EERSTE ANGULAR APPLICATIE OPZETTEN

OPDRACHT 1.1 – NIEUWE APP MET CLI

Installeer Angular en voeg Bootstrap toe aan het project, en start je eerste app

a. Gebruik npm om een Angular project te initialiseren (gebruik minimaal versies van node v.16.14.0 en npm v.8.0.0). Installeer eerst Angular CLI:

```
npm install -g @angular/cli
```

b. Open een commando prompt waar je het project wil maken, en gebruik de CLI om een nieuw project te genereren:

```
ng new angular-workshop
```

De CLI stelt 2 vragen; of je Angular Routing wilt gebruiken (kies voor deze workshop het antwoord 'nee'), en welk Stylesheet Format je wilt gebruiken (voor deze workshop wordt SCSS gebruikt)

c. CD naar de nieuwe project folder, en installeer Bootstrap:

```
npm install bootstrap
```

Angular herkent de Bootstrap stylesheets nog niet. Voeg hiervoor de stylesheet toe aan de styles array in *angular.json* (regel 35: "styles": ["src/styles.scss", "node_modules/bootstrap/scss/bootstrap.scss"]).

d. Gebruik de Angular CLI om het project lokaal te draaien:

```
ng serve
```

De app is nu beschikbaar in je browser op http://localhost:4200. Wijzig de titel in app.component.ts en kijk hoe de nieuwe html in de browser wordt getoond.

OPDRACHT 1.2 – NIEUW COMPONENT MAKEN

Maak een nieuw component aan en voeg deze toe aan de app.

a. Maak een nieuwe folder aan (app/shared/header), maak daarin een nieuw TypeScript bestand (header.component.ts). Maak een nieuwe class met de @Component() decorator, en geef een waarde voor `selector` en `template`. Voeg een console.log() toe aan de ngOnInit methode.

```
@Component( {selector: 'app-header', template: ''} )
export class HeaderComponent implements OnInit {
   public ngOnInit(): void { console.log('het component werkt!'); }
}
```

b. Importeer het nieuwe component in app.module.ts door het toe te voegen aan de lijst met `declarations`. Plaats vervolgens het component in je app.component.html (`<app-header></app-header>`), en dan zou je de log in je browser moeten kunnen zien.

c. Geef nu ook waardes aan de @Component decorator voor `template` en `styles` (bijv. `'<h1>Header </h1>'` als template en `['h1 { color: blue }']` voor styles)

OPDRACHT 1.3 – TEMPLATEURL EN STYLEURLS

a. Maak de nieuwe bestanden header.component.html en header.component.scss aan. Voeg hieraan de code toe die je in je header wilt zien, bijvoorbeeld in het html bestand:

```
<ul class="nav nav-tabs nav-fill">
   <li class="nav-item">
      <a class="nav-link active">Cities</a>
   </li>
   <li class="nav-item">
      <a class="nav-link">Trips</a>
   </li>
</ul>
```

En in het SCSS bestand:

```
:host {
   display: block;
   margin: 3em 0;
}
```

b. Vervang in de @Component decorator `template` met `templateUrl` en `styles` met `styleUrls`:

```
@Component({
   selector: 'app-header',
   templateUrl: './header.component.html',
   styleUrls: ['./header.component.scss']
})
```

DATA BINDING

OPDRACHT 2.1 – NIEUWE COMPONENTEN CITIES-LIST EN CITIES-EDIT

Maak een nieuw component `cities`, en daarin twee nieuwe componenten `cities-list` en `cities-edit`

a. Gebruik de CLI om in totaal 3 nieuwe componenten aan te maken (bijv `ng generate component cities --skip-tests` en `ng g c cities/cities-list --skip-tests` voor de eerste 2 componenten).

b. Plaats `<app-cities>` in app.component.html, en plaats in app.cities.html beide andere componenten:

```
<h3>All the cities you've ever visited!</h3>
<div class="row">
  <div class="col">
    <app-cities-list></app-cities-list>
  </div>
  <div class="col">
    <app-cities-edit></app-cities-edit>
  </div>
</div>
```

OPDRACHT 2.2 – STRING INTERPOLATION EN TWO-WAY-BINDING

Voeg input velden toe aan de html en toon de waardes van de input met string interpolation

a. Maak in cities-edit.component.ts twee nieuwe variabelen, `name` en `country`.

b. Maak in de bijbehorende html 2 input velden. Voeg een input toe voor `name` (zoals hieronder) en voeg ook een input toe voor `country`:

```
<h4>Add new cities</h4>
<div class="row py-2">
  <div class="col">
    <label for="name">City name:</label>
  </div>
  <div class="col">
    <input type="text" id="name" class="form-control">
  </div>
  <!-- TODO: add html for Country as well -->
```

c. Plaats onder de input velden een <p> element waarin de waardes getoond kunnen worden (`<p>City name: {{name}}</p>` en `<p>City country: {{country}}</p>`)

d. Voeg tenslotte de two-way-binding toe aan de input elementen (bijv `[(ngModel)]="name"`). Denk er aan dat het voor ngModel nodig is om FormsModule als import toe te voegen aan app.module.ts. In de browser zou je nu de waardes van je input velden geprint moeten zien worden.

OPDRACHT 2.3 – EVENT-BINDING EN PROPERTY-BINDING

Voeg een knop toe die de waardes van de input velden kan printen, en die disabled is wanneer er geen waarde voor beide velden is opgegeven

a. Plaats een knop onder de input velden (`<button type="button" class="btn btn-primary">Add city</button>`). Geef de knop een event-binding mee, waarmee je naar een click event luistert en een nieuwe methode aanroept: `(click)="addCity()"`

b. Voeg de methode `addCity()` toe aan het TypeScript bestand, waarin je eerst een console.log() gebruikt om de `name` en `country` te loggen, en waarin je daarna beide waardes terug op een lege string `''` zet.

c. Voeg tenslotte ook een property-binding toe aan de knop, op de property disabled, waarmee je een methode isDisabled aanroept: `[disabled]="isDisabled()"`. In deze nieuwe methode controleer je of waardes voor naam en aantal bestaan (`return !this.name || !this.country;`).

