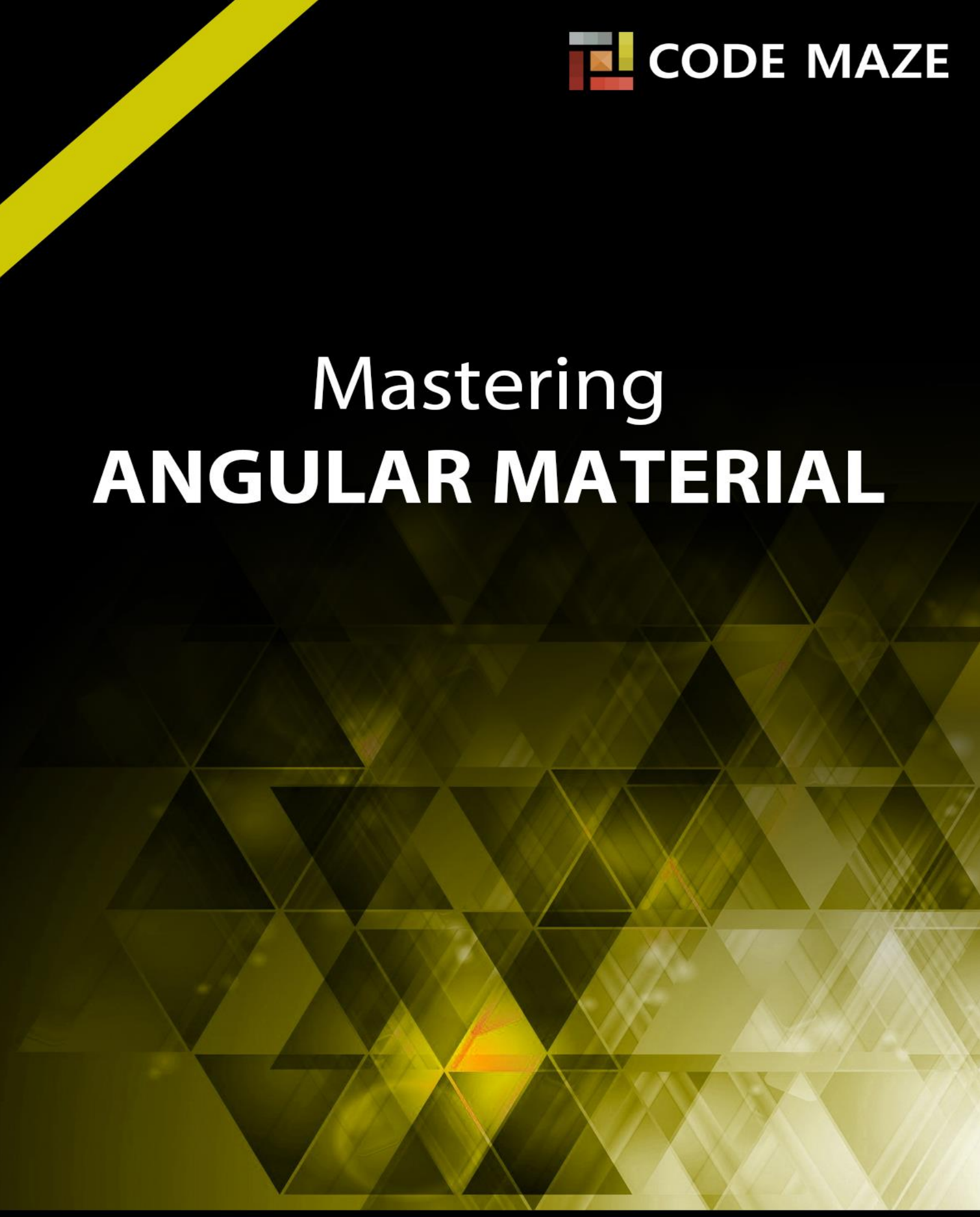
CODE MAZE

Mastering
ANGULAR MATERIAL



https://code-maze.com

CONTENTS

GETTING STARTED WITH ANGULAR MATERIAL 4

- Project Creation 4
- Angular Material Installation 4
- Creating Material Module 6
- Angular Flex Layout And The Layout Component 7
- Material Tab 9
- Additional Mat-Tab Features 12

ANGULAR MATERIAL RESPONSIVE NAVIGATION 13

- Creating Routes 13
- Mat-Sidenav-Container 14
- Additional Mat-Sidenav and Container Features 16
- Mat-Toolbar 16
- Mat-Nav-List 21
- Mat-Nav-List Additional Feature 23
- Mat-Menu to Create Multi-Menu in Side-Nav 24

ANGULAR MATERIAL TABLE, FILTER, SORTING, PAGING 25

- Environment, HTTP and Owner Module 25
- Creating a New Module File 27

- Using Material Table to Display Data 29
- Mat-Sort Component..... 33
- Additional Mat-Sort-Header Features 35
- Filtering Data in Material Table..... 36
- Mat-Paginator Component..... 37
- Mat-Paginator Additional Features 39

ADDITIONAL PROJECT FEATURES WITH MATERIAL COMPONENTS . 41

- Mat-Progress-Bar Component..... 41
- Mat-Checkbox and Mat-Progress-Spinner..... 43
- Mat-Checkbox Additional Features 47
- Error Handling Service..... 48
- Card, Select and Expansion Panel Components..... 50
- Using Material Card and Select Components..... 52
- Using Mat-Expansion-Panel 55
- Mat-Expansion-Panel Additional Features 57

INPUT, DATEPICKER AND MODAL COMPONENTS IN MATERIAL FORMS 59

- Routing Configuration And Component Creation..... 59
- Using Material Input Component for the Form Validation 60
- Modal Dialog Component and Shared Module 64

- Success Dialog Modification..... 66
- Error Dialog Modifications 69

FINAL WORDS 72

GETTING STARTED WITH ANGULAR MATERIAL

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

☙ 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 a 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 the required packages. Besides installing Angular Material, we need to install CDK and Animations as well.

So, let's do that first by navigating to the root folder of our project and running the command:

```
npm install --save @angular/material @angular/cdk @angular/animations
```

After installation finishes, we should see a similar result to this one:

```
+ @angular/animations@6.1.4
+ @angular/cdk@6.4.6
+ @angular/material@6.4.6
added 3 packages and updated 1 package in 12.06s
```

Now, we need to configure animations, by importing the **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 gesture support. In order to have a full feature of some components, we need to install it:

```
npm install --save hammerjs
```

After the 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 module into the **app.module.ts** file. That being said, let's do it:

```
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';

//the rest of the code

imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MaterialModule
],
```

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

Angular Flex Layout And The Layout Component

To start, let's create the Layout component which is going to be an entry point for our entire application. Afterward, we are going to import its selector inside the **app.component.ts** file:

```
ng g component layout --spec false
```

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

Before we modify the HTML component file, we need to install one more library: **@angular/flex-layout**. This library will help us create a responsive application. 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';

//the rest of the code

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 100% of its

parent. If you want to read more about flex-layout, you can find plenty of material here: Flex-Layout-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 actually works.

Now, when all is prepared, we can start using our first Angular Material component.

