Trade Me

Angular Master Class
Training Book

Published with GitBook

Table of Contents

Trade Me's Angular Master Class Training Book

Over the last three years, Trade Me has provided Angular Master Class training to all its developers. This is an intensive deep-dive into a full-spectrum of the Angular platform.

This course is essential for developers to quickly become productive with Angular development.

About this AMC Coding Labs Book

This book contains all the lab exercises documents needed for the hands-on, coding labs.

Modules

Each module of the **Angular Master Class** course has 1 or more lab exercises. Each lab exercise adds features and enhancements to the previous lab exercise.

> These exercise descriptions are used by **students** as a guide for each exercise.

Shown below are quick-links and ordering of the courseware Modules:

- Module 1: **Jump-Start**
- Module 2: **RxJS**
- Module 3: **Forms**
- Module 4: **Routing**
- Module 5: **NgRx**
- Module 6: **Architecture**

For each lab exercise, simply open there `exercise-<xxx>-<xxx>` markdown associated with that exercise.

Web and App Servers

To launch your web application, use a Terminal session with the command:

```
$ ng serve
```

This starts a web server for the Angular 7.x application; open with a browser url `http://localhost:4200` .

And since your Angular application may request external, remote data [from `http://localhost:4201/api`], you will need a local app server to respond to the REST API calls. We have already configured a server as part of this repository.

Simply start a second, separate Terminal session with the commend:

```
$ npm run rest-api
```

Important Message

Remember: never be afraid to ask questions! The **Trade Me** trainers love questions.

> Often your question - and the answers - are very important to other students in the class.

And finally, help each other. Helping other students will reinforce the ideas that you are learning in this lesson... Have fun and good luck!

Thomas, Kenese, Elwyn, Craig, Michael, George, & Gareth

Jump Start

To explore introductory Angular concepts, let's create a Contacts application.

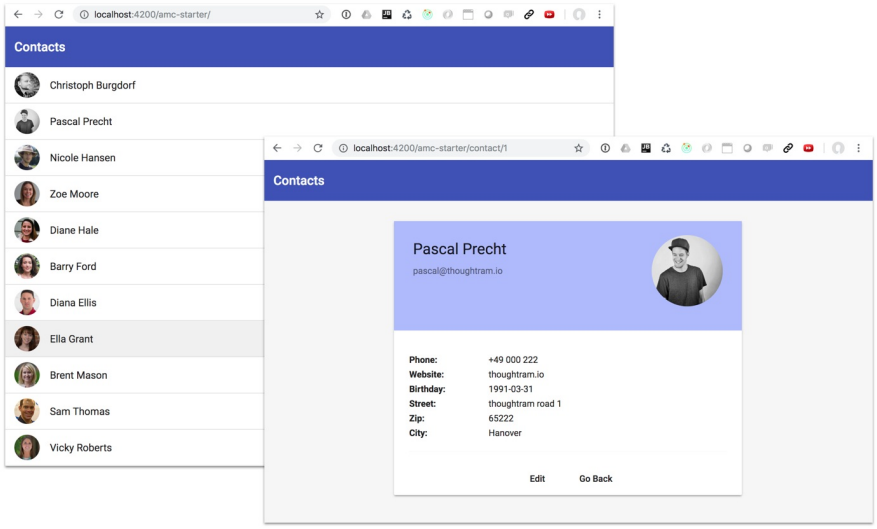


Figure: jrdme

Lab Exercises

- Lab #1: Use a **directives**
- Lab #2: Display a Contact
- Lab #3: Display a Contact List
- Lab #4: Use a ContactService
- Lab #5: Route to Contacts List
- Lab #6: Use Route Params
- Lab #7: Fetch Data w/ HTTP
- Lab #8: Send Data w/ HTTP
- Lab #7: Use Async Pipe & Observables

Exercise: Using Directives

We learned how to bootstrap an Angular component. Let's take it one step further and explore how we can use **other** components inside our `ContactsAppComponent`.

Scenario

Our `ContactsAppComponent` needs a nice header because, well... that's what fancy apps do, right?

The `ContactsModule` already imports the `ContactsMaterialModule`, which provides rich UI components that implement Google's Material Design specification. Let's use the `MatToolbar` component to create our header. `ContactsMaterialModule` takes care of making that component available to us. All we need to do is to use the component in our component's template.

Tasks

1. Change `ContactsAppComponent` 's template to use Angular Material's `<mat-toolbar>` component. Here's a snippet:

```
<mat-toolbar color="primary">Contacts</mat-toolbar>
```

Additional resources and help

- Material Toolbar Component Docs

Next Lab

Go to JumpStart - Lab #2

Exercise: Display first contact

Time to display our first contact! We know that component properties can be used as expressions in a component's template to display their value. In this exercise we're going to learn how to display data structures from a component, using expressions and interpolations.

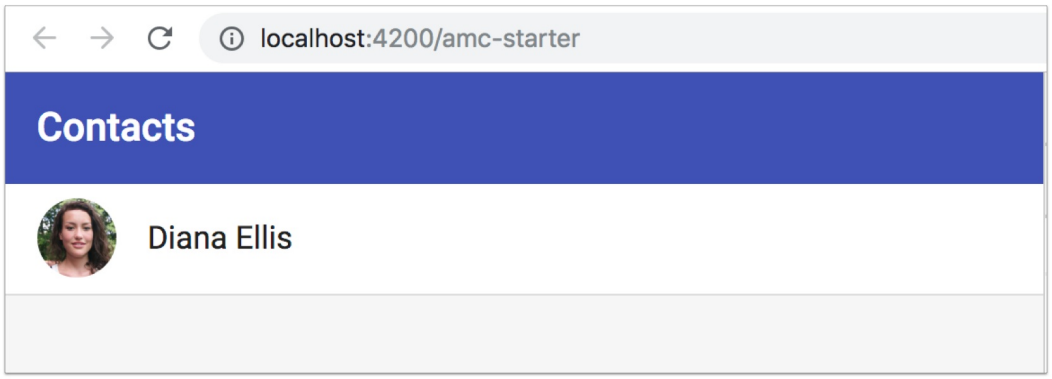


Figure: j3

Scenario

"Hello World" is not enough. We want to build a contacts app, so eventually we're going to display a list of contacts. But let's do it one step at a time. In `src/app/models/contact.ts` you can find an interface definition of `Contact`.

Import this type and create a `contact` property of this type in `ContactsAppComponent` to render the data of a first contact in the component's template (you can find the data in the task list).

Tasks

1. Import `Contact` from `src/app/models/contact.ts`.
2. Create a `contact` property of type `Contact` with the following object:

```
{
 id: 6,
 name: 'Diana Ellis',
 email: '',
 phone: '',
 birthday: '',
 website: '',
 image: 'assets/images/6.jpg',
 address: {
   street: '6554 park lane',
   zip: '43378',
   city: 'Rush',
   country: 'United States'
 }
}
```

3. Render the `contact` information using expressions and interpolations with the following template, **right below the toolbar** (replace the "Hello World" part with this):

```
<mat-list>
 <mat-list-item>
   <img mat-list-avatar [src]="INSERT_IMAGE_EXPRESSION HERE" alt="Picture of INSERT_CONTACT_NAME">
   <h3 mat-line>INSERT_CONTACT_NAME</h3>
 </mat-list-item>
</mat-list>
```

You're probably noticing the brackets in `<img [src]="...">`. Don't be scared! If you're curious, feel free to checkout the resources for more information. We're going to talk about them later.

Code Snippets

`contact.ts`

```
interface Address {
  street: string;
  zip: string;
  city: string;
  country: string;
}

export interface Contact {
  id: number;
  name: string;
  email: string;
  phone: string;
  birthday: string;
  website: string;
  image: string;
  address: Address;
}
```

Additional resources and help

- Angular's template syntax demystified

Next Lab

Go to JumpStart - Lab #3

Exercise: Display list of contacts

The goal of this exercise is to explore and discuss how we create lists dynamically based on data collections that derived from a component's property. Let's create a list of contacts.

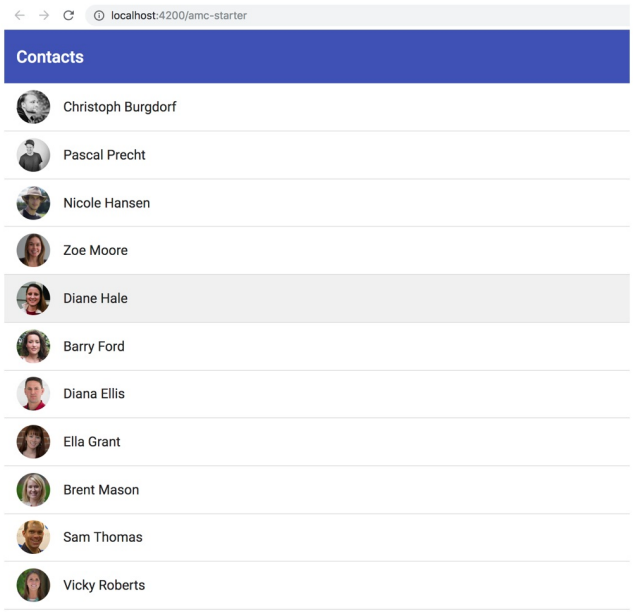


Figure: j3

Scenario

In order to render a list of contacts, we need data that we can actually work with. Take a look at `src/app/data/contact-data.ts`. What you'll find is a data collection of contacts.

Import that data, assign it a `contacts` property of `ContactsAppComponent` and use the `ngFor` directive to render a HTML list, based on that data, in `ContactsAppComponent`'s view.

Tasks

1. Import `CONTACT_DATA` from `data/contact-data.ts` in ContactsAppComponent.
2. Create a property `contacts` in `ContactsAppComponent` and assign the collection.
3. Extend the existing static list in `ContactsAppComponent`'s view with `ngFor` and render a list item for each contact in the collection.

> Do not copy and paste the above code!

Bonus Tasks

1. Figure out what `ngFor`'s `trackBy` property is about and use it to make `ngFor` track the list by the contact ids.

Code Snippets

`contact-data.ts`

```ts
export const CONTACT_DATA = [
  {
    id: 0,
    firstname: 'Christoph',
    lastname: 'Burgdorf',
    street: 'thoughtram road 1',
    zip: '65222',
    city: 'Hanover',
    image: '/images/0.jpg'
  },
  {
    id: 1,
    firstname: 'Pascal',
    lastname: 'Precht',
    street: 'thoughtram road 1',
    zip: '65222',
    city: 'Hanover',
    image: '/images/1.jpg'
  },
  ...
]
```

Figure: js30

`contacts-app.component.ts`

```ts
import { Component } from '@angular/core';
import { Contact } from './models/contact';
import { CONTACT_DATA } from './data/contact-data';

@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar color="primary">Contacts</mat-toolbar>
    <mat-list>
      <mat-list-item *ngFor="">
        <img mat-list-avatar [src]="contact.image" alt="Picture of {{contact.name}}" class="circle">
        <h3 mat-line>{{contact.name}}</h3>
      </mat-list-item>
    </mat-list>
  `,
  styleUrls: ['./app.component.scss']
})
export class ContactsAppComponent {
  contacts: Array<Contact> = CONTACT_DATA;
}
```

Figure: js3 1

Additional resources and help

- See the Angular API: NgFor

Next Lab

Go to JumpStart - Lab #4

Exercise: Building a service and using DI

A static list of data is just okay to try some things out or for demo purposes. However, in a real world app, we surely don't want to rely on static data, but rather have a service that takes care of providing the data we need, no matter where it comes from.

In this exercise we're going to learn how to create services and how to inject them into our application using providers.

Scenario

`CONTACT_DATA` is an implementation detail we don't want to rely on in `ContactsAppComponent`. Create a service called `ContactsService` using angular-cli by running:

```
$ ng generate service contacts
```

Or the shorter version:

```
$ ng g s contacts
```

This will create a file in `src/app/contacts.service.ts`. Implement a method `getContacts()` that simply returns the collection of `CONTACT_DATA` for us.

Use that new service by adding a provider to the `providers` property of the module in `app.module.ts` and injecting the service in the constructor.

Note: our app should render the same, it's just the internal architecture that has changed.

Tasks

1. Create a `ContactsService` using angular-cli
2. Import `CONTACT_DATA`
3. Create a method `getContacts()` which returns the given data
4. Add `ContactsService` provider to the `providers` property of the module in `app.module.ts` to make the service available in your app
5. Inject `ContactsService` in `ContactsAppComponent`
6. Import `OnInit`
7. Make `ContactsAppComponent` implement `OnInit`
8. Add an `ngOnInit()` method to `ContactsAppComponent`
9. Call `ContactsService#getContacts()` to retrieve the contacts data

Code Snippets

```
import { Injectable } from '@angular/core';
import { CONTACT_DATA } from './data/contact-data';

@Injectable()
export class ContactsService {

  constructor() {}

  getContacts () {
    return CONTACT_DATA;
  }
}
```

Figure: j42

```
export class ContactsAppComponent implements OnInit {
  contacts: Array<Contact>;

  constructor(private contactsService: ContactsService) {}

  ngOnInit () {
    this.contacts = this.contactsService.getContacts();
  }
```

Figure: j41

> Bonus: Is the `ngOnInit()` required. How could you make this code more succinct?

Additional resources and help

- Dependency Injection in Angular
- Dependency Injection (angular.io)

Next Lab

Go to JumpStart - Lab #5

Exercise: Route to your first component

The time has come! Let's implement some routing features. In this exercise we're going to route to components.

Scenario

In order to route from our contacts list to a contact detail page, the first thing we need to do is to extract the current contacts list implementation into its own component, so that `ContactsAppComponent` is just a "frame" that holds our app, consisting of multiple components, together.

We can create a new component `ContactsListComponent` using angular-cli by running:

```
$ ng generate component contacts-list --project=amc-starter -m=app
```

Or the shorter version:

```
$ ng g c contacts-list -m=app
```

Using the `-m`, we specify the module this component will belong to. This will make sure the component is added to the `declarations` section of this module and you don't have to do this yourself.

Note that in most scenario's, this step is unnecesary. The angular-cli will try to find the closest module to add this component to. Since in this case it will find two modules (contacts-material and app) at the same level in the directory hierarchy, we do need to specify it.

> Bonus: which command line argument do you use to run the command in test/dry mode?

Tasks

1. Create a component `ContactsListComponent` with the **Angular CLI** and move the current contact list logic there from `ContactsAppComponent`
2. Create `src/app/app.routes.ts`, import the `ContactsListComponent` and export an array with routes (e.g. `export const APP_ROUTES = [...]`), holding one route pointing to `ContactsListComponent`.
3. In `app.module.ts` import `RouterModule` from `@angular/router` and `APP_ROUTES` from `./app/app.routes`
4. Add `RouterModule.forRoot(APP_ROUTES)` to the `imports` array of the module
5. Add `<router-outlet>` in `ContactsAppComponent`'s view to load and render components

Bonus Tasks

1. Figure out how to create a route that redirects to `/` when none of the configured routes are matched by the router.

Code Snippets

`contacts-list.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Contact } from '../models/contact';
import { ContactsService } from '../contacts.service';

@Component({
  selector: 'trm-contacts-list',
  template: `
    <mat-list>
      <mat-list-item *ngFor="let contact of contacts; trackBy:trackByContactId">
        <img mat-list-avatar
             [src]="contact.image"
             alt="Picture of {{contact.name}}"
             class="circle">
        <h3 mat-line>{{contact.name}}</h3>
      </mat-list-item>
    </mat-list>
  `,
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {

  contacts: Array<Contact>;

  constructor(private contactsService: ContactsService) {}

  ngOnInit () {
    this.contacts = this.contactsService.getContacts();
  }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Figure: j51

`app.routes.ts`

```
import { Routes } from '@angular/router';
import { ContactsListComponent } from './contacts-list/contacts-list.component';


export const APP_ROUTES: Routes = [
  { path: '', component: ContactsListComponent }
];
```

Figure: j52

Additional resources and help

- Routing in Angular revisited

- Router Guide
- RouterOutlet

Next Lab

Go to JumpStart - Lab #6

Exercise: Accessing Route Parameters

Next up, we want to route to a contact detail page. In this exercise we're going to create a new component that is able to retrieve routing parameters.

Scenario

The contacts app should be extended with a contact detail page. We need several things to get there:

- a new route configuration
- a new method to access a single contact in our service
- a new detail component
- we need to be able to link to that new component and pass some parameters

Tasks

1. Create a new method `ContactsService::getContact(id: string)` which takes an id and returns a contact by that id. (Hint: you can use `Array#find(fn)`
   > e.g. `this.contacts.find(contact => contact.id.toString() === id);`)
2. Create a new component `ContactsDetailComponent`, using the tools you already know, with the following template:
3. Add a new route to `APPROUTES` with the path `contacts/:id` that points to `ContactsDetailComponent`
4. Change the `<mat-list-item>` in `ContactsListComponent`'s view to `<a mat-list-item>` and use the `RouterLink` directive with a DSL configuration that points to `ContactsDetailComponent`
5. Import and inject `ActivatedRoute` and `ContactsService` in `ContactsDetailComponent`
6. Create a `contact` property in `ContactsDetailComponent` and use `ActivatedRoute` and `ContactsService` to retrieve the requested contact
7. Render correct `contact` properties in `ContactsDetailComponent`'s view
8. Add `RouterLink` to get back to `ContactsListComponent`

Code Snippets

`contacts.service.ts`

```
import { Injectable } from '@angular/core';
import { CONTACT_DATA } from './data/contact-data';

@Injectable()
export class ContactsService {
  get contacts() {
    return [...CONTACT_DATA];
  }

  getContact (id: string) {
    return this.getContacts()
      .find(contact => String(contact.id) === id);
  }
}
```

`contact-details.component.html`

```
<div class="trm-contacts-detail">
  <mat-card>
    <mat-card-title-group class="fullBleed">
      <img mat-card-md-image [src]="INSERT_CONTACT_IMAGE">
      <mat-card-title>INSERT_CONTACT_NAME</mat-card-title>
      <mat-card-subtitle>INSERT_CONTACT_EMAIL</mat-card-subtitle>
    </mat-card-title-group>
    <mat-card-content>
      <dl>
        <dt>Phone:</dt>
        <dd>INSERT_CONTACT_PHONE</dd>
        <dt>Website:</dt>
        <dd>INSERT_CONTACT_WEBSITE</dd>
        <dt>Birthday:</dt>
        <dd>INSERT_CONTACT_BIRTHDAY</dd>
        <dt>Street:</dt>
        <dd>INSERT_CONTACT_STREET</dd>
        <dt>Zip:</dt>
        <dd>INSERT_CONTACT_ZIP</dd>
        <dt>City:</dt>
        <dd>INSERT_CONTACT_CITY</dd>
      </dl>
    </mat-card-content>
    <mat-card-actions fxLayout fxLayoutAlign="center center">
      <a mat-button title="Go back to list">Go Back</a>
    </mat-card-actions>
  </mat-card>
</div>
```

`contact-details.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { ContactsService } from '../contacts.service';
import { Contact } from '../models/contact';

@Component({
  selector    : 'trm-contacts-detail',
  templateUrl : './contacts-detail.component.html',
  styleUrls   : ['./contacts-detail.component.css']
})
export class ContactsDetailComponent {
  contact: Contact = this.contactsService
    .getContact(this.route.snapshot.paramMap.get('id'));

  constructor(
    private contactsService: ContactsService,
    private route: ActivatedRoute) {
  }
}
```

Additional resources and help

- See Angular Router API: ActivatedRoute

Next Lab

Go to JumpStart - Lab #7

Exercise: Fetching data using Http

In this exercise we're going to replace our static `CONTACT_DATA` with actual http requests to fetch the data from a remote web server.

Scenario

Until now we've worked with static (mock) data provided by collection via our `ContactsService`. We now want to take it one step further and fetch this data from a server using Angular's `Http` layer.

In order to make http work, we need to configure our application's dependency injector and teach it some http capabilities. Just like for the router, Angular provides a module `HttpClientModule` that we have to import from `@angular/common/http`.

We also need to import operators for the Reactive Extensions. In this exercise we're mainly interested in the `map()` function which we can import from `rxjs/operators`. We will use these new tools to extend `ContactsService` to use `Http` and fetch the contacts data from `http://localhost:4201/api`.

Tasks

1. Run `npm run rest-api` from another terminal session to start the REST API server
2. Import `HttpClientModule` from `@angular/common/http` and add it to the `imports` array of our module
3. Import `map` from `rxjs/operators` in `src/app/contacts.service.ts`
4. Import `HttpClient` and inject it into `ContactsService`
5. Create a property `API_ENDPOINT` with the value of `http://localhost:4201` in `ContactsService`
6. Change `getContacts()` so that it uses `Http.get()` to fetch contacts data from `http://localhost:4201/api/contacts`.
7. Do the same for `getContact()`. Keep in mind that this API expects a parameter (`http://localhost:4201/api/contacts/${id}`)
8. Change `ContactsService` usage in `ContactsDetailComponent` and `ContactsListComponent` since it's now based on observables
9. Fix rendering in `ContactsDetailComponent`'s view using safe navigation operator (`contact?.[PROPERTY]`)

Bonus Tasks

1. You might notice that the image in `ContactsDetailComponent` cause `404` errors. Can you find a solution for it?
2. Currently the api endpoint is hard-coded right into `ContactsService`. Can you find a way to make it injectable throughout your application? Give it a try!

   (hint: we need **string token** e.g. "API_ENDPOINT" and a new decorator called `@Inject`)

3. Because "API_ENDPOINT" is a very generic name, we want to make sure to not run into naming collisions. Find out how you can use `InjectionToken` and refactor the previous implementation.

Code Snippets

`contacts.service.ts`

```
interface ContactResponse  { item  : Contact    }
interface ContactsResponse { items : Contact[]  }

@Injectable()
export class ContactsService {
  private API_ENDPOINT = 'http://localhost:4201/api';

  constructor(private http: HttpClient) {}

  getContact(id: string): Observable<Contact> {
    return this.http
      .get<ContactResponse>(`${this.API_ENDPOINT}/contacts/${id}`)
      .pipe(map(data => data.item));
  }

  getContacts(): Observable<Array<Contact>> {
    return this.http
      .get<ContactsResponse>(`${this.API_ENDPOINT}/contacts`)
      .pipe(map(data => data.items));
  }

}
```

Figure: j7 1

`contacts-detail.component.ts`

```
@Component({
  selector: 'trm-contacts-detail',
  templateUrl: './contacts-detail.component.html',
  styleUrls: ['./contacts-detail.component.css']
})
export class ContactsDetailComponent implements OnInit {
  contact: Contact;

  constructor(
    private contactsService: ContactsService,
    private route: ActivatedRoute) {
}

  ngOnInit() {
    this.contactsService
        .getContact(this.route.snapshot.paramMap.get('id'))
        .subscribe(contact => this.contact = contact);
  }
}
```

Figure: j7 2

Additional resources and help

- Taking advantage of Observables in Angular
- Taking advantage of Observables in Angular - Part 2
- Understanding OpaqueToken in Angular

Next Lab

Go to JumpStart - Lab #8

Exercise: Two-way data binding and sending data

Combining property binding with event binding results in two-way data binding. This exercise demonstrates how `ngModel` and two-way data binding can be used to create our first simple form.

In the last exercise we've extended our `ContactsAppComponent` to fetch its data from a remote server. Now - in this exercise - we're going to explore how to SEND data using http.

Scenario

Displaying details of a contact is nice, but we also want to be able to edit them. We want to take advantage of Angular's `ngModel` directive that implements two-way data binding to build our first form.

This form displays a `contact` but allows for editing that contact at the same time. We have to add http capabilities to our `ContactsEditorComponent`, since it has to **send** data when we update a contact.

Tasks

1. Create a new component `ContactsEditorComponent` using the tools you already know with the following template:
2. Import `FormsModule` from `@angular/forms` and add it to the `imports` array of the application module
3. Add a new route to `APP_ROUTES` that takes `/contact/:id/edit` that points to `ContactsEditorComponent`
4. Add a button with `RouterLink` to `ContactsDetailComponent` 's view that routes to `ContactsEditorComponent` and passes `contact.id` as `id` route parameter
5. Implement a `ContactsService` method `updateContact(contact: Contact)`, that uses `Http.put(url, data)` to send the contact that has to be updated over the wire.
    - The endpoint is `localhost:4201/api/contacts/${id}`
    - Make sure to use `PUT` method
6. In ContactsEditorComponent, inject `ContactsService`, `ActivatedRoute` and `Router` into `ContactsEditorComponent`
7. Create a `contact` property and Use `ContactsService#getContact()` in `ngOnInit()` to fetch the contact object.
8. Make sure to initialize `contact: Contact = <Contact>{ address: {}};` in `ContactsEditorComponent` since the safe navigation operator can't be used with `ngModel`
9. Render `contact` properties using `ngModel` in `ContactsEditorComponent` 's view
10. Implement a method `cancel(contact: Contact)` in `ContactsEditorComponent` that navigates to `ContactsDetailComponent` of the given contact using `Router#navigate()` (takes a routerLink dsl configuration)
11. Implement a method `save(contact: Contact)` in `ContactsEditorComponent` that uses `ContactsService#updateContact` and navigates back to `ContactsDetailComponent` as soon as the `PUT` request was successful.

Code Snippets

`contacts.service.ts`

