Angular

Angular is a TypeScript-based free and open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS. Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your applications.

Main Components

It has 5 main components:

- Components.
- Modules.
- Templates.
- Services.
- Metadata and decorators.

Components

Components are the basic UI building block. It's what I call a puzzle of an Angular App that forms at the end a tree of Angular components. Components always have a template, and only one component can be instantiated per element in a template.

Modules

Modules are logical boundaries in your application. An Angular app is divided into separate modules. The main job or goal of modules is to separate the app functionality or logic.

```
1  import { NgModule }      from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { AppComponent }  from './app.component';
4
5  @NgModule ({
6     imports:      [ BrowserModule ],
7     declarations: [ AppComponent ],
8     bootstrap:    [ AppComponent ],
9     providers: []
10 })
11 export class AppModule { }
```

Templates

A template is an HTML view where you display data by binding controls to properties of an Angular component. A template is defined by using the template property or by creating an HTML separate file and set after then the template URL property to this file's path.

```
import { Component } from '@angular/core';

@Component ({
   selector: 'my-app',
   template: `
      <div>
         <h1>{{title}}</h1>
         <div>Learn Angular</div>
      </div>
   `
})

export class AppComponent {
   title: string = 'Hello World';
}
```

Services

A service is typically a class with a well-defined purpose. It should do something specific and do it well. It's commonly used to fetch data from a server asynchronously. Let's say you want to create an Angular app that will display all GitHub repositories. This means that you'll need a service dedicated to etching repositories for you and then injected into the component that will use this service.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({ // The Injectable decorator is required for dependency injection to work
  // providedIn option registers the service with a specific NgModule
  providedIn: 'root',  // This declares the service with the root app (AppModule)
})
export class RepoService{
  constructor(private http: HttpClient){
  }

  fetchAll(){
    return this.http.get('https://api.github.com/repositories');
  }
}
```

Metadata And Decorators

Metadata is used to decorate a class so that it can configure the expected behavior of the class. TypeScript Decorator defines it. It is a special kind of declaration that can be attached to a class decoration that can be attached to a class declaration, method, accessor, property, or params.

```
import { NgModule, Component } from '@angular/core';

@Component({
  selector: 'my-component',
  template: '<div>Class decorator</div>',
})
export class MyComponent {
  constructor() {
    console.log('Hey I am a component!');
  }
}

