NgRx

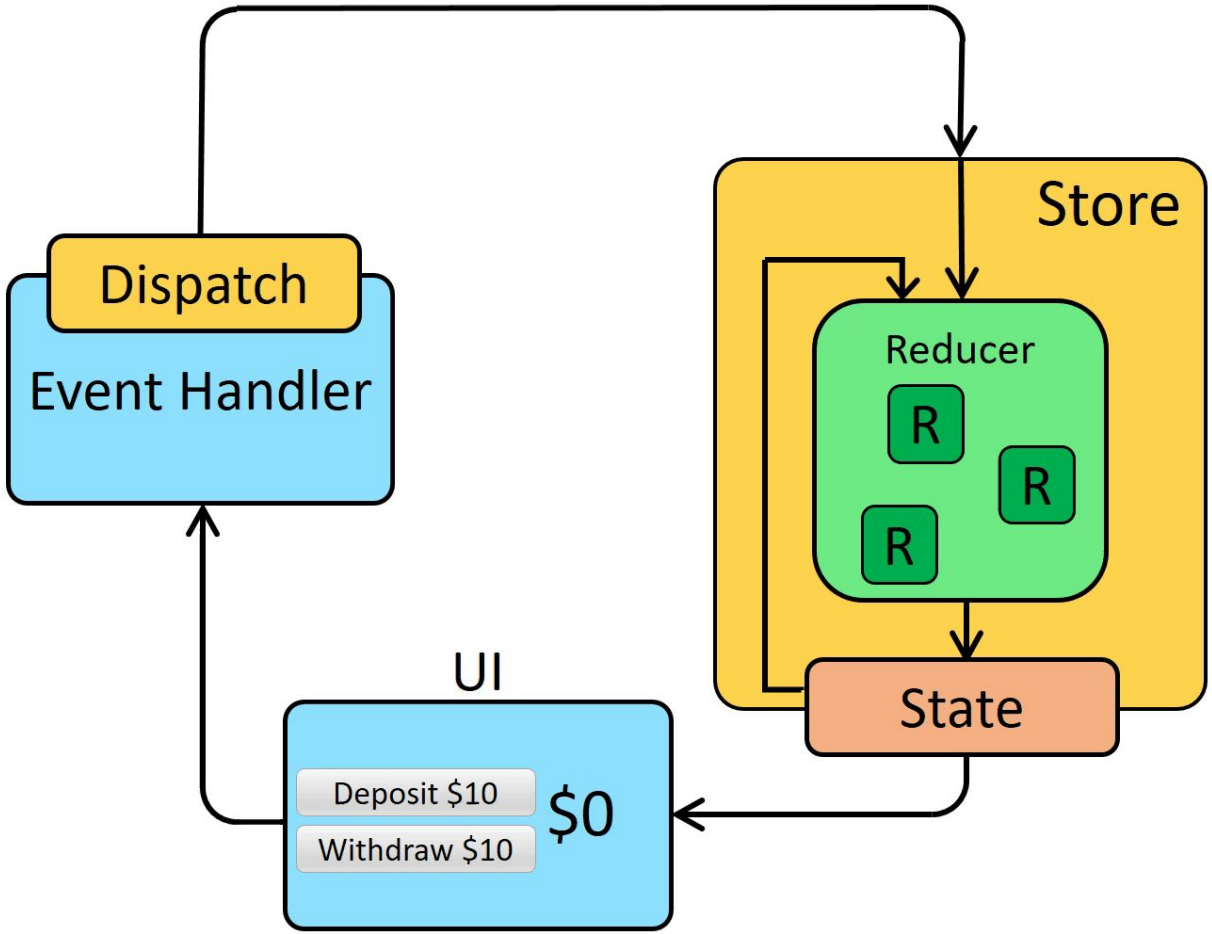
Introduction to NgRx

NgRx is a state management library for Angular applications, inspired by the **Redux** pattern. It helps manage the state of your application in a predictable and scalable way, especially in complex applications where multiple components need to share and update data. NgRx leverages **RxJS** (Reactive Extensions for JavaScript) to handle asynchronous data streams, making it a natural fit for Angular, which also uses RxJS extensively.

