BIPUL KUMAR

MEAN Training Lab ManuaL

# : Getting Started

## **Introduction**

In this course we will deep dive into MEAN stack.

* M- Mongo DB
* E- ExpressJS
* A - Angular
* N - Node

We will be building an Angular application for client side (UI) and will use MongoDB as DB and use ExpressJS & NodeJS as backend services. And therefore, we will see a fullstack UI Application.

We will be using a lot of cool functions while going through UI.

### **Installing Node and Angular CLI**

* 1. **For this application we will need couple of tools and we will install them step by step**

We will install mongoDB later but we will need nodeJS upfront because even Angular needs it. Angular has a complex workflow. Angular will use Typescript superscript on javascript task runner during compilation is nodeJS.

Go to <https://nodejs.org/en/> to install nodeJS application. Download the latest version. This should be available for all OS. Its recommended to install the latest version

* 1. Now we need to install another tool to run the angular application called Angular CLI which is also called as command line interface. It is the main way to create angular apps and as mentioned Angular has lot of complex build workflow. We will need a lot of tool in our workflows. To set up the codebase on our own we will have a lot of overhead. Therefore we will simplify our working with Angular using Angular CLI. To install it we will need node package manager. So when we installed nodeJS we will automatically have npm.

Refer to <https://cli.angular.io/> for more information and certain commands that we will be using throughout this lab

On Windows Open command prompt\powershell and type “npm install -g @angular/cli “. You can also use npm install -g @angular/cli@latest but by default latest angular is installed.

On mac and linux you should use command “sudo npm install -g @angular/cli “

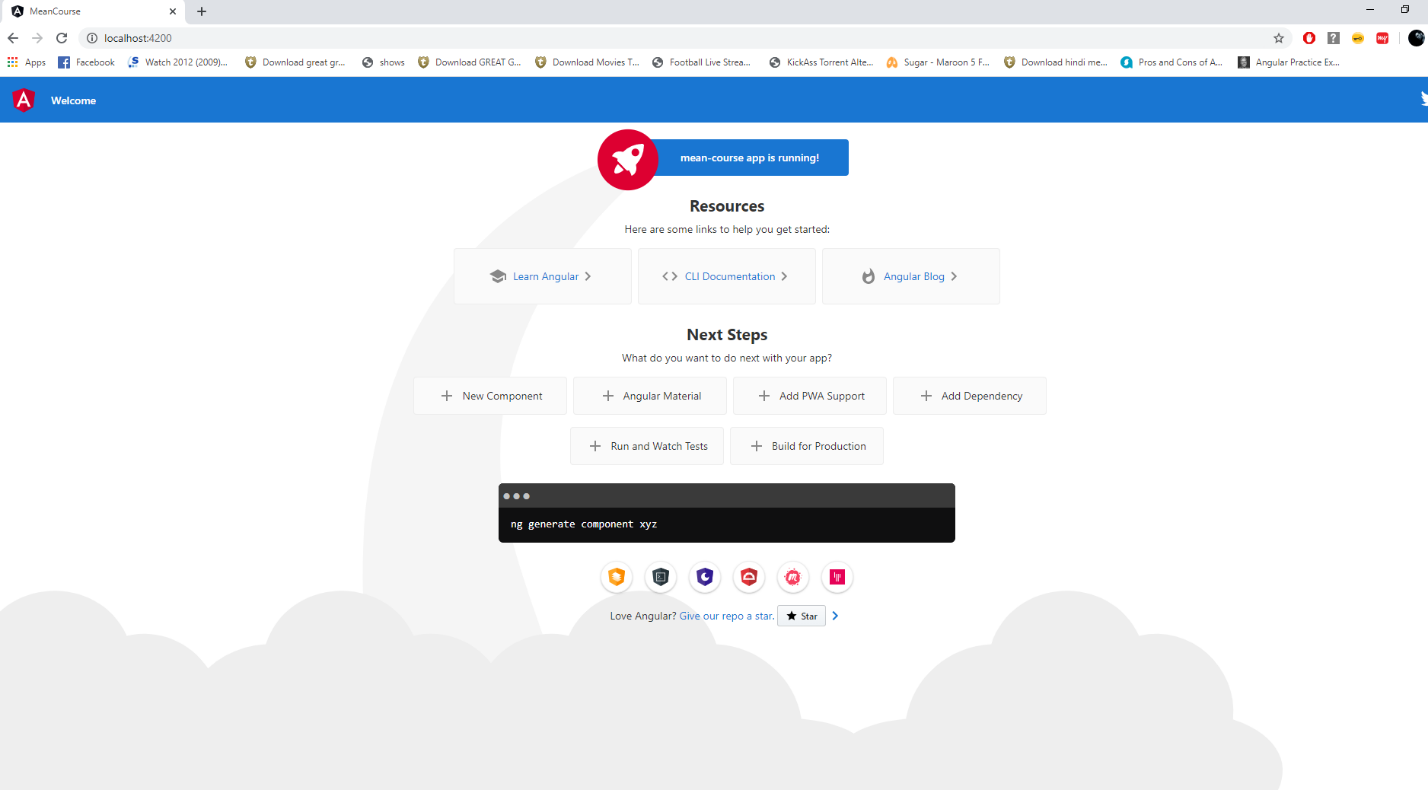
* 1. Now we will create an Angular project using below command. Go to a folder where you want to create the angular app in the command prompt and type

**“ng new mean-course”**

* 1. Now we can run the application to the subdirectory where we added the mean-course by using below command once the application created successfully (it will take some time to complete the previous process:

**“ng serve”**

You can observe that it loaded the angular application like below in the browser:



* 1. Installing IDE – For this course we will be using VS-Code which is freely available. Install it from below link. You can use other tools as well If you are familiar with other tools like WebStorm etc.