INPUT EN OUTPUT DECORATORS

OPDRACHT 3.1 – NIEUW COMPONENT MET *NGFOR

Maak een model voor City, en gebruik deze om een lijst van cities te tonen in cities-list.component

a. Maak een nieuwe folder app/shared/models, en maak hierin een nieuw bestand city.model.ts. Plaats hierin een constructor waar `name` en `country` kunnen worden meegegeven:

```
export class City {
  constructor (public name: string, public country: string) {}
```

b. Maak in city-list.component.ts een variabele: `public cities: City[] | undefined;`. Maak in de bijbehorende html een lijst van deze cities met behulp van de structural directive *ngFor:

```
<h4>Cities list</h4>
<div class="list-group py-2">
   <button class="list-group-item list-group-item-action"
        *ngFor="let city of cities">
      {{ city.name }} - {{ city.country }}
   </button>
</div>
```

OPDRACHT 3.2 – INPUT DECORATOR

Maak een lijst van cities in cities.component.ts, en koppel deze aan de cities-list component met behulp van input decorators

a. Maak en initialiseer een lijst van cities aan in cities.component.ts:

```
public cities: City[] = [
   new City('Vienna, 'Austria), new City('Florence, 'Italy')
];
```

b. Voeg aan de variable `cities` in cities-list.component.ts een `@Input()` decorator toe.

```
@Input() public cities: City[] | undefined;
```

Geef daarna aan het component in de html (cities.component.html) de waarde voor de input door met behulp van property-binding:

```
<app-cities-list [cities]="cities"></app-cities-list>
```

De lijst zou nu zichtbaar moeten worden in je browser.

OPDRACHT 3.3 – OUTPUT DECORATOR

Voeg een eventEmitter toe aan cities-edit waarmee een nieuwe city kan worden toegevoegd aan de lijst in cities.component

a. Voeg een eventEmitter met een `@Output()` decorator toe in cities-edit.component.ts, en noem deze `onAddCity`:

```
@Output() public onAddCity = new EventEmitter<City>();
```

b. Roep deze eventEmitter aan vanuit de functie addCity: `this.onAddCity.emit(new City(this.name, this.country)).`

c. Dit event kun je nu opvangen met event-binding op het element, waarmee je vervolgens een nieuwe methode in het parent-element kunt aanroepen. In cities.component.html:

```
<app-cities-edit (onAddCity)="addCity($event)"></app-cities-edit>
```

En in cities.component.ts:

```
public onAddCity(city: City): void {
   this.cities.push(city));
}
```

SERVICES

OPDRACHT 4.1 – EEN NIEUW COMPONENT TRIPS

Maak een nieuw component Trips, en bouw een simpele link om tussen de componenten te kunnen schakelen

a. Gebruik de Angular CLI om het nieuwe component trips te maken. Maak in het nieuwe TypeScript bestand een variabele trips, zoals het voorbeeld hieronder, en voeg hier wat data aan toe:

```
public trips: Trip[] = [
   { title: 'Summer 2023', cities: [
      new City('Florence', Italy'), new City('Vienna', Austria')
   ]}
];
```

b. Maak ook een nieuw model aan voor Trip in de models folder:

```
export interface Trip {
 title: string;
 cities: City[];
}
```

c. Voeg in de html van het Trips component een lijst toe, waar de trips kunnen worden weergegeven:

```
<h3>City Trips</h3>
<ul class="list-group">
   <li class="list-group-item" *ngFor="let trip of trips">
      <h4>{{ trip.title }}</h4>
      <div class="row">
         <div class="col-8">
            <span *ngFor="let city of trip.cities">
               {{ city.name }} ({{ city.country }}) >
            </span>
            <span>Home</span>
         </div>
      </div>
   </div>
</div>
```

d. Verderop in deze workshop wordt op een nette manier navigatie tussen de componenten toegevoegd (hoofdstuk 6: Routing), maar voor nu wordt een simpele en snelle manier toegepast met de Angular directive `*ngIf`. Voeg hiervoor eerst in app.component.html twee links toe, die met event-binding een waarde aan variabele `page` kunnen meegeven (voeg deze variabele ook toe aan app.component.ts):

```
<button type="button" class="btn btn-link" (click)="page = 0">
    Cities
```

```
  </button>
  <button type="button" class="btn btn-link" (click)="page = 1">
      Trips
  </button>
```

e. Voeg de componenten selectors van Cities en Trips toe, en gebruik een `*ngIf` om te bepalen welk component getoond moet worden:

```
<app-cities *ngIf="page === 0"></app-cities>
<app-trips *ngIf="page === 1"></app-trips>
```

OPDRACHT 4.2 – CITIES SERVICE

Maak een service die de cities kan opslaan en vrijgeven, waardoor de Trips en Cities componenten dezelfde cities lijst kunnen delen.

a. Maak een nieuw bestand cities.service.ts binnen de folder /cities. Maak hierin een nieuwe class aan, en voeg een lijst met cities toe.

```
export class CitiesService {
   private cities: City[] = [
      new City('Barcelona', 'Spain'),
      new City('Venice', 'Italy')
   ];
}
```

b. Voeg twee methodes toe: getCities(`return this.cities.slice()`) en addCity(`this.cities.push(city)`)

c. Voeg de CitiesService vervolgens toe aan app.module, in de *providers* lijst.

