Ullriikka Nokelainen syksy 2023

Angular Tutorial: Tour of Heroes – Tiivistelmä tehtävistä

1. The hero editor

Projektin luonnin jälkeen luon Heroes-komponentin komennolla *ng generate component heroes*. Angular-komponenteissa on oletuksena kolme erillistä tiedostoa: luokkaosa (TypeScript), templaatti (html) ja tyylit (css). Määrittelen luokkaosassa hero-muuttujan arvoksi 'Windstorm'. Sidon sen templaattiin yksisuuntaisella tekstiupotussidoksella (*interpolation binding*) *<h2>{{hero}}</h2>*. Lisään Heroes-komponentin näkymään sovelluksessa eli kirjoitan sen elementtivalitsimen (*element selector*) app.component.html-tiedostoon *<app-heroes></app-heroes>.* Luon rajapintaluokan hero.ts, jossa määritellään Hero-tyypin ominaisuudet *id: number, name: string*. Tuon rajapintaluokan heroes-komponenttiin ja määrittelen hero-muuttujan tyypiksi Hero *hero: Hero*. Templaatin otsikko päivitetään muotoon *{{hero.name}}.* Muutan nimen kapiteeleille *uppercase*-pipella *{{hero.name | uppercase}}*. Pipet ovat yksinkertaisia funktioita, joita voi lisätä suoraan templaattiin. Jotta käyttäjä voisi muokata sankarin nimeä, tarvitaan kaksisuuntainen sidos input-lomakkeen ja hero-muuttujan välillä *<input id="name" [(ngModel)]="hero.name" placeholder="name">*. Että *ngModel*-direktiivi taas toimisi, pitää *FormsModule* tuoda app.module.ts -tiedostoon. Nyt nimen muokkaus käyttöliittymässä on mahdollista.

2. Display a list

Luon mock-heroes.ts -tiedoston, jossa on HEROES-vakio, jonka arvona on taulukko Hero-tyyppisiä olioita. Vakio tuodaan heroes-komponenttiin, jossa heroes-muuttujan arvoksi annetaan HEROES. Heroes-komponentin templaattiin lisätään listaelementti *<li>*, jonka sisällä on painike, joka sisältää sankarin nimen ja id:n näyttävät rivit. Jotta HEROES-taulukko voitaisiin käydä läpi ja näyttää kaikki sankarit, tarvitaan **ngFor*-direktiiviä. Se tuodaan komponentin luokkaosaan ja lisätään templaattiin *<li *ngFor="let hero of heroes">*. **ngFor* toistaa -elementin jokaista listan elementtiä kohden. Näin saadaan joukko painikkeita, joissa kaikissa lukee eri sankarin id ja nimi. Painikkeeseen lisätään

klikkaustapahtumasidos (*event binding*) merkinnällä *<button type="button" (click)="onSelect(hero)">*. Merkintä *(click)* saa Angularin kuuntelemaan painikkeessa tapahtuvia klikkauksia. Kun painiketta klikataan, ajetaan *onSelect(hero*). Päivitän Heroes-komponentin luokkaosassa hero-muuttujan muotoon *selectedHero?: Hero* (kysymysmerkki kertoo, ettei tällä muuttujalla ole pakko olla arvoa) sekä lisään metodin *onSelect(hero: Hero): void { this.selectedHero = hero; }.* Lisäksi tuon **ngIf*-direktiivin. Templaattiin lisään toisen div-elementin, jonka sisällä on sankarin lisätiedot. Jotta lisätiedot näkyvät vain valitusta sankarista, lisään diviin **ngIf*-direktiivin *<div *ngIf="selectedHero"*>. Luokkasidoksella voi lisätä ja poistaa CSS-tyylitystä ehdollisesti. Lisään painikkeeseen luokkasidoksen *[class.selected]="hero === selectedHero"* . Kun nykyisen rivin hero on sama kuin *selectedHero*, näytetään *selected*-luokan CSS-tyyli.

