MODULAARINEN MOBIILISOVELLUS

Case: Digimaa

LAB-AMMATTIKORKEAKOULU
Insinööri (AMK)
Tieto- ja viestintätekniikka
Ohjelmistotekniikka
Opinnäytetyö
Kevät 2020
Karri Miettinen

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Miettinen, Karri	Opinnäytetyö, AMK	Kevät 2020
	Sivumäärä	
	39	

Työn nimi

Modulaarinen mobiilisovellus
Case: Digimaa

Tutkinto

Insinööri AMK, tietotekniikka

Tiivistelmä

Opinnäytetyössä tutkittiin, miten modulaarisen web-sovelluksen voi kehittää Angular-ohjelmistokehyksellä. Opinnäytetyö toteutettiin Lahden Ammattikorkeakoululle osana Digimaa-projektia.

Työssä modulaariseen web-sovelluskehitykseen tutustutaan trendien avulla, mitkä osoittivat osaltaan, että modulaarisuuden suosio on kasvussa. Modulaarisuutta tutkitaan Angularin kehitysympäristön avulla ja tutustutaan sovellustyökaluihin.

Opinnäytetyössä perehdytään Angularin arkkitehtuuriin ja osaan sen tuomista ominaisuuksista. Arkkitehtuurista tutkitaan lähinnä sovelluksen modulaarisuuteen vaikuttavia piirteitä.

Työssä modulaarisuuden soveltamiskohteita suunnitellaan näkymäluonnosten avulla, joista osakokonaisuudet käyvät parhaiten ilmi. Havaintojen perusteella aletaan toteuttaa sovelluksen rakennetta.

Modulaarisen sovelluksen toteutuksessa käydään läpi projektin tiedostorakenteita sekä luotuja moduuleja ja sovellukseen tuotettuja ominaisuuksia. Toteutuksesta tuodaan myös esille roolipohjainen autentikoituminen, jolla hallitaan sovelluksen eri toimijoita.

Opinnäytetyön lopputuloksena valmistui käyttöliittymä, joilla pystytään lähettämään tilauksia ja käsittelemään sitä tilauksen eri vaiheissa. Sovelluksen käyttöliittymä on jo julkinen ja siitä on tehty sovellus, mutta käyttöliittymä osalta tarvitaan vielä jatkokehitystä.

Asiasanat

moduulaarisuus, mobiili, web-sovellus, Digimaa, Angular

Abstract

Author(s)	Type of publication	Published
Miettinen, Karri	Bachelor's thesis	Spring 2020
	Number of pages	
	39	

Title of publication

Modular mobile application
Case: Digimaa

Name of Degree

Bachelor of Engineering

Abstract

This thesis deals with Angular framework's capabilities in producing a modular application. The thesis was made in cooperation with the Digimaa Project of Lahti University of Applied Sciences.

The study first examines the trends of modular web-designs, that indicates its popularity is growing. Then it investigates the Angular development environment and application development tools.

The thesis presents Angular's architecture and its features. Architecture is observed here from the application's modular perspective.

The scope of modularity is designed with the help of the visual drafts, where the subdivisions are easier to perceive. Here the application's structure starts to shape with the help of these observations.

Development of the modular application focuses on the file structure, modules and features that are developed in this project. This implementation also shows the role-based access control system, which controls the user's permissions in the application.

As result of this thesis, the application user-interface was deployed, and it can send and handle orders. The application user-interface is already public and there is an application bundle, but the application bundle is not public. The user-interface is currently waiting for further development.

Keywords

module, mobile, application, Digimaa, Angular

SISÄLLYS

LYHENTEET

API	Application Programming Interface, Ohjelmiston ohjelmarajapinta
CLI	Command-line Interface, Komentoliittymä
CSS	Cascading Style Sheets, Verkkosivujen tyylien merkintäkieli
DI	Dependency Injection, Ohjelmistoarkkitehtuurissa käytettävä riippuvuusinjektio malli, jolla erotellaan ohjelmariippuvuuksia toteutuksista.
DOM	Document Object Model. Dokumenttioliomalli, jota voi käsitellä ohjelmallisesti
GIT	GIT, Versionhallintajärjestelmä
HTTPS	Hypertext Transfer Protocol Secure, Salattu hypertekstin siirtoprotokolla
JavaScript	Verkkosivujen dynaamisuuden ohjelmointikieli
JSON	JavaScript Object Notation, JavaScriptissä olevien objektien ilmaisu merkkijonona
JWT	JSON Web Token, Käyttöoikeus JSON-tyylillä ilmaistuna
NPM	Nuget Package Manager, Pakettien hallintatyökalu
RBAC	Role-Based Access Control, Rooliin perustuvat käyttöoikeudet
REST	Representational State Transfer, Ohjelmointirajapinnan arkkitehtuurimalli
SCSS	Sassy CSS, Laajennus CSS:ään
SPA	Single Page Application, yhden sivun web-sovellus
TypeScript	Tyyppilaajennus Javascript-kieleen
URL	Uniform Resource Locator, web-sivuston osoite, jolla merkitään tiedon sijaintia

1 JOHDANTO

Web-sovelluskehitys on kehittymässä modulaarisempaan suuntaan (Kalvi Group 2020). Monen sovelluskehittäjän mielessä tässä kohtaa onkin yleensä sovellusten laadun parantaminen. Modulaarisuudella haetaan sovelluksiin toimintavarmuutta ja samalla taataan niille paremmat jatkokehitysmahdollisuudet.

Modulaarisuus mahdollistaa sovellusten kehittämisen osissa, jolloin osista voidaan luoda sovelluksille isoja kokonaisuuksia ja osia voidaan kehittää kokonaisuuksista riippumattomina. Sovellusten kehittäjät pystyvät näin jakamaan omaa työtään myös osiin ja pitämään kasvavien sovellusten kokonaisuudet paremmin hallinnassaan.

Moduulaarinen sovelluskehitys helpottaa usein myös sovelluksien testaamista. Kun testejä suoritetaan moduulikohtaisesti, voidaan varmistua pienistä yksityiskohdista ja vähentää näin monimutkaisten kokonaisuuksien testikuormaa. Sovellusten toimintavarmuus kasvaa kattavamman testauksen ansiosta. (Miller 2020.)

Modulaarisuudella pystytään toteuttamaan joskus myös sovelluksissa usein toistuvia asioita pienemmällä koodimäärällä. Kun jokin ominaisuus toistuu tarpeeksi usein, oli se sitten jonkun nappulan muoto tai merkkijonon käsittelytoimenpide, sen koodista voidaan toteuttaa oma pieni kokonaisuutensa, jota käytetään tarvittaessa. Koodin hallinnasta tulee näin selkeämpää ja hallittavampaa myös jatkokehitystä ajatellen.

Opinnäytetyö pohjautuu LAMK:n ja Aalto yliopiston yhteiseen Digimaa-projektiin, jossa suunnitellaan ja toteutetaan maanrakentamiseen liittyvää pilvipalvelua. Palvelun tarpeen taustalla on lisääntyvä tiedonkäsittelyn tarve maanrakennusalalla eri toimijoiden välillä. (Aalto-yliopisto 2019.)

Opinnäytetyön tavoitteena on toteuttaa modulaarinen mobiilikäyttöliittymä Digimaan pilvipalveluun. Mobiilikäyttöliittymästä on tarkoituksena hallita maa-ainesten tilausketju, jonka avulla tieto välittyy tilausten eri osapuolille. Tilausketjun toimijat jaetaan eri rooleihin, rooleja ovat tilaaja, toimittaja, kuljettaja ja vastaanottaja.

Tässä opinnäytetyössä keskitytään mobiilikäyttöliittymän modulaariseen arkkitehtuuriin. Toteutus lähtee liikkeelle Angularin ohjelmistokehyksen avulla, missä aluksi luodaan Angular-ympäristö ja sen jälkeen tutkitaan sen valmiita rajapintoja ja muita ominaisuuksia. Toteutukseen pyritään löytämään Angularin avulla modulaarinen ratkaisu, jolla sovellukselle saadaan selkeä rakenne ja luotua jatkokehityksen edellytykset.

Sovellus toteutetaan web-sovelluksena ja siitä toteutetaan myös mobiiliversio Androidille. Käyttöliittymän muotoillaan mobiilialustoille sopivaksi. Sovelluksen taustajärjestelmää

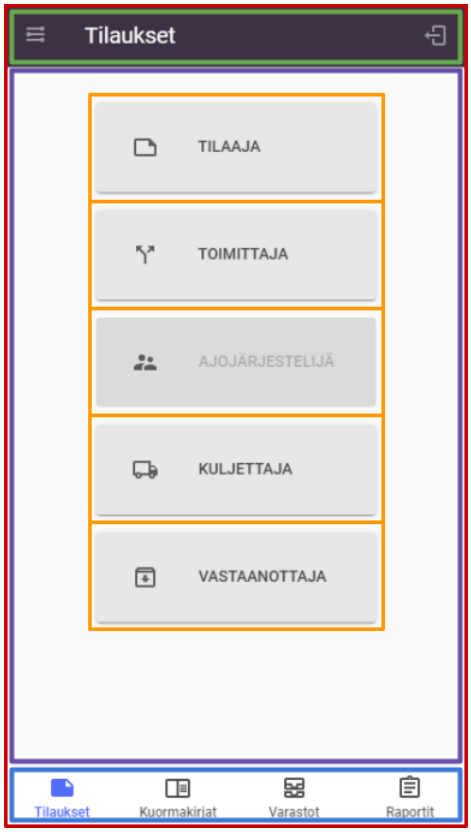
käsitellään tässä opinnäytetyössä vain sen rajapinnan kautta, eikä sen sisäisiin toimintoihin oteta tässä opinnäytetyössä kantaa.

2 MODULAARISEN WEB-SOVELLUKSEN KEHITYSYMPÄRISTÖ

2.1 Modulaarisuus sovelluksessa

Sovelluksissa modulaarisella rakenteella pyritään joustavuuteen, johdonmukaisuuteen ja se mahdollistaa rakenteiden uudelleenkäytön (Claire Patenaude, 2019). Moduulien vastuu on usein selkeästi hahmotettavissa esimerkiksi luonnoksista ja jo sovellusten suunnitteluvaiheessa on mahdollista jakaa sovellusta moduuleiksi.

Kuvassa 1 on esitetty, miten käyttöliittymä on suunniteltu modulaarisesti. Näitä modulaarisia osia käyttöliittymässä kutsutaan usein komponenteiksi. Kuvan 1 vihreän komponentin on tarkoitus mahdollistaa navigointi näkymissä. Sen alla violetin komponentin sisällä on yleinen sivun sisältö ja sinisen komponentin sisällä on välilehtien vaihdin. Oranssien alueiden sisälle on suunniteltu yleiskäyttöinen painikekomponentti.



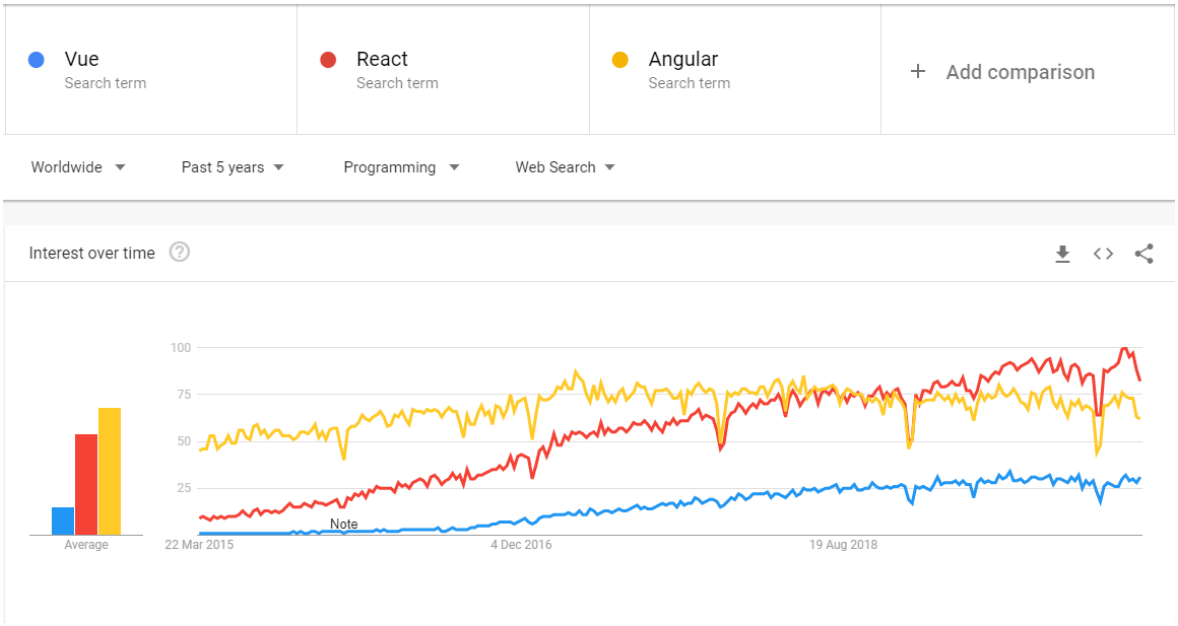
Kuva 1. Komponenttien jaottelu

2.2 Kehitysympäristön valinta

Modulaarisuuden toteuttamisen kannalta ei ole välttämätöntä valita tiettyä ohjelmointikieltä tai ohjelmistokehystä, mutta valinnalla on paljon vaikutusta siihen, millainen rakenne sovellukselle tulee. Ohjelmistokehystä valitessa on kuitenkin hyvä ottaa huomioon

jatkokehitettävyys, sovellusarkkitehtuuri ja käyttöliittymä. Tällaisissa tapauksissa ohjelmistokehyksen suosiosta on hyötyä, eli mahdollisimman moni ohjelmoija tuntisi ohjelmistokehyksen.

Digimaa-projektissa pohjalle valittiin Angular 5, joka myöhemmin päivitettiin Angular 7-versioon. Valintaan vaikuttivat sen pitkä kehityshistoria ja suosio, kuten kuvasta 2 voidaan havainnoida. Komponenttirakenteen tärkeys projektille vaikutti myös valintaan, mistä kerrotaan myöhemmissä luvuissa. Angularissa nähtiin myös hyvät edellytykset tiimityöskentelylle sen rakenteen jaettavuuden ansiosta. Kuvassa 2 on esitetty Googlen trendit Vuen, Reactin ja Angularin osilta vuosina 2015 - 2020.



Kuva 2. Frontend trendit 2015-2020 (Google 2020)

2.3 Sovellustyökalut

Koodin kehittämiseen valittiin Microsoftin Visual Studio Code. Se on tehokas, ilmainen ja avoimen lähdekoodin koodieditori. Angular-kehittäjille on saatavilla siinä myös lisäosia, joista on hyötyä sovelluksen kehityksessä.