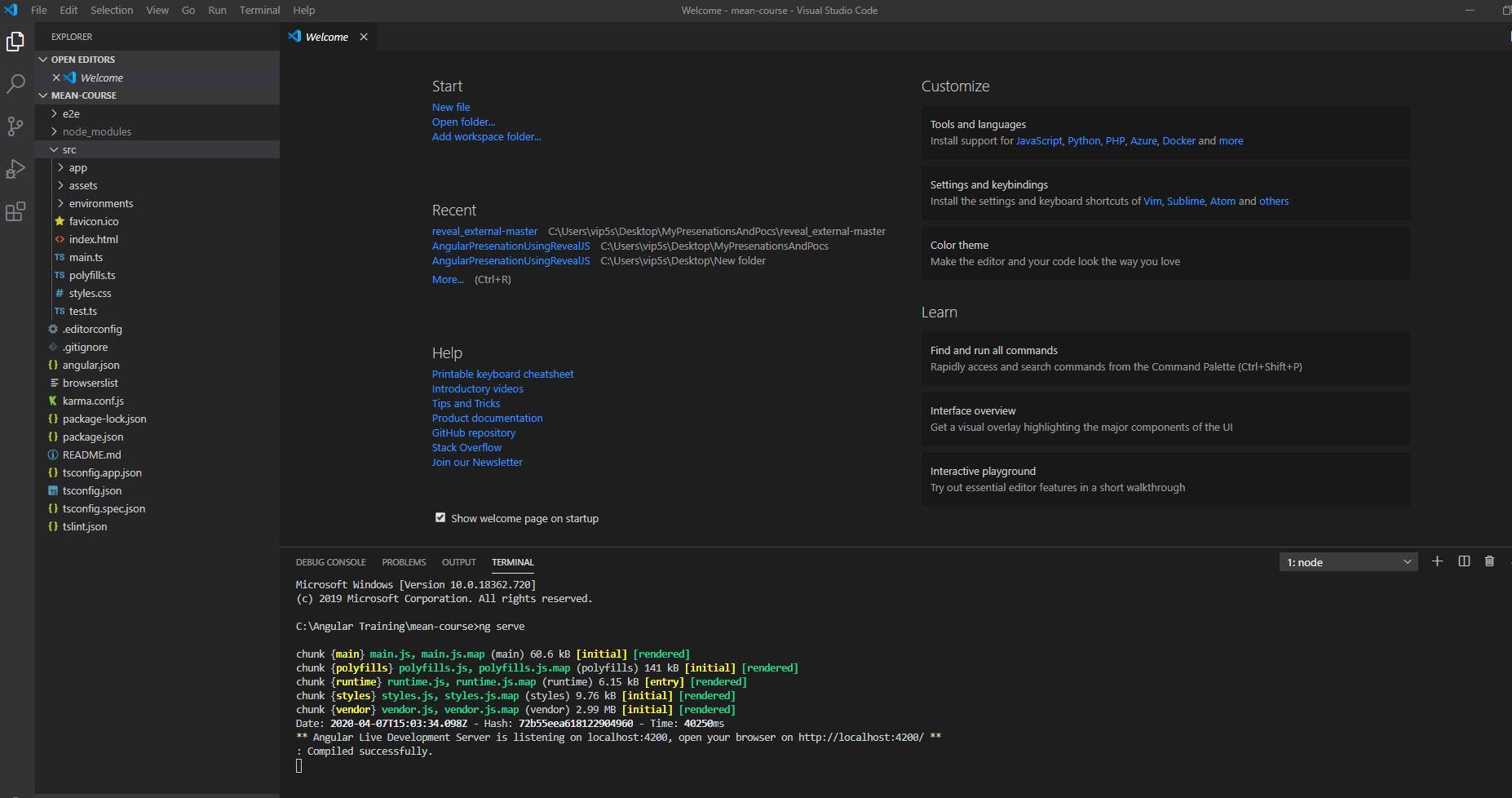
<https://code.visualstudio.com/>

* 1. Understanding the Code Structure on VSCode. Load the application just by going to the shared drive where you have added the application and right click to load in VSCode.
  2. Now lets cleanup the app-componet.html with below text and save it and observe

<div>

<h1>My First Angular App</h1>

</div>



* Package.Json will have all dependencies in the project
* Node\_modules will be having all dependencies that are listed in package.json. We can create this folder by doing npm i
* App folder will have the main code. Make some changes in the HTML and observe the app is running in background gets updated.
* The one HTML in body called app-root.
* App-root is the main section where the component is loaded. Observe the ts file.
  1. Course outline

Getting Started

Angular Frontend

NodeJs + Express Backend

Handling Data with MongoDB

Enhancing the App

Deployment

Authorization

Authentication

Data Pagination

Optimization

Error Handling

Image Upload

# Building an Angular Front End

We will be doing brief introduction and important basics of Angular application. We will be creating a fresh Angular Project. Lets open the Angular Project and understand how the Angular Project Starts. We remember we did ng serve.

Let’s look at the source code. We have basic HTML5 skeleton and not much component.

App-root and the script imports are the important things on HTML 5 which runs the application.

Now look at the app.component.ts and see the component selector .

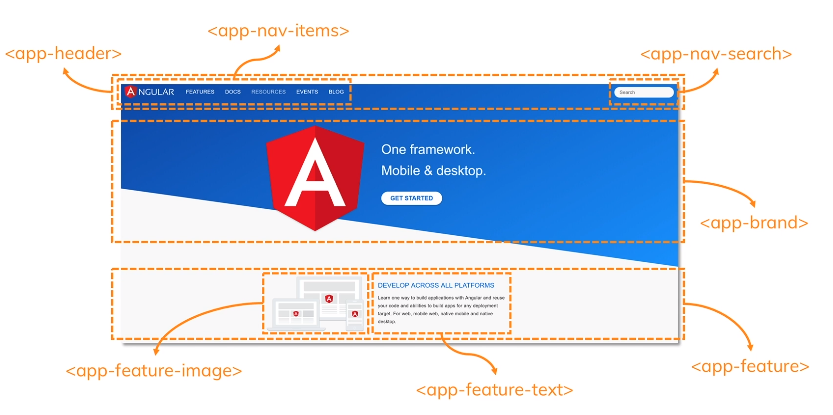
The swapping logic is residing in the app.module.ts. App.module.ts holds the main ingredients. Look at the bootstrap and imports section there.

Main.TS file starts the application and will execute the code.

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.log(err));

1. **Angular thinks in Components:**

Look at below picture of a sample website which can have various components. 

* 1. **Adding our first component:** We don’t add all logic to app.component in one component as it will become really long and cumbersome. Lets do below activities
     1. Now go to terminal in VSCode and type **“ng g c posts/post-create”** inside the folder.
     2. Observe the code that was generated by the angular cli.
     3. Observe the app.module.ts to identify where the link was created. Let’s also understand the selector. We have created a basic component.
     4. Lets add below line to the app.component.ts file

<app-post-create></app-post-create>

* + 1. Lets run the application and see how the component showed up on the application. If you stopped application do a ng serve again or just save the files and you can observe the changes reflected on browser.
  1. **Listening to events:** We want to create a new post. So what we need in our application that can create a new post. At core minimum we need a text area or textbox and a button. So lets try to add below code in the application:
     1. Add following text in the post-create.component.html and save the file and observe the browser:

<textarea rows="6"></textarea>

<button>Save Post</button>

* + 1. **EventBinding**: Now we will add event on the button. Eventbinding feature allows us to listen to events in a declarative way. In javascript where we would go to our code and by using query selector we would write some methods to hook it up with buttons. In angular we will go to template we will add parenthesis like (click) = “{code which you want to execute }”
    2. Lets add below code in post-create.component.ts

onAddPost(){

    alert(" Save Post Button Clicked");

  }

* + 1. Update below code in post-create.component.html and lets try to understand the code

<textarea rows="6"></textarea>

<button (click) = "onAddPost()" >Save Post</button>

* + 1. Now save the code and observe the button click in browser where alert message is bound
  1. **Outputting a content :** Before we fetch user input lets work on outputting a content. Let’s output a post and output a dummy content. For this we need two things:
* We need to set some content – set it in the code
* We need a place to store that content – Property variable on the class
  + 1. **String Interpolation:** In Angular, String interpolation is used to display dynamic data on HTML template (at user end). It facilitates you to make changes on component.ts file and fetch data from there to HTML template. Its syntax is {{ *variable*}}
    2. Update below code on the post-create.component.html and lets try to understand the code

<textarea rows="6"></textarea>

<button (click) = "onAddPost()" >Save Post</button>

<p>{{ newPost }}</p>

* + 1. Update below code on the post-create.component.ts and lets try to understand the code

newPost = '';

  onAddPost(){

    this.newPost= 'The user\'s post';

}

Save the file and now click on the button and observe the application.

