**Angular 11 - CRUD Example with Reactive Forms**

Tutorial built with **Angular 11.0.4**

shows how to build a basic Angular 11 CRUD application with Reactive Forms that includes pages for listing, adding, editing and deleting records from a JSON API. The records in the example app are user records, but the same CRUD pattern and code structure could be used to manage any type of data e.g. products, services, articles etc.

**Fake backend API with CRUD routes**

The example app runs with a [fake backend](https://jasonwatmore.com/#fake-backend-ts) api by default to enable it to run completely in the browser without a real api (backend-less), the fake api contains routes for user CRUD operations (Create, Read, Update, Delete) and it uses browser local storage to save data. To disable the fake backend you just have to remove a couple of lines of code from the [app.module.ts](https://jasonwatmore.com/" \l "app-module-ts) file, you can build your own api or hook it up with the .NET CRUD api available (instructions below).

**Example CRUD App Overview**

The example app includes a basic home page and users section with CRUD functionality, the default page in the users section displays a list of all users and includes buttons to add, edit and delete users. The add and edit buttons navigate to a page containing an Angular Reactive Form for creating or updating a user record, and the delete button executes a function within the user list component to delete a user record. The add and edit forms are both implemented with the same [add/edit component](https://jasonwatmore.com/#users-add-edit-component-ts) which behaves differently depending on which mode it is in ("add mode" vs "edit mode").

The example project is available on GitHub at <https://github.com/cornflourblue/angular-11-crud-example>.

Here it is in action: (See on StackBlitz at <https://stackblitz.com/edit/angular-11-crud-example>)

**Run the Angular CRUD Example Locally**

1. Install Node.js and npm from [https://nodejs.org](https://nodejs.org/).
2. Download or clone the Angular project source code from <https://github.com/cornflourblue/angular-11-crud-example>
3. Install all required npm packages by running npm install or npm i from the command line in the project root folder (where the package.json is located).
4. Start the app by running npm start from the command line in the project root folder, this will compile the Angular app and automatically launch it in the browser on the URL http://localhost:4200.

NOTE: You can also start the CRUD app with the Angular CLI command ng serve --open. To do this first install the Angular CLI globally on your system with the command npm install -g @angular/cli.

For more info on setting up your local Angular dev environment see [Angular - Setup Development Environment](https://jasonwatmore.com/post/2020/06/02/angular-setup-development-environment).

**Run the Angular CRUD App with a .NET API**

For full details about the example .NET API see the tutorial [.NET 5.0 - CRUD API Example and Tutorial](https://jasonwatmore.com/post/2021/09/28/net-5-crud-api-example-and-tutorial). But to get up and running quickly just follow the below steps.

1. Install the .NET SDK from <https://dotnet.microsoft.com/download>.
2. Download or clone the project source code from <https://github.com/cornflourblue/dotnet-5-crud-api>
3. Start the api by running dotnet run from the command line in the project root folder (where the WebApi.csproj file is located), you should see the message Now listening on: http://localhost:4000.
4. Back in the Angular app, remove or comment out the line below the comment // provider used to create fake backend located in the /src/app/app.module.ts file, then start the Angular app and it should now be hooked up with the .NET API.

**Run the Angular CRUD App with a Node.js + MySQL API**

For full details about the example Node.js + MySQL API see the tutorial [Node.js + MySQL - CRUD API Example and Tutorial](https://jasonwatmore.com/post/2021/11/22/nodejs-mysql-crud-api-example-and-tutorial). But to get up and running quickly just follow the below steps.

1. Install MySQL Community Server from <https://dev.mysql.com/downloads/mysql/> and ensure it is started. Installation instructions are available at <https://dev.mysql.com/doc/refman/8.0/en/installing.html>.
2. Download or clone the project source code from <https://github.com/cornflourblue/node-mysql-crud-api>
3. Install all required npm packages by running npm install or npm i from the command line in the project root folder (where the package.json is located).
4. Update the database credentials in /config.json to connect to your MySQL server instance.
5. Start the api by running npm start from the command line in the project root folder, you should see the message Server listening on port 4000.
6. Back in the Angular app, remove or comment out the line below the comment // provider used to create fake backend located in the /src/app/app.module.ts file, then start the Angular app and it should now be hooked up with the Node + MySQL API.

**Project Structure**

The Angular CLI was used to generate the base project structure with the ng new <project name> command and with the "enforce stricter type checking" (strict mode) option enabled, the CLI is also used to build and serve the application. For more info on the Angular CLI see <https://angular.io/cli>, and for more info on Angular strict mode see <https://angular.io/guide/strict-mode>.

The app and code structure of the tutorial mostly follow the best practice recommendations in the official [Angular Style Guide](https://angular.io/guide/styleguide), with a few of my own tweaks here and there.

Each feature has it's own folder (home & users), other shared/common code such as components, services, models, helpers etc are placed in folders prefixed with an underscore \_ to easily differentiate them from features and group them together at the top of the folder structure.

The index.ts files in some folders are barrel files that group the exported modules from that folder together so they can be imported using only the folder path instead of the full module path, and to enable importing multiple modules in a single import (e.g. import { UserService, AlertService } from '@app/\_services').

The users feature is a self contained feature module that manages its own layout, routes and components, and is hooked into the main Angular CRUD app inside the [app routing module](https://jasonwatmore.com/#app-routing-module-ts) with lazy loading.

Path aliases @app and @environments have been configured in [tsconfig.json](https://jasonwatmore.com/" \l "tsconfig-json) that map to the /src/app and /src/environments directories. This allows imports to be relative to the app and environments folders by prefixing import paths with aliases instead of having to use long relative paths (e.g. import MyComponent from '../../../MyComponent').