3. Create a feature component

Luon uuden komponentin HeroDetail. Heroes-komponentin lisätieto-osio siirretään uuteen komponenttiin. *selectedHero* korvataan muuttujalla *hero*. HeroDetail-komponentin luokkaosaan importataan *Hero* ja *Input* sekä kirjoitetaan luokkamääritelmään *@Input() hero?: Hero* eli komponentti voi vastaanottaa *hero*-muuttujaan Hero-tyyppisen olion. *@Input*-dekoraattori mahdollistaa sen, että emokomponentti (eli Heroes-komponentti) voi sitoa arvon *hero*-muuttujaan muuttujasidoksella (*property binding*). Se tapahtuu kirjoittamalla Heroes-komponentin templaattiin *<app-hero-detail [hero]="selectedHero"></app-hero-detail>*. Heroes-komponentin *selectedHero*-muuttuja mapataan HeroDetail-komponentin *hero*-muuttujaan. Kun hero-listaa klikataan, *selectedHero* vaihtuu, muuttujasidos päivittää *heron* ja HeroDetail-komponentti näyttää uuden valitun sankarin.

4. Add services

Koska komponenttien ei tulisi noutaa tai tallentaa dataa suoraan, luon palvelun *ng generate service hero*. *@Injectable()*-dekoraattori kertoo, että tämä luokka on osa Angularin dependency injection -systeemiä ja on siis palvelu. HeroService hakee datan ja välittää sen muihin luokkiin. Tässä tapauksessa data haetaan mock-heroes -tiedostosta. Palveluun importataan Hero-rajapinta ja HEROES-taulukko ja palvelun metodiksi tulee *getHeroes(): Hero[ ] { return HEROES; }*. Palvelussa

määritellään oletuksena, että se on saatavissa juuritasolla eli kaikkien komponenttien pyydettävissä *@Injectable({ providedIn: 'root', }).*

Nyt Heroes-komponenttiin tuodaan HEROES-taulukon sijaan HeroService. Heroes-vakion määrittely korvataan alustuksella *heroes: Hero[ ] = [ ];*. Teen Heroes-komponenttiin myös metodin *ngOnInit(): void { this.getHeroes(); }*. Metodi laitetaan alustuksessa ajettavan ngOnInit-elinkaarimetodin sisään (*lifecycle hook*) eikä konstruktoriin, koska Angularissa konstruktorin ei tulisi tehdä mitään. Tosielämässä getHeroes()-metodi ei saa olla synkroninen, koska, jos tietoja ei palautettaisi välittömästi, selain jäisi odottamaan niitä. Tuon *Observable*- ja *of*-symbolit RxJS-kirjastosta (Reactive Extensions Library for JavaScript). *Observable* on olio, joka lähettää yhden tai useamman tiedon (tai ei tietoja lainkaan) tietämättä vastaanottajaa. Tiedot vastaanotetaan tilaamalla (*subscribe*).

Palvelun getHeroes()-metodi muutetaan muotoon *getHeroes(): Observable<Hero[]> { const heroes = of(HEROES); return heroes; }.* Heroes-komponentin getHeroes()-metodi päivitetään muotoon *getHeroes(): void { this.heroService.getHeroes().subscribe(heroes => this.heroes = heroes); }.* Nyt metodi odottaa Observablen lähettävän taulukon sankareita. Kun subscribe()-metodi saa datan asynkronisesti Observablelta, se välittää sen callback-funktiolle, joka sijoittaa saadun taulukon *heroes*-muuttujaan.

Luon Messages-komponentin ja lisään sen elementtivalitsimen App-komponenttiin *<app-messages></app-messages>*. Lisäksi luon Message-palvelun, jolla on viestisäilö sekä kaksi metodia. *Add()* lisää viestin säilöön ja *clear()* tyhjentää säilön.

MessageService injektoidaan HeroServiceen importtaamalla ja lisäämällä konstruktoriin *constructor(private messageService: MessageService) { }.* HeroServicen getHeroes()-metodi päivitetään lähettämään viesti aina, kun sankarit haetaan *getHeroes(): Observable<Hero[]> const heroes = of(HEROES); this.messageService.add('HeroService: fetched heroes'); return heroes; }* MessageService tuodaan Messages-komponenttiin ja komponentin konstruktoriin lisätään parametri, joka alustaa julkisen *messageService*-muuttujan: *constructor(public messageService: MessageService) { }. MessageService*-muuttujan on oltava julkinen, koska se sidotaan templaattiin (joka on aina julkinen). Muokkaan Messages-komponentin templaattia.

