

# Angular

Superheroic framework



# Training Agenda

- ☐ TypeScript : Introduction
- ☐ Angular : Introduction
- ☐ Angular Building Blocks
  - Component
  - Modules
  - Directives
  - Services
  - Pipes
- ☐ Bindings
- ☐ Forms
- ☐ Remote Calls
- ☐ Routing
- ☐ Authentication / Authorization
- ☐ PWA
- ☐ NGRX

Angular is easy

ES2015+ features

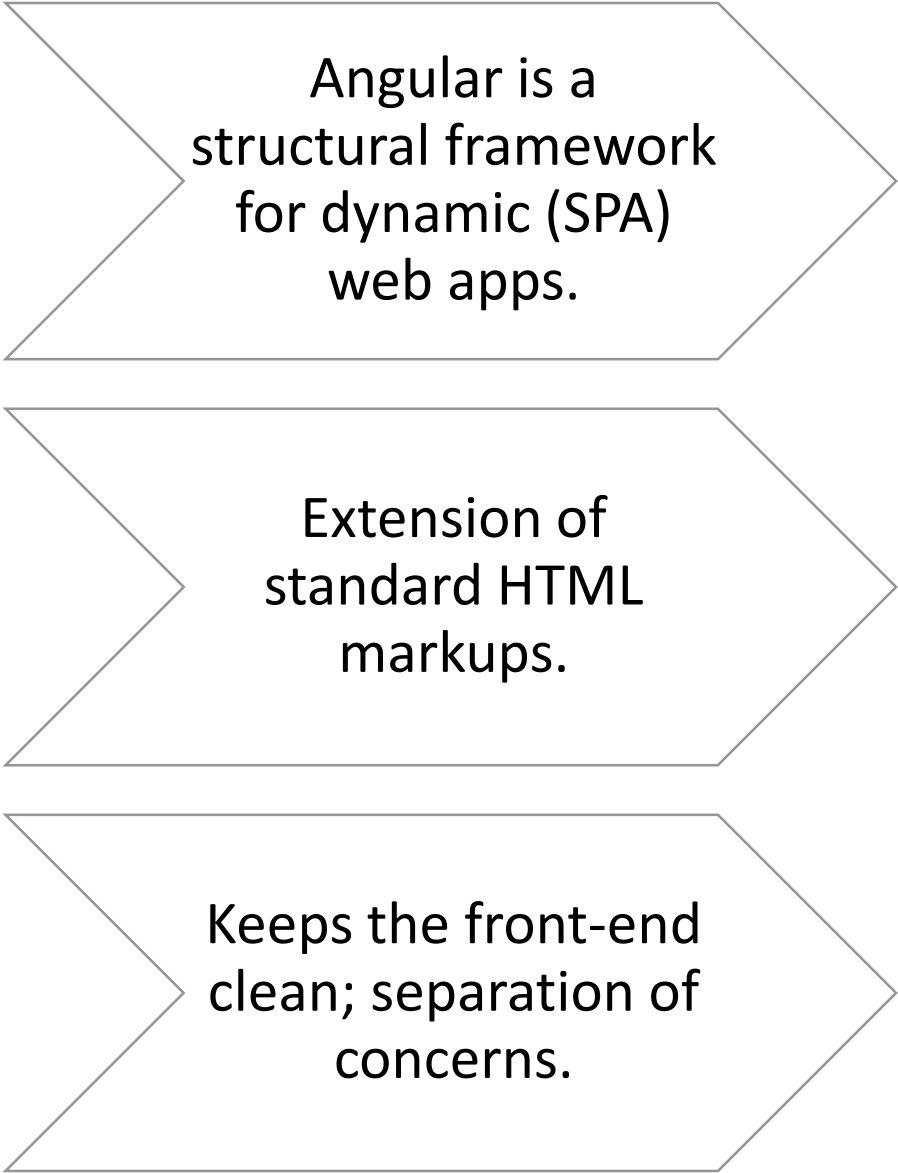
Corporate Care-taker

Performance and Mobile

Project architecture and Maintenance

Component based architecture

# Why Angular ?



Angular is a  
structural framework  
for dynamic (SPA)  
web apps.

Extension of  
standard HTML  
markups.

Keeps the front-end  
clean; separation of  
concerns.

# What is Angular ?

---

Microsoft extension for JS

---

Object oriented features

---

ES6+ features

---

Type definition

---

Angular itself programmed in TS

Why Typescript ?