```
interface ContactResponse  { item  : Contact    }


@Injectable()
export class ContactsService {

  constructor(private http: HttpClient, @Inject(API_ENDPOINT) private apiEndpoint) {}

  updateContact(contact: Contact): Observable<Contact> {
    const url = `${this.apiEndpoint}/contacts/${contact.id}`;
    return this.http.put<ContactResponse>(url, contact)
        .pipe(map(data => data.item));
  }

}
```

Figure: j8 1

`contacts-editor.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { Contact } from '../models/contact';
import { ContactsService } from '../contacts.service';

@Component({
  selector: 'trm-contacts-editor',
  templateUrl: './contacts-editor.component.html',
  styleUrls: ['./contacts-editor.component.css']
})
export class ContactsEditorComponent implements OnInit {

  cancel(contact: Contact) {
    this.goToDetails(contact);
  }

  save(contact: Contact) {
   this.contactsService
       .updateContact(contact)
       .subscribe(() => this.goToDetails(contact));
  }

  private goToDetails(contact: Contact) {
    this.router.navigate(['/contact', contact.id ]);
  }
}
```

Figure: j8 2

`contacts-editor.component.html`

```
<div class="trm-contacts-editor">
```

```
    <mat-card>
      <mat-card-title-group class="fullBleed editing">
        <img mat-card-md-image [src]="INSERT_CONTACT_IMAGE">
        <mat-card-title>INSERT_CONTACT_NAME</mat-card-title>
        <mat-card-subtitle>INSERT_CONTACT_EMAIL</mat-card-subtitle>
      </mat-card-title-group>
      <mat-card-content>
        <div fxLayout="column">
          <mat-form-field fxFlex>
            <input matInput placeholder="Name" name="name">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Email" name="email">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Phone" name="phone">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Website" name="website">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Birthday" name="birthday" type="date">
          </mat-form-field>
          <fieldset fxLayout="column">
            <legend>Address</legend>
            <mat-form-field fxFlex>
              <input matInput placeholder="Street" name="street">
            </mat-form-field>
            <mat-form-field fxFlex>
              <input matInput placeholder="Zip" name="zip">
            </mat-form-field>
            <mat-form-field fxFlex>
              <input matInput placeholder="City" name="city">
            </mat-form-field>
          </fieldset>
        </div>
      </mat-card-content>
      <mat-card-actions fxLayout fxLayoutAlign="center center">
        <button mat-button (click)="save(contact)" title="Save contact">Save</button>
        <button mat-button (click)="cancel(contact)" title="Cancel editing">Cancel</button>
      </mat-card-actions>
    </mat-card>
  </div>
```

Additional resources and help

- Template-driven Forms in Angular 2
- ngModel
- Router#navigate()
- Angular 2 Developer Guide: Server Communication
- Taking advantage of Observables in Angular 2
- Taking advantage of Observables in Angular 2 - Part 2

Next Lab

Go to JumpStart - Lab #9

Exercise: Async Pipe

We can fine-tune our `ContactsAppComponent` a little bit further before we reached the end of this tutorial by using `AsyncPipe` in combination with our observables data structures.

Scenario

Observable data structures can be passed directly from a component into its view and resolve by `AsyncPipe`. This makes the code a more readable since we don't have the imperative logic of resolving/subscribing to an observable. `AsyncPipe` does the job for us.

Tasks

1. In `ContactsListComponent`, Import `Observable` from `rxjs/Observable`.
2. Rename the `contacts` property to `contacts$` and annotate as type `Observable<Array<Contact>>`
3. Remove subscription of `contactsService.getContacts()`
4. Add `AsyncPipe` to `ngFor` expression in `ContactsListComponent`'s view

Code Snippets

`contacts-list.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { Contact } from '../models/contact';
import { ContactsService } from '../contacts.service';

@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent {
  contacts$: Observable<Array<Contact>> = this.contactsService.getContacts();

  constructor(private contactsService: ContactsService) {}

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Figure: j9 1

Additional resources and help

- Pipes - AsyncPipe

- AsyncPipe

Observables

Let's explore Observables and Angular:

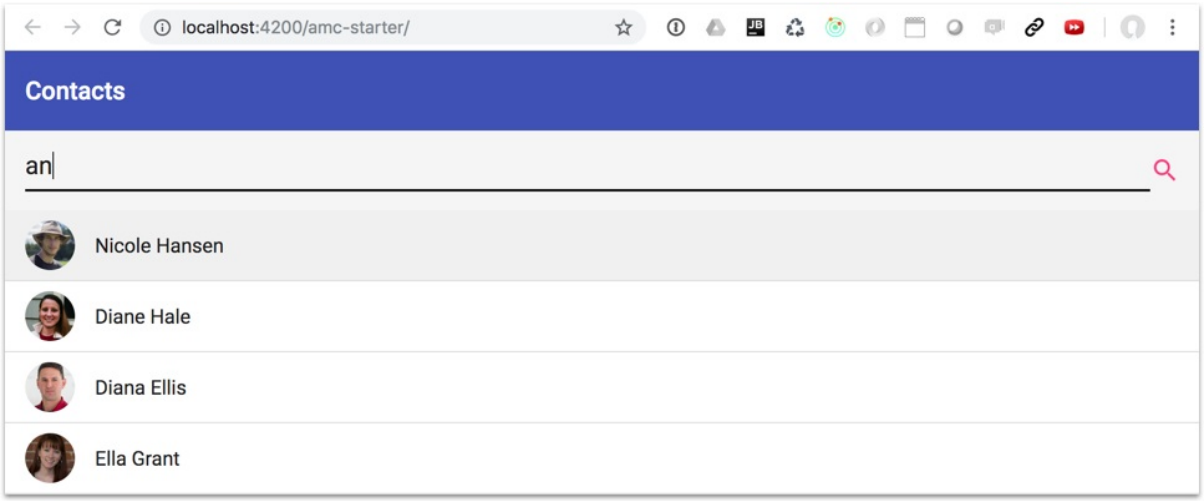


Figure: ordme

Lab Exercises

- Lab #1: Basic Search
- Lab #2: Debounce & DistinctUntilChanged
- Lab #3: Out-of-Order Responses
- Lab #4: Refactor to Service

Lab 1: Write a basic search implementation

The goal of this exercise is to write a first basic implementation of an instant search.

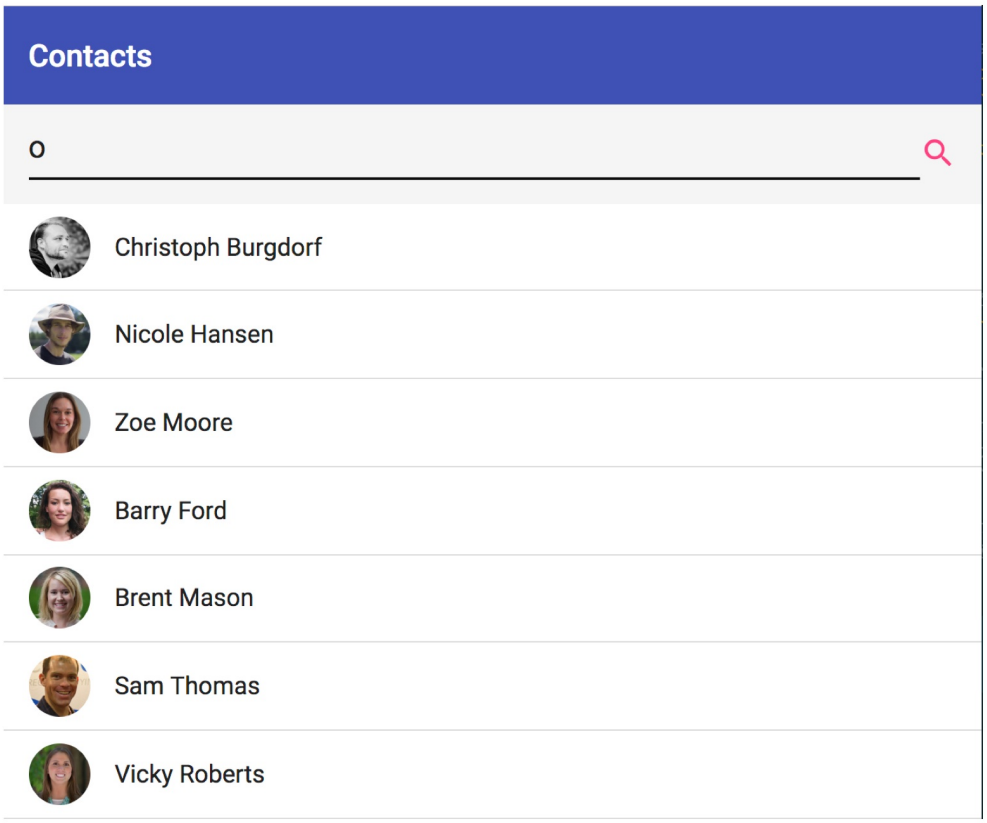


Figure: o1

Scenario

In `src/app/contacts-list/contacts-list.component.html` you already have the proper markup to render a list of contacts. The goal is to reuse the existing markup and add a search field on top of the list to implement the search feature.

Tasks

1. Add a `ContactsService::search(term: string)` method to use the REST endpoint `http://localhost:4201/api/search?text=${term}` ; where the `${term}` is the value of the **search** `term` argument.
2. Implement a `search()` method on `ContactsListComponent` which uses `ContactsService#search()`

   > Note: Change the ContactsListComponent variable to `contacts$` since the `search()` will return an `Observable<Array<Contact>>` .

3. Add a search field to `ContactsListComponent` 's template... on top of the contacts list. Here's a snippet:

```
<mat-toolbar>
 <mat-form-field color="accent" class="trm-search-container">
```

```
    <input matInput type="text" (input)="search($event.target.value)">
  </mat-form-field>
  <mat-icon color="accent">search</mat-icon>
 </mat-toolbar>
```

4. When the input control fires an `input` event, invoke the component's search logic; access and pass the input's value to the event handler.

Code Snippets

`contacts.service.ts`

```
@Injectable()
export class ContactsService {
  constructor(private http: HttpClient, @Inject(API_ENDPOINT) private apiEndpoint) {}

  ...

  search(term: string) : Observable<Array<Contact>> {
      const searchUrl = `${this.apiEndpoint}/search?text=${term}`;

    return this.http
        .get<ContactsResponse>(searchUrl)
        .pipe(map(data => data.items));
  }

}
```

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>>;

  ...

  search(term: string) {
    this.contacts$ = this.contactsService.search(term);
  }
}
```

Web and App Servers

To launch your web application and the REST server, use a Terminal session with the command:

```
$   ./start
```

Next Lab

Go to Observables Lab #2: Debounce

Lab 2: Debounce and deduplicate terms

The goal of this exercise is to fine-tune the existing implementation of our instant search to leverage the power of observables. We will debounce and deduplicate the user supplied search terms in order to reduce the pressure on the server and increase scalability.

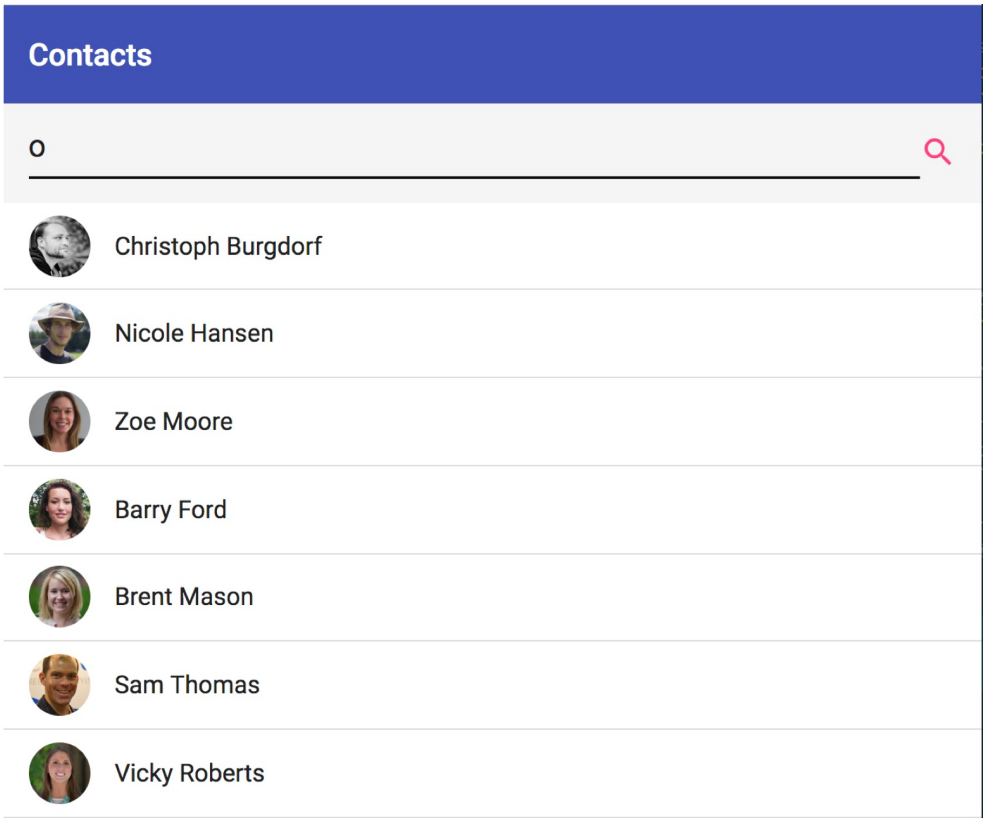


Figure: o1

Scenario

In `src/app/contacts-list/contacts-list.component.ts` create a `Subject` to be used from within the template to push the search term as it changes. We need to have this term changes as an Observable so that we can use the power of the `debounce` and `distinctUntilChanged` operator to reduce the number of requests made to the server.

In `src/app/contacts-list/contacts-list.component.html` change the way the input changes are propagated so that they get pumped into the Subject that our `ContactsListComponent` now holds on to.

Tasks

1. Create a `Subject` in `ContactsListComponent` that can be used from the template to propagate the input changes
   - Update the `(input)` event handler of the search input to call `terms$.next($event.target.value)`
   - import `Subject` from `rxjs`
   - create an instance property `terms$` and initialize it with `new Subject<string>();`

> Make sure to import `debounceTime` and `distinctUntilChanged` just like you did for `map`

2. Subscribe to `terms$` observable and perform `search(term)`
3. Use the `debounceTime(400)` and `distinctUntilChanged()` operator to reduce the number of requests made to the server

Code Snippets

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>>;
  private terms$ = new Subject<string>();

  constructor(private contactsService: ContactsService) {}

  ngOnInit () {
    this.contacts$ = this.contactsService.getContacts();

    this.terms$.pipe(
      debounceTime(400),
      distinctUntilChanged()
    ).subscribe(term => this.search(term));
  }

  ...
```

`constacts-list.component.html`

```
<mat-toolbar>
  <mat-form-field class="trm-search-container">
    <input matInput (input)="terms$.next($event.target.value)" type="text">
  </mat-form-field>
  <mat-icon color="accent">search</mat-icon>
</mat-toolbar>
<mat-list>
  <a mat-list-item
    [routerLink]="['/contact', contact.id]"
    *ngFor="let contact of contacts$ | async; trackBy:trackByContactId"
    title="View {{contact.name}} details">

    <img mat-list-avatar [src]="contact.image"
         alt="Picture of {{contact.name}}" class="circle">
    <h3 mat-line>{{contact.name}}</h3>
  </a>
</mat-list>
```

Next Lab

Go to Observables Lab #3: Out of Order Responses

Lab 3: Deal with out of order responses

Our instant search works pretty decent already but there's one thing that we haven't dealt with yet. Namely, out of order responses.

Consider the following:

The user enters *Christoph*, pauses, clears the search box, and enters *Pascal*. The application issues two search requests, one for *Christoph* and one for *Pascal*.

A load balancer could dispatch the requests to two different servers with different response times. This means, we can't be sure which of the requests will come back first or second. The results from the first *Christoph* request might arrive after the later *Pascal* results. The user will be confused if we display the *Christoph* results to the *Pascal* query.

In order to fix that problem we have to combine the Observable of terms with the Observable of API response using the `switchMap` operator. The `switchMap` will take care of unsubscribing from Observable API responses as soon as new requests are issued. By combining the input stream (what the user types) with the output stream (what the server responds) through the `switchMap` operator we get a new, more sophisticated Observable that only yields the results that we are interested in.

Scenario

In `src/app/contacts-list/contacts-list.component.ts` in the `ngOnInit` method, combine the Observable of terms (`Observable<string>`) and the Observable of API responses (`Observable<Array<Contact>>`) by using the `switchMap` operator. Expose the new Observable to `this.contacts` . This will prevent out of order responses to mess with the UI.

There's one more catch though. So far we've used `this.contacts = this.contactsService.getContacts()` in `ngOnInit` to show the initial list of contacts. Clearly, if we just follow what we explained above we end up overwriting the value of `this.contacts` and end up with a list that's initially empty.

Fortunately we can simply use the `merge` static method to merge in `this.contactsService.getContacts()` to display the initial list of contacts.

Tasks

1. Import `switchMap` just as we did with the other operators
2. Import `merge` from `rxjs`
3. In `ngOnInit` use `switchMap` operator to combine the stream of terms with the `search` method producing an `Observable<Array<Contact>>`
4. Merge the stream with the `Observable` returned by `this.contactsService.getContacts()` for the initial state of the list
5. Remove the obsolete `search(term: string)` method from the component

Code Snippets

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>>;
  private terms$ = new Subject<string>();

  constructor(private contactsService: ContactsService) {}

  ngOnInit () {
    const contactsSearch$ = this.terms$.pipe(
      debounceTime(400),
      distinctUntilChanged(),
      switchMap(term => this.contactsService.search(term))
    );
    const allContacts$ = this.contactsService.getContacts();

    this.contacts$ = merge(contactsSearch$, allContacts$);
  }

}
```

Bonus Tasks

1. Simulate that the initial loading of contacts takes 5000ms (e.g. `contactsService.getContacts().pipe(delay(5000))`). This causes search results to get overwritten by the full list when performing a search quickly after the application starts. Find a way to get around this.
   > Hint: use the `takeUntil()` operator.

Next Lab

Go to Observables Lab #4: Refactor to Service

Bonus Exercise: Move search logic into service

We covered all important cases of our instant search. But wouldn't it be nice if we could just hide all these details of debouncing, deduplicating and out of order responses behind a service?

Why not make our service smarter so that create a search method that has the following signature:

```
search(terms: Observable<string>, debounceMs = 400) : Observable<Array<Contact>>
```

> Question: Why do this?
> The `search` method of our service takes a raw stream of terms and an optional number of ms for the debouncing and returns an `Observable<Array<Contact>>` ... hiding all the complexity of the task so that it doesn't bleed into our `ContactsListComponent`.

Scenario

In `src/app/contacts.service.ts` make a smarter `search` method that hides the complexity of debouncing, deduplicating and out of order responses from the component that uses the service.

Tasks

1. Rename the `search` method in the `ContactsService` to `rawSearch`
2. Move the operators needed in `ContactsService` over to `contacts.service.ts` from `ContactsListComponent`
3. Build a smarter `search` method on top of `rawSearch` that hides the complexity of debouncing, deduplicating and out of order responses from its user
4. Change the `ContactsListComponent` to use the new `ContactsService#search` API

Code Snippets

`contacts.service.ts`

```
@Injectable()
export class ContactsService {
  ...

  search(term: Observable<string>, debounceMs = 400) : Observable<Array<Contact>> {
    return term.pipe(
      debounceTime(debounceMs),
      distinctUntilChanged(),
      switchMap(term => this.rawSearch(<string>term))
    );
  }

  rawSearch(term: string) {
    return this.http.get<ContactsResponse>(`${this.apiEndpoint}/search?text=${term}`)
                    .pipe(map(data => data.items));
  }
}
```

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>>;
  private terms$ = new Subject<string>();

  constructor(private contactsService: ContactsService) {}

  ngOnInit () {
    this.contacts$ = merge(
      this.contactsService.search(this.terms$),
      this.contactsService.getContacts()
    );
  }

}
```

Forms

Let's explore Angular Forms: Template-driven & Reactive:

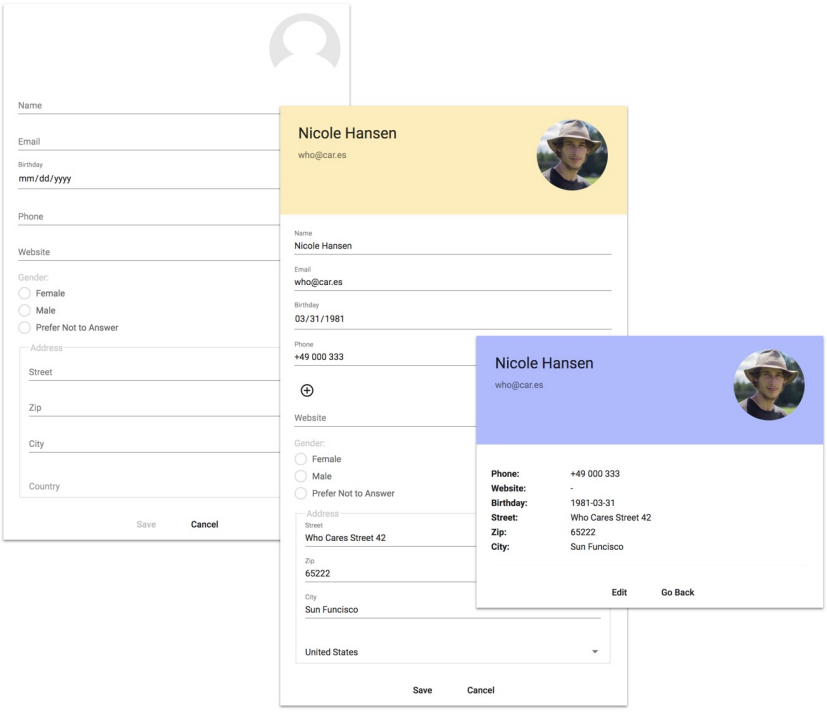


Figure: frdme

Lab Exercises

- Lab #1: Use a Template-Driven From
- Lab #2: Form Validators and Error Messages
- Lab #3: Custom Email Validator
- Lab #4: Custom Async Validator
- Lab #5: Use Reactive Forms
- Lab #6: Use Form Arrays for Dynamic Fields
- Lab #7: Custom Form Controls

Lab 1:Create template-driven form

In this exercise we're going to build a template-driven form to add a new contact to our contacts list using Angular form directives.

