Angular

A framework for building client applications in HTML, CSS and Javascript/Typescript

Benefits of framework

- Gives our application a clean structure
- Includes a lot of re-usable code
- Makes our application more testable
- Improved Speed and Performance
- CLI support from component generation to deployment
- Makes development process faster and developer life easier

Benefits of Angular

Maintained by Google: Angular is developed and maintained by Google. As it is backed by a trusted company, trust is well established within the community. Developers believe in the framework that it will be stable, maintained and issues will be resolved, of course with the help of community contributions.

Use of TypeScript: Angular is built with TypeScript as the primary programming language. It helps developers to keep their code clean and understandable. Bugs are easier to spot and eliminate with the ability to see common errors as you type with code intellisense. This makes it quicker when it comes to debugging and also easier to maintain a large codebase

Support for i18n: Internationalization is the process of making sure that an application is designed and prepared to be used in regions with different languages. Localization is the process of translating your internationalized app into specific languages for particular locales. Angular can take care of most things when it comes to multiple languages. Dates, numbers, times, and other things are easily taken care of based on the locale. On top of that, the Angular CLI allows us to install the @angular/localize package and generate most of the boilerplate code necessary.

Functionality out of the box: The default setup of Angular gives you everything you need. Angular's preconfigured environment not only helps with development, but also facilitates testing. You won't need to use any third-party libraries to create basic functionality for your app. All you need is the official library, which is provided by the Angular team. This means that you can expect better cybersecurity and higher code quality.

Two-way Data binding: Two-way data binding in Angular will help users to exchange data from the component to view and from view to the component. It will help users to establish communication bi-directionally. Two-way data binding can be achieved using a ngModel directive in Angular. This ensures that the model and the view are always kept in sync without any extra effort.

Benefits of Angular

Modular development structure: A module is delegated a certain responsibility and it can comprise of components, pipes, directives or more modules. This allows for easy organization of app functionality and the creation of reusable chunks of code which can vastly reduce development time and cost.

Modules also make it possible to efficiently divide development work across teams while ensuring that code stays clean and organized as well as allowing the application to be scaled seamlessly.

Support for lazy loading: Breaking down an application into modules not only allows to easily manage an application as it grows, but also offers huge performance advantages. Instead of loading all the application code at once in the browser which may cause more loading time and a slower initial render of the web page, it loads only what is necessary. This is called lazy loading.

Only the modules that are required are loaded initially and more modules or rather chunks of code are loaded only when it is required. This offers seamless performance and smaller loading times.

Angular Material: Angular Material is a collection of ready-to-use, well-tested UI components and modules that follow Google's Material Design principles. It contains a range of UI components, such as navigation patterns, form controls, buttons, and indicators. Components are adapted to suit various browsers, well-documented, and written based on the latest guidelines.

Drawbacks of Angular

Steeper learning curve: While Angular is great, it may not be the case for a complete beginner. Even if you have experience with HTML, CSS and JS, you may feel a little bit uncomfortable with the steep learning curve that it has.

Limited SEO Capabilities: Angular is just great for building powerful single-page web applications. However, as with all single-page web applications, there is a disadvantage when it comes to search engine optimization. However, there are techniques and packages to make this work, but it is an extra effort in terms of development.

Decline in popularity: With the advent of newer frameworks like VueJS and ReactJS, Angular has seen a downfall in its popularity. While the job market in Angular continues to expand, it has been observed that the popularity of the framework has been declining.

Lot of boilerplate code: If you are building a simple app, there is a lot of boilerplate code that the application requires. If you decide to start developing an Angular app on your system, you need to install a bunch of things and the most simple “Hello World!” application will have a lot of code that you do not care about.

When to USE Angular

- To build Progressive Web Apps(PWA)
- To build Enterprise web apps like eCommerce applications
- To build data intensive or dynamic web apps like Forbes, Xbox
- To build Video streaming apps like YouTubeTV, Youtube PlayStation three
- To build Real-time Data Application like weather.com
- To build Hybrid app(Web and Mobile app with same code)