☙ Material Tab

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 application 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 as well:

```
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 application.
                Basically, 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 a fully responsive navigation menu, with
                it's side-bar as well.
            </p>
        </mat-tab>
    </mat-tab-group>
</section>
```

Now, we can inspect our result:

☙ Additional Mat-Tab Features

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

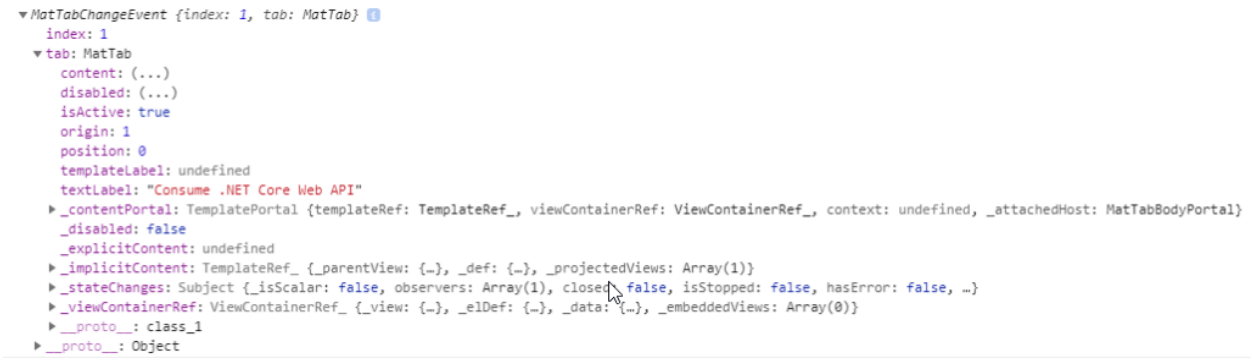
So, let's use the **selectedTabChange** event:

```
<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:



ANGULAR MATERIAL RESPONSIVE NAVIGATION

Every application needs to have some sort of navigation, to provide users with a better experience.

So, in this chapter, we are going to learn how to create a complete responsive navigation menu by using Angular Material Components.

Therefore, let's start with the routes.

☙ Creating Routes

Creating a new module for the routes is always a good practice, so let's do exactly that:

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

A next step is to 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 the 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.

Mat-Sidenav-Container

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, don't we? So, let's start with the `app.component.html` file 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>
```

We create a container for a side navigation bar and specify the part for our content. As you can see the `<mat-sidenav>` element defines a place for a side navigation and the `<mat-sidenav-content>` element defines a place

for our content. We need to use the local reference **#sidenav**, and a little bit later, you will see why.

Of course, this won't work. We need to register the module 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 a 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.

Additional Mat-Sidenav and Container Features

By default, the **mat-sidenav** component will auto focu the first focusable element inside the navigation, but if want to disable that, we can add the **autoFocus** attribute:

```
<mat-sidenav #sidenav role="navigation" [autoFocus]='false'>
    <app-sidenav-list (sidenavClose)="sidenav.close()"></app-sidenav-list>
</mat-sidenav>
```

Furthermore, by default if we open our side-bar, we can close it by simply clicking on the backdrop part. If we don't want this, we can change our code:

```
<mat-sidenav #sidenav role="navigation" [autoFocus]='false' [disableClose]="true">
    <app-sidenav-list (sidenavClose)="sidenav.close()"></app-sidenav-list>
</mat-sidenav>
```

By adding the **disableClose** attribute and setting it to **true**, we are disabling the close functionality once the backdrop is clicked.

Finally, we can disable the backdrop on the mat-sidenav-container component by setting the **hasBackdrop** attribute to false:

```
<mat-sidenav-container [hasBackdrop]='false'>
```

Mat-Toolbar

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), and the Owner-Account part that navigates to the

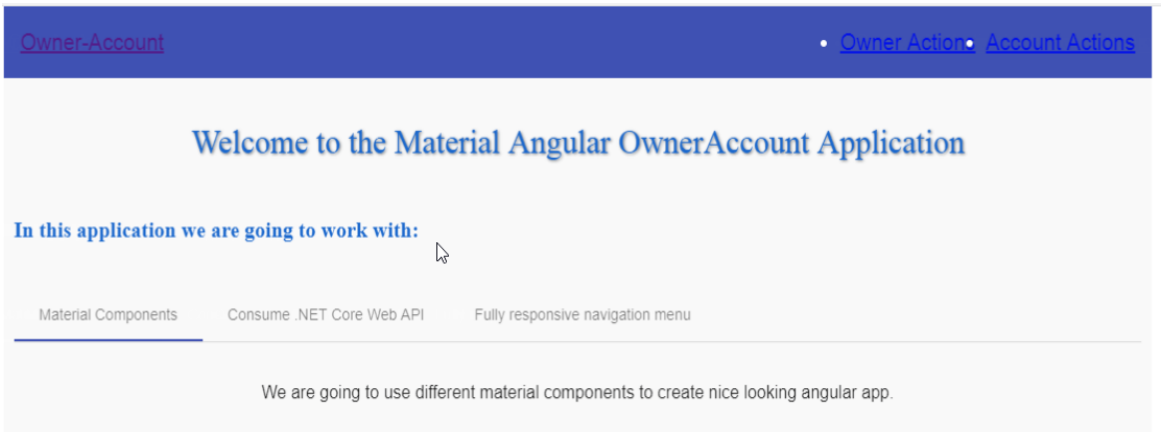
home component. As you can see, we use the `fxHide.gt-xs` directive, which states that this part should be hidden only on the screen that is greater than extra small.

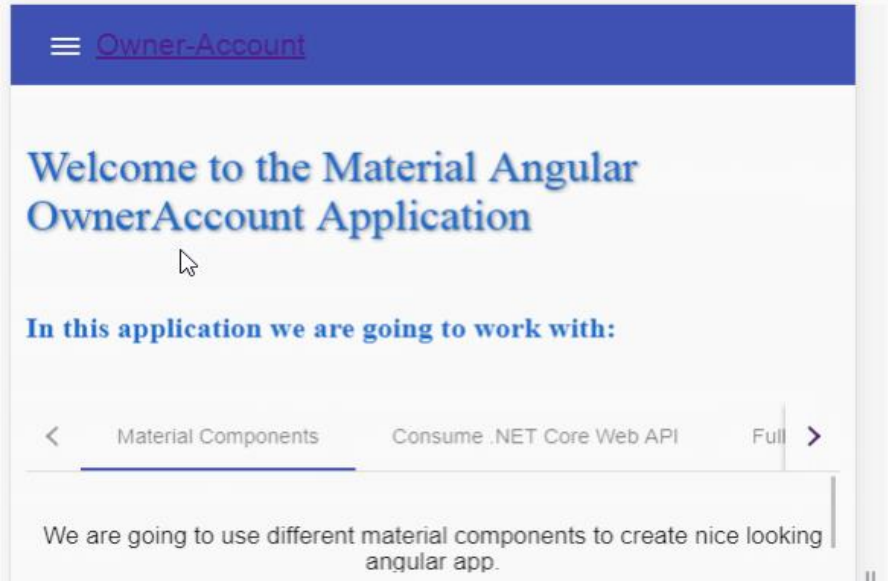
We have another part of navigation which is positioned on the end of the navbar and hidden only for the extra small screen.

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, our menu looks like this:





Let's improve the look of our navigation menu by modifying 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 can have another look at our menu. It looks much nicer, isn't it?

