neueda

Mastering angular

NgRx, Angular > V17, signal

Objectives

- Usage of NgRX: store, actions, reducer, selectors and effects.
- Whats new in recent version of Angular
    - Built-in control flow
    - Deferred views
    - Signal API
- Migrate an existing application using NgRx and the latest Angular features.

NgRx

State, reducer

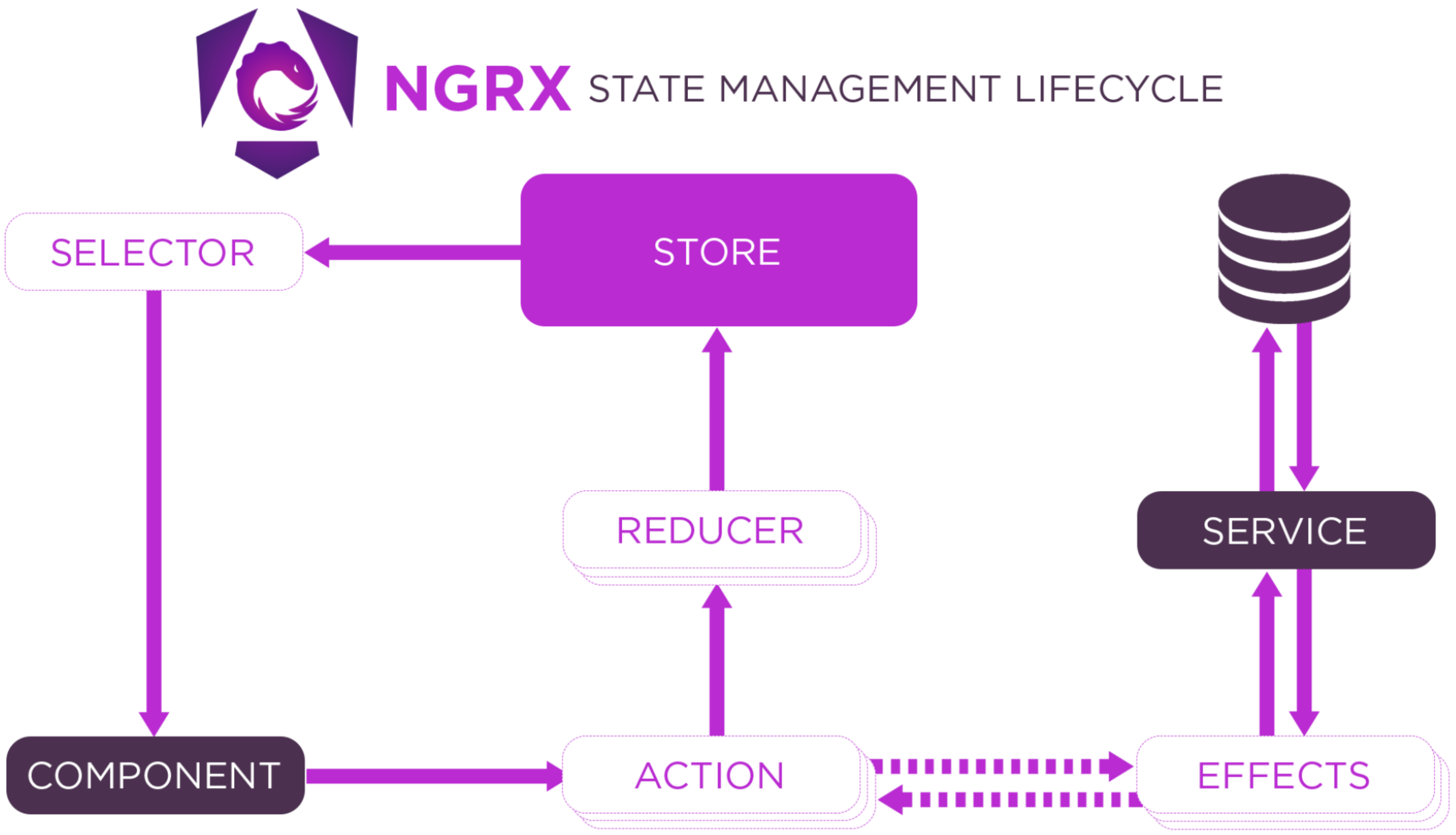
What is NgRx

- State management library for Angular applications
- Inspired by Redux
- Helps manage the state of your Angular applications in a predictable and centralized way

