

@ngrx/señales



NgRx Signals es una biblioteca independiente que proporciona una solución de gestión de estado reactiva y un conjunto de utilidades para Angular Signals.

Principios Clave

- **Simple e Intuitivo:** Diseñado teniendo en cuenta la facilidad de uso, NgRx Signals proporciona una API sencilla e intuitiva para que los desarrolladores trabajen de manera eficiente con Angular Signals.
- **Ligero y Performant:** Mantenga el tamaño de su aplicación óptimo con una biblioteca liviana que agrega una sobrecarga mínima a sus proyectos y funciona de manera eficiente.
- **Declarativo:** NgRx Signals se basa en el concepto de programación declarativa, asegurando un código limpio y conciso.
- **Modular, Extensible y Escalable:** La modularidad y la extensibilidad son los principios rectores de esta biblioteca. NgRx Signals permite la creación de bloques de construcción independientes que se pueden combinar fácilmente para implementaciones flexibles y escalables.
- **Opinión, pero Flexible:** Logre un equilibrio entre flexibilidad y opinión, ofreciendo personalización donde sea necesario mientras proporciona convenciones reflexivas para una experiencia de desarrollo sin problemas.
- **Tipo-seguro:** NgRx Signals está diseñado con un fuerte enfoque en la seguridad de tipos, asegurando la prevención de errores y mal uso en el momento de la compilación.

Instalación

Las instrucciones detalladas de instalación se pueden encontrar en el [Instalación](#) página.

Características Principales

- **SignalStore:** Una solución de gestión de estado con todas las funciones que proporciona soporte nativo para señales angulares y ofrece una forma sólida de administrar el estado de la aplicación.
- **Estado de Señalización:** Una utilidad ligera para administrar el estado basado en señales en componentes y servicios angulares de manera concisa y minimalista.
- **Integración RxJS:** Un plugin para la integración opt-in con RxJS, lo que permite un manejo más fácil de los efectos secundarios asíncronos.
- **Gestión de Entidades:** Un plugin para manipular y consultar colecciones de entidades de una manera simple y performant.

Instalación



Instalación con `ng add`

Puedes instalar el `@ngrx/signals` a su proyecto con lo siguiente `ng add` comando (detalles aquí):

```
ng add @ngrx/signals@latest
```

Este comando automatizará los siguientes pasos:

1. Actualizar `package.json` > `dependencies` con `@ngrx/signals`.
2. Ejecute el administrador de paquetes para instalar la dependencia agregada.

Instalación con `npm`

Para más información sobre el uso `npm` echa un vistazo a los documentos aquí.

```
npm install @ngrx/signals --save
```

Instalación con `yarn`

Para más información sobre el uso `yarn` echa un vistazo a los documentos aquí.

```
yarn add @ngrx/signals
```


Estado de Señalización



SignalState es una utilidad ligera diseñada para administrar el estado basado en señales de manera concisa y minimalista. Es adecuado para administrar estados de tamaño modesto y se puede usar directamente en componentes, servicios o funciones independientes.

Creación de un Estado de Señal

SignalState se instanció utilizando el `signalState` función, que acepta un estado inicial como argumento de entrada.

```
import { signalState } from '@ngrx/signals';
import { User } from './user.model';

type UserState = { user: User; isAdmin: boolean };

const userState = signalState<UserState>({
  user: { firstName: 'Eric', lastName: 'Clapton' },
  isAdmin: false,
});
```

El tipo de estado debe ser un registro/objeto literal. Añadir matrices o valores primitivos a las propiedades.