Here are the main project files that contain the CRUD application logic, I left out some files that were generated by Angular CLI ng new command that I didn't change.

* src
  + app
    - \_components
      * [alert.component.html](https://jasonwatmore.com/#alert-component-html)
      * [alert.component.ts](https://jasonwatmore.com/#alert-component-ts)
      * index.ts
    - \_helpers
      * [error.interceptor.ts](https://jasonwatmore.com/#error-interceptor-ts)
      * [fake-backend.ts](https://jasonwatmore.com/#fake-backend-ts)
      * [must-match.validator.ts](https://jasonwatmore.com/#must-match-validator-ts)
      * index.ts
    - \_models
      * [alert.ts](https://jasonwatmore.com/#alert-ts)
      * [role.ts](https://jasonwatmore.com/#role-ts)
      * [user.ts](https://jasonwatmore.com/#user-ts)
      * index.ts
    - \_services
      * [alert.service.ts](https://jasonwatmore.com/#alert-service-ts)
      * [user.service.ts](https://jasonwatmore.com/#user-service-ts)
      * index.ts
    - home
      * [home.component.html](https://jasonwatmore.com/#home-component-html)
      * [home.component.ts](https://jasonwatmore.com/#home-component-ts)
      * index.ts
    - users
      * [add-edit.component.html](https://jasonwatmore.com/#users-add-edit-component-html)
      * [add-edit.component.ts](https://jasonwatmore.com/#users-add-edit-component-ts)
      * [layout.component.html](https://jasonwatmore.com/#users-layout-component-html)
      * [layout.component.ts](https://jasonwatmore.com/#users-layout-component-ts)
      * [list.component.html](https://jasonwatmore.com/#users-list-component-html)
      * [list.component.ts](https://jasonwatmore.com/#users-list-component-ts)
      * [users-routing.module.ts](https://jasonwatmore.com/#users-routing-module-ts)
      * [users.module.ts](https://jasonwatmore.com/#users-module-ts)
      * index.ts
    - [app-routing.module.ts](https://jasonwatmore.com/#app-routing-module-ts)
    - [app.component.html](https://jasonwatmore.com/#app-component-html)
    - [app.component.ts](https://jasonwatmore.com/#app-component-ts)
    - [app.module.ts](https://jasonwatmore.com/#app-module-ts)
  + environments
    - [environment.prod.ts](https://jasonwatmore.com/#environment-prod-ts)
    - [environment.ts](https://jasonwatmore.com/#environment-ts)
  + [index.html](https://jasonwatmore.com/#index-html)
  + [main.ts](https://jasonwatmore.com/#main-ts)
  + [polyfills.ts](https://jasonwatmore.com/#polyfills-ts)
  + [styles.less](https://jasonwatmore.com/#styles-less)
* [package.json](https://jasonwatmore.com/#package-json)
* [tsconfig.json](https://jasonwatmore.com/#tsconfig-json)

**Alert Component Template**

**Path: /src/app/\_components/alert.component.html**

The alert component template contains the html for displaying alert messages at the top of the page, it renders a notification for each alert in the alerts array of the [alert component](https://jasonwatmore.com/#alert-component-ts) below.

<div \*ngFor="let alert of alerts" class="{{cssClass(alert)}}">

<a class="close" (click)="removeAlert(alert)">&times;</a>

<span [innerHTML]="alert.message"></span>

</div>

[Back to top](https://jasonwatmore.com/#project-structure)

**Alert Component**

**Path: /src/app/\_components/alert.component.ts**

The alert component controls the adding & removing of alerts in the UI, it maintains an array of alerts that are rendered by the component template.

The component subscribes to receive new alerts from the alert service in the ngOnInit method by calling the alertService.onAlert() method, new alerts are added to the alerts array for display. Alerts are cleared when an alert with an empty message is received from the alert service. The ngOnInit method also calls router.events.subscribe() to subscribe to route change events so it can automatically clear alerts on route changes.

The ngOnDestroy() method unsubscribes from the alert service and router when the component is destroyed to prevent memory leaks from orphaned subscriptions.

The removeAlert() method removes the specified alert object from the array, which allows individual alerts to be closed in the UI.

The cssClass() method returns a corresponding bootstrap alert class for each of the alert types, if you're using something other than bootstrap you can change the CSS classes returned to suit your application.

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

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

import { Subscription } from 'rxjs';

import { Alert, AlertType } from '@app/\_models';

import { AlertService } from '@app/\_services';

@Component({ selector: 'alert', templateUrl: 'alert.component.html' })

export class AlertComponent implements OnInit, OnDestroy {

@Input() id = 'default-alert';

@Input() fade = true;

alerts: Alert[] = [];

alertSubscription!: Subscription;

routeSubscription!: Subscription;

constructor(private router: Router, private alertService: AlertService) { }

ngOnInit() {

// subscribe to new alert notifications

this.alertSubscription = this.alertService.onAlert(this.id)

.subscribe(alert => {

// clear alerts when an empty alert is received

if (!alert.message) {

// filter out alerts without 'keepAfterRouteChange' flag

this.alerts = this.alerts.filter(x => x.keepAfterRouteChange);

// reset 'keepAfterRouteChange' flag on the rest

this.alerts.forEach(x => x.keepAfterRouteChange = false);

return;

}

// add alert to array

this.alerts.push(alert);

// auto close alert if required

if (alert.autoClose) {

setTimeout(() => this.removeAlert(alert), 3000);

}

});

// clear alerts on location change

this.routeSubscription = this.router.events.subscribe(event => {

if (event instanceof NavigationStart) {

this.alertService.clear(this.id);