NgRx: Key components

- Key components of NgRx include:
  - **Actions**: Descriptive objects that represent unique events or user interactions that should trigger changes to the application state.
  - **Reducers**: Pure functions that take the current state and an action, and return a new state based on that action. Reducers are responsible for updating the state in response to actions.
  - **Selectors**: Functions that encapsulate the logic for retrieving specific pieces of state from the state tree.
  - **Effects**: Middleware for managing side effects, such as asynchronous data fetching, in NgRx applications. Effects listen for dispatched actions, perform side effects, and then dispatch new actions to update the state.

NgRx: Life cycle



neueda

Installation

- NgRx
  - First we install NgRx using the ng add command, it will update our app.module.ts
    - `ng add @ngrx/store`
- Store-devtools
  - We can install now the store-devtools package, it will allow us to debug our store in our browser, our app.module.ts is updated again.
    - `ng add @ngrx/store-devtools`
  - we need now to install a plugin to our browser so we can see our store in the developer tab
    - Chrome: https://chromewebstore.google.com/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?pli=1
    - Firefox: https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/

Creating our store.

- Store is RxJS powered global state management for Angular applications.
- Controlled state container designed to help write performant, consistent applications on top of Angular.
    - By convention the store is defined in the reducer file and registered in the app.module.ts:

```
//store/counter/counter.reducer.ts
export const initialState = 0;

export const counterReducer = createReducer(
  initialState,
);
```

```
//app.module.ts
 StoreModule.forRoot({countstore:
counterReducer}, {}),
```

Reading from our store

- NgRx provides us with a Store service we can inject into our component. This will return an observable that will emit every time the value of the store changes.

```
// let's update our counter.component.ts

count$: Observable<number>

constructor(private store: Store<{ count: number }>) {
		this.count$ = store.select('count');
 }
```

```
// Update the associated view:

Current Count: {{ count$ | async }}
```

neueda

Updating the store.

- We need now to update the value of our store.
- The common process is the following:
  - The component call the dispatch method with an action as a parameter.
  - The reducer trigger the change according of the action that have been called.
  - Two things are now then required:
    - Create actions we will call in our our components
      - counter.actions.ts
    - Triggers changes when those actions are called
      - counter.reducer.ts

Creating our actions

Action is made up of a simple interface

Action Interface interface Action { type: string; }

- **Upfront** - write actions before developing features to understand and gain a shared knowledge of the feature being implemented.
- **Divide** - categorize actions based on the event source.
- **Many** - actions are inexpensive to write, so the more actions you write, the better you express flows in your application.
- **Descriptive** - provide context that are targeted to a unique event with more detailed information you can use to aid in debugging with the developer tools.

Creating our actions

Let's create three actions that will be triggered by clicks on our buttons.

```
//counter.actions.ts

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

export const increment = createAction('[Counter] Increment');
export const decrement = createAction('[Counter] Decrement');
export const reset = createAction('[Counter] Reset');
```

Updating our reducer.

Let's define a reducer function to handle changes in the counter value based on the provided actions.

It's important to note that the reducer function's responsibility is to handle the state transitions in an immutable way.

```
export const counterReducer = createReducer(
  initialState,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1),
  on(reset, (state) => 0)
);
```

Dispatching actions.

We call the dispatch method on the store and we are passing an action so we can trigger our change in our reducer

```
//app.component.ts

increment() {
  this.store.dispatch(increment())
}
reset() {
  this.store.dispatch(reset())
}
decrement() {
  this.store.dispatch(decrement())
}
```

Passing values to our actions.

