ANGULAR BOOK

Prepared By:MBOTE-JOSEPH

Email: mbotejoseph001@gmail.com

Monday: Angular Objectives

We will be learning how to create basic Angular apps using Angular CLI over the next two weeks.

Angular can be written using Javascript, Typescript or Dart. In our case, we will write our Angular apps using Typescript, which you learned over the pre-course work.

Angular CLI is a command-line interface that helps us build Angular apps quickly.

Plan

This week we will start off by learning how to add content and interactivity to our Angular app. To start with, we will use templates and directives to display data dynamically. Then, we'll organise our data by adding models to our application and use event binding for adding interactivity.

Monday and Tuesday have been structured to have slightly more content so that you have enough basic knowledge which will help you practice on building an Angular application from scratch. Additionally, it also gives you a chance to engage with the content more and better understand the week's concepts.

By the end of the week, you will be able to

1. Create a fully functional Angular Application.
2. Create a custom directive to display data
3. Use event binding to add interactivity to our application

Next week we will explore the framework further by adding pages to our application using routers and so much more.

Independent Project Objectives

The week's independent project will be evaluated for the following objectives:

1. Use Angular CLI to create your Project.
2. Create a model for your Quote objects.
3. Create a custom directive that highlights the quote that has the highest vote.
4. Implement a form component that has input sections for the quote.
5. Initialize an initial value of 0 for both upvote and downvote for each quote
6. Add a custom pipe which displays the time passed since the quote was created.
7. Make a well-documented README that highlights:
    1. Author
    2. A brief description of the project
    3. Program set-up instructions
    4. A link to the deployed site of your project
    5. Copyright and License information
8. At least use bootstrap to style you project
9. High-Quality and portfolio-ready project

Monday: MVC Architecture with Angular

One of the popular ways of organising application is using the **MVC architecture**. Angular uses this architecture to organise its application. Let's look at what is MVC architecture in Angular.

Model-view-controller, or MVC as its popularly known, is a software design pattern for developing web applications. This design pattern is made up of three parts:

1. Model: Responsible for maintaining data
2. View:Responsible for displaying data to the user
3. Controller: Responsible for controlling the interaction between the Model and Views

The MVC design pattern separates the application's logic from the user interface.

The MVC concept can be represented graphically as follows:

Let's walk through what's happening in the diagram. Assume that you have an application that tracks your goals. Each time you delete or add a new goal, the controller updates the model by either deleting or adding a goal. In other words, the controller updates the model as per the user input.

The model then notifies the controller of the changes, which in turn updates the views to display the updated information. When you add your goals and the model is updated, the controller updates the view which displays the new goal on your site.

Let's look at each part separately and how they function:

Model

This is the application's data structure. It represents the actual data that an app deals with, and it responds to requests from views and instructions from controllers to update itself. Moreover, a model does not depend on either view or controller. In Angular applications, models are represented by objects. In the example of the goal application from earlier, our model would be:

```
class Goal{
  constructor(){}
}
```

Don't worry about mastering the code right now - we'll be creating applications soon enough. For now, focus on what each part represents, and how they relate to one another.

View

Views are what is presented to the user and how they interact with it. In other words, it's what the user actually sees on their screen, and can be made with HTML, CSS or Javascript. It displays the model data. In Angular, we can create a view in HTML by using double curly braces.

```
<h4 id="{{i}}" appStrikethrough> {{goal.name}} due on {{goal.completeDate|date|
uppercase}} </h4>
```

Controller

The controller is essentially the glue between the model and the view. It updates the view when the view changes and also adds event listeners to the view. Additionally, it updates the model when the user manipulates the view.

Since the controller links both the model and the views, it can be separated into two; view controllers and model controllers

View controllers are responsible for pulling together the model used by the view and handling the input from the user of the view. For example, if you wanted to delete a goal from your goal application, you would have an event listener in the HTML that listens for when the user clicks the delete button and then a function that actually deletes the clicked goal.

goal.component.html

```
<button class="btn btn-danger btn-sm" (click)='deleteGoal(i)'>Delete Goal</button>
```

In the example above the event listener is `(click)` and the function that deletes the goal is `deleteGoal(i)`. For now, do not worry about how to create the function, we'll explain that later as we look at angular in depth. The important thing to understand for now is how the event listeners work together with functions as view controllers.

On the other hand, model controllers contain the data to be displayed as well as the data to be collected as input in forms. It can also be functions that are invoked based on the user's activity such as clicking a button or making changes to the model data. For example, a function that adds a new goal to the model. Whenever you add a new goal through a form, there is a function that is responsible for adding that goal to your database. This function is a model controller because it interacts directly with your model.

Monday: Angular Apps

Angular Apps

Creating an Angular App

We have a basic understanding of typescript and ES6 from our precourse work so we can now dive into working with Angular. Let us create an app that helps us manage our goals. We create Angular apps using the Angular `CLI` - Command Line Interface. Let us open our terminals and use the CLI to generate a new Angular app. First, create a directory where you want to store your angular apps. Inside the directory you just created, run the command :

```
$ ng new Goals
```

This command prompts us on whether we want to use Angular routing. Type `y` and hit `enter`. Don't worry about what routing means, we'll look at it later on. The command also prompts us on which stylesheet format we want to use. We'll be using CSS, so select the CSS option and hit enter.

This command is taking some time to run. While it runs, it is creating boilerplate code for our angular app as well as creating the folder structure so we do not have to stress about setting up the files and folders. It also downloads and installs the necessary dependencies and packages needed for our app to run.

To see the end result of the command, navigate into the *Goals* folder and open it in your favourite editor. You should see a folder layout similar to the one shown below:

That is how we create angular apps using the angular CLI, we use the command `ng new <app-name>`.

Running an Angular App

Let us run our app on a local development server so we can interact with it. We use the command `ng serve` to run the app on the local development server. Let's navigate to the *Goals* directory in the terminal:

```
$ cd Goals/
```

While inside the *Goals* directory, let's run this command:

```
$ ng serve
```

This command responds by compiling all the files and then starts the local development server with the following output:

```
** Angular Live Development Server is listening on localhost:4200, open your
browser on http://localhost:4200/ **
```

To see the actual app, let's open the URL http://localhost:4200/ on the address bar of our web browser.

We are greeted with a welcome message from Angular with a couple of options we can use to learn and navigate Angular. That's amazing! We have successfully initiated our first angular app.

But wait a minute, we did not write any code and yet we have content displaying on our browser! What really happened?

Behind the scenes, the Angular CLI created boilerplate code for us in the files it generated. We'll notice that we have lots of files created but we'll focus on the ones in the *src/* folder. This is the folder in which our application's **Component**s live and we'll be working with the files inside this folder most of the time. We'll discuss Components in detail later on, so let it not frighten you. For general understanding, let's keep in mind that a component does something in an angular app.

Every new project is created with one **Component,** the *AppComponent*. Inside the *src/* folder, there's a subfolder named /*app* which hosts the *AppComponent.* Inside the /*app* folder, we have other files as well. Let's look at those that we'll work with:

- *app.component.css* - Contains the CSS styles specific to the *AppComponent.*
- *app.component.html* - Contains the HTML template code for the *AppComponent.*
- *app.component.ts* - Contains the typescript logic code for the *AppComponent.* This is the file that we will use to write code for the functionality of the *AppComponent.*
- *app.module.ts* - This file contains the high-level configurations that relate to the angular app in general. We'll explore it later on to understand what it entails.

In Angular, each component is identified with a unique selector. Open the *app.component.ts* file in the *src/app* folder. At the top, we have `@Component` which is a **decorator** function whose purpose is to declare metadata for the *AppComponent.* One of the metadata declared here is `selector` and its value is set as `app-root`.

When we navigate to *src/index.html*, we notice that we have the `<app-root></app-root>` tag in the body of the HTML template. The selector is used to render the *AppComponent* in the *index.html* template file. Anything in the *AppComponent* is now rendered on the app since it's selector is used in the *index.html* file which is the overall HTML file for the app. If you check the HTML file of the *AppComponent* in *src/app/app.component.html* , you'll notice whatever you're seeing on your browser currently is in this file. Different components will, therefore, have different selectors to uniquely identify them. As we create more components, we'll notice that these selectors are important because they allow us to nest components inside each other and render them without much of a do.

Updating our Components

We'll be building a goal tracking application over this week to help us understand concepts in Angular. Let's get our hands dirty and tweak the code to see what happens in our app. Update the *AppComponent* with the following code.

src/app/app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  goal = 'Watch Finding Nemo'
}
```

We have created a property `goal` and given it a value `'Watch Finding Nemo'.`

Now, let's change the HTML template code of the *AppComponent.*

src/app/app.component.html

```
<div>
  <h1>My Goals</h1>
  <p>My goal for today is {{ goal }}</p>
</div>
```

We have deleted the initial content in the file and created a div and inside it, created a <h1> tag and a <p> tag. Inside the <p> tag, we have used double curly braces `{{}}` and put the `goal` property we created in the *AppComponent* class. By doing this, we have used angular's **interpolation binding** syntax to put the value of the goal property inside the <p> tag. Let's serve our application using the *ng serve* command and go to our browser to see the changes we have made.

Monday: Angular File Review

Angular File Review

We can now examine the files inside our goals app. A good starting point is the `README` file where we can see all the Angular commands we can use to work with our app.

We will mostly work inside our *src/* folder:

```
src
|
|---app/
|       | ---app.component.css
|       | ---app.component.html
|       | ---app.component.spec.ts
|       | ---app.component.ts
|       | ---app.module.ts
|---assets/
|       | ---.gitkeep
|---environments/
|       | ---environment.prod.ts
|       | ---environment.ts
|---favicon.ico
|---index.html
|---main.ts
|---polyfills.ts
|---styles.css
|---test.ts
|---tsconfig.app.json
|    tsconfig.spec.json
```

These files and folders are what are found in the `src` folder where our app will live. Angular documentation defines them this way:

app/app.component.{ts,html,css,spec.ts}

Defines the AppComponent along with an HTML template, CSS stylesheet, and a unit test. It is the root component of what will become a tree of nested components as the application evolves.

app/app.module.ts

Defines AppModule, the root module that tells Angular how to assemble the application. Right now it declares only the AppComponent. Soon there will be more components to declare.

assets/*

A folder where you can put images and anything else to be copied wholesale when you build your application.

environments/*

This folder contains one file for each of your destination environments, each exporting simple configuration variables to use in your application. The files are replaced on-the-fly when you build your app. You might use a different API endpoint for development than you do for production or maybe different analytics tokens. You might even use some mock services. Either way, the CLI has you covered.

favicon.ico

Every site wants to look good on the bookmark bar. Get started with your very own Angular icon.

index.html

The main HTML page that is served when someone visits your site. Most of the time you'll never need to edit it. The CLI automatically adds all js and css files when building your app so you never need to add any <script> or <link> tags here manually.

main.ts

The main entry point for your app. Compiles the application with the Just-in-Time (JIT) compiler and bootstraps the application's root module (AppModule) to run in the browser. You can also use the Ahead-of-Time (AOT) compiler without changing any code by appending the–aot flag to the ng build and ng serve commands.

polyfills.ts

Different browsers have different levels of support of the web standards. Polyfills help normalize those differences. You should be pretty safe with core-js and zone.js, but be sure to check out the Browser Support guide for more information.

styles.css

Your global styles go here. Most of the time you'll want to have local styles in your components for easier maintenance, but styles that affect all of your app need to be in a central place.

test.ts

This is the main entry point for your unit tests. It has some custom configuration that might be unfamiliar, but it's not something you'll need to edit.

tsconfig.{app|spec}.json

TypeScript compiler configuration for the Angular app (tsconfig.app.json) and for the unit tests (tsconfig.spec.json).

The root folder

`Goals`

```
|---e2e/
|   |---app.e2e-spec.ts
|   |---app.po.ts
|   |---tsconfig.e2e.json
|---node_modules/...
|---src/...
|---.angular-cli.json
|---.editorconfig
|---.gitignore
|---karma.conf.js
|---package.json
|---protractor.conf.js
|---README.md
|---tsconfig.json
|---tslint.json
```

e2e/

Inside e2e/ live the end-to-end tests. They shouldn't be inside src/ because e2e tests are really a separate app that just so happens to test your main app. That's also why they have their own tsconfig.e2e.json.

node_modules/

Node.js creates this folder and puts all third party modules listed in package.json inside of it.

.angular-cli.json

Configuration for Angular CLI. In this file you can set several defaults and also configure what files are included when your project is built. Check out the official documentation if you want to know more.

.editorconfig

Simple configuration for your editor to make sure everyone that uses your project has the same basic configuration. Most editors support an .editorconfig file. See http://editorconfig.org (Links to an external site.) for more information.

.gitignore

Git configuration to make sure autogenerated files are not commited to source control.

karma.conf.js

Unit test configuration for the Karma test runner, used when running ng test.

package.json

npm configuration listing the third party packages your project uses. You can also add your own custom scripts here.

protractor.conf.js

End-to-end test configuration for Protractor, used when running ng e2e.

README.md (Links to an external site.)

Basic documentation for your project, pre-filled with CLI command information. Make sure to enhance it with project documentation so that anyone checking out the repo can build your app!

tsconfig.json

TypeScript compiler configuration for your IDE to pick up and give you helpful tooling.

tslint.json

Linting configuration for TSLint together with Codelyzer, used when running ng lint. Linting helps keep your code style consistent.

Monday: Components

Angular Components

Components are the building blocks in Angular apps. A component can display data on the screen, listen to user input such as a button click and take action depending on the user input, for example returning a new page after a button click.

Each angular app has to have a root component which acts as the top-level component. This component is rendered first to the user. As we develop our Goals app, we'll be creating more components which will be different from the root component we have now. We can also nest components inside other components to make larger components. We'll see this in practice later as we create more components. The root component will be the **parent component** and the other components we nest inside it will be the **child components**. Whenever we initiate an angular application, we tell the browser to render the parent component, which in turn renders its child components if they exist.

Structure of a Component

An angular component has two parts:

1. The Component annotation.
2. The Component definition class.

Let's use the *AppComponent* to see these two parts.

src/app/app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  goal = 'Watch Finding Nemo'
}
```

At the top, the `import` keyword is used to import modules we want to use in a component. Here, we import `Component` from `@angular/core` which is where `Component` is located.

