# Angular Material – Owner\_Account

# Angular Material Installation

In this article, we are going to show you how to prepare our Angular project and how to install Angular Material in a few simple steps. But first thing first. Before we start with the Angular Material features, we need to create our project first.

So let’s dive right into it.

## Project Creation

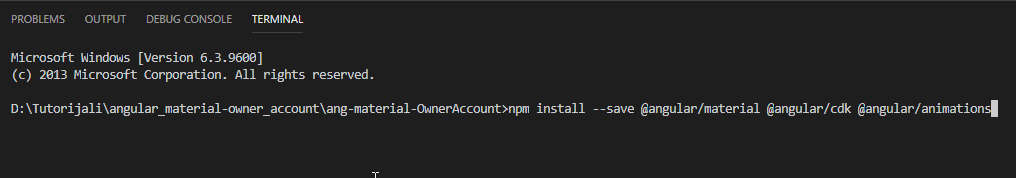
We are going to use Angular CLI through the entire project (and we strongly advise you to do the same), thus creating our project is no exception to that. So, let’s open the command prompt window and create our Angular project:

ng new ang-material-OwnerAccount

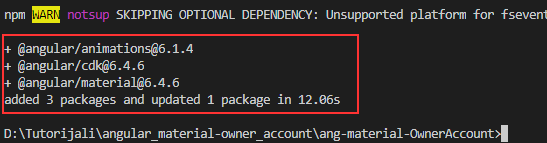
Once the creation is done, we are going to start the Visual Studio Code editor and open our project.

## Angular Material Installation

We are going to use npm to install required packages. Installing just Angular Material is not enough, we need to install CDK and Animations as well. So let’s do that first:



After installation finishes, we should see this result:



Now, we need to configure our animations, by importing BrowserAnimationsModule into the app.module.ts file:

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

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

BrowserAnimationsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

To continue, let’s include the prebuild theme for Angular Material. The theme is required and we can choose one of the available pre-built themes:

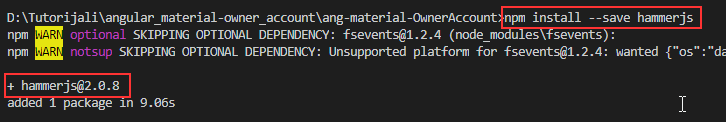
* deeppurple-amber.css
* indigo-pink.css
* pink-blugrey.css
* purple-green.css

To include a theme, we need to open the styles.css file and include the following line:

@import "~@angular/material/prebuilt-themes/indigo-pink.css";

The next step is to install the hammerjs library for the gesture support. In order to have a full feature of some components, we need to install it:

npm install --save hammerjs



After installation, we are going to import it as a first line in the maint.ts file:

import 'hammerjs';

And the last step is to add Material Icons if we want to. This is an optional step, but since we are going to use those icons, we are going to add them as well in the index.html file:

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>AngMaterialOwnerAccount</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

</head>

<body>

<app-root></app-root>

</body>

</html>

## Creating Material Module

Even though we can import all the required components into the app.module.ts file, this is not recommended. A better solution is to create a separate module with all the required material imports, and then import that separate module into the app.module.ts file. That being said, it is time to do that:

ng g module material --spec false

This command will create a new folder material with the material.module.ts file inside. But this file is missing one thing and that’s the exports array. So, let’s add it:

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

import { CommonModule } from '@angular/common';

@NgModule({

imports: [

CommonModule

],

exports: [

],

declarations: []

})

export class MaterialModule { }

Finally, we need to import this MaterialModule int the app.module.ts file:

import { MaterialModule } from './material/material.module';

imports: [

BrowserModule,

BrowserAnimationsModule,

MaterialModule

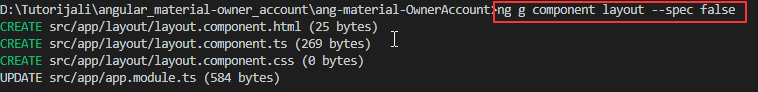
],

That is it. We have prepared everything we need to use the Material components, and without further ado, we are going to start using them.

## Creating the Layout Component

This component is going to be an entry point for our entire application, so let’s create it and import its selector inside the app.component.ts file:

ng g component layout --spec false

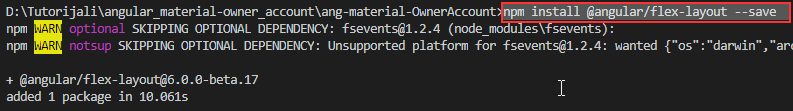


This command will create our component files and import it into the app.module.ts file.

### Angular Flax Layout

Before we modify the html component file, we need to install one more library @angular/flex-layout. We want our application to be responsive and for that purpose, we will use the mentioned library. So, let’s install it:

npm install @angular/flex-layout –save



And we need to register it inside the app.module.ts file:

import {FlexLayoutModule} from '@angular/flex-layout';

imports: [

BrowserModule,

BrowserAnimationsModule,

MaterialModule,

FlexLayoutModule

],

Now we can modify our layout.component.html file:

<div fxLayout="row wrap" fxLayoutAlign="center center" class="layout-wrapper">

<div fxFlex="80%" fxFlex.lt-md="100%" class="flex-wrapper">

<ng-content></ng-content>

</div>

</div>

As we can see, we use some angular/flex directives to create a responsive wrapper around our content. With the fxLayout element, we define the flow order of the child elements inside the container. The fxLayoutAlign will position children according to both main-axis and the cross-axis.

The fxFlex element resizes the child element to 80% of its parent, and if the screen goes below the medium than the child will take a 100% of its parent. If you want to read more about flex-layout, you can do that on this link: [Flex-Layout-Documentation.](https://github.com/angular/flex-layout/wiki/API-Documentation)

With the <ng-content> element, we are using angular content projection.

We have two more classes: layout-wrapper and flex-wrapper, so let’s implement them inside the layout.component.css file:

.layout-wrapper{

height: 100%;

}

.flex-wrapper{

height: 100%;

}

Excellent.

All we have to do is to remove all the content from the app.component.html file and introduce this component by using its selector:

<app-layout>

Application works.

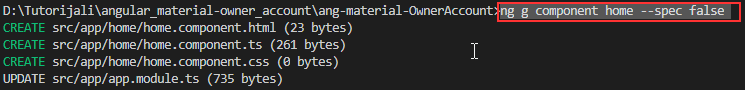
</app-layout>

We can start our application by typing ng serve and see that application truly works.

## Creating Home Component and Using Material Tabs

Let’s create the Home component file structure first:

ng g component home --spec false



Now, let’s modify the home.component.html file:

<section fxLayout="column wrap">

<div fxFlexAlign="center">

<p>Welcome to the Material Angular OwnerAccount Application</p>

</div>

<p>In this applicatoin we are going to work with:</p>

</section>

We need to modify the app.component.html file:

<app-layout>

<app-home></app-home>

</app-layout>

And we need to modify the home.component.css file:

section div p{

color: #3f51b5;

font-size: 30px;

text-shadow: 2px 3px 5px grey;

margin: 30px 0;

}

section div + p{

color: #3f51b5;

font-weight: bold;

font-size: 20px;

padding-bottom: 20px;

}

To use our first material component, the mat-tab component, we need to register it inside the material.module.ts file:

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

import { CommonModule } from '@angular/common';

import { MatTabsModule } from '@angular/material';

@NgModule({

imports: [

CommonModule,

MatTabsModule

],

exports: [

MatTabsModule

],

declarations: []

})

export class MaterialModule { }

And then to modify the home.component.html file:

<section fxLayout="column wrap">

<div fxFlexAlign="center">

<p>Welcome to the Material Angular OwnerAccount Application</p>

</div>

<p>In this application we are going to work with:</p>

<mat-tab-group>

<mat-tab label="Material Components">

<p>We are going to use different material components to create nice looking angular app.</p>

</mat-tab>

<mat-tab label="Consume .NET Core Web API">

<p>We will consume our .NET Core applicatoin. Basicaly, we will create complete CRUD client app.</p>

</mat-tab>

<mat-tab label="Fully responsive navigation menu">

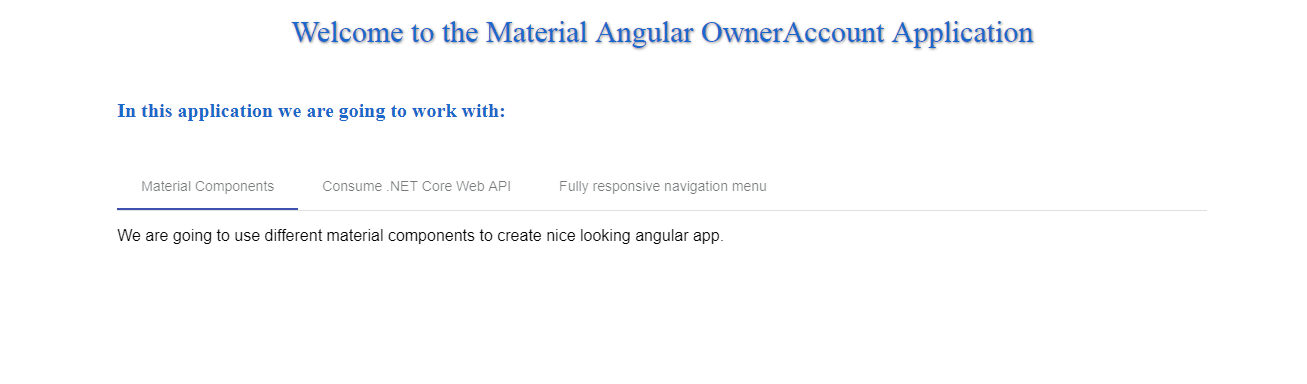
<p>By using material components, we are going to create fully responsive navigation menu, with its side-bar as well.</p>

</mat-tab>

</mat-tab-group>

</section>

Now, we can inspect our result:



## Additional Mat-Tab Features

First of all, let’s style our tab content a little more, to center our text inside every tab:

mat-tab-group {

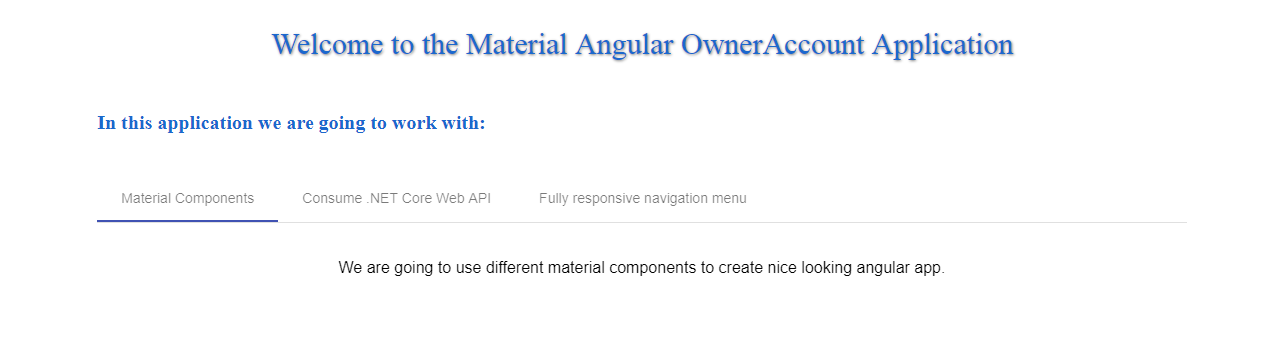
text-align: center;

}

mat-tab-group p {

padding-top: 20px;

}



This control has its own events. The selectedTabChange event is emmited when the active tab changes. The focusChange event is emitted when the user navigates through tabs with a keyboard navigation.