# TypeScript features

- Classes & Inheritance
- Module system
- Arrow function
- Template String
- Constants and block scope
- Destructuring
- Spread & Rest operator
- Decorator
- Additional types

# Understanding Angular Environment Setup

- Node
- TypeScript
- Webpack
- Angular Packages
- RxJS
- ZoneJS

# Components

- A component controls a patch of screen real estate that we could call a view, and declares reusable UI building blocks for an application.
- Passing data to/from components
  - Property binding
  - Event binding
  - Two way data-binding
- Nested components
  - Parent to Child
  - Child to Parent
- Data projection
- Component types :
  - Smart components
  - Dumb components



*ngOnChanges* - called when an input binding value changes

*ngOnInit* - after the first *ngOnChanges*

*ngDoCheck* - after every run of change detection

*ngAfterContentInit* - after component content initialized

*ngAfterContentChecked* - after every check of component content

*ngAfterViewInit* - after component's view(s) are initialized

*ngAfterViewChecked* - after every check of a component's view(s)

*ngOnDestroy* - just before the component is destroyed

# Component Life Cycle

# Directives

- A Directive modifies the DOM to change appearance, behavior or layout of DOM elements.
- Directive Types :
  - *Component Directive* : directive with template
  - *Attribute Directive* : directives that change the behavior of a component or element but don't affect the template
  - *Structural Directives* : directives that change the behavior of a component or element by affecting how the template is rendered

# Pipes

- Pipes are used to filter/format data for template
- Built-in Pipes :
  - Currency
  - Date
  - Uppercase
  - Lowercase
  - Number
  - JSON
  - Percent
  - Async
- Custom pipes
  - Pure
  - Impure

# Forms

## Template Driven Forms

- Angular infers the Form Object from the DOM
- App logic resides inside the template

## Model Driven Forms

- Form is created programmatically and sync with the DOM
- App logic resides inside the component
- Use of FormControl, FormGroup, FormBuilder

---

Angular's DI system is controlled through **@NgModule**.

---

Services implement DI concepts in an Angular App.

---

Services are simple ES6 classes.

---

Services are registered with Angular App using providers.

---

Services are Singleton.

## DI & Services

# Services : Hierarchical Injector

---

|                 |   |
|-----------------|---|
| Root Module     | Same instance of service is available Application-wide                                |
| Root Component  | Same instance of service is available for all components (but not for other services) |
| Other Component | Same instance of service is available for the component and it's own child components |

---

# Services : Hierarchical Injector

---

Use `providedIn: 'root'` for services which should be available in whole application as singletons

---

Never use `providedIn: EagerlyImportedModule`, you don't need it and if there is some super exceptional use case then go with the providers: `[]` instead

---

Use `providedIn: LazyServiceModule` to prevent service injection in the eagerly imported part of the application or even better use providers: `[LazyService]` in the LazyModule.