Angularin kehittäminen vaatii myös Node.js ja NPM (Nuget Package Manager) asennettuna. Node.js mahdollistaa JavaScriptin suoran ajamisen koneella, mikä taas mahdollistaa esimerkiksi NPM:n toiminnan. NPM:llä taas hallitaan JavaScript kirjastojen asentamista, mikä on myös edellytys Angular-kirjastojen asentamiselle.

Sovelluksen kehittämiskokemuksen parantamiseksi lisättiin Visual Studio Codeen myös lisäosia, kuten TSLint, Angular Language Service, JavaScript and TypeScript IntelliSense, npm Intellisense, Path intellisense, SCSS IntelliSense, SCSS Everywhere ja REST Client.

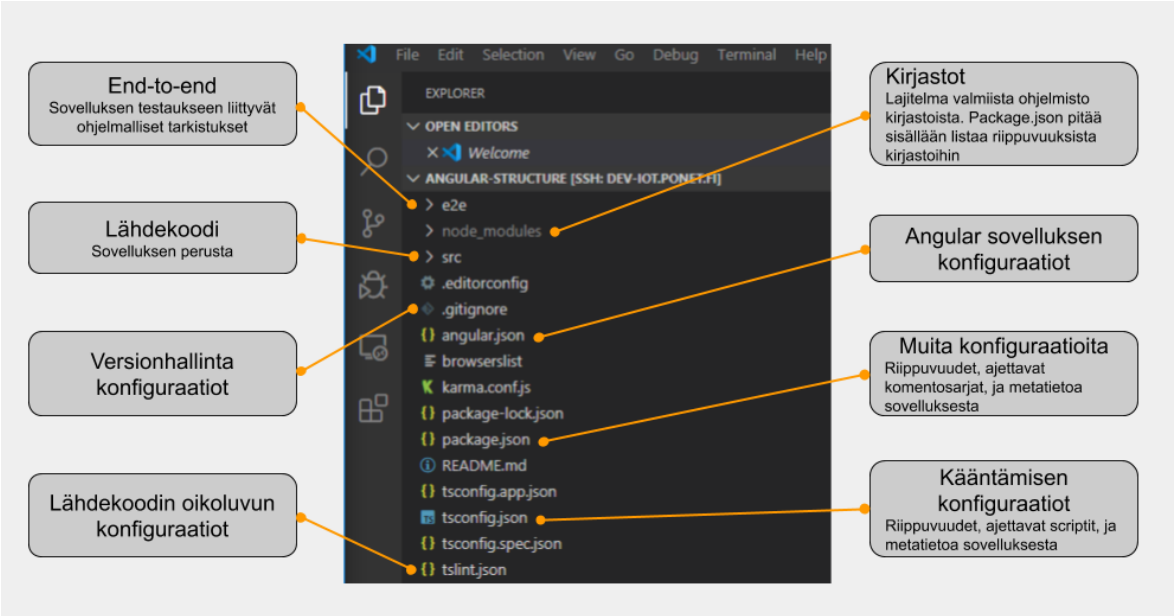
2.4 Kehitysympäristö

Angular-kehitysympäristö luodaan heti projektin alussa. Selkein tapa Angular-kehitysympäristön luomiseen on käyttää Angularin komentoliittymää (CLI). Se toteutetaan komennolla, johon laitetaan tulevan projektin nimi ja sen jälkeen komentoliittymään tulee tarvittavat kysymykset kehitysympäristön alustusta varten.

Alustuksessa voidaan määritellä sovelluksen ominaispiirteitä. Nimi kannattaa määrittää mahdollisimman kuvaavaksi ja ytimekkääksi, jotta se on yksiselitteinen. Kehitysympäristön alustus Angularin komentoliittymällä asentaa Angularin tärkeimmät riippuvuudet ja työkalut. Se luo myös valmiudet sovelluksen tuotantoympäristöön viemiselle.

Piirteitä valittaessa kehitysympäristöön generoituu automaattisesti lisää rakenteita. Tarvittaessa kaikki on muokattavissa jälkeenpäin, mutta perusrakennetta ei kannata muokata liikaa. Näin kehitysympäristö säilyy samankaltaisena muiden Angular-projektien kanssa.

Kuvassa 3 on esiteltynä komentoliittymällä luodun Angular-projektin kansiorakenne päätasolta. Sovellukseen liittyvät konfiguraatiot ovat päätasolla tiedostoina ja niiden rinnalle on luotu kolme alihakemistoa, joista src pitää sisällään lähdekoodia, node_modules käsittää projektiin liittyviä kirjastoja ja e2e hakemisto on sovelluksen testausta varten.



Kuva 3. Angularin projektirakenne

Projektit lähtevät liikkeelle isohkoina kokonaisuuksina. Tähän syynä on suuri määrä valmiita ominaisuuksia ja perusrakenteen laajuus. Edellytykset kokonaisen SPA:n toteutukseen on mietitty kokonaisuutena ja näillä pärjää usein pitkälle sovelluksen kehityksessä.

2.5 Typescript

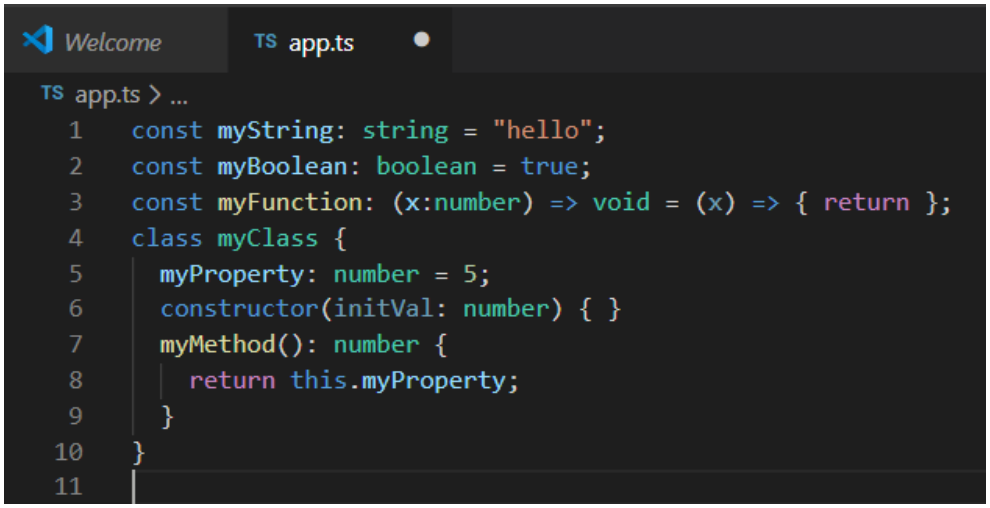
Angularin uudemmissa versioissa (>2) on siirrytty käyttämään TypeScriptiä ohjelmointikielenä. Sen ideana web-ohjelmoinnissa on lisätä tyyppiturvallisuutta ja se pohjautuu monille web-ohjelmoijille tuttuun JavaScriptiin.

> *TypeScript on tyypitetty laajennus JavaScriptille, mikä kääntyy tavalliseksi JavaScriptiksi* (Microsoft 2019).

Laajennuksella tarkoitetaan tässä tapauksessa, että kehittäjä voi käyttää tavallista JavaScriptiä TypeScriptin sisällä, ja näin hyödyntää molempien kielien ominaisuuksia. TypeScript eroaa sen verran JavaScriptistä, että selaimet eivät tue sitä suoraan. Tämän sijaan kaikki tuotettu koodi on käännettävä etukäteen selaimille sopivaksi. Koodin kääntämisellä saavutetaan yleensä suorituskykyisempää koodia, sekä koodin luettavuus ennen kääntämistä on yleensä selkeämpää.

TypeScriptiä varten on oma tiedostopäätteensä (.ts). Kääntämisen aikana kaikki ts-tiedostot, jotka on määritelty projektiin, käännetään JavaScriptiksi (.js). Kääntämisprosessia voidaan hallita komentoparametreillä tai TypeScript-konfiguraatiolla. Käännetty versio on suoritettavissa, joko NodeJS:llä tai HTML:ään upotettuna selaimesta tietyillä edellytyksillä.

TypeScriptin tyyppiturvallisuus mahdollistaa muuttujien ja funktioiden tyypittämisen ennen ajoa. Tyypitys antaa kehittäjälle tiedon odotetusta datan muodosta, mikä ei ole mahdollista JavaScriptissä, missä tyypitykset ovat automaattisia. Kuvassa 4 on esitelty TypeScriptin datatyyppejä, nuolifunktio ja luokka.



Kuva 4. TypeScriptin ominaisuuksia

TypeScript mahdollistaa myös työkalujen, kuten IntelliSensen paremman toimivuuden. Intellisense tuo kehittäjille ehdotuksia soveltuvista koodivaihtoehdoista ja kun kieli on

tyypitetty, pystyy IntelliSensekin tuomaan tarkempia ehdotuksia koodin suhteen. Koodiehdotukset IntelliSensessä pohjautuvat saatavilla oleviin funktioihin ja muuttujiin.

3 ANGULAR-OHJELMISTOKEHYS

3.1 Yleisesti Angularista

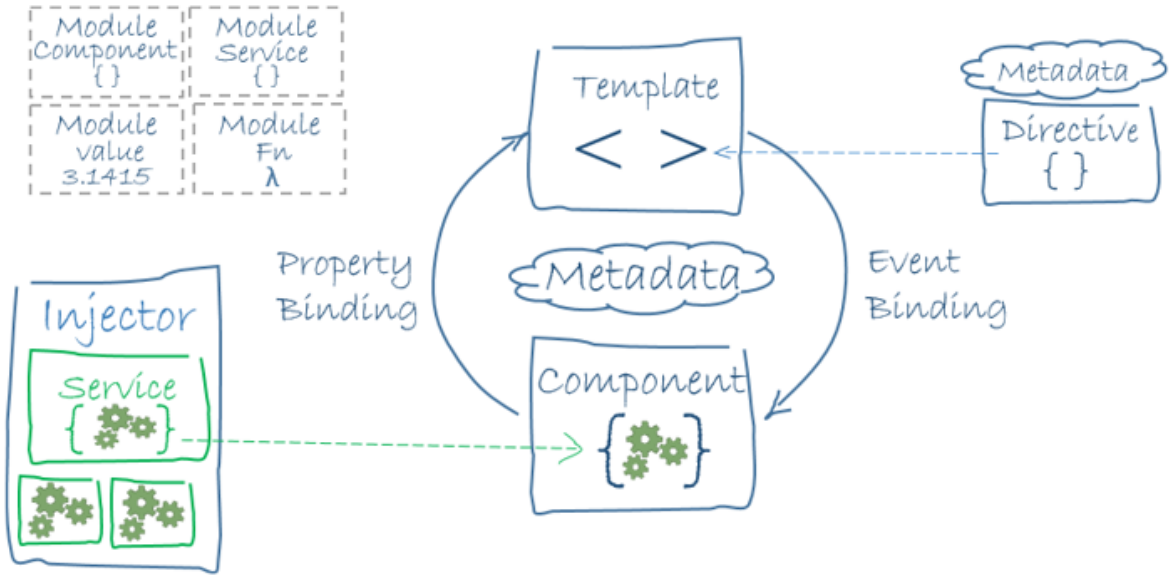
Angular on Googlen kehittämä ohjelmistokehys web-pohjaisten sovellusten kehittämiselle. Sen ideologia on toteuttaa web-sivustosta yksi iso kokonaisuus ja näin vähentää esimerkiksi latausaikaa sivujen vaihdon välillä. (Google, 2020a).

Alun perin AngularJS niminen kirjasto keskittyi dynaamisen datan liittämiseen ja uusien HTML-elementtien luomiseen. Myöhemmin julkaistu Angular 2.0 oli täydellinen AngularJS kirjaston uudelleenkirjoitus. Uudessa Angular 2-versiossa keskityttiin enemmän mobiililaitteisiin, sovellusarkkitehtuuriin ja silloin uuteen TypeScript-ohjelmointikieleen. Myöhemmin julkaistuissa versioissa 4, 5, 6, 7, 8 ja 9 on kehitetty samaa ideaa eteenpäin, mikä version 2 mukana tuli.

Nykyinen Angular on oma kokonaisuutensa ja tukee kokonaisen web-sovellusten toteutuksen. Arkkitehtuuri koostuu monista Angulariin kehitetyistä osista ja näille on määritelty selkeät vastuualueet sovelluslogiikassa. Ulkopuolisten JavaScript kirjastojen liittäminen on myös mahdollista Angularissa. Angular ohjelmistokehys soveltuu myös mobiilisovelluksen toteutukseen. Mobiilisovelluksen voi toteuttaa käärimällä Angular-tuotoksen omaksi web-sovellukseksi. Kokonaisuus pysyy hallussa SPA-toteutuksen ansiosta.

3.2 Arkkitehtuuri

Angularin arkkitehtuuri perustuu valmiiseen pohjaratkaisuun, jossa perusdatan kuljettaminen ja käyttöliittymän reagointi on toteutettu Angularin ajoympäristössä. Kuvassa 5 on havainnollistettu Angularin logiikkaa, josta voidaan sanoa, että kaikki datan siirtyminen on Angularin vastuulla. Sovelluskehittäjän tehtävä on seurata rakennetta ja toteuttaa ohjelmiston osat niille kuuluvin ohjelmistorajapintojen mukaisesti.



Kuva 5. Angularin arkkitehtuuri (Google 2020c)

Angular arkkitehtuurin toteutuminen vaatii siis olemassa olevien käytäntöjen noudattamista ja ohjelmarajapintojen käyttämistä. Arkkitehtuurin osat tulevat myöhemmissä kappaleissa esille.

3.3 Valmiit ohjelmarajapinnat

Angularissa sovelluskehyksenä on laaja valikoima jo valmiita ohjelmarajapintoja (API). Angular 7-versiossa valmiita rajapintoja dokumentaation mukaan on 569 (Google, 2019). Näiden rajapintojen tarkoitus on tuoda hyviksi todettuja web-sovelluksen ominaisuuksia valmiina kehittäjille. Näiden rajapintojen avulla myös luodaan Angular-arkkitehtuurin tärkeimmät osat, kuten moduulit, reitit ykset, komponentit, palvelut, direktiivit ja putket.

Muita Angularin valmiita ohjelmarajapintoja käytetään sovelluksen ominaisuuksien rakentamisessa, esimerkiksi sovellukseen liittyvien lomakkeiden, kalentereiden ja painikkeiden luomiseen. Niiden tehtävä on nopeuttaa sovelluksen kehitystä valmiilla toimivilla konsepteilla.

3.3.1 Reaktiiviset lomakkeet

Angularissa on valmiina reaktiivinen lomakemalli. Sen rakenne ja toiminnallisuus palvelee lomakkeille tyypillisiä ominaisuuksia. Lomakkeet, jotka on toteutettu tällä tyylillä, pystytään liittämään myös Angular-materiaalin lomake-elementteihin.