* + 1. Update below code on the post-create.component.html and let’s try to understand the code

<textarea rows="6" [value] = "newPost"></textarea>

<button (click) = "onAddPost()" >Save Post</button>

<p>{{ newPost }}</p>

* 1. **Getting User Input:** Let’s get user Input. There two ways to do that.
     1. **Template binding -** First one is called local reference. We can add a marker in a HTML with a # and we can use any variable to that like highlighted in below code:
     2. Update below code on the post-create.component.html and let’s try to understand the code

<textarea rows="6" [value] = "newPost" #postInput ></textarea>

<button (click) = "onAddPost(postInput)" >Save Post</button>

<p>{{ newPost }}</p>

* + 1. Now update addPost method like in below code on the post-create.component.ts , save the file and observe the console.log :

  onAddPost(postInput : HTMLTextAreaElement){

    console.dir(postInput);

    this.newPost= 'The user\'s post';

  }

* + 1. Now let’s update the code to see how user Input will show up on the screen here. Copy paste below code and save the file to see updates on the screen:

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

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {

  }

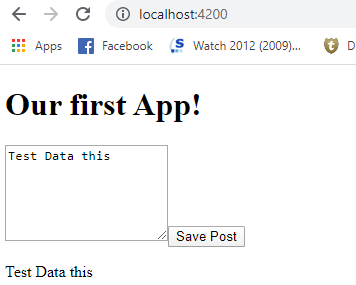
  newPost = 'No Content';

  onAddPost(postInput : HTMLTextAreaElement){

    this.newPost= postInput.value;

  }

}



* + 1. **Two-way binding -** We have seen the first way above. Now there is another feature angular offer called Two way binding. Two-way binding just means that:
* When properties in the model get updated, so does the UI.
* When UI elements get updated, the changes get propagated back to the model.
  + 1. Update below code on the post-create.component.html and let’s try to understand the code

<textarea rows="6" [(ngModel)] ='enteredValue'></textarea>

<hr>

<button (click) = "onAddPost()" >Save Post</button>

<p>{{ newPost }}</p>

* + 1. Update below code on the post-create.component.ts and let’s try to understand the code

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

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {

  }

  enteredValue = '';

  newPost = 'No Content';

  onAddPost(){

    this.newPost=this.enteredValue;

  }

}

* + 1. Upon saving the files you will observe angular was not able to understand the (ngModel) passed in HTML and nothing really happened. This is because till now we have not loaded the angular module which has functionality related to ngModel. So let’s go to app.module.ts and add FormsModule. Replace below code and observe the highlighted section that we added:

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

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { AppRoutingModule } from './app-routing.module';

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

import { PostCreateComponent } from './posts/post-create/post-create.component';

@NgModule({

  declarations: [

    AppComponent,

    PostCreateComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule

  ],

  providers: [],

  bootstrap: [AppComponent]

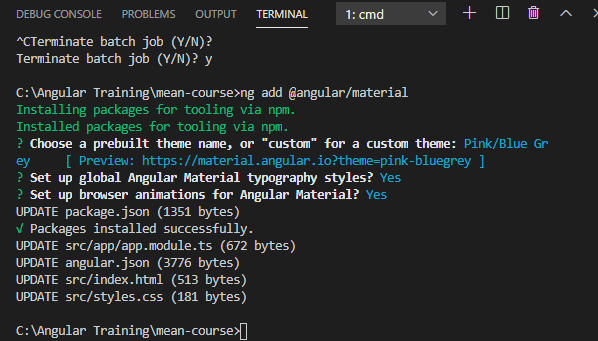
})

export class AppModule { }

* + 1. Now save the file and observe the application running in background. We have covered essentially all kinds of core template binding in Angular. Event Binding, Interpolation, property binding and two-way binding.
    2. Now lets leave that ugly world where our UI didn’t looked go. Lets learn about styling and create better angular UI in next modules.
  1. **Installing Angular material:** Let’s build a beautiful application. We can write css of our own and do styling of the controls ourselves but this is a MEAN stack guide so to avoid learning styling we will use Angular Material and by doing this we will learn how we can use a third party library to integrate it with angular application. Angular Material is also created by the Angular Team so it will be a great working experience. Its just not a styling package like bootstrap, its an angular package and has its own set of angular components that we can drop in our angular application.
     1. It uses Google’s Material Design, so it depends on you and your project to incorporate.
     2. Visit the below website to understand the Material Design and its features:

<https://material.angular.io/>

* + 1. Let’s install the Angular Material in our project now. You can use “npm install --save @angular/material” command. But if you are using angular cli then you can use a different command “**ng add @angular/material**”. Add this command in the terminal on VSCode.
    2. Let’s investigate the code and see what got included in the application. Look into the files mentioned in screenshot below to see what got updated.



* + 1. Now we need to add the Angular Material into the application so for doing that lets update our app.module.ts with below code and understand the highlighted section. What it does is it unlocks all input related components from material design into the angular application. You must use package version 9.0.0. You can find all paths in the material website like in below link:

<https://material.angular.io/components/input/api>

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

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { MatInputModule } from '@angular/material/input';

import { AppRoutingModule } from './app-routing.module';

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

import { PostCreateComponent } from './posts/post-create/post-create.component';

@NgModule({

  declarations: [

    AppComponent,

    PostCreateComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule,

    BrowserAnimationsModule,

    MatInputModule,

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

* + 1. Update below code on the post-create.component.html and let’s try to understand the highlighted code. Also see the changes that happened on the browser.

<mat-form-field>

<textarea  matInput rows="6" [(ngModel)] ='enteredValue'></textarea>

</mat-form-field>

<hr>

<button (click) = "onAddPost()" >Save Post</button>

<p>{{ newPost }}</p>

* + 1. Let’s add another module from material to make the control better. Lets add below code in the app.module.ts:

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

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { MatInputModule } from '@angular/material/input';

import { MatCardModule } from '@angular/material/card';

import { MatButtonModule } from '@angular/material/button';

import { AppRoutingModule } from './app-routing.module';

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

import { PostCreateComponent } from './posts/post-create/post-create.component';

@NgModule({

  declarations: [

    AppComponent,

    PostCreateComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule,

    BrowserAnimationsModule,

    MatInputModule,

    MatCardModule,

    MatButtonModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

* + 1. Update below code on the post-create.component.html and let’s try to understand the highlighted code. Also see the changes that happened on the browser.

<mat-card>

    <mat-form-field>

        <textarea  matInput rows="6" [(ngModel)] ='enteredValue'></textarea>

    </mat-form-field>

</mat-card>

<hr>

<button (click) = "onAddPost()" >Save Post</button>

<p>{{ newPost }}</p>

* + 1. Still We don’t see the UI controls well. So, lets add a few classes in the post-create.component.css. Now observe the look and feel.

mat-card {

    width: 80%;

    margin: auto;

}

mat-form-field,

textarea {

    width:100%;

}

* + 1. Update below code on the post-create.component.html and let’s try to understand the highlighted code. Also see the changes that happened on the browser.