Component Annotation

The `@Component` is a metadata annotation, which means it defines metadata properties related to the app component. In the `@Component` annotation, we have metadata for the `selector` attribute which has a value '`app-root'` that uniquely identifies the *AppComponent*. We have the `templateUrl` metadata attribute whose value points to the HTML template for *AppComponent*. The `styleUrls` metadata attribute has a value that points to the CSS file specific to the *AppComponent*.

Component Definition Class

src/app/app.component.ts

```
export class AppComponent {
  goal = 'Watch Finding Nemo'
}
```

The component definition is a typescript class that we'll use to define the logic of the AppComponent. We `export` the class so that we can import it anywhere else we need to use it the application. Inside the class, we have defined a property `goal` and assigned it a value of `Watch Finding Nemo` which is a string.

Displaying an Array

Typically, we all have more than one goal to achieve. Let's create more goals in an array and then display them in our Goals app.

src/app/app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  goals:string[];
  constructor(){
    this.goals = ['Watch finding Nemo', 'Buy cookies', 'Get new phone case']
  }
}
```

We have created a property `goals` and attributed it to an empty String array. We have then created a **constructor** function and given the empty `goals` array 3 string values so it is no longer empty. A **constructor** function defines the logic that should be executed once the class is instantiated. In this case, when the *AppComponent* is initiated, it creates an instance of the goals array with the 3 values that we have assigned it.

ngFor Directive

Let us now display these goals.
src/app/app.component.html

```
<div>
  <h1>My Goals</h1>
  <ul>
      <li *ngFor='let goal of goals'>
        {{goal}}
      </li>
  </ul>
</div>
```

We have created an unordered list with a list tag that has some logic, the `*ngFor` directive. The `*ngFor` is an angular **repeater directive** that loops through the host element `goals` which is a list. Don't forget the asterisk sign `*`before `ng`, it's part of the syntax. The directive goes through the `goals` array and assigns each item in the array the variable `goal`. We have then displayed the `goal` variable inside the list tag using Angular's interpolation binding syntax.

This directive works the same as the for loop in Javascript that we learned in Moringa Prep. An excerpt of the loop syntax used we used in Javascript is:

```
var arr = [3, 5, 7,11,13];
for (var i in arr){
 console.log(i);}
```

The directive gives a similar output as this loop, only that it now has its own syntax that's different.

If we check our browser now, we see that the app displays a list of the goals that we have created.

Monday: Creating a class

Creating a Class

A goal has much more detail than just a name. It can have a completion date, an ID, number of participants, an owner, etc. So far, we have been creating our goals as pure strings inside our *AppComponent*. We can continue defining these goals as strings in our *AppComponent* but it will get tedious and clumsy as we create more and more goals. To curb this, angular allows us to create a class that will be the blueprint for creating goal objects.

Goal Blueprint Class

While learning Javascript in Prep, we got familiar with Object-Oriented Programming in which we build objects from classes. A class is an extensible program-code-template for creating objects by providing initial values for variables and member functions or methods. Let us create a class with the angular CLI to define the blueprint of a goal which is how we will be creating Goal objects. On our terminal, let's execute this command:

```
$ ng generate class Goal
```

We use the `ng generate class <class-name>` command to create a class using the angular CLI. We will create goals that have a name and an ID. Let us define this logic in the goal class we have just created.

src/app/goal.ts

```
export class Goal {
  id: number;
  name: string;
}
```

Inside the class `Goal`, we have defined the `id` and attributed it to the datatype number and `name` to be a string. We have exported the class to make it available for use anywhere else we need it in the app.

Displaying an Array

Let us create several goals using the goal blueprint class.

src/app/app.component.ts

```
import { Component } from '@angular/core';
import { Goal } from './goal';
....
export class AppComponent {
  goals:Goal[] = [
    {id:1, name:'Watch finding Nemo'},
    {id:2,name:'Buy Cookies'},
    {id:3,name:'Get new Phone Case'},
    {id:4,name:'Get Dog Food'},
    {id:5,name:'Solve math homework'},
    {id:6,name:'Plot my world domination plan'},
  ];
}
```

At the top, we have imported the `Goal` blueprint class we have just created. If we do not import it, we'll get errors in our application because the component in which we are trying to use this class does not recognize it yet. The period signs `...` are just to show that we do not change the code before the *AppComponent* class. Inside the *AppComponent* class, we have created an object `goals` and attributed it to the `Goal` blueprint and then defined the array of goals each with its ID and name.

When we check our browser now, the application is not broken but it does not show us the actual goals. It shows us the text [object Object]. This means that the browser can recognize that we want to display goal objects but our HTML code cannot display anything in specific, neither the id nor the name.

To display the goal name for each item in the list, we change our HTML template code to point us to the goal name instead of the goal object.

src/app/app.component.html

```
<div>
  <h1>My Goals</h1>
  <hr>
  <ul>
      <li *ngFor='let goal of goals'>
        {{goal.name}}
      </li>
  </ul>
</div>
```

When we check our browser, we now see the goal names in the list. For practice, display the goal ID along with the goal name.

NgIf Directive

Let's take a look at another directive in Angular known as **NgIf** directive.

src/app/app.component.html

```
<div>
  <h1>My Goals</h1>
  <hr>
  <ul>
      <li *ngFor='let goal of goals'>
        {{goal.name}}
      </li>
  </ul>
  <p *ngIf='goals.length > 5'>Your goals are too many</p>
</div>
```

We have added a <p> tag and defined the directive logic in it. Here, we check if the length of the goals array is greater than five, and when it is, we display the text in the paragraph tag. When we check our browser now, the text is displayed because we have 6 goals in the array. Delete one goal in the *src/app/app.component.ts* file and check your browser to see how this directive works. It simply checks if a certain condition is met and performs an action based on that condition.

Monday: Creating Components

Creating Components

Till now, the *app* component has been handling everything that's in our application. We want to scale our app and we are therefore going to create more components. The beauty of components is that they enable us to decouple our application so that each functionality is implemented by a specific component. We end up having many components working together to achieve the whole purpose of the app.

Since it is the root components' job to render all other components, let us start out by creating a *goal* component to handle the logic on goals. In your terminal, run the following command:

```
$ ng generate component goal
```

This command has created a folder /*goal* that has the files which contain the boilerplate code for the *goal* component. If we also check the *app.module.ts,* we notice that the goal component has been registered under `declarations` automatically by the angular CLI.

We're moving the goals logic from the *AppComponent* to the *GoalComponent*. Let's move the `goals` array permanently from the *AppComponent* class to the *GoalComponent* class*:*

src/app/goal/goal.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Goal } from '../goal';
@Component({
  selector: 'app-goal',
  templateUrl: './goal.component.html',
  styleUrls: ['./goal.component.css']
})
export class GoalComponent implements OnInit {
  goals:Goal[] = [
    {id:1, name:'Watch finding Nemo'},
    {id:2,name:'Buy Cookies'},
    {id:3,name:'Get new Phone Case'},
    {id:4,name:'Get Dog Food'},
    {id:5,name:'Solve math homework'},
    {id:6,name:'Plot my world domination plan'},
  ];
  constructor() { }
  ngOnInit() {
  }
}
```

At the top, we have imported the `Goal` blueprint class and then defined the array of goals inside the *GoalComponent* definition class. `ngOnInit` is a lifecycle hook. It is called each time the component is created. We use it to put complex initialization logic that we want for the component.

Let's also move the template logic to the *goal* Component.

src/app/goal/goal.component.html

```
<div>
  <h1>My Goals</h1>
  <hr>
  <ul>
    <li *ngFor='let goal of goals'>
      {{goal.name}}
    </li>
  </ul>
</div>
```

Nesting Components

Now that we have all our goals logic inside the *GoalComponent*, let's make it available as a child component of the root component, *AppComponent*. If we check our browser right now, there's nothing displaying because our root component has nothing to display. Using the *GoalComponent*'s selector, let's nest the *GoalComponent* inside the *AppComponent* so we can display the contents of the goal component.

src/app/app.component.html

```
<app-goal></app-goal>
```

If we check our browser now, the goals are displayed just like before. Yes, nesting is as simple as that! It ensures that the child component is loaded while inside the parent component.

Child Components

Let's add something more to our goals, let's give each goal a description. We will use the description as the detail of our goals and display it on our app. We will create a G*oalDetailComponent* which will be a child component of the *GoalComponent*. The G*oal-detailComponent* will be responsible for displaying the details of each goal which in our case will be the description of a goal.

Currently, our goal blueprint allows us to create goal objects with an `id` and `name` only. Let's add `description` to the goal blueprint so it allows us to create a description for each of our goals.

src/app/goal.ts

```
export class Goal {
  id: number;
  name: string;
  description: string;
}
```

We can now add descriptions to our goals in the array so that we can display them in our G*oalDetailComponent*.

src/app/goal/goal.component.ts

```
...
export class GoalComponent implements OnInit {
  goals: Goal[] = [
    {id:1, name:'Watch finding Nemo',description:'Find an online version and watch
merlin find his son'},
    {id:2,name:'Buy Cookies',description:'I have to buy cookies for the parrot'},
    {id:3,name:'Get new Phone Case',description:'Diana has her birthday coming up
soon'},
    {id:4,name:'Get Dog Food',description:'Pupper likes expensive sancks'},
    {id:5,name:'Solve math homework',description:'Damn Math'},
    {id:6,name:'Plot my world domination plan',description:'Cause I am an evil
overlord'},
  ];
...
```

Input Property Binding

Our G*oalDetailComponent* will be receiving the goal description to display from the parent component, *GoalComponent*. For a child component to receive data from a parent component in Angular, we need to do input property binding. **Input property binding** allows us to pass data from a parent component to its child components. Let's see how we do this. On your terminal, use the CLI to generate a component named *goal-detail* and navigate to the G*oalDetailComponent* class file.

src/app/goal-detail/goal-detail.component.ts

```
import { Component, OnInit, Input } from '@angular/core';
import { Goal } from '../goal';
@Component({
  selector: 'app-goal-detail',
  templateUrl: './goal-detail.component.html',
  styleUrls: ['./goal-detail.component.css']
})
export class GoalDetailComponent implements OnInit {
  @Input() goal: Goal;
  constructor() { }
  ngOnInit() {
  }
}
```

At the top, we add an import for `Input` from `@angular/core` which allows us to do Input property binding. We also import the `Goal` blueprint class. In the definition class for the G*oalDetailComponent*, we define `goal` as the property that will undergo input property binding, which is of the type `Goal`*,* from the blueprint class. This means when we will have the property `goal` in the G*oalDetailComponent*, it will have received its data from a parent component, in our case, the *GoalComponent*.

To bind to this `goal` property from the parent component *GoalComponent* to the child component *GoalDetailComponent* , we change our *GoalComponent* template code to this:

src/app/goal/goal.component.html

```
<div>
  <h1>My Goals</h1>
  <hr>
  <ul>
    <li *ngFor='let goal of goals'>
      {{goal.name}}
      <app-goal-detail [goal]='goal'></app-goal-detail>
    </li>
  </ul>
</div>
```

We have used the selector for the *GoalDetailComponent* and nested it into the *GoalComponent*, placing it after the goal name which is where we want to display the description of each goal. We have also added some logic in the child component tags and specified that we are binding the `goal` property to the *GoalDetailComponent*. The *GoalDetailComponent* is now receiving data from the *GoalComponent* so we now need to display this data in the *GoalDetailComponent*.

src/app/goal-detail/goal-detail.component.html

```
<p>
  {{goal.description}}
</p>
```

When we serve our application at this point, we can see the application is now displaying each goal with its description below it.

Tuesday: Output Property Binding

Output Property Binding

Our application at this point shows us our goals and their descriptions at the same time. Why don't we build some interactivity in it, in a way that, we don't always see the description of a goal, but instead, we click a button that toggles between showing and hiding a description? This will help us understand how output property binding works. Just as input property binding passes data into a component, **output property binding** passes data out of a component.

Let's start by adding this property to our goal blueprint to enable us to toggle between showing and hiding a goal description.

src/app/goal.ts

```
export class Goal {
  showDescription: boolean;
  constructor(public id: number,public name: string,public description: string){
    this.showDescription=false;
  }
}
```

We have changed how we create the `Goal` blueprint class by using a constructor function. We have created a property `showDescription` and assigned it to the data type `boolean`. Inside the constructor function, we have passed the goal properties we had before, as arguments of the constructor and declared that the `showDescription` property should be initialized as `false` so that the description is not displayed. We'll write the code to control this logic for hiding and showing goals later on. Notice that we have used the keyword `this` to give the `showDescription` property class access. The `public` keyword is an access modifier and it determines where the class properties are visible which in our case is anywhere outside the class. If we used the `private` keyword, the properties would only be visible inside the class.

Why use a constructor function? A constructor function enables us to define the initialization logic for creating an object. Our Goal object in this case still needs the properties `id,name` and `description` to instantiate our Goal object. The `showDescription` property, on the contrary, is not mandatory when creating a Goal object. That's the reason we have used the constructor function, we are telling our angular application that it should initialize a goal object requiring the `id,name` and `description` as mandatory properties and as well add `showDescription` to a goal object immediately setting its value to `false`. Each goal object we create from now on will, therefore, have the `showDescription` property although we will not explicitly define this property for each Goal object that we create.

Let's also change our Goals so we can put the constructor into use.

src/app/goal/goal.component.ts

```
...
export class GoalComponent implements OnInit {
  goals: Goal[] = [
    new Goal(1, 'Watch finding Nemo', 'Find an online version and watch merlin find
his son'),
    new Goal(2,'Buy Cookies','I have to buy cookies for the parrot'),
    new Goal(3,'Get new Phone Case','Diana has her birthday coming up soon'),
    new Goal(4,'Get Dog Food','Pupper likes expensive snacks'),
    new Goal(5,'Solve math homework','Damn Math'),
    new Goal(6,'Plot my world domination plan','Cause I am an evil overlord'),
  ];
...
```

We have created the property `goals` and specified that it will be an array of type `Goal`. When defining each goal, we use the keyword `new` and call the Goal blueprint class, inside it specifying the three mandatory properties in the constructor function `id, name, & description`

Let's now add the logic for showing and hiding a goal description in our HTML template file.

src/app/goal/goal.component.html

```
<h1>My Goals</h1>
<hr>
<ul>
  <div *ngFor='let goal of goals;let i = index'>
    <li>{{goal.name}}</li>
    <button (click)='toggleDetails(i)'>Toggle Details</button>
    <app-goal-detail *ngIf='goal.showDescription' [goal]='goal'></app-goal-detail>
  </div>
```

```
</ul>
<p *ngIf='goals.length > 5'>You have too many goals</p>
```

In the div we have created, we have added a loop logic,`let i = index` to register the index of each goal item in the goals list. We have then displayed the goal name and below it created a button that also contains some logic.

click Event binding

The logic we have added to the button is an event binding syntax, the click event binding. We are telling the angular app to listen for a click event on this button and once it happens, it should execute the `toggleDetails()` function which we will create in a few. The `toggleDetails()` function takes in the index position of the goal item as an argument. We have then updated the G*oalDetail* template tags with the `*ngIf` directive, instructing it to display the goal description if it exists.