OPDRACHT 4.3 – CITIES SERVICE IN TRIPS COMPONENT

Voeg een methode toe in trips.component die cities kan toevoegen aan de service

a. In de html van trips.component, waar de lijst met trips wordt getoond, moet een link worden toegevoegd met een event-binding die een nieuwe methode aanroept:

```
<div class="col-4">
   <button type="button" class="btn btn-link"
           (click)="addCities(trip.cities)">
      Add Cities to list
   </button>
</div>
```

b. De methode in trips.component.ts geeft de cities door aan de service. Voeg eerst de service als parameter toe aan de constructor (`public constructor (private citiesService: CitiesService) {}`), en roep dan addCities van de service aan om de cities daar toe te voegen

```
public addCities(cities: City[]): void {
   cities.forEach((city) => {
      this.citiesService.addCities(city);
   });
}
```

OPDRACHT 4.4 – CITIES SERVICE IN CITIES COMPONENT

Roep de CitiesService aan vanuit het cities component, om de lijst weer te kunnen geven

a. Ga naar cities.component.ts, en voeg de CitiesService toe als parameter in de constructor.

b. Voeg een aanroep naar de service toe om cities op te halen:

```
public ngOnInit(): void {
    this.cities = this.citiesService.getCities();
}
```

OPDRACHT 4.5 – CITIES SERVICE EN RXJS SUBJECT()

Voeg een RxJS Subject() toe aan de CitiesService, zodat de cities-list mooi up-to-date blijft met de lijst

a. Verwijder eerst de eventEmitter die de communicatie tussen het cities-edit.component en het cities.componenten mogelijk maakt (verwijder de eventListener `(onAddCity)="addCity($event)"` uit cities.component.html, de methode `addCity()` uit cities.component.ts, en de `eventEmitter onAddCity` uit cities-edit.component.ts).

b. Voeg nu de citiesService toe aan cities-edit.component.ts, en voeg in de methode `addCity()` een aanroep toe naar de service:

```
const city = new city(this.name, this.country);
this.citiesService.addCity(city);
```

c. Zoals je ziet, komen nieuw toegevoegde cities nu niet meer in de lijst te staan. Om dit op te lossen, kan gebruik gemaakt worden van een observable Subject () (te importeren op deze manier: `import { Subject } from 'rxjs';`). Deze Subject moet worden toegevoegd aan cities.service:

```
public CitiesChanged = new Subject<City[]>();
```

d. Vervolgens moet Subject worden gebruikt zodra de lijst wordt ge-update, oftewel in de methode `addCities`, nadat `this.cities.push` is uitgevoerd, met behulp van de methode `next()`:

```
this.citiesChanged.next(this.cities.slice());
```

e. Tenslotte moet naar het `Subject` en diens `next()` methode worden geluisterd vanuit de cities.component, zodat de lijst kan worden bijgewerkt zodra wijzigingen zijn doorgevoerd. In de functie ngOnInit van cities.component.ts:

```
this.citiesService.citiesChanged.subscribe((cities: City[]) => {
    this.cities = cities;
});
```

OPDRACHT 4.6 – VOORKOM DATA LEKKAGE OP ONDESTROY()

Om te voorkomen dat er meerdere subsciptions blijven bestaan, is de RXJS pipe `takeUntil()` een geschikte oplossing

a. Maak een nieuw Subject aan in cities.component:

```
private onDestroy = new Subject<void>();
```

b. Geef aan dat het cities.component de `OnDestroy` interface implementeert, en voeg de volgende methode toe:

```
public ngOnDestroy(): void {
    this.onDestroy.next();
    this.onDestroy.complete();
}
```

c. Maak gebruik van de RXJS pipe `takeUntil()`:

```
This.citiesService.citiesChanged.pipe(
    takeUntil(this.onDestroy)
).subscribe(...);
```

HTTP CLIENT

OPDRACHT 5.1 – JSON-SERVER SETUP

Om een REST Backend te simuleren maken we gebruik van de package json-server.

a. Installeer json-server met npm:

```
npm install json-server --save-dev
```

b. Voeg een bestand toe voor de data, genaamd db.json, en vul deze met data (hou dezelfde properties aan voor de cities zoals in de app, dus *name* en *country*, en een extra property *id*):

```
{ "cities": [{ "id": 1, "name": "Florence", "country": "Italy" }] }
```

Voeg ook een bestand toe voor de routing naar de api:

```
{ "/api/*": "/$1" }
```

c. Configureer de proxy settings, eerst in angular.json op regel 68:

```
"serve": {
    "options": { "proxyConfig": "src/proxy.conf.json" }
}
```

En daarna in src/proxy.conf.json:

```
{
    "/api": {
         "target": "http://localhost:3000",
         "secure": false
    }
}
```

d. Voeg ten slotte een nieuw script toe in package.json:

```
"database": "json-server –watch db.json –routes db.routes.json"
```

Wanneer je dit nieuwe script nu runt, zal een simpele server beschikbaar zijn. Kijk op http://localhost:3000/cities voor de data die in db.json is toegevoegd. Dankzij de proxy is deze data ook beschikbaar op http://localhost:4200/api/cities.