So, let’s use the selectedTabChange event:

<mat-tab-group (selectedTabChange)="executeSelectedChange($event)">

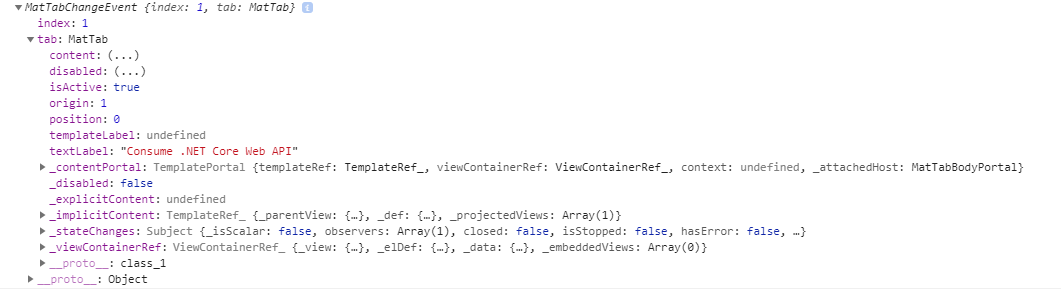
And we need to modify the home.component.ts file:

public executeSelectedChange = (event) => {

console.log(event);

}

Right now, as soon as we switch our tabs, we will see the event object logged into the console window:



## Conclusion

Everything looks great. We have our first material component and in the following articles, we will work with other material components as well.

In this article we have learned:

* How to prepare the Angular project,
* How to install Angular Material, CDK, and Animations
* The way to install and use the @angular/flex-layout library
* To work with the Tab Material component

In the next article, we are going to learn more about creating a navigation menu and menu side-bar with Angular routing.

# Angular Material Navigation Manu

Every application needs to have some sort of navigation, to enable users better navigation experience. Creating such a navigation menu will be our goal in this article. We have to use angular routing as well, and we are going to use it, but we won’t dive deep inside the routing concept. If you want to learn in more detail about the angular routing, you can read this article [Angular Navigation And Routing](https://code-maze.com/net-core-web-development-part8/).

Because this series is above all about the angular material, this article won’t be an exception. We will mainly focus on creating a navigation menu by using different material components. Once we are done, we will have a fully responsive and functional navigation menu with the routing logic to support the complete process.

## Creating Routes

Let’s start with creating a new routing module:

ng g module routing --spec false --module app



Now, let’s modify the routing.module.ts file:

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

import { CommonModule } from '@angular/common';

import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from '../home/home.component';

const routes: Routes = [

{ path: 'home', component: HomeComponent},

{ path: '', redirectTo: '/home', pathMatch: 'full' }

];

@NgModule({

imports: [

CommonModule,

RouterModule.forRoot(routes)

],

exports: [

RouterModule

],

declarations: []

})

export class RoutingModule { }

Finally, let’s modify the app.compnent.html file to complete routing part for now:

<app-layout>

<main>

<router-outlet></router-outlet>

</main>

</app-layout>

We should be able to see our home component again, but this time it is served on the /home route.

## Starting Navigation Development

Angular Material provides different components which we can use to create nicely styled, responsive and effective navigation in our app. But we need to start with something, aren’t we? So, let’s start with app.component.html modification by using the mat-sidenav-container component:

<app-layout>

<mat-sidenav-container>

<mat-sidenav #sidenav role="navigation">

<!--this is a place for us to add side-nav code-->

</mat-sidenav>

<mat-sidenav-content>

<!--in here all the content must reside. We will add a navigation header as well-->

<main>

<router-outlet></router-outlet>

</main>

</mat-sidenav-content>

</mat-sidenav-container>

</app-layout>

So, what we do is creating a container for our side navigation bar and specifying the part for our content. As we can see the <mat-sidenav> defines a place for a side navigation and the <mat-sidenav-content> defines a place for our content. We need to use the local reference #sidenav, and a little bit later, we will see why.

Of course, this won’t work. What we are missing is the module registration in the material.module.ts file:

import { MatTabsModule, MatSidenavModule } from '@angular/material';

@NgModule({

imports: [

CommonModule,

MatTabsModule,

MatSidenavModule

],

exports: [

MatTabsModule,

MatSidenavModule

],

Now, we should have a working application again with some grayish background. Let’s style this a bit in the app.component.css file:

mat-sidenav-container, mat-sidenav-content, mat-sidenav {

height: 100%;

}

mat-sidenav {

width: 250px;

}

main {

padding: 10px;

}

And let’s modify the styles.css file:

/\* for sidenav to take whole page \*/

html, body {

margin: 0;

height: 100%;

}

That is it. We have all prepared and it is time to start working on our navigation header component.

## Creating Navigation Header

To create a navigation header, we need to use the mat-toolbar element. But first thing first.

This component has its own module, so we need to register that module inside the material.module.ts file:

import { ..., MatToolbarModule } from '@angular/material';

imports: [

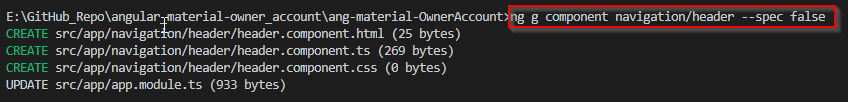
MatToolbarModule,

exports: [

MatToolbarModule,

After that, we are going to create a new header component:

ng g component navigation/header --spec false



Now it is time to include this component inside the app.component.html file, right above the <main> tag:

<mat-sidenav-content>

<app-header></app-header>

<main>

<router-outlet></router-outlet>

</main>

</mat-sidenav-content>

Then, let’s modify the header.component.html file:

<mat-toolbar color="primary">

<div fxHide.gt-xs>

<button mat-icon-button (click)="onToggleSidenav()">

<mat-icon>menu</mat-icon>

</button>

</div>

<div>

<a routerLink="/home">Owner-Account</a>

</div>

<div fxFlex fxLayout fxLayoutAlign="end" fxHide.xs>

<ul fxLayout fxLayoutGap="15px" class="navigation-items">

<li>

<a routerLink="/owner">Owner Actions</a>

</li>

<li>

<a routerLink="/account">Account Actions</a>

</li>

</ul>

</div>

</mat-toolbar>

Basically, we create our navigation with the menu icon (we still need to register its own module inside the material module), and the Owner-Account part that leads to the home component. As we can see, we use the fxHide.gt-xs directive, which states that this part should be hidden only on the screen that is greater than extra small.

We have another part of navigation which is positioned on the end of the navbar and hidden only for the extra small screen.

To continue, let’s register the MatIconModule and MatButtonModule inside the material module file:

import { ... MatIconModule, MatButtonModule } from '@angular/material';

imports: [

MatButtonModule,

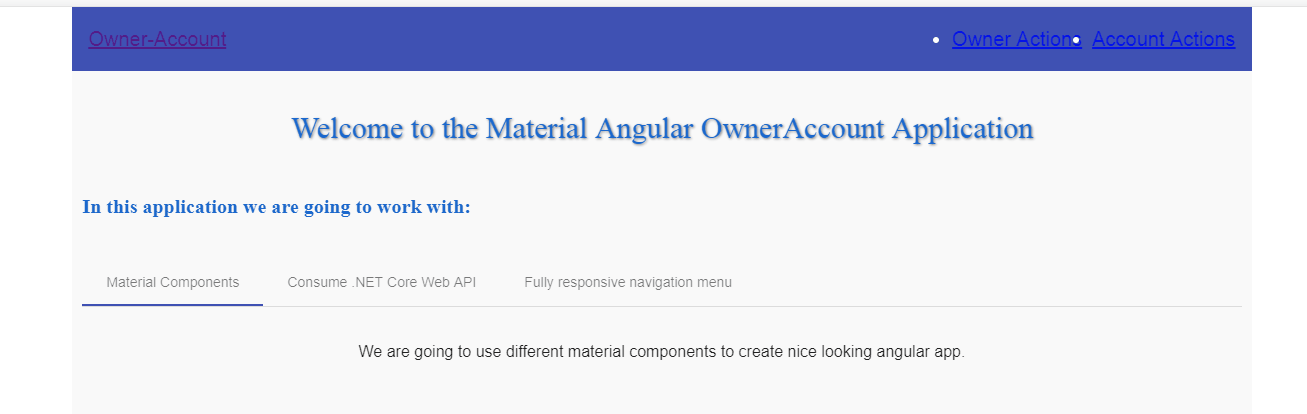
MatIconModule,

exports: [

MatButtonModule,

MatIconModule,

Right now, we have our menu that looks like this:



Looking beautiful right? :D :D Of course not, but we have our starting functionality in place and we are going to make it much nicer.

So, let’s continue and modify the header.component.css file:

a {

text-decoration: none;

color: white;

}

a:hover, a:active{

color: lightgray;

}

.navigation-items{

list-style-type: none;

padding: 0;

margin: 0;

}

mat-toolbar{

border-radius: 3px;

}

@media(max-width: 959px){

mat-toolbar{

border-radius: 0px;

}

}

Now if we look at our menu, it looks much nicer, isn’t it?



If we take a look at our icon button, we are going to see the onToggleSidenav()" event. We need to implement that inside the header.component.ts file:

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

@Component({

selector: 'app-header',

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

styleUrls: ['./header.component.css']

})

export class HeaderComponent implements OnInit {

@Output() public sidenavToggle = new EventEmitter();

constructor() { }

ngOnInit() {

}

public onToggleSidenav = () => {

this.sidenavToggle.emit();

}

}

If you want to learn more about @Output directives, you can read [Angular Series Article About Decorators.](https://code-maze.com/net-core-web-development-part12/)

Finally, we have to react on this event emitter inside our app.component.html file:

<mat-sidenav-content>

<app-header (sidenavToggle)="sidenav.toggle()"></app-header>

<main>

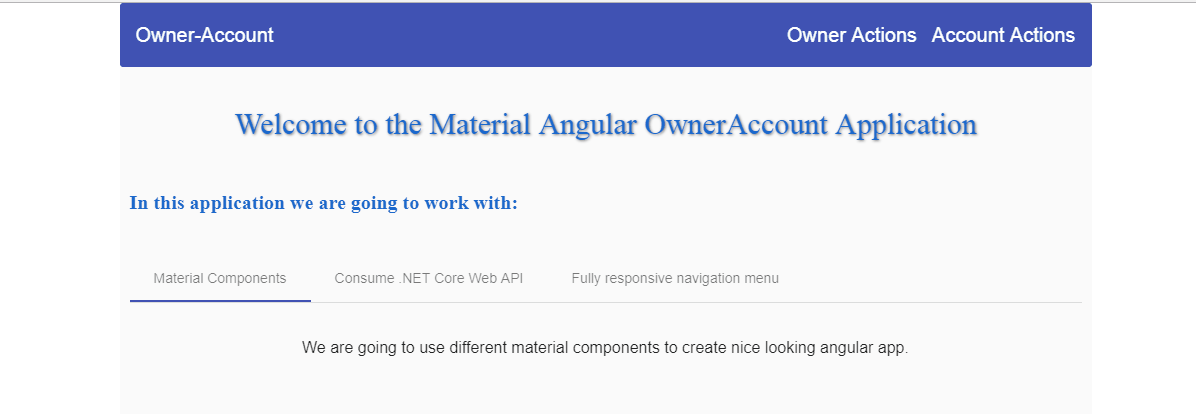
<router-outlet></router-outlet>

</main>

</mat-sidenav-content>

Now we see why we need the #sideNav local reference inside the mat-sidenav component.

Now let’s see how this looks like:



Excellent. Now its time for us to start working on the side-nav.