Show and hide Logic

Let's now create the `toggleDetails()` function that will display and hide a goal description.

src/app/goal/goal.component.ts

```
export class GoalComponent implements OnInit {
...
  toggleDetails(index){
    this.goals[index].showDescription = !this.goals[index].showDescription;
  }
  constructor() { }
  ngOnInit() {
  }
}
```

We have defined the `toggleDetails()` function in our component class and specified that it takes `index` as an argument. Inside the function, we have defined the logic for displaying the goal description, which in our case changes the `showDescription` from `false` to `true` and vice versa each time the function is executed. The `goals[index]` ensures that the function is executed for the goal at the specific index.

If our server is still running, we can now interact with our application on the web browser. We notice that the goal descriptions are no longer displayed and we have the button which toggles between displaying and hiding a goal description.

Tuesday:Emitting Events

Emitting Events

When we want a child component to communicate with a parent component, we make the child component emit an event that is taken up by the parent component. Let us make our child component *GoalDetailComponent*, communicate with its parent component, the *GoalComponent.* We'll do this by adding a button to the *GoalDetailComponent* that deletes a goal once we ascertain that we have completed it.

Let's create this button in our *GoalDetail* HTML template.

src/app/goal-detail.component.html

```
<p>{{goal.description}}</p>
<button (click)= 'goalComplete(true)'>Complete</button>
```

We have added a button below the goal description `<p>` tag and defined some logic inside it. We have added a click event binding syntax which will call the `goalComplete(true)` function once the button is clicked. Let's now create this function in the logic file of the *GoalDetailComponent*.

src/app/goal-detail.component.ts

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
...
export class GoalDetailComponent implements OnInit {
  @Input() goal: Goal;
  @Output() isComplete = new EventEmitter<boolean>();
  goalComplete(complete:boolean){
    this.isComplete.emit(complete);
  }
  constructor() { }
  ngOnInit() {
  }
}
```

At the top, we have imported `Output` decorator and the `EventEmitter` class. We have then used the `Output` decorator to define `isComplete` as an EventEmitter that takes in a boolean. After that, we have created our `goalComplete()` function which calls the `emit` method on the `isComplete` EventEmitter. What this does is pass this event to the parent component. We, therefore, have to make the parent component process this event. Let's write the code for this in our parent component, *GoalComponent*.

src/app/goal/goal.component.html

```
<h1>My Goals</h1>
<hr>
<ul>
  <div *ngFor='let goal of goals;let i = index'>
    <li>{{goal.name}}</li>
    <button (click)='toggleDetails(i)'>Toggle Details</button>
    <app-goal-detail *ngIf='goal.showDescription' [goal]='goal' (isComplete) =
'completeGoal($event,i)'></app-goal-detail>
  </div>
</ul>
<p *ngIf='goals.length > 5'>You have too many goals</p>
```

To make the parent component receive this event, we need to catch the event being emitted in the parent component and define a function that will be triggered once this event is captured.

The event being emitted is `isComplete`. We catch this event in the parent component using parenthesis`()`and then define that when it is emitted, the `completeGoal($event,i)` function should be called. Inside the function, we have used the special variable, `$event` as a placeholder for the values that we expect to be emitted by the event. We have also passed in the index `i` of the goal item in the function.

Finally, let's create this `completeGoal()` function which will be triggered when our event is captured.

src/app/goal/goal.component.ts

```
...
export class GoalComponent implements OnInit {
...
  completeGoal(isComplete, index){
    if (isComplete) {
      this.goals.splice(index,1);
    }
  }

  constructor() { }

  ngOnInit() {
  }
}
```

Inside the function, we have passed in the `isComplete` event emitter and `index` as our arguments then used the `splice` javascript function to delete the goal at the index. Keep in mind, `isComplete` in this case, will return the value `true` that we set in the HTML template file of the *GoalDetailComponent* and `index` will be the exact index of a goal from the loop in the *GoalComponent* HTML template file.

When we serve our application now, we have the complete button which deletes a goal at a specific index from our goals array when clicked.

Tuesday:Directives

Directives in Angular

A directive is a function that executes whenever the Angular compiler finds it in the DOM. Directives in Angular are used to extend the power of HTML by giving it a new syntax that actually processes data from HTML. There are two major types of directives in Angular:

1. Attribute Directives.
2. Structural Directives.

Attribute Directives

Attribute directives manipulate the DOM by changing its behaviour and appearance. We use attribute directives to apply conditional styles to elements, show or hide elements, and, dynamically change the behaviour of an element based on a changing property, say, make a form field blink red if a user tries to submit the form without filling in that specific form field.

Structural Directives

Structural directives, on the other hand, are used to create and/or destroy DOM elements. This type of directives actually add or completely remove elements from the DOM unlike some attribute directives like `hidden` which maintain the DOM as it is. We should, therefore, be careful when we decide to use structural directives.

Angular has its own predefined directives, which all have names. We have encountered some like `*ngIf` and `*ngFor` before. The good news is, we're not limited to using only those that Angular has predefined, we can create our own custom directives that can be attribute or structural directives and be in a position to use them in our app. Let's create a custom directive to understand how we can use this privilege to create our own directives in Angular.

Custom Directives

What if we wanted to strikethrough a goal after finishing it instead of deleting it? To this point, Angular has no inbuilt directive to do this for us. Let us create our own custom directive that will strikethrough a goal after we finish it.

At our terminal, let's create a directive using this command:

```
$ ng generate directive strikethrough
```

This command generates a file *strikethrough.directive.ts* which we are about to use to write the code that will perform a strikethrough and another file *strikethrough.directive.spec.ts* which is a test file. At the same time, it declares this directive in the root level modules file, *app.module.ts,* for us.

src/app/strikethrough.directive.ts

```
import { Directive } from '@angular/core';
@Directive({
  selector: '[appStrikethrough]'
})
export class StrikethroughDirective {
  constructor(){}
}
```

A directive class has the `@directive` annotation. The annotation of a directive only has the `selector` property which is passed in as the attribute to the host element. The brackets `[]` make it an attribute directive. Angular looks in the HTML template for elements that have this selector and applies the logic that follows to the elements.

Let's go on to create this logic.

src/app/strikethrough.directive.ts

```
import { Directive,ElementRef } from '@angular/core';
@Directive({
  selector: '[appStrikethrough]'
})
export class StrikethroughDirective {
  constructor(private elem:ElementRef){}
}
```

We have imported the `ElementRef` at the top which we have used in the constructor of the directive's definition class. We use `ElementRef` to inject a reference to the host DOM element in which we will use this directive.

src/app/strikethrough.directive.ts

```
import { Directive,ElementRef} from '@angular/core';
@Directive({
  selector: '[appStrikethrough]'
})
export class StrikethroughDirective {
  constructor(private elem:ElementRef){
    this.elem.nativeElement.style.textDecoration='line-through';
  }
}
```

We have then targeted the host element's style attribute and changed the text-decoration to line-through. The `ElementRef` grants us direct access to the host DOM element through its `nativeElement` property.

Let's add this directive to our host element to see the changes it creates.

src/app/goal.component.html

```
<h1>My Goals</h1>
<ul>
    <div *ngFor='let goal of goals;let i = index'>
        <li appStrikethrough>{{goal.name}}</li>
        <button  (click)='toggleDetails(i)'>Toggle Details</button>
    <app-goal-details *ngIf="goal.showDescription" [goal]='goal' (isComplete)=
'completeGoal($event,i)'></app-goal-details>

    </div>
</ul>
<p *ngIf='goals.length > 5'>You have too many goals</p>
```

If our local server is still running, we see the goals have a line crossing the goal name which means that our directive actually works. That's great, right? However, using the directive this way is not helpful because we only wanted to strikethrough a goal if we have finished it.

User-initiated events

We'll put the user in control of making the strikethrough work on the app. When the user clicks a goal, it is marked as complete with the strikethrough and when the user double-clicks on the goal, it is marked an incomplete by removing the strikethrough from the goal.

Let's implement this in our directive.

src/app/strikethrough.directive.ts

```
import { Directive,ElementRef} from '@angular/core';
@Directive({
  selector: '[appStrikethrough]'
})
export class StrikethroughDirective {
  constructor(private elem:ElementRef){ }

  private textDeco(action:string){
    this.elem.nativeElement.style.textDecoration=action;
  }
}
```

We have changed our directive's logic by creating a function `textDeco()` which takes in an action and then performs a text-decoration using the action. Let's now create these actions that will feed into our `textDeco()` function.

src/app/strikethrough.directive.ts

```
import { Directive,ElementRef,HostListener} from '@angular/core';
@Directive({
  selector: '[appStrikethrough]'
})
export class StrikethroughDirective {

  constructor(private elem:ElementRef){}
  @HostListener("click") onClicks(){
    this.textDeco("line-through")
  }
  @HostListener("dblclick") onDoubleClicks(){
    this.textDeco("None")
  }
  private textDeco(action:string){
    this.elem.nativeElement.style.textDecoration=action;
  }
}
```

At the top, we have imported `HostListener` and used it to define the events that will be initiated by user actions, the first one being a click that creates a line-through and the second one being a double click which changes the text-decoration to none thus removing the line-through. We can see that for each action, we call our `textDeco()` function, `this.textDeco()` and pass in the name of the action as a string.

Let's serve our application and interact with it by clicking and double-clicking on the goals to see our directive at work.

Tuesday: Pipes

Pipes

In Angular, a pipe takes in data as input and transforms it into the desired output. Imagine having a date that reads like this:

Fri Feb 15 1998 00:00:00 GMT-0700 (Pacific Daylight Time).

This date is a correct and viable date but it has too much detail. This same date in a simple and readable format would look something like this:

February 15, 1998

In Angular, we can do such transformations with the help of pipes. Angular has several inbuilt pipes that we can use. We can also create our own custom pipes if we want to. We'll use some inbuilt pipes first and later create our own custom pipes to understand what this really means.

In our case, we will use an inbuilt Angular pipe to simplify our dates. To do this, we'll add a completion date to our goal blueprint class so that each goal can have a `completeDate` property.

src/app/goal.ts

```
export class Goal {
  public showDescription: boolean;
  constructor(public id: number,public name: string,public description: string,
public completeDate: Date){
    this.showDescription=false;
  }
}
```

We have added the `completeDate` property in our constructor for the goal blueprint and set its datatype to `Date`. We now need to update our `Goals` array to add a completion date to each goal.

src/app/goal/goal.component.ts

```
...
```

```
export class GoalComponent implements OnInit {
  goals: Goal[] = [
    new Goal(1, 'Watch finding Nemo', 'Find an online version and watch merlin find
his son',new Date(2020,3,14)),
    new Goal(2,'Buy Cookies','I have to buy cookies for the parrot',new
Date(2019,6,9)),
    new Goal(3,'Get new Phone Case','Diana has her birthday coming up soon',new
Date(2022,1,12)),
    new Goal(4,'Get Dog Food','Pupper likes expensive snacks',new Date(2019,0,18)),
    new Goal(5,'Solve math homework','Damn Math',new Date(2019,2,14)),
    new Goal(6,'Plot my world domination plan','Cause I am an evil overlord',new
Date(2030,3,14)),
  ];
...
```

The `Date` instance takes 3 arguments the first one is the year, the second argument is the month the third argument is the date. The month is calculated from 0-11 where 0 is January and 11 is December.

Let's now display the completion date along with our goals.

src/app/goal/goal.component.html

```
...
  <div *ngFor='let goal of goals;let i = index'>
    <li appStrikethrough>{{goal.name}} due on {{goal.completeDate}}</li>
...
```

We have added the completion date to our template beside the goal name. If we serve our application, we can see the completion date. This date does not look user-friendly, it looks complicated. Let's use the inbuilt date pipe in Angular to convert it to a readable format.

src/app/goal/goal.component.html

```
...
  <div *ngFor='let goal of goals;let i = index'>
    <li appStrikethrough>{{goal.name}} due on {{goal.completeDate|date}}</li>
...
```

If we look at our application now, the date is in a simpler format that looks user-friendly. We have added the date pipe `|date` in our template, which has converted the initial date we had to a simple readable format. The date pipe has made it simple for us to convert our date into a readable format.

Chaining Pipes

We can also chain pipes to extend the transformation of the input that we give to a pipe.

src/app/goal/goal.component.html

```
...
  <div *ngFor='let goal of goals;let i = index'>
    <li appStrikethrough>{{goal.name}} due on {{goal.completeDate|date|
uppercase}}</li>
...
```

We have added the `uppercase` pipe to our completion date, which as the name suggests, converts the completion date to capital characters. When we serve our application, we see the date is now in all caps.

Tuesday: Create Pipes

Creating Custom Pipes

As we had stated earlier, if the inbuilt angular pipes don't help us to achieve the data transformation we want, we have the liberty to create our own custom pipes to do it. Let's create a pipe that will count the number of days left to the completion of a goal.

On our terminals, let's create a pipe using the following command:

```
$ ng generate pipe date-count
```

This command generates two files, *date-count.pipe.ts* which we'll use to write the code for counting the number of days left and *date-count.pipe.spec.ts* which is the test file for our pipe. The command also adds this pipe to our root level modules file, *app.module.ts*, for us.

Let's now define the logic to count the number of days left to the completion of a goal.

src/app/date-count.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'dateCount'
})
export class DateCountPipe implements PipeTransform {
  transform(value: any): number {
    let today:Date = new Date(); //get current date and time
    let todayWithNoTime:any = new Date(today.getFullYear(), today.getMonth(), 
today.getDate())
    var dateDifference = Math.abs(value - todayWithNoTime) //returns value in 
miliseconds
    const secondsInDay = 86400; //60 seconds * 60 minutes in an hour * 24 hours in 
a day
    var dateDifferenceSeconds = dateDifference*0.001; //converts miliseconds to 
seconds
    var dateCounter = dateDifferenceSeconds/secondsInDay;
    if (dateCounter >= 1 && value > todayWithNoTime){
      return dateCounter;
    }else{
      return 0;
    }
  }
}
```

We first have the `@Pipe` decorator that defines the *name* of the pipe. We then have the `DateCountPipe` class that has a transform function that takes in a value of type any. This value will be our date object from the goals.

