

# **ANGULAR LIBRARIES**

## **LESSON 06**

**SWAFE-01**

# DEPENDENCIES

# OVERVIEW

- Technical questions to consider:
  - **Code quality**—good functionality can be implemented badly
  - **Maintainence**—is the codebase actively maintained, and who is maintaining it?
  - **Documentation**—are there proper documentation for the library?
  - **Bloated bundle size**—are the bundling process optimized?
  - Compatibility
- Business questions to consider:
  - **Vendor lock-in**—what are the cost of switching to another vendor
  - **Control over changes**—what level of influence does the users have on the feature requests and bug fixing

# POPULAR LIBRARIES

- Angular

- **AngularFire** —brings the framework-agnostic Firebase JavaScript SDK to Angular
- **Angular Material** —material design components
- **Angular Universal** —expands Core Angular APIs to enable server-side rendering
- **NgRx** —Reactive Redux-inspired state management for Angular

- Other

- **Axios** —promise based HTTP client for the browser and node.js
- **Lodash** —a modern JavaScript utility library delivering modularity, performance & extras
- **Underscore** —a JavaScript library with useful functional programming helpers

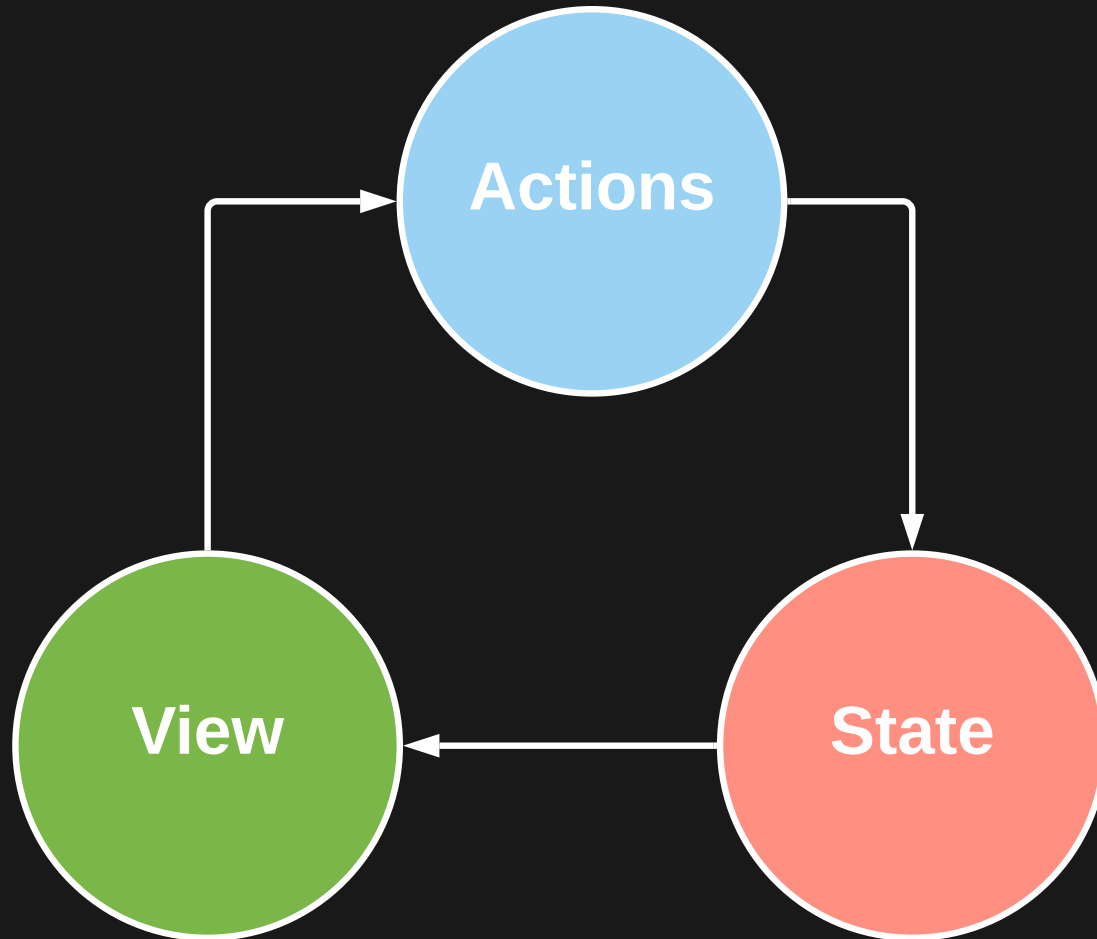
# STATE MANAGEMENT

NGRX

# OVERVIEW

- NgRX is **global** state management for Angular applications
  - Powered by RxJS
- Based on **Redux** architecture
  - Single source of truth (store)
  - **Immutable** state
  - State changes is triggered by **pure** functions
- Use when building applications with a lot of user interaction and multiple data sources
- A **solid** understanding of RxJS and Redux is beneficial before starting using NgRx

# ONE-WAY DATA FLOW



# BREAK

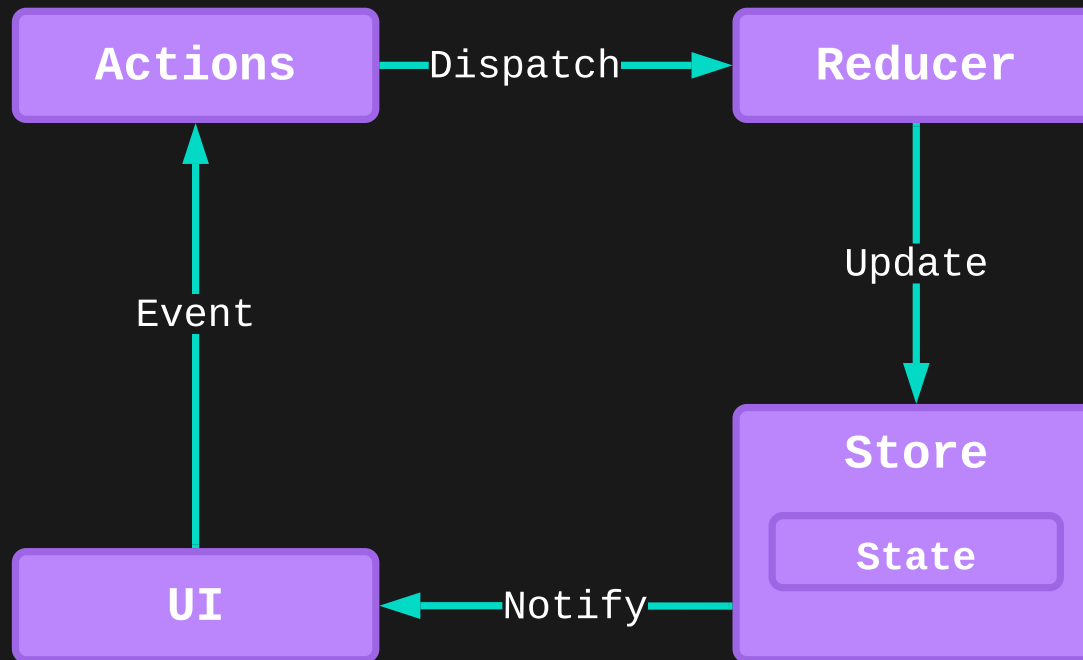
We'll be back

@

14:39:13 GMT+0100 (Central European Standard Time)



# REDUX



# STORE

```
1 class Store<T = object> extends Observable<T> implements Observer<Action>
```

<https://github.com/ngrx/platform/blob/master/modules/store/src/store.ts>

- **Manages** global application state
- Two different store are available:
  - **Store** —used for global application-wide state management
  - **ComponentStore** —used for local state management, e.g. in a component
- Use **Actions** to express state change