If we take a look at the icon button code, we can see the **onToggleSidenav()** event. We need to implement it inside the **header.component.ts** file:

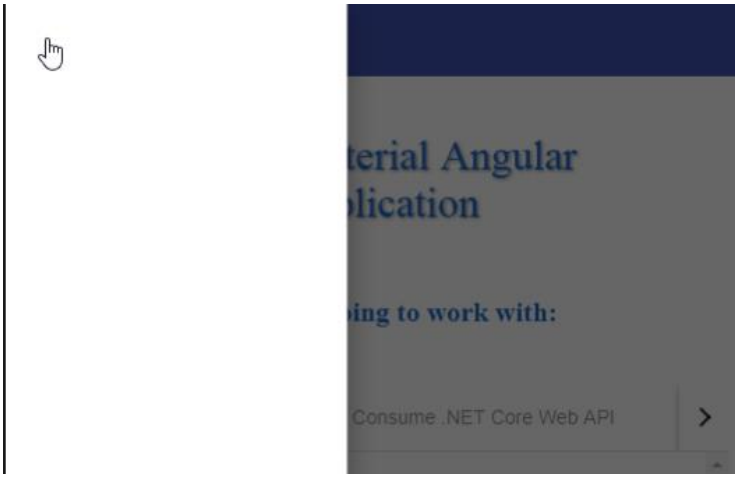
```
import { Component, OnInit, Output, EventEmitter } from '@angular/core';

@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();
    }
}
```

Finally, we have to react to 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 it is obvious why we need the **#sidenav** local reference inside the **mat-sidenav** component.

As soon as we click on the Owner-Account button, we should have the side nav shown:



Excellent. The time has come to implement the side navigation.

Mat-Nav-List

To create side navigation items, we are going to use the `mat-nav-list` element that resides inside `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:

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

and modify the `sidenav-list.component.html` file:

```
<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 you can see, we use the **mat-nav-list** as a container with all the anchor tags containing the **mat-list-item** attributes. The click event is there for every link, to close the side-nav as soon as a user clicks on it. Finally, every link contains its own mat-icon.

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:

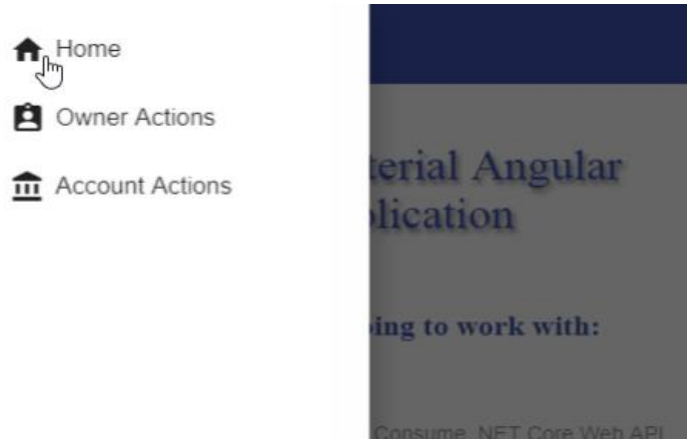
```
import { Component, OnInit, Output, EventEmitter } from '@angular/core';

@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>
```

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



☙ Mat-Nav-List Additional Feature

By default, once we click the list item, the gray ripple will stretch over the list as some kind of animation. If we want to disable that for some reason we can add the `disableRipple` attribute to the `mat-nav-list` component:

```
<mat-nav-list [disableRipple]='true'>
```

Now, we won't see that animation anymore.

☙ Mat-Menu to Create 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 our 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.

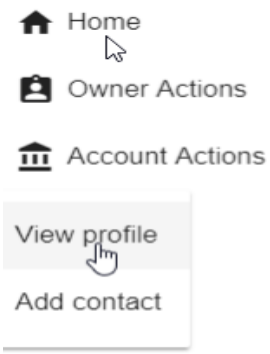
So, in the **sidenav-list.component.html** file, we need to add the following code below the last anchor tag:

```
<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 **MatMenuModule**:

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

imports: [
    MatMenuModule,

exports: [
        MatMenuModule,
```

As a result, we should have a multi-menu option in our side navigation bar:



ANGULAR MATERIAL TABLE, FILTER, SORTING, PAGING

We are going to divide this chapter into two major parts. **First part** will consist of creating environment files, HTTP repository service and creating a new Owner module with the lazy loading feature.

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 table.

The source code, with the additional server-side project, is available at GitHub Angular Material Table – Source Code.

So, 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'
};
```

Having these environment files modified, it is time 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 service file. If you want to learn more about environment files, services, and HTTP, you can read that in the Angular Series Article 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 Module File

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 the lazy loading feature:

```
const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'owner', loadChildren: "./../owner/owner.module#OwnerModule" },
    { path: '', redirectTo: '/home', pathMatch: 'full' }
];
```

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 modify 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 { }
```

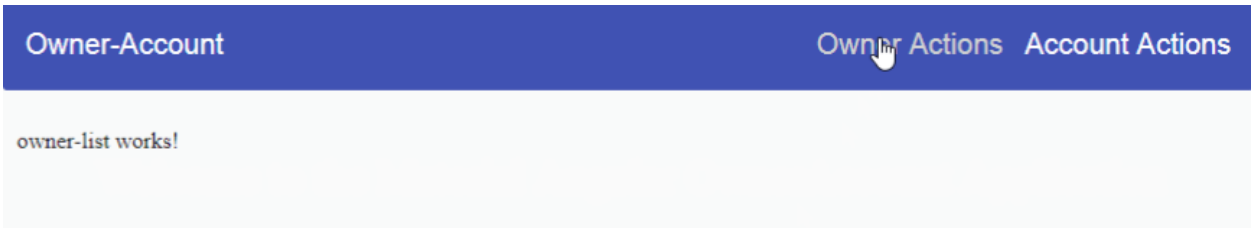
Finally, to make all this to work, we need to modify our routes in the **sidenav-list.component.html** file:

```
<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 confirm now that our routing settings work as it supposed to:



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

☙ Using Material Table to Display 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
],
```

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

For now, 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 into a material one. With the **dataSource** attribute, we provide a data source for our table. 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` properties 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:

Owner-Account			Owner Actions		Account Actions
Name	Date of Birth	Address	Details	Update	Delete
Anna Bosh	Nov 14, 1974	27 Colored Row			
Daniel Batista	Apr 19, 2000	Congress Avenue 56			
Dave	Mar 29, 2009	Dave's street 23			
John Keen	Dec 30, 1980	61 Wellfield Road			
Martin Miller	May 21, 1983	3 Edgar Buildings			
Sam Query	Apr 22, 1990	91 Western Roads			

Mat-Sort Component

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.

Modifying the `table` tag is going to be our first task:

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

Then, we are going to add the `mat-sort-header` directive to the `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>
```

To make sorting functionality up and running, we need to modify the `owner-list.component.ts` file as well:

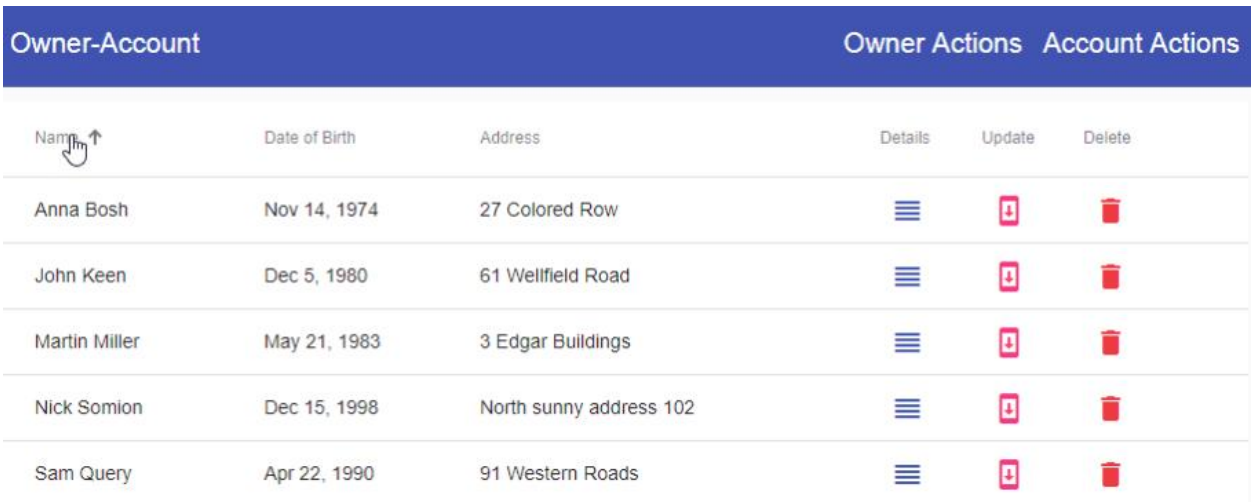
```
export class OwnerListComponent implements OnInit, AfterViewInit {

    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 of 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)">
```

Once we click on the name column it will generate the following JSON object:

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

1. active: "name"
2. direction: "asc"
3. __proto__: Object
```

☙ Additional Mat-Sort-Header Features

By default, the position of an arrow is placed behind the column name, but we can change that by using the **arrowPosition** attribute:

```
<th mat-header-cell *matHeaderCellDef mat-sort-header arrowPosition='before'> Name </th>
```

Pay attention that we will have to add additional styles to our column because it will shift to the right a bit. The default value for this attribute is **after**.

To remove sorting feature out of the column, we can either remove the mat-sort-header directive or add the disabled attribute:

```
<th mat-header-cell *matHeaderCellDef mat-sort-header disabled='true'> Name </th>
```

We can override the initial sorting value for the specified column by using the **start** attribute:

```
<th mat-header-cell *matHeaderCellDef mat-sort-header start='desc'> Name </th>
```

Another value for the start attribute is **asc**.

☙ Filtering Data 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 the following code right above our table in the 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 write the following 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 a new Owner module, we need to import **FlexLayoutModule** into the Owner module file as well:

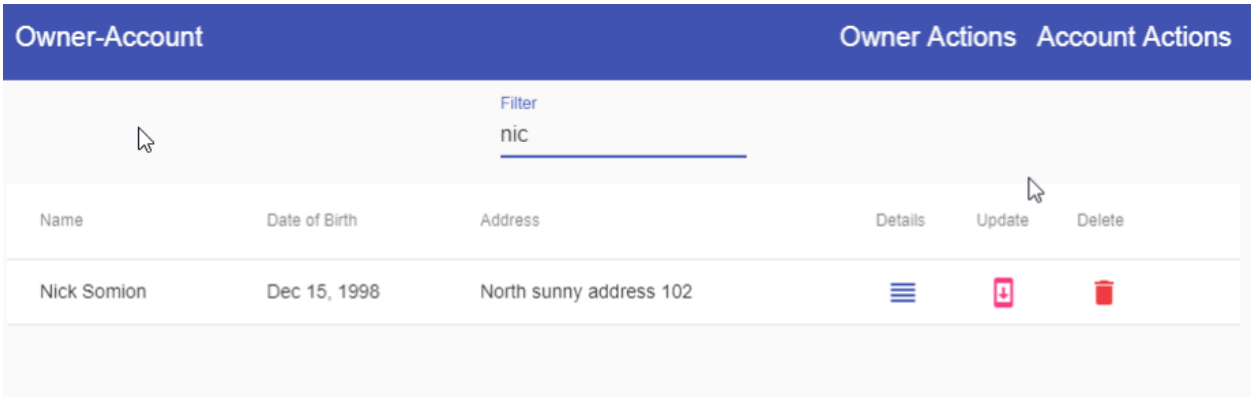
```
import { FlexLayoutModule } from '@angular/flex-layout';

imports: [
```

```
    ...
    FlexLayoutModule
],
```

Of course, this code repetition will be solved as well as soon as we create a `Shared` module.

Excellent.

Now we can inspect the result:



Mat-Paginator Component

To implement paging with a material table, 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,
```

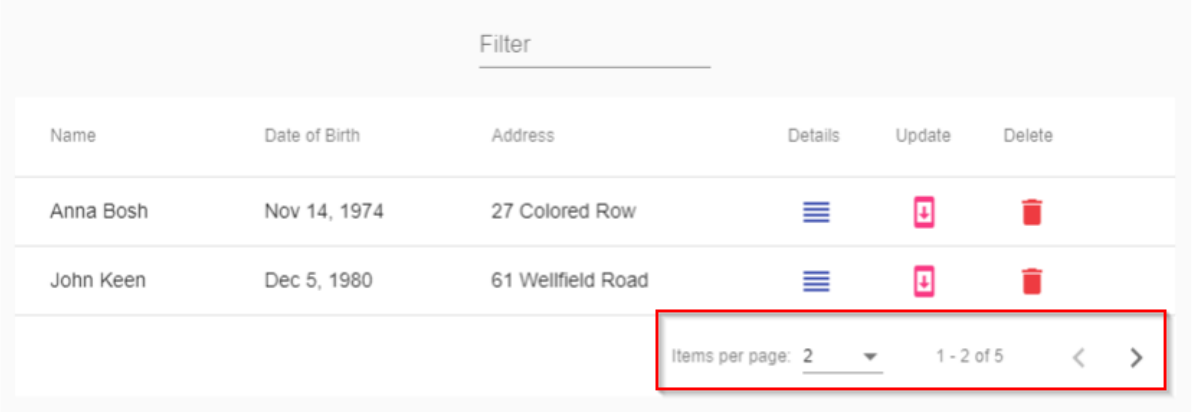
Then, let's add `mat-paginator` inside the HTML file:

```
<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;
}
  ...
```

After these changes, we should have the following 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>
```

Mat-Paginator Additional Features

By default, paginator shows us only the next and the previous buttons for the page navigation. If we want to add the buttons for the first and the last page, we can do that by introducing the **showFirstLastButtons** attribute:

```
<mat-paginator [pageSize]="2" [pageSizeOptions]="[2, 4, 6, 10, 20]"
[showFirstLastButtons]='true'>
</mat-paginator>
```

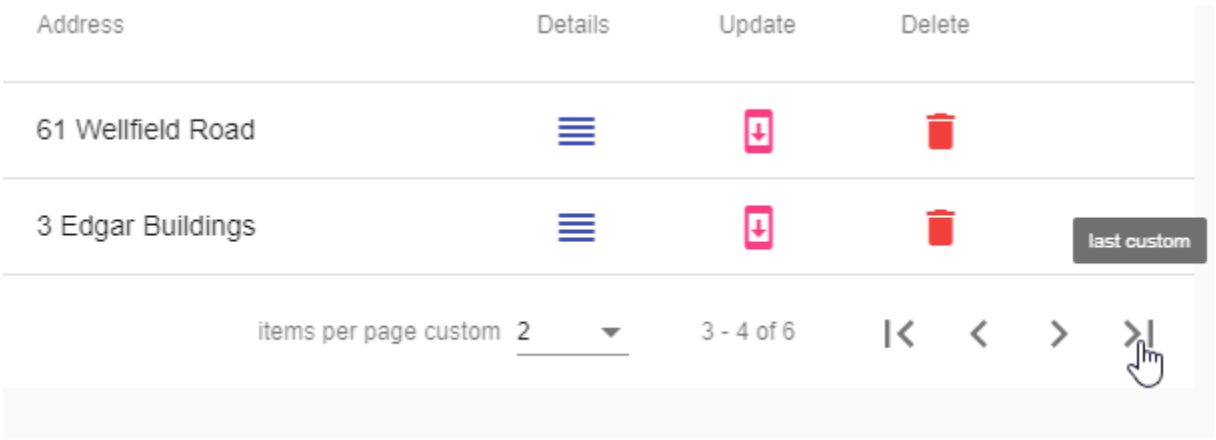
Now, we can change the labels for our buttons from the next page, the last page… to the custom values. To do that we need to introduce a custom class in our project:

```
import { MatPaginatorIntl } from '@angular/material';
export class CustomPaginatorIntl extends MatPaginatorIntl {
    itemsPerPageLabel = 'items per page custom';
    nextPageLabel = 'next custom';
    previousPageLabel = 'previous custom';
    firstPageLabel = 'first custom';
    lastPageLabel = 'last custom';
}
```

The next thing to do is to provide this class in the **AppModule.ts** file:

```
providers: [{ provide: MatPaginatorIntl, useClass: CustomPaginatorIntl }],
```

And our result should be:



Additional Project Features with Material Components

In this chapter, we are going to create three different pages: NotFound, ServerError and the OwnerDetails page. Through these Angular components (pages) we are going to learn how to use different Angular Material components like Progress Bar, Spinner, CheckBox, Card, Select and Expansion Panel.

So, let's start.

Mat-Progress-Bar Component

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 non-existing address, we will get the `NotFound` component instead, with the "not-found works".

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>
```

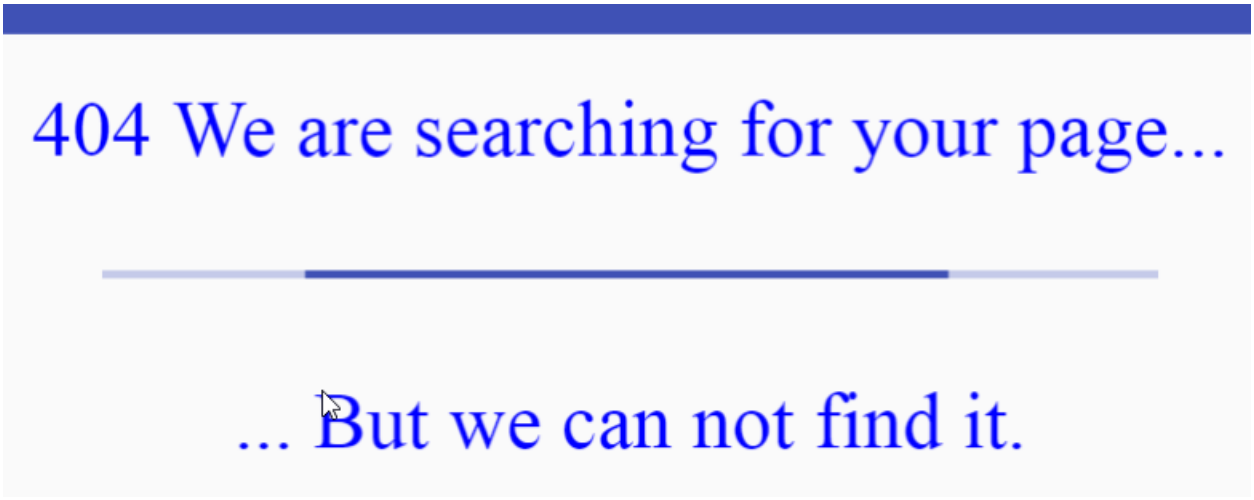
As we can see, we are using the **mat-progress-bar** material component, and for that reason, we need to import the required module into the **material.module.ts** file:

```
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%;
}
```

That is it. We can inspect our result by typing a none-existing URL (*localhost:4200/something*):



This looks good.

Let's continue with the Server-Error component.

Mat-Checkbox and Mat-Progress-Spinner

We are going to start with Server-Error component creation:

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

Having that done, 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' }
];
```

For the visual experience, 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,
    MatCheckboxModule,

exports: [
        MatProgressSpinnerModule,
        MatCheckboxModule,
```

Ok, we have imported all the necessary modules and now we are going to 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() { }
```

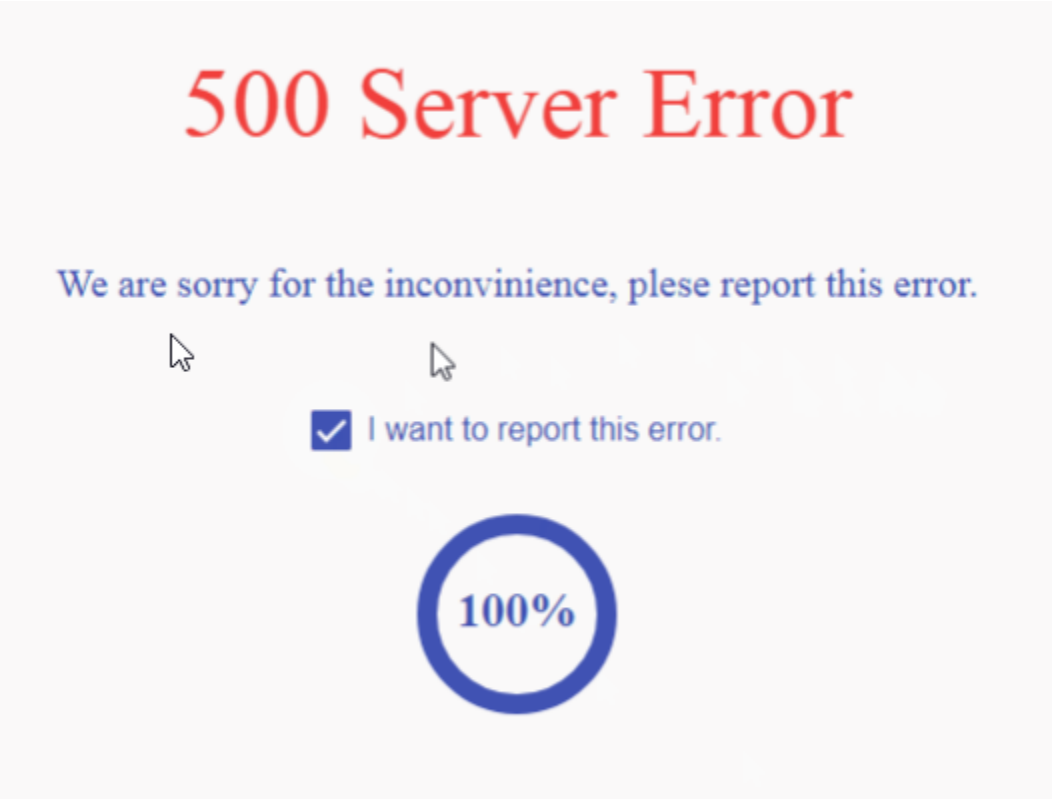
```
    ngOnInit() {
    }