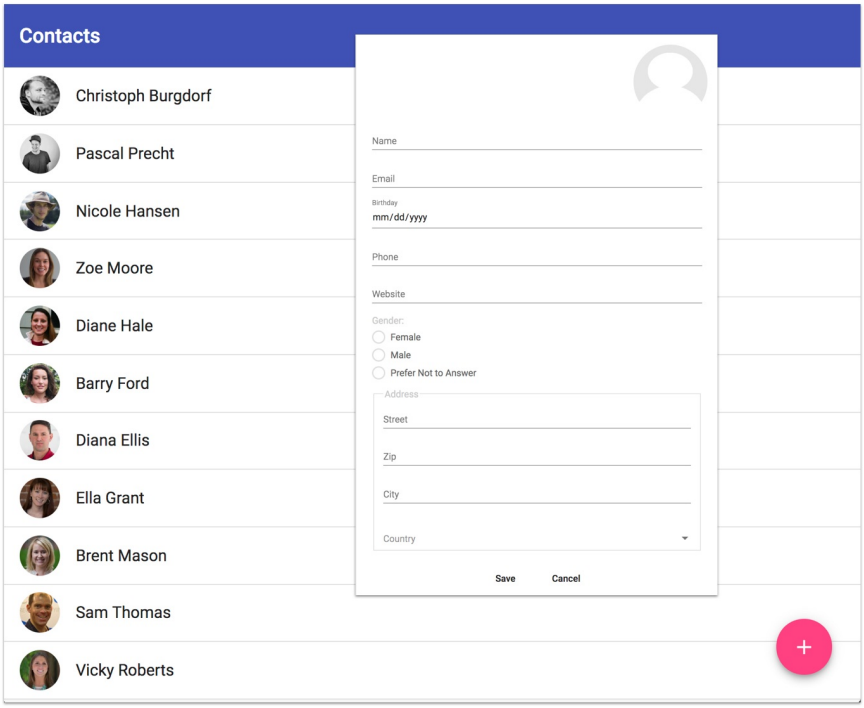


Figure: f1

Scenario

We can list and edit our contacts, but we don't have a way to add new ones. That's why we want to create a new form component that enables us to do that.

```
```console
$ ng generate component contacts-creator
```
or

```console
$ ng g c contacts-creator
```
```

Tasks

1. Create a new component **ContactsCreatorComponent** using angular-cli
2. Render a form to add a new contact using the following HTML:
3. Add a new route configuration that loads `ContactCreatorComponent` with the path `contact/new` (**keep in mind**

that it has to be defined before `contact/:id` route)

4. Append the following HTML to the `ContactsList` template, to create a button that links to `ContactCreatorComponent`:

```
<a routerLink="/contact/new" mat-fab title="Add a new contact" class="trm-floating-button">
 <mat-icon class="md-24">add</mat-icon>
</a>
```

5. Add a new method `addContact(contact: Contact)` to **ContactsService** that performs and http POST request to `http://localhost:4201/api/contacts` with a new contact data object that is passed to this method (you can use `updateContact()` as inspiration).
6. Add `ngModel` and `ngModelGroup` to **ContactCreatorComponent**s template accordingly to submit an Contact object structure on save.
7. Get an `ngForm` reference in the template and use it's value property to submit data on save (`(ngSubmit)="save(form.value)"`).
8. Add a new method `save(contact:Contact)` to **ContactCreatorComponent** that calls `ContactsService#addContact()` with the given contact, and navigates to **ContactsListComponent**, once the method returns.

> Note: if the `save(contact)` generates a server 404 response, ensure that the contact data submitted to the server is valid.

Code Snippets

`contacts-creator.component.html`

```
<div class="trm-contacts-creator">
  <form>
    <mat-card>
      <mat-card-title-group>
        <img mat-card-md-image alt="Placeholder image" src="/assets/images/placeholder.png">
        <mat-card-title></mat-card-title>
        <mat-card-subtitle></mat-card-subtitle>
      </mat-card-title-group>
      <mat-card-content>
        <div fxLayout="column">
          <mat-form-field fxFlex>
            <input matInput placeholder="Name" name="name">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Email" name="email">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Birthday" name="birthday" type="date">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Phone" name="phone">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Website" name="website">
          </mat-form-field>
          <mat-radio-group name="gender">
            <mat-radio-button>
              INSERT_GENDER_VARIANT
            </mat-radio-button>
          </mat-radio-group>
          <fieldset fxLayout="column">
            <legend>Address</legend>
            <mat-form-field fxFlex>
              <input matInput placeholder="Street" name="street">
```

```
            </mat-form-field>
            <mat-form-field fxFlex>
              <input matInput placeholder="Zip" name="zip">
            </mat-form-field>
            <mat-form-field fxFlex>
              <input matInput placeholder="City" name="city">
            </mat-form-field>
            <mat-select placeholder="Country" name="country">
              <mat-option>INSER_COUNTRY_NAME</mat-option>
            </mat-select>
          </fieldset>
        </div>
      </mat-card-content>
      <mat-card-actions fxLayout fxLayoutAlign="center center">
        <button mat-button type="submit">Save</button>
        <a mat-button title="Cancel creating new contact">Cancel</a>
      </mat-card-actions>
    </mat-card>
  </form>
</div>
```

`contact.ts`

```
export interface Contact {
  id: number | string;
  name?: string;
  email?: string;
  phone?: string | string[];
  birthday?: string;
  website?: string;
  image?: string;
  address?: Address;
}
```

`contacts-service.ts`

```
interface ContactResponse  { item  : Contact    }
interface ContactsResponse { items : Contact[]  }

@Injectable()
export class ContactsService {
  constructor(private http: HttpClient, @Inject(API_ENDPOINT) private apiEndpoint) {}

  addContact(contact: Contact): Observable<Contact> {
    return this.http.post<ContactResponse>(`${this.apiEndpoint}/contacts`, contact)
        .pipe(map(data => data.item));
  }
}
```

Additional resources and help

- Template-driven Forms in Angular

Next Lab

Go to Lab #2: Template Validators & Error Messages

Lab 2: Template validators and error messages

In this exercise we add validation and error messages to prevent adding contacts without a name.

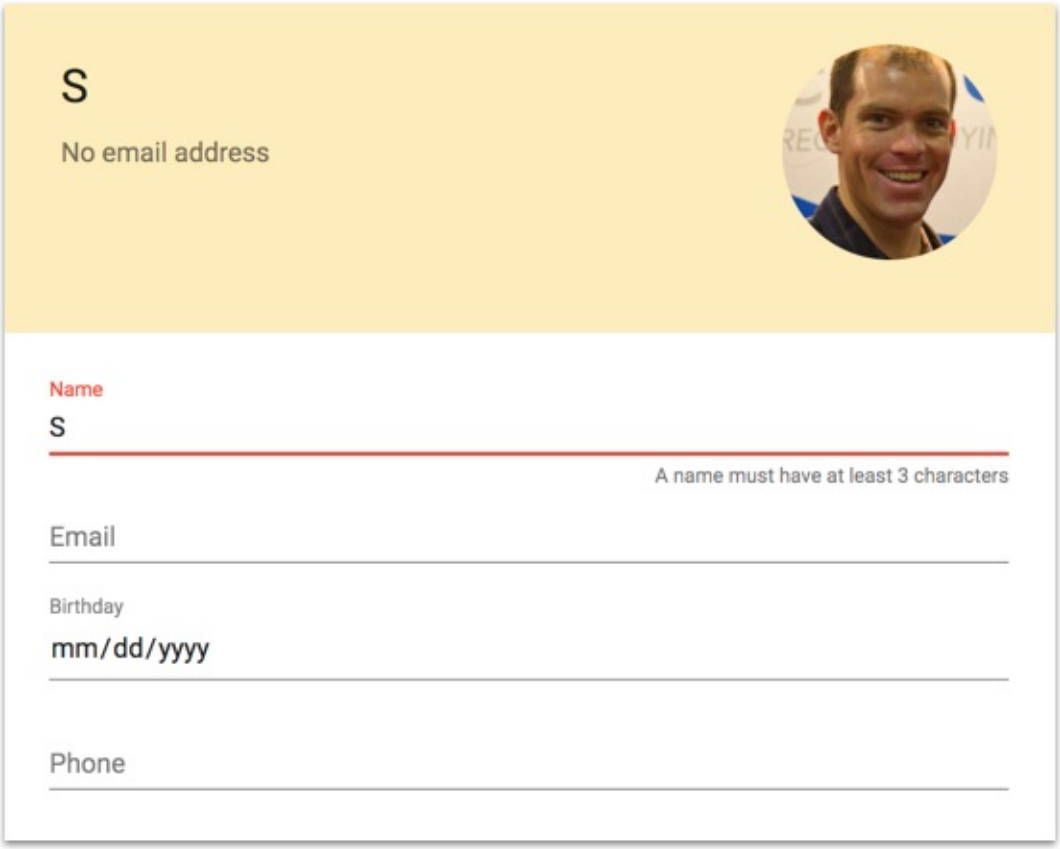


Figure: f20

Scenario

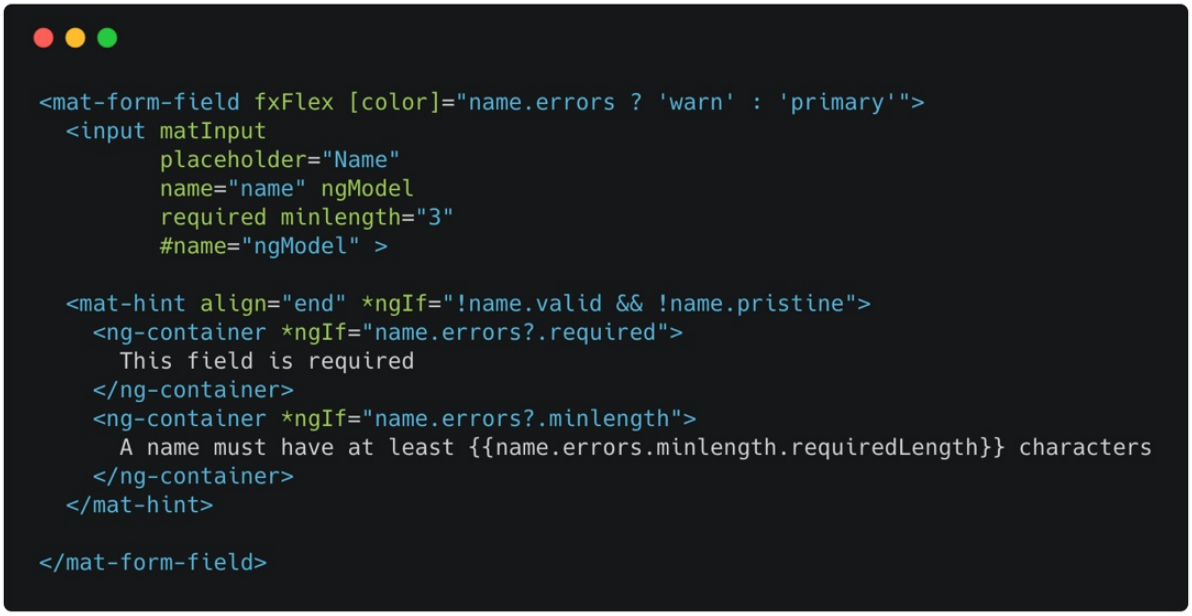
We don't have any validations in our new form. Let's use the built-in `required` and `minlength` validator to have some validation and display appropriate error messages.

Tasks

1. Add the `novalidate` HTML5 attribute to the form, so the browser's default validation is turned off
2. Disable `ContactCreatorComponent` 's save button using the forms validity state
3. Set `<mat-form-field>` 's `color` property to either "warn" or "primary" depending on if the field has errors or not
4. Add `required` and `minlength="3"` validator to the input control
5. Add an `<mat-hint>` element to into the `<mat-form-field>` if the field is not `valid` and not `pristine` . Here's a snippet:
6. Display error message inside the `<mat-hint>` element using `ngIf` for both `required` and `minlength`
7. Add information about `minlength` error in error message (`actualLength`)

Code Snippets

`contacts-creator.component.html`



```
<mat-form-field fxFlex [color]="name.errors ? 'warn' : 'primary'">
  <input matInput
         placeholder="Name"
         name="name" ngModel
         required minlength="3"
         #name="ngModel" >

  <mat-hint align="end" *ngIf="!name.valid && !name.pristine">
    <ng-container *ngIf="name.errors?.required">
      This field is required
    </ng-container>
    <ng-container *ngIf="name.errors?.minlength">
      A name must have at least {{name.errors.minlength.requiredLength}} characters
    </ng-container>
  </mat-hint>

</mat-form-field>
```

Figure: f2 1

Bonus Tasks

1. Refactor `ContactsEditorComponent` to also use template-driven forms and the same validation (keep in mind that one button is of type `submit` whereas the other one is of type `button`.

Additional resources and help

Next Lab

Go to Lab #3: Add Custom eMail Validator

Lab 3: Add custom email validator

Scenario

Write your first custom validator.

The email field is optional, which is okay. However, we're able to enter whatever we want. Let's make sure the entered value is an actual email address.

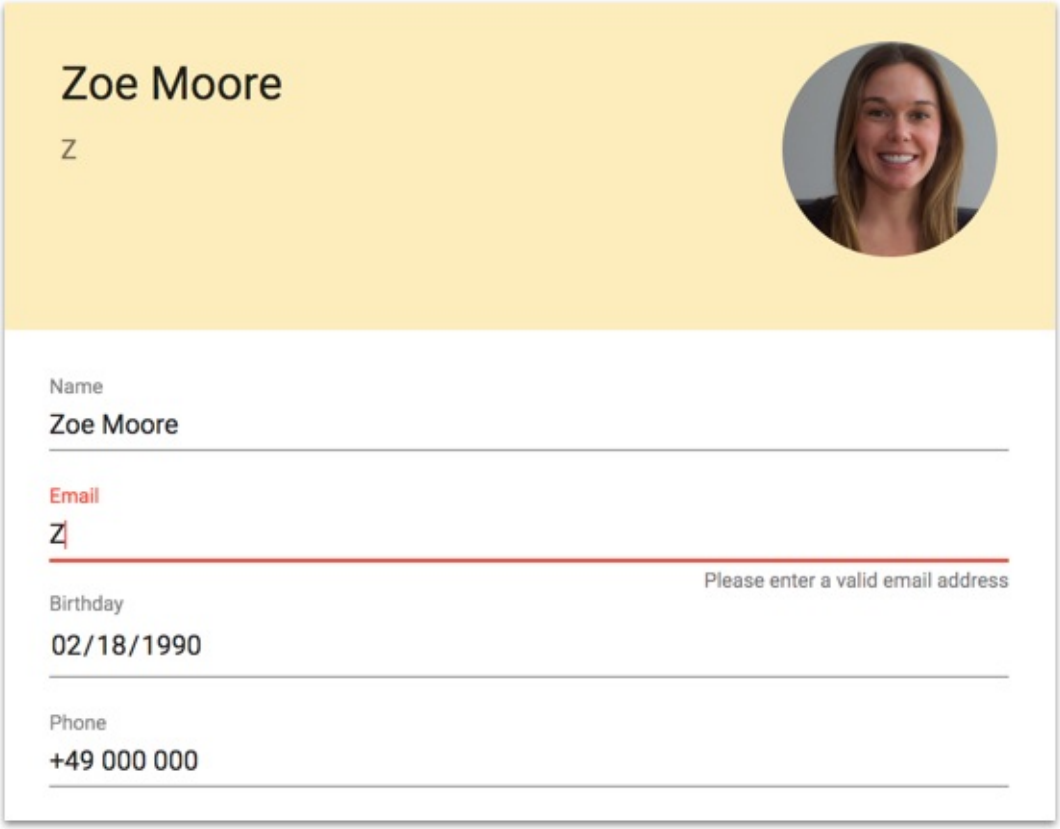


Figure: f22

Tasks

1. Create a new directive using angular-cli `ng generate directive email-validator`
2. Import `FormControl` from `@angular/forms` in that file
3. Write a stand-alone **function** `validateEmail(c: FormControl)` that uses a regular expression to test the value of the given control.
    - The Regexp is:

```
const VALID_EMAIL = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;
```

4. Use `.test()` on the regexp to check if control value is an email address

5. In case of an error, return an object that looks like this:

```
{
 validateEmail: {
   valid: false
 }
}
```

6. Change `EmailValidator` directive's selector to `[trmValidateEmail][ngModel]`
7. Add the new validator to the `NG_VALIDATORS` multi provider
8. Add `EmailValidator` to the ngModule `declarations` in `app.modules.ts` (probably already one by angular cli)
9. Apply `trmValidateEmail` directive on email input control
10. Set `<mat-form-field>` 's `color` property to either "warn" or "primary" depending on if the field has errors or not
11. Add an `<mat-hint>` element to into the `<mat-form-field>` if the field is not `valid` and not `pristine`. Here's a snippet:

Code Snippets

`email-validator.directive.ts`

```
import { Directive } from '@angular/core';
import { NG_VALIDATORS, FormControl } from '@angular/forms';

const EMAIL_REGEXP = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;

export function validateEmail(c: FormControl) {
  return (EMAIL_REGEXP.test(c.value) || c.value === '') ? null : {
    validateEmail: {
      valid: false
    }
  };
}

@Directive({
  selector: '[trmValidateEmail][ngModel]',
  providers: [
    { provide: NG_VALIDATORS, useValue: validateEmail, multi: true }
  ]
})
export class EmailValidatorDirective {}
```

Figure: group 10

`contacts-creator.component.html`

```
<mat-form-field fxFlex [color]="email.errors ? 'warn' : 'primary'">

  <input matInput
         placeholder="Email"
         name="email"
         #email="ngModel"
         [(ngModel)]="contact.email"
         trmValidateEmail>

  <mat-hint align="end" *ngIf="!email.valid && !email.pristine && !email.untouched">
    <ng-container *ngIf="email.errors?.validateEmail">
      Please enter a valid email address
    </ng-container>
  </mat-hint>

</mat-form-field>
```

Bonus Tasks

1. Apply the same validator in the `ContactsEditorComponent` as well

Next Lab

Go to Lab #4: Custom Async Validators

Lab 4: Create async validator directive

In this exercise we'll explore how async validators are implemented in Angular. We'll learn how to create a validator that has service dependencies and how it needs to be added to the `NG_ASYNC_VALIDATORS` providers.

Scenario

Email addresses should be unique. We shouldn't have multiple contacts that share the same email address. That's why we want make sure that, when a contact is added, the email address (if applied) isn't already used in an existing contact.

Tasks

1. Add a method `isEmailAvailable(email)` to `ContactsService`
   - Use `http.get()` to send a request to `{API-ENDPOINT}/check-email?email={EMAIL}`
   - This endpoint returns
     - `{ error: 'NOT_AVAILABLE' }` - in case the given email address is in use
     - `{ msg: 'AVAILABLE' }` - in case the email address is available
2. Create a directive using angular-cli called `email-availability-validator`
3. Add a validator **factory** function `checkEmailAvailability(contactsService: ContactsService)` to `src/app/email-availability-validator.ts`.
4. Return the observable of `ContactsService#isEmailAvailable()`, make sure that it projects/emits the right next value, which is:
   - `null` - if the email address is available or,
   - `{ emailTaken: true }` - if the email address is not available
5. Inject `ContactsService` into `EmailAvailabilityValidator`
6. Use `checkEmailAvailability()` as a factory function to store a validator on the class
7. Implement a method `validate(c: FormControl)` on `EmailAvailabilityValidator` which uses the new validator
8. Change `EmailAvailabilityValidator`'s selector to `[trmCheckEmailAvailability][ngModel]`.
9. Add the validator to `NG_ASYNC_VALIDATORS` using `forwardRef` and `useExisting` when configuring the provider
10. Apply `trmCheckEmailAvailability` directive to email input in `ContactCreatorComponent`s template
11. Display an error message accordingly

Code Snippets

`contacts.service.ts`

```
interface EmailAvailableResponse { msg?: string, error?: boolean }


@Injectable()
export class ContactsService {

  constructor(private http: HttpClient, @Inject(API_ENDPOINT) private apiEndpoint) {}

  isEmailAvailable(email: string) {
    const validationURL = `${this.apiEndpoint}/check-email?email=${email}`;
    return this.http.get<EmailAvailableResponse>( validationURL );
  }
}
```

`email-availability-validator.ts`

```
import { Directive, forwardRef } from '@angular/core';
import { FormControl, NG_ASYNC_VALIDATORS } from '@angular/forms';
import { of } from 'rxjs';
import { map } from 'rxjs/operators';
import { ContactsService } from './contacts.service';

export function checkEmailAvailability(contactsService: ContactsService, allowedEmail?: string) {
  return (c: FormControl) => {
    if (allowedEmail && allowedEmail === c.value) {
      return of(null);
    }
    return contactsService.isEmailAvailable(c.value).pipe(
      map(response => !response.error ? null : {
        emailTaken: true
      })
    );
  };
}

@Directive({
  selector: '[trmCheckEmailAvailability][ngModel]',
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: forwardRef(() => EmailAvailabilityValidatorDirective),
      multi: true
    }
  ]
})
export class EmailAvailabilityValidatorDirective {

  _validate: Function;

  constructor(contactsService: ContactsService) {
    this._validate = checkEmailAvailability(contactsService);
  }

  validate(c: FormControl) {
    return this._validate(c);
  }
}
```

Figure: f42

`contact-creator.component.html`

```
<mat-form-field fxFlex [color]="email.errors ? 'warn' : 'primary'">

  <input matInput
         placeholder="Email"
         name="email"
         #email="ngModel"
         ngModel
         trmValidateEmail
         trmCheckEmailAvailability>

  <mat-hint align="end" *ngIf="!email.valid && !email.pristine">
    <ng-container *ngIf="email.errors?.validateEmail">
      Please enter a valid email address
    </ng-container>
    <ng-container *ngIf="email.errors?.emailTaken">
      This email address is already taken
    </ng-container>
  </mat-hint>

</mat-form-field>
```

Figure: f43

Next Lab

Go to Lab #5: Refactor to Reactive Forms

Lab 5: Refactor to model-driven forms

Template-driven forms are nice and get us going very quickly. However, they are hard to test without e2e tests. In this exercise we're going to refactor our existing form to a model-driven form using `FormControl`, `FormGroup` and `FormBuilder` APIs.

Tasks

1. Add `ReactiveFormsModule` to our application module's imports
2. Export `validateEmail()` from `src/app/email-validator.ts`
3. Export `checkEmailAvailability(contactsService)` from `src/app/email-availability-validator.ts`
4. Import `FormControl`, `FormGroup`, `FormBuilder` and `Validators` from `@angular/forms` in `ContactsCreatorComponent`
5. Import `checkEmailAvailability` and `validateEmail` functions in `ContactsCreatorComponent`
6. Inject `FormBuilder` into `ContactsCreatorComponent`
7. Create a `FormGroup` using `FormBuilder` for all fields in the form and assign it to a `form` property on the component
8. Apply validators to form fields imperatively
9. Use `[formGroup]` to associate `form` with the DOM and remove `#form="ngForm"`
10. Use `formControlName` to associate single form controls to the input DOM elements
11. Use `form.get('FIELDNAME')` APIs to access validity state of fields and display error messages
12. Remove all validation directives from input fields in the template

Code Snippets

`contacts-creator.component.ts`

```
export class ContactsCreatorComponent {

  countries = COUNTRIES_DATA;
  gender = GENDER;

  form = this.formBuilder.group({
      name: ['', [Validators.required, Validators.minLength(3)]],
      email: ['', validateEmail, checkEmailAvailability(this.contactsService)],
      phone: '',
      gender: '',
      birthday: '',
      website: '',
      address: this.formBuilder.group({
        street: '',
        zip: '',
        city: '',
        country: ''
      })
    });
  }

  constructor(
      private router: Router,
      private contactsService: ContactsService,
      private formBuilder: FormBuilder) {}


  save(value: Contact) {
    this.contactsService.addContact(value)
      .subscribe(() => this.router.navigate(['/']));
  }
}
```

Figure: fr1 1

`contact-creator.component.html`

```
<mat-form-field
   fxFlex
   [color]="form.get('email').errors ? 'warn' : 'primary'"
   trmCheckEmailAvailability>

  <input matInput placeholder="Email" formControlName="email">
  <mat-hint align="end" *ngIf="!form.get('email').valid && !form.get('email').pristine">
    <ng-container *ngIf="form.get('email').hasError('validateEmail')">
      Please enter a valid email address
    </ng-container>
    <ng-container *ngIf="form.get('email').hasError('emailTaken')">
      This email address is already taken
    </ng-container>
  </mat-hint>

</mat-form-field>
```

Figure: f43

Bonus Tasks

1. Try to apply reactive form APIs to `ContactsEditorComponent` as well

Next Lab

Go to Lab #6: Use form array for dynamic form fields

Lab 6: Use FormArray for dynamic form fields

So far we've learned how to use reactive form APIs to create `FormGroup` and `FormControl`. While `FormGroup` represents a static known collection of `FormControl`s, Angular also enables us to great an unkown collection of `FormControl`s using the `FormArray` type.

`FormArray` is useful when we deal with forms, which structure can change at runtime.