Reaktiivisen lomakkeen luonti tapahtuu ReactiveFormsModulesta löytyvällä FormGroup-luokan avulla. FormGroup luokalla tarkoitetaan ryhmää lomakkeen syötekentistä ja sillä

hallitaan koko lomakkeen tilaa. Lomakeluokka pystyy itsenäisesti tarkastamaan esimerkiksi sen sisällä olevien syötekenttien kelpoisuutta.

Kuvassa 6 on esitelty FormGroup-luokan kokonaisuutta, josta lomakkeen tilaa voidaan tarkastella. Tilasta voidaan tarkkailla, onko lomake täytetty oikein, tai onko lomakkeeseen koskettu, sekä lomakkeeseen syötetyt arvot ovat value ominaisuudesta luettavissa.

Kuva 6. Angular Reactive FormGroup

Lomakkeen varsinaiset syötekentät voidaan luoda joko FormGroup-instanssin luonnin aikana tai myöhemmin käyttämällä FormGroupissa olevaa addControl-metodia. Syötekentät luodaan käyttämällä ReactiveFormsModulen FormControl-luokkaa. Syötekenttäluokalle voidaan määrittää lomakkeessa nimi ja sen jälkeen luokalle määritellään siihen liittyvät ominaisuudet, kuten sen alustusarvo ja validaattorit.

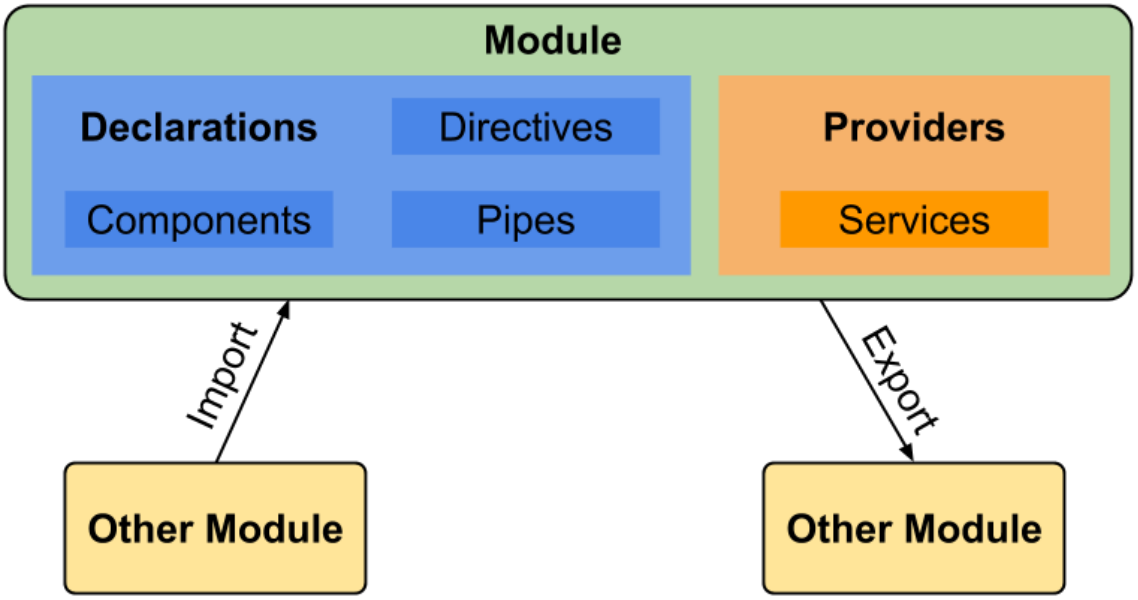
Kun lomakkeen instanssi on luotu, voidaan sille toteuttaa ulkoasu HTML:llä. Mikäli syötekentät ovat perinteisiä tekstikenttiä tai valikkoja, voidaan käyttää valmiita ulkoasukomponentteja, kuten mat-form-field tai mat-select, jotka löytyvät Angularin materiaalikirjastosta. Kirjastosta löytyy myös muita valmiita lomake-elementtejä ja mikäli valmiit elementit eivät riitä, voidaan syötekentät rakentaa Angularissa itse.

Lomakkeen syötekenttiä voidaan hallita metodien avulla ja niiden tilaa voidaan lukea ulkoapäin. Syötekenttiin on upotettu esimerkiksi valueChanges-metodi, jolla voidaan ohjelmallisesti viestiä, mikäli syötekentässä tapahtuu muutos. Ohjelmassa tieto syötekentän muutoksista välittyy niille, jotka muutoksen ovat tilanneet.

Lomakkeiden syötekenttiin voidaan myös upottaa validaattoreita, millä voidaan tarkastella syötekentän sisällön oikeellisuutta. Virheiden tilaa voidaan seurata myös ohjelmallisesti suoraan luokan instanssista, sekä lomake voidaan tarkastaa ennen lähettämistä hasError-metodilla.

3.4 Moduulit

Moduulit ovat Angularin ohjelmarajapinnan korkein taso. Moduulilla tarkoitetaan Angularissa pakettia, mihin on ryhmitelty Angularin osia, kuten komponentit, palvelut, putket ja luokat. (Google, 2020d). Moduuliin voi lisätä tarvittaessa myös reititystä, mikäli moduuliin ollaan toteuttamassa käyttöliittymiä. Moduuliin voidaan liittää myös muita moduuleita Import toiminnolla ja moduulista voidaan viedä ulospäin sen ominaisuuksia Export toiminnolla, kuten kuvassa 7.



Kuva 7. Angularin moduulirakenne

Angularin parhaiden käytäntöjen mukaan moduuleja on kolmen tyyppisiä. Perinteisesti Angular projektin moduulit järjestellään niin, että siellä on jaettu moduuli, ydinmoduuli ja sen jälkeen muut moduulit. (Rik de Vos, 2019).

Jaettuun moduuliin (shared module) toteutetaan yleensä sellaisia osia, mitä on tarkoitus käyttää uudelleen muissa moduuleissa (Rik de Vos, 2019). Esimerkiksi itse toteutettu painike voi olla sellainen, mitä halutaan käyttää muissa paikoissa. Kuvassa 8 on esitelty jaetun moduulin rakenne.

```
import { NgModule, ModuleWithProviders } from '@angular/core';
import { CommonModule } from '@angular/common';
//#region imports···

@NgModule({
  imports: [···
  ],
  declarations: [···
  ],
  entryComponents: [···
  ],
  exports: [···
  ],
  providers: [···
  ]
})
export class SharedModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: SharedModule,
      providers: [···
      ]
    };
  }
}
```

Kuva 8. Jaetun moduulin rakenne

Ydinmoduuliin (core module) lisätään yleensä sellaisia ominaisuuksia, mitkä ladataan yleensä sovelluksen alussa ja vain kerran. Ydinmoduulin palvelut ovat hyödynnettävissä muissa moduuleissa, mutta niitä ei monisteta, koska niissä pyritään säilyttämään yhtenäistä tilaa. (Rik de Vos, 2019.)

Muissa moduuleissa, kuten ominaisuusmoduuleissa (feature module) toteutetaan yleensä käyttöliittymän eri näkymiä. Käyttöliittymissä pyritään hyödyntämään jaetun moduulin ominaisuuksia mahdollisimman paljon, jolloin toistuvaa koodia ei tarvitse toteuttaa. Yksi ominaisuusmoduulin tärkeistä tehtävistä on myös lähdekoodin organisointi. Moduulin voi nimetä vapaasti, mutta sen olisi hyvä kuvastaa sille kuuluvaa vastuualuetta.

> *Koristeet ovat ydinkonsepti, kun kehitetään Angularilla* (Todd Motto, A deep dive on Angular decorators. Ultimate Courses 2017).

Moduulit rakennetaan Angular core:sta tuotavalla NgModule luokka koristeella (Class decorator). Luokan tärkeimpiin ominaisuuksiin kuuluvat import, export, declarations, providers, entryComponents ja bootstrap.

Jotta moduulissa voitaisiin ottaa käyttöön direktiivejä, on ne määriteltävä declarations-listassa. Angular huolehtii näin siitä, että direktiivit ovat käytettävissä moduulin sisäisesti. Palvelut julkaistaan vuorostaan providers listassa ja ovat sitä kautta moduulin käytössä. EntryComponents ominaisuutta käytetään, jos jossain kohtaa tarvitsee luoda komponentteja dynaamisesti. Tätä käytetään esimerkiksi Angular Materiaali dialogien luonnissa. Bootstrap ominaisuutta käytetään yleensä vain ylimmässä, eli AppModulessa ja vain yhdelle komponentille. Sen tarkoitus on käynnistää Angular applikaatio index.html:stä. Jos komponentteja on useampi, niin Angular käynnistää useamman Angular applikaation ja se on harvoin toivottua.

3.5 Reititys

Reitittäminen Angularissa on toteutettu SPA-tyylillä. Reititystapoja Angular tarjoaa kahdenlaista, mutta kummassakin idea on, että sovelluksen ei tarvitse lähettää kyselyitä erikseen palvelimelle. Angularissa reittien sisältö haetaan jo ladatuista moduuleista. (Google, 2020c).

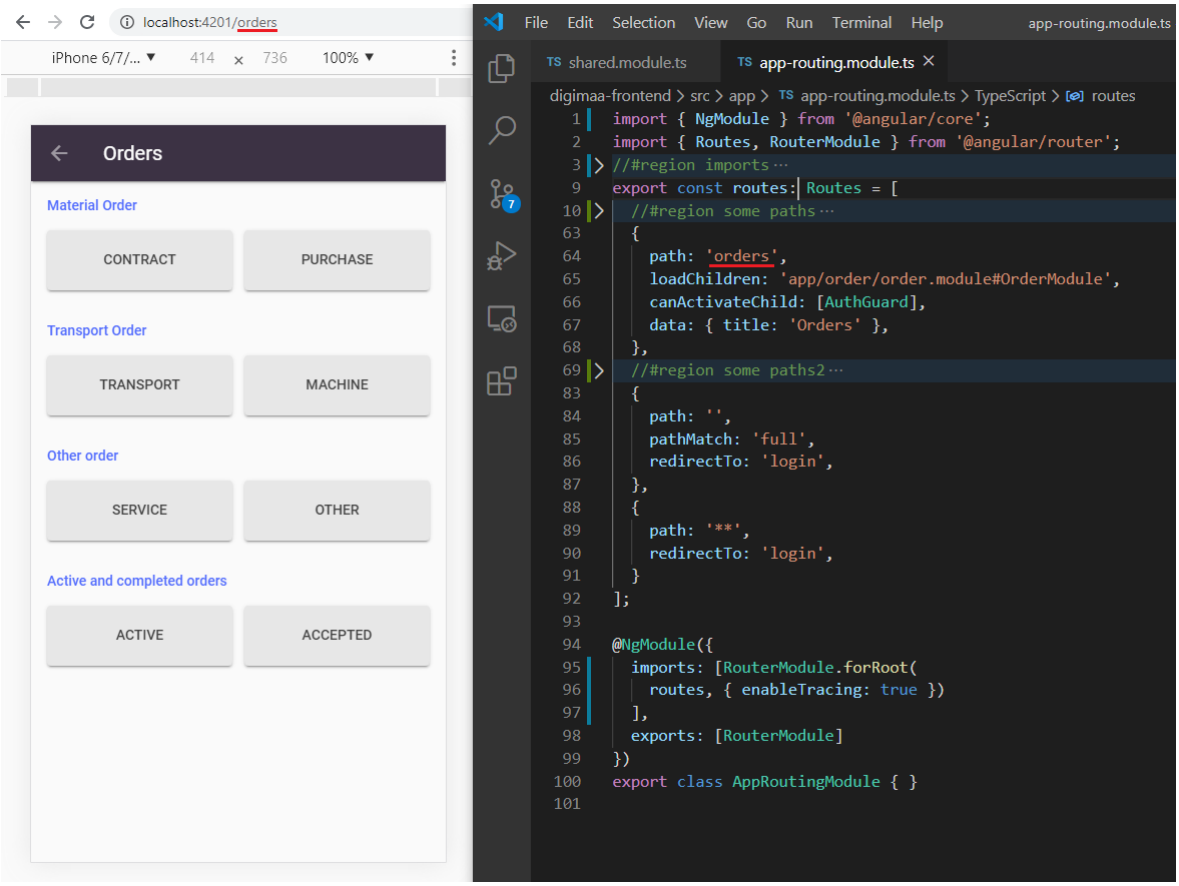
Kun käyttäjä navigoi sovelluksessa, sivuston URL voi muuttua, mutta Angular ei lähetä uusia kutsuja palvelimelle, vaan käyttää oletuksena historia-rajapintaa. Vaikka polku olisi uusi, sellainen missä käyttäjä ei ole käynyt aiemmin, on mahdollista siirtää käyttäjä sivulle ilman uusia pyyntöjä palvelimelle. Tämä on mahdollista, koska koko sovellus latautuu käyttäjälle, jolloin käymättömät näkymätkin ovat tallentuneet jo historian rajapintaan.

Moduulit ovat vastuussa omasta reitityksestä, mutta yleensä reitit toteutetaan erilliseen moduuliin, mikä lisätään kyseiseen moduuliin. Reitityksiin yleensä lisätään juuripolku, alipolut ja sen jälkeen reitityksen kattamattomat reitit poissuljetaan.

Reitityksessä yhdistellään kaikki polut johonkin toteutetuista komponenteista tai ominaisuusmoduuleista. Mikäli reitti ohjataan komponenttiin, niin kyseessä olevan komponentin sisältö renderöidään käyttöliittymään. Moduulin reitittäessä vastuu reitityksestä siirtyy reititetylle moduulille.

Reittien poissulkemisella estetään sovelluksen rikkoutumista. Se toteutetaan joko niin, että käyttäjä uudelleenohjataan toimivaaan osioon tai ei toivottuihin tapauksiin sidotaan siihen suunniteltu virheilmoituskomponentti. Kuvassa 9 on esitelty vasemmalla sovelluksen näkymää orders-reitissä ja oikealla on esitetty Angularin reitittimeen liitettävää reitti

objektia. Reitin kuvailussa määritellään URL-polku ja moduuli tai komponentti, mikä reitissä näytetään.



Kuva 9. Angularin reititys

Reitittämiseen Angularissa voidaan sitoa myös reittiparametrejä ja kyselyparametrejä. Reittiparametrejä voidaan käyttää silloin, kun haetaan jotain tiettyä sisältöä. Kyselyparametrejä käytetään, kun haetaan polkuun liittyen tarkentavia tietoja.

