**Angular**

* Application design framework.
* Development platform for efficient and sophisticated SPA.
  + A component based framework for building scalable web applications.
  + A suite of developer tools to help you develop, build, test and update code.
  + A collection of well integrated Libraries with features like Routing, Forms Management, Data binding, Client-server Communication, Dependency Injection, Directives, Declarative Templates, End-End Tooling.

**COMPONENTS :**

* The building blocks of an application.
* A component includes a TypeScript class with a @[Component](https://angular.io/api/core/Component)() decorator, an HTML template, and styles.
* @Component specifies
* A CSS selector that defines how the component is used in a template.
* HTML elements in your template that match this selector become instances of the component.
* An HTML template that instructs Angular how to render the component.
* An optional set of CSS styles that define the appearance of the template's HTML elements.

Templates :

Every component has an HTML template that declares how component renders.

Angular's declarative templates let you cleanly separate your application's logic from its presentation.

<button (click)="onEditClick()">Make text editable!</button>

<div \*ngIf="canEdit; else noEdit">

<p>You can edit the following paragraph.</p>

</div>

<ng-template #noEdit>

<p>The following paragraph is read only. Try clicking the button!</p>

</ng-template>

<p [contentEditable]="canEdit">{{ message }}</p>

### **Dependency injection :**

State Management in Angular

**State :**

“State” term includes both the state of UI and the state of variables in our code. So any change in your application changes the state.

Each component has its own state and a component has no idea about the other components’ states unless we make enable the data flow between components. We use **@Input** and **@Output decorators**to pass information between parent and child components.

It is easy to pass information between components in simple apps. But it gets complicated and painful when you have complex app architecture like the figure below.

Diagram

Description automatically generated

If you use @Input/@Output it takes 4 hops to get **Component 6**from**Component 2**and the you should involve the other 3 components in this process.

**State Management :**

Redux is an elegant solution architecture which simplifies the state management. There are three core principles in Redux architecture.

**Single Store:** state of the application should be stored in a single “store”. Store is the only responsible in providing data to components. Data flows between component and Store instead of component-to-component.

**Read-only state:** state should be read-only or immutable. To change the state, new action should be emitted.

**Pure function reducers:**Reducers take the current state and an action and return next state. Reducers should not take any other parameters outside the function.

Diagram

Description automatically generated

* Data flow is unidirectional.

Diagram

Description automatically generated

## **@ngrx/store**

@ngrx/store is a state management extension for Angular applications inspired by Redux. It imports all core concepts from Redux. From now on, we will use @ngrx/store terms instead of Redux.

## **Store**

Store is the core structure where actions, reducers and state are located. The store receives actions as inputs, and transmit actions to reducers. Reducers produce a new state depending on the action and emit the new state. The store holds the new state until it changes via another action.

Diagram

Description automatically generated

## **Action**

An action consists of two parts, type and payload. Type is required to make reducer enable to distinguish the actions. Therefore type must be unique for each action.

Diagram

Description automatically generated

## **Reducer**

The reducer is the core element in the store and it is the responsible for changing the state. A reducer takes action and current state as input parameters and returns a new state according to the function inside it.

(The state is immutable in Redux architecture, so the reducer does not mutate the state but produces a new one.)

Diagram

Description automatically generated

When a reducer is called upon a new action, it checks the type of the action and return new state according to action type. As stated previously, reducers must be pure functions.