Scenario

A contact's structure comes with several fields, including `name`, `email` or `birthday`. Another field is the `phone` field. While it's totally fine to have a contact with just a single phone number, it's nowadays very common that a contact can have multiple numbers for different platforms. Let's change `ContactCreatorComponent` to use `FormArray` so we can dynamically add `FormControl`s for `phone`.

Tasks

1. Import `FormArray` and `FormControl` from `@angular/forms` in `contacts-creator.component.ts`
2. Change the `phone` field to be a `FormArray` using `FormBuilder#array()`
3. Change `ContactsCreatorComponent` template to render multiple phone numbers. You can use the snippet below
4. Make sure to insert the correct expression so `NgFor` outputs multiple templates
5. Add the correct `[formControlName]` binding
6. Implement `addPhoneField()` method in `ContactsCreatorComponent`
7. Implement `removePhoneField(index)` method in `ContactsCreatorComponent`

Code Snippets

`contacts-creator.component.html`

```
<div formArrayName="phone">
  <div *ngFor="INSERT_EXPRESSION">
    <mat-form-field>
      <input matInput placeholder="Phone">
    </mat-form-field>
    <button
      mat-icon-button
      type="button"
      *ngIf="i >= 1"
      (click)="removePhoneField(i)"><mat-icon>highlight_off</mat-icon></button>

    <button
      mat-icon-button
      type="button"
      *ngIf="l && phone.value != ''"
      (click)="addPhoneField()"><mat-icon>add_circle_outline</mat-icon></button>
  </div>
</div>
```

`contacts-creator.component.ts`

```
@Component({
  selector: 'contacts-creator',
  templateUrl: './contacts-creator.component.html',
  styleUrls: ['./contacts-creator.component.css'],
})
export class ContactsCreatorComponent implements OnInit {

  removePhoneField(index) {
    const control = <FormArray>this.form.get('phone');
    control.removeAt(index);
  }

  addPhoneField() {
    const control = <FormArray>this.form.get('phone');
    control.push(new FormControl(''));
  }

}
```

Figure: group 14

Bonus Tasks

1. Try to apply the same things to `ContactsEditorComponent`

Next Lab

Go to Lab #7: Create custom form control

Lab 7: Create custom form control

Native HTML form controls are usually very good at what they are doing. They are accessible and are well supported in modern browsers. However, sometimes we want to build more sophisticated controls, or simply reuse existing form logic. That's where custom form controls come into play.

Scenario

You might have noticed that `ContactsCreatorComponent` and `ContactsEditorComponent` are very similar. A lot of logic is redundant, which is usually a good sign to refactor things. Let's start with small steps and refactor all address related fields into a custom form control, so it can be reused in both components, without reimplementing them more than once.

```
$ ng g c address-input
```

Tasks

1. Create a new component `AddressInputComponent` using Angular CLI
2. Change its template to render address related form fields:
3. Import `ControlValueAccessor` , `FormGroup` , `FormBuilder` and `NG_VALUE_ACCESSOR` from `@angular/forms`
4. Create a multi provider for the token `NG_VALUE_ACCESSOR` that registers `AddressInputComponent` using `useExisting` provider strategy and `multi: true`
5. Make `AddressInputComponent` implement `OnInit` and `ControlValueAccessor`
6. Inject `FormBuilder` instance into `AddressInputComponent`
7. Create a form in `ngOnInit()` that has all the form controls the template needs
8. Implement `writeValue(address: Address)`
9. Implement `registerOnChange(fn)` and `registerOnTouched(fn)`
10. Subscribe to form value changes and call your registerd change handler whenever the form emits a change
11. Call onTouched handler whenever any of the form controls emit a `blur` event
12. Use `AddressInputComponent` in `ContactsCreatorComponent` and `ContactsEditorComopnent`

    ```
    <trm-address-input formControlName="address"></trm-address-input>
    ```

> Don't forget to register this custom form control in your module declarations!

Code Snippets

`address-input.component.html`

```
<fieldset [formGroup]="form" fxLayout="column">
  <legend>Address</legend>
  <mat-form-field fxFlex>
    <input matInput placeholder="Street" (blur)="propagateTouch()" formControlName="street">
  </mat-form-field>
  <mat-form-field fxFlex>
    <input matInput placeholder="Zip" (blur)="propagateTouch()" formControlName="zip">
  </mat-form-field>
  <mat-form-field fxFlex>
    <input matInput placeholder="City" (blur)="propagateTouch()" formControlName="city">
  </mat-form-field>
```

```
    <mat-select placeholder="Country" (blur)="propagateTouch()" formControlName="country">
      <mat-option *ngFor="let country of countries" [value]="country.name">{{ country.name }}</mat-option>
    </mat-select>
  </fieldset>
```

`address-input.component.ts`

```
@Component({
  selector: 'trm-address-input',
  templateUrl: './address-input.component.html',
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => AddressInputComponent),
      multi: true
    }
  ]
})
export class AddressInputComponent implements OnInit, ControlValueAccessor {

  countries = COUNTRIES_DATA;
  form: FormGroup;
  propagateChange = (_: Address) => {};
  propagateTouch = (_: any) => {};

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit() {
    this.form = this.formBuilder.group({
      street: '',
      zip: '',
      city: '',
      country: ''
    });

    this.form.valueChanges.subscribe(address => this.propagateChange(address));
  }

  writeValue(address: Address) {
    this.form.setValue(address, {emitEvent: false});
  }

  registerOnChange(fn) {
    this.propagateChange = fn;
  }

  registerOnTouched(fn) {
    this.propagateTouch = fn;
  }
}
```

Figure: group 15

Routing

Let's explore Routing within Angular:

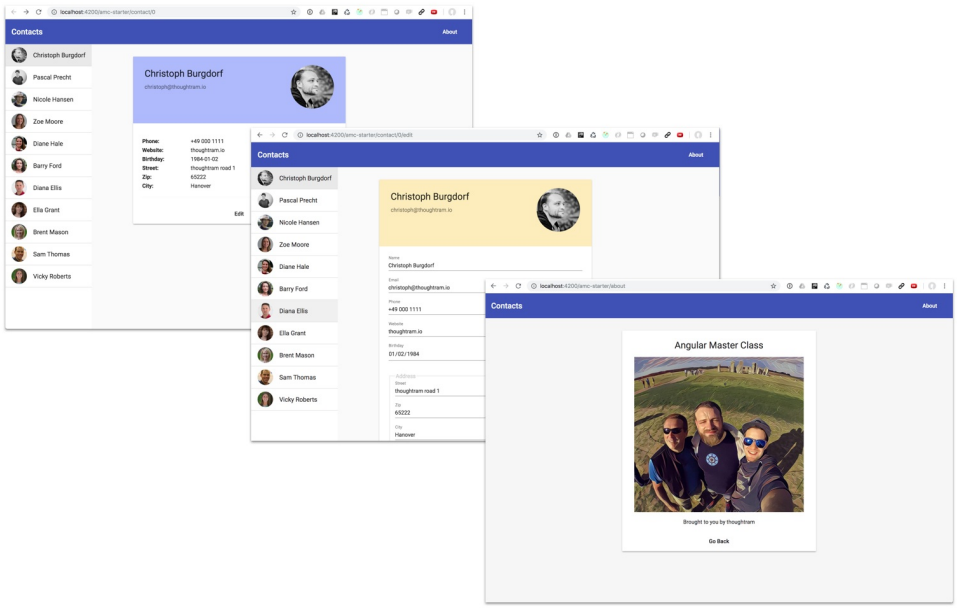


Figure: rrdme

Lab Exercises

- Lab #1: Child Routes
- Lab #2: Router CanDeactivate
- Lab #3: Navigation Gaurds
- Lab #4: Route Resolvers
- Lab #4: Lazy Loading Modules

Lab 1: Children Routes

Using children routes we can build components with nested `<router-outlet>` 's to implement more sophisticated scenarios.

Currently our app always shows just one screen at a time. We like to change our application to have two main views where the first is a split view that has the list of contacts on the left and opens a contact on the right.

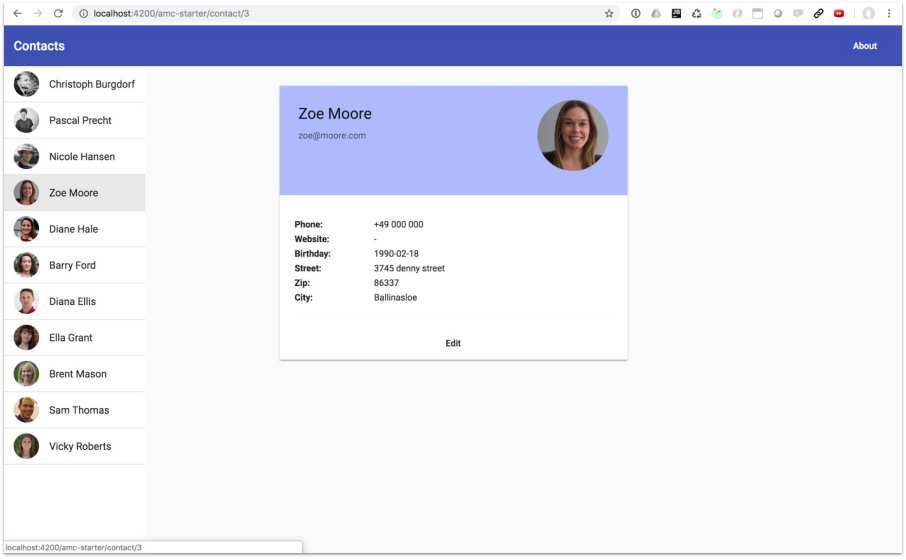


Figure: r1

Scenario

The contacts can still be edited so the right hand side of the split view is either filled with the `ContactsDetail(View)Component` , or the `ContactsEditorComponent` . The default route will simply redirect to `contact/0` .

In terms of `<router-outlet>` 's that means that we need a `<router-outlet>` on our **`ContactsAppComponent`** component to either inject the component that assembles the split view (let's call it **`ContactsDashboardComponent`**) or the component that shows the about page.

At the same time our **`ContactsDashboardComponent`** also needs a `<router-outlet>` to toggle the right hand side between the **`ContactsDetailComponent`** and the **`ContactsEditorComponent`** .

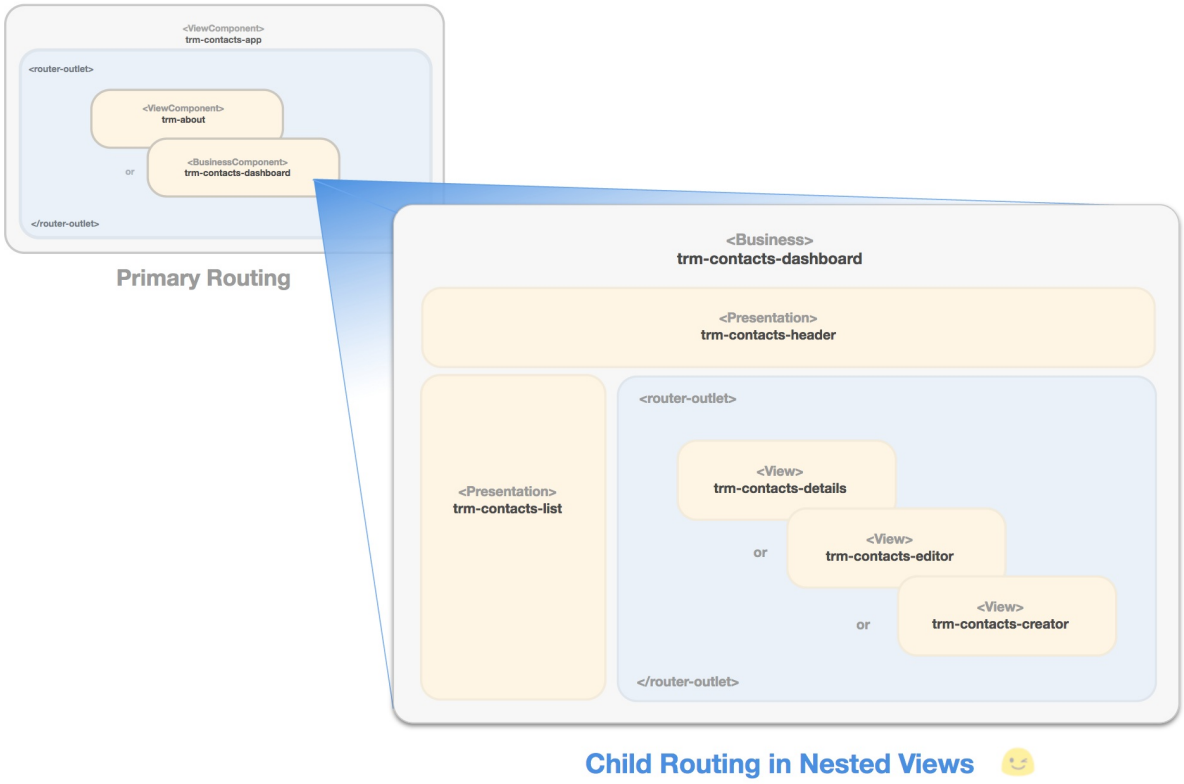


Figure: master-details

Tasks

1. Create a **`ContactsDashboardComponent`** for the split view
2. Configure the routes for the **`ContactsDashboardComponent`** with the different child routes for the right hand side of the split view
3. In the **`ContactDetail(View)Component`** subscribe to `this.route.params` to retrieve the `id` instead of using the snapshot. The snapshot can't be used since the component may be reused
4. Redirect the default route to `contact/0` (`{ path: '', redirectTo: 'contact/0', pathMatch: 'full' }`)
5. Create an **`AboutComponent`** for the about page
6. Configure a route to load the **`AboutComponent`** for the `about` url
7. You may also want to remove the back button from the **`ContactDetail(View)Component`**

Code Snippets

In order to be able to actually navigate to the about page we'll change the header in `ContactsAppComponent` 's template to this:

`app.component.html`

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

@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar color="primary">
      <div fxLayout fxLayoutAlign="space-between center" fxFlex>
        Contacts
        <a mat-button title="Go to about page" routerLink="/about" class="right">
            About
        </a>
      </div>
    </mat-toolbar>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['./app.component.scss']
})
export class ContactsAppComponent {}
```

Figure: r1 2

For the `ContactsDashboardComponent`, we want to use Angular Materials' `MatDrawer` component to render the `ContactsListComponent`. The template can be as simple as this.

`contacts-dashboard.component.html`

```
<mat-drawer-container>

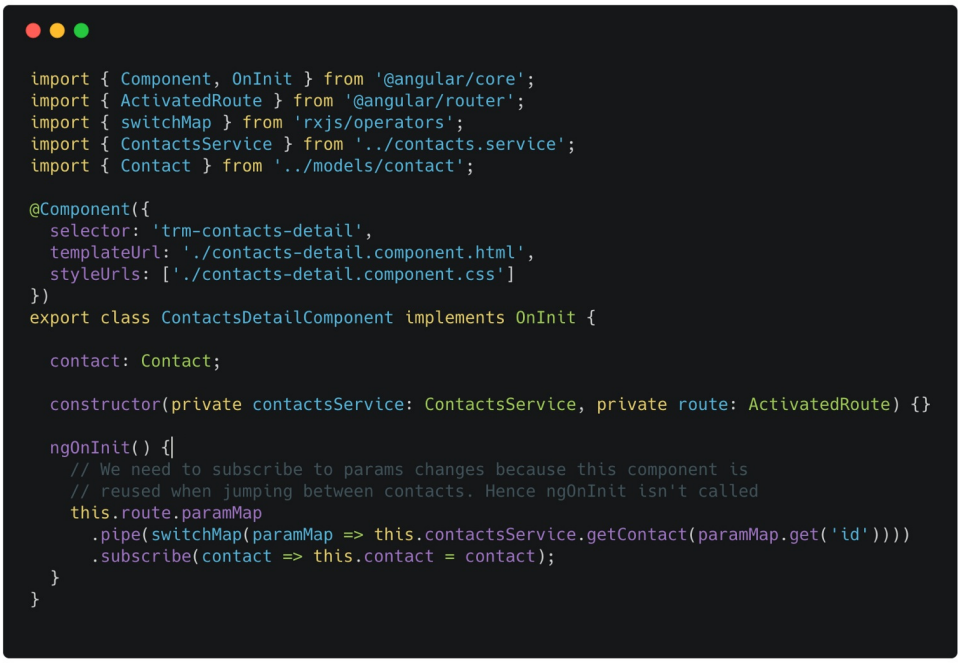
  <mat-drawer mode="side" opened="true">
    <trm-contacts-list></trm-contacts-list>
  </mat-drawer>

  <div class="main-content">
    <router-outlet></router-outlet>
  </div>

</mat-drawer-container>
```

Figure: r1 3

`contact-detail-view.component.ts`



```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { switchMap } from 'rxjs/operators';
import { ContactsService } from '../contacts.service';
import { Contact } from '../models/contact';

@Component({
  selector: 'trm-contacts-detail',
  templateUrl: './contacts-detail.component.html',
  styleUrls: ['./contacts-detail.component.css']
})
export class ContactsDetailComponent implements OnInit {

  contact: Contact;

  constructor(private contactsService: ContactsService, private route: ActivatedRoute) {}

  ngOnInit() {
    // We need to subscribe to params changes because this component is
    // reused when jumping between contacts. Hence ngOnInit isn't called
    this.route.paramMap
      .pipe(switchMap(paramMap => this.contactsService.getContact(paramMap.get('id'))))
      .subscribe(contact => this.contact = contact);
  }
}
```

Figure: r1 1

`about.component.ts`

```
<div class="trm-about">
  <mat-card fxLayout="column" fxFlex fxLayoutAlign="center center">
    <h2 mat-card-title>Angular Master Class</h2>
    <mat-card-content>
      <img src="/assets/images/team.jpg" alt="Team thoughtram">
      <p style="text-align: center;">Brought to you by thoughtram</p>
    </mat-card-content>
    <mat-card-actions>
      <a mat-button title="Go back to dashboard"
         routerLink="/">
        Go Back
      </a>
    </mat-card-actions>
  </mat-card>
</div>
```

Web and App Servers

To launch your web application and the REST server, use a Terminal session with the command:

```
$   ./start
```

Next Lab

Go to Lab #2: Router CanDeactivate

Lab 2: Routings Guard

Using a Guard we can prevent navigation to happen if the Guard returns `false` or an `Observable<boolean>` which emits `false`.

Scenario

Now that we have a split view it may easily happen that we accidentally click on another contact from the list while we have the **`ContactsEditorComponent`** open on the right hand side.

To prevent that from happening we want to implement a **`CanDeactivate`** Guard for the route to prompt the user with a question if they really like to navigate away without saving.

Tasks

1. Create **an exported** navigation guard function `confirmNavigationGuard(component)` in `app.module.ts` which returns `window.confirm('Navigate away without saving?')`
2. Add the guard to the `canDeactivate` array of the route that shows the `ContactsEditorComponent`
3. Prevent showing the confirm dialog when the user clicks on the `Save` button

Code Snippets

`app.module.ts`

```
export function confirmNavigationGuard(component) {
  const question = 'Navigate away without saving?';
  return !component.warnOnClosing || window.confirm(question);
}

@NgModule({
  declarations: [
    ...
  ],
  imports: [
    ...
  ],
  providers: [
    { provide: 'ConfirmNavigationGuard', useValue: confirmNavigationGuard }
    ...
  ]
})
export class ContactsModule {}
```

Figure: r2 1

`app.routes.ts`

```
export const APP_ROUTES: Routes = [
  {
    path     : '',
    component: ContactsDashboardComponent,

    children: [
      { path: '', redirectTo: 'contact/0', pathMatch:'full' },
      { path: 'contact/:id', component: ContactsDetailComponent },
      {
        path         : 'contact/:id/edit',
        component    : ContactsEditorComponent,
        canDeactivate: [ 'ConfirmNavigationGuard' ]
      }
    ]
  }
];
```

Figure: r2 2

Next Lab

Go to Lab #3: Navigation Gaurds

Lab 3: Navigation Guards + MdDialog

In this exercise we'll learn how to create a navigation guard as a class as well as using the Angular Material `MatDialog` component to create nice and shiny dialogs.

Scenario

Let's take our navigation guard to the next level. Right now it's just a function which returns `window.confirm()` and that's totally fine. However, instead of just showing a browser window to confirm or cancel the navigation, we want a proper dialog.

Angular Material comes with an `MatDialog` component that we can use for exactly that.

Tasks

1. Create a file with a new navigation guard class called `CanDeactivateContactsEditorGuard`
2. Import `ContactsEditorComponent`
3. Import `CanDeactivate` from `@angular/router`
4. Create a component `ConfirmDeactivationDialogComponent` with the following template:
5. Import `MatDialogRef` from `@angular/material` and inject it into `ConformDeactivationDialogComponent` with the same generic type (Read `MatDialog` s documentation for more information)
6. Inject `dialog: MatDialog` into `CanDeactivateContactsEditorGuard`
7. Create a method `canDeactivate(component: ContactsEditorComponent)` in `CanDeactivateContactsEditorGuard` in which you call `this.dialog.open(ConfirmDeactivationDialogComponent)`
8. Return `this.dialog.afterClosed()` , which is an `Observable<boolean>`
9. Add the guard to your providers
10. In app.module, add an `entryComponents` for `ConfirmDeactivationDialogComponent` .
11. Apply the guard to the route

Code Snippets

`can-deactivate-contacts-editor.guard.ts`

```ts
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { MatDialogRef, MatDialog } from '@angular/material';
import { of } from 'rxjs';
import { ContactsEditorComponent } from './contacts-editor.component';
import { ConfirmDeactivationDialogComponent } from './confirm-deactivation-dialog.component';

@Injectable()
export class CanDeactivateContactsEditorGuard implements CanDeactivate<ContactsEditorComponent> {

  dialogRef: MatDialogRef<ConfirmDeactivationDialogComponent>;

  constructor(public dialog: MatDialog) {}

  canDeactivate(component: ContactsEditorComponent) {
    if (component.warnOnClosing) {
      this.dialogRef = this.dialog.open(ConfirmDeactivationDialogComponent, {
        disableClose: false
      });

      return this.dialogRef.afterClosed();
    }
    return of(true);
  }
}
```

Figure: r3 1

`confirm-deactivate-dialog.component.ts`

```ts
import { Component } from '@angular/core';
import { MatDialogRef } from '@angular/material';

@Component({
  selector: 'trm-confirm-deactivation-dialog',
  template: `
    <h3 mat-dialog-title>Are you sure?</h3>
    <div mat-dialog-content>All unsaved changes will be gone.</div>

    <mat-dialog-actions fxLayout fxLayoutAlign="center center">
      <button mat-button (click)="dialogRef.close(true)">Yes</button>
      <button mat-button mat-dialog-close>No</button>
    </mat-dialog-actions>
  `
})
export class ConfirmDeactivationDialogComponent {
  constructor(public dialogRef: MatDialogRef<ConfirmDeactivationDialogComponent>) {}
}
```

Figure: r3 2

`app.routes.ts`

```
export const APP_ROUTES: Routes = [
  {
    path     : '',
    component: ContactsDashboardComponent,

    children: [
      { path: '', redirectTo: 'contact/0', pathMatch:'full' },
      { path: 'contact/:id', component: ContactsDetailComponent },
      {
        path          : 'contact/:id/edit',
        component     : ContactsEditorComponent,
        canDeactivate : [ CanDeactivateContactsEditorGuard ]
      }
    ]
  }
];
```

Figure: r3 3

Next Lab

Go to Lab #4: Route Resolvers

Lab 4: Route Resolvers

We often want to postpone the activation of a route until some data is loaded. We can do exactly that using a Resolver.

Scenario

Currently our **`ContactsDetail(View)Component`** and **`ContactsEditorComponent`** are rendered before the contact is actually loaded. In fact the loading of the contact is initiated by these components.

> Let's rewrite our application so that these components are not actually loaded until the contact was fetched successfully.

Tasks

1. Create a file `app/shared/contacts.resolver.ts`
   i. Create a class **`ContactsResolver`** that implements the `Resolve` interface from `@angular/router`
   ii. In the `resolve` method use the `ActivatedRouteSnapshot` and the **`ContactsService`** to fetch the contact and return the Observable
2. Specify a provider for the **`ContactsResolver`** in the `ContactsModule`.
3. Change the routes for **`ContactsDetail(View)Component`** and `ContactsEditorComponent` to use the **`ContactsResolver`**
4. Remove the **`ContactsService`** from `ContactsDetail(View)Component` and use `this.route.data` instead to access the contact
5. Remove all occurrences of the safe navigation operator (`?.`) and the empty contact in both components since they aren't needed anymore.