Mikäli on tarpeen estää pääsy käyttäjältä johonkin tiettyyn reittiin, voidaan sille asettaa vartijoita (Guard). Vartijaan voidaan liittää toiminnallisuutta, esimerkiksi se voi tarkastaa polkuun pyrkivän roolin tai vaikka käyttöluvat. Vartijat voivat päästää ehtojen täyttyessä käyttäjän reitille tai siirtää käyttäjän toiseen reittiin, mikäli käyttäjää ei haluta päästää sen hakemalle reitille.

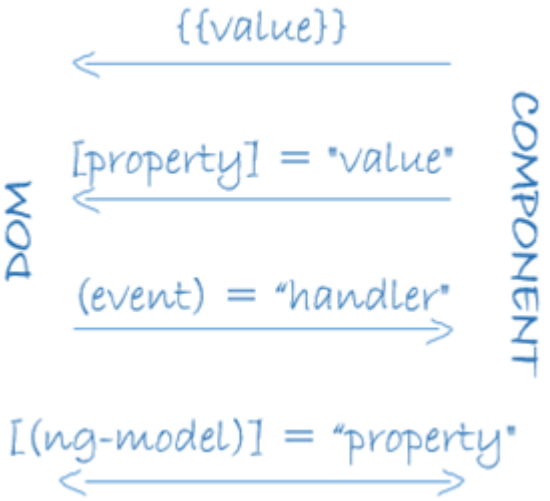
3.6 Komponentit

Komponentit ovat Angularin arkkitehtuurin näkyvin osa. Ne toimivat käyttöliittymien rakennuspalikoina ja vastaavat käyttöliittymän rakenteista, tyyleistä ja toiminnallisuuksista. Angular-komponentteja voidaan luokitella alaluokkiin, kuten säiliö-, esittelijä- ja sekakomponentit (Lars Nielsen, 2018).

Komponentit rakentuvat templatesta (HTML), Tyyliosiosta (CSS) ja toiminnallisuudesta (TS). Komponentin osat jaetaan yleensä omiin tiedostoihin, mutta komponentin toteuttaminen on mahdollista myös yhdellä tiedostolla.

3.6.1 Data-binding

Komponenttien yleisin ominaisuus, mitä voidaan käyttää myös natiivien HTML-elementtien kanssa Angularissa, on datan bindaus eli kytkentä tai liitos käyttöliittymään. Komponentin logiikkaa ja käyttöliittymää voidaan yhdistää neljällä eri tavalla, kuten kuvassa 10. Angular synkronoi näiden keinojen avulla tietoa komponenttien välillä.



Kuva 10. Angularin datan liittäminen (Medium 2016)

Datan interpolointi on yleisintä, eli datan arvo näytetään suoraan elementin sisällä käyttäen kahta aaltosulkua. Tätä metodia käytetään siis HTML-tägien välissä, jolloin elementti käyttäytyy niin kuin sen sisälle olisi kirjoitettu tekstiä. Angularin ajoympäristö huolehtii, että tieto päivittyy, mikäli muuttujan arvo muuttuu logiikkapuolella.

Joissain tapauksissa dataa kuitenkaan ei voida interpoloida suoraan HTML:n avulla, vaan data täytyy esitellä HTML-elementille propertynä, eli ominaisuutena. Esimerkiksi kuvien esittämisessä html:ssä käytettävä img-elementti vaatii sen src ominaisuuden käyttöä. Angularissa src:n ympärille asetetaan hakasulkeet, jonka jälkeen ominaisuuden arvo voidaan esittää elementille ohjelmalogiikan puolelta.

Käyttäjän toimintoja voidaan Angularissa seurata event-bindingin avulla komponenteissa. Esimerkiksi nappulaelementin painallukseen voidaan liittää event-binding kahden kaarisulun avulla, mikä vastaa komponentin eventtiä. Arvoksi syötetään haluttu funktio, jolloin se ajetaan aina, kun eventti kuplii ylöspäin. Yleisten HTML-elementtien

eventtiominaisuudet ovat oletuksena käytössä Angularissa ja omien eventtien toteuttaminen on myös mahdollista komponenttitasolla Output-metodin avulla.

Angularissa on myös askeleen pidemmälle viety toiminnallisuus nimeltä two-way data-binding, mikä vaatii tietynlaista HTML-elementtiä tai oikein toteutettua komponenttia toimiakseen. Sen etuna on, että kaksisuuntainen dataliikenne voidaan toteuttaa yhdellä lauseella. Kaksisuuntaisuudella voidaan hallita yhden muuttujan tilaa siis joko komponentin sisältä tai sen ulkopuolelta. Yleisin käyttökohde two-way data-bindingilla on yksittäinen syötekenttä, missä kentän arvoa halutaan muokata elementin sisä- ja ulkopuolelta.

Kuvassa 11 on esitelty Angularin neljä tapaa liittää dataa ohjelmalogiikan ja HTML:n välillä saman komponentin sisällä. Ohjelmalogiikan myVariable-muuttuja ja myHandler-metodi on liitetty HTML:ään.

Kuva 11. Angularin datan liitostapoja käyttöliittymän ja ohjelmalogiikan välillä

3.6.2 Esittelijäkomponentti

Esittelijäkomponentilla on jokin yksinkertaistettu tarkoitus. Sen sisäinen logiikka on vähäistä ja sen tilaa pystytään muokkaaman yleensä suoraan käyttöliittymästä. Tällaisen komponentin testaaminen on myös usein suoraviivaisempaa, koska sen toiminnallisuus on yksiselitteinen.

Komponenttia voidaan myös tarvittaessa käyttää useassa paikassa. Jotta komponenttia olisi järkevää käyttää uudelleen, sen kannattaa omata sovellukselle ominaisia piirteitä ja olla mahdollisimman geneerinen. Osa uudelleen käytettävyyden ideaa on myös se, että se vähentää toistuvaa lähdekoodia.

3.6.3 Säiliökomponentti

Säiliökomponentti pitää yleensä sisällään muita komponentteja ja sillä hallitaan isompaa kokonaisuutta, kuten yksi käyttöliittymän näkymä. Sen sisällä voidaan myös toteuttaa monimutkaista logiikkaa.

Komponenttia ei yleensä käytetä uudelleen muissa tilanteissa, koska sen monimutkaista logiikkaa on usein vaikeaa sopeuttaa muihin tilanteisiin. Näissä komponenteissa hallitaan myös usein kommunikointia muiden sovellusosien kanssa.

3.6.4 Sekakomponentti

Sekakomponenteissa on sekoitettu sekä esittelijäkomponentin, että säiliökomponentin piirteitä. Testausvaihessa syntyy usein tämänkaltaisia komponentteja, mutta näistä usein pyritään eroon, koska näiden komponenttien hallinta ja ylläpito on yleensä monimutkaista.

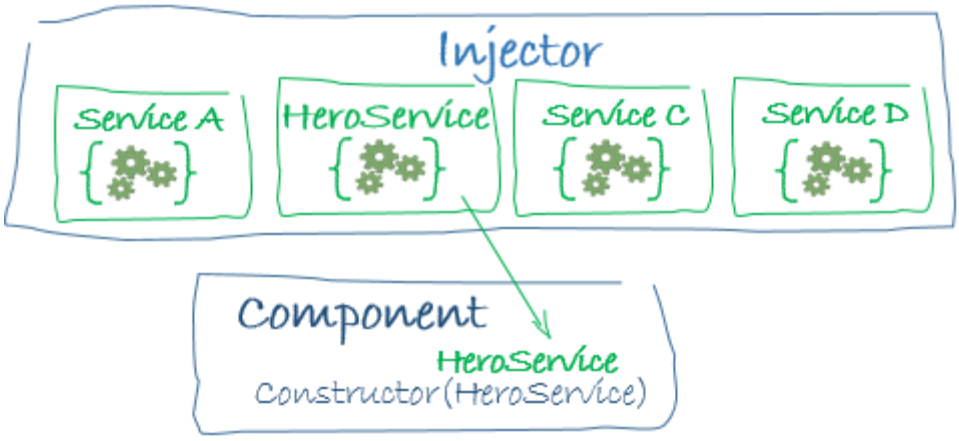
Sekakomponentteja luodaan myös välillä sellaisiin monimutkaisiin komponentteihin, mitkä palvelevat isompaa kokonaisuutta ja pitävät siitä syystä pienempiä komponentteja sisällään. Esimerkiksi listoissa saattaa olla päätason sekakomponentti, mikä pitää sisällään useamman tyylisen osiokomponentin listan muodostamiseksi. Päätason sekakomponentilla saadaan näin yhtenäistettyä kokonaisuutta ja rajattua käytössä olevia osia hallitusti.

3.7 Palvelut

Palveluilla Angularissa tarkoitetaan luokkaa, mistä on luotu yksi instanssi ohjelman alkaessa tai siinä vaiheessa, kun sitä ensimmäisen kerran käytetään. Palveluita voidaan yleensä kutsua muista palveluista tai komponenteista (Google, 2020e).

Palvelut pitävät yllä tilaa, minkä ne voivat jakaa ulospäin käytettäväksi tai muokattavaksi. Jos joku komponentti tai palvelu muokkaa tilaa, muuttuu se kaikissa tilaa seuraavissa komponenteissa ja palveluissa. Tämä johtuu Singleton periaatteesta, mikä sallii yhden instanssin ja on tässä tapauksessa tarkoituksenmukaista.

Angularissa kirjastojen, eli riippuvuuksien lisääminen tapahtuu myös palveluiden kautta. Riippuvuuksien lisäämisessä käytetään riippuvuusinjektiota, eli Dependency Injection (DI) ohjelmistoarkkitehtuurin mallia. Tällä keinolla pyritään erottelemaan sovellusrakenteita objekteista, mikä edesauttaa sovelluksien modulaarisuutta (Google, 2020b). Kuvassa 12 palvelu injektoidaan komponentin muodostimen kautta, jolloin komponentti pääsee käsiksi HeroService:n kautta lisättyihin kirjastoihin.



Kuva 12. Angular palvelun injektointi komponenttiin (Google 2020b)

3.8 Direktiivit

Direktiivien tarkoitus on ohjata dokumenttioliomallia (DOM). Angularin osalta ne on jaettu kolmeen eri kategoriaan niiden käyttötavan perusteella (komponentti-, rakenne- ja attribuuttidirektiivit). Direktiiveillä on Angularin arkkitehtuurissa ainakin osittainen vastuu DOM-puun muokkaamisesta ja siksi suoria muokkauksia koodista DOM-puuhun ei suositella.

Yleisin näistä kolmesta on komponenttidirektiivi (component directive), joka on sama asia kuin komponentti kappaleessa esiteltävä. Komponenttia ei yleensä mielletä direktiiviksi, mutta teknisesti ottaen se on sitä, koska se määrittää DOM-puun rakennetta lisäämällä sinne HTML-elementtejä.

Rakennedirektiiveillä (structural directives) muokataan suoraan DOM-puuta poistamalla tai lisäämällä sinne uusia HTML-elementtejä. Angularissa on valmiiksi luotuna muutamia, kuten NgIf, NgSwitch ja NgForOf. Angular mahdollistaa myös omien direktiivien luonnin, joihin varteenotettavana esimerkkinä voitaisiin mainita NgForIn, mitä ei ole Angulariin valmiiksi toteutettu (Josep Sayol, 2016).

Attribuuttidirektiivit (attribute directives) käsittävät DOM-puun muokkauksia, mutta se ei lisää tai poista HTML-elementtejä. Tällä direktiivillä muokataan olemassa olevien HTML-elementtien arvoja. Angularissa voi luoda näitä attribuuttidirektiivejä itse tai käyttää olemassa olevia kuten NgStyle ja NgClass.

Direktiivinä attribuuttidirektiivi on hieman suorituskykyisempi kuin rakennedirektiivit, sillä tällä ohitetaan uusien HTML-elementtien alustaminen ja vanhojen poistaminen. On suorituskyvyn kannalta nopeampaa, jos elementin voi piilottaa ja tuoda esille, kuin että poistaisi sen ja alustaisi uuden.

Direktiivejä, kuten muitakin Angularin osia voidaan käyttää useamman kerran. Mikäli direktiiviä halutaan käyttää toisissakin moduuleissa, pitää direktiivi ilmoittaa moduulin exports listassa ja sen jälkeen direktiivin moduuli on tuotava haluttuun moduuliin.

3.9 Putket

Angularissa on mahdollista myös muokata dataa putkien avulla. Angularin dokumentaatiosta käy ilmi, että putkia suositellaan käyttämään datan visualisoinnissa silloin, kun ei haluta muokata alkuperäistä dataa, mutta se halutaan visualisoida eri muotoisena (Google, 2020f). Putket mahdollistavat datan käsittelyn käyttöliittymälle sopivaksi muokkaamatta itse muuttujan arvoa.

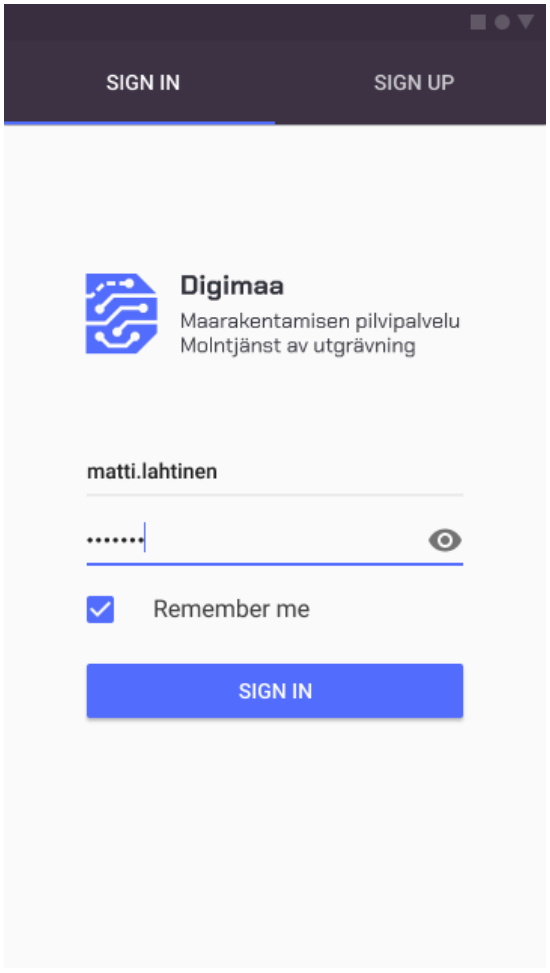
Putkien toteuttamisessa myös yksinkertaisuus on tärkeää. Mikäli monimutkaista putkistoa tarvitaan, voi putkia yhdistellä kuten UNIX-maailmassa. Putkien avulla voidaan muokata esimerkiksi päivämäärät ja kellonajat näkymään käyttäjälle tietyssä muodossa.