}

});

}

ngOnDestroy() {

// unsubscribe to avoid memory leaks

this.alertSubscription.unsubscribe();

this.routeSubscription.unsubscribe();

}

removeAlert(alert: Alert) {

// check if already removed to prevent error on auto close

if (!this.alerts.includes(alert)) return;

if (this.fade) {

// fade out alert

alert.fade = true;

// remove alert after faded out

setTimeout(() => {

this.alerts = this.alerts.filter(x => x !== alert);

}, 250);

} else {

// remove alert

this.alerts = this.alerts.filter(x => x !== alert);

}

}

cssClass(alert: Alert) {

if (alert?.type === undefined) return;

const classes = ['alert', 'alert-dismissable', 'mt-4', 'container'];

const alertTypeClass = {

[AlertType.Success]: 'alert-success',

[AlertType.Error]: 'alert-danger',

[AlertType.Info]: 'alert-info',

[AlertType.Warning]: 'alert-warning'

}

classes.push(alertTypeClass[alert.type]);

if (alert.fade) {

classes.push('fade');

}

return classes.join(' ');

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Error Interceptor**

**Path: /src/app/\_helpers/error.interceptor.ts**

The Error Interceptor intercepts http responses from the api to check if there were any errors, if there were the error message is displayed with the [alert service](https://jasonwatmore.com/#alert-service-ts), logged to the console and re-thrown to the calling object to enable additional error handling logic to be added there if required.

It's implemented using the Angular HttpInterceptor interface included in the HttpClientModule, by implementing the HttpInterceptor interface you can create a custom interceptor to catch all error responses from the server in a single location.

Http interceptors are added to the request pipeline in the providers section of the [app.module.ts](https://jasonwatmore.com/" \l "app-module-ts) file.

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

import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { catchError } from 'rxjs/operators';

import { AlertService } from '@app/\_services';

@Injectable()

export class ErrorInterceptor implements HttpInterceptor {

constructor(private alertService: AlertService) {}

intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(request).pipe(catchError(err => {

const error = err.error?.message || err.statusText;

this.alertService.error(error);

console.error(err);

return throwError(error);

}))

}

}

[Back to top](https://jasonwatmore.com/#projectstructure)

**Fake Backend Provider**

**Path: /src/app/\_helpers/fake-backend.ts**

In order to run and test the Angular 11 CRUD example without a real backend API, the application uses a fake backend that intercepts the HTTP requests from the Angular app and sends back "fake" responses. This is done by a class that implements the Angular HttpInterceptor interface, for more information on Angular HTTP Interceptors see <https://angular.io/api/common/http/HttpInterceptor> or [this article](https://medium.com/@ryanchenkie_40935/angular-authentication-using-the-http-client-and-http-interceptors-2f9d1540eb8).

The fake backend contains a handleRoute function that checks if the request matches one of the faked routes in the switch statement, at the moment this includes requests for handling user CRUD operations. Matching requests are intercepted and handled by one of the below // route functions, non-matching requests are sent through to the real backend by calling next.handle(request);. Below the route functions there are // helper functions for returning different response types and performing small tasks.

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

import { HttpRequest, HttpResponse, HttpHandler, HttpEvent, HttpInterceptor, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { Observable, of, throwError } from 'rxjs';

import { delay, materialize, dematerialize } from 'rxjs/operators';

import { Role } from '@app/\_models';

// array in local storage for registered users

const usersKey = 'angular-11-crud-example-users';

const usersJSON = localStorage.getItem(usersKey);

let users: any[] = usersJSON ? JSON.parse(usersJSON) : [{

id: 1,

title: 'Mr',

firstName: 'Joe',

lastName: 'Bloggs',

email: 'joe@bloggs.com',

role: Role.User,

password: 'joe123'

}];

@Injectable()

export class FakeBackendInterceptor implements HttpInterceptor {

intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const { url, method, headers, body } = request;

return handleRoute();

function handleRoute() {

switch (true) {

case url.endsWith('/users') && method === 'GET':

return getUsers();

case url.match(/\/users\/\d+$/) && method === 'GET':

return getUserById();

case url.endsWith('/users') && method === 'POST':

return createUser();

case url.match(/\/users\/\d+$/) && method === 'PUT':

return updateUser();

case url.match(/\/users\/\d+$/) && method === 'DELETE':

return deleteUser();

default:

// pass through any requests not handled above

return next.handle(request);

}

}

// route functions

function getUsers() {

return ok(users.map(x => basicDetails(x)));

}

function getUserById() {

const user = users.find(x => x.id === idFromUrl());

return ok(basicDetails(user));

}

function createUser() {

const user = body;

if (users.find(x => x.email === user.email)) {

return error(`User with the email ${user.email} already exists`);

}

// assign user id and a few other properties then save

user.id = newUserId();

delete user.confirmPassword;

users.push(user);

localStorage.setItem(usersKey, JSON.stringify(users));

return ok();

}

function updateUser() {

let params = body;

let user = users.find(x => x.id === idFromUrl());

if (params.email !== user.email && users.find(x => x.email === params.email)) {

return error(`User with the email ${params.email} already exists`);

}

// only update password if entered

if (!params.password) {

delete params.password;

}

// update and save user

Object.assign(user, params);

localStorage.setItem(usersKey, JSON.stringify(users));

return ok();

}

function deleteUser() {

users = users.filter(x => x.id !== idFromUrl());

localStorage.setItem(usersKey, JSON.stringify(users));

return ok();

}

// helper functions

function ok(body?: any) {

return of(new HttpResponse({ status: 200, body }))

.pipe(delay(500)); // delay observable to simulate server api call

}

function error(message: any) {

return throwError({ error: { message } })