```
  contact: Contact = <Contact>{ address: {}};
`
```

Code Snippets

`shared/contact.resolver.ts`

```
import { Injectable } from '@angular/core';
import { Resolve, ActivatedRouteSnapshot } from '@angular/router';

import { ContactsService } from '../contacts.service';
import { Contact } from '../models/contact';

@Injectable()
export class ContactResolver implements Resolve<Contact> {

  constructor(private contactsService: ContactsService) {}

  resolve(route: ActivatedRouteSnapshot) {
    return this.contactsService.getContact(route.params['id']);
  }
}
```

Figure: r4 1

Next Lab

Go to Lab #4: Lazy Loading Modules

Lab 5: Lazy Loading Modules

Instead of loading every application part upfront we can specify to load specific modules asynchronously to only fetch the files when a particular route gets activated.

Luckily, Angular's router supports lazyloading `NgModules`, which makes loading modules at runtime a breeze.

Scenario

Considering that our about page may be rarely viewed, we like to cut of the initial size of our app by fetching the `AboutComponent` asynchronously once the route becomes active. To do that, we need to create a new `NgModule` (let's say `AboutModule`), which declares the `AboutComponent`. The `AboutModule` can then be lazyloading using a routes config's `loadChildren` function.

```
$ ng g m about
```

Tasks

1. Create a new "About" `NgModule` using the Angular CLI
2. Remove `AboutComponent` from `ContactsModule`s declarations and remove the import as well
3. Import `AboutComponent` in `about.module.ts` instead and add it to `AboutModule`'s declarations.
4. Import `RouterModule` in `about.module.ts` and configure a default route for the `AboutModule` that points to `AboutComponent`, using `RouterModule.forChild(...)`
5. In order to keep the UI working, we need to add `FlexLayoutModule` and `ContactsMaterialModule` to `AboutModule`s imports as well
6. Change the root route configuration for path `about` to use `loadChildren` instead

Code Snippets

`about/about.module.ts`

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
import { FlexLayoutModule } from '@angular/flex-layout';
import { ContactsMaterialModule } from '../contacts-material.module';

import { AboutComponent } from './about.component';

@NgModule({
  imports: [
    RouterModule.forChild([
      { path: '', pathMatch: 'full', component: AboutComponent }
    ]),
    FlexLayoutModule,
    ContactsMaterialModule
  ],
  declarations: [AboutComponent]
})
export class AboutModule { }
```

Figure: r5 1

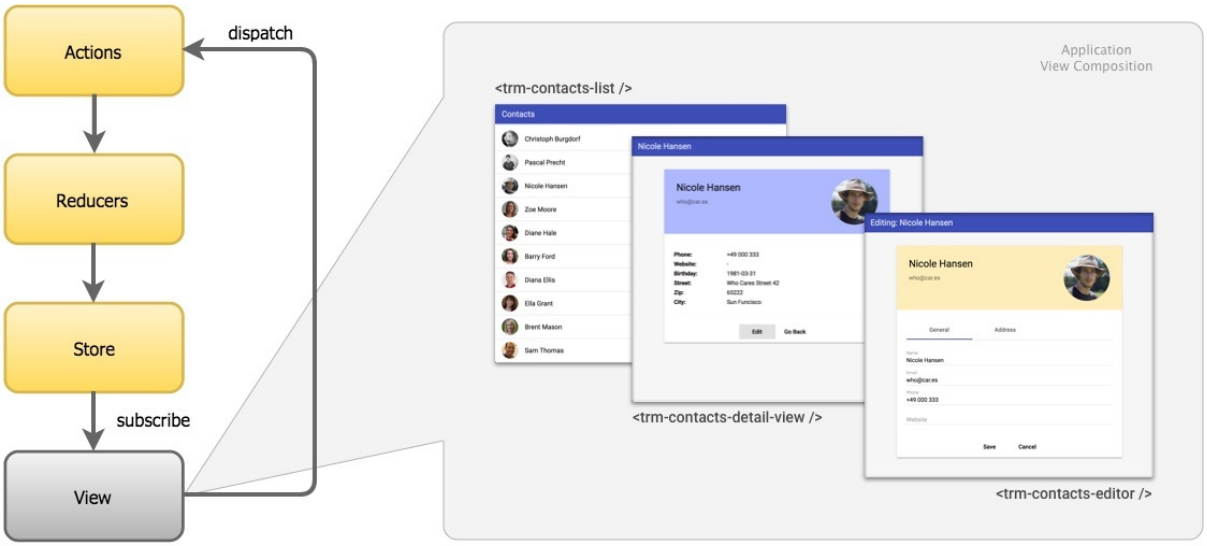
`app.routes.ts`

```
export const APP_ROUTES: Routes = [
  {
    path: '',
    component: ContactsDashboardComponent,
    children: [
      { path: '', redirectTo: 'contact/0', pathMatch:'full' },
      {
        path: 'contact/:id',
        component: ContactsDetailComponent,
        resolve: {
          contact: ContactResolver
        }
      },
      {
        path: 'contact/:id/edit',
        component: ContactsEditorComponent,
        canDeactivate: [CanDeactivateContactsEditorGuard],
        resolve: {
          contact: ContactResolver
        }
      }
    ]
  },
  { path: 'about', loadChildren: './about/about.module#AboutModule' },
  { path: '**', redirectTo: '/amc-starter' }
];
```

Figure: r5 2

@angular/ngrx

Let's use NgRx to manage state in Angular



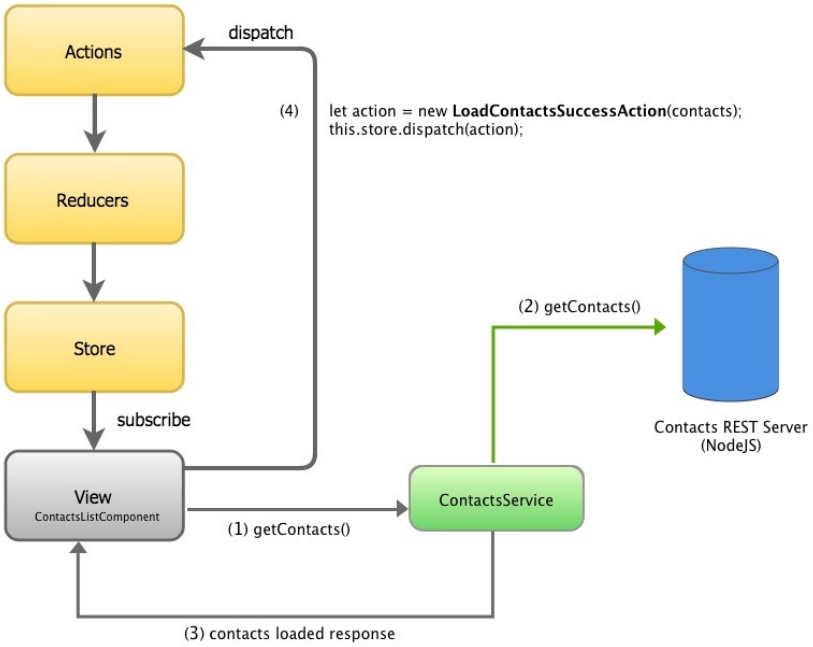


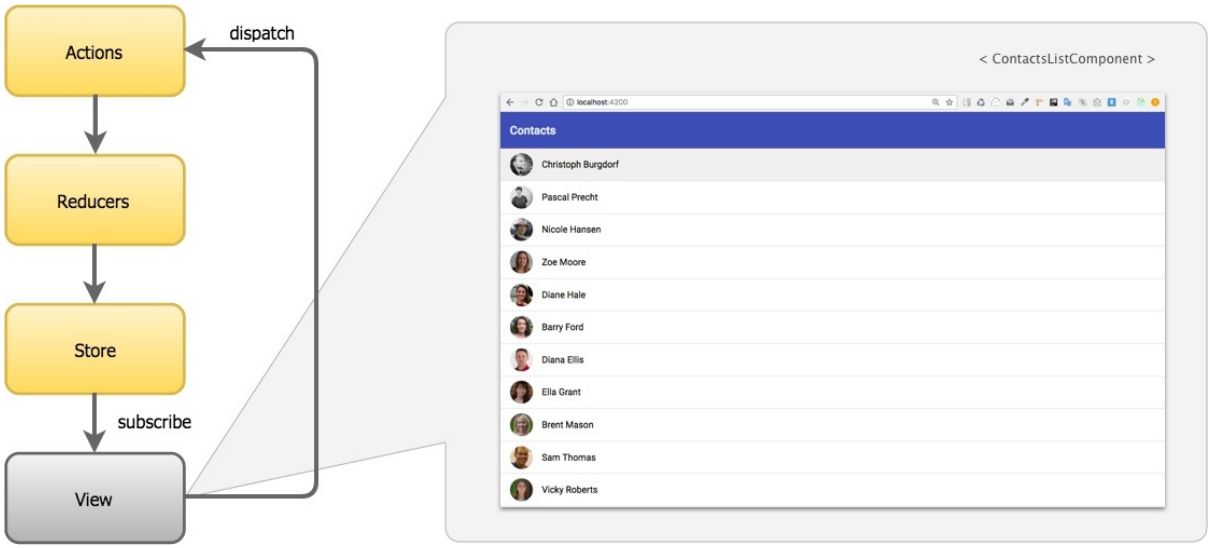
Figure: nrx-intro

To explore the concepts of the **ngrx** patterns, let's return to our Contacts application (created in the Jump Start class). Now, let's use the Angular-native, `ngrx` library to *rebuild* our Contacts application using ngrx stores, reducers, actions, and meta-reducers (previously known as middleware).

Lab Exercises

Exercise: Use ngrx within Contacts App

Let's use **ngrx** to configure our Contacts application to use the the core building blocks from the Redux pattern - store, reducers, and actions.



Scenario

After installing and configuring the **ngrx** library, which you don't have to do yourself as everything is already pre-installed if you followed the preparation guide, we will configure our `ContactsListView` to dispatch an **action** after the contacts list has been loaded:

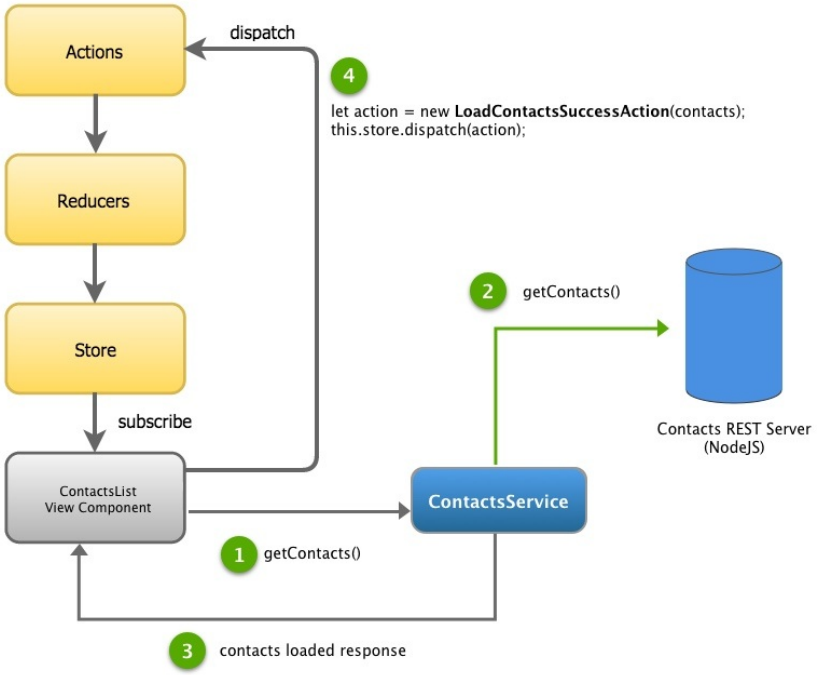


Figure: ngrx2

Tasks

1. Create the initial `src/app/state/contacts/contacts.actions.ts` module.
   - Implement the **ContactsActionTypes** and the associated **LoadContactsSuccessAction** class:
2. Create the Contacts reducer `src/app/state/contacts/contacts.reducer.ts` to process our Contacts actions.
   - Define the `FEATURE_KEY = 'contacts'` to be used to register the reducer (in Step 6 below).
   - Define the initial state
   - Define the **contactsReducer** as a **pure** function:
3. Update the `src/app/contacts-ngrx.module.ts` to use `StoreModule.forFeature()` register the `contacts` state management within our application by using the `StoreModule.forFeature()` function.
4. Modify the **ContactsListComponent**:
   - Inject our custom store dispatcher.
   - Within `ngOnInit()` , use the store to query for a contacts list observable: `Observable<Array<Contact>>` .

     > Remember to use the `select` operator from `@ngrx/store` .

   - Update `ngOnInit()` to dispatch a **LoadContactsSuccessAction** after the REST server responds with the contacts.

Considerations

Currently the view dispatches the contact list (via an action) and the store emits that same list via our observable.

At first glance this looks overly complex when we could simply update our `this.contacts` property directly. In subsequent labs we will continue with our ngrx integration and refactor these views several times... and you will see the benefits.

Code Snippets

`contacts.actions.ts`

```
import { Action } from '@ngrx/store';
import { Contact } from '../../models/contact';

export enum ContactsActionTypes {
  LOAD_CONTACTS_SUCCESS = '[Contacts] Load Contacts Success'
}

export class LoadContactsSuccessAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACTS_SUCCESS;
  constructor(public payload: Array<Contact>) {}
}

export type ContactsActions = LoadContactsSuccessAction;
```

Figure: ng1 1

`contacts.reducer.ts`

```
import { Contact } from '../../models/contact';
import { ContactsActionTypes, ContactsActions } from '../contacts/contacts.actions';

export const FEATURE_KEY = 'contacts';

export interface ContactsState {
  list: Array<Contact>;
}

export const INITIAL_STATE: ContactsState = {
  list: []
};

export function contactsReducer(state: ContactsState = INITIAL_STATE, action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      return {
        ...state,
        list: action.payload
      };
    default:
      return state;
  }
}
```

Figure: ng1 2

`contacts-list.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { Store, select } from '@ngrx/store';

import { Contact } from '../models/contact';
import { ContactsService } from '../contacts.service';

import { ApplicationState } from '../state';
import { LoadContactsSuccessAction } from '../state/contacts/contacts.actions';

@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>>;

  constructor(private store: Store<ApplicationState>, private contactsService: ContactsService) { }

  ngOnInit () {
    this.contacts$ = this.contactsService.getContacts();

    this.contactsService
      .getContacts()
      .subscribe(contacts => this.store.dispatch(new LoadContactsSuccessAction(contacts)));
  }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Figure: ng1 3

`contacts-ngrx.module.ts`

```
import {CommonModule} from '@angular/common';
import {NgModule} from '@angular/core';

import {StoreModule} from '@ngrx/store';
import {EffectsModule} from '@ngrx/effects';
import {StoreDevtoolsModule} from '@ngrx/store-devtools';
import {environment} from '../environments/environment';
import {storeFreeze} from 'ngrx-store-freeze';

import {FEATURE_KEY, contactsReducer, INITIAL_STATE} from './state';

@NgModule({
  imports: [
    CommonModule,
    StoreModule.forRoot({}, {
      metaReducers: !environment.production ? [storeFreeze] : []
    }),
    EffectsModule.forRoot([]),
    StoreModule.forFeature(FEATURE_KEY, contactsReducer, {
      initialState: INITIAL_STATE
    }),
    !environment.production ? StoreDevtoolsModule.instrument() : []
  ]
})
export class ContactsNgRxModule {
}
```

Figure: ng1 4

Next Lab

Go to Step 2: Select and Edit Actions

Exercise: Use Select and Edit Actions

The Contacts application currently uses routing and direct data-access to show the ContactDetails and the ContactEditor views. Let's integrate our use of **ngrx** into the workflow.

Scenario

Let's create two (2) new actions **SelectContactAction** and **UpdateContactAction** that will be used to view contact details or for the editor-view.

Tasks

1. Create the two (2) new actions in the `contacts-actions.ts` module.
   - Implement the **SelectContactAction** class used to select the contact in the store:
   - Implement the **UpdateContactAction** class used to save the contact changes to the server:
2. Update the Contacts reducer `contacts-reducer.ts` .
   - Using **SelectContactAction**, select the contact by the specified id (payload).
   - Using **UpdateContactAction**, add the contact or merge the current values into appropriate contact item in the store list.
3. Modify the **ContactDetailsComponent** to use the logic shown in the UML diagram above
   - ngOnInit() should dispatch `SelectContactAction`
   - `contact$ : Observable<Contact> = this.store.select(...)`
4. Modify the **ContactEditorComponent** to use the logic shown in the UML diagram above.
   - ngOnInit() should dispatch `SelectContactAction`
   - `contact$ : Observable<Contact> = this.store.select(...)`
   - `save()` should call `ContactsService::UpdateContact()` and then dispatch `UpdateContactAction()` and then route to the contact details.

UML Flow Diagrams

Shown below is a UML sequence diagram that clearly shows how actions will be integrated into the workflow process. Using this diagram, make the appropriate source changes to implement this improved, Redux workflow.

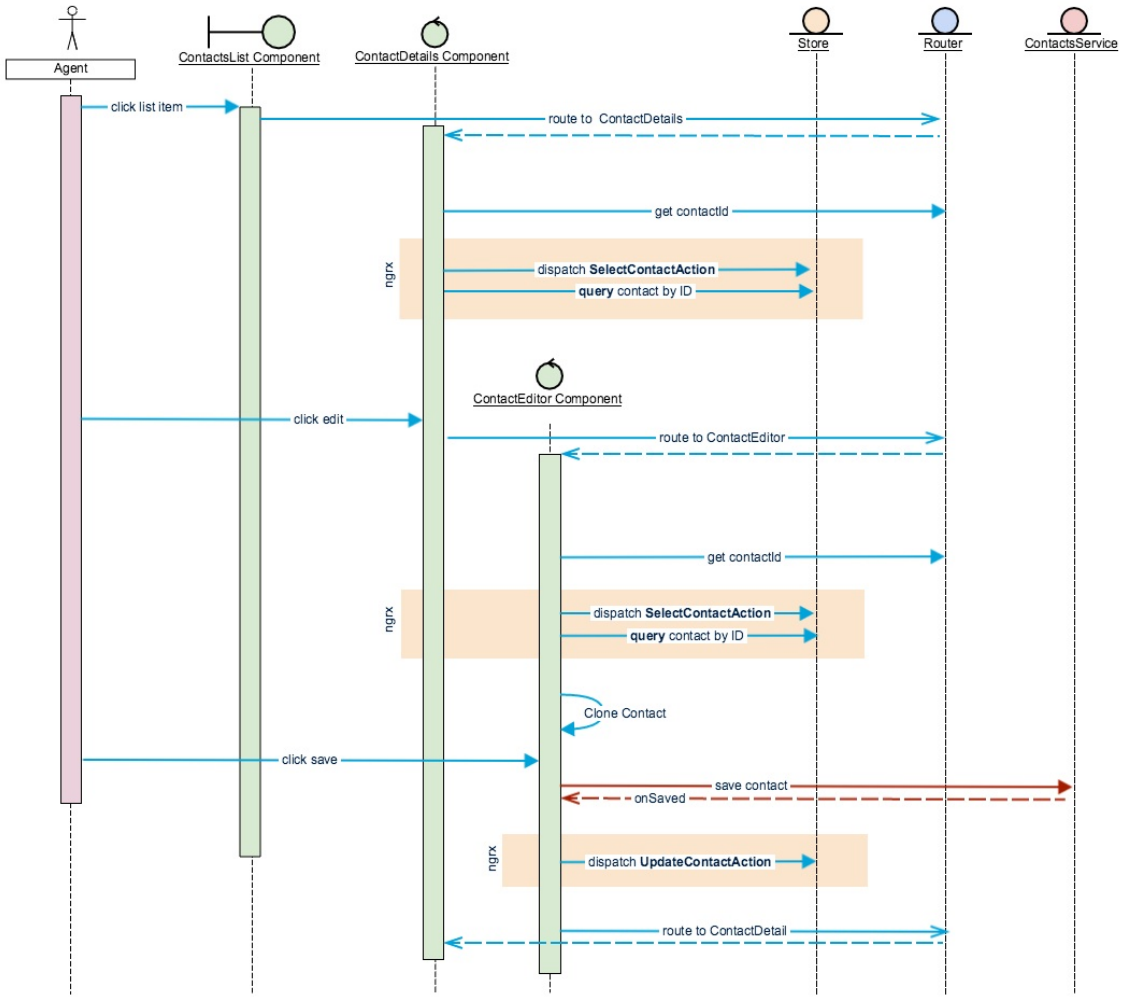


Figure: ngrx-step-2-select-and-edit-contact

Code Snippets

`contacts.actions.ts`

```
export class SelectContactAction implements Action {
  readonly type = ContactsActionTypes.SELECT_CONTACT;
  constructor(public payload: number) {}
}

export class UpdateContactAction implements Action {
  readonly type = ContactsActionTypes.UPDATE_CONTACT;

  constructor(public payload: Contact) {}
}

export type ContactsActions =
    LoadContactsSuccessAction
    | SelectContactAction
    | UpdateContactAction;
```

`contacts.reducer.ts`

```
export function contactsReducer(state: ContactsState = INITIAL_STATE, action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      return {
        ...state,
        loaded: true,
        list: action.payload
      };

    case ContactsActionTypes.SELECT_CONTACT:
      return {
        ...state,
        selectedContactId: action.payload
      };

    case ContactsActionTypes.UPDATE_CONTACT:
      const hasSameId   = contact => contact.id === action.payload.id
      const updatedList = state.list.map(contact => {
        return hasSameId(contact) ? { ...contact, ...action.payload } : contact
      });

      return {
        ...state,
        list: updatedList
      };
    default:
      return state;
  }
}
```

Figure: ngrx2 1

`contacts-detail.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Store, select } from '@ngrx/store';
import { Observable } from 'rxjs';

import { Contact } from '../models/contact';

import { ApplicationState } from '../state';
import { SelectContactAction } from '../state/contacts/contacts.actions';

@Component({
  selector: 'trm-contacts-detail',
  templateUrl: './contacts-detail.component.html',
  styleUrls: ['./contacts-detail.component.css']
})
export class ContactsDetailComponent implements OnInit {
  contact$: Observable<Contact> = this.store.pipe(select(state => {
      let id = state.contacts.selectedContactId;
      return state.contacts.list.find(contact => contact.id == id);
    }));

  constructor(private store: Store<ApplicationState>, private router: ActivatedRoute) { }

  ngOnInit() {
    let contactId = this.router.snapshot.paramMap.get('id');
    this.store.dispatch(new SelectContactAction(+contactId));
  }
}
```

Figure: ngrx2 2

`contacts-editor.component.ts`

```
@Component({
  selector: 'trm-contacts-editor',
  templateUrl: './contacts-editor.component.html',
  styleUrls: ['./contacts-editor.component.css']
})
export class ContactsEditorComponent implements OnInit {

  contact$: Observable<Contact> = this.store.pipe(
      select(state => {
        let id = state.contacts.selectedContactId;
        return state.contacts.list.find(contact => contact.id == id);
      }),
      map(contact => ({...contact}))
  );

  ...