`signalState` devuelve una versión extendida de una señal que posee todas las capacidades de una señal de solo lectura.

```
import { computed, effect } from '@angular/core';

// 📌 Creating computed signals.
const userStateStr = computed(() => JSON.stringify(userState()));

// 📌 Performing side effects.
effect(() => console.log('userState', userState()));
```

Además, el `signalState` la función genera señales para cada propiedad de estado.

```
const user = userState.user; // type: DeepSignal<User>
const isAdmin = userState.isAdmin; // type: Signal<boolean>

console.log(user()); // logs: { firstName: 'Eric', lastName: 'Clapton' }
console.log(isAdmin()); // logs: false
```

Cuando una propiedad estatal mantiene un objeto como su valor, el `signalState` la función genera a `DeepSignal`. Se puede usar como una señal de solo lectura regular, pero también contiene señales para cada propiedad del objeto al que se refiere.

```
const firstName = user.firstName; // type: Signal<string>
const lastName = user.lastName; // type: Signal<string>

console.log(firstName()); // logs: 'Eric'
console.log(lastName()); // logs: 'Clapton'
```

Para un rendimiento mejorado, las señales profundamente anidadas se generan perezosamente e inicializan solo en el primer acceso.

Actualización del Estado

El `patchState` la función proporciona una forma segura de tipo para realizar actualizaciones en partes de estado. Toma una instancia de `SignalState` o `SignalStore` como primer argumento, seguido de una secuencia de estados parciales o actualizadores de estados parciales como argumentos adicionales.

```
import { patchState } from '@ngrx/signals';

// 📌 Providing a partial state object.
patchState(userState, { isAdmin: true });

// 📌 Providing a partial state updater.
patchState(userState, (state) => ({
  user: { ...state.user, firstName: 'Jimi' },
}));

// 📌 Providing a sequence of partial state objects and/or
updaters.
patchState(
  userState,
  { isAdmin: false },
  (state) => ({ user: { ...state.user, lastName: 'Hendrix' } })
);
```

Los actualizadores pasaron a la `patchState` la función debe realizar actualizaciones de estado de una manera inmutable. Si se produce un cambio mutable en el objeto de estado, se generará un error en el modo de desarrollo.

Actualizadores de Estado Personalizados

En lugar de proporcionar estados parciales o actualizadores directamente a la `patchState` función, es posible crear actualizadores de estado personalizados.

```
import { PartialStateUpdater } from '@ngrx/signals';

function setFirstName(firstName: string): PartialStateUpdater<{
  user: User }> {
  return (state) => ({ user: { ...state.user, firstName } });
}

const setAdmin = () => ({ isAdmin: true });
```

Los actualizadores de estado personalizados son fáciles de probar y se pueden reutilizar en diferentes partes de la aplicación.

```
// Before:  
patchState(userState, (state) => ({  
  user: { ...state.user, firstName: 'Stevie' },  
  isAdmin: true,  
}));  
  
// After:  
patchState(userState, setFirstName('Stevie'), setAdmin());
```


Uso

Ejemplo 1: Estado de señal en un componente

contra.component.ts

```
1. import { ChangeDetectionStrategy, Component } from
   '@angular/core';
2. import { signalState, patchState } from '@ngrx/signals';
3.
4. @Component({
5.   selector: 'ngrx-counter',
6.   template: `
7.     <p>Count: {{ state.count() }}</p>
8.
9.     <button (click)="increment()">Increment</button>
10.    <button (click)="decrement()">Decrement</button>
11.    <button (click)="reset()">Reset</button>
12.  `,
13.  changeDetection: ChangeDetectionStrategy.OnPush,
14. })
15. export class CounterComponent {
16.   readonly state = signalState({ count: 0 });
17.
18.   increment(): void {
19.     patchState(this.state, (state) => ({ count: state.count
20.       + 1 }));
21.   }
22.
23.   decrement(): void {
24.     patchState(this.state, (state) => ({ count: state.count
25.       - 1 }));
26.   }
27.
28.   reset(): void {
29.     patchState(this.state, { count: 0 });
30.   }
31. }
```