We have used the `new Date()`; function to get the current date and time. But since we do not need the time aspect we create a new date object for today's date which we store in the `todayWithNoTime` variable.

We have then calculated the difference of the date we get in the pipe input and subtracted the current date to get the difference in milliseconds.

We have converted the milliseconds to seconds and then divided these seconds by the seconds in a day to get how many days are left.

We have then returned the days left if the goal date has not been passed. But if it past due we post 0.

Let us add this pipe to our goal details and serve our application to see our custom pipe in action.

src/app/goal-details/goal-detail.component.html

```
<p>{{goal.description}}</p>
<p>This goal will be complete in {{goal.completeDate|dateCount}} days</p>
```

We have added the pipe to the `completeDate` property to calcute the number of days left to the completion of a goal.

Monday & Tuesday: Practice - To Do List,Flash Cards

Goal: Over the course of the last two days, you have learnt how to create a basic application using Angular, including the file structure, components, property binding, event emitting, directives, and pipes. Take this time to familiarize and solidify your understanding of these concepts.

Code

Goals

First, follow along with the Monday and Tuesday lessons, creating the Goals application.

Only after you have completed the lessons and understand how an Angular application is built, move on to practice your skills further with the next projects.

To Do List

Create a to-do list application in Angular. Just like we did in our Goals application, you will have to hard code your Task objects. For now, your Task object should have properties such as name and description.

Flash Card

You have learnt a lot since you start this journey of becoming a Programmer. Wouldn't it be really cool if you could have somewhere to record all the new programming vocabularies (and their explanations) that you have encountered?

Once you finish creating the to-do list project above and feel comfortable about the concepts you learnt so far, you can challenge yourself further and create a flashcard app. A flashcard app is a site where you create cards that contain new words you learnt and their explanations.

Follow along with the lessons you've covered so far and add functionality similar to the Goals app with the following user stories as your guide:

1. As a user, I would like to see the title of each flash card
2. as a user, I want to be able to click on the card's title to see it's explanation/detail.
3. As a user, I want to be able to hide a card detail when am done viewing them.

Wednesday: Deleting Goals

Deleting Goals

We earlier created a custom directive that strikes through a goal once we complete it by clicking it. We can replace the complete button in the *goal-detail* component with a delete button that deletes a goal once we decide we want to get rid of it. Let's add this button to the *goal-detail* template.

src/app/goal-detail/goal-detail.component.html

```
<p>{{goal.description}}</p>
<p>This goal will be complete in {{goal.completeDate|dateCount}} days</p>
<button (click)= 'goalDelete(true)'>Delete Goal</button>
```

We have modified the button to emit a `click` output event that calls the `goalDelete()` function which takes a boolean as an argument. We also need to change the function that we had defined in the *goal-detail* component class to our new `goalDelete()` function.

src/app/goal-detail/goal-detail.component.ts

```
...
export class GoalDetailComponent implements OnInit {
...
  goalDelete(complete:boolean){
    this.isComplete.emit(complete);
  }
...
}
...
```

We have used the `isComplete` event emitter that we defined earlier and emitted the boolean value passed in. We can now update the goal parent component.

src/app/goal/goal.component.html

```
...
<app-goal-detail *ngIf='goal.showDescription' [goal]='goal' (isComplete) =
'deleteGoal($event,i)'></app-goal-detail>
...
```

We have updated the goal-detail selector, *app-goal-detail*, with our event emitter `isComplete` , specifying that when the event is emitted, `deleteGoal()` is the function that should be executed.

Let's also define the logic for this `deleteGoal()` function which takes in the value of the event being emitted and the index of a goal.

src/app/goal/goal.component.ts

```
...
export class GoalComponent implements OnInit {
...
  deleteGoal(isComplete, index){
    if (isComplete) {
      let toDelete = confirm(`Are you sure you want to delete $
{this.goals[index].name}?`)
      if (toDelete){
        this.goals.splice(index,1)
      }
    }
  }
...
}
```

We have created the `deleteGoal()` function that takes in the boolean value and index of the goal. We have then created the `toDelete` boolean variable that calls the Javascript confirm function that creates a browser prompt for the user to confirm if they actually want to delete the goal. If the user confirms that the goal is to be deleted we use the `splice` function to remove the goal from the list of goals using the index. Note that while creating the confirm prompt, we have used backticks `` ` ` `` which allow us to use the `${this.goals[index].name}` logic inside the confirm prompt. Also, the javascript prompt gives a user two options, ok and cancel and depending on the option the user clicks on, a boolean value is returned. If the user clicks ok, the value true is returned, and if the user clicks cancel, the value false is returned.

Let's serve our application and interact with this feature we have created.

Wednesday: Add Boostrap

Adding Bootstrap

Our application is looking very plain and stale at the moment. We can beautify it using the frontend web component library we covered in prep, bootstrap. We will add bootstrap to Angular using the command line. The boostrap4 documentation also clarifies that some components in bootstrap, like modals, need jquery and popper to work so let's install all of them in case we need to use such components.

```
$ npm install --save bootstrap popper
```

This command will add bootstrap to our node modules. We need to make it available inside the application at the root level for us to use it. We'll do this in our *styles.css* file available at root.

src/app/styles.css

```
@import "~bootstrap/dist/css/bootstrap.css"
```

We have done this by simply importing bootstrap in the root CSS file. We can now use bootstrap in our application. At this point, if we take a look at our application after starting our local server, we can spot a change in the font already since bootstrap is installed.

Let's use a simple layout for our application.

src/app/goal/goal.component.html

```
<div class="container">
  <h1>My Goals</h1>
  <hr>
  <div class="row">
    <div class="col-md-6">
      <div *ngFor='let goal of goals;let i = index'>
        <div>
          <h4 id={{i}} appStrikethrough>{{goal.name}} due on {{goal.completeDate|
date|uppercase}}</h4>
          <button (click)='toggleDetails(i)' class="btn btn-primary">Toggle
Details</button>
        </div>
        <div *ngIf='goal.showDescription'>
          <app-goal-detail  [goal]='goal' (isComplete) =
'deleteGoal($event,i)'></app-goal-detail>
        </div>
      </div>
    </div>
  </div>
  <p *ngIf='goals.length > 5'>You have too many goals</p>
</div>
```

src/app/goal-detail/goal-detail.component.html

```
<p>{{goal.description}}</p>
<p>This goal will be complete in {{goal.completeDate|dateCount}} days</p>
<button (click)= 'goalDelete(true)' class="btn btn-outline-danger">Delete
Goal</button>
```

We have used boostrap classes in our HTML templates tags and the bootstrap grid layout also. Feel free to make adjustments to your application to make it more visually appealing.

Wednesday:Form Component

Form Component

If we want to add goals to our application at the moment, we would have to do it by hard coding the goals in the goals array. What if we wanted our users to add goals of their liking from the frontend interface that they are interacting with? How would we go about it?

Well, thanks to web forms, we can give the user this power. A web form, also called a HTML form, helps us collect data from users by providing inputs in which users can fill in data. We will create a web form in which a user can fill in the details of a goal, that is, name, description and completion date to enable users to add goals from the user interface.

We will create our form the same way we have been creating other components. Let's create this form component using the angular CLI on our terminal:

```
$ ng generate component goal-form
```

This command has created the folder goal-form and inside it created the files that comprise a component. It has also added our goal-form component to our root module declarations array in the *app.module.ts* file.

Let's now define the structure of our form.

src/app/goal-form/goal-form.component.html

```
<div class="container-fluid">
  <h2 class="text-center">Create a new Goal</h2>
  <hr>
  <form>
      <div class="form-group">
          <label for="name">Name</label>
          <input type="text" required class="form-control" id="name">
      </div>
      <div class="form-group">
          <label for="description">Description</label>
          <textarea class="form-control" id="description" rows="4"
required></textarea>
      </div>
      <div class="form-group">
          <label for="complete">Completion</label>
          <input type='date' id="complete" required>
      </div>
  </form>
</div>
```

We have created a form with three inputs, name, description and date which are required to create a goal object. We will now use the selector of the *goal-form* component to nest it inside the goal component.

src/app/goal/goal.component.html

```
<div class="container">
  <h1>MY GOALS</h1>
  <hr>
  <div class="row">
    <div class="col-md-6">
      <div *ngFor='let goal of goals;let i = index'>
        <div>
          <h6 id={{i}} appStrikethrough>{{goal.name}} due on {{goal.completeDate|
date|uppercase}}</h6>
          <button (click)='toggleDetails(i)' class="btn btn-primary">Toggle
Details</button>
        </div>
        <div *ngIf='goal.showDescription'>
          <app-goal-detail  [goal]='goal' (isComplete) =
'deleteGoal($event,i)'></app-goal-detail>
        </div>
      </div>
    </div>
    <div class="col-md-6">
      <app-goal-form></app-goal-form>
    </div>
  </div>
  <p *ngIf='goals.length > 5'>You have too many goals</p>
</div>
```

We have added the form to the right side of our goals array. This form only has input fields and angular does not know how to interact with it at this point. In angular, whenever we want to make HTML forms interact with the angular app, we need to import the forms module from angular forms. We do this by making this import in our root modules file, *app.modules.ts*.

src/app/app.module.ts

```
....
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
....
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
....
```

At the top, we have imported the `FormsModule` and then added it to our `imports` array. We have done this import because the forms module is not available in our angular app by default. Our angular application can now work with any number of forms that we create thanks to the FormsModule.

Wednesday: NgModel

Two-Way Data Binding

Now that we have created a form that we will use to add goals in our application, let's make use of it. We want our users to see what they will be typing in real time, that is as they type it. This is the concept of two-way data binding. With two-way data binding, data moves from our template(the view) to our component class(the model) and vice versa. The essence is that when data changes on the template(view) it simultaneously changes in the component class(model) and if it also changes in the model, the view is automatically updated with the changes.

The FormsModule that we imported has a feature that will help us do the two-way data binding, the `ngModel` directive. This form has to create a goal the same way we defined it in the blueprint, so let's write code to make it create goals in the same format.

src/app/goal-form/goal-form.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Goal } from '../goal';
@Component({
  selector: 'app-goal-form',
  templateUrl: './goal-form.component.html',
  styleUrls: ['./goal-form.component.css']
})
export class GoalFormComponent implements OnInit {
  newGoal = new Goal(0,"","",new Date());
  constructor() { }
  ngOnInit() {
  }
}
```

At the top, we have imported the `Goal` blueprint class. Inside the component definition class, we have created a `newGoal` property and assigned it to the `Goal` class that takes in the format we have been using in creating goals. This new goal object will be changed by the form inputs.

src/app/goal-form/goal-form.component.html

```
<div class="container-fluid">
  <h2 class="text-center">Create a new Goal</h2>
  <hr>
  <form #goalForm='ngForm'>
      <div class="form-group">
          <label for="name">Name</label>
          <input type="text" required class="form-control" id="name"
[(ngModel)]="newGoal.name" name="name">
<!-- Testing to see if we get any data -->
          Display {{newGoal.name}}
      </div>
      <div class="form-group">
          <label for="description">Description</label>
          <textarea class="form-control" id="description" rows="4"
required></textarea>
      </div>
      <div class="form-group">
          <label for="complete">Completion</label>
          <input type='date' id="complete" required>
      </div>
      <button type="submit" class="btn btn-success">Add Goal</button>
  </form>
</div>
```

We have added a template reference variable `goalForm` to our form tag and equated it to `ngForm`. This will provide the form element with additional features and monitor the changes and validity of input elements.

We have then added the `[(ngModel)]` attribute to our name input and equated it to the name attribute of the `newGoal` object we created in our component definition class. We have then defined a `name` attribute which is a requirement when using `[(ngModel)]` and a form. The `Display {{newGoal.name}}` line will temporarily display the data being received.

If our server is still running, we can type anything in the form-input Name, and see two-way data binding in action.

Two-way data binding is one of the key things that Angular makes easy for us to implement. If you want this kind of experience in your app, Angular should be a preferred tool.

Wednesday: Form Validation

Form Validation

Users may end up trying to add a new goal without a name or description or a date while a goal is required to have each one. This may lead to our application developing errors. To mitigate this possible probability, we need to perform form validation on our form inputs and then alert users in case we encounter validity problems.

Let's see how we should do this.

src/app/goal-form/goal-form.component.html

```
<div class="container-fluid">
  <h2 class="text-center">Create a new Goal</h2>
  <hr>
  <form #goalForm='ngForm'>
      <div class="form-group">
          <label for="name">Name</label>
          <input type="text" class="form-control" id="name"
[(ngModel)]="newGoal.name" name="name" #name='ngModel' required>
          <div [hidden]="name.valid || name.pristine" class="alert alert-danger">
            <p>Name is required</p>
          </div>
      </div>
....
```

We have added a template reference variable `#name` to access Angular's form control on this input from the template. We have then created a div that has an attribute `[hidden]` which makes sure the alert is shown only when the name input is invalid. The `pristine` property allows us to hide the alert when we load a blank page.

If you type anything in the name input and then delete it now, the alert will show. Let's apply this to all our form inputs.

src/app/goal-form/goal-form.component.html

