Angular web-sovelluskehys

Eveliina Pakarinen, 014152724

Ohjelmistoarkkitehtuurien harjoitustyö
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 25. syyskuuta 2018

Sisältö

1 Johdanto

Angular on asiakaspuolen web-sovelluskehys ja -sovellusalusta, jonka avulla voidaan toteuttaa asiakaspuolen web-ohjelmistoja (*Architecture overview*, 2010-2018). Tässä harjoitustyössä käsitellään Angularia 2.0 versiopäivityksen jälkeen. Harjoitustyön kirjoittamishetkellä viimeisin Angularin pääversio on Angular 6.

Angularin arkkitehtuurityylejä ovat modulaarisuus ja komponenttipohjaisuus (*Architecture modules*, 2010-2018; *Architecture components*, 2010-2018). Angularissa on oma modulaarisuusjärjestelmänsä, ja Angularin moduulit koostuvat komponenteista. Angularissa toteutetaan myös useita ohjelmistotason ratkaisumalleja kuten riippuvuusinjektio ja yhdensuuntainen tiedonsiirto.

2 Kehyksen esittely

Angular on asiakaspuolen web-sovelluskehys ja -sovellusalusta, jonka avulla voidaan toteuttaa asiakaspuolen web-ohjelmistoja (*Architecture overview*, 2010-2018). Angular on toteutettu TypeScript-ohjelmointikielellä kirjastoina, joista Angularin ydin ja valinnaiset osat koostuvat ja joiden pohjalta voidaan koostaa uusia sovelluksia.

2.1 Arkkitehtuurityylit ja komponenttien roolijako

Angular-sovelluskehyksellä toteutetut sovellukset koostuvat useasta eri pääosasta, joita ovat *moduulit* (*modules*), *komponentit* (*components*) ja *palvelut* (*services*) (*Architecture overview*, 2010-2018). Näiden osien välisiä suhteita kuvataan kuvassa 2.1.

Yksi Angularin arkkitehtuurityyleistä on modulaarisuus. Modulaarisuus on toteutettu Angularissa modulaarisuusjärjestelmällä *NgModules* (*Architecture modules*, 2010-2018). Moduuleilla voidaan kapseloida yhtenäiseksi kokonaisuudeksi yksi toiminnallinen kokonaisuus. Angular-sovellukset koostuvat yhdestä tai useammasta NgModule-moduulista. Jokaisessa Angular-sovelluksessa on vähintään yksi NgModule-luokka juurimoduulina (*Architecture modules*, 2010-2018).

Moduuleihin voidaan myös tuoda lisää toiminnallisuutta importoimalla muita NgModuuleja (*Architecture modules*, 2010-2018). Lisäksi moduulista voidaan exportoida toiminnallisuutta, jotta sitä voidaan käyttää muissa moduuleissa hyödyksi.