Ejemplo 2: SignalState en un servicio

```
1. import { inject, Injectable } from '@angular/core';
2. import { exhaustMap, pipe, tap } from 'rxjs';
3. import { signalState, patchState } from '@ngrx/signals';
4. import { rxMethod } from '@ngrx/signals/rxjs-interop';
5. import { tapResponse } from '@ngrx/operators';
6. import { BooksService } from '../books.service';
7. import { Book } from '../book.model';
8.
9. type BooksState = { books: Book[]; isLoading: boolean };
10.
11. const initialState: BooksState = {
12.   books: [],
13.   isLoading: false,
14. };
15.
16. @Injectable()
17. export class BooksStore {
18.   readonly #booksService = inject(BooksService);
19.   readonly #state = signalState(initialState);
20.
21.   readonly books = this.#state.books;
22.   readonly isLoading = this.#state.isLoading;
23.
24.   readonly loadBooks = rxMethod<void>(
25.     pipe(
26.       tap(() => patchState(this.#state, { isLoading: true
27.         })),
28.       exhaustMap(() => {
29.         return this.#booksService.getAll().pipe(
30.           tapResponse({
31.             next: (books) => patchState(this.#state, {
32.               books }),
33.             error: console.error,
34.             finalize: () => patchState(this.#state, {
35.               isLoading: false })),
36.           })
37.         );
38.       })
39.     );
```

SignalStore



NgRx SignalStore es una solución de administración de estado con todas las funciones que ofrece una forma sólida de administrar el estado de la aplicación. Con su soporte nativo para Señales, proporciona la capacidad de definir tiendas de una manera clara y declarativa. La simplicidad y flexibilidad de SignalStore, junto con su diseño obstinado y extensible, lo establecen como una solución versátil para una gestión estatal efectiva en Angular.

Creando una Tienda

Se crea una SignalStore utilizando el `signalStore` función. Esta función acepta una secuencia de características de la tienda. A través de la combinación de características de la tienda, la SignalStore gana estado, propiedades y métodos, lo que permite una implementación de tienda flexible y extensible. Basado en las características utilizadas, el `signalStore` la función devuelve un servicio inyectable que se puede proporcionar e inyectar donde sea necesario.

El `withState` la función se usa para agregar cortes de estado a SignalStore. Esta característica acepta el estado inicial como argumento de entrada. Como con `signalState`, el tipo del estado debe ser un registro/objeto literal.

libros.store.ts

```
import { signalStore, withState } from '@ngrx/signals';
import { Book } from '../book.model';

type BooksState = {
  books: Book[];
  isLoading: boolean;
  filter: { query: string; order: 'asc' | 'desc' };
};

const initialState: BooksState = {
  books: [],
  isLoading: false,
  filter: { query: '', order: 'asc' },
};

export const BooksStore = signalStore(
  withState(initialState)
);
```

Para cada segmento de estado, se crea automáticamente una señal correspondiente. Lo mismo se aplica a las propiedades de estado anidado, con todas las señales profundamente anidadas generadas perezosamente a pedido.

El `BooksStore` la instancia contendrá las siguientes propiedades:

- `books: Signal<Book[]>`
- `isLoading: Signal<boolean>`
- `filter: DeepSignal<{ query: string; order: 'asc' | 'desc' }>`
- `filter.query: Signal<string>`
- `filter.order: Signal<'asc' | 'desc'>`

El `withState` la característica también tiene una firma que toma la fábrica de estado inicial como un argumento de entrada. La fábrica se ejecuta dentro del contexto de inyección, lo que permite obtener el estado inicial de un token de servicio o inyección.

```
const BOOKS_STATE = new InjectionToken<BooksState>('BooksState', {
  factory: () => initialState,
});

const BooksStore = signalStore(
  withState(() => inject(BOOKS_STATE))
);
```

Proporcionar e Inyectar la Tienda

SignalStore se puede proporcionar a nivel local y global. De forma predeterminada, una SignalStore no está registrada con ningún inyector y debe incluirse en una matriz de proveedores a nivel de componente, ruta o raíz antes de la inyección.

libros.component.ts

```
import { Component, inject } from '@angular/core';
import { BooksStore } from '../books.store';

@Component({
  /* ... */
  // 📌 Providing `BooksStore` at the component level.
  providers: [BooksStore],
})
export class BooksComponent {
  readonly store = inject(BooksStore);
}
```

Cuando se proporciona a nivel de componente, la tienda está vinculada al ciclo de vida del componente, lo que la hace útil para administrar el estado local/componente. Alternativamente, una SignalStore se puede registrar globalmente configurando el `providedIn` propiedad a `root` al definir la tienda.