<mat-card>

    <mat-form-field>

        <textarea  matInput rows="6" [(ngModel)] ='enteredValue'></textarea>

    </mat-form-field>

    <button mat-raised-button

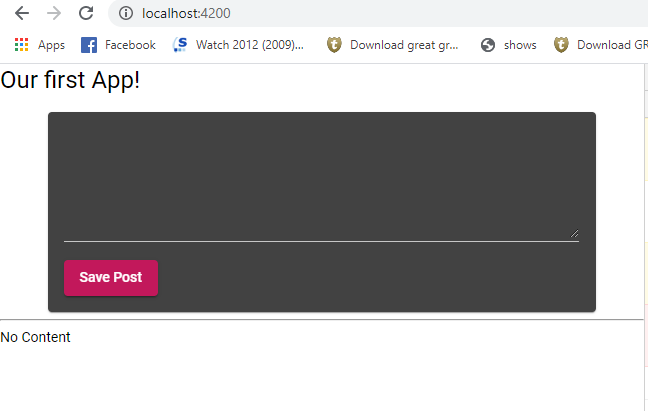
    color = "primary"

    (click) = "onAddPost()" >Save Post</button>

</mat-card>

<hr>

<p>{{ newPost }}</p>



* 1. **Adding a Toolbar –** Let’s add new component called header which can have the toolbar for the application
     1. Go to terminal in VSCode and type **“ng g c header/header”** inside the folder to add the header component.
     2. Now go to app.module.ts and update below code to add MatToolBar. Read more about it on material design website:

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

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { MatInputModule } from '@angular/material/input';

import { MatCardModule } from '@angular/material/card';

import { MatButtonModule } from '@angular/material/button';

import { MatToolbarModule } from '@angular/material/toolbar';

import { AppRoutingModule } from './app-routing.module';

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

import { PostCreateComponent } from './posts/post-create/post-create.component';

import { HeaderComponent } from './header/header/header.component';

@NgModule({

  declarations: [

    AppComponent,

    PostCreateComponent,

    HeaderComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule,

    BrowserAnimationsModule,

    MatInputModule,

    MatCardModule,

    MatButtonModule,

    MatToolbarModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

* + 1. Let’s update header.component.html with below code to add the toolbar on the application

**<mat-toolbar color="primary">My App</mat-toolbar>**

* + 1. Let’s go to app.component.html and add below code and run the application and observe it.

<app-header></app-header>

<app-post-create></app-post-create>

* + 1. Now you can observe that the textbox is coming very close to the header so let’s update app.component.html with below code

<app-header></app-header>

<main>

    <app-post-create></app-post-create>

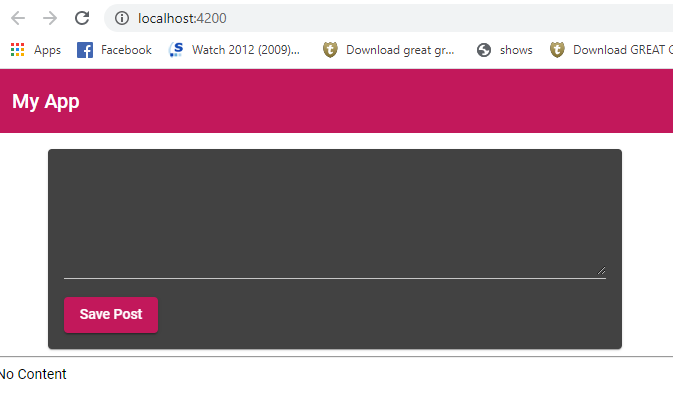
</main>

* + 1. Now Let’s update the class for app.component.css and save the files to see below screenshot in the application.

main {

    margin-top: 1rem;

}



* 1. **Output the Post –** The goal is here to output the post so let’s create a new component that can show the output in a nice format.
     1. Go to terminal in VSCode and type **“ng g c posts/post-list”** inside the folder to add the header component.
     2. Update the app.component.ts code to add the app-post-list component.

<app-header></app-header>

<main>

    <app-post-create></app-post-create>

    <app-post-list></app-post-list>

</main>

* + 1. The goal here is to create a list of posts in the post-list component which user will post in certain networking websites. Lets get rid of paragraph from post-create component.

<p>{{ newPost }}</p>

* + 1. We will use another feature of angular-material package, we will use the MatExpansionModule. Let’s update the below code in app.module.ts

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

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { MatInputModule } from '@angular/material/input';

import { MatCardModule } from '@angular/material/card';

import { MatButtonModule } from '@angular/material/button';

import { MatToolbarModule } from '@angular/material/toolbar';

import { MatExpansionModule } from '@angular/material/expansion';

import { AppRoutingModule } from './app-routing.module';

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

import { PostCreateComponent } from './posts/post-create/post-create.component';

import { HeaderComponent } from './header/header/header.component';

import { PostListComponent } from './posts/post-list/post-list.component';

@NgModule({

  declarations: [

    AppComponent,

    PostCreateComponent,

    HeaderComponent,

    PostListComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule,

    BrowserAnimationsModule,

    MatInputModule,

    MatCardModule,

    MatButtonModule,

    MatToolbarModule,

    MatExpansionModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

* + 1. Let’s add below code in the post-list.component.html, save the file and observe the application.

<mat-accordion>

    <mat-expansion-panel>

        <mat-expansion-panel-header>

            The expansion title

        </mat-expansion-panel-header>

        <p> I am an expansion Panel</p>

    </mat-expansion-panel>

</mat-accordion>

* + 1. To update the look and feel let’s do some updates to the below styling files individually:
       1. post-list.component.css – add below code

:host {

    display:block;

    margin-top: 1rem;

}

* + - 1. post-create.component.css – Remove the other class that you added before and just keep below code on the file.

mat-form-field,

textarea {

width:100%;

}

* + - 1. app.component.css – update with below code Save the files and observe the changes in the browser.

main {

    width:80%;

    margin: 1rem auto;

}

* + 1. Now lets try to see how this post-list.component can show multiple posts. So for that update below code in the post-list.component.ts and lets try to understand the code. It would be great if somehow Angular can show these posts dynamically. Its possible in Angular using directives. We covered a bit of it earlier. Lets look into it more in next module below ‘2.8’

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

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit {

  constructor() { }

  posts = [

    {title: 'First Post', content: " This is the first post content"},

    {title: 'Second Post', content: " This is the Second post content"},

    {title: 'Third Post', content: " This is the Third post content"}

  ]

  ngOnInit(): void {

  }

}

* 1. **Diving into Structural Directives –** Above we saw that we had a list of title and content that we wanted to display on the application. We can create three mat-accordion and hardcode it on the component to display the values. But wouldn’t it be great if all those HTML elements can be created dynamically at runtime. So, to achieve it Angular has something called structural directives.
     1. Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, or manipulating elements. As with other directives, you apply a structural directive to a *host element*. The directive then does whatever it's supposed to do with that host element and its descendants.
     2. Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name
     3. **\*ngfor structural directive-** It renders a template for each item in a collection. The directive is placed on an element, which becomes the parent of the cloned templates.
     4. Let’s update the post-list.component.html with below code and observe the code. Here we used something called ngfor directive and it uses the highlighted syntax. Now save the file and observe the three mat-expansion section created dynamically.