## Creating Side-Navigation

To create the side navigation items, we are going to use the mat-nav-list element that resides inside the MatListModule. So, let’s register this module first in the material.module.ts file:

import { … MatListModule } from '@angular/material';

imports: [

MatListModule,

exports: [

MatListModule,

Then let's create the sidenav-list component and modify the sidenav-list.component.html file:

ng g component navigation/sidenav-list -- spec false

<mat-nav-list>

<a mat-list-item routerLink="/home" (click)="onSidenavClose()">

<mat-icon>home</mat-icon> <span class="nav-caption">Home</span>

</a>

<a mat-list-item routerLink="/owner" (click)="onSidenavClose()">

<mat-icon>assignment\_ind</mat-icon> <span class="nav-caption">Owner Actions</span>

</a>

<a mat-list-item routerLink="#" (click)="onSidenavClose()">

<mat-icon>account\_balance</mat-icon><span class="nav-caption">Account Actions</span>

</a>

</mat-nav-list>

As we can see, we use the mat-nav-list as a container with all the anchor tags with mat-list-item attributes. We have the click event for every link, to close the side-nav once it's clicked, and we have an icon next to the link text.

Let’s continue by adding some styles to the sidenav-list.component.css file:

a {

text-decoration: none;

color: white;

}

a:hover, a:active{

color: lightgray;

}

.nav-caption{

display: inline-block;

padding-left: 6px;

}

And finally let’s modify the sidenav-list.component.ts file:

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

@Component({

selector: 'app-sidenav-list',

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

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

})

export class SidenavListComponent implements OnInit {

@Output() sidenavClose = new EventEmitter();

constructor() { }

ngOnInit() {

}

public onSidenavClose = () => {

this.sidenavClose.emit();

}

}

That’s it. We can now open the app.component.html file and modify it to add the side-nav component:

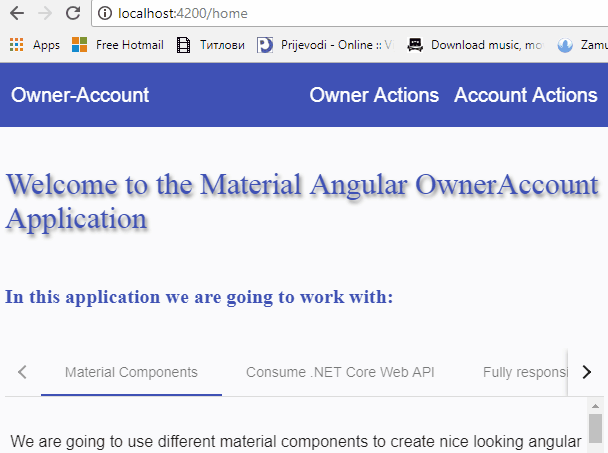
<mat-sidenav #sidenav role="navigation">

<app-sidenav-list (sidenavClose)="sidenav.close()"></app-sidenav-list>

</mat-sidenav>

In this code, we react on the event emitter form the sidenav-list component and close the side-nav by using #sidenav local reference.

Now, all we have to do is to take a look at our result:



## Multi-Menu in Side-Nav

There is one more thing we want to show you. For now, we only have a one clickable link per section, inside ou sidenav. But what if we want to have a menu item and when we click that menu item other options appear? Well, we are going to show you how to do that as well.

So, in the sidenav-list.component.html file, we need to add the following code below the last anchor tag:

<mat-list-item [matMenuTriggerFor]="menu">

<mat-icon>unfold\_more</mat-icon>

<a matline>Example</a>

</mat-list-item>

<mat-menu #menu="matMenu">

<button mat-menu-item (click)="onSidenavClose()">View profile</button>

<button mat-menu-item (click)="onSidenavClose()">Add contact</button>

</mat-menu>

For this to work, we need to register the MatMenuModule:

import { …, MatMenuModule } from '@angular/material';

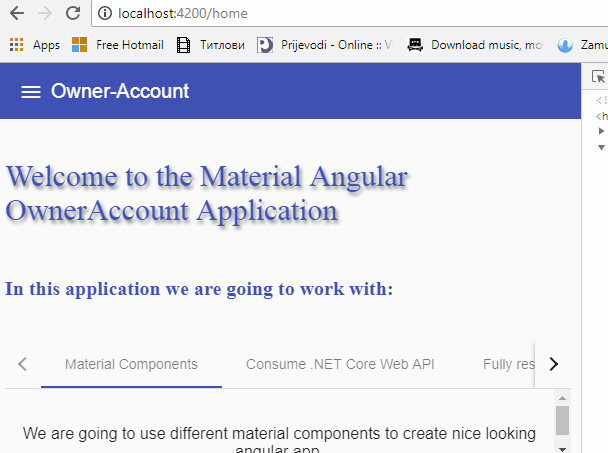
imports: [

MatMenuModule,

exports: [

MatMenuModule,

And the result:



## Conclusion

Awesome.

Now we have our own responsive navigation menu, built from scratch.

…

# Angular Material Table, Filter, Sort, Paging

We are going to divide this article into two major parts. First part will consist of creating environment files, HTTP repository service and creating the new Owner module with the lazy loading feature. As we can see, all of this is Angular specifics, so we won’t dive too deep into this sections. We already have our [Angular Series](https://code-maze.com/angular-series/) in which we talked about these topics in great detail. So if you are not familiar with these topics, we strongly recommend reading the mentioned series.

In our source code, we can find the folder OwnerAccountServer which contains entire .NET Core project, which we have created in [.NET Core Series](https://code-maze.com/net-core-series/). In the same folder, we can find the \_MySQL\_Init\_Script folder which contains a script to initialize data for MySQL database. Just run that script in the MySQL database and you are ready to go.

The second part will consist of creating a material table and populating that table with data from our server. Furthermore, we are going to create the filter, sorting and paging functionalities for that data.

It’s time to start our job.

## Environment, HTTP and Owner Module

Let’s start with the environment files modification.

We are going to modify the environment.prod.ts file first:

export const environment = {

production: true,

urlAddress: 'http://www.ang-material-account-owner.com'

};

After that, let’s modify the environment.ts file:

export const environment = {

production: false,

urlAddress: 'http://localhost:5000'

};

Then, we need to create a service for sending the HTTP requests towards our server.

To do that, we are going to create a service file first:

ng g service shared/repository --spec false

After creation, we have to modify that file:

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

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { environment } from './../../environments/environment';

@Injectable({

providedIn: 'root'

})

export class RepositoryService {

constructor(private http: HttpClient) { }

public getData(route: string) {

return this.http.get(this.createCompleteRoute(route, environment.urlAddress));

}

public create(route: string, body) {

return this.http.post(this.createCompleteRoute(route, environment.urlAddress), body, this.generateHeaders());

}

public update(route: string, body){

return this.http.put(this.createCompleteRoute(route, environment.urlAddress), body, this.generateHeaders());

}

public delete(route: string){

return this.http.delete(this.createCompleteRoute(route, environment.urlAddress));

}

private createCompleteRoute(route: string, envAddress: string) {

return `${envAddress}/${route}`;

}

private generateHeaders() {

return {

headers: new HttpHeaders({'Content-Type': 'application/json'})

}

}

}

Excellent. We have prepared our repository file. If you want to learn more about environment files, services, and HTTP, you can read it in the [Angular Series Article](https://code-maze.com/net-core-web-development-part9/) which covers all of these topics.

One more thing that we need to do is to register HttpClientModule in the app.module.ts file:

import { HttpClientModule } from '@angular/common/http';

imports: [

…

HttpClientModule

],

### Creating a New Owner Module

Let’s create a new Owner module, and the routes for that module as well:

ng g module owner --spec false

We are going to register this module into the main routing module but in such a way to support lazy loading:

const routes: Routes = [

{ path: 'home', component: HomeComponent},

{ path: 'owner', loadChildren: "./../owner/owner.module#OwnerModule" },

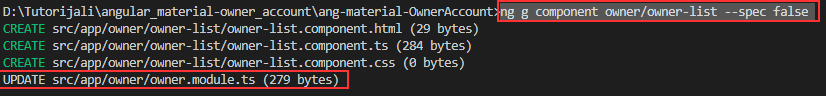
{ path: '', redirectTo: '/home', pathMatch: 'full' }

];

To read more about multiple modules in the angular project and lazy loading as well, visit [Lazy Loading in Angular](https://code-maze.com/net-core-web-development-part10/).

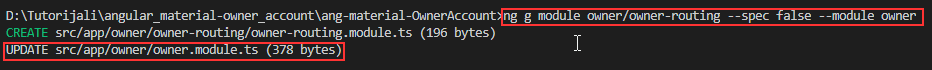
Right now, we have to create a new component to show the list of all the owners from the database:

ng g component owner/owner-list --spec false



We need to have a routing for the components inside this module, so let’s create a new routing module for the Owner module components:

ng g module owner/owner-routing --spec false --module owner



And let’s modfy that module file:

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

import { CommonModule } from '@angular/common';

import { Routes, RouterModule } from '@angular/router';

import { OwnerListComponent } from '../owner-list/owner-list.component';

const routes: Routes = [

{ path: 'owners', component: OwnerListComponent }

];

@NgModule({

imports: [

CommonModule,

RouterModule.forChild(routes)

],

exports: [

RouterModule

],

declarations: []

})

export class OwnerRoutingModule { }

Finally, to make all this to work, we need to modify our routes in the sidenav-list.component.html file:

<a mat-list-item routerLink="/owner/owners" (click)="onSidenavClose()">

<mat-icon>assignment\_ind</mat-icon> <span class="nav-caption">Owner Actions</span>

</a>

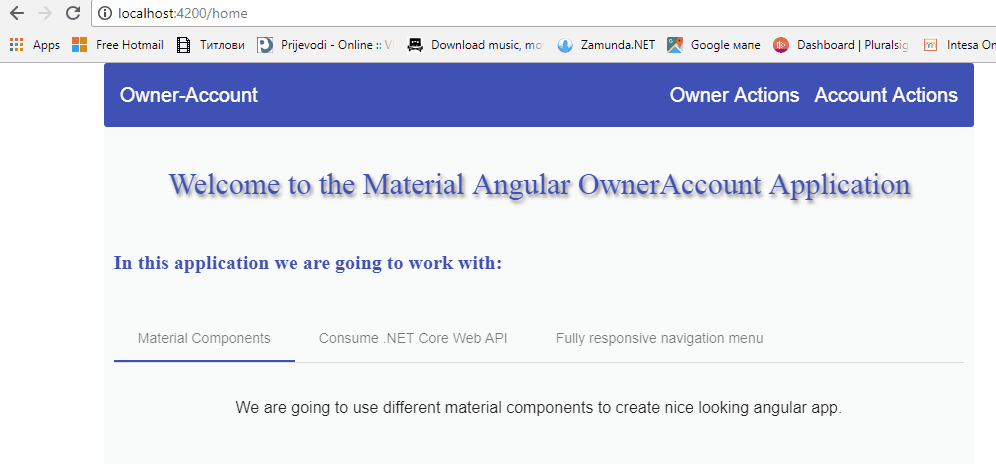
and the header.component.html file:

<li>

<a routerLink="/owner/owners">Owner Actions</a>

</li>

That is it. We can now check that all is good with our routing settings:



Excellent. Right now, we can dedicate our work to fetch some data from the database and showing them inside the material table component.

## Using Material Table to Show Data

Because we have created another module in our Angular app, we need to import the Material module file inside the owner.module.ts file:

import { MaterialModule } from './../material/material.module';

imports: [

…

MaterialModule

],

When we create the Shared module, we will fix this code repetition (MaterialModule inside the App module and Owner module).

