Monday: Building Blocks of Angular

Angular CLI

Angular provides a quick way of setting up our project by using something called **Angular CLI**. CLI stands for Command Line Interface, which is a set of tools that we can access using the terminal to perform a variety of common tasks. The angular CLI is mainly used to create an app with common settings quickly using simple commands.

The CLI has several commands that we can use to create, serve, build and generate components. All the commands start with the **ng** prefix

Let's look at the most common commands that we'll be using in this course

- **ng new NAME -** This command allows us to create a new angular application. It creates all the files we are going to need for a basic angular project. Once you type ng new, you'll replace NAME with the name of your project.
- **ng serve -** As we develop our application, we'll want to see what our application looks like, hence we use this command to run our application in **development mode**. It uses **WebPack** which is a tool that processes the code in our application and runs a temporary live server that updates every time we make changes to our application.
- **ng build** - This command processes our application files and gets them ready for deployment. Once we are done developing our application, the next thing we'll want to do is to publish our application to a server. This process is called **deployment**.

The main building blocks of Angular are:

- Angular CLI
- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection

Before we begin, let's look at an analogy that will help explain some of these building blocks in a more non technical way.

*Consider your **angular application** as a building. A building can have N number of houses. Each house can be considered as a **module**. A house can have an N number of rooms which corresponds to **components**.*

*Each house (**module**) will have rooms (**components**). The building an have lifts (**services**) to facilitate movement in and out of the houses.*

Let's briefly look at what each of these building blocks entails. We might not go into depth here because each of these will be explained exhaustively in the coming lessons.

Modules

Angular applications follow a modular structure. An angular app will have one or more modules, each dedicated to a single purpose. Typically, a module would contain all the code needed for one piece of functionality in your application. *For example, lest's say that we're designing an dashboard for a website, we would have one root module and possibly a module for our article section and a module for our user section*.

Modules in Angular are known as **NgModules (https://angular.io/guide/ngmodules#ngmodules)** and they are denoted by a **@NgModule** decorator.

Note: NgModule is typically a class that has been decorated with a @NgModule decorator. The NgModule simply tells Angular how to compile and run the module code.

NgModules can contain components, services and code files whose scope is defined by the NgModule. They can also import functionality that is exported from another NgModule as well as export some the functionality defined in it for use by other NgModules.

Every Angular application must have a root module, conventionally named as **AppModule** which resides in a file named **app.module.ts**. This root module acts as the application's main entry point. What this means is that all the other modules must be imported to the root module for them to work. Remember, without the root module, your application will not work.

In a small application, the root module is all you need with a few components. As your application grows, you may need to add more modules to represent all your related functionality and then import them into the root module.

@NgModule Decorator

Decorators are functions that modify JavaScript classes. Decorators are basically used for attaching metadata to classes so that, it knows the configuration of those classes and how they should work.

Hence *NgModule* is a decorator function that takes metadata objects whose properties describe the module.

The properties are:

- ***declarations:*** The classes that are related to views and belong to this module. There are three classes of Angular that can contain view: **components**, **directives**, and **pipes**. We will talk about them in a while.
- ***exports:*** The classes that should be accessible to the components of other modules.
- ***imports:*** Other existing Modules whose classes are needed by the component of this module.
- ***providers:*** Services present in one of the modules which are to be used in the other modules or components. Once a service is included in the providers it becomes accessible in all parts of that application
- ***bootstrap:*** The *root component* which is the main view of the application. This root module only has this property and it indicates the component that is to be bootstrapped.

Let us take a look at how the root module (i.e. ***src/app/app.module.ts***) looks like initially:

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

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Notice that there's no export property inside the @NgModule decorator. This is because as the root module its main job is to import other modules & components to use them. Remember we said it's the main entry point to the application.

We are importing a module called the BrowserModule because it's required by default for any web-based angular application.

Components

Angular puts everything in the user interface into self-contained components. A *component* controls one or more sections on the screen called a *view*. For example, if you are building a Goals application, you can have components like App Component (*the root Component)*, Goals Component(which will handle the Goals list), Goals Detail Component(which will handle individual goals details), Goal Form component (which handle rendering the form for adding a new goal)

Each component consist of the following:

- An HTML template that declares what renders on the page.
- A Typescript class that defines the logic of the component.
- CSS styles applied to the HTML template.

The easiest way to create a component is with Angular CLI by running the `ng generate component <component-name>`, where `<component-name>` is the name of your component.