  ngOnInit() {
    let contactId = this.route.snapshot.paramMap.get('id');
    this.store.dispatch(new SelectContactAction(+contactId));
  }

  cancel(contact: Contact) {
    this.goToDetails(contact);
  }

  save(contact: Contact) {
    this.contactsService
        .updateContact(contact)
        .subscribe(() => {
          this.store.dispatch(new UpdateContactAction(contact));
          this.goToDetails(contact);
        });
  }

  private goToDetails(contact: Contact) {
    this.router.navigate(['/contact', contact.id]);
  }
}
```

Figure: ngrx2 3

Next Lab

Go to Step 3: Use Router Guard with Actions

Exercise: Create ContactExistsGuard

Our contacts application has an issue with deep-linking (aka bookmarking). Currently, our logic requires the list of contacts to be loaded when **navigating** to the details page. In fact, `ContactsDetailComponent` retrieves the selected contact to show its details in the HTML template of the component.

One way to fix this problem is to implement a router guard which will "protect" a specific part of our application. Guards can re-route and multiple guards can protect a single route.

With a router guard we can make sure that the contact is loaded before we navigate and show the contact's details. Note that in contrast to `ContactsListComponent` were we would fetch all contacts, `ContactsDetailComponent` loads one (1) specific contact.

> You might wonder why we are using a guard instead of a resolver. The reason being is that we are not interested in the data returned by a resolver because all of our application state is stored in a single immutable state tree (Store). As such a **guard** makes more sense for this scenario.

Scenario

Let's refactor our contacts application and allow deep linking by implementing a `ContactExistsGuard`. Use the UML sequence diagram below to help with your code challenge in this lab:

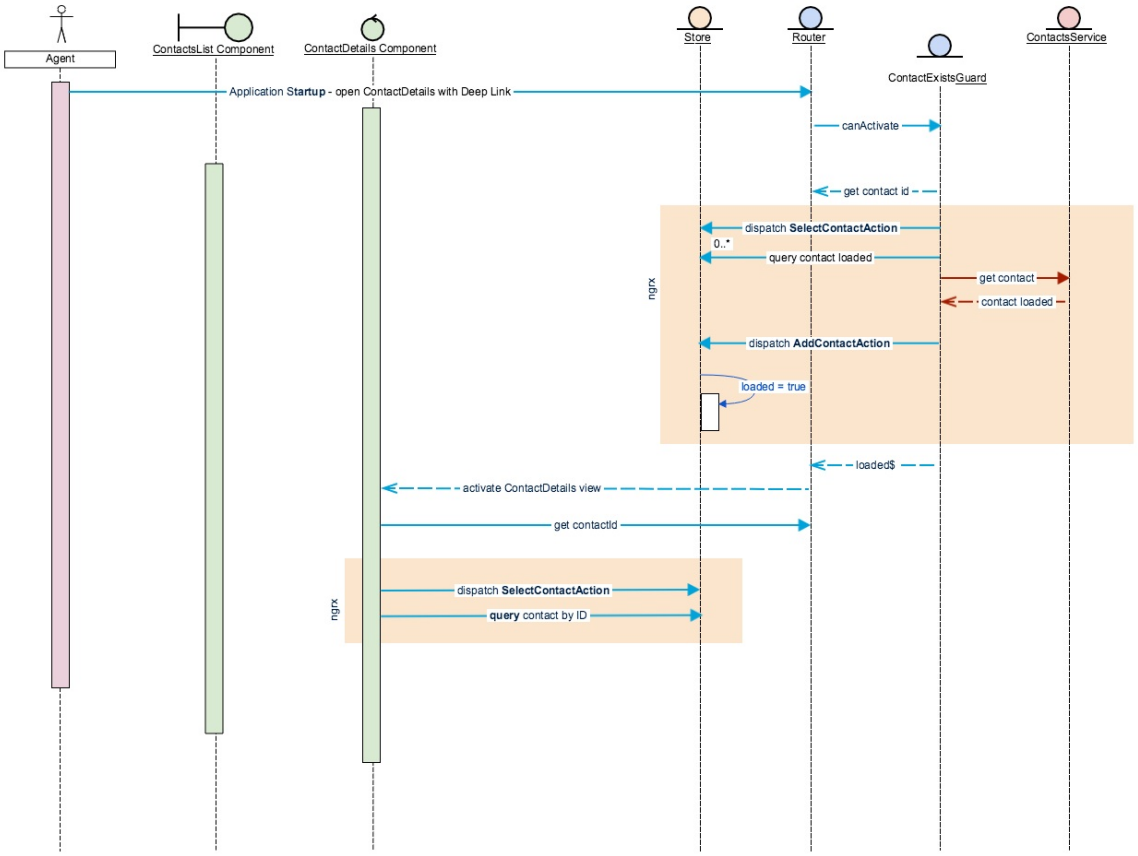


Figure: ngrx_contactdetail_router_guard

Tasks

1. Create a new action type for `ADD_CONTACT` and add it to the `ContactsActionTypes` in `contacts.actions.ts`.
2. Implement `AddContactAction` and use the previously created action type.
   - Make sure the payload of the action class is of type `Contact`.
   - Don't forget to add `AddContactAction` to the union type `ContactsActions` that is used inside the `ContactsReducer`.
3. Update `contacts-reducer.ts` to handle storing a single contact using the action type `ADD_CONTACT`:
   - First, check whether the contact is in the list. Tip: You can use `Array.find`.
   - If the contact is already in the list, do nothing and return a new array containing all contacts.
   - If the contact was not found, extend the list of contacts and add the new contact to the list. Tip: You can use the spread operator for array literals. Example: `[...myCollection, myItem]`. This creates a new list with all the items from `myCollection` plus an additional item `myItem`. If the contact was found, we don't have to do anything and we can return the list unmodified.
   - Add a new property `loaded` to the `ContactState` interface. This is used to determine if the list was already loaded. The type will be `boolean` and the inital value should be `false`. Set this property to `true` when `LoadContactsSuccessAction` was dispatched.
4. Create a `ContactExistsGuard` in `/src/app/contact-exists.guard.ts` and implement the `CanActivate` interface. You can do that using the Angular CLI: `ng g g contact-exists -m app`:
   - In the `canActivate()` method, dispatch `SelectContactAction` and give it the contact id from the `ActivatedRouteSnapshot` using `route.paramMap.get('id')`.
   - Select `loaded` from the store in order to check whether the list was already fetched. Make sure to chain `take(1)` to your stream which returns a specified number of contiguous events. In this case we are only interested in the `loaded` property **once**.

     > Remember that `store.pipe(select(<function>))` returns an observable.

   - Use `switchMap` to map to an inner observable which will be the results of `this.contactsService.getContact(contactId)`.
   - If you are calling `getContact` make sure to dispatch `AddContactAction` once the http call has been resolved.
   - Map the inner observable to a boolean using `.map(contact => !!contact)`.

     > This is necessary because the `canActivate` functions returns either a boolean, `Promise<boolean>` or `Observable<boolean>`.

5. Add `ContactExistsGuard` provider to the `providers` array of the `ContactsModule` in `app.module.ts` to make the guard available in your app. If you used the Angular CLI this should have been taken care of already. Though, it's better to verify this one more time to be safe.
6. Update the route configuration.
   - Add the guard to the `canActivate` array of the routes that show the `ContactsDetailComponent` and `ContactsEditorComponent`.
7. Update both `ContactsDetailComponent` and `ContactsEditorComponent` so that they don't dispatch the `SelectContactAction` anymore.

   > This is now handled by the guard.

Code Snippets

`/src/app/contact-exists.guard.ts`

```
@Injectable()
export class ContactExistsGuard implements CanActivate {

  constructor( public store: Store<ApplicationState>,
               private contactsService: ContactsService) { }

  canActivate(route: ActivatedRouteSnapshot) {
    const contactId        = route.paramMap.get('id');
    const addContactToList = (contact: Contact) => {
      this.store.dispatch(new AddContactAction(contact));
    };
    const loadContactByID  = () => {
      return this.contactsService.getContact(contactId).pipe(
          tap(addContactToList),
          map(contact => !!contact)
      );
    };

    this.store.dispatch(new SelectContactAction(+contactId));
    return this.store.pipe(
        select(state => state.contacts.loaded),
        take(1),
        switchMap(loaded => loaded ? of(true) : loadContactByID())
    );
  }
}
```

Figure: ngrxl3 1

Next Lab

Go to Step 4: Extract Selectors

Exercise: Refactor with Query Selectors & Entity Pattern

Our Contacts applications has selectors defined in almost every component including `ContactsListComponent`, `ContactsDetailComponent` and `ContactsEditorComponent`. That's unfortunate because it's not really maintainable, DRY and therefore not reusable if components share common selectors.

With ngrx > 4.x we can leverage efficient selector composition using the `createSelector` function. As a result, a selector is not recomputed unless one of its arguments change. This practice is known as memoization. In addition, selectors are composable. They can be used as input to other selectors.

> A **selector** in its raw form is just a **function** that takes the application state and returns a value (e.g. contact). Used with ngrx Store, selectors are queries that return observables to values.

In addition, we want to optimize the data structure of our Store by using the Entity Pattern. This allows us to perform fast CRUD operations and simplify our reducers.

Scenario

In this lab, we want to extract all of our selectors from the components into `contacts.reducer.ts` and compose them into a single query `ContactsQuery` to make them reusable, easier to maintain, and reduce components' responsibility. We also need to convert the Array of contacts into an entity object.

> Note: if a **store** is analogous to a database and **reducers** the tables, then **selectors** can be considered the queries into said database/store.

Tasks

1. Change `ContactState` and refactor the `list` property to be `entities`.
   - Rename the property to `entities` and set the type to `{ [key: number]: Contact }`.
   - Don't forget to update `INITIAL_STATE`.
2. Update the `contactsReducer` so that it works with **entities** instead of an Array of contacts.
   - Specifically, you need to update `LOAD_CONTACTS_SUCCESS`, `ADD_CONTACT` and `UPDATE_CONTACT`.
   - For `LOAD_CONTACTS_SUCCESS`, make sure to flatten the Array into entities. Hint: You can use `Array.reduce`.
3. Create a `contacts.selectors.ts` file with a `namespace ContactsQuery`:

```
export namespace ContactsQuery {
  export const getLoaded = (state: ApplicationState) => state.contacts.loaded;
  ...
}
```

4. Add the following queries: `getContactsEntities`, `getSelectedContactId`.
5. Use the `createFeatureSelector()` to load the NgRx state for `contacts`:

```
// Lookup the 'Contacts' feature state managed by NgRx
export const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);

export const getContactsEntities = createSelector(getContactsState, (state:ContactsState) => state.entities);
```

1. Implement the following composed queries using `createSelector`:

- `getContacts` and
- `getSelectedContact`

```
export const getContacts = createSelector(getContactsEntities, (entities) => {
...
});

export const getSelectedContact = createSelector(getContactsEntities, getSelectedContactId,
(contactEntities, id) => {
  ...
});
```

2. Use `ContactsQuery` selectors in `ContactsListComponent`.
3. Use `ContactsQuery` selectors in `ContactsDetailComponent`.
4. Use `ContactsQuery` selectors in `ContactsEditorComponent`. Remember to **clone** the contact **after** you selected it. You can use `map` for this.
5. Use `ContactsQuery` selectors in `ContactExistsGuard`.

Code Snippets

`contacts.reducer.ts`

```
import { Contact } from '../../models/contact';
import { ContactsActionTypes, ContactsActions } from '../contacts/contacts.actions';

export const FEATURE_CONTACTS = 'contacts';

export interface ContactsState {
  entities: { [key: number]: Contact };
  selectedContactId: number | null;
  loaded: boolean;
}

export const INITIAL_STATE: ContactsState = {
  entities: {},
  selectedContactId: null,
  loaded: false
};
```

Figure: ngrx4 1

```
export function contactsReducer(state: ContactsState = INITIAL_STATE, action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      // CODE MISSING HERE ;-)
    case ContactsActionTypes.SELECT_CONTACT:
      return {
        ...state,
        selectedContactId: action.payload
      };
    case ContactsActionTypes.ADD_CONTACT:
      const inStore = state.entities[action.payload.id];

      return inStore ? state : {
        ...state,
        entities: {
          ...state.entities,
          [action.payload.id]: action.payload
        }
      };
    case ContactsActionTypes.UPDATE_CONTACT:
      return {
        ...state,
        entities: {
          ...state.entities,
          [action.payload.id]: action.payload
        }
      };
  }
  return state;
}
```

Figure: ngrx4 4

`contacts.selectors.ts`

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { ContactsState, FEATURE_CONTACTS } from './contacts.reducer';

export namespace ContactsQuery {
  // Lookup the specific 'Contacts' feature data/slice managed by NgRx
  export const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);


  export const getLoaded = createSelector(
    getContactsState, (state:ContactsState) => state.loaded
  );
  export const getSelectedContactId = createSelector(
    getContactsState, (state:ContactsState) => state.selectedContactId
  );


  export const getContactsEntities = createSelector(
    getContactsState, (state:ContactsState) => state.entities
  );
  export const getContacts = createSelector(
    getContactsEntities, entities => Object.keys(entities).map(id => entities[id])
  );

  export const getSelectedContact = createSelector(
    getContactsEntities, getSelectedContactId, (contactEntities, id) => {
        return contactEntities[id];
  });

}
```

Figure: ngrx4 2

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {

  contacts$: Observable<Array<Contact>> = this.store.pipe(select(ContactsQuery.getContacts));;

  constructor(private store: Store<ApplicationState>, private contactsService: ContactsService) { }

  ngOnInit() {
    this.contactsService.getContacts().subscribe(contacts => {
      this.store.dispatch(new LoadContactsSuccessAction(contacts))
    });
  }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Figure: ngrx4 3

Next Lab

Go to Step 5: Apply Meta-Reducers

Exercise: Using Effects for Asynchronous Actions

In **ngrx**-land, we can use the **@Effect** decorators to solve the problems of asynchronous activity. Effects allow developers to centralize all asynchronous work. When each async activity completes then synchronous actions are dispatched.

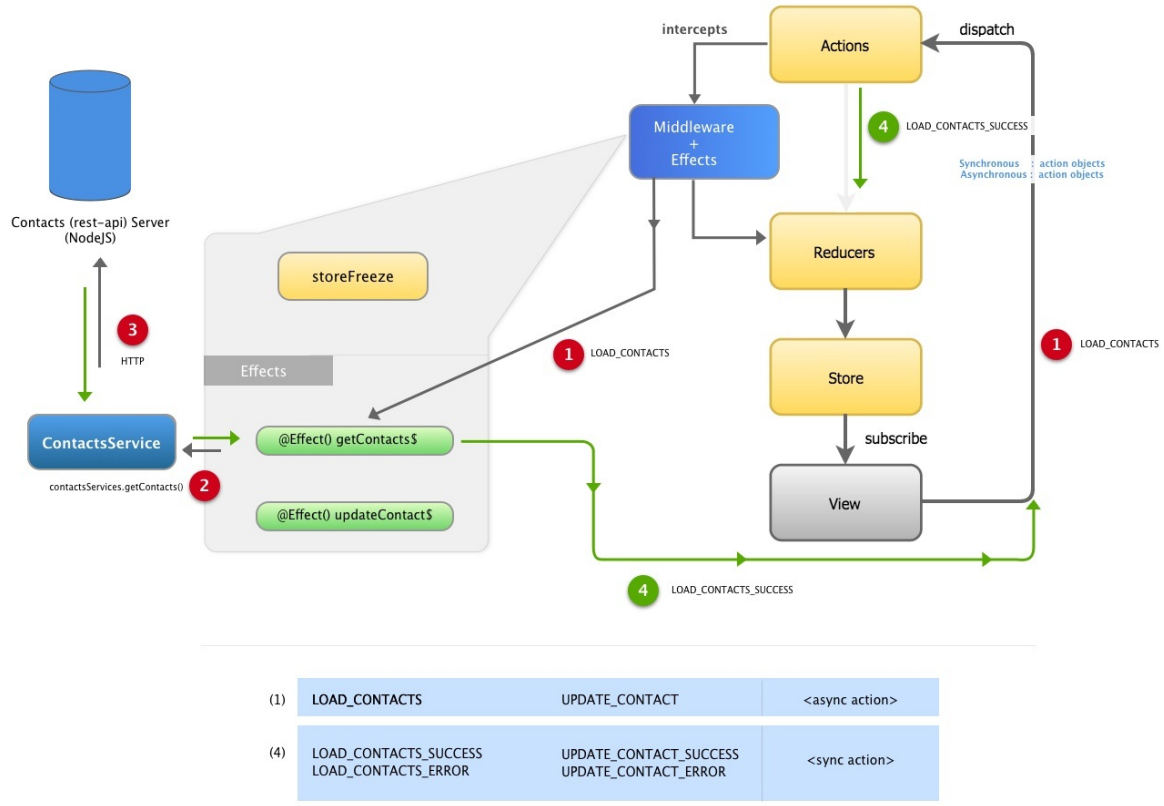
- Effects should be considered **side effect** or **background processes** within an application.
- And synchronous actions can be considered the **foreground processes**.

Scenario

Let's use Effects for the async actions `LOAD_CONTACTS` and `UPDATE_CONTACT`. These async actions are caught by Effects. When the async work completes, the Effect will internally dispatch sync actions to update the store, e.g. `UpdateContactSuccessAction`.

> Unlike Redux, the async and sync actions [in ngrx] are still PJO(s) or class instances. The object/instances with specific action types will be directed to the Effect that has registered to listen to that action type.

Use the following diagram to refactor your application with the ngrx `@Effect()` decorator by the example of loading the list of contacts. Then apply the same concept of Effects to edit a particular contact.



(1)	LOAD_CONTACTS	UPDATE_CONTACT	<async action>
(4)	LOAD_CONTACTS_SUCCESS LOAD_CONTACTS_ERROR	UPDATE_CONTACT_SUCCESS UPDATE_CONTACT_ERROR	<sync action>

Figure: step-6-async-with-effects

Tasks

1. Define more ContactActions in `contacts.actions.ts`:
   - Add two (2) more action types to `ContactsActionTypes`:
     - `LOAD_CONTACTS`, and
     - `UPDATE_CONTACT_SUCCESS`
     - We already have an `UPDATE_CONTACT` action so we don't have to create this one
   - Implement `LoadContactsAction` and `UpdateContactSuccessAction`.
   - Add both actions to `ContactsActions` to extend the union type.
2. Update `ContactsReducer` and change the action type from `UPDATE_CONTACT` to `UPDATE_CONTACT_SUCCESS` keeping the same logic.

   > Remember that we don't have to listen for `UPDATE_CONTACT` nor `LOAD_CONTACTS` because those actions will only trigger our Effects and can be seen as actions that represent async activity. Inside our reducers we only handle actions that represent synchronous activity.

3. Create a service `ContactsEffects` with the Angular CLI and add the following imports:

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { Actions, Effect, ofType } from '@ngrx/effects';
import { Observable } from 'rxjs';

import { ContactsService } from '../../contacts.service';
import { Contact } from '../../models/contact';

import {
  ContactsActionTypes,
  LoadContactsSuccessAction,
  UpdateContactSuccessAction
} from '../contacts/contacts.actions';
```

4. Update the constructor and inject the following dependencies to the Effect class:

```
private actions$: Actions,
private contactsService: ContactsService,
private router: Router
```

5. Implement our Effects using the `@Effect()` decorator and the action stream `actions$`.

   Implement the `getContacts$` effect:

```
@Effect() getContacts$ = this.actions$.pipe(
  ofType(ContactsActionTypes.LOAD_CONTACTS),
  switchMap(payload => this.contactsService.getContacts()),
  map((contacts: Array<Contact>) => new LoadContactsSuccessAction(contacts))
);
```

   Implement the `updateContact$` effect:

```
@Effect() updateContact$ = this.actions$.pipe(
  ofType( <UPDATE_CONTACT_ACTION_TYPE> )
  map((action: UpdateContactAction) => action.payload),
  contactMap((contact: Contact) => <UPDATE_CONTACT_ON_SERVER> ),
  tap((contact: Contact) => this.router.navigate(['/contact', contact.id])),
  map((contact: Contact) => <UPDATE_CONTACT_SUCCESS_ACTION> )
);
```

- The `LOAD_CONTACTS` action is used to trigger the Effect to fetch the list of contacts from the remote database.
- The `UPDATE_CONTACT` action triggers the Effect which will update the contact in the remote database.

6. Update `ContactsListComponent` and `ContactsEditorComponent`.
   - Remove all direct service calls to `this.contactsService.getContacts()`
   - Remove all direct service calls to `this.contactsService.updateContact(contact)`
   - Dispatch async actions, e.g. `LoadContactsAction` or `UpdateContactAction` to trigger the corresponding Effects.
7. Update `ContactsModule` in `app.module.ts` and add the following imports:

```
import { EffectsModule } from '@ngrx/effects';
import { ContactsEffects } from './state-management/contacts/contacts.effects';
```

8. Add the `EffectsModule` to the imports of the `ContactsModule` and call `forRoot`.

> This function expects an array of Effect classes as its first argument (`[ContactsEffect]`).

Code Snippets

`contacts.effects.ts`

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';

import { Actions, Effect, ofType } from '@ngrx/effects';
import { map, exhaustMap, tap, concatMap } from 'rxjs/operators';

import { ContactsService } from '../../contacts.service';
import { Contact } from '../../models/contact';

import {
  ContactsActionTypes,
  LoadContactsSuccessAction,
  UpdateContactSuccessAction,
  UpdateContactAction
} from './contacts.actions';

@Injectable()
export class ContactsEffects {
  @Effect() getContacts$ = this.actions$.pipe(
    ofType(ContactsActionTypes.LOAD_CONTACTS),
    exhaustMap(payload => this.contactsService.getContacts()),
    map((contacts: Array<Contact>) => new LoadContactsSuccessAction(contacts))
  );

  @Effect() updateContact$ = this.actions$.pipe(
    ofType(ContactsActionTypes.UPDATE_CONTACT),
    map((action: UpdateContactAction) => action.payload),
    concatMap((contact: Contact) => this.contactsService.updateContact(contact)),
    tap((contact: Contact) => this.router.navigate(['/contact', contact.id])),
    map((contact: Contact) => new UpdateContactSuccessAction(contact))
  );

  constructor(
    private actions$: Actions,
    private contactsService: ContactsService,
    private router: Router) {
  }
}
```

Figure: ngrx5 1

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent {
  contacts$: Observable<Array<Contact>> = this.store.pipe(select(ContactsQuery.getContacts));

  constructor(private store: Store<ApplicationState>) {
    this.store.dispatch(new LoadContactsAction());
  }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Figure: ngrx5 2

`contacts-ngrx.module.ts`

```
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forRoot({}, {
      metaReducers: !environment.production ? [storeFreeze] : []
    }),
    EffectsModule.forRoot([]),

    StoreModule.forFeature(
      FEATURE_CONTACTS,
      contactsReducer,
      { initialState: contactsInitialState }
    ),
    EffectsModule.forFeature([ContactsEffects]),

    !environment.production ? StoreDevtoolsModule.instrument() : [],
  ]
})
export class ContactsNgRxModule {}
```

Figure: ngrx5 3

Next Lab

Go to Step 7: Refactor to Facade Architecture

Technical Considerations

In our Effects above, we utilize `concatMap` to map the action payload to an inner Observable that performs an http request, e.g. `this.contactsService.getContacts()` or `this.contactsService.updateContact(contact)`. Here, it makes sense to use `contactMap` over `switchMap` because we don't want to cancel previous update requests but rather wait for them to finish before another update action is sent.

It is important to note that the **updateContact$** effect not only saves the contact it also actually routes to the contact details view. To navigate back to the details page, use the `tap` operator to perform user navigation. Notice, the **`tap`** operator only executes a callback for each event without manipulating the stream.

Exercise: Refactor to use Facade Architecture

Effects decorators are not obvious processes. Another approach to managing async activities is to use a facade class to expose the properties desired and publish methods that hide all Store and RESTful server interactions.

Since this facade class publishes properties as **Observables** we can use reactive features and the **async** pipe in our templates; and the templates will re-render when their associated observables receive emitted values.