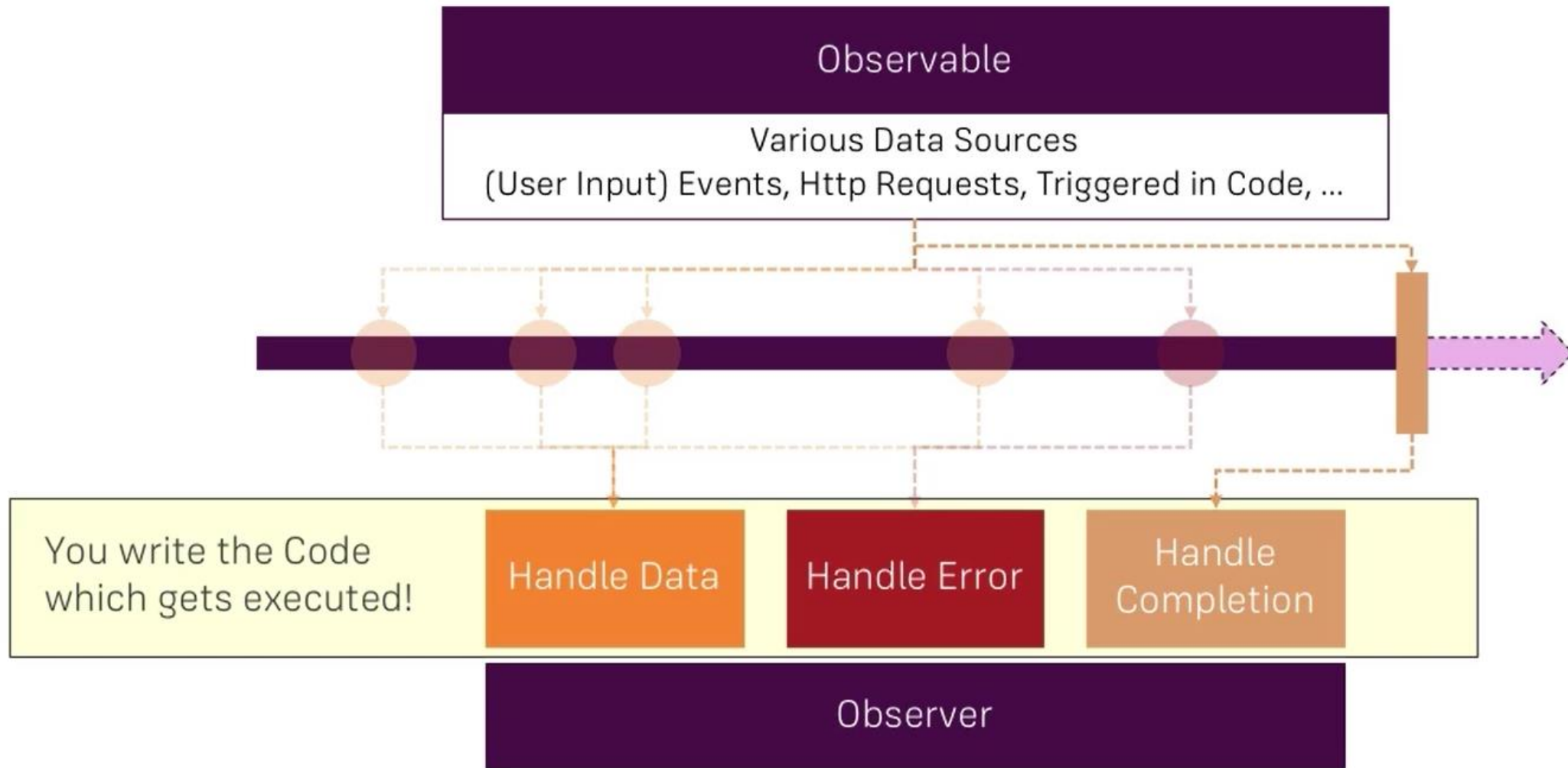
---

If we want to use `LazyServiceModule` then we have to import it in `LazyModule` to prevent circular dependency warning. `LazyModule` will then be lazy loaded using Angular Router for some route in a standard fashion.

---

Use providers: `[]` inside of `@Component` or `@Directive` to scope service only for the particular component subtree which will also lead to creation of multiple service instances (one service instance per one component usage)

# Observables : An Overview





# HttpClient

---

The HttpClient in @angular/common/http offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by browsers.

---

## Benefits of HttpClient:

---

Typed request and response objects

---

Request and response interception

---

Observable APIs

---

Streamlined error handling.

---

# HttpClient : Unlocking



Open the root  
AppModule

Import the  
HttpClientModule  
symbol from  
`@angular/common/http`

Add it to the  
`@NgModule.imports`  
array

Routing allows to:

- Maintain the state of the application
- Implement modular applications
- Implement the application based on the roles (certain roles have access to certain URLs)

5 steps routing:

- Checking the base href tag in index file
- Configuring routes with components
- Tell angular about routing app
- Setting up the routing links
- Provide space on template to load the component

# Routing

---

Programmatic navigation

---

Child routing

---

Routes with parameters

---

Route guard (Authentication)

---

Query Parameters

Routing  
(Cntd..)

# Modules

- A module is a mechanism to group components, directives, pipes and services that are related
- Module Types -
  - Root Module : one per application
  - Feature Module : depends on application features
- Modules can be instantiate lazily

# Debugging Angular Apps

- Prevent Bugs with TypeScript
- Using Debugger Statements to Stop JavaScript Execution
- Inspect Data with the JSON pipe
- Console Debugging
- Augury Chrome Plugin
- Debugging RxJS Observables using 'tap' operator

# PWA : Progressive Web Apps

---

Service worker is a script that runs in the web browser and manages caching for an application.

---

Can completely satisfy the loading of the application, without the need for the network.

---

Deliver a user experience with the reliability and performance on par with natively-installed code.

---

Reducing dependency on the network can significantly improve the user experience.

---

**ng add @angular/pwa**



A predictable state container for JavaScript apps.



# Should I Use Redux?

---

Some suggestions:      You have reasonable amounts of data changing over time

---

You need a single source of truth for your state

---

You find that keeping all your state in a top-level component is no longer sufficient

---

# Redux

---

Redux is a predictable state container for JavaScript apps.