.pipe(materialize(), delay(500), dematerialize()); // call materialize and dematerialize to ensure delay even if an error is thrown (https://github.com/Reactive-Extensions/RxJS/issues/648);

}

function basicDetails(user: any) {

const { id, title, firstName, lastName, email, role } = user;

return { id, title, firstName, lastName, email, role };

}

function idFromUrl() {

const urlParts = url.split('/');

return parseInt(urlParts[urlParts.length - 1]);

}

function newUserId() {

return users.length ? Math.max(...users.map(x => x.id)) + 1 : 1;

}

}

}

export const fakeBackendProvider = {

// use fake backend in place of Http service for backend-less development

provide: HTTP\_INTERCEPTORS,

useClass: FakeBackendInterceptor,

multi: true

};

[Back to top](https://jasonwatmore.com/#project-structure)

**Must Match Validator**

**Path: /src/app/\_helpers/must-match.validator.ts**

The reactive forms MustMatch validator is a custom validator used to ensure that two fields match. It is used by the [users add/edit component](https://jasonwatmore.com/#users-add-edit-component-ts) to validate that the password and confirm password fields match.

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

// custom validator to check that two fields match

export function MustMatch(controlName: string, matchingControlName: string) {

return (group: AbstractControl) => {

const formGroup = <FormGroup>group;

const control = formGroup.controls[controlName];

const matchingControl = formGroup.controls[matchingControlName];

if (matchingControl.errors && !matchingControl.errors.mustMatch) {

// return if another validator has already found an error on the matchingControl

return null;

}

// set error on matchingControl if validation fails

if (control.value !== matchingControl.value) {

matchingControl.setErrors({ mustMatch: true });

} else {

matchingControl.setErrors(null);

}

return null;

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Alert Model and Alert Type Enum**

**Path: /src/app/\_models/alert.ts**

The Alert model defines the properties of each alert object, and the AlertType enum defines the types of alerts allowed in the Angular CRUD application.

export class Alert {

constructor(

public id: string,

public type?: AlertType,

public message?: string,

public autoClose: boolean = true,

public keepAfterRouteChange: boolean = false,

public fade: boolean = false

) {}

}

export enum AlertType {

Success,

Error,

Info,

Warning

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Role Enum**

**Path: /src/app/\_models/role.ts**

The role enum defines the roles that are supported by the Angular CRUD example.

export enum Role {

User = 'User',

Admin = 'Admin'

}

[Back to top](https://jasonwatmore.com/#project-structure)

**User Model**

**Path: /src/app/\_models/user.ts**

The user model is a small class that represents the properties of a user in the Angular CRUD app. It is used by the [user service](https://jasonwatmore.com/#user-service-ts) to return strongly typed user objects from the API.

The exclamation point (!) modifier on most of the properties is the TypeScript *definite assignment assertion* modifier, it tells the TypeScript compiler that these properties are initialized outside of the class (e.g. in the getAll() and getById() methods of the user service). The definite assignment assertion operator prevents errors from the TypeScript compiler when strict property initialization is enabled in the [tsconfig.json](https://jasonwatmore.com/" \l "tsconfig-json) file with "strict": true or "strictPropertyInitialization": true.

import { Role } from './role';

export class User {

id!: string;

title!: string;

firstName!: string;

lastName!: string;

email!: string;

role!: Role;

isDeleting: boolean = false;

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Alert Service**

**Path: /src/app/\_services/alert.service.ts**

The alert service acts as the bridge between any component in an Angular CRUD example and the [alert component](https://jasonwatmore.com/#alert-component-ts) that renders alert messages in the UI. It contains methods for sending, clearing and subscribing to alert messages.

You can trigger alert notifications from any component or service by calling one of the convenience methods for displaying the different types of alerts: success(), error(), info() and warn().

**Alert convenience method parameters**

* The first parameter is the alert message string, which can be plain text or HTML
* The second parameter is an optional options object that supports an autoClose boolean property and keepAfterRouteChange boolean property:
  + autoClose - if true tells the alert component to automatically close the alert after three seconds. Default is true.
  + keepAfterRouteChange - if true prevents the alert from being closed after one route change, this is handy for displaying messages after a redirect such as when a new user is created or updated. Default is false.

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

import { Observable, Subject } from 'rxjs';

import { filter } from 'rxjs/operators';

import { Alert, AlertType } from '@app/\_models';

@Injectable({ providedIn: 'root' })

export class AlertService {

private subject = new Subject<Alert>();

private defaultId = 'default-alert';

// enable subscribing to alerts observable

onAlert(id = this.defaultId): Observable<Alert> {

return this.subject.asObservable().pipe(filter(x => x && x.id === id));

}

// convenience methods

success(message: string, options?: Partial<Alert>) {

this.alert(message, AlertType.Success, options);

}

error(message: string, options?: Partial<Alert>) {

this.alert(message, AlertType.Error, options);

}

info(message: string, options?: Partial<Alert>) {

this.alert(message, AlertType.Info, options);

}

warn(message: string, options?: Partial<Alert>) {

this.alert(message, AlertType.Warning, options);

}

// main alert method

alert(message: string, type: AlertType, options: Partial<Alert> = {}) {

const id = options.id || this.defaultId;

const alert = new Alert(id, type, message, options.autoClose, options.keepAfterRouteChange);

this.subject.next(alert);

}

// clear alerts

clear(id = this.defaultId) {

this.subject.next(new Alert(id));

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**User Service**

**Path: /src/app/\_services/user.service.ts**

The user service handles communication between the Angular 11 CRUD app and the backend api, it contains standard CRUD methods for managing users that make corresponding HTTP requests to the /users endpoint of the api with the Angular HttpClient.