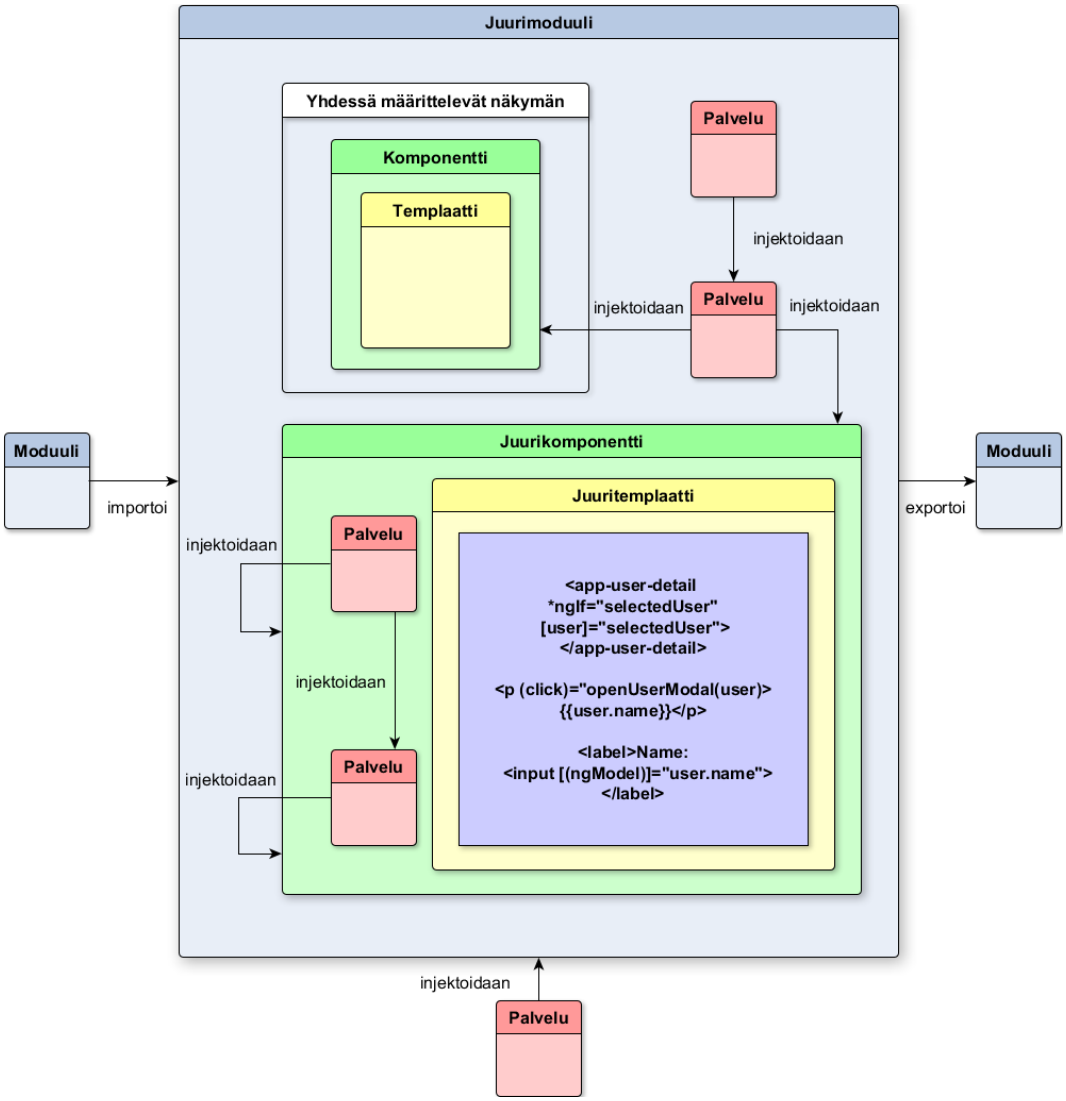
Juurimoduuli ja moduulien importoiminen ja exportoiminen on kuvattu kuvassa 2.1 sinisillä laatikoilla ja nuolilla.

Toinen Angularin arkkitehtuurityyleistä on komponenttipohjaisuus. Angular-moduulit koostuvat komponenteista, joita voi olla yhdessä moduulissa yksi tai useampia (*Architecture modules*, 2010-2018). Jokaiseen juurimoduuliin sisältyy juurikomponentti, joka on kuvattu kaaviossa 2.1 vihreällä laatikolla. Angular-komponenttien tarkoitus on määritellä yksi osa ruudulla näkyvästä sisällöstä ja kontrolloida sen toiminnallisuutta (*Architecture components*, 2010-2018). Komponentit koostuvat komponenttiin liittyvästä sovelluslogiikasta ja komponenttiin liittyvästä *templaatista* (*template*). Komponenttiin liittyvä sovelluslogiikka määritellään komponentin TypeScript-luokassa. Kuvassa 2.1 templaatit on kuvattu komponenttien sisään keltaisilla laatikoilla.

Yhdessä komponenttilogiikka ja komponenttiin liittyvä templaatti määrittelevät *näkymän* (*view*) (*Architecture modules*, 2010-2018). Näkymät ovat sovelluksessa yksittäisiä osia, jotka kontrolloivat yhtä osaa ruudulla näkyvästä sisällöstä (*Architecture components*, 2010-2018). Näkymät voivat muodostaa puumaisen hierarkian, sillä näkymissä voidaan viitata toisiin näkymiin sekä saman moduulin sisällä että importoiduissa moduuleissa (*Architecture modules*, 2010-2018).