Key benefits of using NgRx include:

- **Centralized State Management**: All application state is stored in a single, immutable object.
- **Predictable State Changes**: State changes are handled through pure functions (reducers) in response to actions.
- **Improved Debugging**: Tools like NgRx DevTools allow for time-travel debugging and state inspection.
- **Performance Optimization**: Efficient change detection and memoization through selectors.

In this tutorial, we'll explore how to use NgRx with **Angular 19**, the latest version of Angular at the time of writing. We'll start with a simple counter example and then build a todo list application to demonstrate more advanced features like NgRx Entity and Effects.


Store
Reducer
R
R
R
State
Dispatch
Event Handler
UI
Deposit $10
Withdraw $10
$0

Understanding NgRx Core Concepts

Before diving into code, it's essential to understand the core concepts of NgRx:

Store

The **Store** is a centralized container that holds the entire state of your application. It is immutable, meaning the state cannot be changed directly. Instead, state changes are made by dispatching actions.

Actions

Actions are plain JavaScript objects that describe events in the application. Each action has a type property that indicates the type of action being performed. Actions can also carry additional data (payload).

Reducers

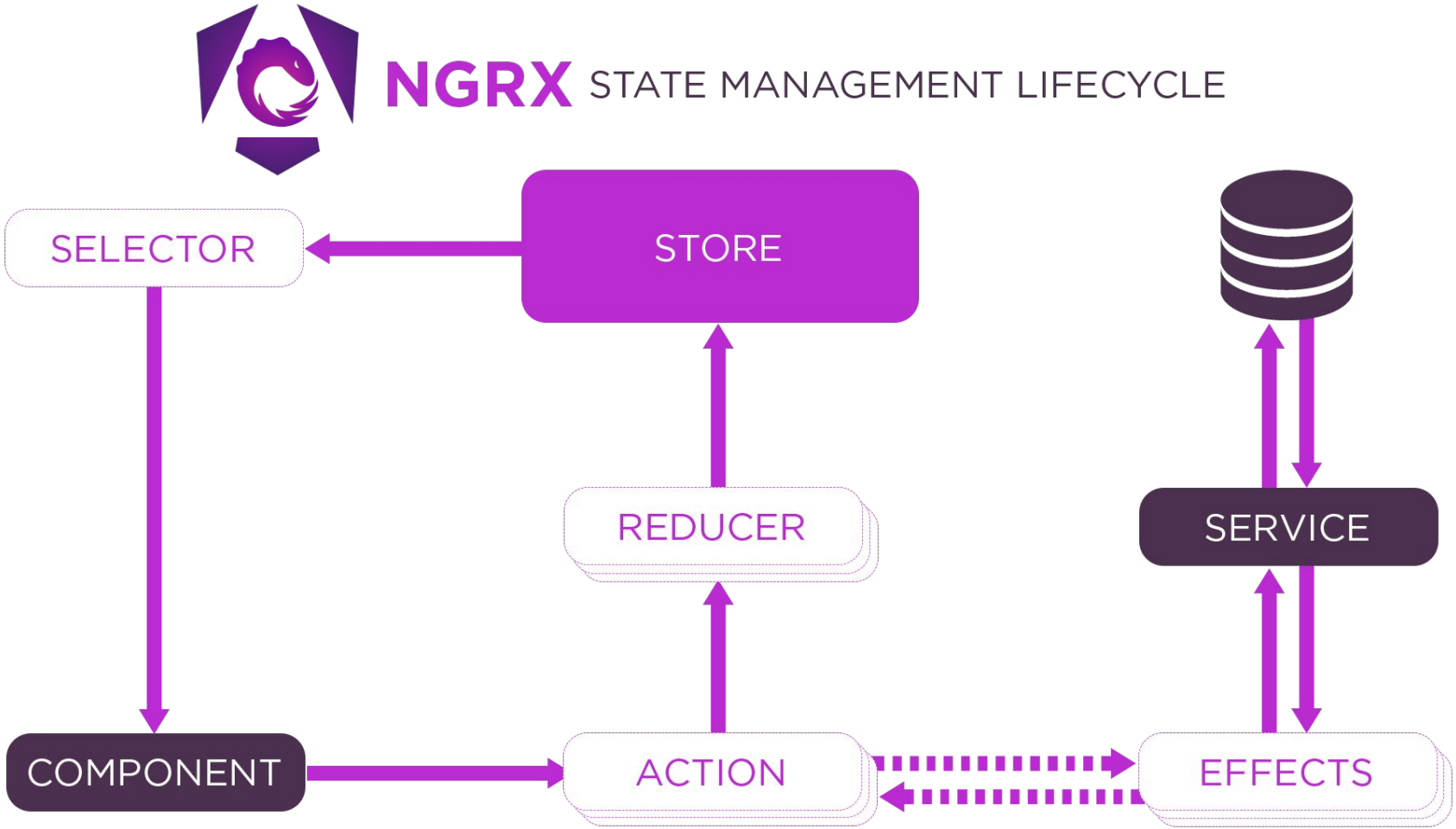
Reducers are pure functions that take the current state and an action as input and return a new state. They define how the state changes in response to actions.