Let’s continue by creating the \_interface folder and inside it the owner.model.ts file:

export interface Owner{

id: string;

name: string;

dateOfBirth: Date;

address: string;

}

Because we want to use the material table component, we need to register its own module in the material.module.ts file:

import { …, MatTableModule } from '@angular/material';

imports: [

MatTableModule,

exports: [

MatTableModule,

Then, let’s modify the owner-list.material.component file:

<table mat-table [dataSource]="dataSource">

<ng-container matColumnDef="name">

<th mat-header-cell \*matHeaderCellDef> Name </th>

<td mat-cell \*matCellDef="let element"> {{element.name}} </td>

</ng-container>

<ng-container matColumnDef="dateOfBirth">

<th mat-header-cell \*matHeaderCellDef> Date of Birth </th>

<td mat-cell \*matCellDef="let element"> {{element.dateOfBirth | date}} </td>

</ng-container>

<ng-container matColumnDef="address">

<th mat-header-cell \*matHeaderCellDef> Address </th>

<td mat-cell \*matCellDef="let element"> {{element.address}} </td>

</ng-container>

<ng-container matColumnDef="details">

<th mat-header-cell \*matHeaderCellDef> Details </th>

<td mat-cell \*matCellDef="let element">

<button mat-icon-button color="primary" (click)="redirectToDetails(element.id)">

<mat-icon class="mat-18">reorder</mat-icon>

</button>

</td>

</ng-container>

<ng-container matColumnDef="update">

<th mat-header-cell \*matHeaderCellDef> Update </th>

<td mat-cell \*matCellDef="let element">

<button mat-icon-button color="accent" (click)="redirectToUpdate(element.id)">

<mat-icon class="mat-18">system\_update</mat-icon>

</button>

</td>

</ng-container>

<ng-container matColumnDef="delete">

<th mat-header-cell \*matHeaderCellDef> Delete </th>

<td mat-cell \*matCellDef="let element">

<button mat-icon-button color="warn" (click)="redirectToDelete(element.id)">

<mat-icon class="mat-18">delete</mat-icon>

</button>

</td>

</ng-container>

<tr mat-header-row \*matHeaderRowDef="displayedColumns"></tr>

<tr mat-row \*matRowDef="let row; columns: displayedColumns;"></tr>

</table>

The mat-table element transforms this table to a material one. With the dataSource attribute, we provide a data source for our table that we want to show. Inside every ng-container tag, we define the column definition and the value to be displayed. It is very important to match the matColumnDef value with the property name of our Owner interface.

Finally, in the last two tr tags, we define an order for our header columns and the row definitions. So, what we need to do right now is to create our datasource and displayedColumns property in the ownerlist.component.ts file:

import { RepositoryService } from './../../shared/repository.service';

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

import { MatTableDataSource } from '@angular/material';

import { Owner } from '../../\_interface/owner.model';

@Component({

selector: 'app-owner-list',

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

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

})

export class OwnerListComponent implements OnInit {

public displayedColumns = ['name', 'dateOfBirth', 'address', 'details', 'update', 'delete'

];

public dataSource = new MatTableDataSource<Owner>();

constructor(private repoService: RepositoryService) { }

ngOnInit() {

this.getAllOwners();

}

public getAllOwners = () => {

this.repoService.getData('api/owner')

.subscribe(res => {

this.dataSource.data = res as Owner[];

})

}

public redirectToDetails = (id: string) => {

}

public redirectToUpdate = (id: string) => {

}

public redirectToDelete = (id: string) => {

}

}

If we change the order of elements inside the displayedColumns array, it will change the order of the columns inside our table.

Right now, if we start our application and navigate to the Owner Actions menu, we are going to see a populated material table. But we are missing some styles, so let's add those in the owner-list.component.css file:

table {

width: 100%;

overflow-x: auto;

overflow-y: hidden;

min-width: 500px;

}

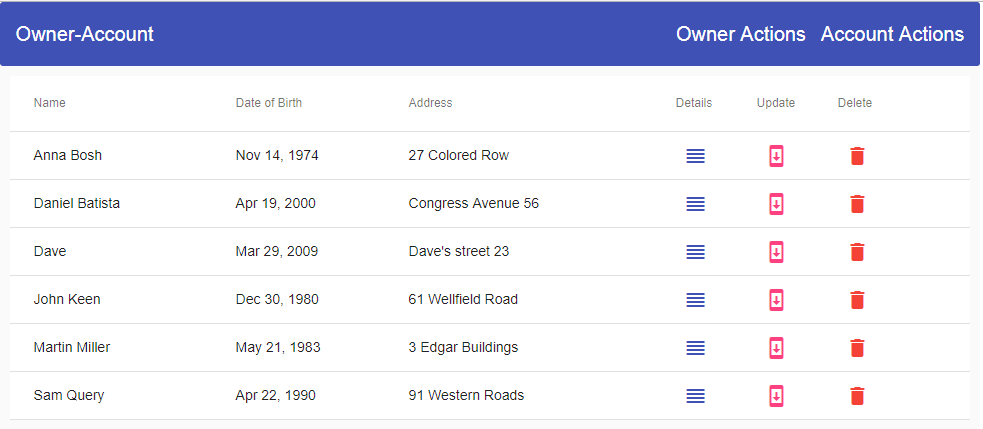
th.mat-header-cell {

text-align: left;

max-width: 300px;

}

Now we should have a better-styled table:



## Sorting Data in Material Table

We want to add the sorting functionality to our table, and for that purpose, we are going to use the matSort directive on the table tag. Moreover, we need to place the mat-sort-header directive for each header cell that will trigger sorting.

So, let’s do that now.

We are going to modify the table tag first:

<table mat-table [dataSource]="dataSource" matSort>

And then, we are going to add the mat-sort-header directive to the th Name, DateOfBirth, and Address tags:

<th mat-header-cell \*matHeaderCellDef mat-sort-header> Name </th>

<th mat-header-cell \*matHeaderCellDef mat-sort-header> Date of Birth </th>

<th mat-header-cell \*matHeaderCellDef mat-sort-header> Address </th>

If we want our sorting to work, we need to modify the owner-list.component.ts file as well:

export class OwnerListComponent implements OnInit, AfterViewInit {

public displayedColumns = ['name', 'dateOfBirth', 'address', 'details', 'update', 'delete'];

public dataSource = new MatTableDataSource<Owner>();

@ViewChild(MatSort) sort: MatSort;

constructor(private repoService: RepositoryService) { }

ngOnInit() {

this.getAllOwners();

}

ngAfterViewInit(): void {

this.dataSource.sort = this.sort;

}

.

.

.

Lastly, we need to add the MatSortModule inside the material.module.ts file:

import {..., MatSortModule } from '@angular/material';

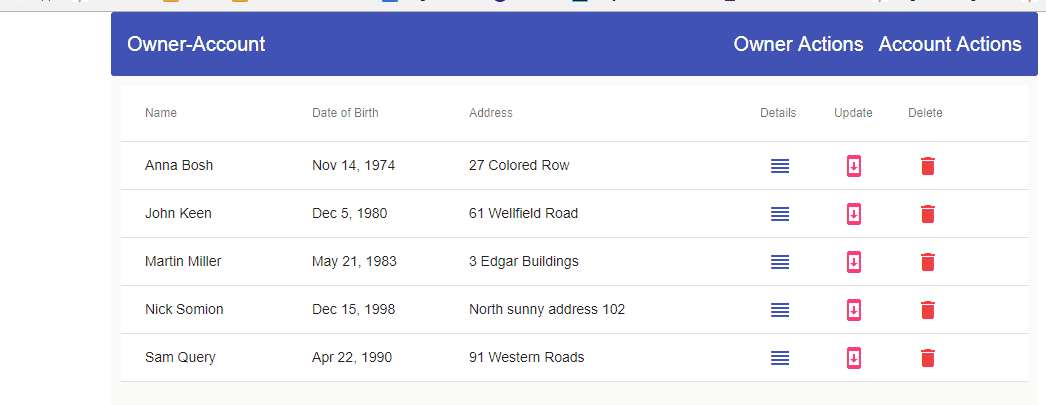
imports: [

MatSortModule,

exports: [

MatSortModule,

Now, we can check our result:



By default, sorting starts with ascending order first and then descending. We can change that behavior by adding the matSortStart attribute to desc next to the matSort directive:

<table mat-table [dataSource]="dataSource" matSort matSortStart="desc">

If we don’t want to use MatTableDataSource for sorting, but to provide our own sorting logic, we can use the (matSortChange) event to receive the active sorting column and the sorting order as well:

<table mat-table [dataSource]="dataSource" matSort (matSortChange)="customSort($event)">

This will produce the following JSON result, if we click on the name column:

*{active: "name", direction: "asc"}*

* 1. active:"name"
  2. direction:"asc"
  3. \_\_proto\_\_:Object

## Filter Functionality in Material Table

For this functionality, we need to provide our own input field and a custom function to filter our data. Only then, we can use MatTableDataSource’s filter property. To implement filtering, we are going to add this code right above our table in html file:

<div fxLayout fxLayoutAlign="center center">

<mat-form-field fxFlex="40%">

<input matInput type="text" (keyup)="doFilter($event.target.value)" placeholder="Filter">

</mat-form-field>

</div>

And then to add this function in the component file:

public doFilter = (value: string) => {

this.dataSource.filter = value.trim().toLocaleLowerCase();

}

Finally, because we are using the matInput directive to transform regular input into the material input field, we need to register its modules inside the material.module.ts file:

import { ..., MatFormFieldModule, MatInputModule } from '@angular/material';

imports: [

MatFormFieldModule,

MatInputModule,

exports: [

MatFormFieldModule,

MatInputModule,

As we can see from the html file, we are using the fxLayout directive. But, because this component is part of the new Owner module, we need to import FlexLayoutModule into the Owner module file as well:

import { FlexLayoutModule } from '@angular/flex-layout';

imports: [

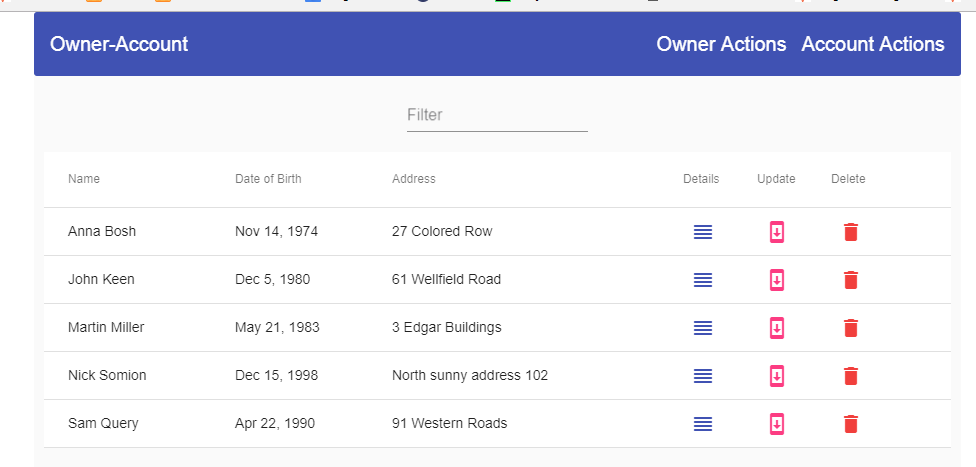
...

FlexLayoutModule

],

Excellent.