libros.store.ts

```
import { signalStore, withState } from '@ngrx/signals';
import { Book } from '../book.model';

type BooksState = { /* ... */ };

const initialState: BooksState = { /* ... */ };

export const BooksStore = signalStore(
  // 📌 Providing `BooksStore` at the root level.
  { providedIn: 'root' },
  withState(initialState)
);
```

Cuando se proporciona a nivel mundial, la tienda está registrada con el inyector raíz y se hace accesible en cualquier lugar de la aplicación. Esto es beneficioso para administrar el estado global, ya que garantiza una única instancia compartida de la tienda en toda la aplicación.

Estado de Lectura

Las señales generadas para cortes de estado se pueden utilizar para acceder a valores de estado, como se demuestra a continuación.

libros.component.ts

```

import { ChangeDetectionStrategy, Component, inject } from
 '@angular/core';
import { JsonPipe } from '@angular/common';
import { BooksStore } from '../books.store';

@Component({
  imports: [JsonPipe],
  template: `
    <p>Books: {{ store.books() | json }}</p>
    <p>Loading: {{ store.isLoading() }}</p>

    <!-- 📌 The `DeepSignal` value can be read in the same way as
    `Signal`. -->
    <p>Pagination: {{ store.filter() | json }}</p>

    <!-- 📌 Nested signals are created as `DeepSignal` properties.
    -->
    <p>Query: {{ store.filter.query() }}</p>
    <p>Order: {{ store.filter.order() }}</p>
  `,
  providers: [BooksStore],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class BooksComponent {
  readonly store = inject(BooksStore);
}

```

Definición de Propiedades de la Tienda

Las señales calculadas se pueden agregar a la tienda usando el `withComputed` característica. Esta característica acepta una función de fábrica como argumento de entrada, que se ejecuta dentro del contexto de inyección. La fábrica debe devolver un diccionario de señales calculadas, utilizando señales de estado previamente definidas y propiedades que son accesibles a través de su argumento de entrada.

libros.store.ts

```
import { computed } from '@angular/core';
import { signalStore, withComputed, withState } from
  '@ngrx/signals';
import { Book } from '../book.model';

type BooksState = { /* ... */ };

const initialState: BooksState = { /* ... */ };

export const BooksStore = signalStore(
  withState(initialState),
  // 📌 Accessing previously defined state signals and properties.
  withComputed(({ books, filter }) => ({
    booksCount: computed(() => books().length),
    sortedBooks: computed(() => {
      const direction = filter.order() === 'asc' ? 1 : -1;

      return books().toSorted((a, b) =>
        direction * a.title.localeCompare(b.title)
      );
    }),
  })),
);
```

El `withProps` la función se puede usar para agregar propiedades estáticas, observables, dependencias y cualquier otra propiedad personalizada a una SignalStore. Para más detalles, vea el Propiedades de la Tienda Personalizada guía.

Definición de Métodos de Tienda

Los métodos se pueden agregar a la tienda usando el `withMethods` característica. Esta característica toma una función de fábrica como argumento de entrada y devuelve un diccionario de métodos. Similar a `withComputed`, el `withMethods` la fábrica también se ejecuta dentro del contexto de inyección. La instancia de la tienda, incluidas las señales de estado, propiedades y métodos definidos previamente, es accesible a través de la entrada de fábrica.

libros.store.ts

```
import { computed } from '@angular/core';
import {
  patchState,
  signalStore,
  withComputed,
  withMethods,
  withState,
} from '@ngrx/signals';
import { Book } from '../book.model';

type BooksState = { /* ... */ };

const initialState: BooksState = { /* ... */ };

export const BooksStore = signalStore(
  withState(initialState),
  withComputed(/* ... */),
  // 📌 Accessing a store instance with previously defined state
  // signals,
  // properties, and methods.
  withMethods((store) => ({
    updateQuery(query: string): void {
      // 📌 Updating state using the `patchState` function.
      patchState(store, (state) => ({ filter: { ...state.filter,
        query } }));
    },
    updateOrder(order: 'asc' | 'desc'): void {
      patchState(store, (state) => ({ filter: { ...state.filter,
        order } }));
    },
  })))
);
```

El estado de SignalStore se actualiza utilizando el `patchState` función. Para más detalles sobre el `patchState` función, consulte el Actualización del Estado guía.