For now, our reducer is executing an action based on the previous value of our store.
What if now, we want to update the counter based on a defined value, from a form input for instance.

Creating a new action with a props as a second parameter.

```
export const updateValue = createAction('[Counter] Update Value', props<{newValue: number}>());
```

Updating the reducer:

```
…
on(updateValue, (state, {newValue}) =>  newValue)
…
```

Dispatching the action:

```
this.store.dispatch(updateValue({newValue: 50}))
```

neueda

NgRx

selectors, effects

Selectors

Selectors are pure functions used for obtaining slices of store state. @ngrx/store provides a few helper functions for optimizing this selection. Selectors provide many features when selecting slices of state:

- Portability
- Memoization
- Testability
- Composition
- Type Safety

When using the createSelector functions @ngrx/store keeps track of the latest arguments in which your selector function was invoked. The last result can be returned when the arguments match without reinvoking your selector function. This can provide performance benefits, particularly with selectors that perform expensive computation. This practice is known as memoization.

Selectors

Selectors are usualy defined in a file
- yourstorename.selectors.ts

```
export const selectUser = (state: AppState) => state.selectedUser;
export const selectAllBooks = (state: AppState) => state.allBooks;
```

Then you can call your selector in the select method of your store:

```
this.counter = this.store.select(selectAllBooks)
```

Effects: installation

Since effects are not mandatory to use allong ngrx. We need to install the corresponding library:

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

It will update our app.module.ts where we will need to declare our effects.

```
EffectsModule.forRoot([MovieEffects]),
```

Effects

In traditional approach a components using a service to making http calls has multiple responsabilities for instance, if we are dealing with movies:

- •Managing the *state* of the movies.
- •Using the service to perform a *side effect*, reaching out to an external API to fetch the movies.
- •Changing the *state* of the movies within the component:

```
@Component({
  template: `
    <li *ngFor="let movie of movies">
      {{ movie.name }}
    </li>
  `
})
export class MoviesPageComponent {
  movies: Movie[];

  constructor(private movieService: MoviesService) {}

  ngOnInit() {
    this.movieService.getAll().subscribe(movies => this.movies = movies);
  }
}
```

Effects

Effects when used along with Store decrease the responsibility of the component.
Let's write our first effect, usually in it's own file:

```
@Injectable()
export class MoviesEffects {

  loadMovies$ = createEffect(() => this.actions$.pipe(
      ofType('[Movies Page] Load Movies'),    // <-- Action that the effect is listening for,
called by dispatch
      exhaustMap(() => this.moviesService.getAll()
        .pipe(
        // When done, we basically switch to another action using a RxJs mapping function
          map(movies => ({ type: '[Movies API] Movies Loaded Success', payload: movies })),
          catchError(() => EMPTY)
        ))
    )
  );
  constructor(
    private actions$: Actions,
    private moviesService: MoviesService
  ) {}
}
```