```
<div class="container-fluid">
  <h2 class="text-center">Create a new Goal</h2>
  <hr>
  <form #goalForm='ngForm'>
      <div class="form-group">
          <label for="name">Name</label>
          <input type="text" class="form-control" id="name"
[(ngModel)]="newGoal.name" name="name" #name='ngModel' required>
          <div [hidden]="name.valid || name.pristine" class="alert alert-danger">
            <p>Name is required</p>
          </div>
      </div>
      <div class="form-group">
          <label for="description">Description</label>
          <textarea class="form-control" id="description" rows="4"
[(ngModel)]="newGoal.description" name="description" #description="ngModel"
required></textarea>
          <div [hidden]="description.valid || description.pristine" class="alert
alert-danger">
            <p>Description is required</p>
          </div>
      </div>
      <div class="form-group">
          <label for="complete">Completion</label>
          <input type='date' id="complete" [(ngModel)]="newGoal.completeDate"
name="completeDate" #completeDate="ngModel" required>
          <div [hidden]="completeDate.valid || completeDate.pristine" class="alert
alert-danger">
            <p>Date is required</p>
          </div>
      </div>
      <button type="submit" class="btn btn-success">Add Goal</button>
  </form>
</div>
```

If we try adding to an input and then deleting it, the respective error messages will show indicating that the user will now be aware of any invalid inputs which improves user experience and also makes sure our application does not break from invalid data.

Wednesday: Submit Form

Submit Form

Submitting forms using ngSubmit

At the bottom of our form, we have a submit button that's supposed to add a new goal to our array but if we click it right now, nothing happens. Let's make it work the way it is supposed to. We want it to add a goal to the goals array when this button is clicked.

To do this, we need to add use the `ngSubmit` directive from angular forms module that we imported earlier. Let's add it to our form:

src/app/goal-form/goal-form.component.html

```
<div class="container-fluid">
  <h2 class="text-center">Create a new Goal</h2>
  <hr>
  <form (ngSubmit)='submitGoal()' #goalForm='ngForm'>
....
```

We have added the `(ngSubmit)` event attribute which calls the `submitGoal()` function once the event is emitted. Let's now write the code to be executed when the `submitGoal()` function is called.

src/app/goal-form/goal-form.component.ts

```
import { Component, OnInit, Output, EventEmitter } from'@angular/core';
...
exportclass GoalFormComponent implements OnInit {
  newGoal = new Goal(0,"","",newDate());
@Output() addGoal = new EventEmitter<Goal>();
  submitGoal(){
this.addGoal.emit(this.newGoal);
  }
...
```

At the top, we have imported the `Output` and `EventEmitter` functions from `@angular/core`. We have then created the `addGoal` event emitter object which is of type `Goal` that will emit an event to the parent component. We have then created the `submitGoal()` function in which we use the emit method and pass in the new goal object we want to create.

Since the `addGoal` event is being emitted to a parent component, we need to make sure the parent component is informed of this event being emitted. We want the goal component to be the parent component in this case, so let's write the code that will handle this event.

src/app/goal/goal.component.html

```
...
<div class="col-md-6">
      <app-goal-form (addGoal)="addNewGoal($event)"></app-goal-form>
 </div>
```

```
...
```

In the template, we catch the `addGoal` event being emitted and define that it should call the `addNewGoal()` function which takes in an event placeholder. Let's now define this function that is supposed to be executed by our parent component.

src/app/goal/goal.component.ts

```
....
export class GoalComponent implements OnInit {
  goals: Goal[] = [
    new Goal(1, 'Watch finding Nemo', 'Find an online version and watch merlin find
his son',new Date(2019,9,14)),
    new Goal(2,'Buy Cookies','I have to buy cookies for the parrot',new
Date(2019,6,9)),
    new Goal(3,'Get new Phone Case','Diana has her birthday coming up soon',new
Date(2019,1,12)),
    new Goal(4,'Get Dog Food','Pupper likes expensive snacks',new
Date(2019,11,18)),
    new Goal(5,'Solve math homework','Damn Math',new Date(2019,2,14)),
    new Goal(6,'Plot my world domination plan','Cause I am an evil overlord',new
Date(2019,3,14)),
  ];
....
  addNewGoal(goal){
    let goalLength = this.goals.length;
    goal.id = goalLength+1;
    goal.completeDate = new Date(goal.completeDate)
    this.goals.push(goal)
  }
....
```

We have created the `addNewGoal()` function that takes a goal object as an argument. We first need to change the `id` property of the goal. We get the length of the array of goals and store it in the variable `goalLength` we then add one to the `goalLength` and set that as the new `id` for the goal. We then set the `completeDate` property of the goal object to a `Date` Object. Lastly, we push the new goal to our array of goals.

If our server is still running, we can now add a new goal to the array and it will display on our application.

Wednesday: Bootstrapping

Bootstrapping

All Angular apps have at least one Angular Module which is the root module that is used to launch the application. By convention, it is called the `AppModule` whose configuration is contained in the *app.module.ts* file.

Imports

app.module.ts

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


import { AppComponent } from './app.component';
import { GoalDetailsComponent } from './goal-details/goal-details.component';
import { GoalComponent } from './goal/goal.component';
import { StrikethroughDirective } from './strikethrough.directive';
import { DateCountPipe } from './date-count.pipe';
import { GoalFormComponent } from './goal-form/goal-form.component';

..........
```

Using the `AppModule` in our application, let's examine it. The first part is the Import section where we import external modules, components and directives that are used in our application.

NgModule decorator

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    GoalDetailsComponent,
    GoalComponent,
    StrikethroughDirective,
    DateCountPipe,
    GoalFormComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

The `@ngModule` decorator identifies the `AppModule` as an `ngModule` class. It has metadata that tells Angular how to run the application.

Declarations array

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    GoalDetailsComponent,
    GoalComponent,
    StrikethroughDirective,
    DateCountPipe,
    GoalFormComponent
  ],
 ......
```

Angular knows what apps belong to the `AppModule` by finding them in the declaration array. We also place custom directives and pipes that we create in the declarations array.
The Angular CLI’s `generate` command adds the generated pipes, components and directives to the declaration array automatically for us.

Imports Array

app.module.ts

```
@NgModule({
.........
imports: [
    BrowserModule,
    FormsModule
  ],
```

In Angular, we group features into specific units called modules. We add a module to the imports array when the application wants to use its features. For example, since our application runs on the browser, we use features provided in the `BrowserModule`. We have also used some features like two-way data binding which are provided by the formsModule in our application

Providers Array

We can deliver services to different parts of an application using `dependency injection`. We use the providers array to register the different services needed by our application. We do not have any services yet that is why our array is empty. We'll learn about services later on and we'll see them being registered in this array.

Bootstrapping Array

app.module.ts

```
@NgModule({
.........
 bootstrap: [AppComponent]
 .......
```

We launch the application by bootstrapping the root component which is the `AppComponent`.

Bootstrapping an Application

main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
```

```
if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

We bootstrap an Angular Application in the *main.ts* file. Here angular dynamically creates a browser platform and bootstraps the root Module which is the `AppModule`.The process of bootstrapping sets up an execution environment finding the `AppComponent` from the `bootstrap` array in the `AppModule` and creating an instance of it within its selector tag in the *index.html* file.

We have now seen why the AppModule is important in our application and where it is used.

Wednesday: Deployment

Deploying Angular Apps

After we create an application, we can deploy it to a remote server so that users can interact with it online. There are various ways of deploying applications and the one that has lots of advantages is Cloud Hosting. There are many cloud platforms which offer hosting services such as AWS(Amazon Web Services), Google Cloud Platform and Microsoft Azure. Most of these cloud hosting platforms offer payment plans for hosting but in our case, we will use Github which is free.

Let's, therefore, see how we deploy our angular application and publish it to gh-pages on Github. At this point, we all have Github accounts so let's do the first thing.

1. Create Github repo

If you had not created a Github repository prior to this, go ahead and create a repo in your Github account. After creating the repo, commit all your changes and push them to the repository you have created.

If you had created your repo prior to this, also commit and push all your changes to your Github repository.

2. Install angular-cli-ghpages

We have a tool at our disposal that we can use to easily deploy our angular app to gh-pages, the angular-cli-pages tool. We use this tool as a command on the angular CLI for the purpose of deployment. To install it, let us run this command on our terminal:

```
$ npm i angular-cli-ghpages --save-dev
```

This command has installed angular-cli-ghpages globally in our operating system. We, therefore, don't have to install it again in case we need to use it.

3. Run build

We need to build our angular app for use in production. This simply means configuring our app to be useable on a remote server. While we were creating our app, we were serving our application on our local development server http://localhost:4200/ (Links to an external site.). The configuration for the local server will not serve the application when it is deployed on the remote server, gh-pages, which is why we need to build our app. Let's run this command on our terminal to build the application:

```
$ ng build --prod --base-href "https://GithubUserName.github.io/GithubRepoName/"
```

Make sure you put in your Github username in GithubUserName and the repository name you created in GithubRepoName.

4. Deploy to gh-pages

It's now time to use angular-cli-ghpages. Let's run this command in our terminal:

```
$ npx angular-cli-ghpages --dir=dist/Project-name
```

Make sure you put the name of the project in the place of Project-name. You can find this in the *angular.json* file under *defaultProject* which is at the bottom of the file.

This command has created a gh-pages branch on Github for us. It has then read all the changes in our remote branch and pushed these changes to gh-pages while maintaining the build configuration that will ensure our application works while on the remote server. If we visit the URL we created earlier on our terminal, https://GithubUserName.github.io/GithubRepoName/,we can see our application running remotely, which means it has successfully been published on gh-pages.

Thursday: Diary

The Diary

This week you will create a personal diary application where you can write down your thoughts.

User Story

You as the user you should be able to:

1. Create a new diary entry.
2. Set the date for a new entry.
3. Delete the Diary entry.
4. Able to highlight titles in the diary entry

Technical Requirements

1. Use the Angular CLI to create the application.
2. Create a clear README file.
3. Have a minimum of 15 clear commit messages.

4. Create a diary model class.
5. Create a custom directive to highlight the titles.
6. Create custom pipe to customize dates.
7. Have form component with the necessary Inputs.
8. Your app should be deployed.

Quotes

This weeks project is to create an application where users can create quotes and have those quotes voted on whether they are terrible or are inspirational.

User Requirements

As a user I would like to:

1. Create a new quote.
2. Upvote or downvote a quote.
3. Delete a quote
4. See the number of upvotes and number of downvotes for each quote.

Project Objectives

1. Your Project should be created using the Angular CLI.
2. Your project should contain a custom model class for the quote
3. Your project should have a custom **directive** to highlight the quote that has the highest upvotes.
4. Your project should contain a form component that has input sections for the quote, the author, and the person who submitted the quote.
5. Each quote must begin with an initial upvote and downvote value of `0`
6. Your project should have a component to display the quote and a component to display quote details. For example, the quote details could have the author of the quote and the votes it has.
7. Your project should contain a custom `pipe` that shows how much time has passed since the quote was created.

Technical Requirements

1. Your app must be of high quality in terms of basic functionality and it should be visually appealing.
2. You must have a well-documented README document that gives a brief description of your application, How it works, How to install your application, A link to the deployed website and a license section.
3. Your project must be deployed and a link to the deployed website should be submitted.
4. Your project should have regular, well-defined commits with clear messages.

Rubric

Quotes Project Objectives

Quotes Project Objectives

Criteria	Ratings		Pts
This criterion is linked to a learning outcome A well documented readme is the first documentation any developer should use	1.0 Pts A well documented Readme file on Github A project README that includes: - project or program name - author name - description of project - project setup instructions - link to live site on GitHub Pages - copyright and license information	0.0 Pts No Readme file on Github	1.0 pts
This criterion is linked to a learning outcome Project is in a polished, portfolio-quality state. Suggestions for what this can mean: Intuitive, easy to follow layout. Simple yet polished styling. Form field labels where appropriate. Form fields that are validated correctly, and get cleared after submitting. Detailed, well put together readme. No typos. And much more.	1.0 Pts Went above and beyond in the visual aspect Suggestions for what this can mean: Intuitive, easy to follow layout. Simple yet polished styling. Form field labels where appropriate. Form fields that are validated correctly, and get cleared after submitting. Detailed, well put together readme. No typos. And much more.	0.0 Pts No effort to make it visually pleasant	1.0 pts
This criterion is linked to a learning outcome Does the project work functionally achieving the expected objective?	2.0 Pts Yes, the project works as expected All objectives are functionally met in the deployed project	0.0 Pts No, the project does not work as specified in the objectives	2.0 pts
This criterion is linked to a learning outcome Commits are made regularly with clear messages associated with them	1.0 Pts 20 + commits in the project with well detailed commit messages	0.0 Pts Less than 10 Github commits present in the project	1.0 pts
This criterion is linked to a learning outcome Project is link and description is provided on Github repository	1.0 Pts Project description is present This is used as a blurb of the project, explains what the project is bout and the linked to a deployed site if present	0.0 Pts Did not make use of Github gh-pages	1.0 pts
This criterion is linked to a learning outcome Page incorporates a custom-made stylesheet			2.0 pts

Criteria	Ratings	Pts
2.0 Pts Used custom CSS stylesheet All objectives are functionally met in the deployed project	0.0 Pts Used custom CSS stylesheet	
This criterion is linked to a learning outcome Custom model class The project contains a custom model class for the quote	2.0 Pts Full marks; 0.0 Pts no custom model class	2.0 pts
This criterion is linked to a learning outcome Project has a custom directive to highlight the quote that has the highest upvotes.	3.0 Pts Full marks; 0.0 Pts NO custom directive	3.0 pts
This criterion is linked to a learning outcome Your project must contain a form component that has input sections for the quote,the author,the person who submitted the quote	2.0 Pts Full marks; 0.0 Pts No Form Component	2.0 pts
This criterion is linked to a learning outcome Your project must have a component to display the quote and a component to display quote details such as the person who submitted the quote and the votes it has	3.0 Pts Full marks; 0.0 Pts Your project does not have a component to display the quote and a component to display quote details such as the person who submitted the quote and the votes it has	3.0 pts
This criterion is linked to a learning outcome Each quote must begin with an initial upvote and downvote value of 0	1.0 Pts Full marks; 0.0 Pts No Initial value of 0	1.0 pts
This criterion is linked to a learning outcome Your project must contain a custom pipe that shows how much time has passed since the quote was created	3.0 Pts Full marks; 0.0 Pts No Custom Pipe	3.0 pts

Total points: 22.0

Monday: Angular Week 2 Objectives

This week we'll dive slightly deeper into the Angular framework by learning a bit more complex and interesting concepts.

We will start by learning a concept called *dependency injection*. By understanding this concept, it will help us create something called a *service* which will, in turn, allow us to organize and share code across our application.

From there, we will also to learn about *the Http Client* module that will enable us to use APIs in our Angular application.

Finally, we will finish off by learning how to use a router to create and add dynamic pages to our application. Here, we'll learn how to pass data and information between different routes.

By the end of the week, you will be able to;

1. Consume an API to display random quotes in our application
2. Understand the concept of Dependency Injection and how to use it
3. Use routing modules create dynamic pages in the application.

Independent Project Objectives

In this section, you mentor will review your project based on the following objectives:

1. Have a well-designed landing page that displays Github information such as username, profile photo and a list of repositories.
2. Use Http Client services to access GitHub API.
3. Use two classes in your project.
4. Implement Http requests to search for GitHub users and GitHub repositories.
5. Have a routing module.
6. Implement custom directives and custom pipes in your project.
7. Polished and portfolio-quality.
8. Have a well-documented README file.
9. The project must be deployed and link submitted.

Monday: Dependency Injections

Dependency Injection

Dependency injection is a design pattern in which a class asks for dependencies from external sources instead of creating them itself. The reason we use dependency injection is because it separates the creation of an object from its usage, which enables us to replace dependencies without changing the class that uses them. This means that a class concentrates on fulfilling its responsibilities instead of creating the objects that help it fulfill those responsibilities.

What does this really mean? Let's do it practically. Let us create a folder outside our Goals project anywhere on your laptop and call it Car. Inside this folder, let's create 4 files, car.ts, engine.ts, wheels.ts and main.ts. We're going to familiarize ourselves with DI(Dependency Injection) with the concept of a car.

Let's write code for a class that allows us to create an instance of a car. This class would look like this:

Car/car.ts