De forma predeterminada, el estado de `SignalStore` está protegido de modificaciones externas, lo que garantiza un flujo de datos consistente y predecible. Este es el enfoque recomendado. Sin embargo, las actualizaciones externas al estado se pueden habilitar configurando el `protectedState` opción a `false` al crear una `SignalStore`.

```
export const BooksStore = signalStore(  
  { protectedState: false }, // 📌  
  withState(initialState)  
);  
  
@Component({ /* ... */ })  
export class BooksComponent {  
  readonly store = inject(BooksStore);  
  
  addBook(book: Book): void {  
    // ⚠️ The state of the `BooksStore` is unprotected from  
    external modifications.  
    patchState(this.store, ({ books }) => ({ books:  
      [...books, book] }));  
  }  
}
```

Además de los métodos para actualizar el estado, el `withMethods` la característica también se puede utilizar para crear métodos para realizar efectos secundarios. Los efectos secundarios asíncronos se pueden ejecutar utilizando API basadas en Promise, como se demuestra a continuación.

libros.store.ts

```
import { computed, inject } from '@angular/core';
import { patchState, signalStore, /* ... */ } from
  '@ngrx/signals';
import { Book } from './book.model';
import { BooksService } from './books.service';

type BooksState = { /* ... */ };

const initialState: BooksState = { /* ... */ };

export const BooksStore = signalStore(
  withState(initialState),
  withComputed(/* ... */),
  // 📌 `BooksService` can be injected within the `withMethods`
  factory.
  withMethods((store, booksService = inject(BooksService)) => ({
    /* ... */
    // 📌 Defining a method to load all books.
    async loadAll(): Promise<void> {
      patchState(store, { isLoading: true });

      const books = await booksService.getAll();
      patchState(store, { books, isLoading: false });
    },
  })))
);
```

Métodos Reactivos de Tienda

En escenarios más complejos, es aconsejable optar por RxJS para manejar los efectos secundarios asíncronos. Para crear un método SignalStore reactivo que aproveche las API de RxJS, utilice el `rxMethod` función de la `rxjs-interop` plugin.

libros.store.ts

```

import { computed, inject } from '@angular/core';
import { debounceTime, distinctUntilChanged, pipe, switchMap, tap
} from 'rxjs';
import { patchState, signalStore, /* ... */ } from
 '@ngrx/signals';
import { rxMethod } from '@ngrx/signals/rxjs-interop';
import { tapResponse } from '@ngrx/operators';
import { Book } from '../book.model';
import { BooksService } from '../books.service';

type BooksState = { /* ... */ };

const initialState: BooksState = { /* ... */ };

export const BooksStore = signalStore(
  withState(initialState),
  withComputed(/* ... */),
  withMethods((store, booksService = inject(BooksService)) => ({
    /* ... */
    // 📌 Defining a method to load books by query.
    loadByQuery: rxMethod<string>(
      pipe(
        debounceTime(300),
        distinctUntilChanged(),
        tap(() => patchState(store, { isLoading: true })),
        switchMap((query) => {
          return booksService.getByQuery(query).pipe(
            tapResponse({
              next: (books) => patchState(store, { books,
isLoading: false })),
              error: (err) => {
                patchState(store, { isLoading: false });
                console.error(err);
              },
            })
          )
        })
      )
    ),
  })))
);

```



Para aprender más sobre el `rxMethod` función, visita el [Integración RxJS](#) página.