Angular 17

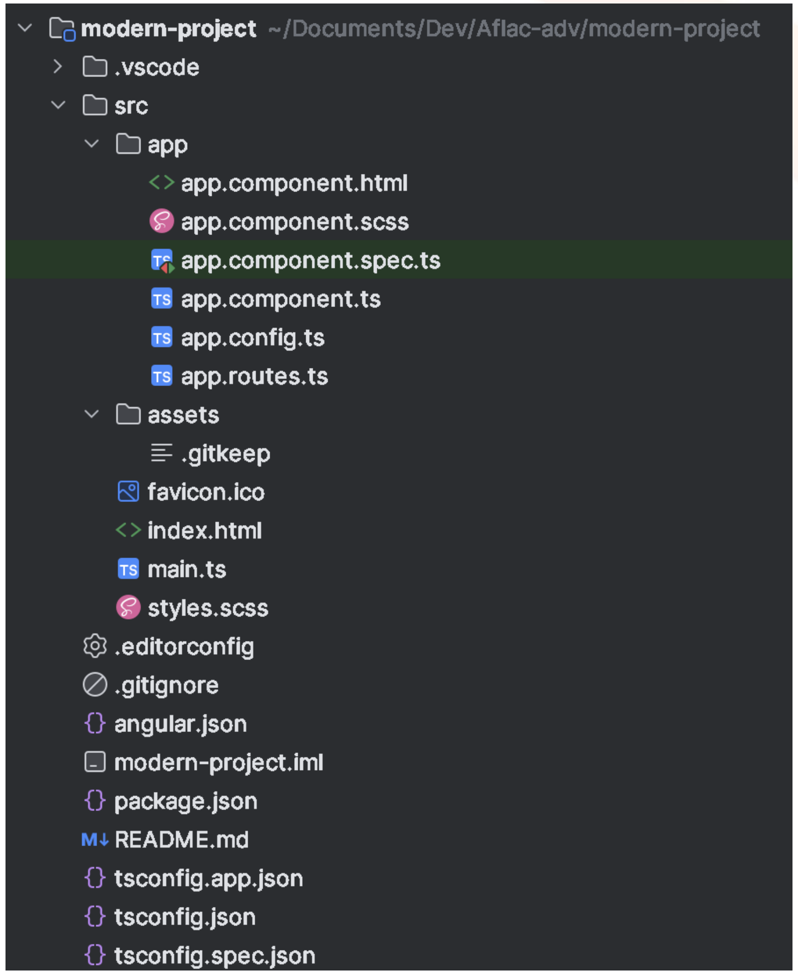
What's new!

Angular 17

- New logo!
- New documentation: https://angular.dev/
- New built-in control flow
- Deferrable views
- Standalone component
- Signal based component
- Hybrid Rendering Applications: up to 87% faster builds.
- Client-Side Rendering Applications: up to 67% faster builds.

Project structure

- There is no more app.module.ts
- Components are standalone by default
- Configuration moved from app.module.ts to app.config.ts
- app.routes.ts is generated by default.



```
modern-project ~/Documents/Dev/Aflac-adv/modern-project
  .vscode
  src
    app
      app.component.html
      app.component.scss
      app.component.spec.ts
      app.component.ts
      app.config.ts
      app.routes.ts
    assets
      .gitkeep
    favicon.ico
    index.html
    main.ts
    styles.scss
  .editorconfig
  .gitignore
  angular.json
  modern-project.iml
  package.json
  README.md
  tsconfig.app.json
  tsconfig.json
  tsconfig.spec.json
```

Angular 17

Built-in control flow

@if

Angular templates support *control flow blocks* that let you conditionally show, hide, and repeat elements.

@if:

```
@if (a > b) {
   {{a}} is greater than {{b}}
} @else if (b > a) {
   {{a}} is less than {{b}}
} @else {
   {{a}} is equal to {{b}}
}
```

@for

@for:

- The @for block requires a track expression.

```
@for (item of items; track item.id) {
    {{ item.name }}
}
```

- Many variables can be used in the @for loop:

Variable	Meaning
$count	Number of items in a collection iterated over
$index	Index of the current row
$first	Whether the current row is the first row
$last	Whether the current row is the last row
$even	Whether the current row index is even
$odd	Whether the current row index is odd

@for

@for:

- More advanced for loop example

```
@for (item of items; track item.id; let idx = $index, e = $even) {
  Item #{{ idx }}: {{ item.name }}
}
```

- @Empty keyword

```
@for (item of items; track item.name) {
    <li> {{ item.name }}</li>
} @empty {
    <li> There are no items.</li>
}
```