<mat-accordion multi = "true">

    <mat-expansion-panel \*ngFor="let post of posts">

        <mat-expansion-panel-header>

            {{post.title}}

        </mat-expansion-panel-header>

        <p>   {{post.content}}</p>

    </mat-expansion-panel>

</mat-accordion>

* + 1. You can allow **multi** above to open multiple accordion on the panel by setting multi=true on the accordion as mentioned in highlighted section.
    2. Of course, we don’t want to use hardcoded data, we want to use data filled in the post-create component and which we want to save upon save button click. So here now we will have to establish communication between two components with flow of data. And to do that we will comment the posts array on the below code, save the file and observe the browser where nothing is displayed for post-list component

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

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit {

  constructor() { }

  // posts = [

  //   {title: 'First Post', content: " This is the first post content"},

  //   {title: 'Second Post', content: " This is the Second post content"},

  //   {title: 'Third Post', content: " This is the Third post content"}

  // ]

  posts = []

  ngOnInit(): void {

  }

}

* + 1. Now of course we want to display something like No Content if there are no posts. To achieve that we can update the code with below code and let’s understand the \*ngif directive.
    2. **\*ngIf –** A structural directive that conditionally includes a template based on the value of an expression coerced to Boolean. When the expression evaluates to true, Angular renders the template provided in a then clause, and when false or null, Angular renders the template provided in an optional else clause. The default template for the else clause is blank.
    3. Lets update the posts-list.component.ts with below code and lets try to understand the code. Obviously, the syntax says it all as what will display and what will not display.

<mat-accordion multi = "true" \*ngIf ="posts.length > 0">

    <mat-expansion-panel \*ngFor="let post of posts">

        <mat-expansion-panel-header>

            {{post.title}}

        </mat-expansion-panel-header>

        <p>   {{post.content}}</p>

    </mat-expansion-panel>

</mat-accordion>

<p class="info-text mat-body-1" \*ngIf ="posts.length <= 0"> No Posts added yet!</p>

* + 1. Now lets also add the class in the post-list.component.css with below code and see the application in browser after saving the file:

:host {

    display:block;

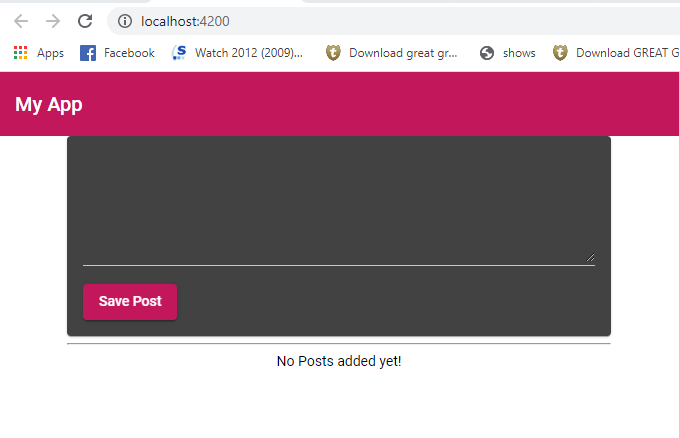
    margin-top: 1rem;

}

.info-text{

    text-align: center;

}



* 1. **Creating Post with property and event binding. –** It would be nice if we can add posts. To be able to do so, lets update below code in post-create.component.html

<mat-card>

    <mat-form-field>

        <input matInput type="text" [(ngModel)]= 'enteredTitle'>

    </mat-form-field>

    <mat-form-field>

        <textarea  matInput rows="6" [(ngModel)] ='enteredContent'></textarea>

    </mat-form-field>

    <button mat-raised-button

    color = "primary"

    (click) = "onAddPost()" >Save Post</button>

</mat-card>

* + 1. Lets update post-create.component.ts file with below code and understand the highlighted code:

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

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {

  }

  enteredTitle = '';

  enteredContent = '';

  onAddPost(){

    const post = {

      title: this.enteredTitle,

      content:this.enteredContent

    }

  }

}

* + 1. Now above we get the post now the question is how we can pass this value to the post-list component. We can use the property and event binding things that we learned before. We can emit our own events and we can send data from one component to another component. So, let’s do it step by step:
       1. Lets emit an event from post-create.component we will need something that is available in Angular called EventEmitter. Lets update below code in the post-create.component.ts and understand the highlighted code. So, in below highlighted code we added an event using Eventemitter. And to emit this event we can add an output **decorator @output**. By doing this we can listen to this from the outside. By Outside we mean we can get the even listed to parent component which is app.component.html

import { Component, OnInit, EventEmitter, Output } from '@angular/core';

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor() { }

  @Output() postCreated = new EventEmitter();

  ngOnInit(): void {

  }

  enteredTitle = '';

  enteredContent = '';

  onAddPost(){

    const post = {

      title: this.enteredTitle,

      content:this.enteredContent

    }

    this.postCreated.emit(post);

  }

}

* + - 1. So, we can add below code to app.component.html like below. Look at the (postCreated) component which calls this parent components onPostAdded method. We are passing argument from child component see above bullet. So $event variable is passed. This give you access to data that you passed from above, This variable is true for all click and other events. Let’s try understanding understand the [posts] which is added to the app-post-listy component. Where we are doing the property binding with storedPosts. Lets see the next bullet to understand it more.

<app-header></app-header>

<main>

    <app-post-create (postCreated)= "onPostAdded($event)" ></app-post-create>

    <app-post-list [posts] = "storedPosts"></app-post-list>

</main>

* + - 1. Lets copy below code in app.component.ts file. So we are having the method onPostAdded where we added the post to the storedPosts.

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

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  storedPosts = [];

  onPostAdded(post){

    this.storedPosts.push(post);

  }

}

* + - 1. Still our components are not linked, we need to add a few things into the post-list.component.ts file. Let’s update below code and try to understand the whole flow of events. So, this child component has to receive the data from the outside. From outside means the parent component app.component. So here we will use @Input decorator in child component which essentially means that this component is expecting an Input of a property from a parent component , in this case which is app.component

import { Component, OnInit, Input } from '@angular/core';

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit {

  constructor() { }

  @Input() posts = []

  ngOnInit(): void {

  }

}

* + 1. We are listening to the postCreated event from post.create component. Getting Data from there and add it to the storedPosts list and then binding it to the property of child component “posts” which is postListComponent. So when user tries now to add title and content, the change detection in Angular will detect the button click and bind the text from post-create component to post list component via event binding and propery binding and with help of **@Input and @Output decorators** of Angular.
    2. Run the application and now try seeing the multiple post with user Activity.
  1. **Creating a Post Model –** Now we must ensure that the post should have same structure. In above example we handled data from various places right from one child to parent and then back to another child. We want to keep the data definition (structure of data) intact in here otherwise we may run into issues while running the application. Also, if we want to comeback after a time in our code, we may have a hard time to understand how our data was structured. So, it make sense to create some models so basically we will create some contracts like how our post object will look.
     1. So now let’s create a model class file inside the posts folder called post.model.ts. You can also use below command to add the file by using command **“ng g class posts/post --type=model”** in terminal.
     2. Update below code in the post.model.ts file. Let’s try to understand the code:

export interface Post{

    title : string;

    content: string;

}

* + 1. Now let’s update the code at all the places where we intent to use this model.
       1. **App.component.ts –** Update below code and understand the added lines. Remove the post from push method and see the warning thrown by typescript.