    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;
}
```

Our result should look like this:

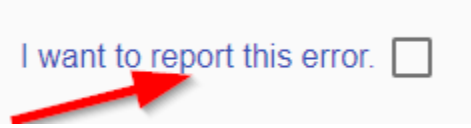

500 Server Error
We are sorry for the inconvinience, plese report this error.
I want to report this error.
42%



☙ Mat-Checkbox Additional Features

We can modify our check-box in a few different ways. For example, if we want to switch the label position, we can use the **labelPosition** attribute:

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



The default value for this attribute is **after**.

We can disable this control with the **disabled** attribute:

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

If we don't want a ripple to be present on this control, we can remove that as well:

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

By default, the check-box is unchecked, but we can change that:

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

☙ 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:

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

and modify the component file:

```
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.

☙ Card, Select and Expansion Panel Components

In this section, we are going to create 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 the **owner-routing.module.ts** file:

```
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';

//other code

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

//other code

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. The next thing we are going to do is to add an additional interface:

```
export interface Account {
    id: string;
    dateCreated: Date;
    accountType: string;
    ownerId?: string;
}
```

and change our existing one:

```
import { Account } from './account.model';
export interface Owner {
    id: string;
    name: string;
    dateOfBirth: Date;
    address: string;

    accounts?: Account
}
```

After all of 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 prepared the logic to fetch the data from the server, so the obvious continuation is 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
```

☙ Using Material Card and Select Components

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,
```

Next thing we need to do is to 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 {
```

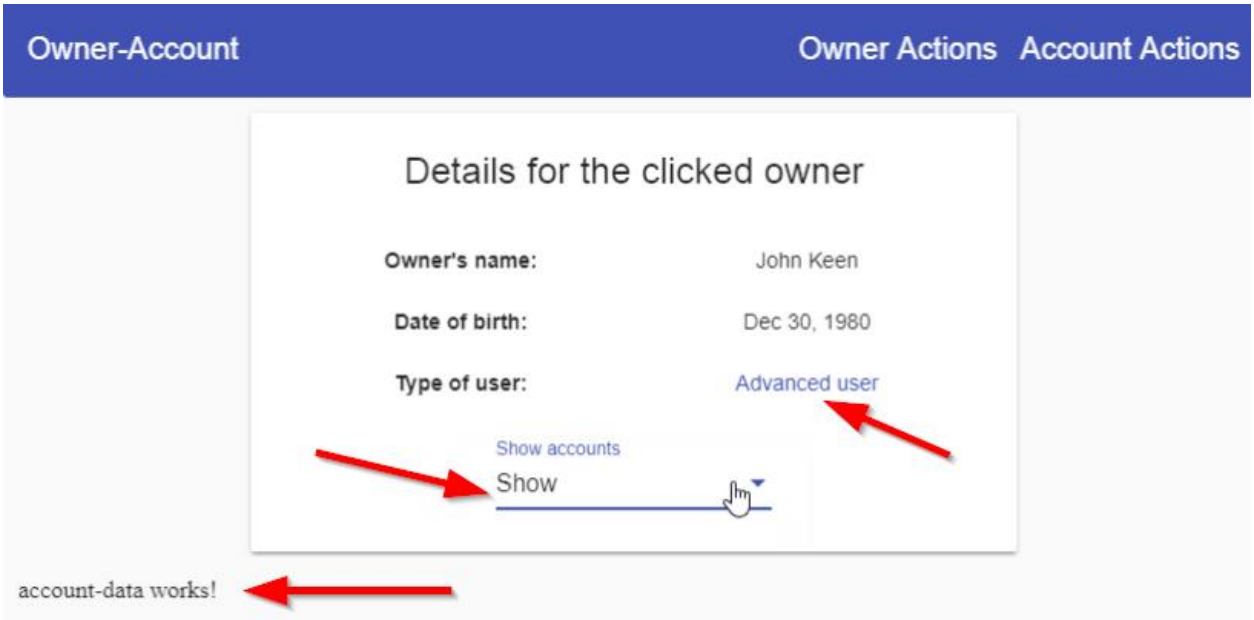
```
    @Input() public owner: Owner;
    public selectOptions = [{ name: 'Show', value: 'show' }, { name: `Don't Show`, value:
'' }];
    @Output() selectEmitt = new EventEmitter();

    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;
}
```

In order to show our data, 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>
```



Using Mat-Expansion-Panel

Let's modify the HTML part of the Account-Data 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>  
{{account?.accountType}}</div>
                <div fxFlex class="text-color"><strong>created:</strong>  
{{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:

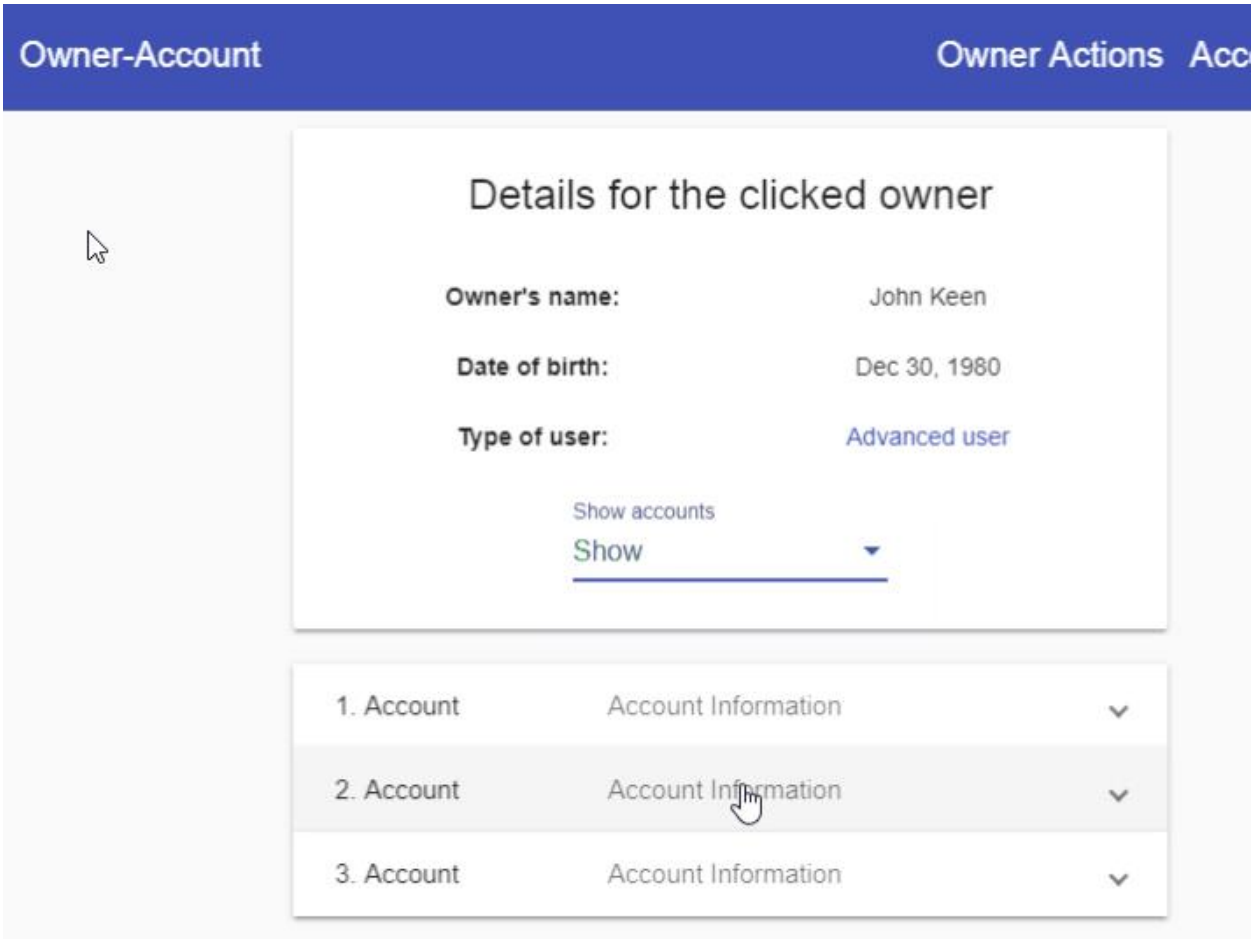
```
.text-color {
    color: #3f51b5;
}