Heroes-komponentin *onSelect()*-metodiin lisätään kutsu MessageService-palveluun *this.messageService.add(`HeroesComponent: Selected hero id=${hero.id}`),* jotta jokaisesta valitusta sankarista luodaan uusi viesti.

5. Add navigation

Luon reititinmoduulin eli AppRoutingModulen *ng generate module app-routing --flat --module=app*. Reitittimessä määritellään reitit *const routes: Routes = [{ path: 'heroes', component: HeroesComponent }]*;. Polku (*path*) on merkkijono, joka vastaa selaimessa näkyvää URL-osoitetta ja komponentti (*component*) kertoo, mikä komponentti reitittimen tulisi esittää, kun mennään kyseiseen osoitteeseen. @NgModulessa reititin alustetaan sekä exportataan koko sovelluksen käyttöön. *@NgModule({ imports: [RouterModule.forRoot(routes)], exports: [RouterModule]})*.

App-komponentin templaatissa vaihdan *<app-heroes>*-elementtivalitsemen sijaan *<router-outlet></router-outlet>*. Nyt Heroes-komponentti näytetään vain, kun siihen navigoidaan. *<router-outlet>* kertoo missä kohtaa sivulla reititin näyttää sen hetkisen valitun reitin. Lisään App-komponenttiin navigaation *<nav> <a routerLink="/heroes">Heroes </nav>*. RouterLink-direktiivi (jonka valitsin on *routerLink*) tekee elementistä linkin, joka käynnistää navigaation, joka taas avaa reittiä vastaavan komponentin sivulle sijainnissa *<router-outlet>.*

Luon Dashboard-komponentin, joka näyttää osittaisen (*sliced*) listan sankareista. Se importataan AppRoutingModuleen ja sille lisätään oma reitti *{ path: 'dashboard', component: DashboardComponent }*. Dashboard-komponentista tehdään myös sovelluksen oletusreitti *{ path: '', redirectTo: '/dashboard', pathMatch: 'full' }*. Kun muuta reittiä ei ole valittuna, reititin näyttää Dashboard-komponentin. Dashboardille lisätään linkki App-komponentin nav-elementtiin.

HeroDetail-komponentti tuodaan AppRoutingModuleen. Reitteihin lisätään *{ path: 'detail/:id', component: HeroDetailComponent }*. Polussa oleva kaksoispiste ilmaisee, että *:id* on *placeholder* tietylle id-arvolle. Dashboard-komponentin sankarilinkit muutetaan muotoon *<a *ngFor="let hero of heroes" routerLink="/detail/{{hero.id}}"> {{hero.name}} *. Nyt sankarin nimeä painettaessa selaimen URL päivittyy ja reititin näyttää oikean komponentin.

Koska sankarin lisätietoja ei enää valita saman osoitteen sisällä, vaan niihin navigoidaan, korvaan Heroes-komponentin painikkeet reitititintä hyödyntävillä linkeillä *<li *ngFor="let hero of heroes"> <a*

*routerLink="/detail/{{hero.id}}"> {{hero.id}} {{hero.name}} *. Tyylillisesti ne näyttävät pienen päivityksen jälkeen samoilta kuin aiemminkin.

6. Get data from a server

HttpClient tuodaan AppModuleen *import { HttpClientModule } from '@angular/common/http'* ja lisätään @ngModulen imports-listaan. Asennan In-memory Web API -paketin *npm install angular-in-memory-web-api –save* ja luon ulkopuolista API:a simuloivan InMemoryData-palvelun. HeroServicessä lisään importtiin *{ HttpClient, HttpHeaders } from '@angular/common/http'*. HttpClient injektoidaan konstruktoriin yksityiseen http-muuttujaan *private http: HttpClient*.