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

import { Post } from './posts/post.model';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  storedPosts : Post[] = [];

  onPostAdded(post){

    this.storedPosts.push(post);

  }

}

* + - 1. **post-list.component.ts** – Update below code in the file.

import { Component, OnInit, Input } from '@angular/core';

import { Post } from '../post.model';

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit {

  constructor() { }

  @Input() posts: Post[] = []

  ngOnInit(): void {

  }

}

* + - 1. **Post-create.component.ts –** And finally update below code in the component. Here we can also set the eventEmitter to have the data being emitted as Post. So, we can have the datatype being passed as Post. Now run the application and everything should work as expected.

import { Component, OnInit, EventEmitter, Output } from '@angular/core';

import { Post } from '../post.model';

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor() { }

  @Output() postCreated = new EventEmitter<Post>();

  ngOnInit(): void {

  }

  enteredTitle = '';

  enteredContent = '';

  onAddPost(){

    const post: Post = {

      title: this.enteredTitle,

      content:this.enteredContent

    }

    this.postCreated.emit(post);

  }

}

* 1. **Adding forms –** Its time now for another improvement. From Create Post, we are binding we are doing binding using 2-way binding. Its not necessarily wrong but angular also makes it easy to work with Forms. Implements a set of directives and providers to communicate with native DOM elements when building forms to capture user input.
     1. Use Forms API to register directives, build form and data models, and provide validation to your forms. Validators can be synchronous or asynchronous depending on your use case. You can also extend the built-in functionality provided by forms in Angular by using the interfaces and tokens to create custom validators and input elements.
        1. Angular forms allow you to:
* Capture the current value and validation status of a form.
* Track and listen for changes to the form's data model.
* Validate the correctness of user input.
* Create custom validators and input elements.
  + - 1. You can build forms in one of two ways:
* **Reactive forms** use existing instances of a FormControl or FormGroup to build a form model. This form model is synced with form input elements through directives to track and communicate changes back to the form model. Changes to the value and status of the controls are provided as observables.
* **Template-driven forms** rely on directives such as NgModel and NgModelGroup create the form model for you, so any changes to the form are communicated through the template.
  + 1. Now let’s update the post.create.component.html and understand the highlighted code lines below:

<mat-card>

    <form (submit) = "onAddPost(postForm)" #postForm="ngForm">

        <mat-form-field>

            <input  matInput type="text" minlength="3" required name="title" #title="ngModel" ngModel>

            <mat-error \*ngIf="title.invalid">Please enter a Post Title</mat-error>

        </mat-form-field>

        <mat-form-field>

            <textarea matInput rows="4" required name="content" #content="ngModel" ngModel></textarea>

            <mat-error \*ngIf="content.invalid">Please enter a post Content</mat-error>

        </mat-form-field>

        <button mat-raised-button color="primary" type="submit">Save Post</button>

    </form>

</mat-card>

* + 1. Let’s understand each highlighted lines of code above.
       1. Purple highlighting – We added **form** element on the screen. It’s a normal HTML element. We wrapped all the controls on PostCreateComponent inside the forms HTML element. Now after this we don’t need to use two-way binding anymore (though we can). We can use another mechanism that Angular provides which is described in point below. When Angular detects a Form Element and we got the FORMS module included in the app.module.ts. It will automatically create the JavaScript objects behind the scenes which represents the forms that we are showing in below code snippet. We can easily register html components like input below as Forms Controls of which it will keep track of, and stores value of these controls and we can easily add validation too and submit the form and use the value of the form. So, we can get rid of two-way binding.
       2. Gray highlighting – We can remove the two-way binding that we had before and instead replace it with a **ngModel**. It’s a directive without any bindings. This will register the input form control above as a control to form elements behind the scenes.
       3. Green highlighting – Angular needs to know how to **name** the inputs. We can put any name to our choice. Above we named them as title and content. Now Angular is aware of input controls on the form.
       4. Dark Yellow highlighting – Now we don’t want the click on the button control, and we can remove that binding. Instead we will use **type=” submit”.** So, default HTML behavior is that a button of type Submit in a form will submit the Form. And submit will trigger a special event to which we can listen to submit event on form.
       5. Yellow highlighting – We added **(submit)**= “onAddPost” on the form’s element on HTML.
       6. Dartk Blue highlighting - Now we need to get access of form elements on the javascript. In here the typescript file. So, when the form gets submitted, we need access to the input fields. We can add a reference to the form, and we can add any names to it. We can assign a value to the reference which has to be “**ngForm**”. Now that directive Angular implicitly attaches to the form element. Now this strage syntax does is that it gives us access to form object on the javascript. To understand how this is used we can see the bullet point section 2.11.4 below.
       7. Bright Green Highlighting – We can do the validation; we can use default HTML5 validators. What Angular will do that it will detect and then runs behind the scenes logic on the forms object and will act appropriately. **required, minlegth** are some examples added above.
       8. Teal highlighting – We added mat-error component from material.
       9. Blue highlighting – in the \*ngif structural directive, we can access the input controls like mentioned above. For doing this we need to add the below highlighted reference on the input control.
       10. Red highlighting - #content="ngModel" create a reference to the input control which will be used in the that mat-error element.
       11. Now by doing this, the application will not work at this point until we make the changes in typescript file accordingly.
    2. Let’s copy below code to post-create.component.ts and try to understand it accordingly in below sections.

import { Component, OnInit, EventEmitter, Output } from '@angular/core';

import { Post } from '../post.model';

import { NgForm } from "@angular/forms";

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor() { }

  @Output() postCreated = new EventEmitter<Post>();

  ngOnInit(): void {

  }

  enteredTitle = '';

  enteredContent = '';

  onAddPost(form: NgForm){

    if(form.invalid){

      return;

    }

    const post: Post = {

      title: form.value.title,

      content:form.value.content

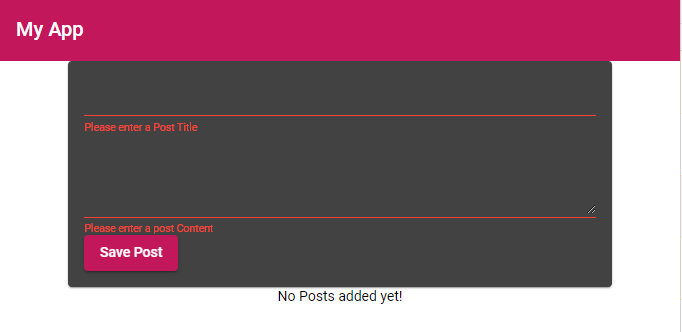
    }

    this.postCreated.emit(post);

  }

}

* + - 1. Red highlight – We added the Forms Module on the typescript file as we need to use the NgForm directive.
      2. Green highlight - As mentioned in bullet point 2.11.3.5 where we added the Submit event on the form element. We are now passing the forms object from HTML to the OnAddPost method. So that we can access the form values on the javascript for two-way binding.
      3. Blue highlight – We are now able to access the form.value and we can access individual input elements because we added ngModel and name attributes to the input element as described in bullet 2.11.3.2 and 2.11.3.3 above.
      4. Purple highlight – We are checking if Form is invalid do not update anything in Post and emit any event so that the blank accordions are not added on screen.
    1. Now let’s save the changes and run the application to see below validation functionality on the application.