Poniéndolo Todo Juntos

La final `BooksStore` la implementación con estado, señales calculadas y métodos de esta guía se muestra a continuación.

libros.store.ts

```
import { computed, inject } from '@angular/core';
import { debounceTime, distinctUntilChanged, pipe, switchMap, tap
} from 'rxjs';
import {
  patchState,
  signalStore,
  withComputed,
  withMethods,
  withState,
} from '@ngrx/signals';
import { rxMethod } from '@ngrx/signals/rxjs-interop';
import { tapResponse } from '@ngrx/operators';
import { Book } from '../book.model';
import { BooksService } from '../books.service';

type BooksState = {
  books: Book[];
  isLoading: boolean;
  filter: { query: string; order: 'asc' | 'desc' };
};

const initialState: BooksState = {
  books: [],
  isLoading: false,
  filter: { query: '', order: 'asc' },
};

export const BooksStore = signalStore(
  withState(initialState),
  withComputed(({ books, filter }) => ({
    booksCount: computed(() => books().length),
    sortedBooks: computed(() => {
      const direction = filter.order() === 'asc' ? 1 : -1;

      return books().toSorted((a, b) =>
        direction * a.title.localeCompare(b.title)
      );
    }),
  })),
  withMethods((store, booksService = inject(BooksService)) => ({
    updateQuery(query: string): void {
```

```

    patchState(store, (state) => ({ filter: { ...state.filter,
query } })));
  },
  updateOrder(order: 'asc' | 'desc'): void {
    patchState(store, (state) => ({ filter: { ...state.filter,
order } })));
  },
  loadByQuery: rxMethod<string>(
    pipe(
      debounceTime(300),
      distinctUntilChanged(),
      tap(() => patchState(store, { isLoading: true })),
      switchMap((query) => {
        return booksService.getByQuery(query).pipe(
          tapResponse({
            next: (books) => patchState(store, { books }),
            error: console.error,
            finalize: () => patchState(store, { isLoading:
false })),
          })
        );
      })
    );
  );
});

```

El `BooksStore` la instancia contendrá las siguientes propiedades y métodos:

- Señales de estado:
 - `books: Signal<Book[]>`
 - `isLoading: Signal<boolean>`
 - `filter: DeepSignal<{ query: string; order: 'asc' | 'desc' }>`
 - `filter.query: Signal<string>`
 - `filter.order: Signal<'asc' | 'desc'>`
- Señales calculadas:
 - `booksCount: Signal<number>`
 - `sortedBooks: Signal<Book[]>`
- Métodos:
 - `updateQuery(query: string): void`
 - `updateOrder(order: 'asc' | 'desc'): void`
 - `loadByQuery: RxMethod<string>`

El `BooksStore` la implementación se puede mejorar aún más utilizando el `entities` plugin y creación de características personalizadas de `SignalStore`. Para más detalles, consulte el [Gestión de Entidades y Características Personalizadas de la Tienda guías](#).

El `BooksComponent` puede usar el `BooksStore` para gestionar el estado, como se demuestra a continuación.

libros.component.ts


```
import { ChangeDetectionStrategy, Component, inject, OnInit }
from '@angular/core';
import { BooksFilterComponent } from '../books-filter.component';
import { BookListComponent } from '../book-list.component';
import { BooksStore } from '../books.store';

@Component({
  imports: [BooksFilterComponent, BookListComponent],
  template: `
    <h1>Books ({{ store.booksCount() }})</h1>

    <ngrx-books-filter
      [query]="store.filter.query()"
      [order]="store.filter.order()"
      (queryChange)="store.updateQuery($event)"
      (orderChange)="store.updateOrder($event)"
    />

    <ngrx-book-list
      [books]="store.sortedBooks()"
      [isLoading]="store.isLoading()"
    />
  `,
  providers: [BooksStore],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class BooksComponent implements OnInit {
  readonly store = inject(BooksStore);

  ngOnInit(): void {
    const query = this.store.filter.query;
    // 📌 Re-fetch books whenever the value of query signal
    changes.
    this.store.loadByQuery(query);
  }
}
```



Además de los ganchos del ciclo de vida de los componentes, SignalStore también ofrece la capacidad de definirlos a nivel de tienda. Obtenga más información sobre los ganchos del ciclo de vida de SignalStore [aquí](https://ngrx.io/guide/signals/signal-store).