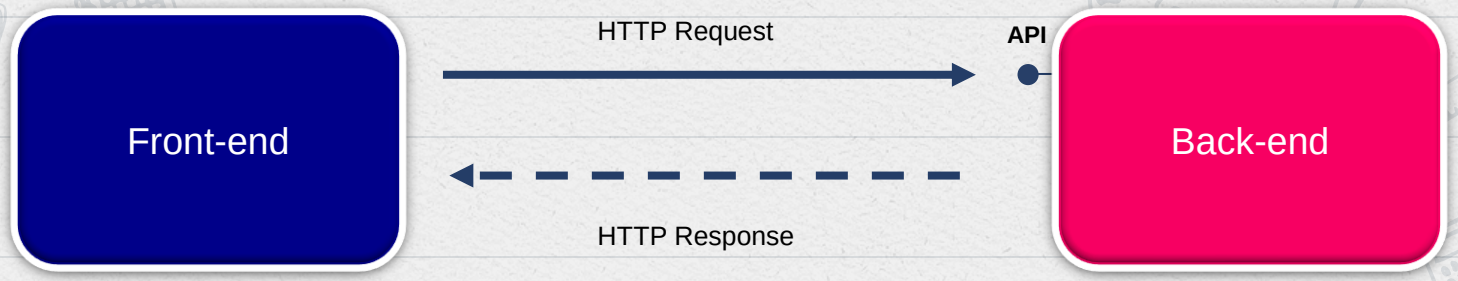
When NOT to use Angular

- Websites with Static Content. Example: Landing pages, informative websites, event pages, etc.
- For small teams & limited resources as Angular demands skillful available resources for quick problem-solving in large scale projects.
- To build computation intensive apps like Game Apps & Heavy Analytical Apps

Architecture

User Interface (UI)

Data & Processing



HTML Templates

Presentation logic

Databases + APIs

Business logic

*Application Programming Interface(API): Endpoints that are accessible via the HTTP protocol

Angular Fundamentals

- Components
- Templates
- Directives
- Services
- Dependency Injection

Components

Components are the fundamental building block of Angular applications. We can build larger Components from smaller ones.

An Angular application is therefore just a tree of such Components, when each Component renders, it recursively renders its child Components.

At the root of that tree is the top-level Component, the *root* Component.

When we *bootstrap* an Angular application, we are telling the browser to render that top-level *root* Component which renders its child Components and so on.

Component should be smaller, maintainable, reusable

3 STEPS

Since it is not a native HTML element, Angular doesn't know

CLI

ng generate component *<component-name>*
ng g c *<component-name>*

Services

Why separate service file?

- To make component unit testable by faking the real HTTP endpoint
- It can be re-used in other components
- To separate and decouple the logic other than presentation logic in the component

Dependency Injection

Injecting or providing dependency of the class in the constructor

Singleton

Angular creates single instance of the service class and it will pass the same instance to all the components

3 STEPS

CLI

ng generate service *<service-name>*
ng g s *<service-name>*

@Injectable decorator

It is not compulsory to add. Add if the injected class has some dependency which needs to be injected.

Angular should be able to inject dependencies of this class into its constructor.

Bindings and Handling Events

- Property binding
- Class binding
- Style binding
- Event binding
- Event filtering
- Two way binding

Property Binding

```
<h1>{{title}}</h1>
<h1 [textContent]="title"></h1>
```

```
<img src="{{imageURL}}" />
<img [src]="imageURL" />
```

Add or remove class dynamically

Add or remove CSS style dynamically

Class Binding

We can add multiple class/style binding to same element and all will be evaluated

Style Binding

```
[class.<class-name>]='expression'
```

```
[style.<propertyName>]='value'
```

```
[style.<propertyName>.<unit>]='value'
```

ngClass directive

We can add multiple ngClass/ngStyle to same element but later will get evaluated

ngStyle directive

```
[ngClass]="{'<class-name>': expression}"
```

```
[ngStyle]="{'<property-name>': value}"
```

```
[ngStyle]="{'<property-name>.<unit>': value}"
```

Event Binding

```
(<eventName>)="<function(arguments)>"
```

Event Filtering

```
(keyup.enter)="onEnter()"
```

Building reusable components

Component Interaction:

Passing data to child component using @Input

Giving alias name to component’s Input property

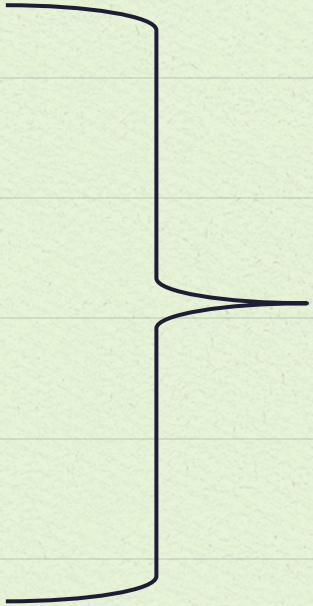
Raising event to parent component using @Output and EventEmitter