4 MODULAARISEN SOVELLUKSEN SUUNNITTELU

Sovelluksen käyttöliittymää suunniteltiin Adoben XD:llä. Sillä pystyttiin toteuttamaan koe versioita käyttöliittymistä, missä sovelluksen toiminnallisuuden idea pystyttiin tuomaan esille nopeammin. Käyttöliittymä muotoiltiin mobile first -tyylillä, jolloin sovelluksen responsiivisuus toimisi paremmin myös isoilla näytöillä. Käyttöliittymän esittely alkaa kirjautumisesta.

4.1 Kirjautuminen ja rekisteröinti

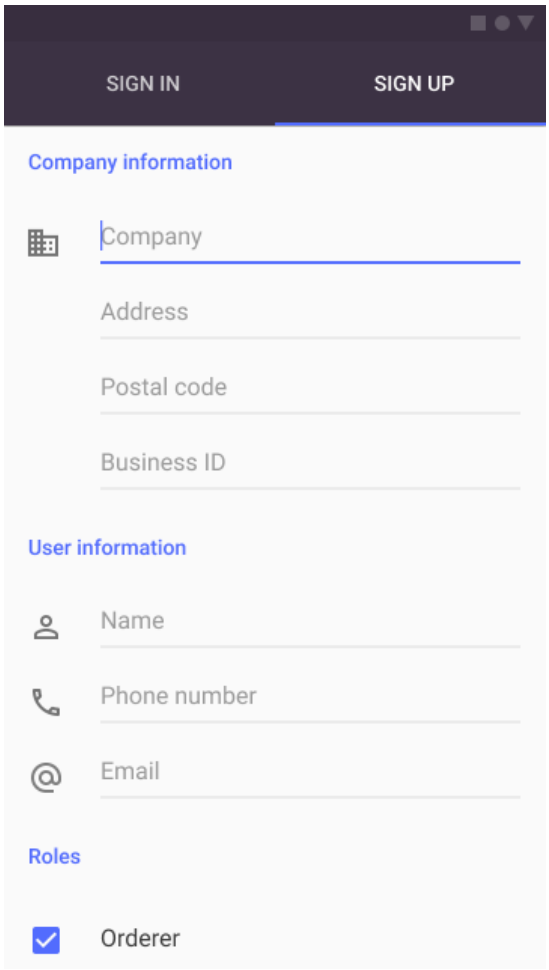
Käyttöliittymän alussa aukeaa käyttäjälle kuvan 13 mukainen kirjautumisruutu. Kirjautumisnäkymässä on jo suunniteltu ilme, mikä säilyy kautta sovelluksen. Rekisteröinti palveluun suunniteltiin toiseen välilehteen, mikä on vaihdettavissa yläoikeasta kulmasta.



Kuva 13. Kirjautuminen

Rekisteröinnin sivulla on muutama syötekenttä ja sinertävä alaotsikko. Sinertävää väriä käytettiin aksenttina, mikä erottuu hyvin muusta sovelluksesta. Käyttäjän huomiota pyrittiin aksentin avulla fokusoimaan sen hetkisiin tärkeisiin osiin. Syötekentät toteutettiin samalla

ulkoisella tyylillä koko sovelluksessa. Kuvassa 14 on esitelty käyttäjien rekisteröinti osion luonnos, missä on ylhäällä navigointimahdollisuus takaisin kirjautumisnäkymään ja alhaalla rekisteröinnin syötekentät.



Kuva 14. Rekisteröinti

4.2 Sivurakenne ja latautuminen

Käyttäjän kirjautuessa sovellukseen, näytetään käyttäjälle sovelluksen olevan latausti-lassa. Tämän jälkeen sovellus ottaa yhteyttä taustajärjestelmään autentikoinnin toteuttamiseksi. Kun taustajärjestelmä vastaa, voidaan viesti käsitellä ja laittaa latauksen tila pois päältä.

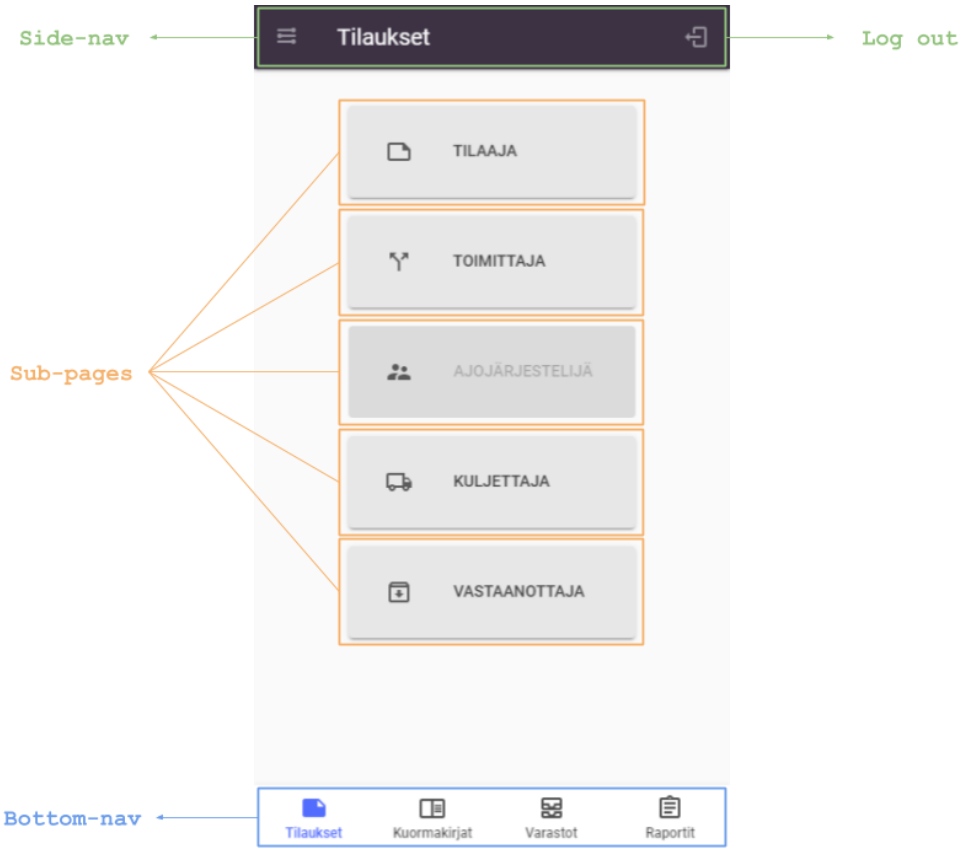
Kaikki sovelluksen näkymät käyttävät samanlaista tyyliä kuin kuvan 15 näkymä, missä sen yläosassa on tumma navigointi komponentti ja keskellä vaalea sisältö komponentti. Sovelluksen lataustila näkyy vihreän sisältöosioin keskellä keskitettynä, mikäli sovellus asettaa lataustilan syystä tai toisesta päälle. Sovelluksen kaikki toiminnallisuus on myös estetty tällöin, jotta käyttäjä ei voisi keskeyttää toimintaa. Sovelluksella on pyritty

tilanteeseen, jossa käyttäjää ei tarvitsisi keskeyttää, mutta oikeanlaisen toiminnallisuuden takaamiseksi käytetään tätä ominaisuutta tarvittaessa.

Kuva 15. Sivujen tyyli

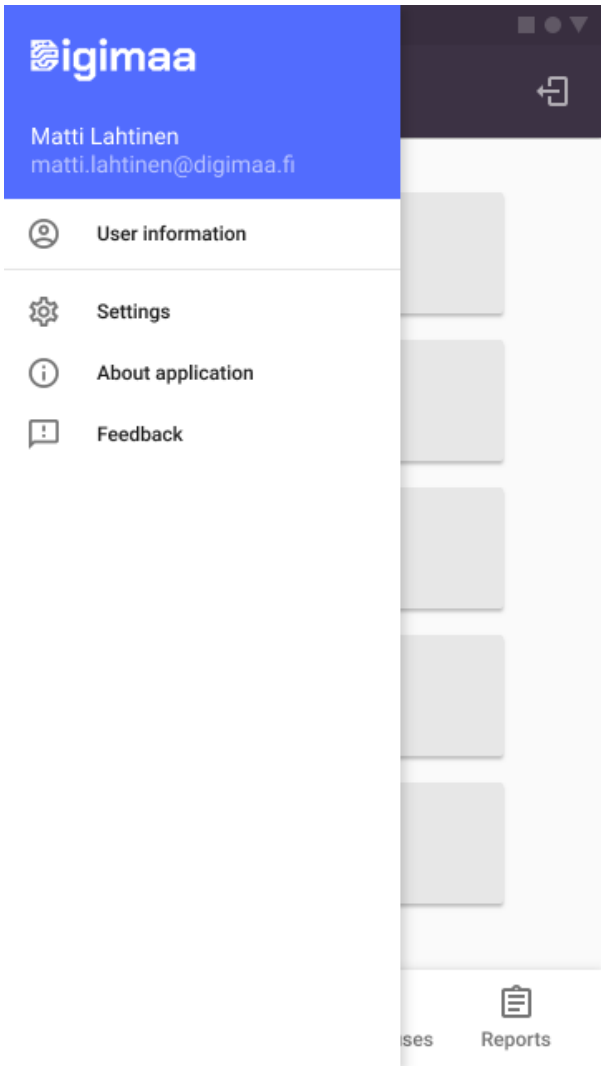
4.3 Navigointi

Navigointi suunniteltiin mobiililaitteita ajatellen. Se toteutettiin niin, että jokainen polku huomioitiin käyttöliittymässä ja palautuminen onnistuisi jokaiselta aukaistulta näkymältä takaisinpäin painikkeen avulla. Kuvassa 16 on esiteltynä sovelluksessa olevia navigointi mahdollisuuksia.



Kuva 16. Päänäkymän navigointi mahdollisuudet

Sivupalkkiin piilotettiin kuvan 17 mukaisesti lisää sovelluksen toimintoja, sekä kirjautuneen käyttäjän tietoja. Sivupalkin idea mobiilikäyttöliittymässä on mahdollistaa uusien toimintojen lisääminen, mikäli ne eivät mahdu päänäkymälle. Sovelluksen asetus- ja käyttäjätietosivu lisättiin sivupalkkiin navigointina. Palaute ja sovellusinformaatio toiminnot lisättiin myös sivupalkkiin, mutta niistä aukesi dialogi näkymän sijaan.

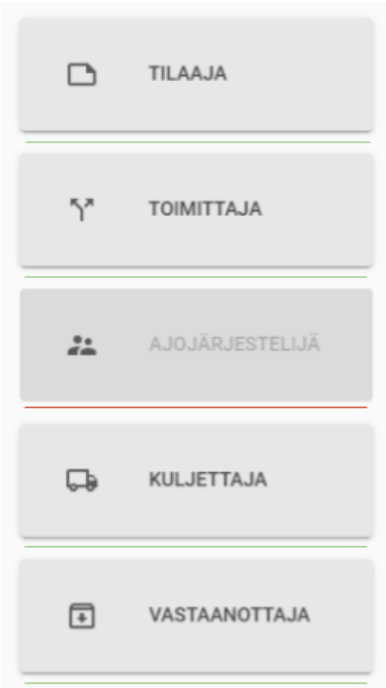


Kuva 17. Sivupalkki

4.4 Roolit

Sovelluksessa toimivat roolit olivat ratkaiseva osa sovelluksen toimintaa. Rooleilla mahdollistettiin sovellukseen useampi eri toimija tilausketjusta. Roolivalinta kuvassa 19 esittää yhden käyttäjän tilannetta. Roolit suunniteltiin niin, että yhdellä käyttäjällä voi olla useampi rooli. Käyttäjällä olevat roolit esitellään käyttöliittymässä käyttäjälle käytettävinä ja

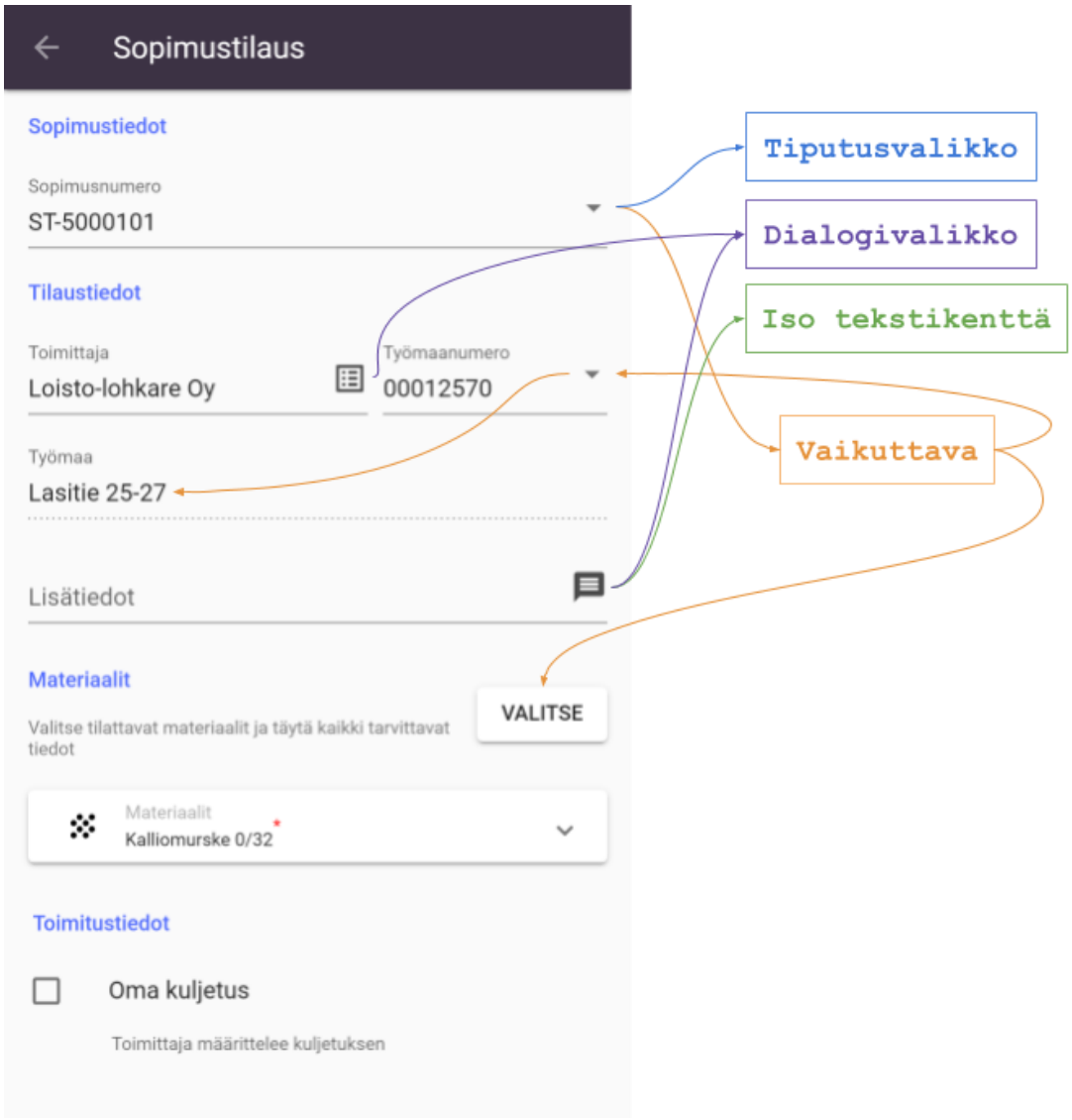
puuttuvat roolit estetään. Kehittäjien moduuli on piilotettu kokonaan käyttöliittymästä, mutta on olemassa sovelluksessa.



