

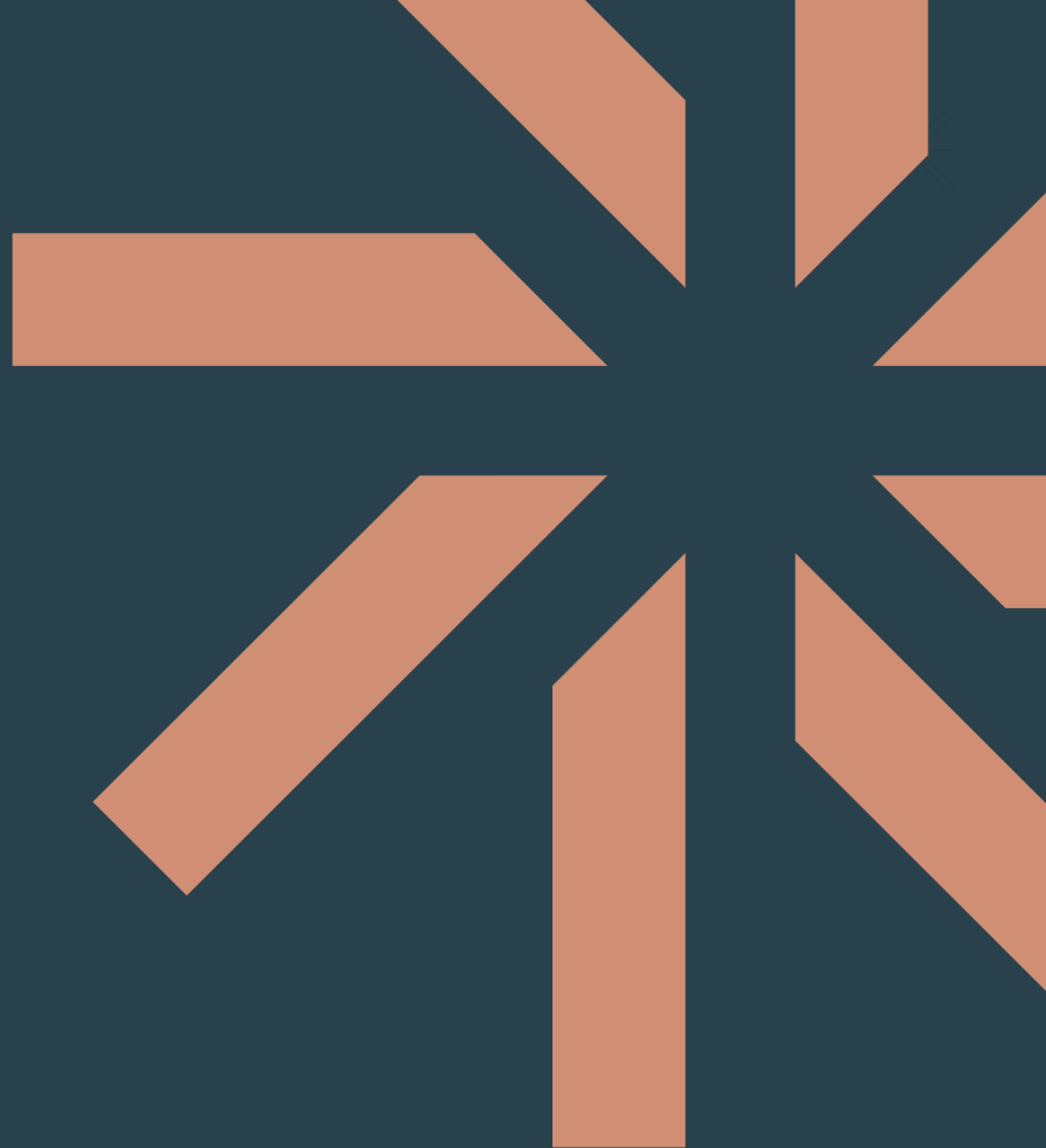
# Angular

Qinshift   
Academy

# NgRx SignalStore

# What is NgRx SignalStore?

NgRx SignalStore is a fully-featured state management solution that offers a robust way to manage application state. With its native support for Signals, it provides the ability to define stores in a clear and declarative manner. The simplicity and flexibility of SignalStore, coupled with its opinionated and extensible design, establish it as a versatile solution for effective state management in Angular.



# Creating a Store



A `SignalStore` is created using the `signalStore` function. This function accepts a sequence of store features. Through the combination of store features, the `SignalStore` gains state, computed signals, and methods, allowing for a flexible and extensible store implementation. Based on the utilized features, the `signalStore` function returns an injectable service that can be provided and injected where needed.

The `withState` feature is used to add state properties to the `SignalStore`. This feature accepts initial state as an input argument. As with `signalState`, the state's type must be a record/object literal.

# Creating a Store



```
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)  
);
```

# Creating a Store



For each state property, a corresponding signal is automatically created. The same applies to nested state properties, with all deeply nested signals being generated lazily on demand.

The BooksStore instance will contain the following properties:

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

# Creating a Store



The `withState` feature also has a signature that takes the initial state factory as an input argument. The factory is executed within the injection context, allowing initial state to be obtained from a service or injection token.

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

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

# Providing and Injecting the Storelink



SignalStore can be provided locally and globally. By default, a SignalStore is not registered with any injectors and must be included in a providers array at the component, route, or root level before injection.

books.component.ts

content\_copy

```
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);
```

```
}
```



# Providing and Injecting the Storelink



When provided at the component level, the store is tied to the component lifecycle, making it useful for managing local/component state. Alternatively, a SignalStore can be globally registered by setting the `providedIn` property to `root` when defining the store.

```
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)  
);
```

When provided globally, the store is registered with the root injector and becomes accessible anywhere in the application. This is beneficial for managing global state, as it ensures a single shared instance of the store across the entire application.

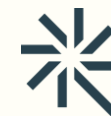
# Consuming State



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

```
@Component({
  standalone: true,
  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);
}
```

# Consuming State



The `@ngrx/signals` package also offers the `getState` function to get the current state value of the `SignalStore`. When used within the reactive context, state changes are automatically tracked.

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

@Component({ /* ... */ })
export class BooksComponent {
  readonly store = inject(BooksStore);

  constructor() {
    effect(() => {
      // 📌 The effect will be re-executed whenever the state changes.
      const state = getState(this.store);
      console.log('books state changed', state);
    });
  }
}
```

# Defining Computed Signals



Computed signals can be added to the store using the `withComputed` feature. This feature accepts a factory function as an input argument, which is executed within the injection context. The factory should return a dictionary of computed signals, utilizing previously defined state and computed signals that are accessible through its input argument.

```
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 and computed signals.
  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)
      );
    }),
  }))
);
```

# Defining Store Methods



Methods can be added to the store using the `withMethods` feature. This feature takes a factory function as an input argument and returns a dictionary of methods. Similar to `withComputed`, the `withMethods` factory is also executed within the injection context. The store instance, including previously defined state, computed signals, and methods, is accessible through the factory input.

# Defining Store Methods



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

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

```
export const BooksStore = signalStore(  
  withState(initialState),  
  withComputed(/* ... */),  
  // 📌 Accessing a store instance with previously defined state,  
  // computed signals, 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 } }));  
    },  
  }))  
);
```



# Questions?

Trainer Name

Trainer

trainer@mail.com

Assistant Name

Assistant

assistant@mail.com