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

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

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

import { User } from '@app/\_models';

const baseUrl = `${environment.apiUrl}/users`;

@Injectable({ providedIn: 'root' })

export class UserService {

constructor(private http: HttpClient) { }

getAll() {

return this.http.get<User[]>(baseUrl);

}

getById(id: string) {

return this.http.get<User>(`${baseUrl}/${id}`);

}

create(params: any) {

return this.http.post(baseUrl, params);

}

update(id: string, params: any) {

return this.http.put(`${baseUrl}/${id}`, params);

}

delete(id: string) {

return this.http.delete(`${baseUrl}/${id}`);

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Home Component Template**

**Path: /src/app/home/home.component.html**

The home component template contains html and angular template syntax for displaying a simple welcome message and a link to the users section.

<div class="p-4">

<div class="container">

<h1>Angular 11 - CRUD Example with Reactive Forms</h1>

<p>An example app showing how to list, add, edit and delete user records with Angular 11.</p>

<p><a routerLink="/users">&gt;&gt; Manage Users</a></p>

</div>

</div>

[Back to top](https://jasonwatmore.com/#project-structure)

**Home Component**

**Path: /src/app/home/home.component.ts**

The home component is the default component of the CRUD example, it is bound to the [home template](https://jasonwatmore.com/#home-component-html) with the templateUrl property of the angular @Component decorator.

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

@Component({ templateUrl: 'home.component.html' })

export class HomeComponent { }

[Back to top](https://jasonwatmore.com/#project-structure)

**Users Add/Edit Component Template**

**Path: /src/app/users/add-edit.component.html**

The users add/edit component template contains a dynamic form that supports both creating and updating users. The isAddMode property is used to change what is displayed based on which mode it is in, for example the form title and password optional message.

<h1 \*ngIf="isAddMode">Add User</h1>

<h1 \*ngIf="!isAddMode">Edit User</h1>

<form [formGroup]="form" (ngSubmit)="onSubmit()">

<div class="form-row">

<div class="form-group col">

<label>Title</label>

<select formControlName="title" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.title.errors }">

<option value=""></option>

<option value="Mr">Mr</option>

<option value="Mrs">Mrs</option>

<option value="Miss">Miss</option>

<option value="Ms">Ms</option>

</select>

<div \*ngIf="submitted && f.title.errors" class="invalid-feedback">

<div \*ngIf="f.title.errors.required">Title is required</div>

</div>

</div>

<div class="form-group col-5">

<label>First Name</label>

<input type="text" formControlName="firstName" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.firstName.errors }" />

<div \*ngIf="submitted && f.firstName.errors" class="invalid-feedback">

<div \*ngIf="f.firstName.errors.required">First Name is required</div>

</div>

</div>

<div class="form-group col-5">

<label>Last Name</label>

<input type="text" formControlName="lastName" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.lastName.errors }" />

<div \*ngIf="submitted && f.lastName.errors" class="invalid-feedback">

<div \*ngIf="f.lastName.errors.required">Last Name is required</div>

</div>

</div>

</div>

<div class="form-row">

<div class="form-group col-7">

<label>Email</label>

<input type="text" formControlName="email" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.email.errors }" />

<div \*ngIf="submitted && f.email.errors" class="invalid-feedback">

<div \*ngIf="f.email.errors.required">Email is required</div>

<div \*ngIf="f.email.errors.email">Email must be a valid email address</div>

</div>

</div>

<div class="form-group col">

<label>Role</label>

<select formControlName="role" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.role.errors }">

<option value=""></option>

<option value="User">User</option>

<option value="Admin">Admin</option>

</select>

<div \*ngIf="submitted && f.role.errors" class="invalid-feedback">

<div \*ngIf="f.role.errors.required">Role is required</div>

</div>

</div>

</div>

<div \*ngIf="!isAddMode">

<h3 class="pt-3">Change Password</h3>

<p>Leave blank to keep the same password</p>

</div>

<div class="form-row">

<div class="form-group col">

<label>Password</label>

<input type="password" formControlName="password" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.password.errors }" />

<div \*ngIf="submitted && f.password.errors" class="invalid-feedback">

<div \*ngIf="f.password.errors.required">Password is required</div>

<div \*ngIf="f.password.errors.minlength">Password must be at least 6 characters</div>

</div>

</div>

<div class="form-group col">

<label>Confirm Password</label>

<input type="password" formControlName="confirmPassword" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.confirmPassword.errors }" />

<div \*ngIf="submitted && f.confirmPassword.errors" class="invalid-feedback">

<div \*ngIf="f.confirmPassword.errors.required">Confirm Password is required</div>

<div \*ngIf="f.confirmPassword.errors.mustMatch">Passwords must match</div>

</div>

</div>

</div>

<div class="form-group">

<button [disabled]="loading" class="btn btn-primary">

<span \*ngIf="loading" class="spinner-border spinner-border-sm mr-1"></span>

Save

</button>

<a routerLink="/users" class="btn btn-link">Cancel</a>

</div>

</form>

[Back to top](https://jasonwatmore.com/#project-structure)

**Users Add/Edit Component**

**Path: /src/app/users/add-edit.component.ts**

The users add/edit component is used for both adding and editing users in the angular CRUD app, the component is in "add mode" when there is no user id route parameter, otherwise it is in "edit mode". The property isAddMode is used to change the component behaviour based on which mode it is in, for example in "add mode" the password field is required, and in "edit mode" (!this.isAddMode) the user service is called when the component initializes to get the user details (this.userService.getById(this.id)) to pre-populate the field values.