Now we can inspect the result:



## Paging Functionality

To implement this functionality, we need to use a <mat-paginator> bellow our table. So, let’s start implementation by adding MatPaginatorModule inside the Material module:

import {..., MatPaginatorModule } from '@angular/material';

imports: [

MatPaginatorModule,

exports: [

MatPaginatorModule,

Then, let’s add the mat-paginator inside the html file:

<mat-paginator [pageSize]="2" [pageSizeOptions]="[2, 4, 6, 10, 20]">

</mat-paginator>

And finally, let’s modify the owner-list.component.ts file:

import { MatTableDataSource, MatSort, MatPaginator } from '@angular/material';

...

@ViewChild(MatPaginator) paginator: MatPaginator;

constructor(private repoService: RepositoryService) { }

ngOnInit() {

this.getAllOwners();

}

ngAfterViewInit(): void {

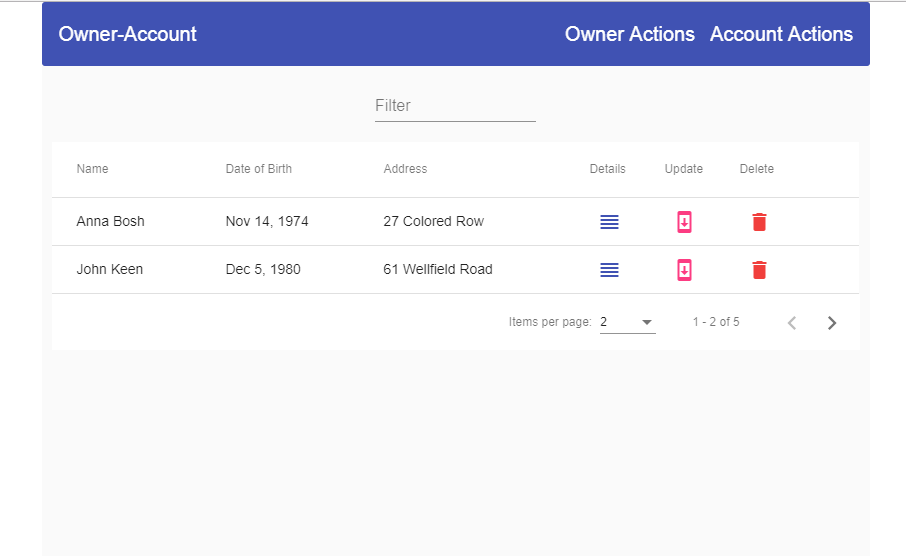
this.dataSource.sort = this.sort;

this.dataSource.paginator = this.paginator;

}

...

This is our result:



If we want to write our custom pagination logic, we should use the (page) output event:

<mat-paginator [pageSize]="2" [pageSizeOptions]="[2, 4, 6, 10, 20]" (page)="pageChanged($event)">

</mat-paginator>

## Conclusion

So, that’s it. Now we have our material table with all the features like sorting, paging and filtering data.

In this article, we have learned:

* How to use Environment files, HTTP client module, and Lazy Loading feature
* To create a material table
* To apply sorting, filtering, and pagination to the material table

In the next article, we are going to create the error pages by focusing on the material components and to create owner details component.

# Angular Material Progress Bar, Spinner, CheckBox, Card, Select, Expansion Panel – Create Error and Details Pages

In this article, we are going to create the error pages and the details component as well but with the main focus on the material components.

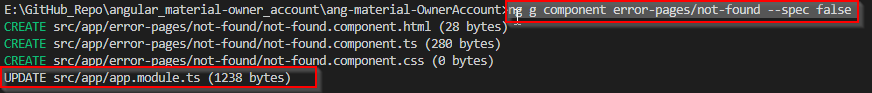
We are going to create the Not-Found component, the Server-Error component, and finally the Owner-Details component.

So, let’s start.

## Not-Found Component and Using Progress Bar

The first thing we need to do is to create a new not-found component:

ng g component error-pages/not-found --spec false



After that, we are going to change the routes in the main routing module:

import { NotFoundComponent } from '../error-pages/not-found/not-found.component';

const ownerRoutes: Routes = [

{ path: 'home', component: HomeComponent},

{ path: 'owner', loadChildren: "./../owner/owner.module#OwnerModule" },

{ path: '404', component: NotFoundComponent},

{ path: '', redirectTo: '/home', pathMatch: 'full' },

{ path: '\*\*', redirectTo: '/404', pathMatch: 'full'}

];

Now, if we try to load some unexisting address, we will get the NotFound component instead, with the “not-found works” message.

Of course, we don’t want this message, so we are going to modify the not-found.component.html file:

<section fxLayout="column wrap" fxLayoutGap="60px" fxLayoutAlign="center center">

<div fxFlex>

404 We are searching for your page...

</div>

<div fxFlex>

<mat-progress-bar mode="indeterminate"></mat-progress-bar>

</div>

<div fxFlex>

... But we can not find it.

</div>

</section>

As we can see, we are using the mat-progress-bar material component, and for that, we need to import the required module into the material.module.ts file:

import {..., MatProgressBarModule } from '@angular/material';

imports: [

MatProgressBarModule,

exports: [

MatProgressBarModule,

Finally, let’s add some styles to the not-found.component.css file:

section div:nth-child(1), section div:nth-child(3){

color: blue;

font-size: 50px;

}

section div:nth-child(1){

margin-top: 20px;

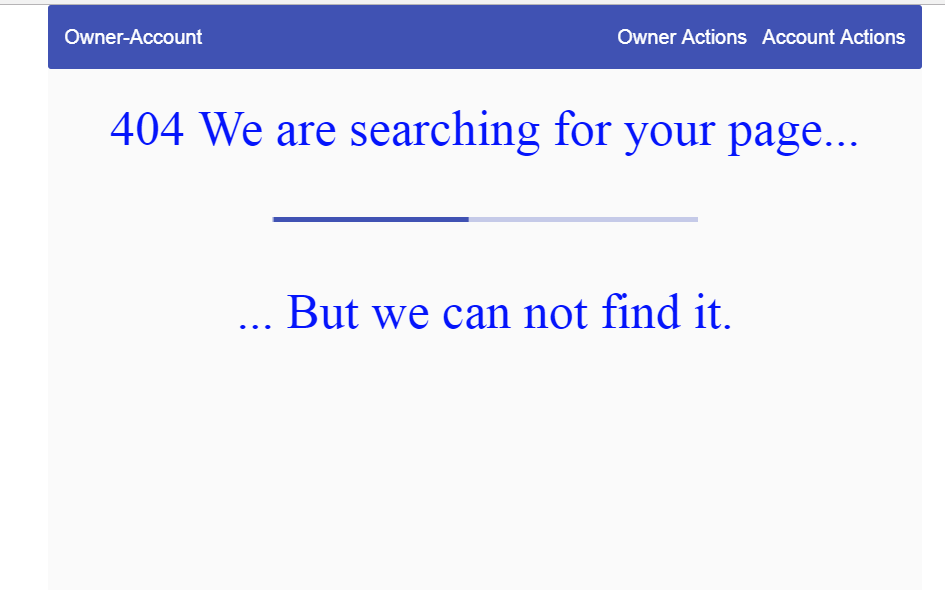
}

section div:nth-child(2){

width: 50%;

}

Now, we can check our result if we type none-existing url (localhost:4200/something):



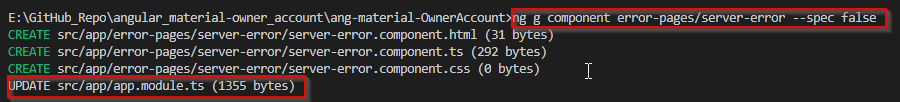
This looks good.

Let’s continue with the Server-Error component.

## Server-Error Component –Spinner and CheckBox Material Components

What we need to do first is to create the Server-Error components:

ng g component error-pages/server-error --spec false



After that let’s modify the routing file:

import { ServerErrorComponent } from '../error-pages/server-error/server-error.component';

const ownerRoutes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'owner', loadChildren: "./../owner/owner.module#OwnerModule" },

{ path: '404', component: NotFoundComponent },

{ path: '500', component: ServerErrorComponent },

{ path: '', redirectTo: '/home', pathMatch: 'full' },

{ path: '\*\*', redirectTo: '/404', pathMatch: 'full' }

];

Then, we need to modify the server-error.component.html file:

<section fxLayout="column wrap" fxLayoutAlign="center center" fxLayoutGap="30px">

<div fxFlex>

<p>500 Server Error</p>

<p>We are sorry for the inconvinience, plese report this error.</p>

</div>

<div fxFlex>

<mat-checkbox (change)="checkChanged($event)" color="primary">I want to report this error.</mat-checkbox>

</div>

<div fxFlex \*ngIf="reportedError">

<mat-progress-spinner mode="determinate" [value]="errorPercentage"></mat-progress-spinner>

<h1>{{errorPercentage}}%</h1>

</div>

</section>

Because we are using the checkbox and progress-spinner components, we need to import their modules into the material.module.ts file:

import {..., MatCheckboxModule, MatProgressSpinnerModule } from '@angular/material';

imports: [

MatProgressSpinnerModule,

exports: [

MatProgressSpinnerModule,

Then, let’s modify the server-error.component.ts file:

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

@Component({

selector: 'app-server-error',

templateUrl: './server-error.component.html',

styleUrls: ['./server-error.component.css']

})

export class ServerErrorComponent implements OnInit {

public reportedError: boolean;

public errorPercentage: number = 0;

public timer;

constructor() { }

ngOnInit() {

}

public checkChanged = (event) => {

this.reportedError = event.checked;

this.reportedError ? this.startTimer() : this.stopTimer();

}

private startTimer = () => {

this.timer = setInterval(() => {

this.errorPercentage += 1;

if (this.errorPercentage === 100) {

clearInterval(this.timer);

}

}, 30);

}

private stopTimer = () => {

clearInterval(this.timer);

this.errorPercentage = 0;

}

}

And finally, let’s modify the server-error.component.css file:

section div p:nth-child(1){

font-size: 50px;

text-align: center;

color: #f44336;

}

section div p:nth-child(2){

font-size: 20px;

text-align: center;

color: #3f51b5;

}

mat-checkbox {

color: #3f51b5;

}

section div h1{

text-align: center;

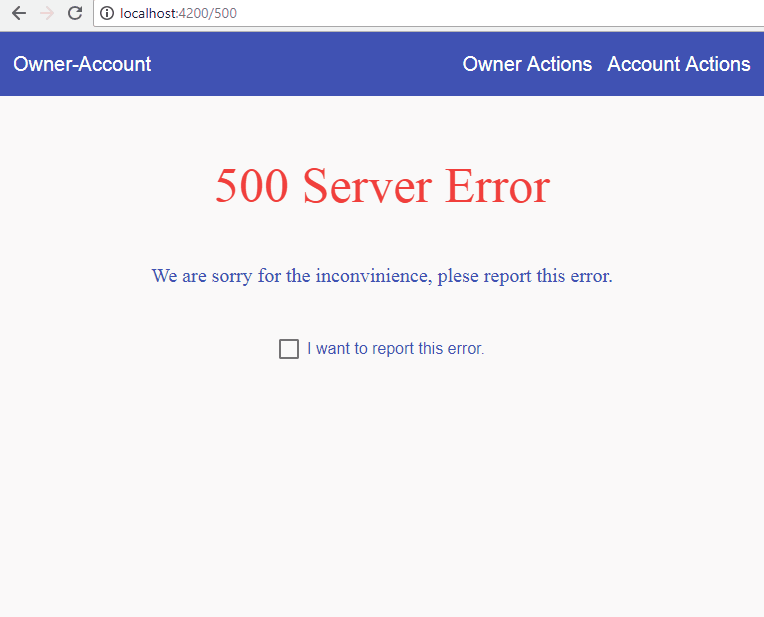
color: #3f51b5;

position: relative;

top: -85px;

}

Now we can check our result:



## Error Handling Service

It is not enough just to have the error pages, we need to handle errors and to redirect the user to the required page. For that, we are going to create an error-handler service.

