# 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 Flex–provides a sophisticated layout API using Flexbox CSS + mediaQuery
  - 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
  - jQuery–a JavaScript library, that makes document traversal and manipulation, event handling, animation, and Ajax simpler
  - Underscore—a JavaScript library with useful functional programming helpers

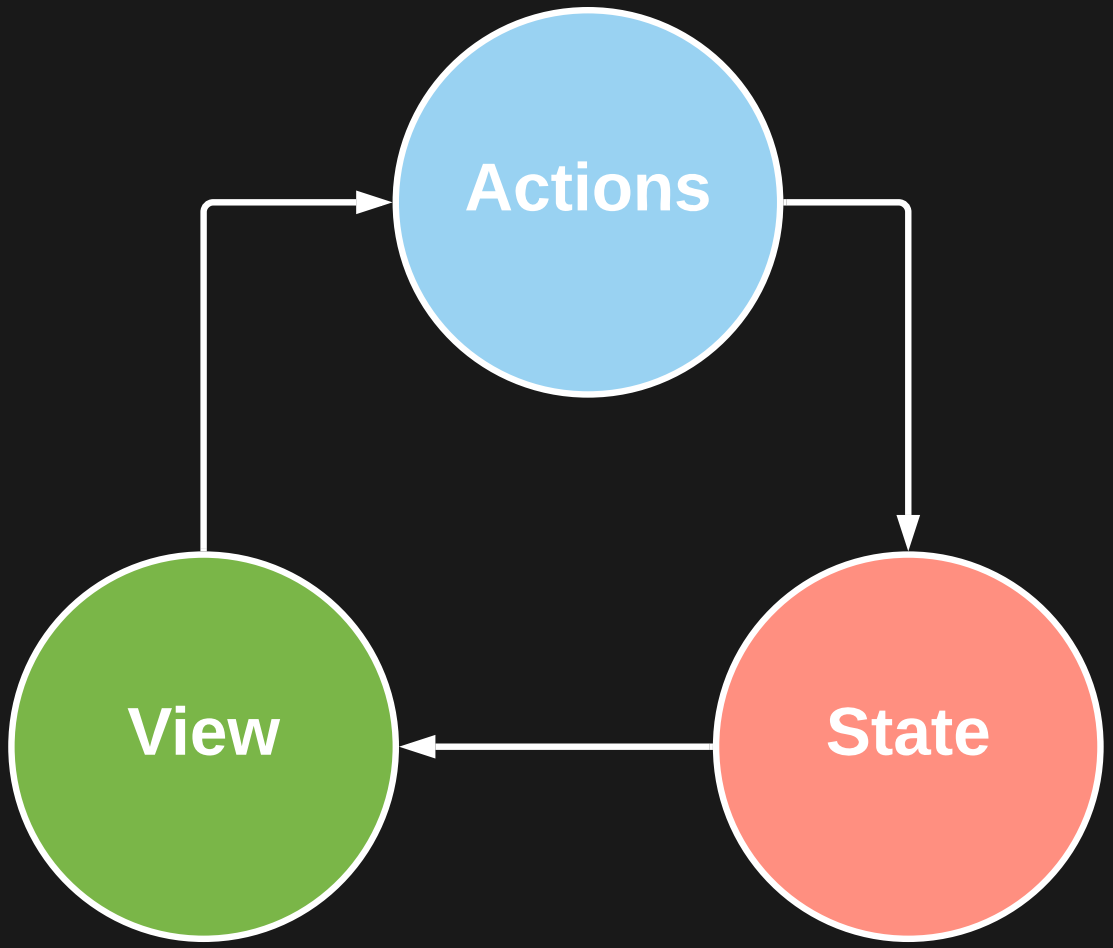
# STATE MANAGEMENT

## NGRX

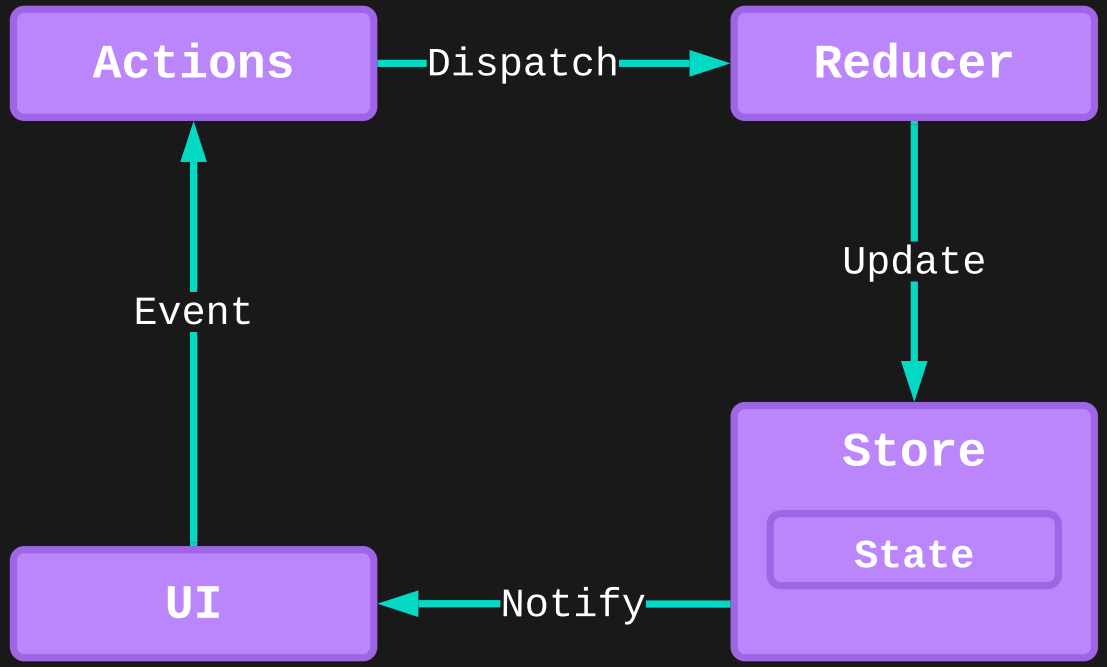
# OVERVIEW

- NgRX is global state management for Angular applications
  - Powered by RxJS
  - Inspired by Redux
- Based on Redux architecture
  - Single source of thruth
  - 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



# REDUX


Actions
Dispatch
Reducer
Update
Store
State
Event
UI
Notify

# 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 managemeent
  - `ComponentStore` —used for local state management, e.g. in a component
- Use `Actions` to express state change

# STORE

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CollectionEffects } from './state/collection.effects';
import { collectionReducer } from './state/collection.reducer';
import { DeviceEffects } from './state/device.effects';
import { DeviceListComponent } from './device-list/device-list.component';
import { devicesReducer } from './state/device.reducer';
import { StoreModule } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';

@NgModule({
  declarations: [
    AppComponent,
    DeviceListComponent
  ],
  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

```
import { createAction, props } from "@ngrx/store";
import { Device } from "./device.model";

export const addDevice = createAction(
  `[Device] Add Device`,
  props<{ deviceId: string}>()
)

export const removeDevice = createAction(
  `[Device] Remove Device`,
  props<{ deviceId: string}>()
)

export const loadDevices = createAction(
  `[Device/API] Load Devices`
)

export const retrievedList = createAction(
  `[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 the current 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

```
import { createReducer, on } from '@ngrx/store';
import { addDevice, removeDevice } from './device.actions';

export const initialState: ReadonlyArray<string> = [];