Templaatit toteutetaan HTML-kielellä ja Angularin omalla templaatti syntaksilla, jonka avulla HTML-rakennetta voidaan muuttaa sovelluslogiikan perusteella (*Architecture components*, 2010-2018). Templaateissa voidaan käyttää *direktiivejä* (*directive*) sovelluslogiikan tuomiseksi templaattiin (*Template Syntax*, 2010-2018). Kuvassa 2.1 direktiivistä esimerkkinä toimii `*ngIf`, joka löytyy liilasta laatikosta juuritemplaatti-laatikon sisältä.

Datasidonnan (*data binding*) avulla yhdistetään templaatissa DOM-mallissa näytettävä data ja komponentin sovelluslogiikassa käsiteltävä data toisiinsa (*Architecture components*, 2010-2018). Templaatissa käytettävästä datasidontasyntaksista kuvassa 2.1 esimerkkeinä toimivat `[user]`, `(click)`, `[(ngModel)]` ja `{{user.name}}`. Toisiin komponentteihin viittaamisesta kuvassa 2.1 esimerkkinä toimii HTML-tagi `<app-user-detail>`.



Kuva 2.1: Angular-sovelluskehyksen pääosien suhteet toisiinsa

Yksi Angularin modulaarisuutta ja uudelleenkäytettävyyttä tukeva piirre on erottaa palvelut komponenteista (*Architecture services*, 2010-2018). Palvelut ovat luokkia, joiden avulla voidaan toteuttaa yksi hyvin määritelty sovelluksen tarvitsema toiminnallisuus. Kuvassa 2.1 palvelut on kuvattu punaisilla laatikoilla. Palveluita voidaan injektoida komponentteihin riippuvuuksina (*Architecture services*, 2010-2018).

3 Kehyksen yleiskuva

Angular web-sovelluskehys pohjautuu modulaariseen komponenttipohjaiseen arkkitehtuurityyliin. Angular toteuttaa kuitenkin myös useita ohjelmistotason ratkaisumalleja, joita ovat esimerkiksi riippuvuusinjektio, yhdensuuntainen tiedonsiirto ja tarkkailijasuunnittelumallin käyttö.

3.1 Ohjelmistotason ratkaisumallit

Yksi Angularissa sovellettavista ohjelmistotason ratkaisumalleista on *riippuvuusinjektio* (*dependency injection*), jonka avulla komponenteille voidaan tuoda käyttöön niiden tarvitsemaa lisätoiminnallisuutta (*Dependency injection pattern*, 2010-2018). Riippuvuusinjektion avulla komponenteille voidaan tarjota palveluissa toteutettua uudelleenkäytettävää toiminnallisuutta. Palvelut voidaan injektoida niin, että palvelun näkyvyysalue on koko sovelluksen laajuisesti, tietyn moduulin sisällä tai tietyn komponentin sisällä (*Dependency injection*, 2010-2018). Palveluiden näkyvyysalueet tulevat esiin myös kuvasta 2.1 punaisten palvelu-laatikoiden sijoittelusta moduuli- ja komponenttilaatikoiden sisä- ja ulkopuolelle.

Toinen Angularissa sovellettavista ratkaisumalleista on *yhdensuuntainen tiedonsiirto* (*unidirectional data flow*). Angularissa DOM-mallissa näytettävä data ja komponentin sovelluslogiikassa käsiteltävä data yhdistetään toisiinsa datasidonnan avulla (*Template Syntax*, 2010-2018). Datasidontaa on kolmea eri tyyppiä: datan lähteeltä näkymää kohti (*source-to-view*), näkymästä datan lähdettä kohti (*view-to-source*) ja kahdensuuntainen datasidonta (*view-to-source-to-view*), joka on yhdistelmä kahdesta ensin mainitusta datasidontatyypistä (*Template Syntax*, 2010-2018).

Datasidonnan avulla näkymään sisältyvät templaatti ja komponentti voivat välittää tietoa toisilleen (*Architecture components*, 2010-2018). Lisäksi Angular-sovelluksen puumaisen rakenteen vanhempi- ja lapsikomponentti voivat välittää tietoa toisilleen datasidonnan avulla (*Architecture components*, 2010-2018). Datasidonnalla on siis tärkeä merkitys sovelluksen sisäisten osien kommunikaation kannalta (*Architecture components*, 2010-2018). Kahdensuuntaisessa datasidonnassa syötekenttä voi saada lähtöarvon komponentilta *source-to-view*-sidonnan avulla ja käyttäjän tekemän muutoksen jälkeen muuttunut sisältö päivittyy takaisin komponentille *view-to-source-*

sidonnalla.