Giving alias name to component’s Output event

Passing data from child to parent using EventEmitter’s emit event

Pipes

- ✓ CurrencyPipe
- ✓ DatePipe
- ✓ DecimalPipe
- ✓ JsonPipe
- ✓ LowerCasePipe
- ✓ UpperCasePipe
- ✓ PercentPipe
- ✓ SlicePipe
- ✓ AsyncPipe

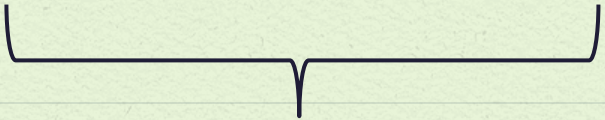


Build in Pipes

Directives

Build in Directives

ngFor | ngIf | ngSwitchCase | ngSwitch | ngStyle | ngClass | ngNonBindable | hidden



Structural directive - prefixed with *

Angular Forms

Types of Angular Forms

Overview on Form structure

Template Driven Form:

✓ Building template driven form using `ngForm`, `ngModel`

✓ Adding validations and displaying specific validation messages

✓ Grouping controls under `ngModelGroup`

✓ Styling invalid input fields

✓ Working with checkbox, radio button and dropdown

Reactive Form:

✓ Creating controls programmatically

✓ Adding validations and displaying specific validation messages

✓ Nested `FormGroup`

✓ Implementing custom validation

✓ Working with `FormArray` and `FormBuilder`

✓ Working with Async validation

✓ Setting default values

✓ Form methods – setValue, patchValue, reset

Types of Form

Template driven

Created using directive

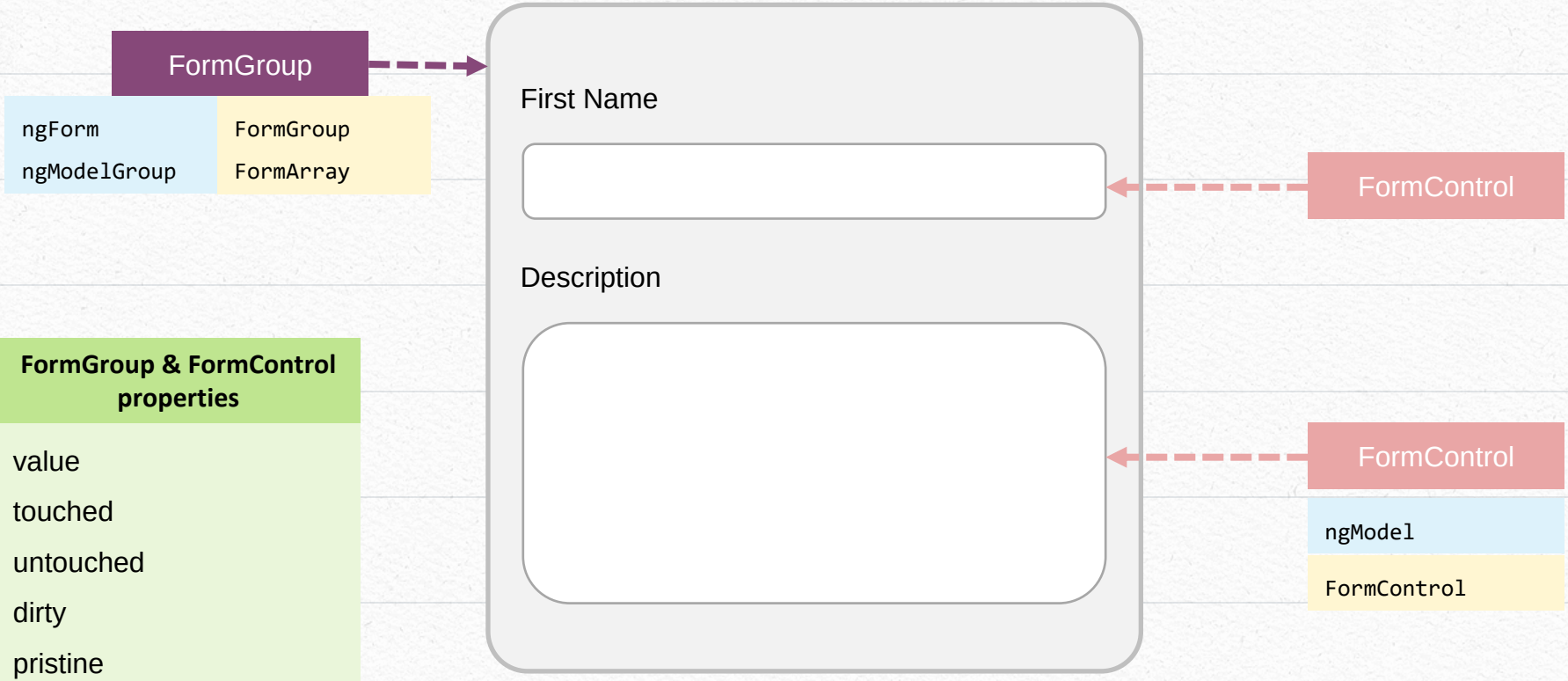
- ✓ Good for simple forms
- ✓ Simple validation
- ✓ Easier to create
- ✓ Less code