export const collectionReducer = createReducer(
  initialState,
  on(removeDevice, (state, { deviceId }) => state.filter((id) => id !== device
  on(addDevice, (state, { deviceId }) => {
    if (state.indexOf(deviceId) > -1) {
      return state;
    }
    return [...state, deviceId];
  })
);
```

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

# SELECTORS

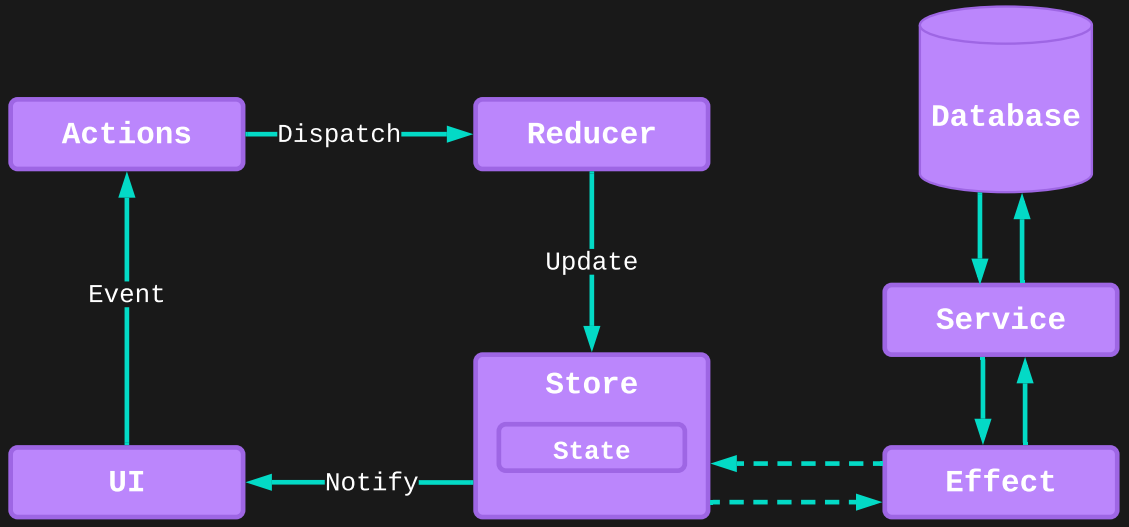
- Selectors are pure functions used for obtaining slices of store state
- Selectors implements memoization
  - The store keeps track of latest selector argument values, and since they are pure functions, it can return the current state without invoking the function
  - This can provide performance benefits,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

# SELECTORS

```
1  import { createFeatureSelector, 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 = createFeatureSelector<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


Actions
Dispatch
Reducer
Database
Event
Update
Service
Store
State
UI
Notify
Effect

# EFFECTS

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

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

# EFFECTS

- NgRx and Redux differences:
  - Effect lives outside `Store` objects
  - Effets 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

# 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.selecto
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 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
  - Material Design (Google)
  - Fluent Design System (Microsoft)
  - Carbon Design System (IBM)
  - Lightning Design System (Salesforce)

# 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 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

```scss
// Custom Theming for Angular Material
// For more information: https://material.angular.io/guide/theming
@use '~@angular/material' as mat;
@use './orbit-palette-primary' as olp;
@use './orbit-palette-secondary' as ols;

// Plus imports for other components in your app.

// Include the common styles for Angular Material. We include this here so tha
// have to load a single css file for Angular Material in your app.
// Be sure that you only ever include this mixin once!
@include mat.core();

// Define the palettes for your theme using the Material Design palettes avail
// (imported above). For each palette, you can optionally specify a default, l
// hue. Available color palettes: https://material.io/design/color/
$material-primary: mat.define-palette(olp.$orbit-palette-primary);
$material-accent: mat.define-palette(ols.$orbit-palette-secondary);