We are just going to write our code, without any explanation because this is all angular specific. We have a great article [.NET Core 2.0, Angular and Mysql. Error Handling](https://code-maze.com/net-core-web-development-part11/) in which we have explained all related to the topic.

So, let’s create a service and modify it afterwards:

ng g service shared/error-handler --spec false

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

import { HttpErrorResponse } from '@angular/common/http';

import { Router } from '@angular/router';

@Injectable({

providedIn: 'root'

})

export class ErrorHandlerService {

public errorMessage: string = '';

constructor(private router: Router) { }

public handleError(error: HttpErrorResponse){

if(error.status === 500){

this.handle500Error(error);

}

else if(error.status === 404){

this.handle404Error(error)

}

else{

this.handleOtherError(error);

}

}

private handle500Error(error: HttpErrorResponse){

this.createErrorMessage(error);

this.router.navigate(['/500']);

}

private handle404Error(error: HttpErrorResponse){

this.createErrorMessage(error);

this.router.navigate(['/404']);

}

private handleOtherError(error: HttpErrorResponse){

this.createErrorMessage(error);

//TODO: this will be fixed later;

}

private createErrorMessage(error: HttpErrorResponse){

this.errorMessage = error.error ? error.error : error.statusText;

}

}

For now, this service can be implemented only in the owner-list component, so let’s do that:

import { ErrorHandlerService } from '../../shared/error-handler.service';

constructor(private repoService: RepositoryService, private errorService: ErrorHandlerService) { }

public getAllOwners = () => {

this.repoService.getData('api/owner')

.subscribe(res => {

this.dataSource.data = res as Owner[];

},

(error) => {

this.errorService.handleError(error);

})

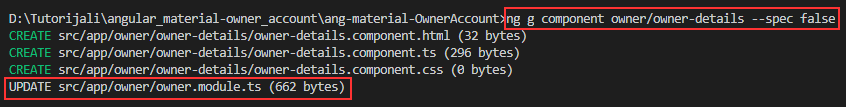
}

That is it. Now if our server returns the not found response we will redirect a user to the not found page. Same will happen for the internal server error, just another page.

## Details Page – Select and Expansion Panel

In this section, we are going to create a details functionality for our application. To do that, let’s first create the owner-details component:

ng g component owner/owner-details --spec false



Then, let’s configure the routes for this new component:

import { OwnerDetailsComponent } from '../owner-details/owner-details.component';

const routes: Routes = [

{ path: 'owners', component: OwnerListComponent },

{ path: 'details/:id', component: OwnerDetailsComponent}

];

After that, let’s modify the owner-list.component.ts file:

import { Router } from '@angular/router';

constructor(private repoService: RepositoryService, private errorService: ErrorHandlerService, private router: Router) { }

public redirectToDetails = (id: string) => {

let url: string = `/owner/details/${id}`;

this.router.navigate([url]);

}

Right now, we can navigate to our details page by clicking on the details button on the owner-list component. Now, we have to add additional interface and to change our existing one:

export interface Account{

id: string;

dateCreated: Date;

accountType: string;

ownerId?: string;

}

import { Account } from './account.model';

export interface Owner{

id: string;

name: string;

dateOfBirth: Date;

address: string;

accounts?: Account

}

After all these changes, we need to modify the owner-details component, to show our details data on the page.

So, let’s start with the owner-details.component.ts file:

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

import { Owner } from './../../\_interface/owner.model';

import { Router, ActivatedRoute } from '@angular/router';

import { RepositoryService } from './../../shared/repository.service';

import { ErrorHandlerService } from './../../shared/error-handler.service';

@Component({

selector: 'app-owner-details',

templateUrl: './owner-details.component.html',

styleUrls: ['./owner-details.component.css']

})

export class OwnerDetailsComponent implements OnInit {

public owner: Owner;

public showAccounts;

constructor(private repository: RepositoryService, private router: Router,

private activeRoute: ActivatedRoute, private errorHandler: ErrorHandlerService) { }

ngOnInit() {

this.getOwnerDetails();

}

private getOwnerDetails = () =>{

let id: string = this.activeRoute.snapshot.params['id'];

let apiUrl: string = `api/owner/${id}/account`;

this.repository.getData(apiUrl)

.subscribe(res => {

this.owner = res as Owner;

},

(error) =>{

this.errorHandler.handleError(error);

})

}

}

Excellent.

We have created a logic to fetch the details data and now, we need to show that data on the html page.

Because we are going to have a lot of code for this component, we are going to create two additional components to spread our html code between them.

That being said, let’s create those components:

ng g component owner/owner-details/owner-data --spec false

ng g component owner/owner-details/account-data --spec false

### Owner-Data Component

We are going to modify the owner-data component first.

The html part:

<section fxLayout="row wrap" fxLayoutAlign="center center">

<mat-card fxFlex="500px" fxFlex.xs="100%">

<mat-card-title>Details for the clicked owner</mat-card-title>

<mat-card-content>

<div fxLayout="column wrap" fxLayoutGap="40px">

<div fxLayout="row wrap" fxFlex>

<div fxFlex><strong>Owner's name:</strong></div>

<div fxFlex>{{owner?.name}}</div>

</div>

<div fxLayout="row wrap" fxFlex>

<div fxFlex><strong>Date of birth:</strong></div>

<div fxFlex>{{owner?.dateOfBirth | date}}</div>

</div>

<div fxLayout="row wrap" fxFlex \*ngIf='owner?.accounts.length <= 2; else advancedUser'>

<div fxFlex><strong>Type of user:</strong></div>

<div fxFlex class="beginner-color">Beginner user.</div>

</div>

<ng-template #advancedUser>

<div fxLayout="row wrap" fxFlex>

<div fxFlex><strong>Type of user:</strong></div>

<div fxFlex class="advanced-color">Advanced user</div>

</div>

</ng-template>

</div>

</mat-card-content>

<mat-card-actions>

<mat-form-field>

<mat-select placeholder="Show accounts" (selectionChange)="onChange($event)">

<mat-option \*ngFor="let opt of selectOptions" [value]="opt.value">

{{opt.name}}

</mat-option>

</mat-select>

</mat-form-field>

</mat-card-actions>

</mat-card>

</section>

Because we are using the mat-card component and the mat-select component, we need to import modules inside the material.module.ts file:

import { …, MatCardModule, MatSelectModule } from '@angular/material';

imports: [

MatSelectModule,

MatCardModule,

exports: [

MatSelectModule,

MatCardModule,

Now let’s modify the owner-data.component.ts file:

import { Owner } from './../../../\_interface/owner.model';

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

@Component({

selector: 'app-owner-data',

templateUrl: './owner-data.component.html',

styleUrls: ['./owner-data.component.css']

})

export class OwnerDataComponent implements OnInit {

@Input() public owner: Owner;

public selectOptions = [{name:'Show', value: 'show'}, {name: `Don't Show`, value: ''}];

@Output() selectEmitt = new EventEmitter();

constructor() { }

ngOnInit() {

}

public onChange = (event) => {

this.selectEmitt.emit(event.value);

}

}

And finally, let’s modify the css file:

mat-card-content, mat-card-title, mat-card-actions {

text-align: center;

}

mat-card-content{

padding-top: 20px;

padding-bottom: 20px;

}

.advanced-color{

color: #3f51b5;

}

.beginner-color{

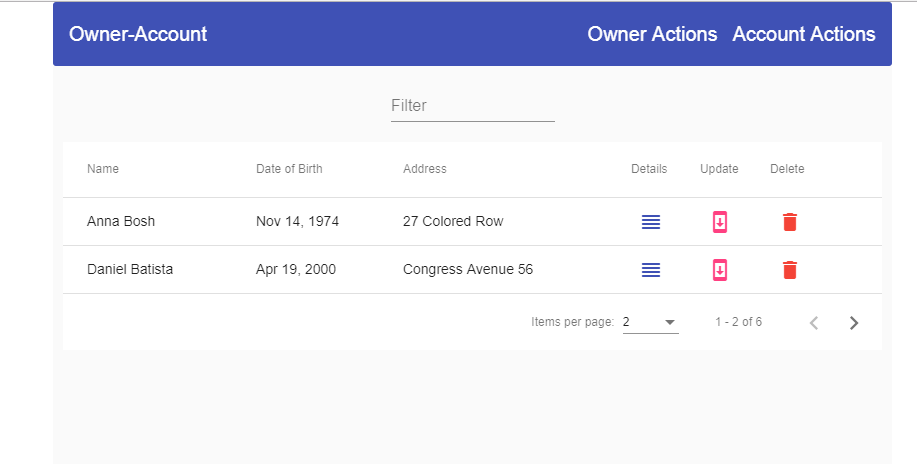
color: #f44336;

}

Now we need to include our components inside the owner-details component and check the current progress:

<app-owner-data [owner]='owner' (selectEmitt)='showAccounts = $event'></app-owner-data>

<app-account-data \*ngIf='showAccounts'></app-account-data>



### Account-Data Component

Let’s modify the html part of this component first:

<section fxLayout="row wrap" fxLayoutAlign="center center">

<mat-accordion fxFlex="500px" fxFlex.xs="100%">

<mat-expansion-panel \*ngFor="let account of accounts; let i = index">

<mat-expansion-panel-header>

<mat-panel-title>

{{i+1}}. Account

</mat-panel-title>

<mat-panel-description>

Account Information

</mat-panel-description>

</mat-expansion-panel-header>

<div fxLayout="row wrap" fxLayoutAlign="center center">

<div fxFlex="35%" class="text-color"><strong>type:</strong> &nbsp; {{account?.accountType}}</div>

<div fxFlex class="text-color"><strong>created:</strong> &nbsp; {{account?.dateCreated | date}}</div>

</div>

</mat-expansion-panel>

</mat-accordion>

</section>

Then, let’s import the module for the accordion:

import { …, MatExpansionModule } from '@angular/material';

imports: [

MatExpansionModule,

exports: [

MatExpansionModule,

After that, we need to modify the account-data.component.ts file:

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

@Component({

selector: 'app-account-data',

templateUrl: './account-data.component.html',

styleUrls: ['./account-data.component.css']

})

export class AccountDataComponent implements OnInit {

@Input() public accounts: Account[];

constructor() { }

ngOnInit() {

}

}

And, to modify the css file:

.text-color {

color: #3f51b5;

}

mat-accordion{

margin-top: 20px;

}

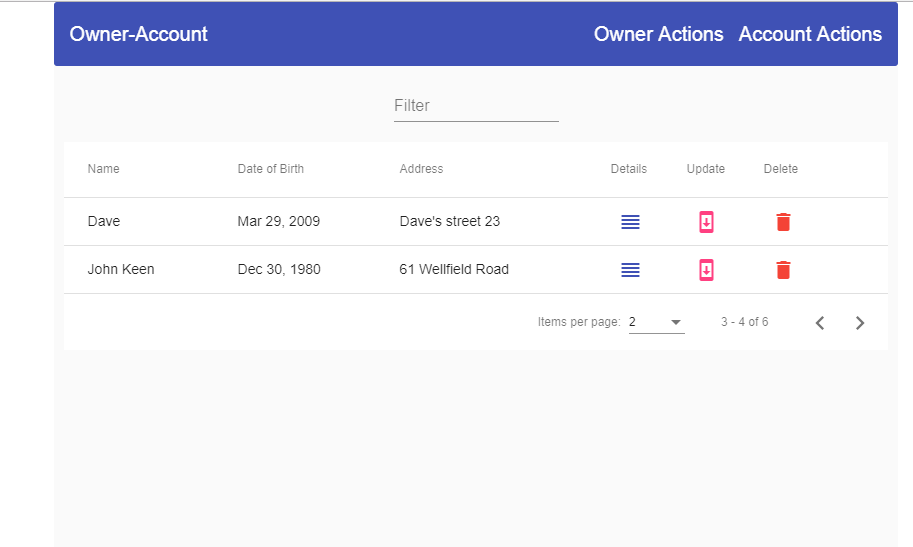
Excellent.