Kuva 18. Roolien esittely

4.5 Lomakesysteemi

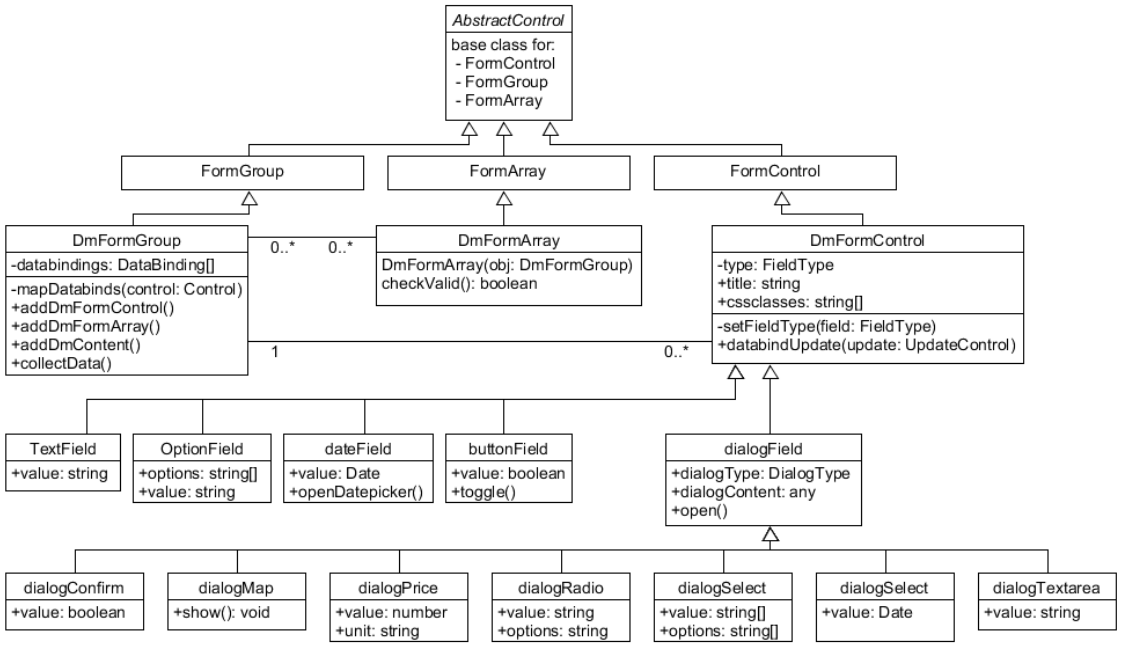
Sovelluksessa tärkeintä oli uusien tilausten luominen ja niiden käsittely. Lomakkeiden käsittelyyn suunniteltiin käytettäväksi melkein kaikki lomakkeisiin liittyvät ominaisuudet, kuten tekstikentät, numerokentät, monivalinnat, valintaruudut, tiputusvalikot, päivämäärän ja kellonajan valitsimet. Lisäksi lomakekenttien valintojen tahdottiin vaikuttavan muihin lomakekenttiin. Kuvassa 19 on esitelty uuden tilauksen luontiin tarkoitettua lomaketta, jonka toiminnallisuus on kuvattu kuvan oikeassa laidassa. Oranssissa vaikutusosiossa on kuvattu lomakkeen keinoa vaikuttaa lomakkeen valinnoilla myöhempiin vaiheisiin, mikä nopeuttaa tilausten tekemistä.



Kuva 19 Digimaa lomakesysteemi

Angularissa on valmiina kahdenlaista lomakesysteemiä. Niillä peruslomakkeiden teko on mahdollista, mutta näin laajan toiminnallisuuden toteuttaminen vaatii taipuvamman systeemin. Kuvassa 20 on kuvailtu pääpiirteittäin sovellukseen lisätyn oman lomakesysteemin luokkarakennetta.

Lomakesysteemin luokat ovat jatkeita Angularin ReactiveForms moduulissa olevista luokista. Näillä jatketuilla luokilla pystytään luomaan halutunlaisia syötekenttiä, sekä yhdistämään niiden tiedonsiirtoa lomakkeen sisäisesti. Näin pystyttiin toteuttamaan lomakkeita, jotka ovat vaatimusten mukaisia ja pystyvät tarvittaessa hyödyntämään muiden kenttien tietoja tehokkaammin.



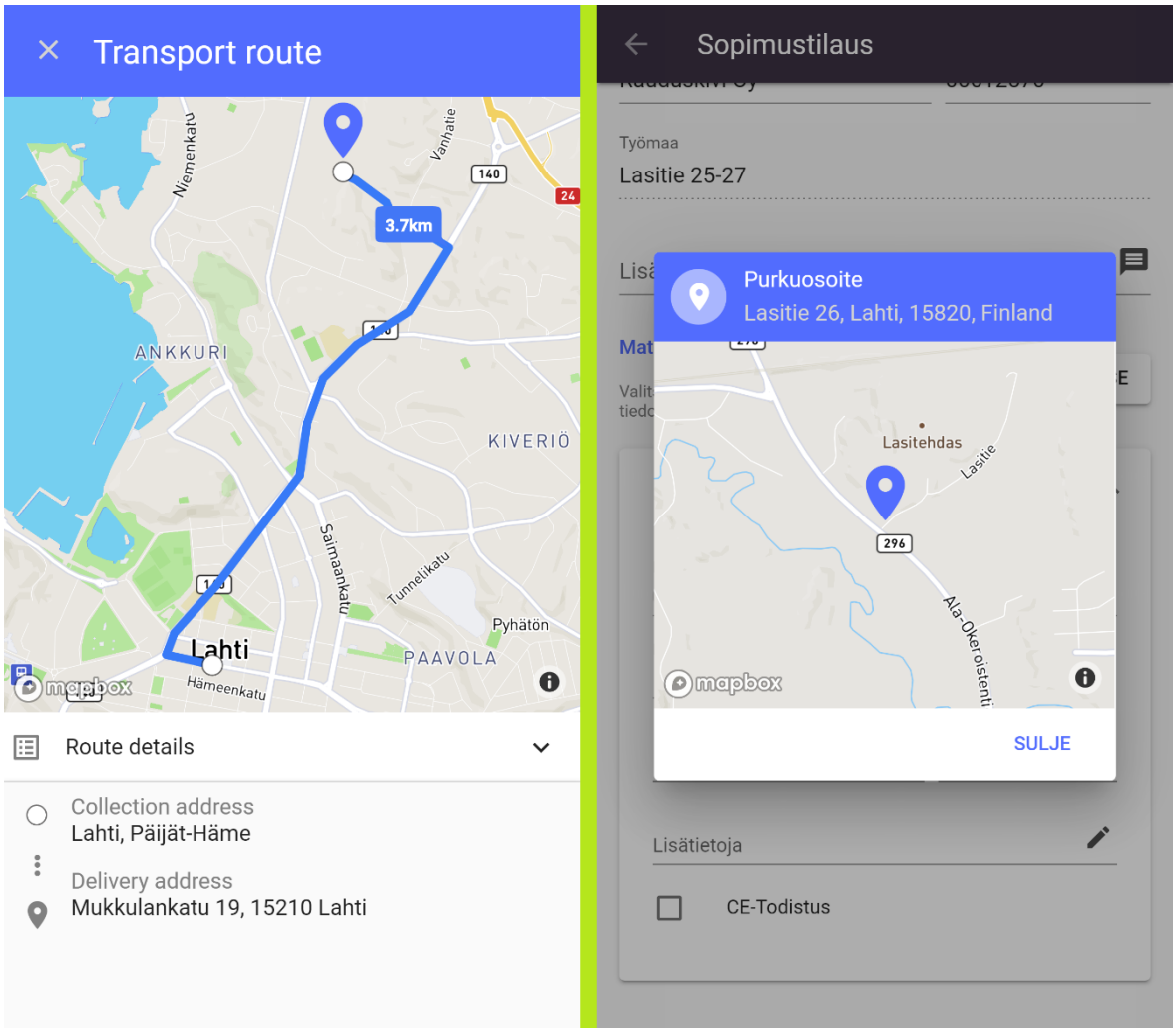
Kuva 20. Lomakesysteemin luokkakaavio

Kuvassa 20 olevat DmFormGroupit kuvastavat lomakeryhmiä, jotka koostuvat syötekentistä tai uusista sisäisistä lomakeryhmistä. Lomakejonot mahdollistivat uudet sisäiset lomakeryhmät. Näistä oli hyötyä esimerkiksi tilauksiin liittyvien tuotteiden tietojen listauksessa ja tietojen syöttämisessä niihin.

4.6 Karttapalvelu

Tässä sovelluksessa karttapalvelua ei alkuun nähty pakollisena, mutta myöhemmin kuljettajien roolien selvitessä oli melkein pakollista, että karttapalvelut liitetään jollakin tavalla sovellukseen. Kartta suunniteltiin toimivaksi dialogimaisesti. Sen tarkoitus ei ollut tarjota kokonaista navigointipalvelua, vaan sillä pyrittiin näyttämään maa-ainesten sijainteja.

Sovellukselle suunniteltiin myös kokonäytön kokoinen karttanäkymä. Sen idea oli piirtää reittiä ja näyttää matkaan liittyviä tietoja. Kuvassa 21 vasemmalla on kokoruudun reittiversio ja oikealla on dialogipohjainen pistesijainti.



Kuva 21. Digimaan karttapalvelut

Kun karttapalvelu lisättiin palveluun kuljettajalle, huomattiin myös, että sitä voitaisiin käyttää tilaajan ja toimittajan näkymissäkin helpottamaan osoitteiden hahmottamisessa. Maa-ainesten karttapohjaistakin hakua suunniteltiin, mutta mobiililaitteelle sellaisen toteutus olisi ollut haastavaa pienen ruutukoon takia.

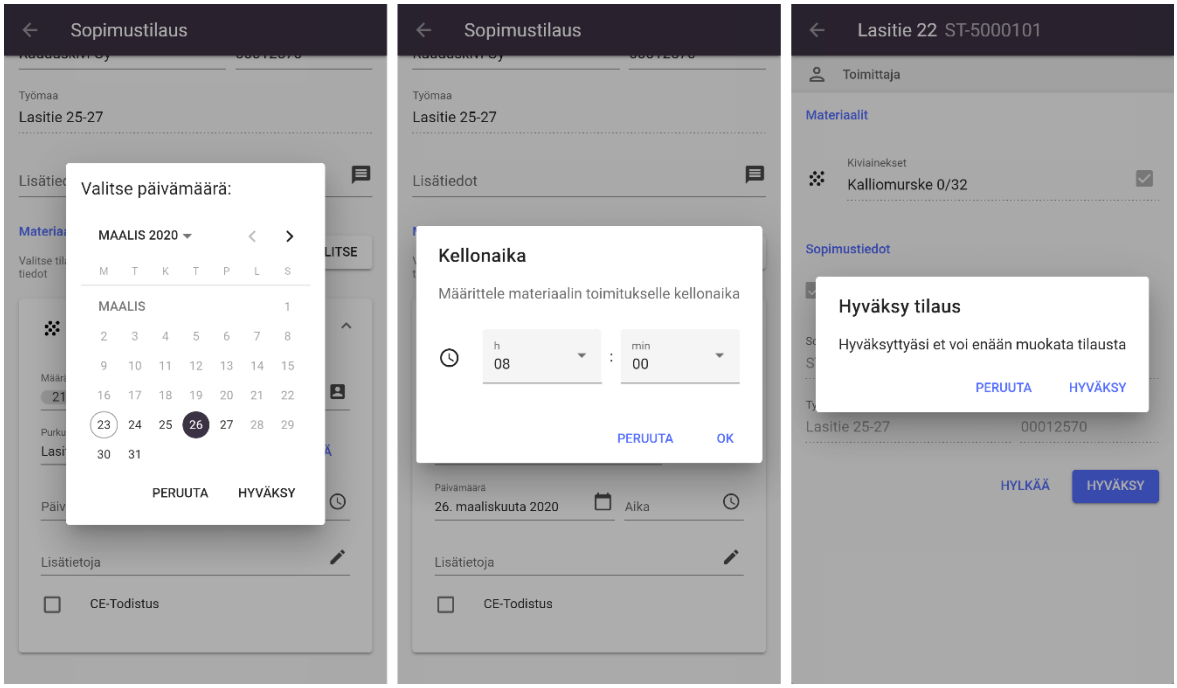
4.7 Kielisyys

Sovellus suunniteltiin alun perin kaksikieliseksi, myöhemmin sovelluksesta koitettiin tehdä myös espanjankielinen versio. Kielisyyden suunnittelussa ja sovelluksen toteutuksessa oli otettava huomioon käyttöliittymän muotoutumista käännösten kanssa. Monet suunnitellut komponentit saatiin toimimaan kummallakin kielellä, mutta sivujen muoto kärsi muutamista käännöksistä. Tämä johtui käännöksien eri merkkipituuksista. Kuvassa 21 on esitelty vasemmalla englanninkielistä versiota ja oikealla suomenkielistä.

4.8 Dialogit

Mobiililaitteiden pienen koon vuoksi on keksittävä keinoja hyödyntää ruutua paremmin. Sivustolle kun kertyy paljon kaikenlaista, on yhä vaikeampaa käyttää ahdasta sivustoa. Dialogien avaaminen sivulta antaa lisää tilaa työskennellä. Dialogit suunniteltiin sisällöllisesti kuitenkin lyhyeksi ja yksittäisiksi syötteiksi, jotta käyttäjien fokus ei häviä pääasiasta.

Alkuun sivuille suunniteltiin kalenteria, jonka jälkeen todettiin, että mobiililaitteilla kätevin olisi dialogimainen kalenteri, kuten kuvassa 22 vasemmalla. Samaa laajennettiin myöhemmin ajan valintaan. Näin pieneen noin 150 pikseliä leveälle ja 40 pikseliä korkealle alueelle pystyttiin toteuttamaan päivämäärän ja ajan valinta. Myöhemmin dialogiajatusta jatkettiin sovellusinfon näyttämisessä ja palautteen antamisessa. Myös aiemmin mainitut karttaominaisuudet toteutettiin dialogien avulla.