OPDRACHT 5.2 – HTTPCLIENT MODULE

Maak in de cities.service gebruik van de HttpClientModule om data op te halen van en weg te schrijven naar de backend service

a. Voeg de HttpClientModule toe aan de imports lijst in app.module.ts

b. Verwijder in de cities.service de lijst van cities, gebruik de decorator `@Injectable()` en voeg een constructor aan de service toe waarmee de `HttpClient` wordt geïnjecteerd:

```
@Injectable()
export class CitiesService {
    public constructor(private httpClient: HttpClient) {}
```

OPDRACHT 5.3 – GET CITIES MET HTTPCLIENT.GET()

In `getCities()` kan nu een GET call naar de json-server gemaakt worden om daar de data op te halen

a. Vervang de return type van de methode, en return een get methode:

```
public getCities: Observable<City[]> {
    return this.httpClient.get<City[]>('api/cities');
}
```

b. Vervang in cities.component de manier waarop cities een waarde wordt gegeven. Maak een nieuwe `private` methode `getCities()`, en roep deze aan in de `ngOnInit` methode, en in de `citiesChanged.subscribe` methode. In `getCities()` worden de cities opgehaald via de service:

```
private getCities(): void {
    this.citiesService.getCities().subscribe((cities) => {
        this.cities = cities;
    }
}
```

OPDRACHT 5.4 – ADD CITY MET HTTPCLIENT.POST()

In `addCity()` kan nu de nieuwe lijst naar de backend worden gestuurd

a. In plaats van data naar de oude city array te schrijven, kan nu een POST request naar de server gemaakt worden:

```
this.httpClient.post('api/cities', city).subscribe(() => {
    this.citiesChanged.next();
}
```

b. Let op: nu de lijst met cities niet meer in de server wordt bewaard maar in de backend, is het niet meer logisch om met het `citiesChanged` Subject een lijst van cities op te sturen. In plaats daarvan wordt het Subject alleen gebruikt om aan te geven dat de lijst is verandert, waarna het aan de observers zelf is om de nieuwe lijst op te halen. Het Subject ziet er in de CitiesService dan ook als volgt uit:

```
public citiesChanged = new Subject<void>();
```

Het is nu aan de afnemers, zoals in dit geval cities.component, om opnieuw de lijst met cities op te halen. Daarom wordt de methode getCities in cities.component gebruikt.

ROUTING

OPDRACHT 6.1 – ROUTING MODULE

Voeg een nieuwe module toe waar de routes kunnen worden geconfigureerd

a. Maak een nieuw bestand app/app-routing.module.ts. Maak hierin een class `AppRoutingModule`, en geef deze een `@NgModule` decorator.

b. Maak in datzelfde bestand een `const routes: Routes = []` aan (buiten de AppRoutingModule class), en vul deze array met vier nieuwe routes (een redirect naar /cities, de cities en trips componenten, en een catch welke ook weer naar /cities redirect)

```
{ path: '', redirectTo: '/cities', pathMatch: 'full' },
{ path: 'cities', component: CitiesComponent },
{ path: 'trips', component: TripsComponent },
{ path: '**', redirectTo: '/ cities' }
```

c. Geef de @NgModule decorator een imports (`imports: [RouterModule.forRoot(routes)]`) en exports (`exports: [RouterModule]`) array mee

d. Voeg de nieuwe AppRoutingModule toe als import in app.module

OPDRACHT 6.2 – ROUTER OUTLET EN NAVIGATIE

Voeg een router-outlet element toe waar de componenten kunnen worden weergegeven, en gebruik de navigatie uit de header om de juiste paden op te geven

a. Voeg de router-outlet toe in app.component.html, waarbij de <button> elementen, <app-cities> en <app-trips> kunnen worden verwijderd.

```
<app-header></app-header>
<router-outlet></router-outlet>
```

b. Navigatie in de browser werkt nu al, maar de links werken nog niet. Voeg hiervoor een property-binding toe aan de <a> elementen in header.component.html. Bind hierbij aan de property [routerLink], en geef hier een array aan mee waarin je de paths van de componenten meegeeft:

```
<a [routerLink]="['cities']" class="nav-link">Cities</a>
```

c. Om de links active te maken, kan de Angular directive routerLinkActive worden gebruikt. Geef aan beide <a> elementen deze directive mee (`<a routerLinkActive="active">`)

d. Om vanuit Trips meteen te kunnen navigeren naar Cities wanneer je cities hebt toegevoegd, moet navigatie vanuit de class mogelijk zijn. Voeg in de constructor van trips.component.ts daarvoor eerst de module Router toe (`constructor (private router: Router) {}`). Deze router kan vervolgens worden gebruikt in de methode `addCities()`, nadat de `citiesService` is aangeroepen om de cities op te slaan:

```
this.router.navigate(['cities']);
```

OPDRACHT 6.3 – CHILD PATHS

Voeg sub paden toe vanaf het pad /cities, zodat meer gedetailleerde navigatie mogelijk wordt

a. Voeg 2 nieuwe paden toe in app-routing.module, als children van het pad /cities:

```
{ path: 'cities', component: CitiesComponent, children: [
      { path: 'new', component: CitiesEditComponent },
      { path: 'edit/:id', component: CitiesEditComponent }
    ]},
```

b. Vervang het component `<app-cities-edit>` in cities.component.html met een `router-outlet`. Beide paden openen nu (nog) hetzelfde component.

c. De volgende stap is om beide nieuwe paden beschikbaar te maken vanuit de html, en dat doen we in cities-list.component.html. Om naar /cities/new te navigeren, voegen we een button toe onder de lijst, met een routerLink property:

```
<div class="d-grid py-2">
    <button class="btn btn-outline-primary" [routerLink]="['new']">
         New City
    </button>
</div>
```

d. Om naar /cities/edit/:id te navigeren, hebben we een `id` property nodig op het city model. Voeg deze toe aan de constructor in city.model:

```
constructor(public name: string, public country: string, public id?: number)
```

e. Met dit `id` kunnen we nu navigeren vanuit city-list.component. Hiervoor voegen we een (click) event-binding toe aan de button elementen. Roep daarbij een nieuwe methode aan en geef `city.id` mee als parameter:

```
<button (click)="editCity(city.id!)">
```

f. Om in de methode `editCity()` te navigeren, hebben we de modules Router en ActivatedRoute nodig; voeg deze toe aan de constructor van city-list.component.ts. Daarna kan vanuit de methode worden genavigeerd naar de nieuwe route:

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

```
public editCity(index: number): void {
    this.router.navigate(['edit', index], {relativeTo: this.route});
}
```

OPDRACHT 6.4 – ROUTE PARAMS UITLEZEN

Update het cities-edit.component zodat nieuwe cities kunnen worden toegevoegd, en bestaande cities kunnen worden aangepast.

a. De eerste stap is om in cities-edit.component.ts te kijken naar wat het pad is waar de gebruiker op zit. Hiervoor is de module ActivatedRoute nodig; voeg deze toe aan de constructor:

```
constructor (private citiesService: CitiesService,
             private route: ActivatedRoute) {}
```

b. Van ActivatedRoute kan `params` worden aangeroepen, welke een observable teruggeeft. Hier kan een `subscribe` methode op aangeroepen worden, en in de callback functie kunnen de params van ActivatedRoute worden opgeslagen. Deze params worden gebruikt om een variabele `cityIndex` gelijk te zetten aan de `param:id`, of om deze op `null` te zetten:

```
public cityIndex: number | undefined;

public ngOnInit(): void {
    this.route.params.subscribe((params: Params) => {
       if (params['id']) {
          this.cityIndex = +params['id'];
       }
    })
 }
```

c. Met de waarde cityIndex kunnen we nu in de html verschil maken tussen cities/add en cities/edit/:id. Bijvoorbeeld met verschillende titels en knoppen (de methodes voor de Update en Delete knoppen worden later nog toegevoegd):

```
<h4>{{ cityIndex === undefined ? 'Add new' : 'Edit'}} cities</h4>
 ...
<button type="button" *ngIf="cityIndex === undefined" (click)="addCity()"
         [disabled]="isDisabled()" class="btn btn-primary">
    Add City
</button>
<button type="button" *ngIf="cityIndex !== undefined" (click)="updateCity()"
         [disabled]="isDisabled()" class="btn btn-primary">
    Update City
</button>
<button type="button" *ngIf="cityIndex !== undefined" (click)="deleteCity()"
         [disabled]="isDisabled()" class="btn btn-outline-primary">
    Delete City
 </button>
```

OPDRACHT 6.5 – CITY-ADD COMPONENT AFMAKEN

De laatste stap is om onder het pad cities/edit de mogelijkheid te bieden om cities uit de lijst te wijzigen en te verwijderen.

a. In de citiesService moeten twee nieuwe methodes worden toegevoegd, die deze functionaliteit kunnen aanbieden. Eerst moeten we een `getCities()` toevoegen, die met een `id` als input parameter een enkel City object kan teruggeven:

```
public getCity(id: number): Observable<City> {
   return this.httpClient.get<City>(`api/cities/${id}`);
}
```

b. De tweede methode die we nodig hebben in de citiesService is een `updateCity()`, met `id` en een `city` object als input parameters. Hiermee wordt een PUT request gemaakt, om de lijst in de database bij te werken:

```
public updateCity(id: number, city: City): Observable<City> {
    return this.httpClient.put<City>(`api/cities/${id}`, city);
}
```

c. Net als bij de `addCity()` methode, willen we na de `updateCity()` methode aangeven dat de lijst is gewijzigd, via het `citiesChanged` subject. Om dit te doen maken we gebruik van een nieuwe RXJS operator, namelijk `tap()`. Voeg in addCity() een pipe toe, direct na de put():

```
.pipe(
    tap(() => this.citiesChanged.next())
);
```

d. De derde methode die we in de citiesService nodig hebben, is een `deleteCity()`, met alleen een `id` als input parameter. Hiermee wordt een DELETE request gemaakt, en ook na deze request willen we laten weten dat de lijst gewijzigd is en dus de `tap()` operator gebruiken:

```
public deleteCity(id: number): Observable<void> {
    return this.httpClient.delete<void>(`api/cities/${id}`)
       .pipe(
          tap(() => this.citiesChanged.next())
       );
}
```

e. Deze nieuwe methodes kunnen nu vanuit het cities-edit component worden aangeroepen. Eerst in de OnInit functie, binnen de if-loop van de `params.subscribe()` wanneer een `params['id']` beschikbaar is:

```
this.cityIndex = +params['id'];
this.getCity(this.cityIndex);
```

Deze nieuwe getCity() methode haalt het City object op via de city.service:

```
private getCity(id: number): void {
    this.citiesService.getCity(id).subscribe((city) => {
         this.name = city.name;
         this.country = city.country;
    });
}
```

f. Tenslotte worden twee nieuwe methodes toegevoegd, namelijk `deleteCity()` en `updateCity()`, welke allebei de citiesService aanroepen en daarna terug navigeren naar de Cities pagina. Deze methodes worden zelf aangeroepen door middel van de `<button>` elementen die al eerder in de html zijn toegevoegd (zie opdracht 6.4.c):

```
public deleteCity(): void {
   this.citiesService.updateCity(this.cityIndex!).subscribe(() => {
      this.router.navigate(['cities']);
   });
}

public updateCity(): void {
   const city = new City(this.name, this.country);
   this.cityService.updateCity(city).subscribe(() => {
      this.router.navigate(['cities']);
   });
}
```