All we have left to do is to modify our owner-details.component.html file:

<app-owner-data [owner]='owner' (selectEmitt)='showAccounts = $event'></app-owner-data>

<app-account-data \*ngIf='showAccounts' [accounts]='owner?.accounts'></app-account-data>

Now let’s check the result:



## Conclusion

What we have learned….

In the next part…

[subscription]

# Angular Material Inputs, DatePicker, Form Validation, Modals – Create Owner Component

One of the most important components in Angular Material is the input component. Angular Material supports different types of input elements like color, date, email, month, number, password etc. In this article, we are going to use the input components to create a Create-Owner component thus creating a new Owner object in our database.

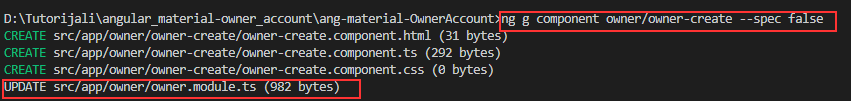
Of course, we will show how easy it is to apply form validation with the material input components and also how to create dialogs to show the error or success messages. We won’t dive deeply into the validation logic because we already have a great article that covers that topic [.NET Core 2.0, Angular and MySQL Form Validation.](https://code-maze.com/net-core-web-development-part13/)

Let’s start step by step.

## Create Owner Component and Routing Configuration

The first thing we need to do is to create our component:

ng g component owner/owner-create --spec false



Once we have our component in the place, let’s configure the routing part in the owner-routing.module.ts file:

import { OwnerCreateComponent } from '../owner-create/owner-create.component';

const routes: Routes = [

{ path: 'owners', component: OwnerListComponent },

{ path: 'details/:id', component: OwnerDetailsComponent},

{ path: 'create', component: OwnerCreateComponent}

];

Finally, let’s modify the owner-list.component.html file, to add the link that will point to the owner-create component:

<div fxLayout fxLayout.lt-md="column wrap" fxLayoutAlign="center center" fxLayoutGap.gt-sm="250px" fxLayoutGap.lt.md="20px">

<mat-form-field>

<input matInput type="text" (keyup)="doFilter($event.target.value)" placeholder="Filter">

</mat-form-field>

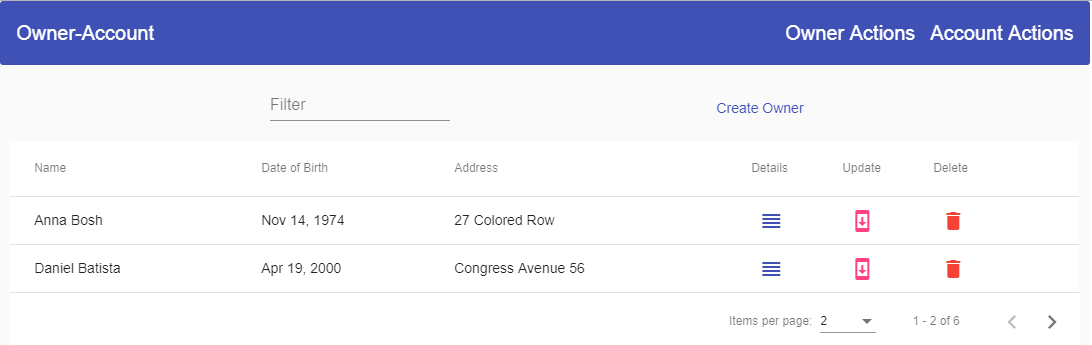
<div>

<a [routerLink]="['/owner/create']" mat-button color="primary">Create Owner</a>

</div>

</div>

Now we can inspect our result:



And if we click on the Create Owner button, we are going to be directed to the owner-create component for sure.

## Adding Material Inputs and Form Validation

Before we start adding input fields, we need to import one more module into the owner.module.ts file:

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

imports: [

…,

ReactiveFormsModule

],

We need this one for the reactive form validation. Moreover, we are going to use the datepicker material component and for that, we need the MatDatepickerModule and MatNativeDataModule inside the material.module.ts file:

Import{ …, MatDatepickerModule, MatNativeDateModule } from '@angular/material';

imports: [

MatDatepickerModule,

MatNativeDateModule,

exports: [

MatDatepickerModule,

MatNativeDateModule,

Now, we can start with the owner-create.compnent.html file modification:

<section fxLayout="row wrap" fxLayoutAlign="center center">

<mat-card fxFlex="500px" fxFlex.xs="100%">

<mat-card-title>Create a new owner</mat-card-title>

<mat-card-content>

</mat-card-content>

<mat-card-actions>

</mat-card-actions>

</mat-card>

</section>

The mat-cart-content and mat-card-actions elements, needs to be wrapped with the form tag:

<form [formGroup]="ownerForm" autocomplete="off" novalidate (ngSubmit)="createOwner(ownerForm.value)" fxLayout="column wrap"

fxLayoutAlign="center center" fxLayoutGap="10px">

<mat-card-content>

</mat-card-content>

<mat-card-actions>

</mat-card-actions>

</form>

Then inside the mat-card-content, we are going to add the following code:

<mat-form-field>

<input matInput type="text" placeholder="Owner's name" formControlName="name" id="name">

<mat-hint align="end">Not more then 60 characters long.</mat-hint>

<mat-error \*ngIf="hasError('name', 'required')">Name is required</mat-error>

<mat-error \*ngIf="hasError('name', 'maxlength')">You have more than 60 characters</mat-error>

</mat-form-field>

<mat-form-field>

<input matInput [matDatepicker]="picker" placeholder="Choose a date of birth" formControlName="dateOfBirth" id="dateOfBirth"

readonly (click)="picker.open()">

<mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>

<mat-datepicker #picker></mat-datepicker>

</mat-form-field>

<mat-form-field>

<input matInput type="text" placeholder="Owner's address" formControlName="address">

<mat-hint align="end">Not more then 100 characters long.</mat-hint>

<mat-error \*ngIf="hasError('address', 'required')">Address is required</mat-error>

<mat-error \*ngIf="hasError('address', 'maxlength')">You have more than 100 characters</mat-error>

</mat-form-field>

Finally, let’s modify the mat-card-actions element:

<mat-card-actions align="center">

<button mat-raised-button color="primary" [disabled]="!ownerForm.valid">Create</button>

<button type="button"> mat-raised-button color="warn" (click)="onCancel()">Cancel</button>

</mat-card-actions>

Now, we have completed the html part, and we are ready to modify the owner-create.component.ts file. But before we do that, we are going to create a new interface OwnerForCreation:

export interface OwnerForCreation {

name: string;

dateOfBirth: Date;

address: string;

}

And right after that, we are going to modify our .ts file:

import { RepositoryService } from './../../shared/repository.service';

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

import { FormControl, FormGroup, Validators } from '@angular/forms';

import { Location } from '@angular/common';

import { OwnerForCreation } from '../../\_interface/ownerForCreation.model';

@Component({

selector: 'app-owner-create',

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

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

})

export class OwnerCreateComponent implements OnInit {

public ownerForm: FormGroup;

constructor(private location: Location, private repository: RepositoryService) { }

ngOnInit() {

this.ownerForm = new FormGroup({

name: new FormControl('', [Validators.required, Validators.maxLength(60)]),

dateOfBirth: new FormControl(new Date()),

address: new FormControl('', [Validators.required, Validators.maxLength(100)])

});

}

public hasError = (controlName: string, errorName: string) =>{

return this.ownerForm.controls[controlName].hasError(errorName);

}

public onCancel = () => {

this.location.back();

}

public createOwner = (ownerFormValue) => {

if (this.ownerForm.valid) {

this.executeOwnerCreation(ownerFormValue);

}

}

private executeOwnerCreation = (ownerFormValue) => {

let owner: OwnerForCreation = {

name: ownerFormValue.name,

dateOfBirth: ownerFormValue.dateOfBirth,

address: ownerFormValue.address

}

let apiUrl = 'api/owner';

this.repository.create(apiUrl, owner)

.subscribe(res => {

//this is temporary, until we create our dialogs

this.location.back();

},

(error => {

//temporary as well

this.location.back();

})

)

}

}

The last thing we need to do is to modify the .css file:

mat-form-field{

width: 400px;

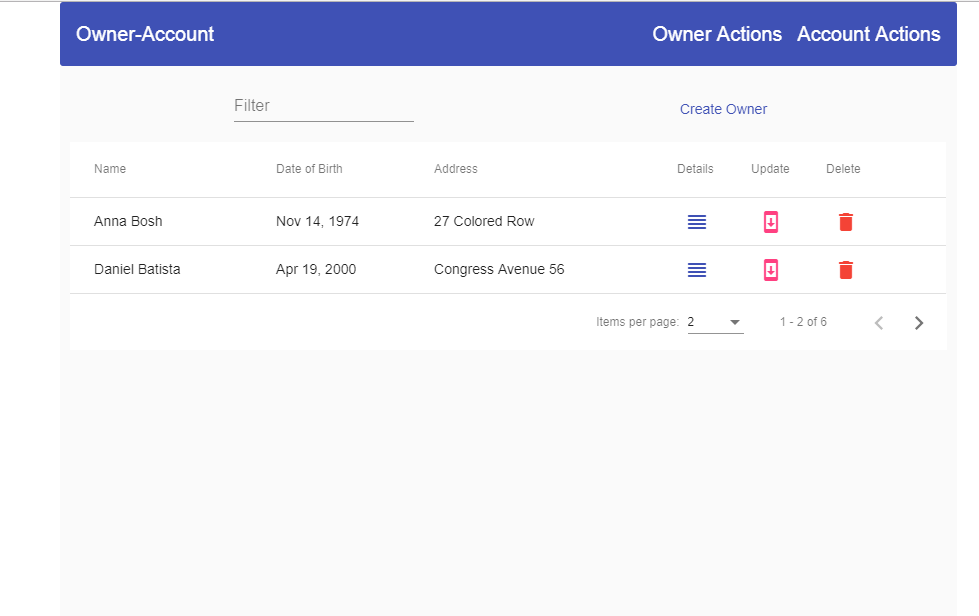
}

mat-card-title{

text-align: center;

}

Excellent. We can check our result:



## Adding Dialogs and Shared Module

We have finished with the owner creation, but, we need to inform a user about the creation result, whether it was successful or not. For that purpose, we are going to create two dialog components. One for the success message and another one for the error message.

### Shared Module

Before we do that, we are going to create a shared module to register our dialog components and to register modules that are already registered inside the app module and owner module as well.