Tarkkailija-suunnittelumallia käytetään Angularissa muun muassa tapahtumien lähettämisessä tapahtumien kuuntelijoille ja asynkronisten HTTP-pyyntöjen ja vastausten käsittelyssä (*Observables in Angular*, 2010-2018). Tarkkailija-suunnittelumallissa tarkkailija voi rekisteröityä tapahtumia tuottavalle tapahtumien lähteelle (Koskimies, 2005). Angularissa tarkkailija-suunnittelumalli on toteutettu `Observable`-olioilla, joiden tuottamaa tapahtumavirtaa kuuntelemaan voi rekisteröityä kuuntelija, joka käsittelee tapahtumavirtaan saapuvat tapahtumat (*Observables*, 2010-2018). Tapahtumavirrassa käsiteltävät arvot voivat olla kontekstista riippuen eri tyyppisiä kuten esimerkiksi vakioita, viestejä tai tapahtumia (*Observables*, 2010-2018).

`Observable`-olioissa määritellään funktio, jossa määritellään `Observable`-olion tapahtumavirtaan tuotetut arvot (*Observables*, 2010-2018). Tätä `Observable`-oliossa määriteltyä funktiota ei kuitenkaan suoriteta ennen kuin kuuntelija rekisteröityy kuuntelemaan `Observable`-olion tapahtumavirtaa (*Observables*, 2010-2018). `Observable`-olio toimittaa tapahtumavirtaan rekisteröityneelle kuuntelijalle viestejä niin kauan kuin funktion suoritus kestää tai kunnes kuuntelija lakkaa kuuntelemasta tapahtumavirtaa (*Observables*, 2010-2018).