```
import { Engine } from './engine';
import { Wheels } from './wheels';
export class Car{
  engine: Engine;
  wheels: Wheels;
  constructor(){
    this.engine = new Engine();
    this.wheels = new Wheels();
  }
  startEngine(){
    this.engine.start();
  }
}
```

At the top, we have imported the objects/dependencies we need to create a Car. Inside the car class, we have created properties engine and wheels and assigned them their types respectively. In the constructor function, we have created instances of these dependencies using the new keyword. We have then created a method startEngine() which calls another function in our engine dependency. Let's create this function in engine dependency.

Car/engine.ts

```
export class Engine{

  start(){
    console.log("Vroooooooom!");
  }
}
```

We have created a start function which simply logs a string on our console. We will not be using our wheels but let's create an empty class and assume that it's also a dependency that has its own properties and methods just like our engine.

Car/wheels.ts

```
export class Wheels{

}
```

We'll use our main.ts file to run this car app. Let's write the code to run this app in this file.

Car/main.ts

```
import { Car } from './car';
import { Engine } from './engine';
import { Wheels } from './wheels';
function main(){
  let car = new Car();
  car.startEngine();
}
main();
```

At the top, we have imported our car, engine, and wheels and then created a function main(). We have created the car instance and called the startEngine() method in the function and at the bottom, we have called our main() function so it can execute this code.

Since we have node installed, we can run this code on our terminal. Let's transpile this typescript code to javascript code using the typescript transpiler on our terminal:

```
$ tsc main.ts
```

Now let's run the transpiled javascript code directly on our terminal with node. If you recall, we installed node so we could execute javascript code directly from our terminal instead of using the browser console, so let's go ahead and do it:

```
$ node main.js
```

This command creates our car and starts the engine for us which is great, we can see the output in the terminal. If we, however, wanted to use this car in another environment that has different dependencies, it would be difficult because the car creates the dependencies that it needs for itself. For example, in the state our car class is in right now, if we want to add doors to the car, we would have to add the code to make it define doors and still define this property in the constructor function. The same would apply if we wanted to change anything that exists in the car already. Point is, our code will be hard to maintain and scale, which may also end up making it messy and prone to errors.

That's where Dependency Injection comes in handy. It relieves our Car class the responsibility of creating the dependencies it needs and makes it just consume these dependencies from external sources.

Let's implement this in our car.

Car/car.ts

```
import { Engine } from './engine';
import { Wheels } from './wheels';
export class Car{
  constructor(private engine: Engine,private wheels: Wheels){
  }
  startEngine(){
    this.engine.start();
  }
}
```

At the top, we still have the imports for our dependencies. Our constructor function no longer builds the dependencies the car class needs. This means that the car class consumes its dependencies, the engine and wheels class. We have used the private keyword to make these properties available only inside the class.

Inside the constructor function, we still call our startEngine() method. When we refactor our code like this, the Car class does not know how to create its dependencies which is amazing because we can now alter the dependencies that the class consumes more easily now. We now need to change the code in our main.ts file so we can fire up the engines again.

Car/main.ts

```
import { Car } from './car';
import { Engine } from './engine';
import { Wheels } from './wheels';
function main(){
  let engine = new Engine();
  let wheels = new Wheels()
  let car = new Car(engine,wheels);
  car.startEngine();
}
main();
```

At the top, we have still imported our classes. Inside the main function, we are now creating instances of engine, wheels, and car. When we create the car instance now, we specify that it should consume the engine and wheels dependencies that we have just instantiated. We end up by starting our engine and calling our main function.

Let's transpile this code in our terminal again to see the effect it has:

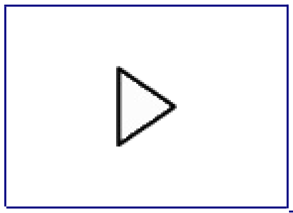
```
$ tsc main.ts
```

Now let's run our JS code on our terminal using node:

```
$ node main.js
```

Amazing, our engines still fire up! This is how Dependency Injection works. The car class does not create the dependencies it needs, it consumes them in the constructor. Now it is the *main.ts* file that creates these dependencies and also creates an instance of the car, so the car can now consume these dependencies that have been created outside it and still work as expected. This is impactful because we can create a custom engine for each car rather than having all cars create the same default engine in their constructor.

Angular has its own inbuilt framework for dependency injection which we will use and that is how it works. This DI framework enables us to create an injector with which we can register some classes and it figures out how to create these dependencies. Our job would be to ask the injector for the created dependencies. Dependency injection is good practice in so many languages and frameworks, not only in Angular. If you need more resources to understand the concept of dependency injection, feel free to look for them online, and also take a look at this video (Links to an external site.)

.

Monday: Creating a service

Creating a Service

Angular implements Dependency Injection using **services**. A **service** shares data and information among classes that don't know each other.

Our goals are stored in an array in a file that's in our project right now. This is because it's best practice that a component does not fetch or save data knowingly. A component should focus on presenting data and delegate data access to a service. This is why we moved our goals array to a new file, *goals.ts*. Let's also note that this data can come from a different source, like a database, a remote server or an API. Let's create a service that will share our goals among classes that need it.

In our terminal, let's use this command to create the service:

```
$ ng generate service goal-service/goal
```

This command creates a folder named goal-service and inside it creates the service class file and the service test file.

src/app/goal-service/goal.service.ts

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class GoalService {
  constructor() { }
}
```

The class file has the **Injectable** symbol imported and the class annotation uses it as a decorator. This marks this class as one that participates in dependency injection. The decorator accepts metadata for the class which means our service can also have its own dependencies. The **providedIn** property has a value **'root'** which means that this service is injectable throughout the application.

Let's write code to make this service access our goals.

src/app/goal-service/goal.service.ts

```
import { Injectable } from '@angular/core';
import { Goals } from '../goals';
@Injectable({
  providedIn: 'root'
})
export class GoalService {
  getGoals(){
    return Goals
  }
  constructor() { }
}
```

At the top, we have imported our Goals array. In the service class, we have created a method getGoals() which returns our Goals array. We now have a service that gets goals for components or other services that need it.

Monday: Register a Service

Register a Service

Now that we have a service, Angular needs to know that this service is available so it can inject it into components or other services that need it. To do this, we need to register a **provider**. A **provider** is something that can create or deliver a service, in our case, it instantiates the **GoalService** class to provide the service.

We register our **GoalService** as the provider for this service with an **injector**. An **injector** is an object responsible for choosing and injecting the provider wherever it's required.

The Angular CLI, by default, registers a provider with the **root** injector after running the command $ ng generate service <service-name>. This happens when the CLI automatically supplies the provider metadata in the **@Injectable** decorator.

src/app/goal-service/goal.service.ts

```
....
@Injectable({
  providedIn: 'root'
})
....
```

The **providedIn** metadata specifies that our service is available in the application at root level. When you provide a service at root level, Angular creates a single shared instance of the service and injects it into any class that asks for it. Registering a provider in the **@Injectable** metadata is best practice since it allows Angular to optimize the app by removing the service if it turns out not to be used at all.

Since our service is registered with a provider that injects it at root level, our service is now readily available throughout the application so we can inject it to any component or class that requires it.

Let's inject it into our goal component.

src/app/goal/goal.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Goal } from '../goal';
import { GoalService } from '../goal-service/goal.service';
....
export class GoalComponent implements OnInit {
  goals:Goal[];
  constructor(goalService:GoalService) {
    this.goals = goalService.getGoals()
  }
....
}
```

At the top, we import our **GoalService** and also get rid of the **Goals** array since it is the service that will inject the goals from now on. In the component class, we create a property **goals** and assign it a type. We then tweak the Goal component class to consume the GoalService with the constructor. Inside the constructor function, we use the **getGoals()** method of the goal service to supply the goals.

If your local server is still running, you'll notice that our application still displays our goals which means that the service works as expected.

alert Service

At the moment, when the user deletes a goal, a prompt asks the user to confirm whether or not they want to delete the goal. Let's create another service that alerts the user once they delete a goal.

On our terminals, let's create a new service with this command:

```
$ ng generate service alert-service/alert
```

Let's write the code to describe what the service will do:

src/app/alert-service/alert.service,ts

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class AlertService {
  alertMe(message:string){
    alert(message)
  }
  constructor() { }
}
```

Our service is available at root level so we can inject it anywhere we like. Inside the service class, we have created the **alertMe()** method which accepts a message of string type. The method should use the alert function in javascript to display the message it receives.

Let's inject this service in our goal component.

src/app/goal/goal.component.ts

```
...
import { AlertService } from '../alert-service/alert.service';
...
export class GoalComponent implements OnInit {
  goals:Goal[];
  alertService:AlertService;
....
  deleteGoal(isComplete, index){
    if (isComplete) {
      let toDelete = confirm(`Are you sure you want to delete $
{this.goals[index].name}?`)
      if (toDelete){
        this.goals.splice(index,1)
        this.alertService.alertMe("The goal has been deleted")
      }
    }
  }
....
  constructor(goalService:GoalService, alertService:AlertService) {
    this.goals = goalService.getGoals()
    this.alertService = alertService;
  }
...
}
```

At the top, we have imported the **AlertService**. Inside the component class, we have created a property **alertService** and assigned it our AlertService type. In the **deleteGoal()** function, we have added code that uses the alertMe() method from the alert service to display the message inside after the user has confirmed to delete a goal.

To make the service available in the component, we have added it to the constructor function and instantiated it inside the constructor function. If our server is still running, we can delete a goal to see the alert service at work.

Monday :HttpClient

HttpClient

HttpClient is a mechanism used by Angular to enable communication to remote servers and backend services using the HTTP protocol. We will use a random quote API which is a backend service in our app to display quotes from the API on our app. This will give us an understanding of how to use HttpClient in Angular to access backend services and remote servers. To make HttpClient available everywhere in the app, we import it in our root modules as follows.

src/app/app.module.ts

```
...
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
...
  imports: [
...
    FormsModule,
    HttpClientModule
  ],
...
})
```

At the top, we have imported the **HttpClientModule** and added it to the imports array. We are going to use this module to access a random quote API and display the quotes on our app. If you paste this http://quotes.stormconsultancy.co.uk/random.json (Links to an external site.) link in your browser's address bar, you'll see a random quote that changes every time you refresh the webpage. You can install JSONView (Links to an external site.), or JSONFormatter (Links to an external site.) from chrome web store to make the response more readable. The response looks something like this but the quote and author may vary:

{

- author: "James O. Coplien",
- id: 23,
- quote: "You should name a variable using the same care with which you name a first-born child.",
- permalink: "http://quotes.stormconsultancy.co.uk/quotes/23 (Links to an external site.)"

}

The request returns a response that has four properties, author, id, quote, and permalink. We will make a request to the API and display the quotes in our app. We will display the author and quote only in our app. We, therefore, need to make Angular sieve out what we need by creating a quote class that will help us create quote instances. To create a class, let's execute this command in our terminal:

```
$ ng generate class quote-class/quote
```

This command creates a folder quote-class, a class file, quote.ts, and a test file quote.spec.ts. Inside the class file, let's define how we want our quote instances created.

src/app/quote-class/quote.ts

```
export class Quote {
  constructor(public author:string, public quote:string ){}
}
```

We have created a constructor inside the Quote class and defined two public properties, author and quote, both of the string type.