# STORE

```
1 import { NgModule } from '@angular/core';
2 import { BrowserModule } from '@angular/platform-browser';
3
4 import { AppRoutingModule } from './app-routing.module';
5 import { AppComponent } from './app.component';
6 import { CollectionEffects } from './state/collection.effects';
7 import { collectionReducer } from './state/collection.reducer';
8 import { DeviceEffects } from './state/device.effects';
9 import { DeviceListComponent } from './device-list/device-list.component';
10 import { devicesReducer } from './state/device.reducer';
11 import { StoreModule } from '@ngrx/store';
12 import { EffectsModule } from '@ngrx/effects';
13
14 @NgModule({
15   declarations: [
16     AppComponent,
17     DeviceListComponent
18   ],
19   imports: [
```

examples/lesson06-angular-libraries/projects/ngrx/src/app/app.module.ts

# ACTIONS

- Actions are the **main** building blocks of NgRx
- They define *unique* events that happens in applications
- Writing actions
  - Define action **upfront** to understand and gain knowledge about the feature being implemented
  - **Categorize** actions based on event sources
  - Capture **events** not commands, **separate** description and handling of events
  - Provide **unique** context for events to aid debugging
- They serve as both **inputs** and **outputs**

# ACTIONS

```
1 import { createAction, props } from "@ngrx/store";
2 import { Device } from "../device.model";
3
4 export const addDevice = createAction(
5   `[Device] Add Device`,
6   props<{ deviceId: string }>()
7 )
8
9 export const removeDevice = createAction(
10   `[Device] Remove Device`,
11   props<{ deviceId: string }>()
12 )
13
14 export const loadDevices = createAction(
15   `[Device/API] Load Devices`
16 )
17
18 export const retrievedList = createAction(
19   `[Device/API] Retrieved List`,
```

examples/lesson06-angular-libraries/projects/ngrx/src/app/state/device.actions.ts

# REDUCERS

- Reducers are responsible for handling state **transitions** in applications
- Reducers are pure functions that produce the **same** output for a given input
- A reducer function consists of:
  - An **interface** or type that defines the shape of the state
  - **Arguments** for current state and action
  - A **function** that handles state change
- Reducers are registered with the **Store** in module files
  - Use feature states to separate state handling for individual features modules
  - Can be loaded eagerly or lazily depending on the application needs

# REDUCERS

```
1 import { createReducer, on } from '@ngrx/store';
2 import { addDevice, removeDevice } from '../device.actions';
3
4 export const initialState: ReadonlyArray<string> = [];
5
6 export const collectionReducer = createReducer(
7   initialState,
8   on(removeDevice, (state, { deviceId }) => state.filter((id) => id !== deviceId),
9   on(addDevice, (state, { deviceId }) => {
10     if (state.indexOf(deviceId) > -1) {
11       return state;
12     }
13     return [...state, deviceId];
14   })
15 );
```

examples/lesson06-angular-libraries/projects/ngrx/src/app/state/collection.reducer.ts

# SELECTORS

- Selectors are **pure** functions used for obtaining **slices** of store state
- Selectors implements **memoization**
  - The store keeps track of **latest** selector argument values, and since they are pure functions, it can return the current state without invoking the function
  - This can provide **performance** benefits, if selectors perform **expensive** computation and/or I/O operations
- `createSelector` can be used to select data from several slices (up to 8) of the same state