Scenario

Use the illustration below to guide you through the lab steps:

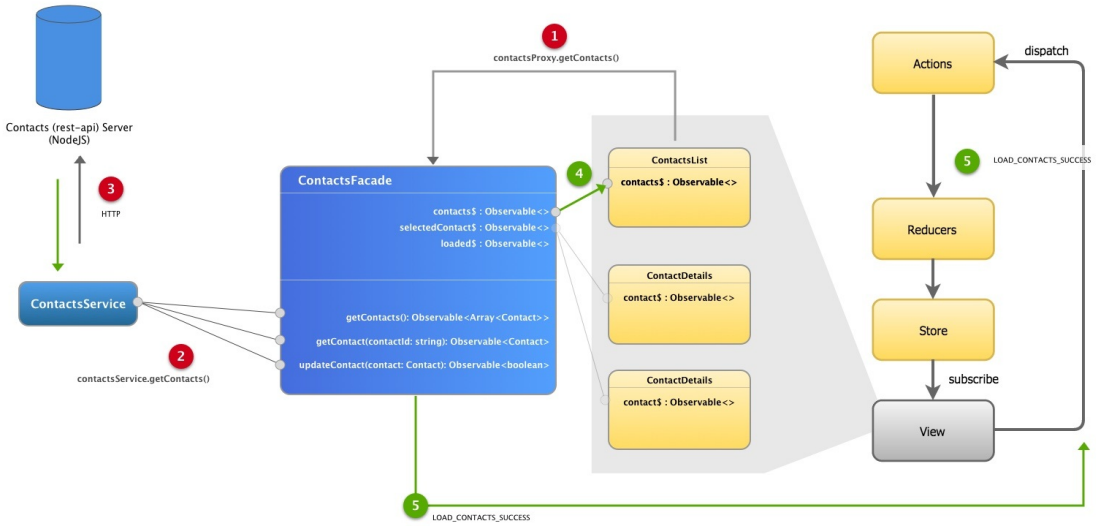


Figure: lab7-using-facades

Let's implement and refactor our application to use a `ContactsFacade` class like this:

```
@Injectable()
export class ContactsFacade {
  // Exposed selectors
  contacts$: Observable<Array<Contact>>;
  selectedContact$: Observable<Contact>;
  loaded$: Observable<boolean>;

  constructor(
    private store: Store<ApplicationState>,
    private contactsService: ContactsService
  ) { }

  // Exposed public APIs
  getContactById(contactId: string): Observable<Contact> { ... }
  updateContact(contact: Contact): Observable<boolean> { ... }
}
```

Tasks

1. Implement the class `ContactsFacade` in `state/contacts/contacts.facade.ts` as shown above.
    - For the Observable properties, use `this.store.pipe(select( <QUERY> ))` to initialize each property with the appropriate Observable.

      ```
      contacts$ = this.store.pipe(select( <GET_CONTACTS_QUERY> ));
      selectedContact$ = this.store.pipe(select( <GET_SELECTED_CONTACT_QUERY> ));
      loaded$ = this.store.pipe(select( <GET_IS_LOADED_QUERY> ));
      ```

````
For `getContact()`, use this as your starter code template:

```js
getContactById(contactId:string):Observable<Contact> {
  // Select contact id >

  return this.loaded$.pipe(
    take(1),
    withLatestFrom(this.selectedContact$),   // Get latest value from selectedContact$ stream
    mergeMap(([loaded, selectedContact]) => {
      if (loaded) return Observable.of(null);

      return this.contactsService
        .getContact(contactId)
        .pipe(tap( <ADD_CONTACT_TO_LIST> ));
    }),
    // Map to contact$ stream
    mergeMap(() => this.selectedContact$)
  );
}
```

For `updateContact()`, use this as your starter code template:

```js
updateContact(contact: Contact): Observable<boolean> {
  return this.contactsService
    .updateContact(contact).pipe(
        map(() => {
          <UPDATE_CONTACT_ACTION>
          return true;
        }),
        catchError(() => Observable.of(false))
    );
}
```
````

1. Update the `ContactsModule` in `app.module.ts`:
    - Register a provider for `ContactsFacade`.
2. Update the `ContactExistGuard`, `ContactsListComponent`, `ContactsDetailComponent` and `ContactsEditorComponent` to use `ContactsFacade`.
    - Remove use of the `ContactsService`, `ApplicationState`, store, and actions
3. Update the `ContactsActions` to remove deprecated actions
    - Remove **LoadContactsAction**
    - Remove **UpdateContactAction**

Code Snippets

`contacts.facade.ts`

```
@Injectable()
export class ContactsFacade {
  loaded$ = this.store.pipe(select(ContactsQuery.getLoaded));
  contacts$ = this.store.pipe(select(ContactsQuery.getContacts));
  selectedContact$ = this.store.pipe(select(ContactsQuery.getSelectedContact));

  constructor(private store: Store<Schema>,
              private contactsService: ContactsService) {
    this.store.dispatch(new LoadContactsAction());
  }

  updateContact(contact: Contact): Observable<boolean> {
    return this.contactsService.updateContact(contact).pipe(
        map(() => {
          this.store.dispatch(new UpdateContactSuccessAction(contact));

          return true;
        }),
        catchError(() => of(false))
    );
  }

  getContactById(contactId: number): Observable<Contact> {
    this.selectContact(contactId);

    return this.loaded$.pipe(
        take(1),
        withLatestFrom(this.selectedContact$),
        switchMap(([loaded, selectedContact]) => {
          let addContactToList = (contact: Contact) => {
            if (!selectedContact) {
              this.store.dispatch(new AddContactAction(contact));
            }
          };

          let getContact = (id: number) => this.contactsService
              .getContact(contactId.toString())
              .pipe(tap(addContactToList));

          return loaded ? of(null) : getContact(contactId);
        }),
        switchMap(() => this.selectedContact$)
    );
  }


  private selectContact(id: number): void {
    this.store.dispatch(new SelectContactAction(id));
  }
}
```

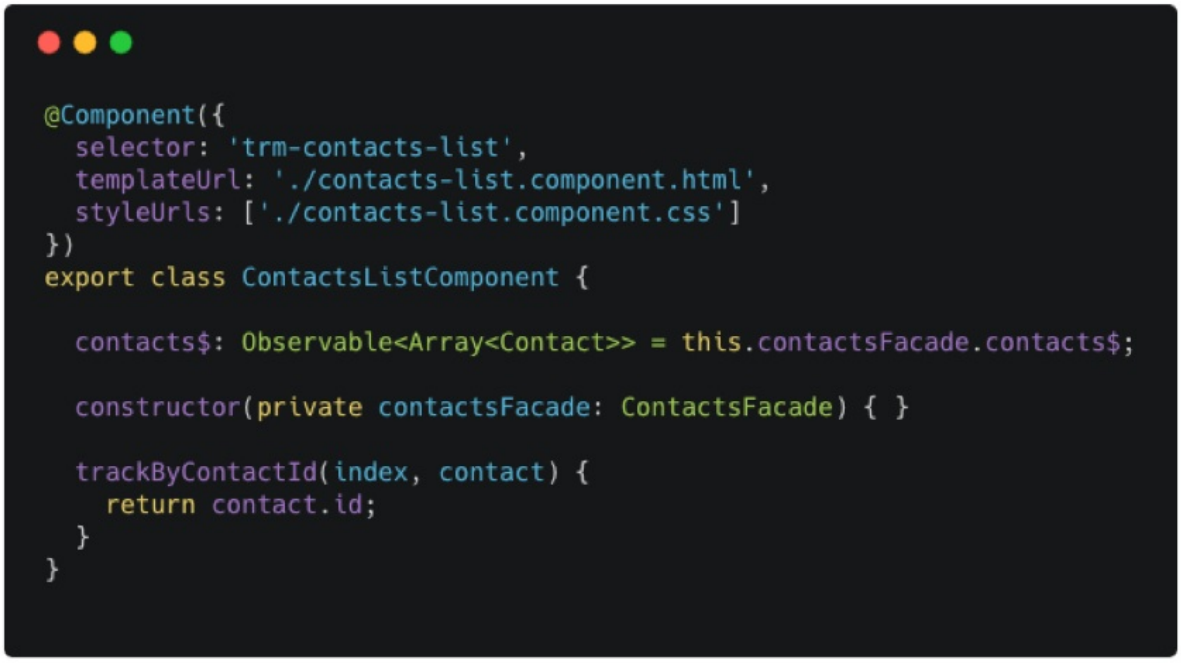
Figure: ngrx6 1

`contact-exists.guard.ts`

```
@Injectable()
export class ContactExistsGuard implements CanActivate {

  constructor(private contactsFacade: ContactsFacade) { }

  canActivate(route: ActivatedRouteSnapshot) {
    const contactId = route.paramMap.get('id');
    const contact$ = this.contactsFacade.getContactById(+contactId);

    return contact$.pipe(map(contact => !!contact));
  }
}
```

Figure: ngrx6 2

`contacts-list.component.ts`



```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent {

  contacts$: Observable<Array<Contact>> = this.contactsFacade.contacts$;

  constructor(private contactsFacade: ContactsFacade) { }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Figure: ngrx6 3

Next Lab

Go to Lab 7: Improve your Effects

Exercise: Improve Effects with Deciders/Splitters

Effects can also be used to:

- Split 1 action into multiple actions; to spawn 1..n other actions
- Decide on *output action(s)* based on input content or biz logic

Scenario

Let's improve our `contacts.effects.ts` to improve our logic to load all contacts. Now the Effect should contain the logic to check `facade.loaded$` and decide if the all contacts should be loaded from the server or if the action should be ignored if that list was already loaded.

Let's also list for `LOAD_CONTACT_DETAILS` to BOTH (1) select the contactId and (2) load the contact details.

Tasks

1. In **`contacts.actions.ts`** create
   - a `class NoopAction` action class that does not have a payload.
   - a `class LoadContactAction` to load a contact information (used in Content-based Decider below)
   - a `class LoadContactDetailsAction` class that requests to load details of a specific contact. (used in Action Splitter below)
2. In **`contacts.effects.ts`** create an
   - **Content-Based Action Decider** `@Effect() getContact$` that checks `facade.loaded$` to determine if the action should be ignored (and return the NoopAction) or if the contact should be loaded (using the contactsService).
   - **Action Splitter** `@Effect() getContactDetails$` that will emit 2 actions: `SelectContactAction` and `LoadContactAction`
3. Make sure all view components are using ONLY the ContactsFacade observables and methods; remove all uses of Store and ContactsService
4. Update **`ContactsFacade::getContactById(contactId)`** to dispatch actions to use these new effects.

Code Snippets

`contacts.effect.ts`

```
@Injectable()
export class ContactsEffects {

  @Effect() getContacts$ = this.actions$.pipe(
      ofType(ContactsActionTypes.LOAD_CONTACTS),
      exhaustMap(payload => this.contactsService.getContacts()),
      map((contacts: Array<Contact>) => new LoadContactsSuccessAction(contacts))
  );

  // Action Decider (Splitter)
  // It map's one action to an array of actions
  @Effect() getContactDetails$ = this.actions$.pipe(
      ofType(ContactsActionTypes.LOAD_CONTACT_DETAILS),
      map((action: LoadContactDetailsAction) => action.payload),
      mergeMap(contactId => [
        new SelectContactAction(+contactId),
        new LoadContactAction(+contactId)
      ])
  );

  // Action Decider (Content-Based Decider)
  // A content-based decider uses the payload of an action to map it to a different action
  @Effect() getContact$ = this.actions$.pipe(
      ofType(ContactsActionTypes.LOAD_CONTACT),
      map((action: LoadContactAction) => action.payload),
      withLatestFrom(this.facade.loaded$, this.facade.selectedContact$),
      switchMap(([contactId, loaded, selectedContact]) => {
        const contactLoaded = selectedContact && selectedContact.id === +contactId;

        return loaded || contactLoaded
            ? of(null)
            : this.contactsService.getContact(contactId.toString());
      }),
      map((contact: Contact | null) => {
        return contact
            ? new AddContactAction(contact)
            : new NoopAction();
      })
  );


  @Effect() updateContact$ = this.actions$.pipe(
    ofType(ContactsActionTypes.UPDATE_CONTACT),
    map((action: UpdateContactAction) => action.payload),
    concatMap((contact: Contact) => this.contactsService.updateContact(contact)),
    tap((contact: Contact) => this.router.navigate(['/contact', contact.id])),
    map((contact: Contact) => new UpdateContactSuccessAction(contact))
  );

  constructor(
      private actions$: Actions,
      private contactsService: ContactsService,
      private router: Router,
      private facade: ContactsFacade) {
  }
}
```

Figure: ngrx8 1

`contacts.facade.ts`

```
@Injectable()
export class ContactsFacade {
  loaded$          = this.store.pipe(select(ContactsQuery.getLoaded));
  contacts$        = this.store.pipe(select(ContactsQuery.getContacts));
  selectedContact$ = this.store.pipe(select(ContactsQuery.getSelectedContact));

  constructor(private store: Store<Schema>,
              private contactsService: ContactsService) {
    this.loadContacts();
  }

  loadContacts(): Observable<Array<Contact>> {
    this.store.dispatch(new LoadContactsAction());
    return this.contacts$;
  }

  getContactById(contactId: number): Observable<Contact> {
    this.store.dispatch(new LoadContactDetailsAction(String(contactId)));
    return this.selectedContact$;
  }

  updateContact(contact: Contact): Observable<boolean> {
    this.store.dispatch(new UpdateContactAction(contact));
    return this.loaded$;
  }

}
```

Figure: ngrx8 3

`contacts.actions.ts`

```
import { Action } from '@ngrx/store';
import { Contact } from '../../models/contact';

export enum ContactsActionTypes {
  NOOP = '[Contacts] Ignore Action',

  LOAD_CONTACTS = '[Contacts] Load Contacts',
  LOAD_CONTACTS_SUCCESS = '[Contacts] Load Contacts Success',

  LOAD_CONTACT = '[Contacts] Load Contact',
  LOAD_CONTACT_DETAILS = '[Contacts] Load Contact Details',
}

export class NoopAction implements Action {
  readonly type = ContactsActionTypes.NOOP;
}

export class LoadContactsAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACTS;
}

export class LoadContactsSuccessAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACTS_SUCCESS;
  constructor(public payload: Array<Contact>) {}
}


export class LoadContactDetailsAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACT_DETAILS;
  constructor(public payload: string) { }
}

export class LoadContactAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACT;
  constructor(public payload: number) { }
}
```

Figure: ngr9 5

Next Lab

Go to Lab 8: Use @ngrx/entity

Exercise: Using @ngrx/entity

@ngrx/entity provides utilties to handle CRUD operations and queries for your collections.

Scenario

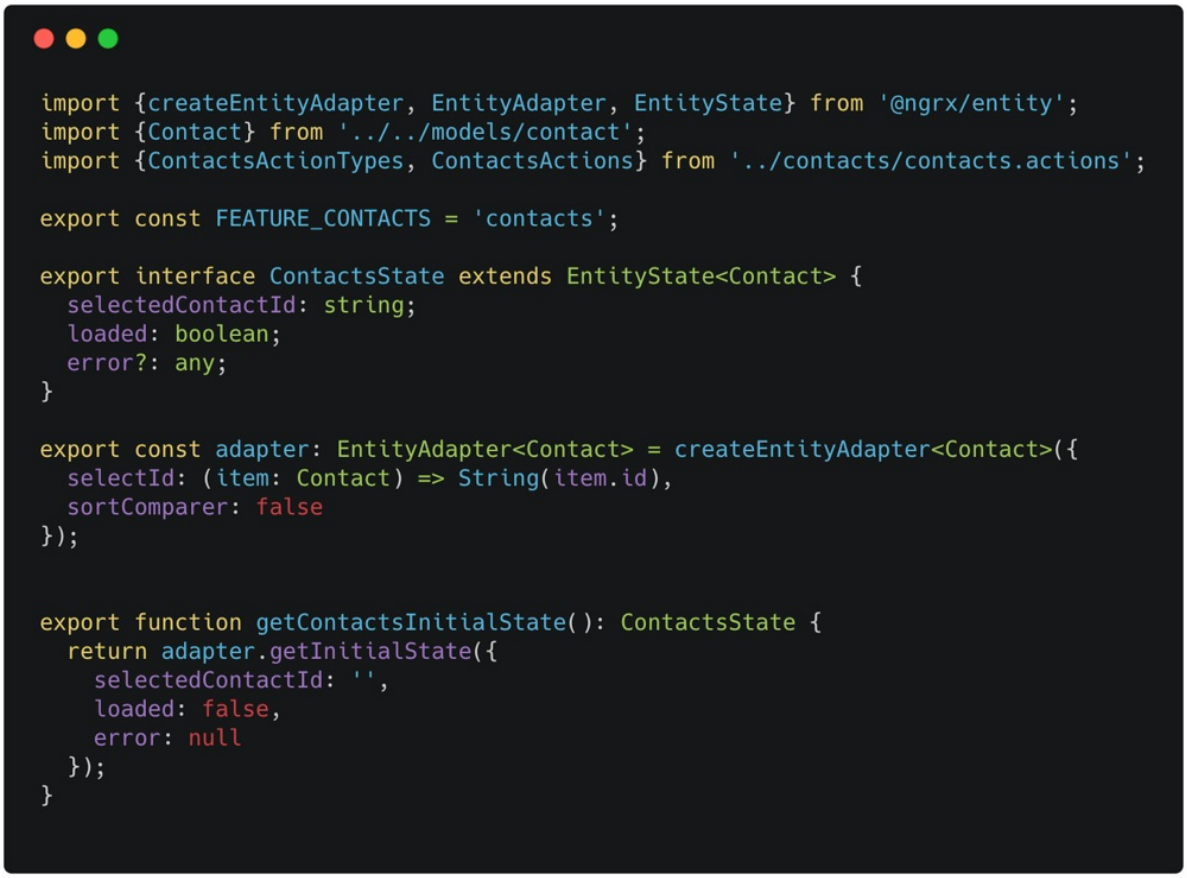
Let's change our `contacts.reducer.ts` to use @ngrx/entity adapters and EntityState. Then update your `contacts.selectors` to use @ngrx/entity selectors.

Tasks

1. In **`contacts.reducer.ts`**
   - use EntityState and EntityAdapter to define your feature state.
   - use `adapter.getInitialState()` to build your initial state
   - update your reducer to use `adapter.addAll()` and `adapter.upsertOne()`
2. Update your **`contacts.selectors`**
   - use `adapter.getSelectors()`
3. Update your **`contacts-ngrx.module.ts`** use the `getContactsInitialState` function to build the initialState

Code Snippets

`contacts.reducer.ts`



```
import {createEntityAdapter, EntityAdapter, EntityState} from '@ngrx/entity';
import {Contact} from '../../models/contact';
import {ContactsActionTypes, ContactsActions} from '../contacts/contacts.actions';

export const FEATURE_CONTACTS = 'contacts';

export interface ContactsState extends EntityState<Contact> {
  selectedContactId: string;
  loaded: boolean;
  error?: any;
}

export const adapter: EntityAdapter<Contact> = createEntityAdapter<Contact>({
  selectId: (item: Contact) => String(item.id),
  sortComparer: false
});


export function getContactsInitialState(): ContactsState {
  return adapter.getInitialState({
    selectedContactId: '',
    loaded: false,
    error: null
  });
}
```

Figure: ngrx9 1

```
import {createEntityAdapter, EntityAdapter, EntityState} from '@ngrx/entity';
import {Contact} from '../../models/contact';
import {ContactsActionTypes, ContactsActions} from '../contacts/contacts.actions';


export function contactsReducer(state: ContactsState, action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      return {
          ...adapter.addAll(action.payload, state),
        loaded : true
      };

    case ContactsActionTypes.SELECT_CONTACT:
      return {
        ...state,
        selectedContactId: action.payload
      };

    case ContactsActionTypes.ADD_CONTACT:
    case ContactsActionTypes.UPDATE_CONTACT_SUCCESS:
     return adapter.upsertOne(action.payload,state);
  }

  return state;
}
```

Figure: ngrx9 2

`contacts.selectors.ts`

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import {adapter, ContactsState, FEATURE_CONTACTS} from './contacts.reducer';

const {
  selectAll,
  selectEntities,
} = adapter.getSelectors();

export namespace ContactsQuery {

  const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);

    // Use @ngrx/entity selectors
  const getContactsEntities = createSelector(getContactsState, selectEntities);
  export const getContacts = createSelector(getContactsState, selectAll);

  export const getSelectedContactId = createSelector(getContactsState, (state:ContactsState) =>{
    return state.selectedContactId
  });
  export const getSelectedContact = createSelector(
    getContactsEntities, getSelectedContactId, (contactEntities, id) => {
        return contactEntities[id];
  });

  export const getLoaded = createSelector(getContactsState, (state:ContactsState) => state.loaded);
}
```

Figure: ngrx9 3

`contacts-ngrx.module.ts`

```
import {
  getContactsInitialState,
  contactsReducer,
  FEATURE_CONTACTS
} from './state/contacts/contacts.reducer';

@NgModule({
  imports: [
    StoreModule.forFeature(
      FEATURE_CONTACTS,
      contactsReducer,
      { initialState: getContactsInitialState }
    ),
  ],
})
export class ContactsNgRxModule {}
```

Figure: ngr9 4

Next Lab

Congratulations! You are done.

Exercise: Restoring Search

With all the NgRx features add, you may have noticed that the *User Search* feature no longer works. Even if we fixed the current RTE, we have an issue with state management.

> When navigating between the list and a contact details, we currently lose the search criteria last used.

For this exercise, we need to:

- support search for users by partial-name searching,
- save/restore the search criteria whenever we show the contact list
- filter the contacts list to show only the matches to the in-memory, partial-name search

Scenario

For this exercise, let's:

- enhance our NgRx to manage `searchCriteria` state as part of the `contacts` NgRx features.
- use a Reactive FormControl provide an observable stream of input search changes.
- update the ContactsFacade to provide a `searchCriteria$` observable and a `search()` feature
- listen for input changes and call `ContactFacade::search()`
- use the `ContactsFacade::searchCriteria$` to initialize the search input field with the current searchCriteria value.
- auto-focus the Search Users input field when activating the ContactsList view.