We can now make a HTTP request to the API in our goal component

src/app/goal/goal.component.ts

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
...
import { Quote } from '../quote-class/quote';
....
export class GoalComponent implements OnInit {

  goals:Goal[];
  alertService:AlertService;
  quote:Quote;
....
  constructor(goalService:GoalService, alertService:AlertService, private
http:HttpClient) {
    this.goals = goalService.getGoals()
    this.alertService = alertService;
  }
  ngOnInit() {
    interface ApiResponse{
      author:string;
      quote:string;
    }

this.http.get<ApiResponse>("http://quotes.stormconsultancy.co.uk/random.json").subs
cribe(data=>{
      // Succesful API request
      this.quote = new Quote(data.author, data.quote)
    })
  }
}
```

At the top, we have imported HttpClient and the Quote class. Inside the component class, we have created a property quote an assigned it the type Quote. In the constructor function, we have created a property http which is of the type HttpClient. Before we make the request, we need to inform Angular the kind of response we'll receive from the API by defining an interface which we have named ApiResponse. Inside the interface, we have specified that we'll be expecting a property author and quote which are both of the type string. We have then made a request to the API with the get function passing in the API URL accompanied by the interface for the data we expect to receive. We have then called the subscribe function which has a data function that is executed when the request is successful. We then create a new quote instance with the properties we get from the response.

Let's display this quote on our app.

src/app/goal/goal.component.html

```
....
<p *ngIf='goals.length > 5'>You have too many goals</p>
  <div class="row">
    <blockquote class="blockquote text-center">
      <p class="mb-0">{{quote.quote}}</p>
      <footer class="blockquote-footer"><cite>{{quote.author}}</cite></footer>
    </blockquote>
  </div>
</div>
```

We have displayed our quote in a bootstrap blockquote, and if our server is still running, we can see a random quote displayed on our app below our goals.

Monday: Error Handling

Error Handling

What would happen if we make a bad request to the API and we get no response, or even worse, the servers are down and not working? We need to prepare our application for such an event so that it is not blank or broken. The subscribe function has an err function that gives us the capacity to handle errors. Let's use it to handle the occurrence of an error in which we get no response from the API.

src/app/goal/goal.component.ts

```
...
export class GoalComponent implements OnInit {

this.http.get<ApiResponse>("http://quotes.stormconsultancy.co.uk/random.json").subs
cribe(data=>{
      // Succesful API request
      this.quote = new Quote(data.author, data.quote)
    },err=>{
        this.quote = new Quote("Winston Churchill","Never never give up!")
        console.log("An error occurred")
    })
...
```

We have added the err function and specified the quote instance that should be created when we get no response and the error message to be logged in the console. Try messing with the URL by in this file by adding or omitting a character, and you'll see the err function handling the response for us.

Using Loaders

Since we are making requests to a remote server, it would be necessary to convince the user that there is something happening in the background, as the app awaits a response. This enhances the user experience of the app. We will do this using a loader (Links to an external site.) which is a type of a progress bar. It creates a visual animation in our app that convinces the users that there is something happening. To install this loader module, let's run this command in our terminal:

```
$ npm install --save @ngx-progressbar/core@3.0.2 @ngx-progressbar/http-client@3.0.2
```

This module needs a supplementary module that looks at observable data for the loader to work properly called rxjs-compat. Let's install it using the following command:

```
$ npm install --save rxjs-compat
```

Let's now add it to our root modules of our app to make it available for use.

src/app/app.module.ts

```
...
import { NgProgressModule } from '@ngx-progressbar/core';
import { NgProgressHttpClientModule } from '@ngx-progressbar/http-client';
...
imports: [
    ....
    NgProgressModule.forRoot(),
    NgProgressHttpClientModule
],
```

We have imported the normal loader and the loader that listens for our HTTP requests from the app and automatically displays progress according to our apps requests. We have then added both to the **imports** array. The forRoot() method makes the loader available and configurable at the root level of our app and supplies the dependencies it needs. We don't need any more configurations since the

NgProgressHttpClientModule works with the requests made from our app. We only need to display the loader in our template.

src/app/goal/goal.component.html

```
<div class="container">
  <ng-progress></ng-progress>
  <h1>MY GOALS</h1>
.....
```

We have added tags in the template placed at the top where we want to see our progress bar. When we run our server and go to our app in the browser, we see the loader appear every time we refresh the page showing the progress of every new request.

Monday: HTTP service

Create an HTTP service

We earlier saw that services are very helpful in dependency injection. To clean up our app's code, let's create a service that will deliver our quotes to any component that needs them.

Environment Variables

As we clean up our code, it is also necessary and best practice to keep crucial data like API keys, database passwords, e.t.c, away from prying eyes in our apps. The reason is that when we push this kind of data, it puts our app at the risk of malicious attacks since any user can see and interfere with your code on Github if you're not using a private repository, making the app insecure for users to interact with. To hide this kind of data, we put it in a file or folder which we then include in a **gitignore** file and when we push our code to production, all the folders or files in the **gitignore** file are not exposed to users and are not being tracked on version control. In our case, we will use the environment file in the environments folder to hide our API's url. Let's open this file and include this code in the file:

src/environments/environment.ts

```
export const environment = {
  production: false,
  apiUrl:"http://quotes.stormconsultancy.co.uk/random.json"
};
```

We have created a property apiUrl and assigned it to our random quotes API url. Please note that we have put the API url in this file for practice, assuming that it is as crucial as a database password. This means when we push our code to Github and deploy our app, the url will be hidden and therefore requests to the API will not happen. This also implies that we will always receive an error message and our error quote in deployment. In future, we'll hide more crucial data, unlike the url in this case.

To hide this file, let's go to our .gitignore file and add the file there.

Goals/.gitignore

```
....
environment.ts
```

Our API url is now hidden whenever we push our code to Github. In our terminals, let's execute this command to create the service for our quotes:

```
$ ng generate service quote-http/quote-request
```

Let's write the code for our service in the service class.

HTTP requests using Promises

A promise is an object representing the eventual completion or failure of an asynchronous process. Our asynchronous process here is the request we are making to the API.

Let's use a promise in our service.

src/app/quote-http/quote-request.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import {environment } from '../../environments/environment';
import { Quote } from '../quote-class/quote';
@Injectable({
  providedIn: 'root'
})
export class QuoteRequestService {
  quote: Quote;
  constructor(private http:HttpClient) {
    this.quote = new Quote("","");
   }
   quoteRequest(){
     interface ApiResponse{
       quote:string;
       author:string;
     }
     let promise = new Promise((resolve,reject)=>{
       this.http.get<ApiResponse>(environment.apiUrl).toPromise().then(response=>{
         this.quote.quote = response.quote
         this.quote.author = response.author
         resolve()
       },
       error=>{
         this.quote.quote = "Never, never, never give up"
         this.quote.author = "Winston Churchill"
         reject(error)
       })
     })
     return promise
   }
}
```

At the top, we have imported HttpClient to enable us to make a request to the API, the Quote class and the environment class in which we put our API url. We have then created a property quote and assigned it the type of our Quote class initializing it with empty strings inside the constructor function. We have also injected a private http property of the type HttpClient in the constructor.

We have then defined a quoteRequest() method that defines the ApiResponse interface and the promise instance. We have then used the get function and passed in the apiUrl from the environment object with the interface. We have used toPromise() to convert the Http Request to a promise. We have called the then function and passed in the response and error functions as arguments. The response function is called when the HTTP request is successful and returns a response. If successful, we update the properties of the quote instance with values from the response and call the resolve function. If we encounter an error, we have passed in default values for creating a quote instance in the error function which is called when there is an error.

Let's now call the service in our goal component.

src/app/goal/goal.component.ts

```
....
import { GoalService } from '../goal-service/goal.service';
import { AlertService } from '../alert-service/alert.service';
import { QuoteRequestService } from '../quote-http/quote-request.service';
export class GoalComponent implements OnInit {
...
  quote:Quote;
...
  constructor(goalService:GoalService, alertService:AlertService, private
quoteService:QuoteRequestService) {
    this.goals = goalService.getGoals()
    this.alertService = alertService;
  }
  ngOnInit() {
    this.quoteService.quoteRequest()
    this.quote = this.quoteService.quote
  }
```

We have imported our QuoteRequestService at the top and then injected it into our constructor. Inside the ngOnInit lifecycle hook, we have called the quoteRequest() method from the service and created a quote instance with the promise object we will receive from the service. If we run our server at this point, our quotes will be displayed on the application as usual but using a service this time round.

Monday:Practice: Meal Tracker

Goal: Continue practising creating applications with Angular Cli while familiarizing yourself with the process of using services.

Warm up

- What is Dependency Injection? Why is it important?
- What are services in angular 6?
- What is Http service used for?

Code

Meal Tracker

Create a meal tracking application where the user gets to keep track of the food they consume in each day. Here are some user stories to help you get started;

1. As a user, I want to log a food I have eaten by submitting a form with food name, calories and details.
2. As a user, I want to view a list of foods I have logged.
3. As a user, I want options to view all foods, only high-calorie foods (more than 500 calories), or only lower-calorie foods (less than 500 calories).
4. I want to click a food to edit its name, details or calories (in case I decide to pretend my fries were 100 calories instead of 365).

Make sure you use at least one service to help you retrieve the meal from the meal array.

Tuesday: Routing

Routing

In Angular, routing enables us to navigate from one view to another. This means users can click links or buttons that change whatever is displayed on the app. In our app, we will create a navigation bar, that will enable us to change to different views in order to understand how routing works.

Let's start by creating a component that will give more information about the app. We will call it the *about* component. On our terminals, let's execute this command:

```
$ ng generate component about
```

In the HTML template file of our about component, let's add a description of our app.

src/app/about.about.component.html

```
<div class="jumbotron">
  <h1 class="display-4 text-center">About Goals App</h1>
  <hr class="my-4">
  <p class="lead text-center">An Angular app that let's you create your goals and
gives you quotes from the world of computing!</p>
</div>
```

We have used a bootstrap jumbotron and written a description of our app. Feel free to describe your app in your own words and style.

AppRoutingModule

To implement routing in Angular, we use a module known as the AppRoutingModule. We import it and feed it with simple instructions in the form of code and it does the navigation for us. Interestingly, when we generate a new app in angular with the command ng new <app-name>, this is one of the modules that the Angular CLI automatically adds to our app for us. If we recall using this command when creating our Goals app, the first prompt was whether or not we want to use Angular routing, to which we agreed. This is what made the CLI add this module for us. Under the directory src/app, there is a file named app-routing.module.ts which the CLI created for us and looks like this:

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

This file has everything we will need to create routes in our app, which we will do soon. Checking our root level modules file, *app.module.ts*, we see the **AppRoutingModule** module is already imported and added to the @NgModule **imports** array meaning we can use it in our app.

We configure our routes using **Routes** which is a service that presents a given component view for a specific URL. We have imported it at the top of our routing module file.

Adding Routes

Routes tell the router which view to display when a user clicks a link or pastes a URL in the browser address bar. A typical Angular route has two properties, a path which is a string that matches the URL in the browser address bar and a component which is the component that the router should create when navigating to this route.

Let us add two routes, one that will navigate to our goals component and another to the about component we have just created.

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { GoalComponent } from './goal/goal.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
  { path: 'goals', component: GoalComponent},
  { path: 'about', component: AboutComponent},
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
```

```
export class AppRoutingModule { }
```

We have first imported our two components at the top of our file. Inside our routes array, we have defined the path and component for each route. The RouterModule.forRoot(routes) in the imports initializes the router and gets it listening to the browser for location changes. The forRoot() method makes the router configurable at root level and supplies the service providers and directives needed for routing. This is the same method we used when we configured the loader because it serves the same purpose, making a module available at root level and supplying any dependencies it needs.

RouterOutlet

It is now the router's job to take care of what is being displayed on the application. RouterOutlet is a directive of the RoutingModule that has a selector <router-outlet> which handles routing for us. Let's use this selector and put the router in charge of displaying different views.

src/app/app.component.html

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

We have replaced the <app-goal> selector with the <router-outlet> selector because the router will be responsible for views from now on. If we keep the <app-goal> selector, our app will display the goal component and its child components only, which beats our purpose for creating routes.

Let's run our server and try out these routes. On the browser address bar, put in your routes and see the difference, for example, http://localhost:4200/goals (Links to an external site.) to display the view for the goals, and http://localhost:4200/about (Links to an external site.) to change the view to our about component. Works, right? Our router is functional which is great!

Tuesday: Router-Outlet

RouterLink

It's annoying if a user has to keep putting in URLs in the browser address bar to navigate to a different view. This is an example of bad user experience. Let's create a navbar with which users can quickly switch to different views with this command on our terminal:

```
$ ng generate component navbar
```

We will use a bootstrap navbar for this component.

src/app/navbar/navbar.component.html

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand"><b>GOALS APP</b></a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup" aria-
expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
    <div class="navbar-nav">
      <a class="nav-item nav-link active">Goals<span class="sr-
only">(current)</span></a>
      <a class="nav-item nav-link">About</a>
    </div>
  </div>
</nav>
```

We have created a navbar with two elements, Goals and About. To display our navbar component whether or not we have a view, let's add its selector to our root template file.

src/app/app.component.html

```
<app-navbar></app-navbar>
<router-outlet></router-outlet>
```

When we run our server, we can see our navbar displayed. If we click any of the navbar links however, there is no change in the views.

RouterLink

To make these navigation links work, we use the routerLink attribute which is from the RouterLink directive found in the RouterModule. Remember directives? Let's make our links work:

src/app/navbar/navbar.component.html

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" routerLink="goals"><b>GOALS APP</b></a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup" aria-
expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
    <div class="navbar-nav">
      <a class="nav-item nav-link active" routerLink="goals">Goals<span class="sr-
only">(current)</span></a>
      <a class="nav-item nav-link" routerLink="/about">About</a>
    </div>
  </div>
</nav>
```

We have added the attribute routerLink to the navbar-brand element, Goals navigation link and About navigation link and equated each to the exact route whose view we want to display. The RouterLink directive will now turn user clicks on the elements with the routerLink attribute into router navigations. If our server is still running, we can click on these navbar elements and we'll see the views change respectively to the routes we have assigned them.

RouterLinkActive

The RouterLinkActive directive also has a routerLinkActive attribute which changes the CSS properties of an active route. This improves the user experience in our app by visually informing users that they are on a specific page(or a specific view). Let's add it to the elements in our navbar.

src/app/navbar/navbar.component.html

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" routerLink="/goals"><b>GOALS APP</b></a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup" aria-
expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
    <div class="navbar-nav">
      <a class="nav-item nav-link active" routerLink="goals"
routerLinkActive='active'>Goals<span class="sr-only">(current)</span></a>
      <a class="nav-item nav-link" routerLink="/about"
routerLinkActive='active'>About</a>
    </div>
  </div>
</nav>
```

We have added the routerLinkActive attribute to all our navbar elements that navigate to a specific route and equated it to active. Any given routerLink will be active only if its URL is the exact match of the current URL. To test out this feature, click any of these links on the navbar and see their visual appearance change.

Tuesday: 404 Pages

404 Pages

We have all experienced times when we are web browsing and the browser responds with a 404 error of not finding the resource we are looking for which looks similar to this:

This is google's 404 page and different applications have their own customized 404 pages. It's very informative to a user as opposed to having a blank page. As much as it might be disappointing to the user, it improves the user experience. In severe cases, it keeps our application from breaking when a user requests for a resource that is generated by the app but isn't present and the occurrence of that error isn't mitigated by the app. Let's create a component for error pages in our application with this command on our terminal:

```
$ ng generate component not-found
```

In the template file of this component, let's include this code:

src/app/not-found/not-found.component.html

```
<div class="alert alert-danger" role="alert">
  <h1 class="text-center"><b>Sorry, we couldn't find what you're looking for</
b></h1>
  <hr>
  <p class="text-center">Please check your URL!</p>
</div>
```

We have used a bootstrap alert component in the template file. We now need to inform our router to direct users to this component if they look for a resource that isn't found in the app.

src/app/app-routing.module.ts

```
...
import { NotFoundComponent } from './not-found/not-found.component';
const routes: Routes = [
  { path: 'goals', component: GoalComponent},
  { path: 'about', component: AboutComponent},
  { path:'**', component:NotFoundComponent},
];
...
```

At the top, we have imported our NotFoundComponent. We have then defined a route whose path has wildcards and directed this path to the NotFoundComponent. The wildcards define a route that is not present in our routes array. So in the case a user tries to look up a route that does not exist in the routes of our app, they will be taken to the not-found component. Let's fire up our servers and try requesting a route in the address bar that doesn't exist in our app to see our apps error page working.