On submit a user is either created or updated by calling the user service, and on success you are redirected back to the users list page with a success message.

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

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

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

import { first } from 'rxjs/operators';

import { UserService, AlertService } from '@app/\_services';

import { MustMatch } from '@app/\_helpers';

@Component({ templateUrl: 'add-edit.component.html' })

export class AddEditComponent implements OnInit {

form!: FormGroup;

id!: string;

isAddMode!: boolean;

loading = false;

submitted = false;

constructor(

private formBuilder: FormBuilder,

private route: ActivatedRoute,

private router: Router,

private userService: UserService,

private alertService: AlertService

) {}

ngOnInit() {

this.id = this.route.snapshot.params['id'];

this.isAddMode = !this.id;

// password not required in edit mode

const passwordValidators = [Validators.minLength(6)];

if (this.isAddMode) {

passwordValidators.push(Validators.required);

}

const formOptions: AbstractControlOptions = { validators: MustMatch('password', 'confirmPassword') };

this.form = this.formBuilder.group({

title: ['', Validators.required],

firstName: ['', Validators.required],

lastName: ['', Validators.required],

email: ['', [Validators.required, Validators.email]],

role: ['', Validators.required],

password: ['', [Validators.minLength(6), this.isAddMode ? Validators.required : Validators.nullValidator]],

confirmPassword: ['', this.isAddMode ? Validators.required : Validators.nullValidator]

}, formOptions);

if (!this.isAddMode) {

this.userService.getById(this.id)

.pipe(first())

.subscribe(x => this.form.patchValue(x));

}

}

// convenience getter for easy access to form fields

get f() { return this.form.controls; }

onSubmit() {

this.submitted = true;

// reset alerts on submit

this.alertService.clear();

// stop here if form is invalid

if (this.form.invalid) {

return;

}

this.loading = true;

if (this.isAddMode) {

this.createUser();

} else {

this.updateUser();

}

}

private createUser() {

this.userService.create(this.form.value)

.pipe(first())

.subscribe(() => {

this.alertService.success('User added', { keepAfterRouteChange: true });

this.router.navigate(['../'], { relativeTo: this.route });

})

.add(() => this.loading = false);

}

private updateUser() {

this.userService.update(this.id, this.form.value)

.pipe(first())

.subscribe(() => {

this.alertService.success('User updated', { keepAfterRouteChange: true });

this.router.navigate(['../../'], { relativeTo: this.route });

})

.add(() => this.loading = false);

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Users Layout Component Template**

**Path: /src/app/users/layout.component.html**

The users layout component template is the root template of the users feature / section of the CRUD example, it contains the outer HTML for all /users pages and a <router-outlet> for rendering the currently routed component.

<div class="p-4">

<div class="container">

<router-outlet></router-outlet>

</div>

</div>

[Back to top](https://jasonwatmore.com/#project-structure)

**Users Layout Component**

**Path: /src/app/users/layout.component.ts**

The users layout component is the root component of the users feature / section of the Angular CRUD app, it binds the component to the users layout template with the templateUrl property of the angular @Component decorator.

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

@Component({ templateUrl: 'layout.component.html' })

export class LayoutComponent { }

[Back to top](https://jasonwatmore.com/#project-structure)

**Users List Component Template**

**Path: /src/app/users/list.component.html**

The users list component template displays a list of all users and contains buttons for adding, editing and deleting users in the Angular CRUD example.

<h1>Users</h1>

<a routerLink="add" class="btn btn-sm btn-success mb-2">Add User</a>

<table class="table table-striped">

<thead>

<tr>

<th style="width: 30%">Name</th>

<th style="width: 30%">Email</th>

<th style="width: 30%">Role</th>

<th style="width: 10%"></th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let user of users">

<td>{{user.title}} {{user.firstName}} {{user.lastName}}</td>

<td>{{user.email}}</td>

<td>{{user.role}}</td>

<td style="white-space: nowrap">

<a routerLink="edit/{{user.id}}" class="btn btn-sm btn-primary mr-1">Edit</a>

<button (click)="deleteUser(user.id)" class="btn btn-sm btn-danger btn-delete-user" [disabled]="user.isDeleting">

<span \*ngIf="user.isDeleting" class="spinner-border spinner-border-sm"></span>

<span \*ngIf="!user.isDeleting">Delete</span>

</button>

</td>

</tr>

<tr \*ngIf="!users">

<td colspan="4" class="text-center">

<span class="spinner-border spinner-border-lg align-center"></span>

</td>

</tr>

</tbody>

</table>

[Back to top](https://jasonwatmore.com/#project-structure)

**Users List Component**

**Path: /src/app/users/list.component.ts**

The users list component gets all users from the [user service](https://jasonwatmore.com/#user-service-ts) in the ngOnInit() angular lifecycle method and makes them available to the [users list template](https://jasonwatmore.com/#users-list-component-html) via the users property.

The deleteUser() method first sets the property user.isDeleting = true so the template displays a spinner on the delete button, it then calls this.userService.delete() to delete the user and removes the deleted user from component users array so it is removed from the UI.

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

import { first } from 'rxjs/operators';

import { UserService } from '@app/\_services';

import { User } from '@app/\_models';

@Component({ templateUrl: 'list.component.html' })

export class ListComponent implements OnInit {

users!: User[];

constructor(private userService: UserService) {}

ngOnInit() {

this.userService.getAll()

.pipe(first())

.subscribe(users => this.users = users);

}

deleteUser(id: string) {

const user = this.users.find(x => x.id === id);

if (!user) return;

user.isDeleting = true;

this.userService.delete(id)

.pipe(first())

.subscribe(() => this.users = this.users.filter(x => x.id !== id));

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**Users Routing Module**

**Path: /src/app/users/users-routing.module.ts**

The users routing module defines the routes for the [users feature module](https://jasonwatmore.com/#users-module-ts) of the example CRUD app. It includes routes for listing, adding and editing users, and a parent route for the [layout component](https://jasonwatmore.com/#users-layout-component-ts) which contains the common layout code for the users section.