Kuva 22. Digimaa dialogeja

5 MODULAARISEN SOVELLUKSEN TOTEUTUS (DIGIMAA)

5.1 Tiedostorakenne

Tiedostorakenteessa on kiinnitettävä huomiota kasvaviin kansiorakenteisiin. Tällaisessa modulaarisessa sovelluksessa jaottelua tulee paljon ja sitä kautta kansiorakenteisiin syntyy paljon tiedostoja ja alikansioita. Tiedostojen ja kansioiden nimeämisen merkitys kasvaa. Kuvassa 23 on esitelty projektin loppuvaiheilta lähdekoodin määrä ja tiedostojakauma.

Project: digimaa-frontend

594 text files.
591 unique files.
212 files ignored.

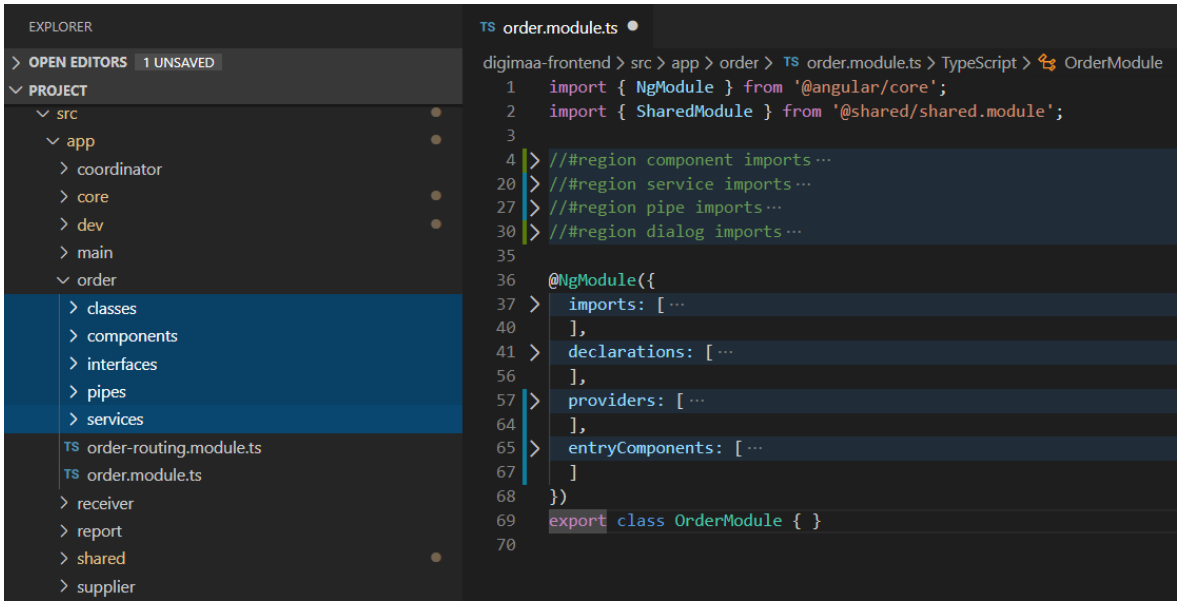
Language	files	blank	comment	code
TypeScript	241	1346	989	14240
Sass	73	690	899	3185
HTML	78	30	200	2101
JSON	7	0	0	503
JavaScript	2	4	4	51
Markdown	1	21	0	34
YAML	1	2	0	8
SUM	403	2093	2092	20122

Kuva 23. Digimaan koodi tilanne 16.09.2019

Jotta kansiorakenne olisi hallittavissa, on ymmärrettävä osin ihmisen hahmottamiskykyä. Ihminen muistaa keksimäärin 7 asiaa lyhyessä ajassa (Saul McLeod, 2009). Jotta kansioiden sisältö ei kävisi liian raskaaksi, koitetaan kansioiden rakenne pitää 7-8 tiedoston tai kansion kokoisena. Mikäli tiedostojen tai kansioiden määrä kasvaa paljon sen yli, koitetaan luoda tiedostoille alikansioita sitä mukaa.

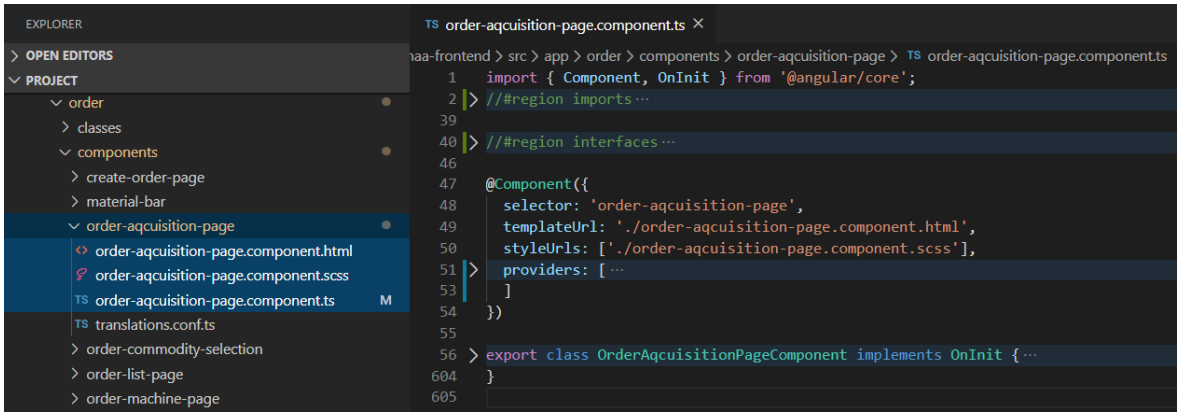
Kansioiden nimeämisessä käytetään Angularin termejä, kun mahdollista, jotta kuka tahansa rakennetta lukeva voi ymmärtää Angular dokumentaation perusteella, mitä kansiot

pitävät sisällään. Kuvan 24 vasemmassa laidassa on esiteltynä tilaajamoduulin kansiorakenne, joka nimeämisellään vastaa Angularin rakenteita.



Kuva 24. Tilaajamoduuli

Itse tiedostojen ja osien nimeämisessä pyritään kolmiosaisuuteen. Ensimmäisenä tiedoston nimessä mainitaan moduulin nimi mihin tiedosto kuuluu. Toinen osa nimeä kertoo, mitä tiedostolla tehdään ja kolmannella pyritään tarkentamaan tiedostotyyppiä, esimerkiksi määritellään sen olevan näkymä (page). Näin tiedostojen nimistä tulee yksilöiviä, muttei liian pitkiä. Nimeämisessä kiinnitetään huomiota myös aakkostukseen, jotta tiedostot olisivat järjestyksessä. Kuvassa 25 vasemmalla on havaittavissa tiedostojen kolmijakoinen nimeämiskäytäntö. Tiedostojen pisteiden jälkeiset osat voidaan lukea tiedoston muodoksi, eikä vaikuta tähän nimeämiskäytänteeseen.



Kuva 25. Komponentin nimeäminen

5.2 Ydinmoduuli

Ydinmoduulin rooli jäi pienehköksi tässä projektissa. Ydinmoduuliin toteutettiin kuitenkin kirjautuminen, koska se oli pakollinen sovelluksen käyttämiselle. Ydinmoduuli ladattiin myös vain kerran sovelluksen aikana AppModulesta, eikä sitä käytetty muualta. Ydinmoduuliin kuitenkin lisättiin jaetun moduulin ominaisuudet. Näin kirjautumisessa ja rekisteröinnissä pystyttiin käyttämään jaettuja sovelluksen ominaisuuksia.

5.3 Jaettu moduuli

Jaettuun moduuliin sijoitettiin pääosin komponentteja, luokkia, rajapintoja, direktiivejä, putkia ja palveluita. Näissä osissa yhteistä oli se, että niitä voitiin käyttää uudelleen kaikissa eri rooleissa, eli moduuleissa. Moduuliin lisättiin myös paljon Angularin materiaaliteeman moduuleita.

Jaetut osat toimivat sovelluksen modulaarisuuden tukipilarina. Näin mahdollistettiin sovelluksessa esimerkiksi lomakkeiden ominaisuudet, sivustojen mallipohjat, kartta ominaisuudet ja dialogit uudelleen hyödynnettäviksi.

Osia hyödynnettiin toistuvasti käyttöliittymän eri vaiheissa. Se loi sovellukselle yhdenmukaisen toimintamallin ja lisäsi osaltaan sovelluskehityksen hallittavuutta. Se vähensi myös riippuvuuksien merkitystä, mikä nähtiin hyvänä asiana. Mikäli jonkin ominaisuuden riippuvuutta olisi muokattu, olisi sen kaikki logiikka korjattu todennäköisesti yhden tiedoston sisällä ja ominaisuutta käyttäviä osia ei olisi ollut tarpeen muokata.

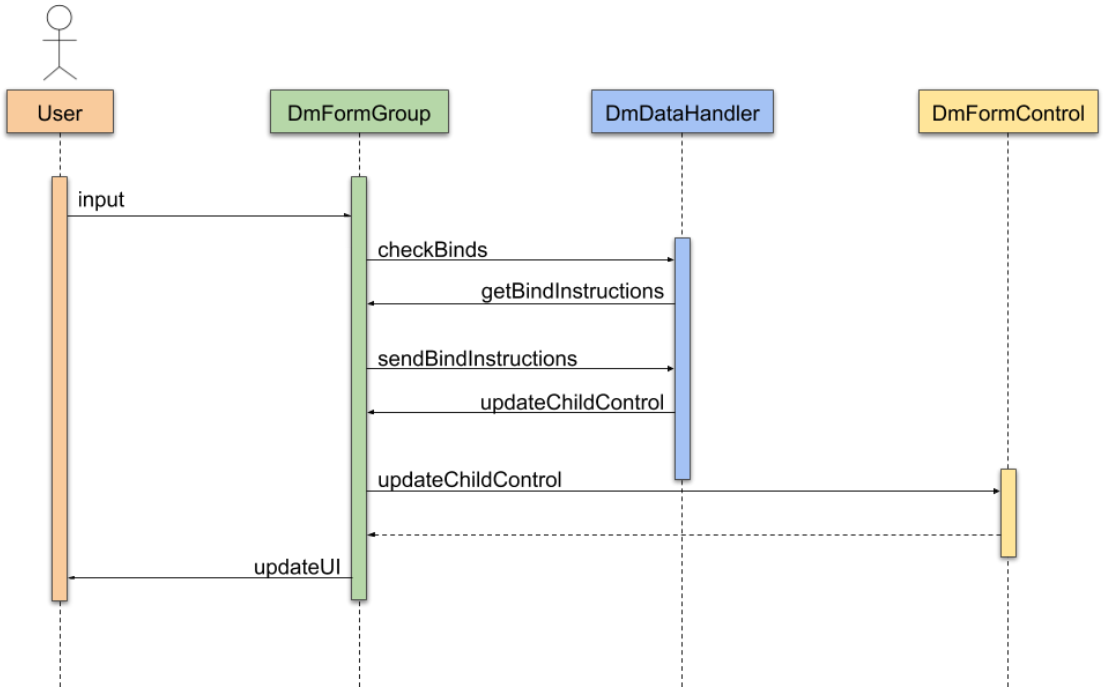
5.3.1 Lomakesysteemin toiminta

Oman lomakesysteemin toteutus lähti liikkeelle, kun huomattiin, että lomaketietojen hallinta lisäsi paljon logiikkaa näkymiin, mistä sitä ei enää voitu hyödyntää uudelleen. Myöhemmin lomakkeiden tietoja pyrittiin hallitsemaan palveluilla. Tämä kuitenkaan ei vaikuttanut riittävältä ratkaisulta, sillä lomakkeiden erilaisuus oli otettava huomioon jotenkin ja palveluilla ei pystytty ratkaisemaan lomakkeen ominaisuuksiin liittyviä seikkoja.

Olemassa oleva ReactiveForms nähtiin kuitenkin toimivana pohjana, joten lomakesysteemiä lähdettiin toteuttamaan sen pohjalta. ReactiveForms-luokkia jatkettiin dm-etuliitteellä, kuten aiemmin suunnittelu vaiheesta kävi ilmi. Laajennettuihin luokkiin pystyttiin lisäämään näin ominaisuuksia.

Seuraavaksi lähdettiin toteuttamaan syötekenttien välistä kommunikointia. Jotta syötekentillä voitaisiin vaikuttaa toisiin kenttiin, oli lomakeryhmään kehitettävä keino tarkkailla sellaisia syötekenttiä, jonka on tarkoitus vaikuttaa muihin kenttiin.

Kuvassa 26 on lomakesysteemin sekvenssikaavio, missä käyttäjä tekee lomakkeeseen syötteen ja se aktivoi lomakkeen sisäisen datanohjautumisen. Data siirtyy kaaviossa toiseen syötekenttään käsittelijän kautta, joka tarkastelee dataliitosta ja kutsuu lomakeryhmästä ohjeita datan hakuun.



Kuva 26. Lomakesysteemin sekvenssikaavio

Lomakkeiden syötekenttien väliseen kommunikointiin toteutettiin aluksi callback funktioiden avulla ohjeita siitä, mitä dataa lapsisyötekenttään syötettäisiin, jos vanhemman arvo muuttuisi. Lomakeryhmässä tarkkailtiin datakytköksillä varustettuja syötekenttiä, jonka arvon vaihtuessa lähdettiin dataliitoksia käsittelemään itse kehitetyllä dataHandler-metodilla. DataHandler-metodilla pystyttiin käsittelemään kaikki dataliitokset, hakemaan tarvittavat ohjeet ja ohjata lomakeryhmää niin, että syötekentät päivittyivät.