@switch

```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

Angular 17

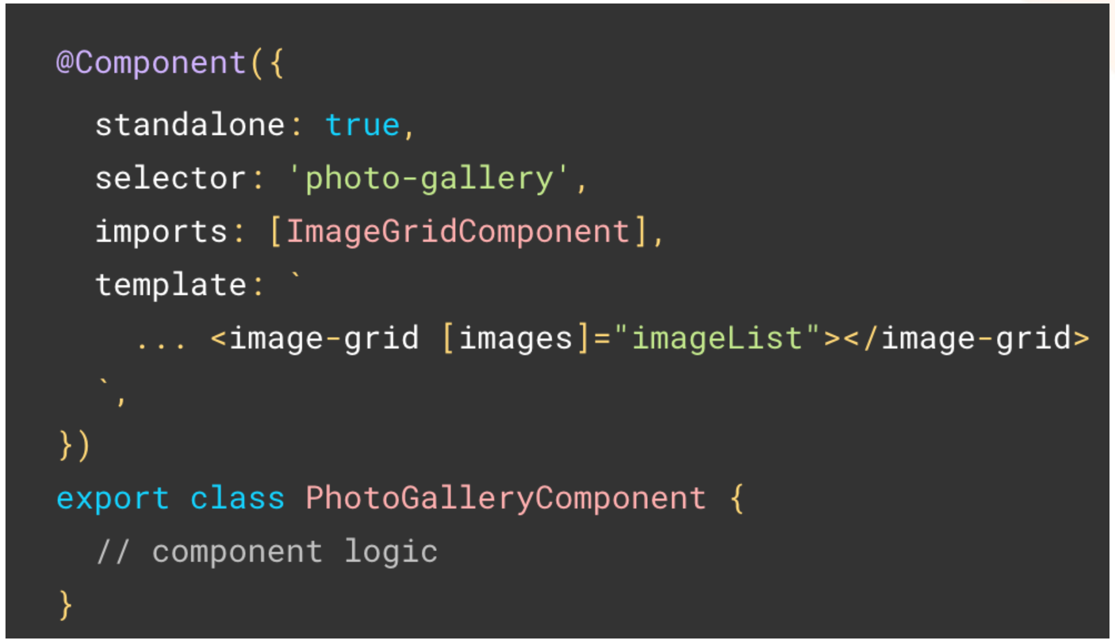
Standalone components

New standalone components

- Simplified way to build Angular applications
- Reducing the need for NgModules
- All generated components are now standalone by default

How does it work

- Declared as "standalone: true" in component metadata
  - Means that the component no longer needs to be declared in an app.module.ts.
- Need to import everything the component is using



```
@Component({
  standalone: true,
  selector: 'photo-gallery',
  imports: [ImageGridComponent],
  template: `
    ... <image-grid [images]="imageList"></image-grid>
  `,
})
export class PhotoGalleryComponent {
  // component logic
}
```

How does it work

- Can be lazy-loaded!

```
export const ROUTES: Route[] = [
  {path: 'admin', loadComponent: () => import('./admin/panel.component').then(mod =>
mod.AdminPanelComponent)},
  // ...
];
```

neueda

Angular 17

Deferrable views

What is it

- Defer the loading of select dependencies within that template.
- Reduce the initial bundle size of your application
- Great to load heavy components
- Lots of sub-blocks that allow us to have control on the deferred view

How to use

Here the view will be lazy-loaded (rendered only when the host component is rendered and when the browser status is "idle).

```
@defer {
  <large-component />
}
```

Sub-blocks

Deferrable views support a series of triggers, prefetching, and several sub blocks used for placeholder, loading, and error state management. You can also create custom conditions with when and prefetch when.

Blocks:

- @placeholder
- @loading
- @error

Triggers:

- When: when a instruction is == true
- On viewport: when the element becomes visible in the viewport, using a template variable
- On interaction: triggers the loading of the block when the user interacts with the element
- On hover: triggers the loading of the block when the user hover the element
- On timer: triggers the loading of the block after a given duration

Examples