The add and edit routes are different but both load the same component (AddEditComponent) which modifies its behaviour based on the route.

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

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

import { LayoutComponent } from './layout.component';

import { ListComponent } from './list.component';

import { AddEditComponent } from './add-edit.component';

const routes: Routes = [

{

path: '', component: LayoutComponent,

children: [

{ path: '', component: ListComponent },

{ path: 'add', component: AddEditComponent },

{ path: 'edit/:id', component: AddEditComponent }

]

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class UsersRoutingModule { }

[Back to top](https://jasonwatmore.com/#project-structure)

**Users Module**

**Path: /src/app/users/users.module.ts**

The users module defines the feature module for the users section of the Angular CRUD tutorial application along with metadata about the module. The imports specify which other angular modules are required by this module, and the declarations state which components belong to this module. For more info about angular modules see <https://angular.io/docs/ts/latest/guide/ngmodule.html>.

The users module is hooked into the main app inside the [app routing module](https://jasonwatmore.com/#app-routing-module-ts) with lazy loading.

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

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

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

import { UsersRoutingModule } from './users-routing.module';

import { LayoutComponent } from './layout.component';

import { ListComponent } from './list.component';

import { AddEditComponent } from './add-edit.component';

@NgModule({

imports: [

CommonModule,

ReactiveFormsModule,

UsersRoutingModule

],

declarations: [

LayoutComponent,

ListComponent,

AddEditComponent

]

})

export class UsersModule { }

[Back to top](https://jasonwatmore.com/#project-structure)

**App Routing Module**

**Path: /src/app/app-routing.module.ts**

The app routing module defines the top level routes for the angular CRUD application and generates a root routing module by passing the array of routes to the RouterModule.forRoot() method. The module is imported into the main [app module](https://jasonwatmore.com/#app-module-ts) below.

The home route maps the root path of the app to the [home component](https://jasonwatmore.com/#home-component-ts) and the users route lazy loads the [users module](https://jasonwatmore.com/#users-module-ts) and maps it to /users.

For more information on angular routing see <https://angular.io/guide/router>.

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

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

import { HomeComponent } from './home';

const usersModule = () => import('./users/users.module').then(x => x.UsersModule);

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', loadChildren: usersModule },

// otherwise redirect to home

{ path: '\*\*', redirectTo: '' }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

[Back to top](https://jasonwatmore.com/#project-structure)

**App Component Template**

**Path: /src/app/app.component.html**

The app component template is the root component template of the Angular CRUD application, it contains the main nav bar, a global alert component and a router-outlet component for displaying the contents of each view based on the current route / path.

<!-- nav -->

<nav class="navbar navbar-expand navbar-dark bg-dark">

<div class="navbar-nav">

<a class="nav-item nav-link" routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">Home</a>

<a class="nav-item nav-link" routerLink="/users" routerLinkActive="active">Users</a>

</div>

</nav>

<!-- main app container -->

<div class="app-container bg-light">

<alert></alert>

<router-outlet></router-outlet>

</div>

[Back to top](https://jasonwatmore.com/#project-structure)

**App Component**

**Path: /src/app/app.component.ts**

The app component is the root component of the CRUD example, it defines the root tag of the app as <app></app> with the selector property of the @Component() decorator and is bound to the [app component template](https://jasonwatmore.com/#app-component-html) with the templateUrl property.

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

@Component({ selector: 'app', templateUrl: 'app.component.html' })

export class AppComponent { }

[Back to top](https://jasonwatmore.com/#project-structure)

**App Module**

**Path: /src/app/app.module.ts**

The app module defines the root angular module of the CRUD application along with metadata about the module. For more info about angular modules see <https://angular.io/docs/ts/latest/guide/ngmodule.html>.

This is where the fake backend provider is added to the application, to switch to a real backend simply remove the fakeBackendProvider located below the comment // provider used to create fake backend.

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

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

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

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

// used to create fake backend

import { fakeBackendProvider } from './\_helpers';

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

import { ErrorInterceptor } from './\_helpers';

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

import { AlertComponent } from './\_components';

import { HomeComponent } from './home';

@NgModule({

imports: [

BrowserModule,

ReactiveFormsModule,

HttpClientModule,

AppRoutingModule

],

declarations: [

AppComponent,

AlertComponent,

HomeComponent

],

providers: [

{ provide: HTTP\_INTERCEPTORS, useClass: ErrorInterceptor, multi: true },

// provider used to create fake backend

fakeBackendProvider

],

bootstrap: [AppComponent]

})

export class AppModule { };

[Back to top](https://jasonwatmore.com/#project-structure)

**Production Environment Config**

**Path: /src/environments/environment.prod.ts**

The production environment config contains variables required to run the CRUD application in production. This enables you to build the application with a different configuration for each different environment (e.g. production & development) without updating the app code.

When you build the application for production with the command ng build --prod, the output environment.ts is replaced with environment.prod.ts.

export const environment = {

production: true,

apiUrl: 'http://localhost:4000'

};

[Back to top](https://jasonwatmore.com/#project-structure)

**Development Environment Config**

**Path: /src/environments/environment.ts**

The development environment config contains variables required to run the Angular CRUD application in development.