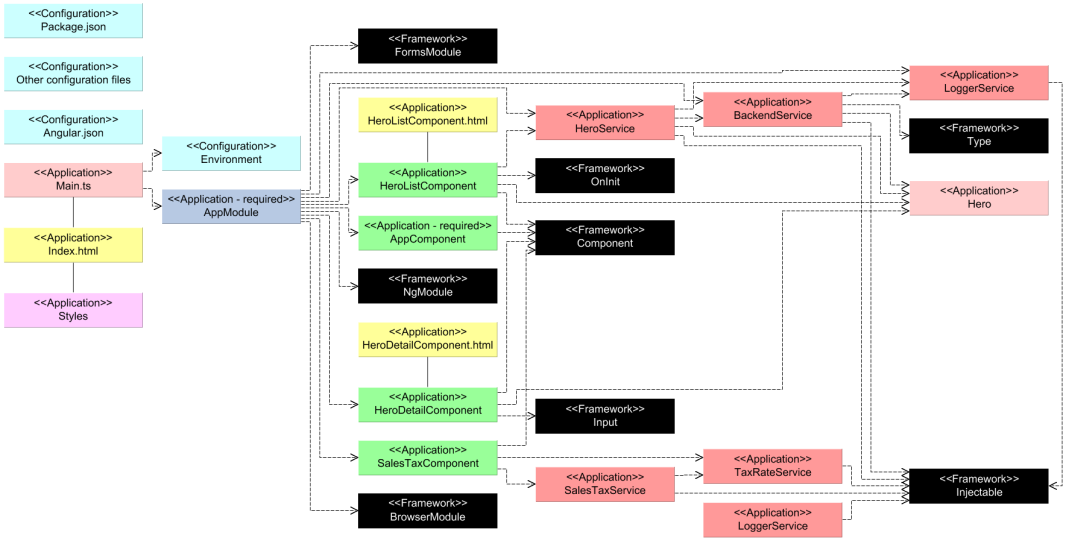
3.2 Sovelluskohtainen erikoistaminen

Angular-sovelluskehyksellä toteutettu sovellus koostuu sovelluskohtaisista osista ja Angular-kehykseen kuuluvista osista. Angular-kehys tarjoaa moduuleja sovellusten käyttöön importoitavina kirjastoina (*NgModules*, 2010-2018). Kuvassa 3.1 esitellään Angularin arkkitehtuuria kuvaavan esimerkkisovelluksen (*Example architecture application*, 2010-2018) luokkakaavio. Kuvassa Angular-sovelluskehyksen tarjoamat moduulit ja palvelut on merkitty mustilla laatikoilla ja stereotyypillä `<<Framework>>`. Sovelluskohtaiset ja sovelluskohtaisesti erikoistetut osat on merkitty kuvaan punaisilla, vihreillä, keltaisilla, liiloilla ja sinisillä laatikoilla ja stereotyypeillä `<<Application>>` ja `<<Application - required>>`. Kuvasta nähdään, että sovelluskohtaisesti erikoistettu juurimoduuli `AppModule` importoi Angular-kehyksen tarjoamat moduulit `FormsModule`, `NgModule` ja `BrowserModule` käyttöönsä.

Angular-kehys vaatii, että jokaisella sovelluksella on vähintään juurimoduuli, jonka sisällä on juurikomponentti. Kuvassa 3.1 `AppModule` ja `AppComponent` vastaavat

juurimoduulia ja -komponenttia. Angular-sovellukseen kuuluu myös konfiguraatiotiedostoja, joista esimerkiksi `Angular.json`-tiedostossa konfiguroidaan Angular CLI -komentorivityökalua (*QuickStart*, 2010-2018). Sovelluksen juurena toimivat `main.ts`- ja `index.html`-tiedostot. `Main.ts`-tiedostossa määritellään muun muassa sovelluksen juurimoduuli, josta sovellus käynnistetään ja `index.html`-tiedosto on sovelluksen pääsivu, joka näytetään käyttäjälle käyttäjän vieraillessa sovelluksen sivulla (*QuickStart*, 2010-2018).

Sovelluskohtaisesti Angular-sovelluksen erikoistaminen tapahtuu metadatan ja dekoraattorien avulla. Sekä komponenteilla, palveluilla että moduuleilla on jokaisella oma dekoraattori. Esimerkiksi `@Component`-dekoraattorilla määritellään Angular-sovelluksessa tavallisesta JavaScript-luokasta komponentti (*Architecture components*, 2010-2018). `@Component`-dekoraattorin metadatan avulla määritellään muun muassa, mitä palveluita komponenttiin injektoidaan ja mitä templaattia komponentti käyttää (*Architecture components*, 2010-2018).



Kuva 3.1: Angular esimerkkisovelluksen luokkakaavio

Esimerkissä 1 on esitelty `SalesTaxComponent`-komponentti (*Example architecture application*, 2010-2018). `@Component`-dekoraattorilla määritellään riveillä 5 ... 17 metadata, jonka avulla `SalesTaxComponent`-luokasta tehdään Angular-komponentti. Metadatassa määritellään muun muassa, että `SalesTaxService`- ja `TaxRateService`-palvelut injektoidaan riippuvuusinjektion avulla `SalesTaxComponent`-komponenttiin.

Lisäksi `@Component`-dekoraattorissa on määritelty `SalesTaxComponent`-komponentin templaatti riveillä 7 ... 15.

Esimerkki 1: SalesTaxComponent-komponentti (*Example architecture application*, 2010-2018)

```
1  import { Component }       from '@angular/core';
2  import { SalesTaxService } from './sales-tax.service';
3  import { TaxRateService }  from './tax-rate.service';
4
5  @Component({
6    selector:    'app-sales-tax',
7    template: `
8      <h2>Sales Tax Calculator</h2>
9      <label>Amount: <input #amountBox (change)="0"></label>
10
11     <div *ngIf="amountBox.value">
12     The sales tax is
13      {{ getTax(amountBox.value) | currency:'USD':true:'1.2-2' }}
14     </div>
15   `,
16   providers: [SalesTaxService, TaxRateService]
17 })
18 export class SalesTaxComponent {
19   constructor(private salesTaxService: SalesTaxService) { }
20
21   getTax(value: string | number) {
22     return this.salesTaxService.getVAT(value);
23   }
24 }
```