```
@defer (when show) {
  <ns-chart />
}
@placeholder {
  <div>Something until the loading starts</div>
}
@loading {
  <div>Loading...</div>
}
@error {
  <div>Something went wrong</div>
}
```

neueda

Angular 17

Signal

What is it

- **Angular Signals** is a system that granularly tracks how and where your state is used throughout an application, allowing the framework to optimize rendering updates.
- A **signal** is a wrapper around a value that can notify interested consumers when that value changes. Signals can contain any value, from simple primitives to complex data structures.
- Always read through a getter function, which allows Angular to track where the signal is used.
- Signals may be either *writable* or *read-only*.

Defining and reading

Defining a signal value

```
count = signal(0);

albums = signal<Album[]>([])
```

Signals are getter functions - calling them reads their value.

```
console.log('The count is: ' + count());
```

```
<div> Count: {{count()}}</div>
```

neueda

Changing value

- Changing the value
  - Using 'set' to change the value directly
  - Using 'update' so you can have the current value

```
count.set(10)

count.update(c => c+1)
```

Computing values

- So far, signals var we have defined are of types `WritableSignal`.
- We can define some specifics signal variable based on others signal variables.
- We use the `computed` function
- `Signal`Type

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

Side effects.

- An **effect** is an operation that runs whenever one or more signal values change.
- You can create an effect with the effect function:
- Effects always run **at least once**
- Have to be defined in the constructor!

```
constructor(){
    effect( () => {
        console.log(`The current count is: ${count()}`);
    });
}
```

neueda

Signals with OnPush

- When an OnPush component uses a signal's value in its template, Angular will track the signal as a dependency of that component. When that signal is updated, Angular automatically marks the component to ensure it gets updated the next time change detection runs.
- Many says it's paving the way for zoneless app, which seems to be the future of Angular

neueda

Signal inputs

Signal inputs allow values to be bound from parent components. Those values are exposed using a Signal and can change during the lifecycle of your component.

- Type safe
- Values can be easily derived whenever an input changes using computed.
- Monitored using Effect
- Read Only

```
// In the component
firstName = input<string>();  //InputSignal<string|undefined>
age = input(0); // InputSignal<number>

lastName = input.required<string>(); // InputSignal<string>

// In the template, lastName is mandatory
<app-user [lastName]="MacGowan"></app-user>
```

Signal model inputs

Model inputs are a special type of input allowing to do communication between two component (two-way binding)

```
// app.component.ts

    person: Person = {
      name: 'MacGowan',
      age: 65
    }
```

```
//app.component.html

<app-person [(person)]="person"></app-
person>
```

```
// person.component.ts
person = model.required<Person>();

increasePersonAge() {
  this.person.update(
       p => ({ ...p, age: p.age + 1})
  )}
}
```

Signal queries

Queries have now their signal equivalent

Traditionnal	Signal approach	Usage

@viewChildren	viewChildren	`<div #el></div>`
		`@if (show) {`
		`    <div #el></div>`
		`}`
		
		`divEls = viewChildren<ElementRef>('el');// Signal<ReadonlyArray<ElementRef>>`

@viewChild	viewChild	`<div #el></div> <my-component />`
		
		`divEl = viewChild<ElementRef>('el'); // Signal<ElementRef|undefined>`
		`cmp = viewChild(MyComponent); // Signal<MyComponent|undefined>`

@contentChildren	contentChildren	`divEls = contentChildren<ElementRef>('h');`
		`//Signal<ReadonlyArray<ElementRef>>`

@contentChild	contentChild	`headerEl = contentChild<ElementRef>('h'); // Signal<ElementRef|undefined>`
		`header = contentChild(MyHeader); // Signal<MyHeader|undefined>`

neueda

Angular 17

Reminder about change detection.

(explaining component-tree app)

Angular 17

Further considerations

takeUntilDestroyed

- Perfect when you want to unsubscribe an observable automatically when the component is destroyed

```
data$ = http.get('...').pipe(takeUntilDestroyed());
```