Reactive/Model driven

Created using code

- ✓ More control over validation logic
- ✓ Good for complex forms
- ✓ Unit testable

Form structure



FormGroup & FormControl properties

- value
- touched
- untouched
- dirty
- pristine
- valid
- errors

Template driven control	Reactive Form control

Asynchronous programming

Understanding asynchronous programming

Promise vs Observable

Perform CRUD operation with API using HTTP

Error handling with API

Build reusable HTTP data service

Promise vs Observable

Promise	Observable
Eager - execute immediately after creation	**Lazy** - not executed until we subscribe to them using the subscribe() method
Always **Asynchronous**	Can be **Synchronous or Asynchronous**
Promise object may only deliver a **single value**	Observable instance may emit **multiple values or a stream of values**
Operators not available	Can apply RxJS operators
Are **not cancellable**	Subscriptions are **cancellable** using the unsubscribe() method, which stops the listener from receiving further values

Reference:

https://medium.com/javascript-everyday/javascript-theory-promise-vs-observable-d3087bc1239a

https://www.stackchief.com/blog/Observable%20vs%20Promise%20%7C%20What's%20the%20difference%3F

Error Handling

Unexpected

- Server is offline
- Network is down
- Unhandled exceptions due to bug in API

Expected

- **Not found errors** - 404 - can be due to data that we are updating/deleting may not available in server
- **Bad request errors** - 400 - can be due to invalid input/data to server

Routing and Navigation

Configure the Routes

Adding router-outlet and routerLink

Adding css class using routerLinkActive

Parameterized routes and getting the route parameter

Routes with multiple parameters

Query parameters

Programmatic Navigation

Lazy loading the modules

Configure the routes

```
const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {
    path: 'artist/:artistId',
    component: ArtistComponent,
    children: [
      {path: '', redirectTo: 'tracks'},
      {path: 'tracks', component: ArtistTrackListComponent},
      {path: 'albums', component: ArtistAlbumListComponent},
    ]
  },
  {path: '**', component: HomeComponent, //redirectTo: 'home'}
];
```

Add router outlet

```
<router-outlet></router-outlet>
```

This directive tells Angular where it should insert each of those components in the route

Add router link

```
<!-- This will reload the entire page so application wont behave like SPA-->
<a href="home"> Home </a>
<!-- Proper way is to use routerLink-->
<a routerLink="home"> Home </a>
```

catch-all/wildcard route

```
{path: '**', component: HomeComponent, //redirectTo: 'home'}
```

We can add a catch-all route by using the path **, if the URL doesn't match any of the other routes it will match this route.

Either we can render the PageNotFound component in the not found url or we can redirect to some page (changes url)

Add css class using routerLinkActive

```
<a routerLink="home" routerLinkActive="active"> Home </a>
<!-- Add one or more css class by separating in space -->
<a routerLink="home" routerLinkActive="active current"> Home </a>
```

Authentication and Authorization

Creating login page and AuthService

Protecting our routes using CanActivate guard

Redirecting the user to login page and navigating back to the accessed page on successful login

Adding authorization to the route

Miscellaneous

Change detection strategy – Things to consider

Angular lifecycle hooks

Configuring deployment environments

Configuring custom environment

```
ng serve --prod
ng serve --c=<environment>
```


Keep Learning