* 1. **Getting Post from Post-Create to Post-List-** Right now we have chain of property and event binding between PostCreateComponent to AppComponent to PostListConmponent. This clearly works. But in bigger applications this will become cumbersome. Because we will have longer chains of property and event binding to get data from component A to component B, C D etc.
     1. It would be easier if we can have an alternate to this. So, for such an alternative we have something called service.
     2. **Service** – Service is a class that we will add to our Angular application which can be injected by Angular into its components. We will comeback to what inject means. This class will be able to centralize some task and provide easy access to Data from various components without using property and event binding.
     3. Let’s create a service by using command **“ng g service posts/post “** in the terminal.
     4. Add below code in the service class and try to understand the code in below points with highlights:

import { Injectable } from '@angular/core';

import { Post } from './post.model'

@Injectable({

  providedIn: 'root'

})

export class PostsService {

  private posts: Post[] = [];

  constructor() { }

  getPosts(){

    return [...this.posts];

  }

  addPost( title: string, content: string){

    const post: Post = { title: title , content: content};

    this.posts.push(post);

  }

}

* + - 1. Dark Red highlight – The getPosts() method is created to get the Posts which were just added from the app. We are using the spread operator here to get a true copy of the Posts array from this method. **Spread operator** allows an iterable to expand in places where 0+ arguments are expected. It is mostly used in variable array where there is more than 1 values are expected.It allows us the privilege to obtain a list of parameters from an array. Here we created a new array from old array and the objects of old array are copied. We are making object immutable. We don’t want to update the object posts to be updated by any other component from anywhere else. So, it’s a good practice to make the reference objects immutable.
      2. Blue highlight – We are adding an addPost method to the service whose intent is to be called from component which is trying to update the Posts array like in our case from PostCreateComponent.
      3. **Dependency Injection** – Now we can use these methods from post-create and post-list components without having to pass data using any property or event binding. For this we need to get this service added into these components. And this is done through a standard feature called dependency Injection.
      4. Violet highlight- What @Injectable will do is that this service is now ready to be consumed by any component in Angular. There is another way to inject this service into this Angular app. We can do that by adding the service in the provider section of app.module.ts. This also creates a single instance of the service which is ready to be consumed by Angular components. Its very important that we don’t create multiple instances of this service in our app otherwise our object POSTS will lose its integrity as a unique object being handled in this application.
    1. Let’s update the below code in post-list.component.ts and understand the code. Here we added the Service in the constructor and made it public. Essentially this is handled by Angular’s complex dependency injection feature. So, what we did was that we injected the PostsService instance into this component and now we can use the two methods of this service in this component. The public keyword in the component will automatically create a public property of postsService in this component. Let’s tie all of this together in next section 2.13.

import { Component, OnInit, Input } from '@angular/core';

import { Post } from '../post.model';

import { PostsService } from '../posts.service';

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit {

  constructor(public postService : PostsService) { }

  @Input() posts: Post[] = []

  ngOnInit(): void {

  }

}

* 1. **Calling GetPosts –** We will not use the constructor of the component to call GetPosts though that will be possible, we will instead use Angular’s Component lifecycle hooks called ngOnInit ( we will look into other lifecycle hooks later ) which is a better practice. ngOnInit is a life cycle hook called by Angular to indicate that Angular is done creating the component. If we called service from classes, the AngularUI wouldn’t have created until say there is a service call which has not completed. Mostly we use ngOnInit for all the initialization/declaration and avoid stuff to work in the constructor. The constructor should only be used to initialize class members but shouldn't do actual "work". So you should use constructor () to setup Dependency Injection and not much else. ngOnInit() is better place to "start" - it's where/when components' bindings are resolved.
     1. Lets update below code in post-list.component.ts and call the methods to getPosts of PostsService

import { Component, OnInit, Input } from '@angular/core';

import { Post } from '../post.model';

import { PostsService } from '../posts.service';

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit {

  constructor(public postsService : PostsService) { }

  @Input() posts: Post[] = []

  ngOnInit(): void {

    this.posts = this.postsService.getPosts();

  }

}

* + 1. But now we cannot get Posts added to this component. We need to do the similar logic on post-create.component.ts. Update below code there. You can see we cleaned up a lot of code from previous code amd got rid of a few things.

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

import { NgForm } from "@angular/forms";

import { PostsService } from '../posts.service';

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor( public postsService : PostsService) { }

  ngOnInit(): void {

  }

  enteredTitle = '';

  enteredContent = '';

  onAddPost(form: NgForm){

    if(form.invalid){

      return;

    }

    this.postsService.addPost(form.value.title, form.value.content);

  }

}

* + 1. Update app.component.ts and app.component.html with below code:
       1. App.component.ts –

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

import { Post } from './posts/post.model';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

}

* + - 1. App.component.html –

<app-header></app-header>

<main>

    <app-post-create></app-post-create>

    <app-post-list></app-post-list>

</main>

* + 1. Let’s try to run the application and try to Add title and content and try to save post. Nothing happens on the screen yet. What’s wrong there? Observe the code in the PostsService where we have initialized and empty array and now getPosts method will always gets blank array as that was the original state where we returned the empty array. There are now couple of ways to solve this.
    2. First way to update the application with below code where we have removed the **spreadOperator** and are now directly referencing the posts object. Save the file and run it in browser.

import { Injectable } from '@angular/core';

import { Post } from './post.model'

@Injectable({

  providedIn: 'root'

})

export class PostsService {

  private posts: Post[] = [];

  constructor() { }

  getPosts(){

    return this.posts;

  }

  addPost( title: string, content: string){

    const post: Post = { title: title , content: content};

    this.posts.push(post);

  }

}

* + 1. But above process is not the cleanest way of doing this. We should avoid unwanted manipulation of post in any component which is fetching our posts. So better way is to use an event driven approach where we actively push the information about new posts being available to components which are interested and for that we can use EventEmitter but EventEmitter should be used with conjunction of @output operator.
    2. **rxjs** - Instead we will use a feature provided by another package called **rxjs** which is not part of Angular. Lets see in package.json if its installed or not. We can find that its installed from the beginning. **Rxjs** is all about observables and a complex concept to grasp. Its essentially about objects that help us to pass data arounds.
    3. So, lets update the code in posts.service.ts with below code and try to understand it.

import { Injectable } from '@angular/core';

import {Subject } from 'rxjs';

import { Post } from './post.model'

@Injectable({

  providedIn: 'root'

})

export class PostsService {

  private posts: Post[] = [];

  private postsUpdated = new Subject<Post[]>();

  constructor() { }

  getPosts(){

    return [...this.posts];

  }

  getPostUpdateListener(){

    return this.postsUpdated.asObservable();

  }

  addPost( title: string, content: string){

    const post: Post = { title: title , content: content};

    this.posts.push(post);

    this.postsUpdated.next([...this.posts]);