```

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

# CUSTOM PALETTE

```scss
$dark-primary-text: rgba(white, 0.87);
$light-primary-text: white;

$orbit-palette-primary: (
  50: #e2eef4,
  100: #b7d5e6,
  200: #8cbbd6,
  300: #65a0c5,
  400: #4a8ebb,
  500: #317db1,
  600: #2871a6,
  700: #1d6195,
  800: #145284,
  900: #053765,

  contrast: (
    50: $dark-primary-text,
    100:$dark-primary-text,
    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
    - Title
    - Subtitle
    - Avatar
  - Content
  - Actions
  - Footer

# `mat-table`

- `mat-table` is focused on rendering rows of table data
- Table features
- Define column and row templates with `matColumnDef` and `matRowDef`
  - Pagination
  - Sorting
  - Filtering
  - Selection
- 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`
- Can be used with various date libraries (date implementation agnostic)
  - `MatNativeDateModule`
  - `MatLuxonDateModule`
  - `MatLuxonDateModule`
- A datepicker is composed of a text input and a calendar pop-up, connected via the `matDatepicker` property on the text input.

# ANGULAR FLEX

# OVERVIEW

- Angular Flex Layout combines Flexbox CSS and media queries
- Automates the process of applying appropiate Flexboxx CSS to browser view hierarchies
- Addresses many of the complexities and workarounds with traditional, manual application of Flexbox CSS
- The Responsive API enables developers to easily specify different layout, sizing for different viewports sizes and display devices
- It provides an Angular-friendly API for dealing with media queries

# DECLARATIVE API OVERVIEW

- Contains two APIs
  - Static API–provides directives to create responsive and adaptive layouts
  - Responsive API–provides responsive features to change layouts configuration for different display devices

# API FOR DOM CONTAINERS — `flex`

- These directives affects the flow and layout of children elements in the container
  - `fxLayout` –Used on DOM container whose children should layout
    - A combination of `display: flex` and `flex-direction: row|column|row-reverse|column-reverse`
  - `fxLayoutAlign`
    - A combination of `justify-content` and `align-items`

# API FOR DOM CONTAINERS — `grid`

- These directives affects the flow and layout of children elements in the container
  - `gdColumns` –equivalent to `grid-template-columns`
  - `gdRows` –equivalent to `grid-template-rows`
  - `gdAlignRows` –aligns items row-wise
    - `justify-content` (container) and `justify-items` (items)
  - `gdAlignColumns` –aligns items column-wise
    - `align-content` (container) and `align-items` (items)

# API FOR DOM ELEMENTS

- These directives affects the layout and size of the host element (expected to be in a container)
  - `fxFlex` –sets properties for grow, shrink and basis for an item
  - `fxFlexAlign` –equivalent to `align-self`
  - `gdRow` -determines start and end row for an item
  - `gdColumn` —determines start and end column for an item
- These are directives for any element
  - `fxShow` –specifies if its host element should be displayed
  - `fxHide` –specifies if its host element should NOT be displayed

# RESPONSIVE API

- Enhances the Static API by adding responsive suffixes
- Angular Layout will automatically handle all the details
  - Monitoring media queries activations
  - Applying responsive values to the host DOM elements

# BREAKPOINTS AND ALIASES

- Breakpoints
  - `xs` –Extra small `screen and (max-width: 599px)`
  - `sm` –Small `screen and (min-width: 600px) and (max-width: 959px)`
  - `md` –Medium `screen and (min-width: 960px) and (max-width: 1279px)`
  - `lg` –Large `screen and (min-width: 1280px) and (max-width: 1919px)`
  - `xl` –Extra large `screen and (min-width: 1920px) and (max-width: 5000px)`
- Less than
  - `lt-sm` –Less than small `screen and (max-width: 599px)`
  - `lt-md` –Less than medium `screen and (max-width: 959px)`
  - `lt-lg` –Less than large `screen and (max-width: 1279px)`
  - `lt-xl` –Less than extra large `screen and (max-width: 1919px)`
- Greater than
  - `gt-xs` –Greater than extra small `screen and (min-width: 600px)`
  - `gt-sm` –Greater than small `screen and (min-width: 960px)`
  - `gt-md` –Greater than medium `screen and (min-width: 1280px)`
  - `gt-lg` –Greater than large `screen and (min-width: 1920px`

# COMBINE STATIC API AND `alias`

Support resposive breakpoints by combining an `alias` with a directive

```
<api>.<breakpoint alias>="<value>"
[<api>.<breakpoint alias>]="<expression>"
```

# BREAKPOINT

```
<div
  fxLayout.gt-xs="row"
  fxLayout.xs="column"
  [ngStyle.lt-sm]="{'background-color': 'cornflowerblue'}">
  <div fxFlex.xs="1 0 auto" fxFlexAlign.lt-sm="end" class="child"></div>
  <div class="child"></div>
  <div class="child"
    [ngClass.xs]="{'small-border small': hasStyle }"
    [ngClass.gt-xs]="{'small': hasStyle }">
  </div>
</div>
<p fxShow.xs fxHide.gt-xs>Hidden element</p>
<div fxHide.xs class="child"></div>
```

examples/lesson06-angular-libraries/projects/flex/src/app/breakpoint/breakpoint.component.html