log("Utilisateur connecté :", this._user());
      }
    });
  }

  login(username: string, password: string) {
    this._loading.set(true);
    this._error.set(null);

    this.http
      .post<AuthResponse>('https://mockapi.io/api/login', { username, password })
      .subscribe({
        next: (res) => {
          this._token.set(res.token);
          this._user.set(res.user);
        },
        error: () => {
          this._error.set('Identifiants invalides');
        },
        complete: () => this._loading.set(false),
      });
  }

  logout() {
    this._token.set(null);
    this._user.set(null);
  }

  user$ = toObservable(this.user);
  isAuthenticated$ = toObservable(this.isAuthenticated);
}
```

3. Composant – login.component.ts

```
import { Component } from '@angular/core';
import { AuthStore } from './auth.store';

@Component({
 selector: 'app-login',
 templateUrl: './login.component.html',
})
export class LoginComponent {
 username = '';
 password = '';

 constructor(public auth: AuthStore) {}

 login() {
  this.auth.login(this.username, this.password);
 }
}
```

4. Template HTML – login.component.html

```
<div *ngIf="!auth.isAuthenticated(); else connected">
 <input [(ngModel)]="username" placeholder="Nom d'utilisateur" />
 <input [(ngModel)]="password" placeholder="Mot de passe" type="password" />
 <button (click)="login()" [disabled]="auth.loading()">Connexion</button>
 <p *ngIf="auth.error()">{{ auth.error() }}</p>
</div>

<ng-template #connected>
 <p>Bienvenue, {{ auth.user()?.username }} !</p>
 <button (click)="auth.logout()">Déconnexion</button>
</ng-template>
```

Avantages des signaux et computed

- signal() : État réactif sans abonnement
- computed() : Valeur dérivée automatiquement mise à jour
- store : Centralise logique + accès API
- Pas de subscribe() manuel dans le composant
- Code plus simple et plus maintenable

Ancienne approche vs Nouvelle approche

Ancienne pratique (RxJS)	Nouvelle avec Signals
BehaviorSubject<Todo[]>	signal<Todo[]>
todos$.pipe(map(...))	computed(() => ...)
subscribe() dans le composant	signal() dans le template
ngOnDestroy() pour unsubscribe	Plus nécessaire

Conclusion

Cette approche moderne te permet de créer une application réactive avec un code plus clair et plus simple. Pour un vrai temps réel, il faudra utiliser Firebase, Supabase ou WebSocket.

Exercice Gestion d’un compteur de recettes

Objectif

Créer une application Angular permettant de gérer un compteur de portions pour une recette, et d’ajuster automatiquement les quantités d’ingrédients via computed().

Énoncé

Vous développez une interface pour une application de recettes. Elle affiche une recette simple (par exemple : Crêpes) avec des ingrédients et leurs quantités pour 2 personnes.
Le but est de :
- Pouvoir augmenter ou diminuer le nombre de portions via des boutons + / -
- Adapter automatiquement les quantités d’ingrédients grâce à computed()

Données de base (exemple de 2 personnes)

```
{
 title: 'Crêpes',
 basePortions: 2,
 ingredients: [
  { name: 'Farine', unit: 'g', quantity: 200 },
  { name: 'Lait', unit: 'ml', quantity: 400 },
  { name: 'Œufs', unit: '', quantity: 2 }
 ]
}
```

À faire

1. 1. Créez un RecipeStore :

- - Signal `portions` (ex: signal(2))
- - Méthodes `increasePortion()` et `decreasePortion()`
- - `computed()` qui adapte les quantités en fonction des portions

2. 2. Créez un composant :

- - Affiche la recette (nom, portions)
- - Boutons `+ / -` pour changer le nombre de portions

- - Liste des ingrédients recalculée dynamiquement

Exemple attendu à l'écran

Recette : Crêpes
Portions : 4 [+] [-]

Ingrédients :
- Farine : 400 g
- Lait : 800 ml
- Œufs : 4

Compétences pratiquées

- Utilisation de signal() pour gérer l'état local

- Utilisation de computed() pour recalculer les valeurs dynamiques

- Interaction entre l'UI et le store

- Séparation claire entre logique métier (store) et affichage (composant)