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>
```

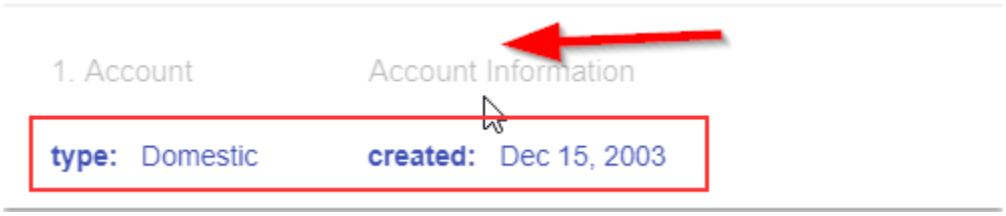
Our completed page should look like this:



Mat-Expansion-Panel Additional Features

We can disable our panel or make it expanded with the **disabled** and **expanded** attributes:

```
<mat-expansion-panel [expanded]='true' [disabled]='true' *ngFor="let account of accounts; let i = index">
```



We can react on the open and close events on the panel component by using the **opened** and **closed** @Output decorators:

```
<mat-expansion-panel *ngFor="let account of accounts; let i = index"
(closed)='onClosed()' (opened)='onOpened()'>
```

By default, the expansion panel content is going to be initialized even though the panel itself is not expanded. If we have a lot of data to show in our panel, the better way would be to defer the content loading until the panel is expanded. We can do that by using the **ng-template**:

```
<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>
    <ng-template matExpansionPanelContent>
        <div fxLayout="row wrap" fxLayoutAlign="center center">
            <div fxFlex="35%" class="text-color"><strong>type:</strong>  
{{account?.accountType}}</div>
            <div fxFlex class="text-color"><strong>created:</strong>  
{{account?.dateCreated | date}}</div>
        </div>
    </ng-template>

</mat-expansion-panel>
```

Input, Datepicker and Modal Components in Material Forms

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 chapter, we are going to use the input components to create a Create-Owner component and use it to create a new Owner object in our database.

Of course, we will show how easy it is to apply Angular Material Form Validation with the material input components and also how to create dialogs to show the error or success messages.

Let's start step by step.

☙ Routing Configuration And Component Creation

To create our new component, we have to type a familiar command:

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

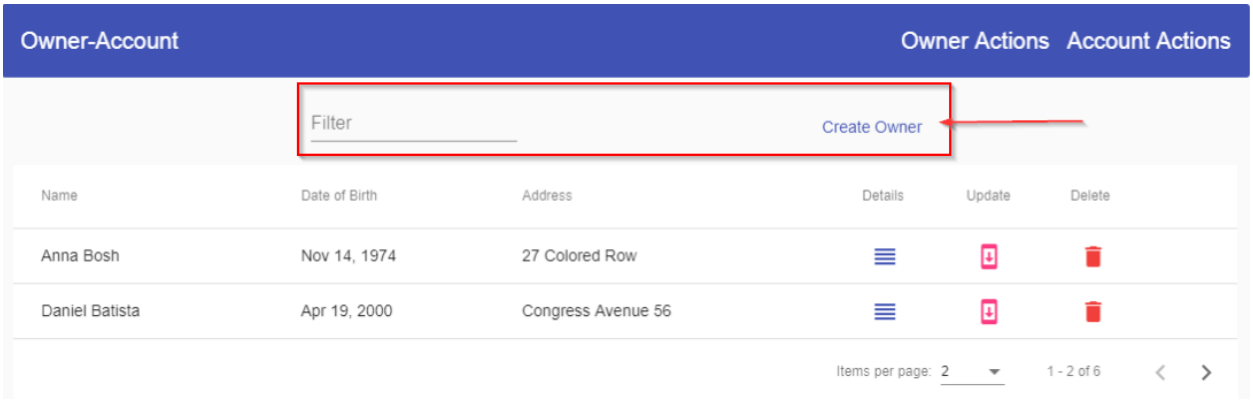
Once we have our component created, let's configure the routing part in the **owner-routing.module.ts** file:

```
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.

☙ Using Material Input Component for the 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 module for the reactive form validation.

In addition to input components, 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,
```

After all these imports and registrations, 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-cart-actions** elements need 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** tag, 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>
```

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;
}
```

Right after that, we are going to modify our `.ts` file:

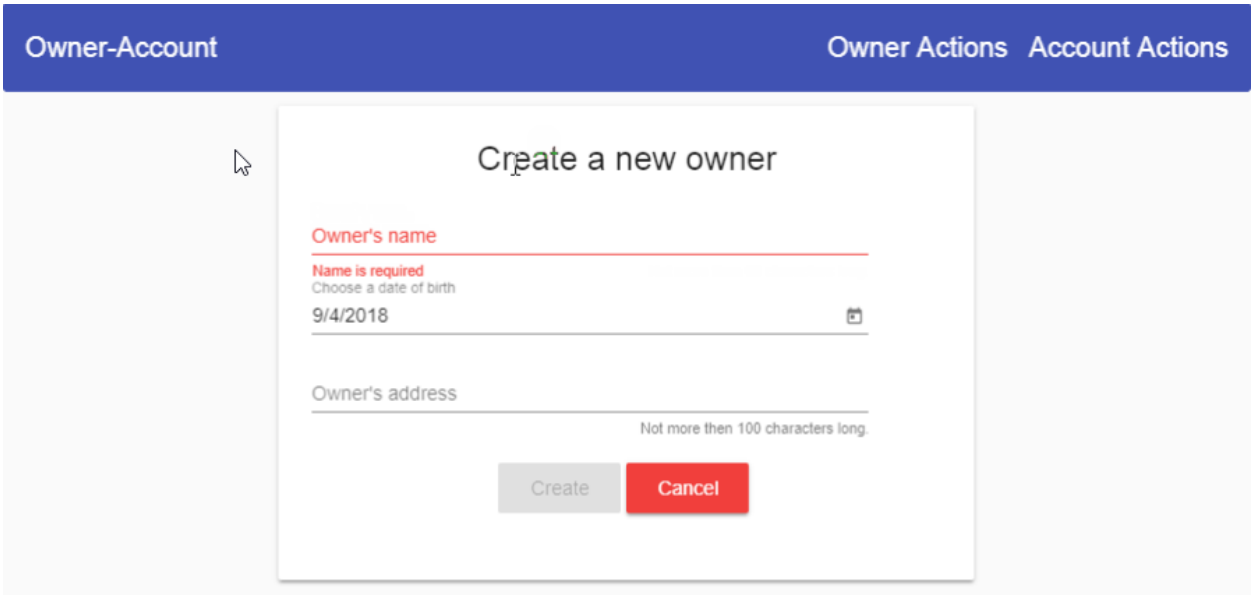
```
import { RepositoryService } from './../../shared/repository.service';
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { Location } from '@angular/common';
import { OwnerForCreation } from '../../_interface/ownerForCreation.model';