Environment config is accessed by importing the environment object into any Angular service of component with the line import { environment } from '@environments/environment' and accessing properties on the environment object, see the [user service](https://jasonwatmore.com/#user-service-ts) for an example.

export const environment = {

production: false,

apiUrl: 'http://localhost:4000'

};

[Back to top](https://jasonwatmore.com/#project-structure)

**Main Index Html File**

**Path: /src/index.html**

The main index.html file is the initial page loaded by the browser that kicks everything off. The Angular CLI (with Webpack under the hood) bundles all of the compiled javascript files together and injects them into the body of the index.html page so the scripts can be loaded and executed by the browser.

<!DOCTYPE html>

<html>

<head>

<base href="/" />

<title>Angular 11 - CRUD Example with Reactive Forms</title>

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

<!-- bootstrap css -->

<link href="//netdna.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" rel="stylesheet" />

</head>

<body>

<app>Loading...</app>

</body>

</html>

[Back to top](https://jasonwatmore.com/#project-structure)

**Main (Bootstrap) File**

**Path: /src/main.ts**

The main file is the entry point used by angular to launch and bootstrap the CRUD application.

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

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

import { AppModule } from './app/app.module';

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

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.error(err));

[Back to top](https://jasonwatmore.com/#project-structure)

**Polyfills**

**Path: /src/polyfills.ts**

Some features used by Angular are not yet supported natively by all major browsers, polyfills are used to add support for features where necessary so the Angular CRUD tutorial application works across all major browsers.

This file is generated by the Angular CLI when creating a new project with the ng new command, I've excluded the comments in the file for brevity.

import 'zone.js/dist/zone'; // Included with Angular CLI.

[Back to top](https://jasonwatmore.com/#project-structure)

**Global LESS/CSS Styles**

**Path: /src/styles.less**

The global styles file contains LESS/CSS styles that are applied globally throughout the CRUD application.

/\* You can add global styles to this file, and also import other style files \*/

a { cursor: pointer }

.app-container {

min-height: 320px;

overflow: hidden;

}

.btn-delete-user {

width: 40px;

text-align: center;

box-sizing: content-box;

}

[Back to top](https://jasonwatmore.com/#project-structure)

**npm package.json**

**Path: /package.json**

The package.json file contains project configuration information including package dependencies that get installed when you run npm install and scripts that are executed when you run npm start or npm run build etc. Full documentation is available at <https://docs.npmjs.com/files/package.json>.

{

"name": "angular-crud-example",

"version": "0.0.0",

"scripts": {

"ng": "ng",

"start": "ng serve --open",

"build": "ng build",

"test": "ng test",

"lint": "ng lint",

"e2e": "ng e2e"

},

"private": true,

"dependencies": {

"@angular/animations": "~11.0.4",

"@angular/common": "~11.0.4",

"@angular/compiler": "~11.0.4",

"@angular/core": "~11.0.4",

"@angular/forms": "~11.0.4",

"@angular/platform-browser": "~11.0.4",

"@angular/platform-browser-dynamic": "~11.0.4",

"@angular/router": "~11.0.4",

"rxjs": "~6.6.0",

"tslib": "^2.0.0",

"zone.js": "~0.10.2"

},

"devDependencies": {

"@angular-devkit/build-angular": "~0.1100.4",

"@angular/cli": "~11.0.4",

"@angular/compiler-cli": "~11.0.4",

"@types/jasmine": "~3.6.0",

"@types/node": "^12.11.1",

"codelyzer": "^6.0.0",

"jasmine-core": "~3.6.0",

"jasmine-spec-reporter": "~5.0.0",

"karma": "~5.1.0",

"karma-chrome-launcher": "~3.1.0",

"karma-coverage": "~2.0.3",

"karma-jasmine": "~4.0.0",

"karma-jasmine-html-reporter": "^1.5.0",

"protractor": "~7.0.0",

"ts-node": "~8.3.0",

"tslint": "~6.1.0",

"typescript": "~4.0.2"

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

**TypeScript tsconfig.json**

**Path: /tsconfig.json**

The tsconfig.json file contains the TypeScript compiler configuration for all projects in the Angular workspace, it configures how the TypeScript code will be compiled / transpiled into JavaScript that is understood by the browser. For more info see <https://angular.io/guide/typescript-configuration>.

Most of the file is unchanged from when it was generated by the Angular CLI, only the paths property has been added to map the @app and @environments aliases to the /src/app and /src/environments directories. This allows imports to be relative to the app and environments folders by prefixing import paths with aliases instead of having to use long relative paths (e.g. import MyComponent from '@app/MyComponent' instead of import MyComponent from '../../../MyComponent').

/\* To learn more about this file see: https://angular.io/config/tsconfig. \*/

{

"compileOnSave": false,

"compilerOptions": {

"baseUrl": "./",

"outDir": "./dist/out-tsc",

"forceConsistentCasingInFileNames": true,

"strict": true,

"noImplicitReturns": true,

"noFallthroughCasesInSwitch": true,

"sourceMap": true,

"declaration": false,

"downlevelIteration": true,

"experimentalDecorators": true,

"moduleResolution": "node",

"importHelpers": true,

"target": "es2015",

"module": "es2020",

"lib": [

"es2018",

"dom"

],

"paths": {

"@app/\*": ["src/app/\*"],

"@environments/\*": ["src/environments/\*"]

}

},

"angularCompilerOptions": {

"strictInjectionParameters": true,

"strictInputAccessModifiers": true,

"strictTemplates": true

}

}

[Back to top](https://jasonwatmore.com/#project-structure)

I'm currently attempting to travel around Australia by motorcycle with my wife Tina on a pair of Royal Enfield Himalayans. You can follow our adventures on YouTube, Instagram and Facebook.