@NgModule({
  imports: [],
  declarations: [],
})
export class MyModule {
  constructor() {
    console.log('Hey I am a module!');
  }
}
```

File Structure

Workspace configuration files

WORKSPACE CONFIG FILES	PURPOSE
.editorconfig	Configuration for code editors.
.gitignore	Specifies intentionally untracked files that git should ignore.
README.md	Introductory documentation for the root application.
angular.json	CLI configuration defaults for all projects in the workspace, including configuration options for build, serve, and test tools that the CLI uses, such as Karma and Protractor.

package.json	Configures npm package dependencies that are available to all projects in the workspace.
package-lock.json	Provides version information for all packages installed into node_modules by the npm client.
src/	Source files for the root-level application project.
node_modules/	Provides npm packages to the entire workspace. Workspace-wide node_modules dependencies are visible to all projects.
tsconfig.json	The base TypeScript configuration for projects in the workspace. All other configuration files inherit from this base file.

Application source files

APP SUPPORT FILES	PURPOSE
app/	Contains the component files in which your application logic and data are defined.
assets/	Contains image and other asset files to be copied as-is when you build your application.
environments/	Contains build configuration options for particular target environments. By default, there is an unnamed standard development environment and a production ("prod") environment. You can define additional target environment configurations.
favicon.ico	An icon to use for this application in the bookmark bar.
index.html	The main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you typically don't need to add any <script> or <link> tags here manually.
main.ts	The main entry point for your application. Compiles the application with the JIT compiler and bootstraps the application's root module (AppModule) to run in the

	browser. You can also use the AOT compiler without changing any code by appending the --aot flag to the CLI build and serve commands.
polyfills.ts	Provides polyfill scripts for browser support.
styles.sass	Lists CSS files that supply styles for a project. The extension reflects the style preprocessor you have configured for the project.
test.ts	The main entry point for your unit tests, with some Angular-specific configuration. You don't typically need to edit this file.

Inside the src/ folder, the app/ folder contains your project's logic and data. Angular components, templates, and styles go here.

SRC/APP/FILES	PURPOSE
app/app.component.ts	Defines the logic for the application's root component, named AppComponent. The view associated with this root component becomes the root of the view hierarchy as you add components and services to your application.
app/app.component.html	Defines the HTML template associated with the root AppComponent.
app/app.component.css	Defines the base CSS stylesheet for the root AppComponent.
app/app.component.spec.ts	Defines a unit test for the root AppComponent.
app/app.module.ts	Defines the root module, named AppModule, that tells Angular how to assemble the application. Initially declares only the AppComponent. As you add more components to the app, they must be declared here.

Application configuration files

APPLICATION-SPECIFIC CONFIG FILES	PURPOSE
.browserslistrc	Configures sharing of target browsers and Node.js versions among various front-end tools.
karma.conf.js	Application-specific Karma configuration.
tsconfig.app.json	Application-specific TypeScript configuration, including TypeScript and Angular template compiler options.
tsconfig.spec.json	TypeScript configuration for the application tests.

Library project files

LIBRARY SOURCE FILES	PURPOSE
src/lib	Contains your library project's logic and data. Like an application project, a library project can contain components, services, modules, directives, and pipes.
src/test.ts	The main entry point for your unit tests, with some library-specific configuration. You don't typically need to edit this file.
src/public-api.ts	Specifies all files that are exported from your library.
karma.conf.js	Library-specific Karma configuration.
ng-package.json	Configuration file used by ng-packager for building your library.
package.json	Configures npm package dependencies that are required for this library.
tsconfig.lib.json	Library-specific TypeScript configuration, including TypeScript and Angular template compiler options.
tsconfig.lib.prod.json	Library-specific TypeScript configuration that is used when building the library in production mode.

tsconfig.spec.json	TypeScript configuration for the library tests. See TypeScript Configuration.

Angular CLI And Build

The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications directly from a command shell.

Install the CLI using the npm package manager:

```
npm install -g @angular/cli
```

Basic workflow

For online help regarding angular:

```
ng help
ng generate --help
```

To create, build, and serve a new, basic Angular project on a development server, go to the parent directory of your new workspace use the following commands:

```
ng new my-first-project
cd my-first-project
ng serve
```

For build:

```
ng build <project> [options]
```

Angular Basic Concepts

Modules

Angular has some prebuilt modules to help developers, Angular calls these NgModules. NgModules are always marked with the @NgModule annotation in Angular. Some common modules are the FormsModule, RouterModule, HttpClientModule, and the Angular material design module. The idea of modules comes from the single responsibility principle. While a module can do many things, at a higher level its focus is on one thing such as forms or routing. This prevents application bloat and orders the application into concise blocks of functionality. Organizing code into modules also allows lazy loading, meaning the modules load as needed rather than all at once on application initialization. Lazy loading can help improve the load times and speed of your application, a crucial metric.

Root Module

Every Angular app will have a root module, named the AppModule, that lives in an app.module.ts file by default. This root module allows the application to bootstrap, or initialize and load, itself.

AppModule.

```
/* JavaScript imports */
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

/* the AppModule class with the @NgModule decorator */
@NgModule({
  declarations: [AppComponent],
  imports: [],
  exports: [],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Components

An Angular component is a Typescript class that interacts with a view. A view consists of a component and a template. A template is HTML combined with the **data-binding** syntax of Angular. A component has fields and methods that support a view. The view then gets updated using these properties and methods. Angular creates and destroys all components as a user moves through the application.

```
// Metadata
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})

// Actual component class
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>
```

Data Binding

Data binding is a great quality of life improvement in Angular. It allows data to flow between your component and template. Without data binding, the Angular developer would have to write custom code to push data between the template and component. Data binding is not limited to communication between a parent template and component it can also pass data from a parent to a child component. Angular has four types of data binding, in my experience, the mustache syntax is the most used but it also depends on developer preference.

Services

An Angular service should be reusable and it should follow the single responsibility principle of doing one thing and doing that one thing well. While a component handles the view a service handles everything not related to the view. Things like repeatable business logic, fetching data, or validating input that may be essential to the app but doesn't have anything to do with the visuals belong in a service. By defining this logic in a service, you make it available for injection anywhere in your application, giving you infinite reusability.