@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 have implemented Angular Material Form Validation in a couple of steps and now we can check the result:



☙ Modal Dialog Component and Shared Module

We have finished the owner component 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.

But 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 create a shared module first:

```
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 now have the FlexLayoutModule and MaterialModule inside of the shared module file, we don't need them anymore in the app and owner module files. Therefore, we can remove the FlexLayoutModule and MaterialModule imports from the app and owner module files and just import the SharedModule in both mentioned module files (app and owner).*

To continue, we are going to create our dialog components:

```
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 need to 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 and 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 instructs this button to close the dialog and submits a

result (true in this case). To fetch this result, we need to subscribe to the `afterClosed()` function. 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: {}
    }
}
```

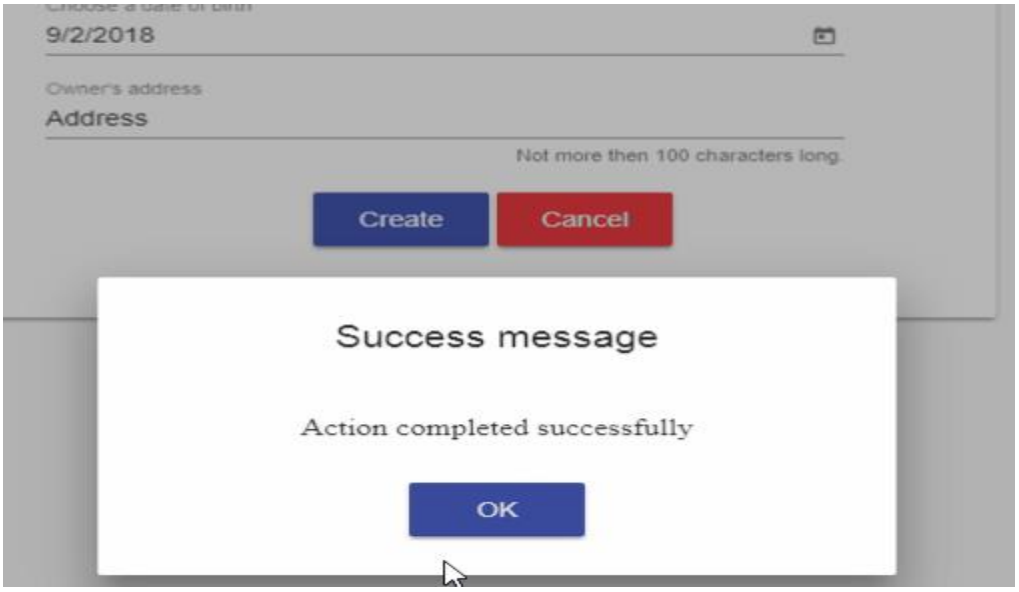
To start our success dialog, we need to import `MatDialog` and to create a private variable in a constructor of the same type.

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 `data` 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 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 data acceptance 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>
```

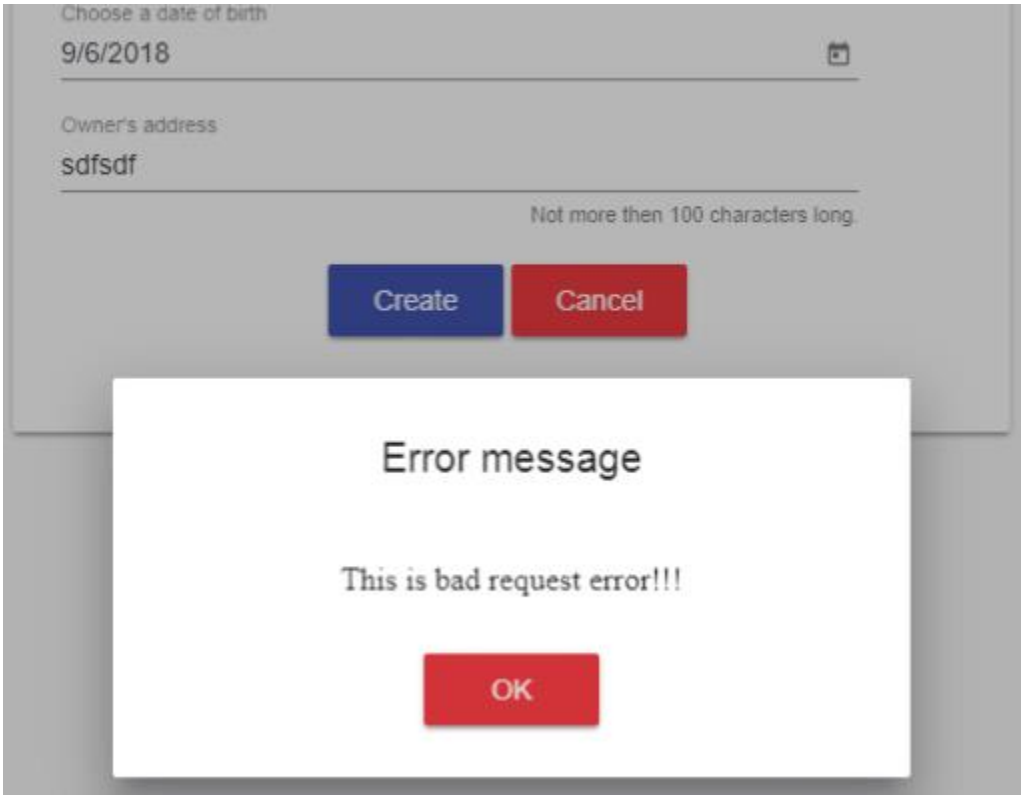
Having that done, let's modify the **error-handler.service** file:

```
...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:



Final Words

Excellent. We have covered a lot of Angular Material components and lots of features with those components as well. This project should help you with your own project development for sure and in the overall understanding of Material design.

If you want to dive even deeper into the Material development, you can visit the official Angular Material page on this link: https://material.angular.io/.

We hope you have enjoyed reading this book as well as we enjoyed writing it.