TEMPLATE DRIVEN FORMS

OPDRACHT 7.1 – CITIES-EDIT-FORM COMPONENT INPUT

Om een formulier te maken, gebruiken we een nieuw component

a. Gebruik de CLI om het component aan de app toe te voegen:

```
ng g c cites/cities-edit/cities-edit-form --skip-tests -m app
```

b. In het nieuwe component plaatsen we de input velden die gekopieerd kunnen worden uit de cities-edit template

```
<div class="row py-2">
    <div class="col">
         <label for="name">City name:</label>
         ...
```

c. De properties `name` en `country` gaan we niet meer gebruiken, maar in plaats daarvan geven we het hele City object mee als input parameter. In cities-edit-form.component.ts:

```
@Input() public city: City | undefined;
```

In de template moet `ngModel` nu gebind worden aan dit City object (voor name én country):

```
<input [(ngModel)]="city.name">
```

En om problemen te voorkomen in het geval dat City nog undefined is, plaatsen we het hele blok van de template nu nog even in een `<div>` met een structural directive:

```
<div *ngIf="city">...</div>
```

d. Vervolgens moet het nieuwe component worden gebruikt in het cities-edit component, met de juiste input binding. Definieer eerst in cities-edit.component.ts een City object, en geef dit object een waarde in de `route.params.subscribe` methode:

```
public city: City | undefined;
...
    if(params['id']) {
       this.getCity(params['id']);
    else {
       this.city = new City('', '');
    }
```

Vergeet niet om in `getCity()` de resultaten van de service call aan het nieuwe City object toe te schrijven, in plaats van deze aan de aparte variabele voor `name` en `country` toe te schrijven.

e. Voeg als laatste ook <app-cities-edit-form> toe aan de template en geef City mee als input:

```
<h4>{{city && city.id ? 'Edit' : 'Add new'}} cities</h4>
<app-cities-edit-form [city]="city"></app-cities-edit-form>
```

OPDRACHT 7.2 – CITIES-EDIT-FORM COMPONENT OUTPUT

De Delete functionaliteit en <button> kunnen in het edit-cities component blijven, maar de Update en Add functionaliteit worden naar het nieuwe edit-cities-form verplaatst, omdat daar de nieuwe waardes voor `name` en `country` ook zitten. Aanroepen van de service gebeurt nog wel in edit-cities

a. Pas de methodes aan in cities-edit.component. AddCity() krijgt een City object als parameter:

```
public addCity(city: City): void {
    this.citiesService.addCity(city);
    this.router.navigate(['cities']);
}
```

DeleteCity() gebruikt niet meer cityIndex (deze mag helemaal verwijderd worden):

```
public deleteCity(): void {

    this.citiesService.deleteCity(this.city!.id).subscribe(...
```

En updateCity() krijgt ook een City object als parameter:

```
public deleteCity(city: City): void { ...
```

b. In de template voegen we output bindings toe aan `<app-cities-edit-form>` waarmee we de bestaande methodes kunnen aanroepen:

```
<app-cities-edit-form [city]="city" (onAddCity)="addCity($event)"
(onUpdateCity)="updateCity($event)">
```

De Update en Add buttons kunnen worden verwijderd, en alleen de Delete button blijft over:

```
<div class="row py-2 justify-content-end">
    <div class="col-6 d-grid">
         <button type="button" (click)="deleteCity()" *ngIf="city?.id">
              Delete City
         </button>
    </div>
</div>
```

c. In cities-edit-form.component.ts moeten de outputs worden toegevoegd:

```
@Output public onAddCity = new EvenEmitter<City>();
@Output public onUpdateCity = new EvenEmitter<City>();
```

d. In de nieuwe template wordt een button toegevoegd die gebruikt kan worden voor Add of Update:

```
<div class="row py-2 justify-content-end">
    <div class="col-6 d-grid">
       <button type="button" (click)="onSubmit()" class"btn btn-primary">
          {{ city && city.id ? 'Update' : 'Add' }} City
       </button>
    </div>
</div>
```

e. Tenslotte wordt een methode toegevoegd `onSubmit()` waardoor via het click event op de buttons de outputs kunnen worden aangeroepen:

```
public onSubmit(): void {
   this.city!.id
       ? this.onUpdateCity.emit(this.city)
       : this.onAddCity(this.city);
 }
```

OPDRACHT 7.3 – FORM VALIDATION & SUBMIT

Maak gebruik van Angular directives om het formulier te valideren

a. Vervang de buitenste met een <form> element en voeg het `ngForm` directive toe. Voeg ook een event binding toe voor `(ngSubmit)`, en roep hierop de onSubmit() functie aan. Let ook op dat je nu het `(click)` event verwijderd van de button, en type `submit` meegeeft:

```
<form *ngIf="city" #form="ngForm" (ngSubmit)="onSubmit()">
...
<button type="submit" class="btn btn-primary">
```

b. Voeg nu ook de property `name` toe aan het name input element, de variabele declaratie `name` en een validator `required`:

```
<input [(ngModel)]="name" #name="ngModel" name="name" required>
```

c. Omdat dit veld nu required is, kunnen ook errors getoond worden wanneer het formulier invalid is. Voeg het volgende element toe, meteen onder het <input> element:

```
<span *ngIf="name.touched && name.invalid">
    <i>Please enter a valid name</i>
</span>
```

d. Om te voorkomen dat het formulier wordt ge-submit als het niet geldig is, kan de event binding op (ngSubmit) worden uitgebreid, op het <form> element:

```
(ngSubmit)="form.valid && onSubmit()"
```

e. Er kunnen meerdere validators worden toegevoegd, met verschillende errors. Zet op het country input veld de volgende validators, met daaronder verschillende error berichten:

```
<input [(ngModel)]="country" #country="ngModel" name="country" required
       minlength="2">
<span *ngIf="country.touched && country.errors.['required']">
  <i>Please enter a valid country</i>
</span>
<span *ngIf="country.touched && country.errors.['minlength']">
  <i>Country value must be at least 2 characters</i>
</span>
```