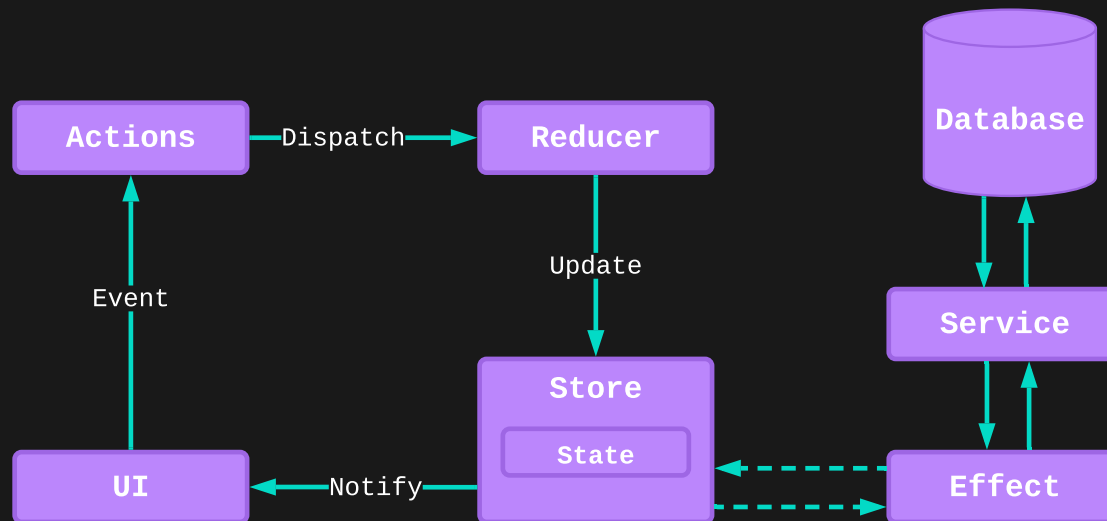


# SELECTORS

```
1 import { createSelector, createSelector } from "@ngrx/store";
2 import { AppState } from "../state/app.state";
3 import { Device } from "../device.model";
4
5 export const selectDevices = createSelector(
6   (state: AppState) => state.devices,
7   (devices: ReadonlyArray<Device>) => devices
8 )
9
10 export const selectCollectionState = createSelector<AppState, ReadonlyA
11
12 export const selectDeviceCollection = createSelector(
13   selectDevices,
14   selectCollectionState,
15   (devices: ReadonlyArray<Device>, collection: ReadonlyArray<string>) => {
16     return collection.map(id => devices.find((device) => device.name === id)
17   }
18 );
```

examples/lesson06-angular-libraries/projects/ngrx/src/app/state/device.selector.ts

# NGRX



# EFFECTS

- RxJS-powered **side effect** model for **Store**
- Effects uses **streams** to provide new sources of actions to reduce state based on **external** actions
  - Network requests
  - Web socket message
  - Time-based events

# EFFECTS

- Effects **isolate** side effects from components, allowing for more **pure** components
- Long-running services that listens to an **Observable** of every **Action** dispatched from the **Store**
- Effects **filter** **Action** objects based on interest
- Effects perform tasks that **produces** new **Action** objects which is handled by the **Store**

# EFFECTS

```
1 import { Injectable } from '@angular/core';
2 import { Actions, createEffect, ofType } from '@ngrx/effects'
3 import { EMPTY } from 'rxjs';
4 import { map, mergeMap, catchError } from 'rxjs/operators';
5 import { DeviceService } from '../device.service';
6 import { loadDevices, retrievedList } from './device.actions';
7
8 @Injectable()
9 export class DeviceEffects {
10   loadDevices$ = createEffect(() =>
11     this.actions$.pipe(
12       ofType(loadDevices),
13       mergeMap(() => this.deviceService$.getDevices()
14         .pipe(
15           map(devices => (retrievedList({ devices }))),
16           catchError(() => {
17             return EMPTY
18           })
19         ))
20   ))
```

examples/lesson06-angular-libraries/projects/ngrx/src/app/state/device.effects.ts

# EFFECTS

- NgRx and Redux differences:
  - Effects lives **outside** **Store** objects
  - Effects **communicate** with services (which can produce side-effects)
  - Redux actions are **only** handled in **reducers**, but in NgRx, some actions are handled in **reducers**, some in **effects**, and some in **both**

# BREAK

We'll be back

@

14:39:13 GMT+0100 (Central European Standard Time)

# COMPONENT

```
1 import { Component, OnInit } from '@angular/core';
2 import { select, Store } from '@ngrx/store';
3 import { AppState } from '../state/app.state';
4 import { addDevice, loadDevices, removeDevice } from '../state/device.actions'
5 import { selectDeviceCollection, selectDevices } from '../state/device.selectors'
6
7 @Component({
8   selector: 'app-device-list',
9   templateUrl: './device-list.component.html',
10  styleUrls: ['./device-list.component.scss']
11 })
12 export class DeviceListComponent implements OnInit {
13
14   devices$ = this.store.pipe(select(selectDevices))
15   deviceCollection$ = this.store.pipe(select(selectDeviceCollection));
16
17   constructor(private store: Store<AppState>) { }
18
19   ngOnInit(): void {
```

examples/lesson06-angular-libraries/projects/ngrx/src/app/state/device.effects.ts