Tasks

1. In **`contacts.reducer.ts`**
   - use EntityState and EntityAdapter to define your feature state.
   - use `adapter.getInitialState()` to build your initial state
   - update your reducer to use `adapter.addAll()` and `adapter.upsertOne()`
2. Update your **`contacts.selectors`**
   - use `adapter.getSelectors()`
3. Update your **`contacts-ngrx.module.ts`** use the `getContactsInitialState` function to build the initialState

Code Snippets

`contacts.reducer.ts`

```
import {createEntityAdapter, EntityAdapter, EntityState} from '@ngrx/entity';
import {Contact} from '../../models/contact';
import {ContactsActionTypes, ContactsActions} from '../contacts/contacts.actions';

export const FEATURE_CONTACTS = 'contacts';

export interface ContactsState extends EntityState<Contact> {
  selectedContactId: string;
  loaded: boolean;
  error?: any;
}

export const adapter: EntityAdapter<Contact> = createEntityAdapter<Contact>({
  selectId: (item: Contact) => String(item.id),
  sortComparer: false
});


export function getContactsInitialState(): ContactsState {
  return adapter.getInitialState({
    selectedContactId: '',
    loaded: false,
    error: null
  });
}
```

Figure: ngrx9 1

```
import {createEntityAdapter, EntityAdapter, EntityState} from '@ngrx/entity';
import {Contact} from '../../models/contact';
import {ContactsActionTypes, ContactsActions} from '../contacts/contacts.actions';


export function contactsReducer(state: ContactsState, action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      return {
          ...adapter.addAll(action.payload, state),
        loaded : true
      };

    case ContactsActionTypes.SELECT_CONTACT:
      return {
        ...state,
        selectedContactId: action.payload
      };

    case ContactsActionTypes.ADD_CONTACT:
    case ContactsActionTypes.UPDATE_CONTACT_SUCCESS:
     return adapter.upsertOne(action.payload,state);
  }

  return state;
}
```

Figure: ngrx9 2

`contacts.selectors.ts`

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import {adapter, ContactsState, FEATURE_CONTACTS} from './contacts.reducer';

const {
  selectAll,
  selectEntities,
} = adapter.getSelectors();

export namespace ContactsQuery {

  const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);

    // Use @ngrx/entity selectors
  const getContactsEntities = createSelector(getContactsState, selectEntities);
  export const getContacts = createSelector(getContactsState, selectAll);

  export const getSelectedContactId = createSelector(getContactsState, (state:ContactsState) =>{
    return state.selectedContactId
  });
  export const getSelectedContact = createSelector(
    getContactsEntities, getSelectedContactId, (contactEntities, id) => {
        return contactEntities[id];
  });

  export const getLoaded = createSelector(getContactsState, (state:ContactsState) => state.loaded);
}
```

Figure: ngrx9 3

`contacts-ngrx.module.ts`

```
import {
  getContactsInitialState,
  contactsReducer,
  FEATURE_CONTACTS
} from './state/contacts/contacts.reducer';

@NgModule({
  imports: [
    StoreModule.forFeature(
      FEATURE_CONTACTS,
      contactsReducer,
      { initialState: getContactsInitialState }
    ),
  ],
})
export class ContactsNgRxModule {}
```

Figure: ngr9 4

Next Lab

Congratulations! You are done.

Architecture

Let's explore Angular component architectures

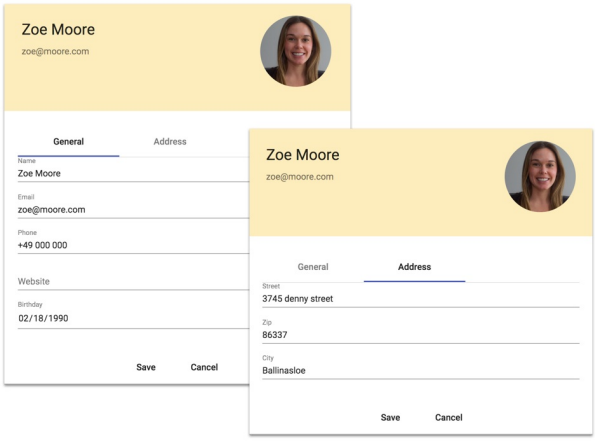


Figure: ardme

Lab Exercises

- Lab #1: Smart + Dumb Components
- Lab #2: Parent Injection into Children
- Lab #3: Using ContentChildren Query
- Lab #4: App Global Communications

Lab 1: Refactor Details to presentation and view components

So far, all the components we've built are what's often referred to as *Business Components* or *Smart Components*. "Smart" in the sense that they know a lot about their environment. They consume business services depend on routing and more.

There's nothing wrong about this category of components. However, these components are often not really reusable. What if we would like to display contact details somewhere else in our app? Maybe embedded in another view with a different URL? That's not possible with it's current design.

In this exercise we want to learn how to refactor our *component* into:

- a *Presentation Component* that strictly communicates via inputs and outputs
- a *View Component* that embeds and controls the *Presentation Component*

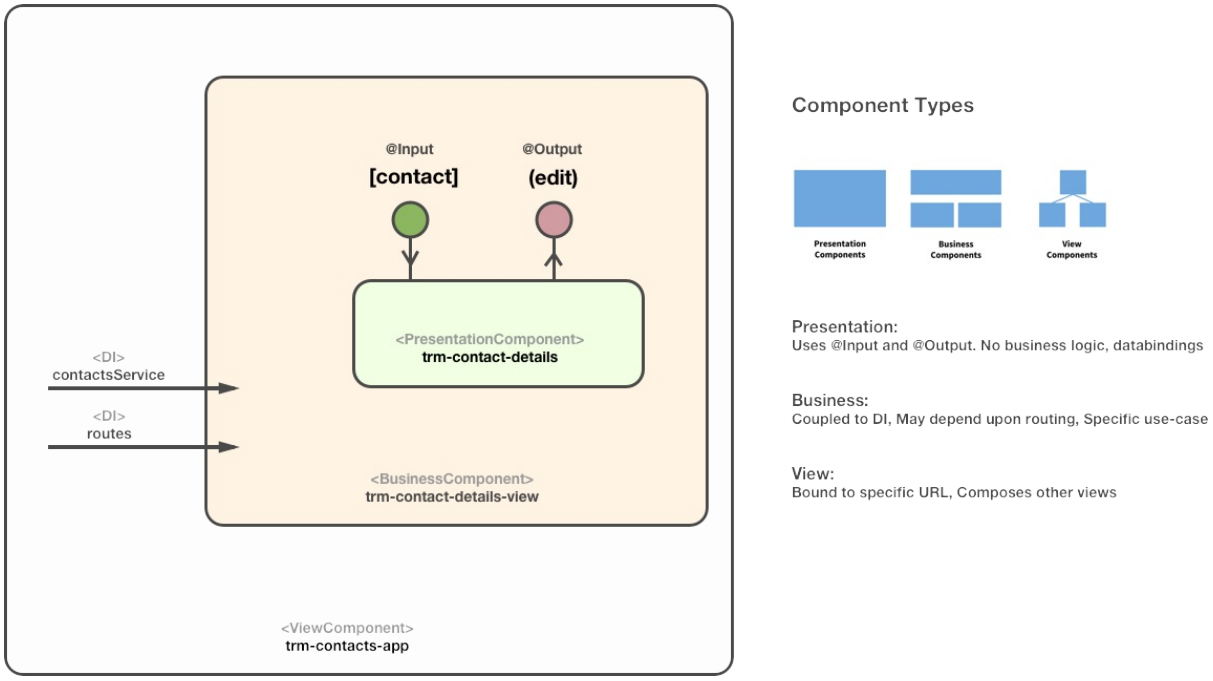


Figure: architecture-step-1

Scenario

Let's refactor our `ContactsDetailComponent` into two components:

- a `ContactsDetailComponent` is dumb presentation view; *dumb* in the sense that it doesn't know about any routing concerns or the `ContactsService`
- a `ContactsDetailViewComponent` is the view component which does know about the `ContactsService` and `RouteParams` but doesn't know how to display a contact.

By separating these two concerns (reusable renderer and specific view component) we get to an application architecture that embraces reusability with only a slight increase in complexity.

```
$ ng generate component contacts-detail-view -m app # or ng g c contacts-detail-view -m app
```

> The `-m app` is a long-hand version for `--module app.module.ts`. That said, if you stick to the convention of naming your module files `<name-of-module>.module.ts`. If you deviate from this naming convention you have to specify the entire filename, e.g. if you name your module file `app.feature.module.ts` you would have to pass the entire filename to the Angular CLI command: `ng g c contacts-detail-view -m app.feature.module.ts`.

Tasks

1. Create the `navigateToEditor` and `navigateToList` methods which use the `router#navigate(dsl)` API to route to different Components; note that these are methods of the `ContactsDetailViewComponent` class.
2. Refactor the `ContactDetailComponent` to a *Presentation* component:
    i. Move the code that handles the `RouteParams` from the `ContactDetailsComponent` to the `ContactsDetailViewComponent` and expose the contact as a `contact` property
    ii. Remove all routing concerns from the `ContactDetailComponent`
    iii. Import `EventEmitter` and `Output` into the `ContactDetailComponent` and emit `edit` and `back` events for the button clicks
    iv. Import `Input` and decorate the `contact` property as such
    v. Keep in mind that the buttons in the template are no longer anchors with links, but buttons with click handlers

Code Snippets

`contacts-details-view.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { ContactsService } from '../contacts.service';
import { Contact } from '../models/contact';

@Component({
  selector: 'trm-contacts-detail-view',
  template: `
    <trm-contacts-detail [contact]="contact"
                         (edit)="navigateToEditor($event)"
                         (back)="navigateToList()">
    </trm-contacts-detail>
  `,
  styleUrls: ['./contacts-detail-view.component.css']
})
export class ContactsDetailViewComponent implements OnInit {
  contact: Contact;

  constructor(
    private contactsService: ContactsService,
     private route: ActivatedRoute,
     private router: Router) {
  }

  ngOnInit() {
    this.contactsService
      .getContact(this.route.snapshot.params['id'])
      .subscribe(contact => this.contact = contact);
  }

  navigateToList () {
    this.router.navigate(['/']);
  }

  navigateToEditor (contact) {
    this.router.navigate(['/contact', contact.id, 'edit']);
  }
}
```

Figure: a1

`contacts-detail.component.ts`

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Contact } from '../models/contact';

@Component({
  selector: 'trm-contacts-detail',
  templateUrl: './contacts-detail.component.html',
  styleUrls: ['./contacts-detail.component.css']
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
  @Output() back = new EventEmitter<void>();
  @Output() edit = new EventEmitter<Contact>();
}
```

Figure: a2

Web and App Servers

To launch your web application, use a Terminal session with the command `ng serve`. This starts a web server for the Angular 2 application at url `http://localhost:4200`.

And since your Angular application requests data from `http://localhost:4201/api`, you will need a local server to respond to the REST API calls. Start a second, separate Terminal session with the commend: `npm run rest-api`.

> Note that most likely these sessions are already running... from previous exercises.

Additional resources and help

- Presentation vs Business vs View components
- Output
- Input

Next Lab

Go to Lab #2: Parent Injection into Children

Lab 2: Inject Parent into View Children

In this exercise we're going to build a generic Tab component. To be more specific we'll build a `tab` and a `tabs` component that work together as shown below.

```
<trm-tabs>
  <trm-tab title="General">
    ...
  </trm-tab>
  <trm-tab title="Address">
    ...
  </trm-tab>
</trm-tabs>
```

We'll apply the parent injection pattern to make that work.

Scenario

The `ContactsEditorComponent` shows general information as well as an address for each contact. With this exercise we want to build a tab component to display the general info and the address in two seperate tabs rather than one below the other.

Tasks

1. Create a new directory `tabs` inside your `app` directory
2. Go into the `tabs` directory and generate a `tab` and a `tabs` component
3. The markup for the `TabsComponent` loops through an array of `tab` instances to construct a list of tab headers. It uses content projection to project the actual `tab` elements so that they sit below the header.
4. The markup for the `TabComponent` is really simple. It uses content projection to display whatever is between the opening and the closing `tab` tag.

   ```
   <ng-content *ngIf="selected"></ng-content>
   ```

5. Create a boolean property `selected` as well as an string property `title` on the `TabComponent`
6. Import the `@Input` decorator and apply it to the `selected` and `title` property
7. Import the `TabsComponent` and use it as a typed parameter in the constructor. This will inject the `TabsComponent` instance which is a direct parent of the `tab` component in the DOM tree
8. In the `TabsComponent` implement a method `addTab(tab: TabComponent)` which takes a tab instance and puts it into a local array. We can think of it as a way to register instances of `TabComponent` at the `TabsComponent`
9. Implement a `select(tab: TabComponent)` method in the `TabsComponent` that sets `selected` to `false` for all tabs except the one that was passed by the caller
10. Make sure to call `select(tab: TabComponent)` for the first tab that gets registered
11. Use `<trm-tabs>` and `<trm-tab>` in the markup of the `ContactsEditorComponent` to reorganize the content in tabs.

Code Snippets

`tabs.component.html`

```
<nav mat-tab-nav-bar>
    <button type="button" mat-button mat-tab-link
       *ngFor="let tab of tabs"
       [class.disabled]="!tab.selected"
       [active]="tab.selected"
       (click)="select(tab)">
       {{  tab.title }}
    </button>
</nav>
<ng-content></ng-content>
```

Figure: a3

`tabs.component.ts`

```
import { Component } from '@angular/core';
import { TabComponent } from '../tab/tab.component';

@Component({
  selector: 'trm-tabs',
  templateUrl: './tabs.component.html',
  styleUrls: ['./tabs.component.css']
})
export class TabsComponent {
  private tabs: Array<TabComponent> = [];

  addTab(tab: TabComponent) {
    if (this.tabs.length === 0) {
      this.select(tab);
    }
    this.tabs.push(tab);
  }

  select(tab: TabComponent) {
    this.tabs.forEach(tab => tab.selected = false);
    tab.selected = true;
  }
}
```

Figure: a4

`contacts-editor.component.html`

```
<div class="trm-contacts-editor">
  <mat-card>
    <mat-card-title-group class="fullBleed editing">
      ...
    </mat-card-title-group>

    <mat-card-content>
      <trm-tabs>
        <trm-tab title="General" fxLayout="column">
          <mat-form-field fxFlex>
            <input matInput placeholder="Name" name="name" [(ngModel)]="contact.name">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Email" name="email" [(ngModel)]="contact.email">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Phone" name="phone" [(ngModel)]="contact.phone">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput placeholder="Website" name="website" [(ngModel)]="contact.website">
          </mat-form-field>
          <mat-form-field fxFlex>
            <input matInput name="birthday" [(ngModel)]="contact.birthday" type="date">
          </mat-form-field>
        </trm-tab>
        <trm-tab title="Address" fxLayout="column">
          ...
        </trm-tab>
      </trm-tabs>
    </mat-card-content>

    <mat-card-actions fxLayout fxLayoutAlign="center center">
      ...
    </mat-card-actions>
  </mat-card>
</div>
```

Figure: a2

Next Lab

Go to Lab #3: Using ContentChildren Query

Lab 3: Refactoring the Tabs Component to use ContentChildren

In this exercise we'll revisit our approach to building a tab component and try out a different approach by access a component's children components using `ContentChildren`.

Scenario

In the previous exercise we used the parent injection pattern to build a tab component consisting of two different elements `tab` and `tabs`.

```
<trm-tabs>
  <trm-tab title="General" fxLayout="column">
    ...
  </trm-tab>
  <trm-tab title="Address" fxLayout="column">
    ...
  </trm-tab>
</trm-tabs>
```

Turns out, that isn't the only way to implement such a thing. In fact there's a technique that is often superior using `ContentChildren`.

Tasks

1. In `TabComponent` remove the `TabsComponent` and `OnInit` imports and remove all the affected code
2. In `TabsComponent` add imports for `ContentChildren`, `QueryList` and `AfterContentInit`.
3. Change the type of `tabs` from `Array<TabComponent>` to `QueryList<TabComponent>` and decorate it as `@ContentChildren(TabComponent)` as shown below.

   ```
   @ContentChildren(TabComponent)
   tabs: QueryList<TabComponent>;
   ```

4. Implement the `AfterContentInit` interface and call `this.select(this.tabs.first)` in its implementation method

Code Snippets

`tabs.component.ts`

```
import { Component, ContentChildren, QueryList, AfterContentInit } from '@angular/core';
import { TabComponent } from '../tab/tab.component';

@Component({
  selector: 'trm-tabs',
  templateUrl: './tabs.component.html',
  styleUrls: ['./tabs.component.css']
})
export class TabsComponent implements AfterContentInit {

  @ContentChildren(TabComponent)
  tabs: QueryList<TabComponent>;

  ngAfterContentInit() {
   this.select(this.tabs.first)
  }

  select(tab: TabComponent) {
    this.tabs.forEach(tab => tab.selected = false);
    tab.selected = true;
  }
}
```

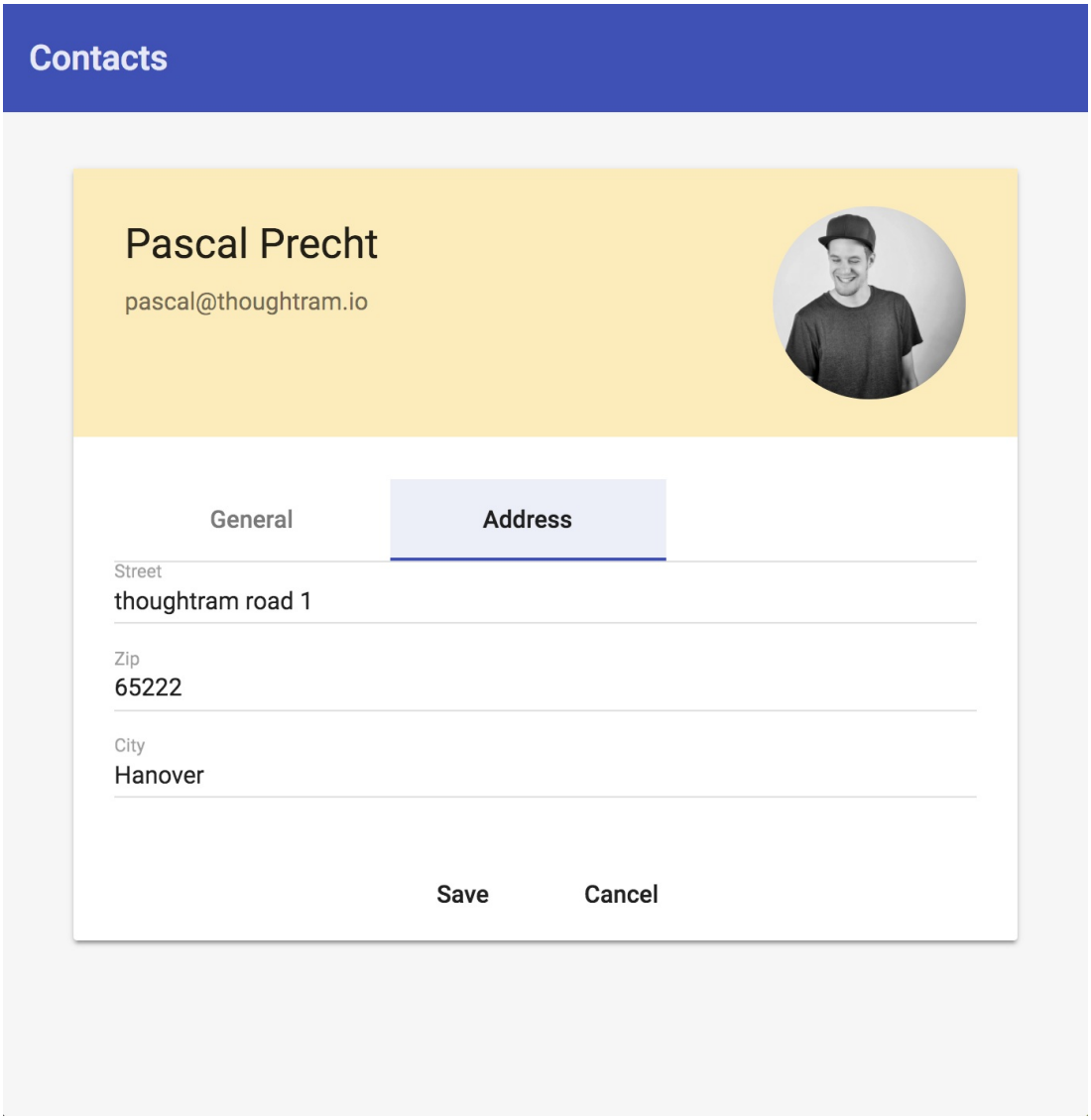
Figure: a5

Bonus Tasks

Our `TabsComponent` is great! However, it turns out, Angular Material comes with its own `MatTabs` component that comes with features like keyboard control and accessiblity.

1. Let's refactor our ContactsEditor to use `MatTabs` instead of our custom `TabsComponent`. Read the component's documentation to learn how to do it.

> Note: you will need to wrap your `mat-tab` content inside a `<div fxLayout="column"></div>` ; since `mat-tab` will (by default) layout in a *horizontal* flow-direction.



Code Snippets

`contacts-editor.component.html`

```
<div class="trm-contacts-editor">
  <mat-card>
    <mat-card-title-group class="fullBleed editing">
      ...
    </mat-card-title-group>
    <mat-card-content>

      <trm-tabs>
        <trm-tab title="General" fxLayout="column">
          ...
        </trm-tab>
        <trm-tab title="Address" fxLayout="column">
          ...
        </trm-tab>
      </trm-tabs>
    </mat-card-content>

    <mat-card-actions fxLayout fxLayoutAlign="center center">
     ...
    </mat-card-actions>
  </mat-card>
</div>
```

Figure: a3 1

Next Lab

Go to Lab #4: App Global Communications

Lab 4: Global Communicate via custom event bus

In this exercise we like to build a generic event bus to do inter-component communication in a generic and reusable way.

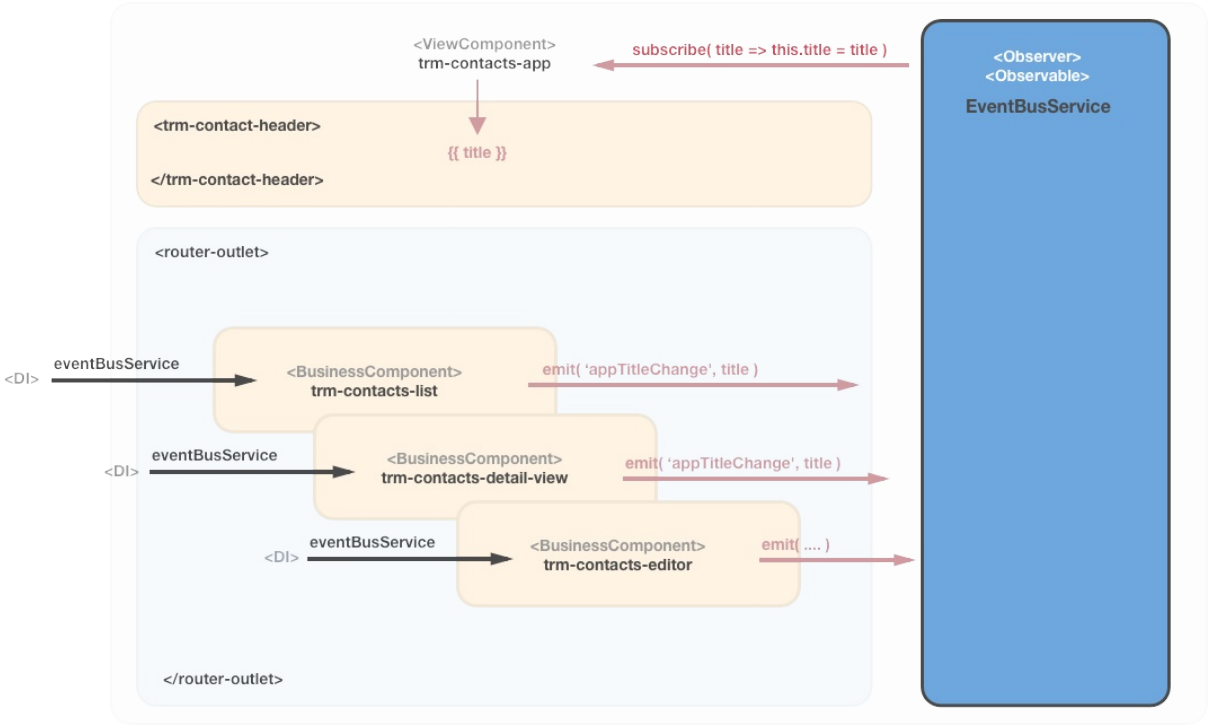


Figure: architecture-step-2-solution

Scenario

We want to create an `EventBusService` with the following two APIs.

```
EventBusService#emit(eventType: string, data: any)
```

```
EventBusService#observe(eventType: string) -> Observable<any>
```

With this service and its two APIs we have a central place to emit events of any event type with any data. We can also observe messages passed through the bus from any place that has access to the `EventBusService`.

Use the EventBusService to update the application `title` based on the current, routed/active view.

Tasks

1. Create an `EventBusService` using angular-cli
    i. Import `Observable` from `rxjs`
    ii. Import `Observer` from `rxjs`
    iii. Import `Subject` from `rxjs`
    iv. Create an interface `EventBusArgs` with a field `type` of type `string` and a field `data` of type `any`
    v. Create a new `Subject` generic of type `EventBusArgs` on a `messages` property

   vi. Implement a method `emit(eventType: string, data: any)` which emits on the subject
   vii. Implement a method `observe(eventType: string) -> Observable<any>` which filters on the subject
2. Add `EventBusService` as a provider to our application module
3. Import and expose the `EventBusService` on the `ContactsAppComponent`
4. Observe the `appTitleChange` event and inject the current title on the `title` property

```
ngOnInit () {
  this.eventBusService.observe('appTitleChange')
                      .subscribe(title => this.title = title);
}
```

5. For each of the router-outlet views, import the `EventBusService` and emit the `appTitleChange` event with the desired value on all relevant code paths

Code Snippets

`event-bus.service.ts`

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';
import { map, filter } from 'rxjs/operators';

export interface EventBusArgs {
  type: string;
  data: any;
}

@Injectable()
export class EventBusService {
  private announcer = new Subject<EventBusArgs>();

  emit(eventType: string, data: any) {
    this.announcer.next({
      type: eventType,
      data: data
    });
  }

  observe(eventType: string): Observable<string> {
    const isRequestedEvent = (event) => event.type === eventType;
    const getTitle = (event) => event.data;
    const events$ = this.announcer.asObservable();

    return events$.pipe(
      filter(isRequestedEvent),
      map(getTitle)
    );
  }
}
```

Figure: image

`app.component.ts`

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar color="primary">
      {{title$ | async}}
    </mat-toolbar>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['./app.component.scss']
})
export class ContactsAppComponent {
  title$ = this.eventbus.observe(
    'appTitleChange'
  );

  constructor(private eventbus: EventBusService){}
}
```

Figure: image

`contacts-list.component.ts`

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>>;

  constructor(
    private service: ContactsService,
    private publisher: EventBusService) {}

  ngOnInit () {
    const loadContacts$ = this.service.getContacts();

    this.contacts$ = loadContacts$.pipe(
      tap(list => this.publisher.emit(
        'appTitleChange',
        `${list.length} Contacts`
      ))
    );
  }
}
```

Figure: image