Esimerkissä 2 on esitelty `SalesTaxService`-palvelu (*Example architecture application*, 2010-2018). `@Injectable`-dekoraattorilla määritellään, että JavaScript-luokkaan voidaan injektoida muita palveluita (*Architecture services*, 2010-2018). `@Injectable`-dekoraattorin metadatan voidaan määritellä myös, että JavaScript-luokka on injektoitavissa komponenttiin riippuvuutena (*Architecture services*, 2010-2018). `@Injectable`-dekoraattori löytyy esimerkistä 2 riviltä 4. Tässä esimerkissä `@Injectable`-dekoraattorin käyttö mahdollistaa `TaxRateService`-palvelun injektoimisen `SalesTaxService`-palveluun rivillä 6.

Esimerkki 2: SalesTaxService-palvelu (*Example architecture application*, 2010-2018)

```
1  import { Injectable } from '@angular/core';
2  import { TaxRateService } from './tax-rate.service';
3
4  @Injectable()
5  export class SalesTaxService {
6    constructor(private rateService: TaxRateService) { }
7
8    getVAT(value: string | number) {
9      let amount = (typeof value === 'string') ?
10       parseFloat(value) : value;
11     return (amount || 0) * this.rateService.getRate('VAT');
12   }
13 }
```

`@NgModule`-dekoraattorin metadatan avulla kuvataan Angular-moduuli (*Architecture modules*, 2010-2018). Esimerkissä 3 määritellään Angular-esimerkkisovelluksen juurimoduuli `AppModule` (*Example architecture application*, 2010-2018). Esimerkissä `@NgModule`-dekoraattorin metadatassa on määritelty riveillä 13 ... 16 toiset moduulit, joita `AppModule`-moduulin sisällä olevat komponentit tarvitsevat. Riveillä 17 ... 22 määritellään `declarations`-ominaisuuden sisällä komponentit, jotka kuuluvat `AppModule`-moduulin sisälle. `declarations`-ominaisuuden sisällä voidaan määritellä myös direktiivit, jotka kuuluvat tietyn moduulin sisälle (*Architecture modules*, 2010-2018). `providers`-ominaisuudessa riveillä 23 ... 27 määritellään palvelut, joiden näkyvyysalue on koko sovelluksen laajuinen. Riviltä 28 löytyy viittaus `AppComponent`-juurikomponenttiin `bootstrap`-ominaisuuden sisältä.