# THE SHARI PRINCIPLES

- You should consider using a state management system if you have these challenges:
  - **S—Shared** State is accessed by many components and services
  - **H—Hydrated** State that is persisted and hydrated from storage
  - **A—Available** State that needs to be available when re-entering routes
  - **R—Retrieved** State that needs to be retrieved with a side effect
  - **I—Impacted** State that is impacted by actions from other sources
- Using NgRx comes with the price of code **complexity**, **indirection** and high code cost

# WRAP-UP

- NgRx is a global state management **system** for Angular
  - Extends Redux with **effects** to accommodate Angular patterns
- Encapsulates **side-effects** in injectable effects
  - Allows implementation of pure and simple components
  - Promotes the single responsibility principle
- Immutability and performance
  - Immutable data structures makes change detection easy
  - Memoized selectors **optimizes** retrieval of data from state
- Consider the SHARI principles to determine if you really need NgRx

# ANGULAR MATERIAL

Run `ng serve --port 4200 --project slide-examples` in `examples/lesson06-angular-libraries`

# OVERVIEW

- The Angular Material library provides **common** UI components and tools to build custom components
- The library is **maintained** by the Angular team
- Other interesting libraries:
  - `@angular/cdk` —write custom UI components with common interaction patterns
  - `@angular/material` —Material Design components for Angular applications
  - `@angular/google-maps` —Angular components built on top of the Google Maps JavaScript API
  - `@angular/youtube-player` —Angular components built on top of the YouTube Player API
- Customizable within the bounds of the Material Design specification

# DESIGN SYSTEMS

- A design systems is a system of **patterns** and **styles** that is used when creating applications
  - Patterns describes **flows**, **behaviors** and how to combine elements
    - Animation
    - Layout
    - Sounds
  - Styles describes **visual** elements
    - Typography
    - Colors
    - Iconography
- There are several design systems available (see notes)

# THEMES

- A theme is used to define **colors** and **typography** for an application
- It is what makes up "the look and feel" of applications
- The framework exposes a theming API
  - Built with **Sass**
- Material Design comes with **built-in** themes, that can be used as-is

# PALETTES

- Palettes apply color to UI in a meaningful way
  - Primary and secondary colors
  - Variants of primary and secondary colors
  - Additional colors for specific uses, e.g. backgrounds, errors, typography, etc.
- It is possible to define custom themes based on custom palettes

# CUSTOM THEME

```
1 // Custom Theming for Angular Material
2 // For more information: https://material.angular.io/guide/theming
3 @use '~@angular/material' as mat;
4 @use './orbit-palette-primary' as olp;
5 @use './orbit-palette-secondary' as ols;
6
7 // Plus imports for other components in your app.
8
9 // Include the common styles for Angular Material. We include this here so tha
10 // have to load a single css file for Angular Material in your app.
11 // Be sure that you only ever include this mixin once!
12 @include mat.core();
13
14 // Define the palettes for your theme using the Material Design palettes avail
15 // (imported above). For each palette, you can optionally specify a default, l
16 // hue. Available color palettes: https://material.io/design/color/
17 $material-primary: mat.define-palette(olp.$orbit-palette-primary);
18 $material-accent: mat.define-palette(ols.$orbit-palette-secondary);
19
```

examples/lesson06-angular-libraries/projects/material/src/styles.scss



# CUSTOM PALETTE

```
1 $dark-primary-text: rgba(white, 0.87);
2 $light-primary-text: white;
3
4 $orbit-palette-primary: (
5   50: #e2eef4,
6   100: #b7d5e6,
7   200: #8cbbd6,
8   300: #65a0c5,
9   400: #4a8ebb,
10  500: #317db1,
11  600: #2871a6,
12  700: #1d6195,
13  800: #145284,
14  900: #053765,
15
16  contrast: (
17    50: $dark-primary-text,
18    100: $dark-primary-text,
19    200: $dark-primary-text,
```