---

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

---

It provides a great developer experience, such as live code editing combined with a time traveling debugger.

---

```
npm install --save redux
```

```
npm install --save @ngrx/store
```

# Redux : Three Principles

## Single source of truth

- The *state* of your whole application is stored in an object tree within a single *store*.

## State is read-only

- The only way to change the state is to emit an *action*, an object describing what happened.

## Changes are made with pure functions

- To specify how the state tree is transformed by actions, you write pure *reducers*.

# Actions

- *Actions* are payloads of information that send data from your application to your *store*.
- They are the only source of information for the store.
- You send them to the store using `store.dispatch()`.

```
{  
  type: ADD_TODO,  
  text: 'Build my first Redux app'  
}
```

# Reducers

- The *reducer* is a pure function that takes the previous *state* and an *action*, and returns the next state.
- *Actions* only describe *what happened*, but don't describe how the application's state changes.
- **Reducers** specify how the application's state changes in response to *actions* sent to the *store*.

`(previousState, action) => newState`

# Reducers : Don'ts

## Things you should **never** do inside a reducer:

- Mutate its arguments.
- Perform side effects like API calls and routing transitions.
- Call non-pure functions, e.g. `Date.now()` or `Math.random()`.

# Store

The Store is the single *object* that has the following responsibilities:

- Holds application state.
- Allows access to state.
- Allows state to be updated via `dispatch(action)`.
- Registers listeners via `subscribe(listener)`.
- Handles unregistering of listeners via the function returned by `subscribe(listener)`.

# Creating Store with Root Reducer

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

@NgModule({
    ...
    imports :      [StoreModule.forRoot({prop : reducer})]
})
export class AppModule{}
```



# Data Flow

Redux architecture revolves around a **strict unidirectional data flow**.

The data lifecycle in any Redux app follows these 4 steps:

- You call `store.dispatch(action)`.
- The Redux store calls the reducer function you gave it.
- The root reducer may combine the output of multiple reducers into a single state tree.
- The Redux store saves the complete state tree returned by the root reducer.

# Angular : Creating Libraries

---

A library typically includes *reusable code* that defines components, services, and other Angular artifacts

---

Libraries extend existing Angular functionality.

---

Libraries by default, built with the AoT compiler.

---

Export functionality for library is maintained in *public-api.ts* file

# Creating Angular Library : Steps

## Create Workspace

- `ng new my-workspace --createApplication=false`

## Create Lib

- `ng generate library <libname> --prefix=incedo-lib`

## Make modification in library and PUBLIC API

- `ng build <libname>`

## Install lib in project/application

- `npm install <libname>`

## Import the Lib Module in App Module

## Use the lib functionalities

# Securing Angular Apps

- Best Practices :
  - Keep current with the latest Angular library releases
  - Don't modify your copy of Angular.
  - Avoid Angular APIs marked in the documentation as “*Security Risk.*”
- Preventing cross-site scripting (XSS)
  - Angular treats all values as untrusted by default.
  - Angular sanitizes and escapes untrusted values.
    - Interpolated content is always escaped
    - Angular recognizes the *binded value* as unsafe and automatically sanitizes it
  - Never generate template source code by concatenating user input and templates, instead use the offline template compiler (template injection)

# Securing Angular Apps

---

**Trusting Safe Values :** To mark a value as trusted, inject *DomSanitizer* and call one of the following methods -

---

bypassSecurityTrustHtml

---

bypassSecurityTrustScript

---

bypassSecurityTrustStyle

---

bypassSecurityTrustUrl

---

bypassSecurityTrustResourceUrl

---

**HTTP-level vulnerabilities :** Angular has built-in support to help prevent two common HTTP vulnerabilities -

Cross-Site Request Forgery (CSRF or XSRF)

---

Cross-Site Script Inclusion (XSSI) / JSON vulnerability

---

# Optimizing Angular App Performance

---

Using onPush change detection strategy

---

Using trackBy function

---

Avoid computing values in templates

---

Using lazy loading

---

Disable change detection (if required)

# Deployment Steps :

## Build the App

- Consider AOT Compilation

## Set the correct <base>

- Domain name with App Directory

## Server should return index file

- Server would not know the routes

# References

## Books

- Rangle's Angular Training Book
- Ngbook

## Web

- <http://angular.io>
- <http://rangle.io>
- <http://reactivex.io>
- <http://learnrxjs.io>
- <https://redux.js.org>