REACTIVE FORMS

OPDRACHT 8.1 – CITIES FORMGROUP

In dit hoofdstuk vervangen we het Template Driven Form door een Reactive Form

a. Maak een Form aan in cities-edit-form.component.ts:

```
public cityForm!: FormGroup;
```

b. Voeg `implements OnChanges` aan het component toe, en gebruik die methode om het formulier te initialiseren:

```
public ngOnChanges(changes: SimpleChanges): void {
    this.initForm();
    if (changes['city'] && changes['city'].currentValue) {
         this.setFormValues(changes['city'].currentValue);
    }
}
```

c. Voeg de twee nieuwe methodes toe:

```
private initForm(): void {
    this.cityForm = new FormGroup<CityForm>({
       name: new FormControl(''),
       country: new FormControl('')
    });
}

private setFormValues(city: City): void {
    this.cityForm.patchValue({
       name: city.name, country: city.country
    )};
}
```

d. De FormGroup wordt getypeerd als een <CityForm>. Maak een nieuw model waarin wordt gedefinieerd hoe dit CityForm eruit moet zien, in models/city.form.model.ts:

```
export interface CityForm {
    name: FormControl<string, null>;
    country: FormControl<string, null>;
}
```

e. Pas de template aan. Op het <form> element moet de formGroup worden meegegeven:

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

En op de <input> elementen moet de formControlName worden meegegeven (doe dit voor beide inputs):

```
<input type="text" id="name" [formControlName]="'name'">
```

OPDRACHT 8.2 – FORMGROUP VALIDATION

Validators worden in Reactive Forms gezet op de FormControl objecten

a. Voeg validators toe in de methode `initForm()`:

```
name: new FormControl('', Validators.required),
country: new FormControl('', [Validators.required, Validators.minlength(2)])
```

b. Voeg error messages in de template toe om errors op de controls te laten zien:

```
<span *ngIf="cityForm.get('name')!.touched && cityForm.get('name').invalid">
    <i>Please enter a valid name</i>
</span>
```

c. De template wordt veel leesbaarder als we de controls in `get()` methodes beschikbaar maken. Voeg de volgende methode toe in cities-edit-form.component.ts:

```
public get nameControl(): FormControl {
    return this.cityForm.get('name') as FormControl;
}
```

En gebruik deze getters in de template:

```
<span *ngIf="nameControl.touched && nameControl.invalid">
```

Maak ook een getter en error messages voor het country veld.

d. Het is nog altijd mogelijk om het formulier te submitten. Controleer daarom op de `invalid` status in onSubmit:

```
public onSubmit(): void {
    if (this.cityForm.valid) {
         this.city!.id
            ? this.onUpdateCity.emit(this.getCityForm())
            : this.onAddCity.emit(this.getCityForm());
    } else {
         this.markFormAsTouched();
    }
}
```

e. Voeg tenslotte ook de helper functies markFormAsTouched() en getCityFromForm() toe:

```
private markFormAsTouched(): void {
    this.nameControl.markAsTouched();
    this.countryControl.markAsTouched();
}
```

```
private getCityFromForm(): City {
    return new City(this.cityForm.value.name, this.cityForm.value.country);
}
```

OPDRACHT 8.3 – FORMARRAY

Met Reactive Forms kun je het formulier makkelijk uitbreiden zodra dit nodig is

a. Voeg aan het City model een nieuwe optionele property toe in de constructor (`public yearsVisited?: number[]`), en voeg ook aan de city.form.model deze property toe:

```
yearsVisited: FormArray<FormControl<number | null>>;
```

b. In het CitiesEditForm component moet de nieuwe property worden toegevoegd in de initForm() methode:

```
yearsVisited: new FormArray([new FormControl()])
```

c. Om de FormArray en de bijbehorende controls makkelijk te benaderen, voegen we twee nieuwe get methodes toe:

```
public get yearsVisited(): FormArray {
    return this.cityForm.get('yearsVisited') as FormArray;
}

public get yearsVisitedControls(): FormControl[] {
    return (<FormArray<FormControl>>this.cityForm.get('yearsVisited'))
            .controls;
}
```

d. Daarnaast moet de nieuwe FormArray kunnen worden gevuld in de setFormValues() methode. Zorg eerst dat de FormArray de juiste lengte heeft:

```
if (city && city.yearsVisited && city.yearsVisited.length > 1) {
    for (let i = 1; i < city.yearsVisited.length; i++) {
         this.yearsVisited.push(new FormControl());
    }
}
```

En zorg dan dat in `patchValue()` de juiste waardes worden gevuld:

```
yearsVisited: city.yearsVisited
```

e. In de helper functie getCityFromForm() moet de waarde van de FormArray worden toegevoegd aan het City model:

```
this.cityForm.value.yearsVisited.filter(Number)
```

f. In de template moeten nieuwe input velden worden toegevoegd:

```
<div class="row">
    <div class="col py-2">
       <label for="yearsVisited">Years visited:</label>
    </div>
    <div class="col" [formArrayName]="'yearsVisited'">
       <div *ngFor="let year of yearsVisitedControls; let i = index"
            class="py-2">
          <div class="input-group">
             <input type="number" class="form-control" id="yearsVisited"
                  [formControl]="year">
```

g. Meteen onder de input velden worden ook twee nieuwe buttons toegevoegd, waarmee items aan de FormArray kunnen worden toegevoegd of verwijderd:

```
<button type="button" class="btn btn-outline-primary" *ngIf="i === 0"
         (click)="addYearVisited()">+</button>
<button type="button" class="btn btn-outline-primary"
         (click)="removeYearVisited(i)">x</button>
```

h. Tenslotte moeten twee nieuwe methodes in het component worden toegevoegd, die reageren op de button events:

```
public addYearVisited(): void {
    this.yearsVisited.push(new FormControl());
}

public removeYearVisited(index: number): void {
    this.yearsVisited.removeAt(index);
    if (this.yearsVisited.value.length === 0) {
       this.addYearVisited();
    }
 }
```