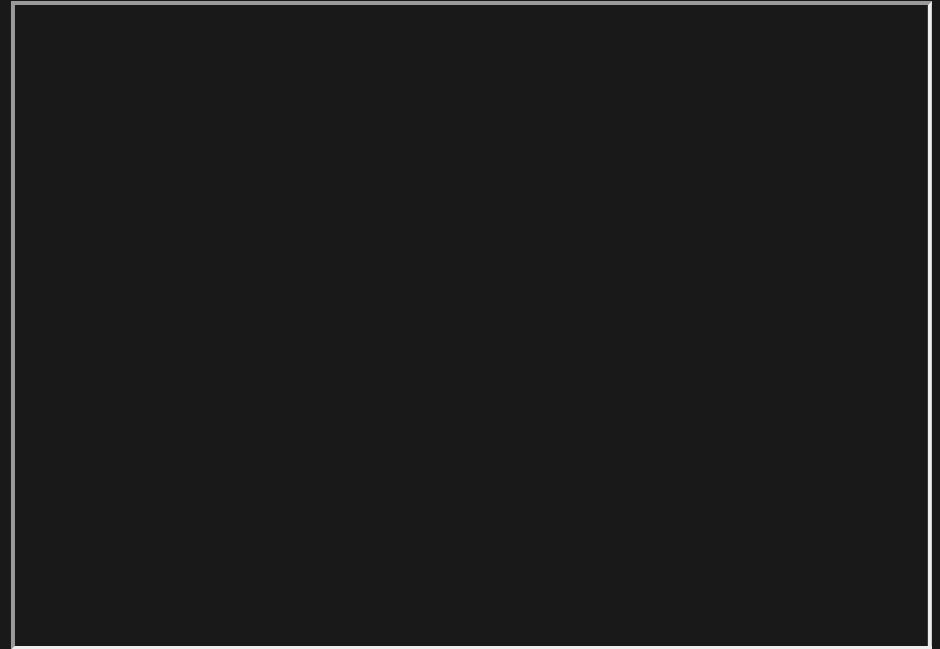
examples/lesson06-angular-libraries/projects/material/src/\_orbit-palette-primary.scss

# COMPONENTS

- Angular Material provides a wide range (36) of UI components based on Material Design
- Some examples
  - **App bars**—the top app bar displays information and actions related to the current screen
  - **Buttons**—allows users to take actions, and make choices, with a single tap
  - **Cards**—contains content and actions about a single subject
  - **Date picker**—allows users to select a date, or a range of dates
- Check out [Components – Material Design](#) for an in-depth description of use of each component

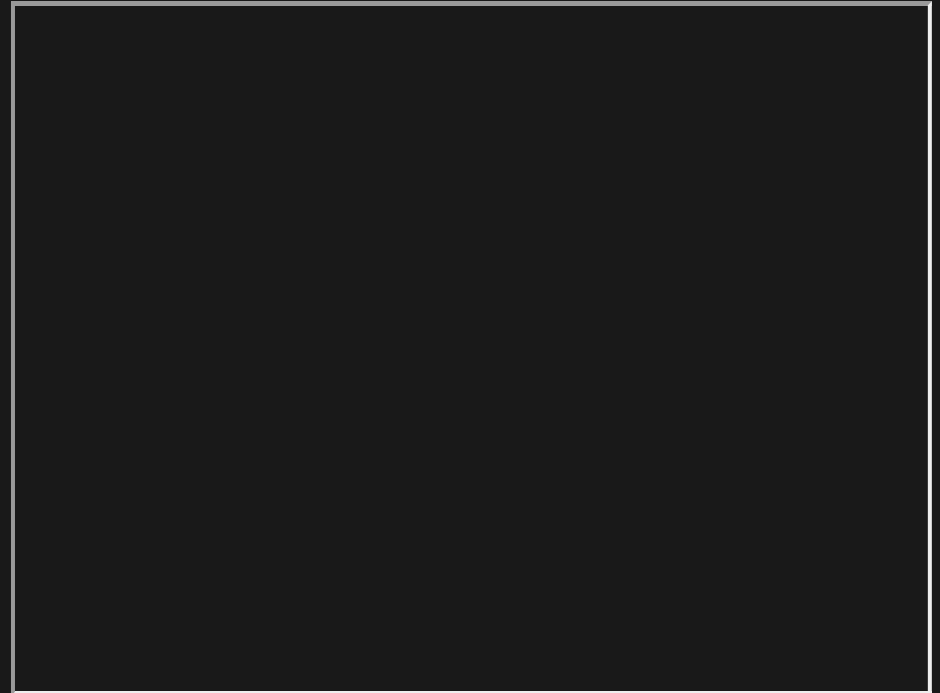
# mat-card

- Content container for text, photos, and actions the context of a single subject
- A `mat-card` has sections
  - Header
  - Content
  - Actions
  - Footer