So, let’s first create a shared module:

ng g module shared --spec false

Now, let’s modify the shared.module.ts file:

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

import { CommonModule } from '@angular/common';

import { MaterialModule } from '../material/material.module';

import { FlexLayoutModule } from '@angular/flex-layout';

@NgModule({

imports: [

CommonModule,

MaterialModule,

FlexLayoutModule,

],

exports: [

MaterialModule,

FlexLayoutModule

],

declarations: []

})

export class SharedModule { }

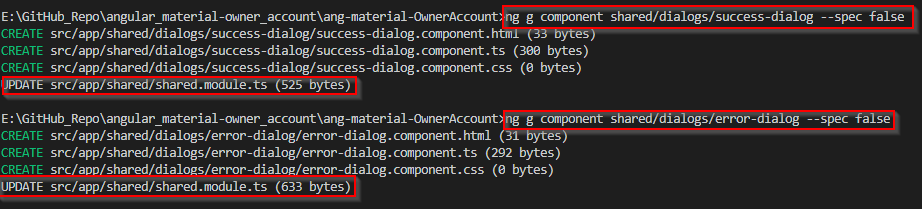
**Important:** *Because we have the FlexLayoutModule and MaterialModule now inside of this module file, we don’t need them anymore in the app and owner module files. Therefore, we can remove the FlexLayout and Material modules from the app and owner module files and just import the SharedModule in both files.*

### Dialogs Creation

Now we can continue, with the dialog creation:

ng g component shared/dialogs/success-dialog --spec false

ng g component shared/dialogs/error-dialog --spec false



These modules are imported automatically in the shared.module.ts file, but we need to export them as well. Moreover, we must place the dialog components inside the entryComponents array because we are not going to use routing nor app selector to call these components. We are going to use them as a template reference for the dialog’s open() function, thus the need for the entryComponents array.

exports: [

MaterialModule,

FlexLayoutModule,

SuccessDialogComponent,

ErrorDialogComponent

],

entryComponents: [

SuccessDialogComponent,

ErrorDialogComponent

]

### Success Dialog Modification

Let’s open the success-dialog.component.html file and modify it:

<section fxLayout="column" fxLayoutAlign="center center">

<h1 mat-dialog-title>Success message</h1>

<mat-dialog-content>

<p>Action completed successfully</p>

</mat-dialog-content>

<mat-dialog-actions>

<button mat-raised-button color="primary" [mat-dialog-close]="true">OK</button>

</mat-dialog-actions>

</section>

One important thing to notice here is a usage of the mat-dialog-close attribute which makes this button to close the dialog and submits a result (true in this case). To fetch this result, we will have to subscribe to the afterClosed() function, and we are going to do that a bit later.

We haven’t used the mat-dialog elements in our project, therefore we need to register it in the material module:

import { ..., MatDialogModule} from '@angular/material';

imports: [

MatDialogModule,

exports: [

MatDialogModule,

Now, to use this success dialog, we are going to modify the owner-create.component.ts file. Our dialog needs to have a configuration, and we are going to provide that:

import { MatDialog } from '@angular/material';

...

private dialogConfig;

constructor(private location: Location, private repository: RepositoryService, private dialog: MatDialog) { }

ngOnInit() {

this.ownerForm = new FormGroup({

name: new FormControl('', [Validators.required, Validators.maxLength(60)]),

dateOfBirth: new FormControl(new Date()),

address: new FormControl('', [Validators.required, Validators.maxLength(100)])

});

this.dialogConfig = {

height: '200px',

width: '400px',

disableClose: true,

data: { }

}

}

To start our success dialog, we need to import the MatDialog and to create a private variable in a constructor of the same type.

The dialogConfig object consists of several properties which describe the height, width, disable close dialog when clicking outside of the dialog and passing data to the dialog. Because we don’t want to pass anything to the success dialog, we have left an empty object.

Now, let’s modify the subscribe part of the component, to call this dialog:

let apiUrl = 'api/owner';

this.repository.create(apiUrl, owner)

.subscribe(res => {

let dialogRef = this.dialog.open(SuccessDialogComponent, this.dialogConfig);

//we are subscribing on the [mat-dialog-close] attribute as soon as we click on the dialog button

dialogRef.afterClosed()

.subscribe(result => {

this.location.back();

});

},

(error => {

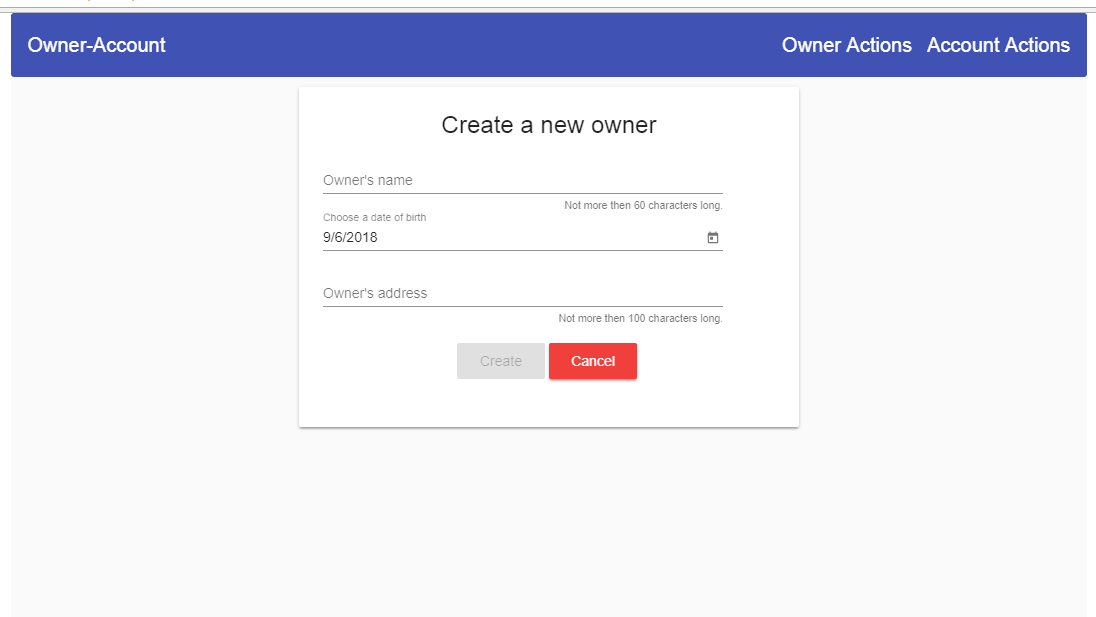
//temporary as well

this.location.back();

})

)

This is the result:



### Error Dialog Modifications

We are going to send an error message to the error dialog and we don’t want to emit any event when we click the dialog button, therefore the Error dialog implementation will be a little different.

Let’s start by modifying the error-dialog.component.ts file:

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

import { MAT\_DIALOG\_DATA, MatDialogRef } from '@angular/material';

@Component({

selector: 'app-error-dialog',

templateUrl: './error-dialog.component.html',

styleUrls: ['./error-dialog.component.css']

})

export class ErrorDialogComponent implements OnInit {

constructor(public dialogRef: MatDialogRef<ErrorDialogComponent>, @Inject(MAT\_DIALOG\_DATA) public data: any) { }

ngOnInit() {

}

public closeDialog = () => {

this.dialogRef.close();

}

}

The dialogRef variable is here to help us manipulate with our opened dialog and the data variable is here to accept any information passed to this component. Of course, we must use the MAT\_DIALOG\_DATA injection token to enable accepting data from other components.

Now we have to modify the error-dialog.component.html file:

<section fxLayout="column" fxLayoutAlign="center center">

<h1 mat-dialog-title>Error message</h1>

<mat-dialog-content>

<p>{{data.errorMessage}}</p>

</mat-dialog-content>

<mat-dialog-actions>

<button mat-raised-button color="warn" (click)="closeDialog()">OK</button>

</mat-dialog-actions>

</section>

After that, let’s modify the error-handler.service file:

... Other imports

import { ErrorDialogComponent } from './dialogs/error-dialog/error-dialog.component';

@Injectable({

providedIn: 'root'

})

export class ErrorHandlerService {

public errorMessage: string = '';

public dialogConfig;

constructor(private router: Router, private dialog: MatDialog) { }

... Other code

private handleOtherError(error: HttpErrorResponse){

this.createErrorMessage(error);

this.dialogConfig.data = { 'errorMessage': this.errorMessage };

this.dialog.open(ErrorDialogComponent, this.dialogConfig);

}

And finally, let’s modify the owner-create.component.ts file:

import { ErrorHandlerService } from '../../shared/error-handler.service';

constructor(private location: Location, private repository: RepositoryService, private dialog: MatDialog, private errorService: ErrorHandlerService) { }

this.repository.create(apiUrl, owner)

.subscribe(res => {

let dialogRef = this.dialog.open(SuccessDialogComponent, this.dialogConfig);

//we are subscribing on the [mat-dialog-close] attribute as soon as we click on the dialog button

dialogRef.afterClosed()

.subscribe(result => {

this.location.back();

});

},

(error => {

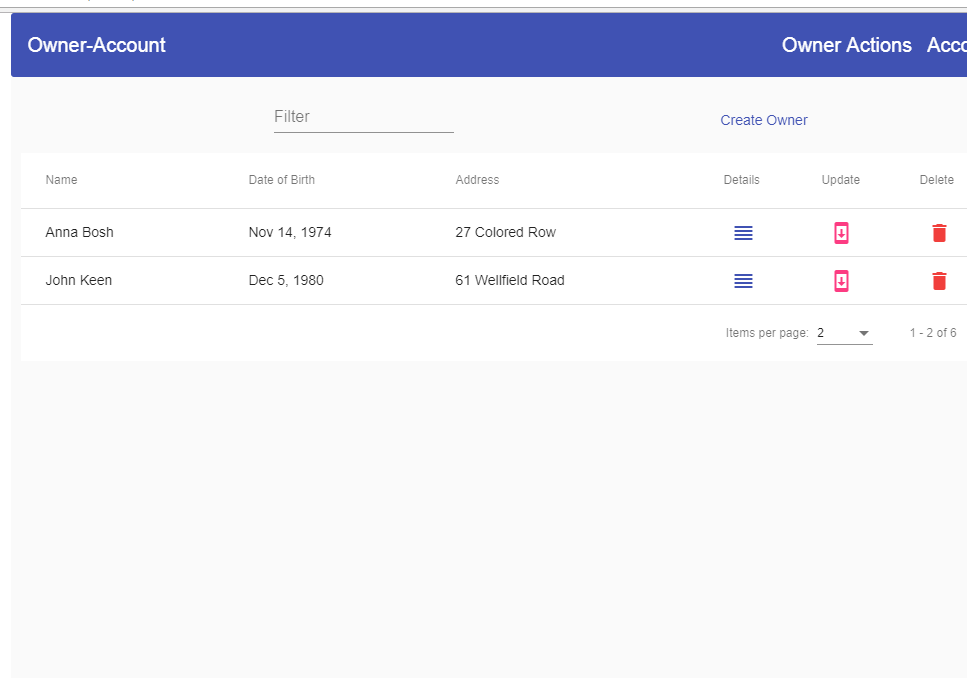
this.errorService.dialogConfig = { ...this.dialogConfig };

this.errorService.handleError(error);

})

)

And that is all. Now, we can check the result:



## Conclusion

# Excellent.

# We have created a lot of components and covered a lot of material features as well. We have left the update and delete functionalities, but this is something that you could do on your own now. In all the previous articles, we have covered all the material related logic we need for the Update and Delete actions (Forms, Validations, Input Fields, Dialogs etc) and in these articles [.NET Core 2.0, Angular and MySQL. PUT Actions](https://code-maze.com/net-core-web-development-part14/) and [.NET Core 2.0, Angular and MySQL. Delete Actions](https://code-maze.com/net-core-web-development-part15/) we have all required Angular related logic.

So, feel free to read them, if you haven’t already, and make your own Update and Delete parts of this project.

If you want to download the complete source code with the Update and Delete features included, you can do that here…(link).