  }

}

* + - 1. In above highlighted code pieces are explained below:
      2. Violet highlight- We added the import for rxjs and created a new private property of the type subject where we are passing list of Posts. **Subject** which we imported below is something like eventEmitter we can say which has broader usage than what angular ships with it. We also reverted the getPosts method to return the true object using spreadOperator which will be blank at first and will then have values which we will update later.
      3. Blue highlight – When we call addPost, we just donmt want to update the local property posts, but also take our subject “postUpdated” and just like emit we add next where we push the new value ( the true copy of our posts after we updated them before this line of code)
      4. Red highlight - What now we want to do is that we want to listen to the subject that we just created. So how do we listen to it. Its private so we can’t directly access it to prevent other components to emit data along with it. We will add a new method (we can put any name to the method). This new method will return the subject (postUpdated) with a special method called asObservable(). Now this method returns an object which we can listen, but we can’t emit. It still emits from inside this file but cannot be emitted outside the scope of this file.
    1. Lets update the post-list.component.ts and understand the code in following sub bullet points

import { Component, OnInit, Input , OnDestroy } from '@angular/core';

import { Subscription} from 'rxjs';

import { Post } from '../post.model';

import { PostsService } from '../posts.service';

@Component({

  selector: 'app-post-list',

  templateUrl: './post-list.component.html',

  styleUrls: ['./post-list.component.css']

})

export class PostListComponent implements OnInit, OnDestroy {

  constructor(public postsService : PostsService) { }

  private postSubscription : Subscription;

  @Input() posts: Post[] = []

  ngOnInit(): void {

    this.posts = this.postsService.getPosts();

    this.postSubscription =  this.postsService.getPostUpdateListener()

    .subscribe((postsReceived: Post[])=> {

        this.posts = postsReceived;

    });

  }

   ngOnDestroy() {

     this.postSubscription.unsubscribe();

   }

}

* + - 1. Blue highlight – We still fetch the posts from getPosts though its guaranteed to be empty right now. Let’s setup listener to the subject that we just created in the Posts Service. To get that we call the method that we just created getPostUpdateListener(). We can call the subscribe method which is made available. So subscribe now sets up a subscription which can take three argument:

the first argument is function which gets executed whenever a new data is emitted

the second argument is a functional which gets called when error is emitted.

The third argument will be a function that is called whenever the observable is completed, which means there are no more values to be expected

* + - 1. Pink highlight – So here we are using only the first argument as of now so we are using a arrow function with the method where we are expecting someData like postsReceived which we are emiting from this method which will be type of Posts array. So here we do assignments of posts that we received from the listener into the posts that we have on this component.
      2. Green highlight – One important thing is that the subscription that we added doesn’t cancel here whenever this component is teared down. Right now, this component never disappears but later when we have more components this component doesn’t shows up on screen. We need to make sure when this component is not showing up on screen then we need to unsubscribe to this listener or otherwise we will have memory leaks. So, to achieve this here we will use another angular pageLifecycle called ngOnDestroy which gets called whenever this component is destroyed from UI when you want to load another component(s) on screen.
    1. Now let’s run the application and see if we can see the posts getting added on the screen or now.
  1. **Working on our Form –** Our current Angular app is far from getting finished. But for now lets do some polishing**.** For example, lets add some labels on the UI. We can add it easily as we are using angular material.
     1. Let’s update below code. Save the file and observe the floating label showing on browser

<mat-card>

    <form (submit) = "onAddPost(postForm)" #postForm="ngForm">

        <mat-form-field>

            <input placeholder = "Post Title" matInput type="text" minlength="3" required name="title" #title="ngModel" ngModel>

            <mat-error \*ngIf="title.invalid">Please enter a Post Title</mat-error>

        </mat-form-field>

        <mat-form-field>

            <textarea placeholder = "Post Content" matInput rows="4" required name="content" #content="ngModel" ngModel></textarea>

            <mat-error \*ngIf="content.invalid">Please enter a post Content</mat-error>

        </mat-form-field>

        <button mat-raised-button color="primary" type="submit">Save Post</button>

    </form>

</mat-card>

* + 1. Now we observe upon adding the post, the value is still there on the controls. We can clear it on the onAddpostin post-create.component.ts. Update below code , save the file and observe the app in browser now.

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

import { NgForm } from "@angular/forms";

import { PostsService } from '../posts.service';

@Component({

  selector: 'app-post-create',

  templateUrl: './post-create.component.html',

  styleUrls: ['./post-create.component.css']

})

export class PostCreateComponent implements OnInit {

  constructor( public postsService : PostsService) { }

  ngOnInit(): void {

  }

  enteredTitle = '';

  enteredContent = '';

  onAddPost(form: NgForm){

    if(form.invalid){

      return;

    }

    this.postsService.addPost(form.value.title, form.value.content);

    form.resetForm();

  }

}

* + 1. Now let’s add the edit and delete button the mat accordion. The functionality for this will be done in later exercises. Lets update the code in post-list.component.ts

<mat-accordion multi = "true" \*ngIf ="posts.length > 0">

    <mat-expansion-panel \*ngFor="let post of posts">

        <mat-expansion-panel-header>

            {{post.title}}

        </mat-expansion-panel-header>

        <p>   {{post.content}}</p>

        <mat-action-row>

            <button mat-raised-button color= "accent" >EDIT</button>

            <button mat-raised-button color= "warn" >DELETE</button>

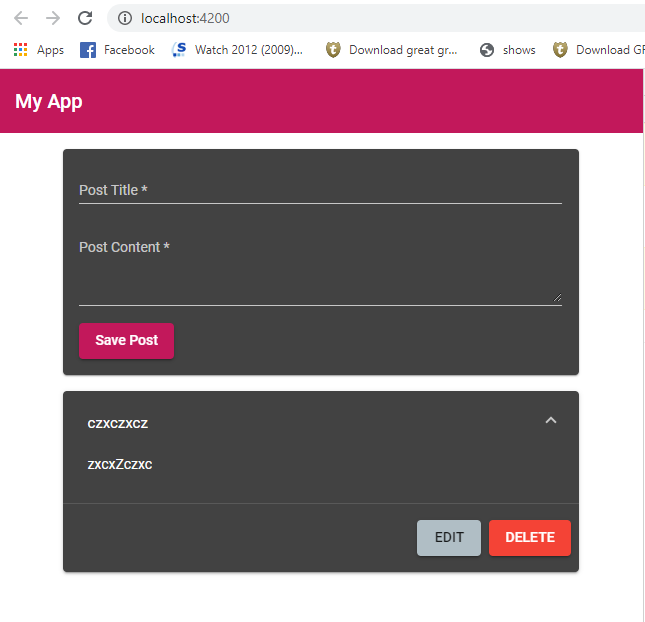
        </mat-action-row>

    </mat-expansion-panel>

</mat-accordion>

<p class="info-text mat-body-1" \*ngIf ="posts.length <= 0"> No Posts added yet!</p>

* + 1. Observe the application in browser and it should look something like below:



* 1. **More about Observables:** Observables is all about emitting data and listening to that data in different places of our application which makes it helpful. This is because we can subscribe to certain updates and push these changes from a totally different place.
     1. To understand this theory lets look into attached presentation



This is last topic we will complete at this moment in Angular, we will comeback to this one. For now, let’s move to backend. And will cover angular after the backend topics of mongo, node and express. After that we will comeback and see how that fits into our angular application.

* 1. Section Resources
* Angular: <https://angular.io/>
* Angular Material Docs: <https://material.angular.io/>
* Reference vs Primitive Types in JS: <https://academind.com/learn/javascript/reference-vs-primitive-values/>
* RxJS : <https://rxjs-dev.firebaseapp.com/>