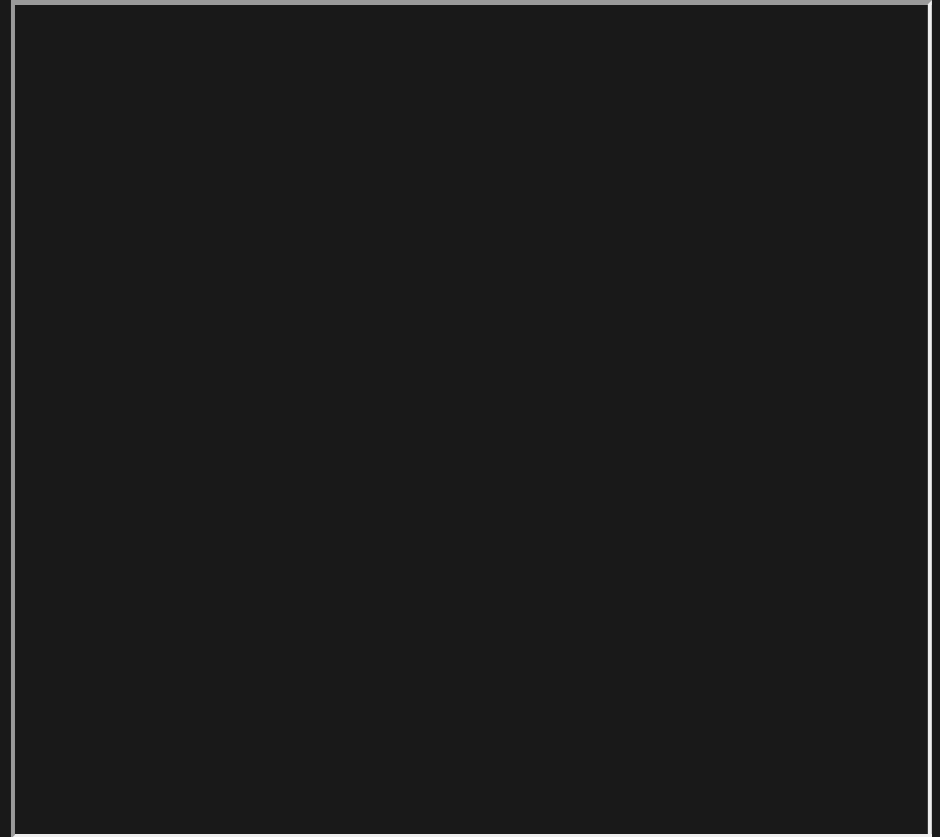
# mat-table

- `mat-table` is focused on rendering rows of table data
- Bind array or a `DataSource` instance
- Provide `Observable` streams where the table will render when new data is emitted



# mat-date-range-input

- Allow users to enter a date through text input, or by choosing a date from the calendar
  - Made up of several components, directives and modules
- Can be used with `FormGroup` directive from `@angular/forms`



# CHANGE DETECTION

# OVERVIEW

- Most changes happens asynchronously
- Angular uses the `zone.js` library
  - Subscribes to Zone events
  - Keeps the component's model and UI in sync
- An Angular application is structured as a tree of views

# CHANGE DETECTION STRATEGIES

- Angular offers two CD strategies for UI updates
  - `Default` —the Zone checks the entire component tree
  - `OnPush` —the Zone checks the component and children only if the component's input properties has changed



# CHANGE DETECTION

Event in the descendant triggers CD cycle, starting from top. Child2 and its descendants are excluded from CD cycle unless bindings to input properties of Child2 change.



Figure 9.2 Excluding a branch from a CD cycle

# WRAP-UP

- Dependencies
- State management
  - Actions
  - Reducers
  - Stores
- Angular Material
  - Patterns and styles
  - Components
- Change detection