ASYNC VALIDATION

OPDRACHT 9.1 – FORM UITBREIDEN

Om een Async Validator toe te voegen, gebruiken we een nieuwe City property Country Code

a. Breid het bestaande City model uit met een nieuwe property in de constructor:

```
public countryCode: string
```

Voeg deze property ook toe aan het CityFormModel:

```
countryCode: FormControl<string | null>;
```

Zorg ervoor dat de code compileert, door de nieuwe property toe te voegen daar waar de constructor wordt aangeroepen (trips component, city-edit component)

b. In cities-edit-form.component.ts moet de nieuwe property ook worden toegevoegd. Voeg eerst de FormControl toe in initForm():

```
countryCode: new FormControl('', [Validators.required,
Validators.minlength(2)]);
```

c. Maak een get() methode voor `countryCodeControl`, voeg `city.countryCode` toe aan patchValue(), en voeg `countryCodeControl.marckAsTouched()` toe in de helper functie markFormAsTouched().

d. Voeg een input veld toe in de template van edit-cities-form.component.html:

```
<input type="text" id="countryCode" [formControlName]="countryCode">
```

Voeg ook error berichten toe voor de errors `required` en `minlength`:

```
<span *ngIf="countryCodeControl.touched &&
                            countryCodeControl.errors?.['required']">
    <i>Please enter a valid code</i>
</span>
```

OPDRACHT 9.2 – CUSTOM ASYNC VALIDATOR

Maak een nieuw bestand aan: cities-edit-form/validators/country-code.validator.ts. In deze custom validator gaan we een API call maken die de countryCode kan valideren.

a. Maak in het nieuwe bestand een `class` CountryCodeValidator, met een `@Injectable` annotatie en geef aan dat deze class de interface `AsyncValidator` implementeert:

```
@Injectable({
    providedIn: 'root'
})
export class CountryCodeValidator implements AsyncValidator {}
```

b. In deze nieuwe class definieren we een readonly variable die de url naar de REST Countries API aangeeft, een constructor waarmee we HttpClient kunnen importeren, en een validate() functie:

```
private readonly REST_COUNTRIES_URL = https://restcountries.com/v3.1/alpha';

public constructor(private httpClient: HttpClient) {}

validate(control: AbstractControl): Observable<ValidationErrors | null> {}
```

c. In de validate functie controleren we eerst of we een waarde hebben meegekregen voor `country` en `countryCode`:

```
const country = control.get(country');
const countryCode = control.get(countryCode');
```

Als we geen waardes hebben, kunnen we de validation niet doen, en willen we dus ook geen error zetten. Daarom returnen we `null`:

```
if (country && countryCode) {
    ...
} else {
    return of(null);
}
```

d. Als we wel een `country` en `countryCode` hebben, kunnen we een GET call maken naar de REST Countries API:

```
return this.httpClient.get<RestCountry[]>(
    `${this.REST_COUNTRIES_URL}/${countryCode.value}`
)
```

De interface RestCountry maakt het makkelijk om het resultaat van de call te gebruiken. Maak hiervoor een nieuw bestand aan genaamd models/rest-country.model.ts:

```
export interface RestCountry {
    name: { common: string };
}
```

e. Het resultaat van de GET call bepaald of we een error willen teruggeven of niet. Hiervoor gebruiken we een `map()` operator, waarin we de resultaten interpreteren:

```
.pipe(
    map((countryData) => {
       if (countryData[0].name.common === country.value) {
          return null;
```

```
        } else {
            return { countryCode: true };
        }
    }
```

f. Tenslotte wordt de call uitgebreid met 2 extra operators. Eerst een `delay()` voor een fijner demo effect:

```
delay(500),
```

En daarna ook een `catchError()` voor wanneer de API een code niet herkent:

```
catchError(() => of({ countryCode: true }))
```

OPDRACHT 9.3 – DE ASYNC VALIDATOR GEBRUIKEN

De nieuwe error kan in de template van het formulier gebruikt worden om error berichten te tonen

a. Voeg de nieuwe async validator toe aan cities-edit-form. Allereerst moet deze in de constructor worden toegevoegd:

```
public constructor(private countryCodeValidator: CountryCodeValidator) {}
```

b. Daarna moet de validator worden toegevoegd aan de FormGroup. Omdat we twee controls nodig hebben voor deze validatie, voegen we de validator toe een de gehele FormGroup:

```
new FormGroup({
    ...
}, {
    asyncValidators: this.countryCodeValidator.validate
});
```

Let op! Wanneer de Validator wordt gebruikt door het Angular formulier, is alleen de validate() functie beschikbaar. Dat betekent dat wanneer het formulier deze functie toepast, het keyword `this` een andere scope heeft. In runtime zie je dus errors op `this.httpClient` of `this.REST_COUNTRIES_URL`. Om dit op te lossen kun je de scope meegeven met `.bind()`:

```
this.countryCodeValidator.validate.bind(this.countryCodeValidator)
```

c. Nu de validator is toegevoed, kan ook een error bericht worden toegevoegd in de template:

```
<span *ngIf="countryCodeControl.touched && cityForm.errors.['countryCode']">
    <i>Code does not match country</i>
</span>
```

d. Als laatste kan gebruik worden gemaakt van de Form property `pending`, zodat in de template te zien is dat er iets op de achtergrond gebeurt. Voeg een property binding toe op de submit button:

```
[disabled]="cityForm.pending"
```