Selectors

Selectors are functions that extract specific slices of the state. They are used to retrieve data from the store in a component. Selectors can also compute derived state for efficiency.

Effects

Effects handle side effects, such as asynchronous operations (e.g., API calls). They listen for specific actions, perform tasks, and dispatch new actions based on the results.


NGRX STATE MANAGEMENT LIFECYCLE
SELECTOR
STORE
REDUCER
SERVICE
COMPONENT
ACTION
EFFECTS

Setting Up an Angular 19 Project

First, ensure you have the latest **Angular CLI** installed:

```
npm install -g @angular/cli
```

Create a new Angular 19 project with standalone components:

```
ng new my-ngrx-app --standalone --style=scss
```

- **--standalone**: Ensures a standalone component structure (default in Angular 19, but included for clarity).
- **--style=scss**: Uses SCSS for styling.

Navigate to the project directory:

```
cd my-ngrx-app
```

Installing NgRx

Install the NgRx Store package:

```
ng add @ngrx/store@latest
```

This installs the core NgRx library. We'll add other NgRx packages (e.g., Entity, Effects) as needed later.

Building a Counter with NgRx

Let's build a simple counter to understand NgRx basics in a standalone setup.

Defining Actions

Actions describe events that modify the state. Create src/app/counter.actions.ts:

```
import { createAction } from '@ngrx/store';

export const increment = createAction('[Counter] Increment');

export const decrement = createAction('[Counter] Decrement');
```

Creating a Reducer

Reducers are pure functions that handle state changes based on actions. Create src/app/counter.reducer.ts:

```
import { createReducer, on } from '@ngrx/store';
import { increment, decrement } from './counter.actions';
export const initialState = 0;
const _counterReducer = createReducer(
  initialState,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1)
);
export function counterReducer(state: number | undefined, action: any) {
  return _counterReducer(state, action);
}
```

- The counter state is a simple number, initialized to 0.
- increment adds 1, and decrement subtracts 1.

Setting Up the Store

In a standalone Angular 19 app, we configure the store at the application level using provideStore in main.ts. Update src/main.ts:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideStore } from '@ngrx/store';
import { AppComponent } from './app/app.component';
import { counterReducer } from './app/counter.reducer';
bootstrapApplication(AppComponent, {
  providers: [
        provideStore({ counter: counterReducer })
  ]
});
```

- provideStore registers the counterReducer under the counter feature key.

Creating the Counter Component

Generate a standalone counter component:

```
ng generate component counter --standalone
```

Update src/app/counter/counter.component.ts:

```
import { Component } from '@angular/core';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { increment, decrement } from '../counter.actions';
@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
        <button (click)="decrement()">-</button>
        <span>{{ count$ | async }}</span>
        <button (click)="increment()">+</button>
  `,
  styles: [`span { margin: 0 10px; }`]
})
```

```
export class CounterComponent {
  count$: Observable<number>;
  constructor(private store: Store<{ counter: number }>) {
        this.count$ = this.store.select('counter');
  }
  increment() {
        this.store.dispatch(increment());
  }
  decrement() {
        this.store.dispatch(decrement());
  }
}
```

- The component is marked standalone: true.
- It uses the Store to select the counter state and dispatch actions.
- No additional imports are needed since the template only uses basic Angular features.

Managing a Todo List with NgRx Entity

Now, let’s add a todo list using **NgRx Entity** to manage a collection efficiently.

Defining the Todo Model

Create src/app/todo.model.ts:

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

Creating Todo Actions

Define actions for todo operations. Create src/app/todo.actions.ts:

```
import { createAction, props } from '@ngrx/store';

import { Todo } from './todo.model';

export const addTodo = createAction('[Todo] Add Todo', props<{ todo: Todo }>());

export const removeTodo = createAction('[Todo] Remove Todo', props<{ id: string }>());

export const toggleTodo = createAction('[Todo] Toggle Todo', props<{ id: string }>());
```

Creating a Todo Reducer

Install NgRx Entity:

```
ng add @ngrx/entity@latest
```

Create src/app/todo.reducer.ts:

```
import { createReducer, on } from '@ngrx/store';
import { createEntityAdapter, EntityAdapter, EntityState } from '@ngrx/entity';
import { Todo } from './todo.model';
import { addTodo, removeTodo, toggleTodo } from './todo.actions';

export interface TodoState extends EntityState<Todo> {}

export const adapter: EntityAdapter<Todo> = createEntityAdapter<Todo>();

export const initialState: TodoState = adapter.getInitialState();
```

```
const _todoReducer = createReducer(
  initialState,
  on(addTodo, (state, { todo }) => adapter.addOne(todo, state)),
  on(removeTodo, (state, { id }) => adapter.removeOne(id, state)),
  on(toggleTodo, (state, { id }) => {
        const todo = state.entities[id];
        return adapter.updateOne(
        { id, changes: { completed: !todo?.completed } },
        state
        );}));
export function todoReducer(state: TodoState | undefined, action: any) {
  return _todoReducer(state, action);}
export const { selectAll } = adapter.getSelectors();
```

- EntityAdapter simplifies collection management.
- selectAll is a selector to retrieve all todos.

Update src/main.ts to include the todo reducer:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideStore } from '@ngrx/store';
import { AppComponent } from './app/app.component';
import { counterReducer } from './app/counter.reducer';
import { todoReducer } from './app/todo.reducer';
bootstrapApplication(AppComponent, {
  providers: [
      provideStore({ counter: counterReducer, todos: todoReducer })
  ]
});
```

Creating the Todo List Component

Generate a standalone todo list component:

```
ng generate component todo-list --standalone
```

Update src/app/todo-list/todo-list.component.ts:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { Todo } from '../todo.model';
import { addTodo, removeTodo, toggleTodo } from '../todo.actions';
import { selectAll } from '../todo.reducer';

@Component({
  selector: 'app-todo-list',
  standalone: true,
  imports: [CommonModule],
```

```
template: `
    <input
    #todoInput
    type="text"
    placeholder="Add a todo"
    (keyup.enter)="addTodo(todoInput.value); todoInput.value = ""
    />
    <ul>
```

```
        <li *ngFor="let todo of todos$ | async">
        <input
        type="checkbox"
        [checked]="todo.completed"
        (change)="toggleTodo(todo.id)"
        />
        {{ todo.description }}
        <button (click)="removeTodo(todo.id)">Remove</button>
        </li>
        </ul>
  `,
  styles: [`ul { list-style: none; padding: 0; } li { margin: 5px 0; }`]
})
```

```
export class TodoListComponent {
  todos$: Observable<Todo[]>;
  constructor(private store: Store<{ todos: TodoState }>) {
        this.todos$ = this.store.select(selectAll);
  }
  addTodo(description: string) {
        const todo: Todo = {
        id: Date.now().toString(),
        description,
        completed: false
        };
        this.store.dispatch(addTodo({ todo }));
  }
```

```
  removeTodo(id: string) {
      this.store.dispatch(removeTodo({ id }));
  }

  toggleTodo(id: string) {
      this.store.dispatch(toggleTodo({ id }));
  }
}
```

- standalone: true marks it as a standalone component.
- imports: [CommonModule] is required for *ngFor and other common directives.

Handling Asynchronous Operations with Effects

Let’s add **NgRx Effects** to simulate fetching todos from an API.

Installing NgRx Effects

Install NgRx Effects:

```
ng add @ngrx/effects@latest
```

Creating Effects

Create a service to simulate an API call. Create src/app/todo.service.ts:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { Todo } from './todo.model';
@Injectable({
  providedIn: 'root'
})
export class TodoService {
  // Simulate an API call
  getTodos(): Observable<Todo[]> {
            return of([
            { id: '1', description: 'Learn Angular 19', completed: false },
            { id: '2', description: 'Explore NgRx', completed: true }
            ]);}}
```

Add HTTP support in src/main.ts:

```
import { provideHttpClient } from '@angular/common/http';


bootstrapApplication(AppComponent, {
  providers: [
      provideStore({ counter: counterReducer, todos: todoReducer }),
      provideHttpClient()
  ]
});
```

Add actions for loading todos. Update src/app/todo.actions.ts:

```
export const loadTodos = createAction('[Todo] Load Todos');
export const loadTodosSuccess = createAction(
  '[Todo] Load Todos Success',
  props<{ todos: Todo[] }>()
);
export const loadTodosFailure = createAction(
  '[Todo] Load Todos Failure',
  props<{ error: any }>()
);
```

Create src/app/todo.effects.ts:

```
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { of } from 'rxjs';
import { catchError, map, mergeMap } from 'rxjs/operators';
import { TodoService } from './todo.service';
import { loadTodos, loadTodosSuccess, loadTodosFailure } from './todo.actions';
@Injectable()
export class TodoEffects {
  loadTodos$ = createEffect(() =>
        this.actions$.pipe(
        ofType(loadTodos),
        mergeMap(() =>
this.todoService.getTodos().pipe(map((todos) => loadTodosSuccess({ todos })),     catchError((error) => of(loadTodosFailure({ error }))))));
  constructor(private actions$: Actions, private todoService: TodoService) {}}
```

Update src/main.ts to provide effects:

```
import { provideEffects } from '@ngrx/effects';
import { TodoEffects } from './app/todo.effects';
bootstrapApplication(AppComponent, {
  providers: [
      provideStore({ counter: counterReducer, todos: todoReducer }),
      provideEffects([TodoEffects]),
      provideHttpClient()
  ]
});
```

Update the reducer to handle loadTodosSuccess. Modify src/app/todo.reducer.ts:

```
import { loadTodosSuccess } from './todo.actions';
const _todoReducer = createReducer(
  initialState,
  on(addTodo, (state, { todo }) => adapter.addOne(todo, state)),
  on(removeTodo, (state, { id }) => adapter.removeOne(id, state)),
  on(toggleTodo, (state, { id }) => {
        const todo = state.entities[id];
        return adapter.updateOne(
        { id, changes: { completed: !todo?.completed } },
        state); }),
  on(loadTodosSuccess, (state, { todos }) => adapter.setAll(todos, state)));
```

Dispatch loadTodos in the todo list component. Update src/app/todo-list/todo-list.component.ts:

```
import { loadTodos } from '../todo.actions';
@Component({
  // ... existing metadata ...
})
export class TodoListComponent {
  todos$: Observable<Todo[]>;
  constructor(private store: Store<{ todos: TodoState }>) {
        this.todos$ = this.store.select(selectAll);
        this.store.dispatch(loadTodos()); // Load todos on init
  }
  // ... existing methods ...
}
```

Updating the Root Component

Update src/app/app.component.ts to include both components:

```
import { Component } from '@angular/core';
import { CounterComponent } from './counter/counter.component';
import { TodoListComponent } from './todo-list/todo-list.component';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CounterComponent, TodoListComponent],
  template: `       <h1>NgRx with Angular 19</h1>
            <app-counter></app-counter>
            <app-todo-list></app-todo-list> `,
  styles: [`h1 { text-align: center; }`]})
export class AppComponent {}
```

- imports includes the standalone components used in the template.

Running the Application

Start the development server:

```
ng serve
```

Open your browser to http://localhost:4200. You'll see a counter and a todo list with initial todos loaded from the simulated API.

Conclusion

This tutorial demonstrates how to integrate **NgRx** with **Angular 19**'s standalone component structure. We:

- Bootstrapped the app without AppModule using bootstrapApplication.
- Set up a store with reducers for a counter and todos.
- Used standalone components with direct imports.
- Managed collections with NgRx Entity and handled async operations with Effects.

For further exploration, check the NgRx documentation. This standalone approach simplifies Angular development while leveraging NgRx's robust state management capabilities.