This command creates the following:

- A folder named after the component
- A component file, `<component-name>.component.ts`
- A template file, `<component-name>.component.html`
- A CSS file, `<component-name>.component.css`
- A testing specification file, `<component-name>.component.spec.ts`

We'll look at what each of these files entails in the coming lessons

Metadata

Metadata tells Angular how to process a class. To tell Angular that App Component is a component, **metadata** is attached to the class. In TypeScript, you attach metadata by using a **decorator**. Below is an example of how metadata is attached to the App Component

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

Here, the @Component decorator identifies the class immediately below it as a component class. The @Component decorator takes the required configuration object which Angular needs to create and present the component and its view.

The most important configurations of @Component decorator are:

- *selector*: Selector tells Angular to create and insert an instance of this component where it finds <app-root> tag. For example, if an app's HTML contains <app-root></app-root>, then Angular inserts an instance of the AppComponent view between those tags.
- *templateUrl*: It contains the path of this component's HTML template.
- *styleUrls*: It is the component-specific style sheet.

The metadata in the @Component tells Angular where to get the major building blocks you specify for the component. *The template, metadata, and component together describe a view.* The architectural takeaway is that we must add metadata to our code so that Angular knows what to do.

Templates

We associate the component's view with its companion **template**. A template is nothing but a form of HTML tags that tells Angular about how to render the component. A template looks like regular HTML, except for a few differences. Here's an example of our Goals Component template

```
<h2>All Goalsh2>


<ul class="goals">



<li *ngFor="let goal of goals">
      <a routerLink="/detail/{{goal.id}}"></a>
    <span class="badge">{{goal.id}}</span> {{goal.name}}
</ul>
```

Directives

Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by **directives**.

There are two kinds of directives: ***structural*** and ***attribute*** directives.

Directive tends to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

Structural directives alter layout by adding, removing, and replacing elements in DOM.

This example template uses two built-in structural directives:

```
<li *ngFor="let goal of goals"></li>

<goal-detail *ngIf="selectedGoal"></goal-detail>
```

- ***ngFor** tells Angular to retrieve one <li> per goal in the goals. Basically, it's the equivalent of a For loop.
- ***ngIf** includes the GoalDetail component only if a selected goal exists. in other words, it works similar to an if statement

Attribute directives alter the appearance or behavior of an existing element. In templates, they look like regular HTML attributes. The ngModel directive, which implements two-way data binding, is an example of an attribute directive. ngModel modifies the behaviour of an existing element by setting its display value property and responding to change events.

```
<input [(ngModel)]="courses.name" placeholder="name"/>
```

Data Binding

Angular supports **data binding**, a mechanism for coordinating parts of a template with parts of a component. We add binding markup to the Angular supports **data binding**, a mechanism for coordinating parts of a template with parts of a component. For example:

```
<li> {{goal.name}}</li>

<goal-detail [goal]="selectedGoal"></goal-detail>


<li (click)="selectGoal(Goal)"></li>
```

- The {{goal.name}} *interpolation* displays the component's name property value within the <li> element.
- The [goal] *property binding* passes the value of selectedGoal from the GoalComponent to the goal property of the child GoalDetailComponent.
- The(click) *event binding* calls the component's selectGoal method when the user clicks a goals' name.

Two-way data binding is an important part as it combines property and event binding in a single notation, using the ngModel directive. Here's an example from the GoalDetailComponent template:

```
<input [(ngModel)]="courses.name" placeholder="name"/>
```

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding. Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.

Data binding plays an important role in communication between a template and its component. Data binding is also important for communication between parent and child components.

Services

Service is a broad category encompassing any value, function, or feature that your application needs. A service is typically a class with a well-defined purpose. Anything can be a service.

Services are fundamental to any Angular application. Components are the consumers of services.

Here's an example of a service class:

```
import { Injectable } from '@angular/core';
import { Goals } from '../goals';@Injectable({
  providedIn: 'root'
})
export class GoalService {  constructor() { }
  getGoals(){
    return Goals;
  }
  getGoal(id){
    for (let goal of Goals){
      if (goal.id == id){
        return goal;
      }
    }
  }
}
```

Services are everywhere. Component classes don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.

A component's job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic. A good component presents properties and methods for data binding. Angular does help us *follow* these principles by making it easy to factor our application logic into services and make those services available to components through *dependency injection*.

Dependency Injection

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need. Angular can tell which services a component needs by looking at the types of its constructor parameters.