Redirect

None of our routes caters for an empty path in our app. This is why when we load the path [http://localhost:4200 (Links to an external site.)](http://localhost:4200), only the navbar shows on our app. We can implement a redirect in our app so that when a user loads an empty path, they don't have an empty screen but instead see the goals. Let's configure a route for empty paths.

src/app/app-routing.module.ts

```
...
import { NotFoundComponent } from './not-found/not-found.component';
const routes: Routes = [
  { path: 'goals', component: GoalComponent},
  { path: 'about', component: AboutComponent},
  { path:'**', component:NotFoundComponent},
  { path: '', redirectTo:"/goals", pathMatch:"full"},
];
...
```

We have created an empty path with the redirectTo property that points to the path of the GoalComponent. A redirect requires the pathMatch property which tells the router how to match the URL to with the path given. In our case, the router will only redirect to the /goals path if the URL given is empty ' '. If our server is still running and we load our application, we are redirected to the GoalComponent instantly.

Tuesday: Passing Parameters

Passing Parameters

We can make one more change in our application. We can have the details of a goal displayed separately. This will help us learn how to pass parameters in a route. Let's create a route that maps to a specific goal:

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { GoalComponent } from './goal/goal.component';
import { AboutComponent } from './about/about.component';
import { NotFoundComponent } from './not-found/not-found.component';
import { GoalDetailComponent } from './goal-detail/goal-detail.component';
const routes: Routes = [
  { path: 'goals', component: GoalComponent},
  { path: 'about', component: AboutComponent},
  { path: 'goals/:id', component: GoalDetailComponent },
  { path: '', redirectTo:"/goals", pathMatch:"full"},
  { path:'**', component:NotFoundComponent},
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

We have added a route with the path goals/:id which displays the GoalDetailComponent. The :id token allows us to pass a parameter in the route. We will pass in the id of a goal to retrieve a specific goal. We'll also need to make some changes in our templates.

src/app/goal/goal.component.html

```
<div class="container">
  <ng-progress></ng-progress>
  <h1>MY GOALS</h1>
  <hr>
  <div class="row">
    <div class="col-md-6">
      <div *ngFor='let goal of goals;let i = index'>
        <div>
          <h6 id={{i}} appStrikethrough>{{goal.name}} due on {{goal.completeDate|
date|uppercase}}</h6>
          <button class="btn btn-primary" (click)='goToUrl(goal.id)'>View Details</
button>
          <button class="btn btn-outline-danger" (click)='deleteGoal(i)'>Delete
Goal</button>
        </div>
      </div>
    </div>
    <div class="col-md-6">
      <app-goal-form (addGoal)="addNewGoal($event)"></app-goal-form>
    </div>
  </div>
  <p *ngIf='goals.length > 5'>You have too many goals</p>
  <div class="row">
    <blockquote class="blockquote text-center">
      <p class="mb-0">{{quote.quote}}</p>
      <footer class="blockquote-footer"><cite>{{quote.author}}</cite></footer>
    </blockquote>
  </div>
</div>
```

Notice we have gotten rid of the toggle Details button and replaced it with a view Details button and next to it a Delete Goal button. We have also gotten rid of the div that was displaying the Goal Detail component. The View Details button has a click event listener which calls the goToUrl() function that takes in the goal id that is responsible for triggering the navigation to the GoalDetails component of a specific goal. The Delete Goal button also has a click event listener that calls the deleteGoal() function which deletes a goal from the goals array.

Let's create these functions that we have added to the GoalComponent.

src/app/goal/goal.component.ts

```
...
import { Router } from '@angular/router';
...
export class GoalComponent implements OnInit {

goToUrl(id){
    this.router.navigate(['/goals',id])
  }
  deleteGoal(index){
    let toDelete = confirm(`Are you sure you want to delete $
{this.goals[index].name}`)
    if (toDelete){
      this.goals.splice(index,1)
      this.alertService.alertMe("Goal has been deleted")
    }
  }
...
  constructor(goalService:GoalService, alertService:AlertService, private
quoteService:QuoteRequestService, private router:Router) {
    this.goals = goalService.getGoals()
    this.alertService = alertService;
  }
...
}
```

At the top, we have imported the Router module and injected it into the constructor with the router property. We have then created the goToUrl() function with an id as an argument. We have used the router's navigate function and passed an array that has the first part of the arguments as the path to the goals and the second part being the id of the goal. We have then kept the deleteGoal() function which deletes a goal from the array of goals.

Our buttons are too close to each other. Let's add some CSS to create some space between them.

src/app/goal/goal.component.css

```
.btn{
  margin:2px;
}
```

We have added a margin to the elements in our component that have the btn class. Remember that this style will only apply for those elements in the goal component. If we wanted to make this style global, we would put it in the app.component.css file instead.

Creating a getGoal method

Since our goal service is still delivering our goals, let's create a method in the service that will help us retrieve one goal.

src/app/goal-service/goal.service.ts

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

We have created a getGoal() method that takes in the id of a goal. We have then created a for-loop that checks whether the id of a goal is the same as the id parameter passed in, and if it is, then that specific goal is returned and we break out of that loop.

It will be the GoalDetail Component that will be displaying the specific goal so let's get it ready for that.

src/app/goal-detail/goal-detail.component.ts

```
...
import { Goal } from '../goal';
import {  ActivatedRoute, ParamMap } from '@angular/router';
import { GoalService } from '../goal-service/goal.service';
...
export class GoalDetailComponent implements OnInit {
  goal:Goal;
  constructor(private route:ActivatedRoute, private service:GoalService) { }
  ngOnInit() {
    let id = this.route.snapshot.paramMap.get('id');
    this.goal = this.service.getGoal(id)
  }
}
```

At the top, we have imported GoalService, ActivatedRoute to retrieve parameters from the route and ParamMap to provide methods that handle parameter access from the router. We have then injected the ActivatedRoute and GoalService in our constructor. Inside the lifecycle hook, ngOnInit(), we have used the route.snapshot to get the initial value of the route parameter, then we have extracted the id using the get method provided by paramMap function. We have passed this id we have retrieved to our service's getGoal() method which returns the specific goal which we assign to our goal property.

Let's change the look of our GoalDetail.

src/app/goal-detail/goal-detail.component.html

```
<div class="row">
  <div class="col-md-2">
  </div>
  <div class="col-md-8">
    <div class="card">
      <div class="card-body">
        <h5 class="card-title">{{goal.description}}</h5>
        <p class="card-text">This goal will be complete in {{goal.completeDate|
dateCount}} days.</p>
      </div>
    </div>
  </div>
  <div class="col-md-2">
  </div>
</div>
```

We have put our goal details in a bootstrap card. We can now serve our application and see what happens when we click the view details button. That's exactly how we wanted to display our goal details.

Tuesday:Practice: Moringa Overflow

Goal: Get comfortable using Angular Cli and practice the process of angular routing.

Warm Up

Discuss the following questions with your pair

- What is routing in Angular 6?
- What is this tag used for <router-outlet></router-outlet> ?
- What is a routerLink?

Code

Moringa Overflow

The project for today will be to create a basic discussion /forum site where users can post questions or news about a certain topic. Much like stack overflow but we will call it Moringa overflow. Here are some user stories to help you get the bigger picture of the project.

1. As a user, I would like to see several discussion categories on the landing page
2. As a user, I would like to click a category and view its collection of posts
3. As a user, I would like to click on a post and see its details
4. As a user, I would like to add a post to my desired category
5. As a user, I would like to have the option of deleting my post
6. As a user, I would like to update my post if necessary.

Wednesday: Two Day Project

Two Day Project

For the next two days, you will be creating a giphy website (Links to an external site.) replica, using the giphy api (Links to an external site.). By the end of the two days, your app should

- display trending gifs on app load shown in a column/grid
- have an input which allows you to search for specific gifs
- at the bottom of the results, there is a ‘load more’ button, which gets more gifs using that search term.

Use the concepts you have been learning in Angular to build up the application. Make sure the application is at portfolio quality. If you have some time left off you can experiment by adding some extra features like sharing on social media.

Thursday: Interview Questions

Two Day Project Continues

Continue building with your pair the giphy **clone website.** Make sure you make regular commits while pushing your code to Github and make sure you have clear commit messages

Angular 6 Interview Questions

These are some of the Interview questions you might encounter on Angular 6. discuss with your pair and also search online for other questions.

1. What is Angular 6?
2. What are the key components of Angular 6?
3. Explain Modules in Angular 6
4. Explain Components in Angular 6
5. What are Angular 6 directives? Explain with examples.
6. What is CLI?
7. What is Dependency Injection? Explain with examples.
8. When does ngOnInit event get called in Angular 6 Application Lifecycle?
9. What are Event Emitters and how do they work in Angular 6?

10.Explain Bootstrapping in an Angular Project

Angular week 2 IP

- Due No due date
- Points 22
- Submitting a website url

Github Search

Create a website where users may enter a GitHub username into a form, submit it, and see names and descriptions of that person's public repositories. A person can also look for repositories

Use the GitHub API (Links to an external site.)Links to an external site. to retrieve this information. This API allows 5,000 requests per hour with an API key, but only 60 requests per hour _without_ one. Everyone is therefore required to use their own unique key. GitHub refers to these keys as "Personal Access Tokens".

Creating Personal Access Tokens

- Visit the _Settings_ area of your GitHub account like this
-
- select Personal Access Tokens from the sidebar, and hit Generate New Token
-
- GitHub will offer a list of options **Do not select any**. These grants read/write permissions and access to personal data. Finally, select **Generate Token.**
-

A sample request URL might look like this.

```
'https://api.github.com/users/daneden?access_token=' + apiKey
```

Technical Requirements

1. Your project must contain a well-designed landing page that displays your GitHub information such as your username, your profile photos and a list of your repositories.
2. You must place your access key inside the `environment.ts` file and place it inside the `gitignore` file.
3. You must create a HTTP service that uses a promise to make the requests to the GitHub API.
4. The HTTP request should be able to search for both GitHub users and GitHub repositories.
5. Your project must have two classes for the `user` and `repository`.
6. Your project must have a proper routing structure that links a GitHub username to the users GitHub repositories.
7. Your project must have a separate routing module.
   5.Your project must contain a custom directive and a custom pipe.
8. Your Project must be well designed and visually appealing and of portfolio quality.
9. Your project must have a well-documented README file.
10. Your project must be deployed and the deployed link should be submitted.

Rubric

Github Project Objectives

Github Project Objectives

Criteria	Ratings	Pts
This criterion is linked to a learning outcome A well documented readme is the first documentation any developer should use		1.0 pts

Criteria		Ratings	Pts
1.0 Pts A well documented Readme file on Github A project README that includes: - project or program name - author name - description of project - project setup instructions - link to live site on GitHub Pages - copyright and license information	0.0 Pts This is the default readme that is generated by Github on new projects. Usually contains the name of the project and a description if one is provided		
This criterion is linked to a learning outcome Project is in a polished, portfolio-quality state. Suggestions for what this can mean: Intuitive, easy to follow layout. Simple yet polished styling. Form field labels where appropriate. Form fields that are validated correctly, and get cleared after submitting. Detailed, well put together readme. No typos. And much more.			
2.0 Pts Went above and beyond in the visual aspect Suggestions for what this can mean: Intuitive, easy to follow layout. Simple yet polished styling. Form field labels where appropriate. Form fields that are validated correctly, and get cleared after submitting. Detailed, well put together readme. No typos. And much more.	0.0 Pts No effort to make it visually pleasant		2.0 pts
This criterion is linked to a learning outcome Does the project work functionally achieving the expected objective?			
2.0 Pts Yes, the project works as expected All objectives are functionally met in the deployed project	0.0 Pts No, the project does not work as specified in the objectives		2.0 pts
This criterion is linked to a learning outcome Commits are made regularly with clear messages associated with them			
2.0 Pts 20 + commits in the project with well detailed commit messages	0.0 Pts Less than 10 Github commits present in the project		2.0 pts
This criterion is linked to a learning outcome Project is link and description is provided on Github repository			
2.0 Pts Project description is present This is used as a blurb of the project, explains what the project is bout and the linked to a deployed site if present	0.0 Pts Did not make use of Github gh-pages		2.0 pts
This criterion is linked to a learning outcome Page incorporates a custom-made stylesheet			
1.0 Pts Used custom CSS stylesheet All objectives are functionally met in the deployed project	0.0 Pts Used custom CSS stylesheet		1.0 pts

Criteria	Ratings		Pts
This criterion is linked to a learning outcome Project contains a well-designed landing page.	1.0 Pts Well designed landing page	0.0 Pts Default HTML	1.0 pts
This criterion is linked to a learning outcome HTTP service that uses a promise to make the requests to the Github API	3.0 Pts Use working HTTP service	0.0 Pts No service available	3.0 pts
This criterion is linked to a learning outcome Two classes for the User and Repository	2.0 Pts Has two separate classes that are in separate file	0.0 Pts Classes are all in the same file	2.0 pts
This criterion is linked to a learning outcome Your project must have a separate routing module	3.0 Pts Project has a routing module linked to the app module	0.0 Pts Routing is not done inside the app module	3.0 pts
This criterion is linked to a learning outcome Contains a custom directive and a custom pipe	3.0 Pts Custom pipe and directive	0.0 Pts No custom pipe or directive	3.0 pts

Total points: 22.0

```
$ git add .
$ git status // to see what changes are going to be commited
$ git commit -m 'Some descriptive commit message'
$ git push origin master

$ git checkout gh-pages // go to the gh-pages branch
$ git rebase master // bring gh-pages up to date with master
$ git push origin gh-pages // commit the changes
$ git checkout master // return to the master branch
```

```
git pull origin master --allow-unrelated-histories
```