Esimerkki 3: AppModule-moduuli (*Example architecture application*, 2010-2018)

```
import { BrowserModule }       from '@angular/platform-browser';
import { FormsModule }         from '@angular/forms';
import { NgModule }          from '@angular/core';
import { AppComponent }      from './app.component';
import { HeroDetailComponent } from './hero-detail.component';
import { HeroListComponent }   from './hero-list.component';
import { SalesTaxComponent }   from './sales-tax.component';
import { HeroService }         from './hero.service';
import { BackendService }      from './backend.service';
import { Logger }              from './logger.service';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    HeroDetailComponent,
    HeroListComponent,
    SalesTaxComponent
  ],
  providers: [
    BackendService,
    HeroService,
    Logger
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

4 Kehyksen arviointi

Angular web-sovelluskehyksellä toteutetun web-ohjelmiston arkkitehtuurissa on sekä hyviä että huonoja puolia. Angularissa käytetään `Observable`-olioita tarkkailija-suunnittelumallin toteuttamiseen. Yhtenä etuna `Observable`-olioiden käytössä on, että `Observable`-olion tapahtumien kuuntelijoille tarjoama rajapinta tapahtumavirtaan rekisteröitymiseksi on sama riippumatta siitä, tapahtuvatko tapahtumat asynkronisesti vai eivät (*Observables*, 2010-2018).

Angularissa hyödynnetään `Observable`-olioita myös Angularin sisäisessä toteutuksessa. Esimerkiksi Angularin sisäinen `HttpClient`-luokka palauttaa `Observable`-olioita HTTP-pyyntöjen vastauksina (*HttpClient*, 2010-2018). Yksi hyöty HTTP-pyyntöjen toteuttamisessa `Observable`-olioilla on, että tapahtumien kuuntelijan on mahdollista toistaa helposti `Observable`-olioiden avulla toteutetut HTTP-pyynnöt (*Observables in Angular*, 2010-2018). Tämä mahdollistaa myös HTTP-pyyntöjen helpon perumisen, sillä kuuntelija voi perua pyynnön rekisteröitymällä pois tapahtumavirrasta. Koska `Observable`-olioita hyödynnetään myös Angularin sisäisessä toteutuksessa, on kehittäjän helppo ottaa tämä ratkaisumalli käyttöön myös toteuttaessaan itse Angular-ohjelmistoa ja miettiessään oman ohjelmistonsa arkkitehtuuria.

Yksi haaste Angularin komponentteihin pohjautuvassa rakenteessa on, että sovelluksen jakaminen komponentteihin toimivalla tavalla voi olla haastavaa. Kysymyksiä, joita komponentteihin jaon yhteydessä herää ovat muun muassa: minkälainen Angular web-sovelluksen rakenne tulisi olla, kuinka komponenteista tehdään uudelleenkäytettäviä ja kuinka komponentit kommunikoivat keskenään (*Angular Architecture - Smart Components vs Presentational Components*, 2016)?

Angular web-sovelluskehyksellä toteutettujen ohjelmistojen yksi huono puoli on ollut, että jopa ”Hello World”-sovelluksen tiivistetyn version koko on vaihdellut 45Kb:n ja 36Kb:n välillä (Green, Wormald, Hevery, Eagle, & Larsen, 2016; Green, Erickson, & Hevery, 2018). Angulariin on kuitenkin kehitteillä uusi renderöijä Ivy, jonka avulla tiivistetyn ”Hello World”-sovelluksen koko on saatu pienennettyä 2,7Kb:n kokoiseksi (Green et al., 2018; Minar, 2018).

4.1 Laatuskenaariot

Laatuskenaarioiden avulla voidaan arvioida laatuominaisuuksia. Seuraavassa on ATAM-pohjaisia laatuskenaarioita, joiden avulla arvioidaan uudelleenkäytettävyyttä ja muunneltavuutta.

Laatuominaisuus: uudelleenkäytettävyys	
Tarkennus	Komponentin uudelleenkäyttö
Skenaario	Lomakekenttä-komponenttia voidaan hyödyntää sovelluksen eri osassa
Painotus (vaativuus, työmäärä)	(L, L)
Toteutumismahdollisuuksien kuvaus	Angularin komponenttipohjaisuus tukee ohjelmiston jakamista komponentteihin, joten uudelleenkäytettävien komponenttien tekemiseksi ei vaadita muutoksia sovelluskehykseen. Vaikka toimivasti jaoteltujen komponenttien toteuttaminen saattaa olla työlästä, on lomakekenttäkomponentti yksittäinen pieni ja tarkasti määritelty komponentti, joten toteuttamisen työmäärä ei olisi todennäköisesti suuri.

Laatuominaisuus: uudelleenkäytettävyys	
Tarkennus	Moduulien uudelleenkäyttö
Skenaario	`FormsModule`-moduulia voi hyödyntää oman ohjelmiston toteuttamisessa
Painotus (vaativuus, työmäärä)	(L, L)
Toteutumismahdollisuuksien kuvaus	Angularin modulaarinen arkkitehtuurityyli mahdollistaa moduulien vaivattoman käyttöönoton moduulien importoimisen avulla, joten Angular sovelluskehys tukee kirjastoitujen moduulien käyttöä. Kirjastoidut moduulit, kuten `FormsModule`-moduuli (*FormsModule*, 2010-2018), voidaan ottaa käyttöön helposti seuraavasti: `import { FormsModule } from '@angular/forms';`

Laatuominaisuus: muunneltavuus	
Tarkennus	JavaScriptin käyttäminen Angularissa
Skenaario	Siirtyminen TypeScriptin käyttämisestä JavaScriptin käyttämiseen ohjelmiston toteuttamisessa
Painotus (vaativuus, työmäärä)	(M, H)
Toteutumismahdollisuuksien kuvaus	Angular web-sovelluskehys on toteutettu TypeScript-ohjelmoitikielellä toteutetuilla kirjastoilla. Siirtyminen käyttämään omassa ohjelmistossa JavaScriptiä TypeScriptin sijaan määrittelee Angularin rajoja. Siirtyminen JavaScriptin käyttöön on mahdollista, mutta se voi olla työlästä.

5 Yhteenveto

Angular web-sovelluskehys on tarkoitettu asiakaspuolen web-ohjelmistojen toteuttamiseen. Angularilla toteutetut ohjelmistot koostuvat TypeScript-ohjelmointikielellä toteutetuista kirjastoista. Angularilla toteutetut sovellukset koostuvat useasta pääosasta, joita ovat moduulit, komponentit ja palvelut. Arkkitehtuurityylejä Angularissa ovat modulaarisuus ja komponenttipohjaisuus, ja Angularissa sovelletaan lisäksi useita ohjelmistotason ratkaisumalleja.

Yksi Angularin ohjelmistotason ratkaisumalleista on tarkkailija-suunnittelumalli, joka on toteutettu `Observable`-olioiden avulla. `Observable`-olioiden käytössä on useita etuja, mutta Angularissa käytetyissä arkkitehtuurityyleissä on myös huonoja puolia. Esimerkiksi komponenttipohjaisuudessa haasteena on jakaa sovellus uudelleenkäytettäviin komponentteihin.

Lähteet

Angular architecture - smart components vs presentational components. (2016, Päivitetty 18. kesäkuuta 2018). Tarkistettu saatavilla `https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/` (Luettu 24.9.2018)

Architecture components. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/architecture-components` (Version 6.0.3-build.44499+sha.43ee10f.)

Architecture modules. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/architecture-modules` (Version 6.0.3-build.44499+sha.43ee10f.)

Architecture overview. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/architecture` (Version 6.0.3-build.44499+sha.43ee10f.)

Architecture services. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/architecture-services` (Version 6.0.3-build.44499+sha.43ee10f.)

Dependency injection. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/dependency-injection` (Version 6.0.3-build.44499+sha.43ee10f.)

Dependency injection pattern. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/dependency-injection-pattern` (Version 6.0.3-build.44499+sha.43ee10f.)

Example architecture application. (2010-2018). Tarkistettu saatavilla `https://angular.io/generated/zips/architecture/architecture.zip` (Version 6.1.9-build.49423+sha.0ec925b.)

Formsmodule. (2010-2018). Tarkistettu saatavilla `https://angular.io/api/forms/FormsModule` (Version 6.1.9-build.49564+sha.37f3b92.)

Green, B., Erickson, K., & Hevery, M. (2018). *Ng-conf day 1 keynote 2018 - public.* Tarkistettu saatavilla `https://docs.google.com/presentation/d/1zgpjyVkDgUPfGKuxOcU0lLusCiMSfLZZjYHWrFvl71I/edit#slide=id.g389359a262_0_212` (Luettu 24.9.2018)

Green, B., Wormald, R., Hevery, M., Eagle, A., & Larsen, H. (2016). *Day 2 keynote ng-conf 2016 public.* Tarkistettu saatavilla `https://docs.google.com/presentation/d/1pqn5uhqg1km1AXR15qugtAYV-bnMtDl_HC1HnK9b1xQ/edit#slide=id.g12e00921d2_19_90` (Luettu 24.9.2018)

Httpclient. (2010-2018). Tarkistettu saatavilla `https://angular.io/api/common/http/HttpClient` (Version 6.1.9-build.49564+sha.37f3b92.)

Koskimies, K. (2005). *Ohjelmistoarkkitehtuurit* (T. Mikkonen, toim.). Helsinki: Talentum.

Minar, I. (2018). *Ivy renderer (beta).* Tarkistettu saatavilla `https://github.com/angular/angular/issues/21706` (Luettu 24.9.2018)

Ngmodules. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/ngmodules` (Version 6.0.3-build.44499+sha.43ee10f.)

Observables. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/observables` (Version 6.1.9-build.49423+sha.0ec925b.)

Observables in angular. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/observables-in-angular` (Version 6.1.9-build.49423+sha.0ec925b.)

Quickstart. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/quickstart` (Version 6.0.3-build.44499+sha.43ee10f.)

Template syntax. (2010-2018). Tarkistettu saatavilla `https://angular.io/guide/template-syntax` (Version 6.0.3-build.44499+sha.43ee10f.)