5.3.2 Angular Materiaalien hyödyntäminen

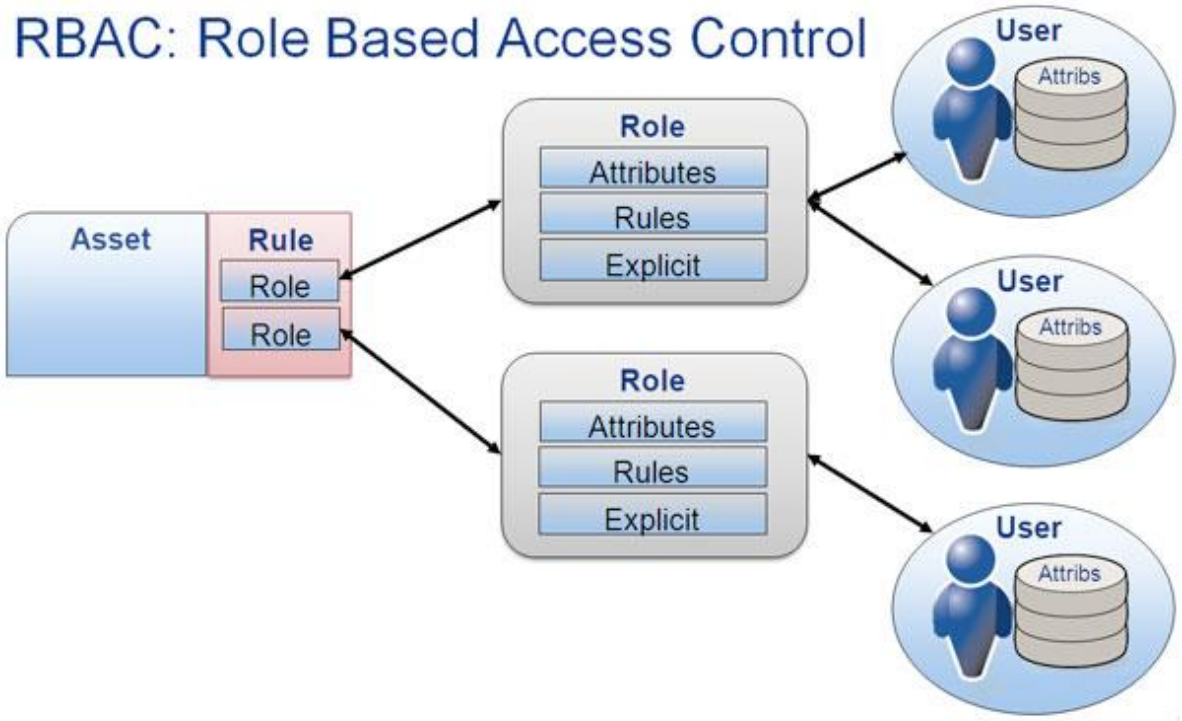
Aluksi ajateltiin käyttää vain muutamaa materiaaliteeman moduulia. Pikkuhiljaa kuitenkin ymmärrys materiaaliteeman moduuleista kasvoi ja käyttö lisääntyi sen seurauksena. Moduulien kautta saatiin nopeasti toteutettua sovellukseen ominaisuuksia, minkä kehittämiseen olisi voinut upota muuten paljon aikaa.

Materiaaliteemasta tuli kaiken kaikkiaan otettua 34 moduulia käyttöön. Käyttämällä paljon teeman moduuleita, tuli sovelluksen ulkoasustakin hyvin googlemainen.

5.4 Pääsynhallinta

Sovellukseen toteutettiin RBAC (Role-Based Access Control) tyyppinen ratkaisu, jolla sallittiin ja estettiin käyttäjiä sen oikeuksien perusteella. Koska moduulit yhdistettiin sovelluksessa rooleihin, oli tällöin myös toteutettava sovellukseen auktorointi moduuleille. Sovellukselle luotiin auktorointi palvelu, joka ylläpitää tilaa käyttäjän luvista ja rooleista.

Kun moduuleita alettiin rajaamaan, ajateltiin aluksi, että ne rajattaisiin suoraan roolien nimiin perustuen. Kuva 27 esittää tarkasti, miten roolit ja oikeudet toimivat sovelluksessa. Sovellus haki kirjautumisen aikana käyttäjän roolit, joista pystyttiin lukemaan roolien oikeudet.

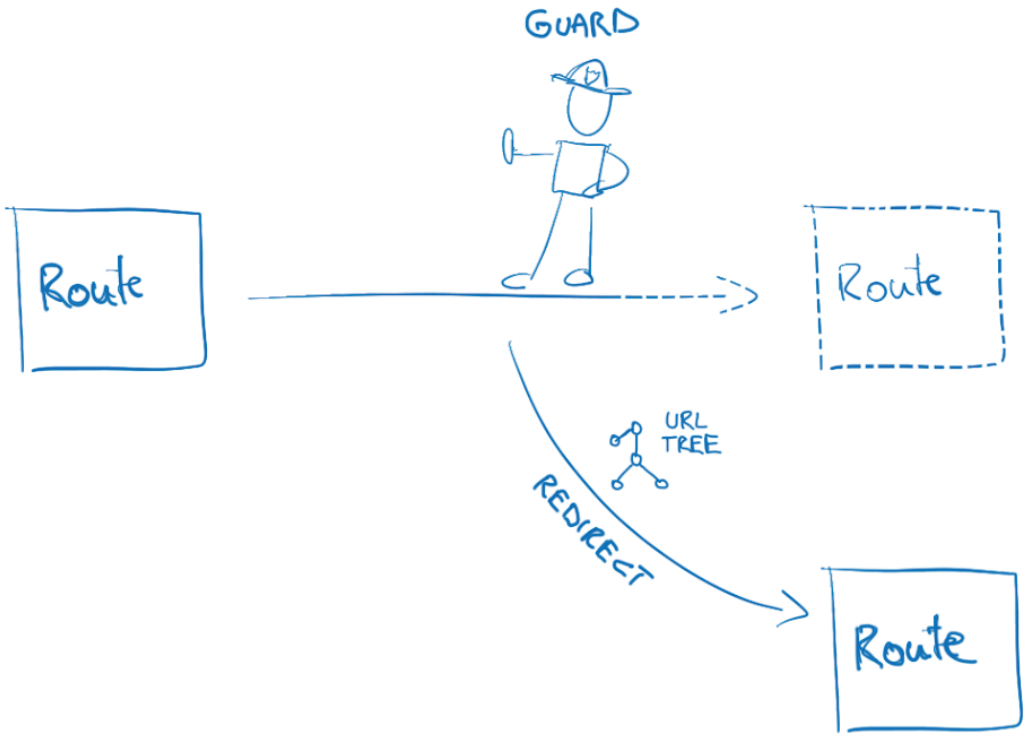


Kuva 27. RBAC yleisesti (Sander, 2009)

Myöhään projektin lopulla kuitenkin alettiin valmistelemaan ajatusta siitä, että mitä jos rooleille annetaankin pääsyoikeuksia taustajärjestelmän rajapintojen päätepisteisiin. Näin moduuleita voitaisiin sallia päätepisteiden perusteella, jolloin oikeuksien jakaminen olisi tarkemmin säädeltävissä.

Kun roolit ja oikeudet oli haettu ja jaettu autentikointipalvelun kautta, pystyttiin polkujen vartijassa (Kuva 28) määrittämään ketkä pääsivät moduuleihin käsiksi. Tämän lisäksi jokaisessa reitittimessä täytyi ilmoittaa, että tämä vartija oli käytössä. Vartijalle pystyttiin asettamaan parametreinä haluttu moduulinnimi, jolloin jokaiseen polkuun ei tarvinnut

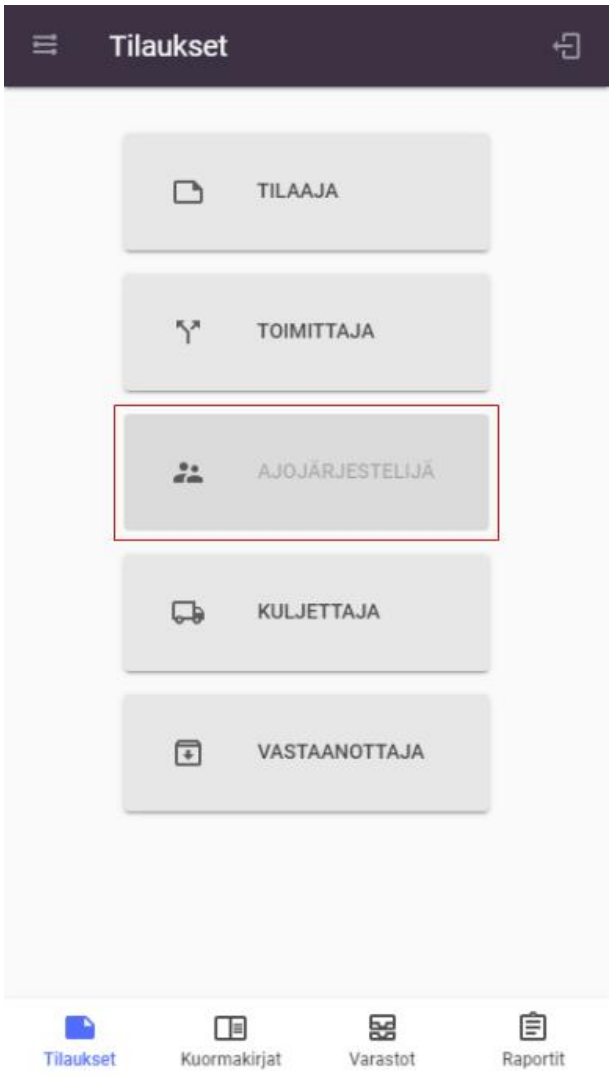
kehittää omaa vartijaa, vaan samalla vartijan kopiolla kyettiin hallitsemaan jokaisen eri moduuliin pääsyä.



Kuva 28. Reitinvartija luonnostelma (Strumpflohner, 2018)

Aiemmin mainitut asiat tapahtuvat käyttäjältä piilossa, eli näitä ei pystytä suoraan käyttöliittymässä havainnoimaan. Jotta käyttäjä voisi havainnoida nämä vartijat etukäteen, oli käyttöliittymään myös lisättävä ominaisuus, mikä visualisoi tai viestii sen käyttäjälle.

Sovelluksen pääsivulla käytettiin kahta eri keinoa visualisoinnissa. Kuvassa 27 voidaan havainnoida, kuinka käyttäjän painike ajojärjestelijä rooliin on harmaana. Tällä osoitettiin käyttäjälle, että sellainen rooli on olemassa, mutta pääsy häneltä on estetty. Toista keinoa ei kuvasta voi havainnoida, mutta esimerkiksi sovelluksen käyttöliittymään on lisätty kehittäjien oma rooli. Tämä piilotettiin sovelluksen käyttäjiltä, joilla kehittäjäroolia ei ole. Mikäli sovellukseen kirjauduttaisiin käyttäjällä, jolla se rooli olisi, niin kuvan 29 vastaanottajaroolin alapuolelle ilmestyisi ”Kehittäjä” niminen painike.



Kuva 29. Käyttäjälle asetettu rajoite

6 YHTEENVETO

Opinnäytetyön tavoitteena oli toteuttaa modulaarinen mobiilisovellus maanrakentamisen pilvipalvelua varten. Sovelluksella oli tarkoitus toteuttaa tilausketju, missä tilaukset tulee käsitellyksi. Toistaiseksi sovelluksella pystyy vasta tekemään tilauksen. Jatkovaiheet on sovelluksessa myös testattu, ja niillä pystytään sovelluksen osalta hoitamaan tilaukset loppuun asti, mutta oikea taustajärjestelmä ei ole vielä niiltä osin valmis, että sitä vasten voitaisiin sovellusta käyttää.

Sovelluksesta tuli hyvin modulaarinen. Itse kehitetyt esittelijäkomponentit toimivat applikaatiossa johdonmukaisesti ja niiden yksinkertaisesta liitettävyydestä ja monistettavuudesta oli hyötyä. Lähdekoodin koko rakenne muotoutui myös lukukelpoiseksi. Siinä on selkeä rakenne, mutta toki parantamisen varaakin jäi. Joidenkin tiedostojen nimeämiset eivät kaikki olleet yhdenmukaisia. Tämä johtuu osin ajattelutavan kehittymisestä kesken projektin ja osa tiedostoista oli helpompaa muistaa jo opituilla nimillä, joten niitä ei tullut vaihdettua.

Sovellukseen kerettiin kehittämään suhteellisen paljon ominaisuuksia ja sovellus koki myös versio loikan uudempaan kesken kehityksen. Sovelluksen kehittämisessä tuli myös haasteita muun muassa Angularin ajoympäristön kanssa. Ei ollut aina selvää, milloin Angular huomaa muutoksia. Ongelmaan törmättiin isojen objektien kohdalla, missä ilmeisimmin syvemmän tason muutoksia ei tarkkailtu optimoinnin takia.

Sovellus toimi mobiiliwrapperilla hienosti ja se mahdollisti päivitysten tekemisen ilman, että laitteilla olevaa sovellusta päivitettiin erikseen. Päivitykset latautuivat sovellukseen, kun web-sivustoa päivitettiin palvelimen puolella.

Käyttöliittymän kehittämisessä opittiin kuitenkin, että toiminnallisuuden toteuttaminen olisi tehokkaampaa, jos taustajärjestelmä olisi ollut jo valmiimpi tai että se etenisi rajapintojen osalta samaan tahtiin. Käyttöliittymässä jouduttiin varautumaan suuriin määriin rajapinta- ja logiikkamuutoksia, kun ei tiedetty etukäteen, miten taustajärjestelmä tulee toimimaan. Tämän varautumisen ansiosta kuitenkin käyttöliittymän osista tuli hyvin monikäyttöisiä ja todella modulaarisia, mutta itse työn tavoitteet kärsivät siinä hitauden takia.

Projekti on tällä hetkellä välietapissa ja odottaa jatkorahoitusta. Seuraavina kehityksen kohteina Digimaassa tulee olemaankin taustajärjestelmän testaaminen ja muiden ominaisuuksien kehittäminen, kuten muut tilaustyypit, tilausten raportointi ja analysoinnit. Näillä ominaisuuksilla täydennettäisiin kokonaisuutta tehokkaammassa kierrätettyjen maa-ainesten digitaalisessa käsittelyssä.

LÄHTEET

Aalto-yliopisto 2019. Digimaa [viitattu 18.04.2020]. Aalto-yliopisto. Saatavissa: https://www.aalto.fi/en/research-art/digimaa

Claire Patenaude, 2019. Website Design: Making the Case for Modular Design [viitattu 05.04.2020]. Celerity. Saatavissa: https://www.celerity.com/the-case-for-modular-website-design

Google 2019. API List [viitattu 18.04.2020]. Google. Saatavissa: https://v7.angular.io/api

Google 2020a. Angular [viitattu 18.04.2020]. Google. Saatavissa: https://angular.io/

Google 2020b. Dependency injection in Angular [viitattu 06.04.2020]. Google. Saatavissa: https://angular.io/guide/dependency-injection

Google 2020c. Introduction to Angular concepts [viitattu 05.04.2020]. Google. Saatavissa: https://angular.io/guide/architecture

Google 2020d. Introduction to modules [viitattu 18.04.2020]. Google. Saatavissa: https://angular.io/guide/architecture-modules

Google 2020e. Introduction to services and dependency injection [viitattu 06.04.2020]. Google. Saatavissa: https://angular.io/guide/architecture-services

Google 2020f. Pipes [viitattu 18.04.2020]. Google. Saatavissa: https://angular.io/guide/pipes

Jonathan Sander, 2009. RBAC and ABAC and Roles, oh my [viitattu 06.04.2020]. Identity Sander. Saatavissa: https://identitysander.wordpress.com/2009/11/03/rbac-and-abac-and-roles-oh-my/

Josep Sayol, 2016. Extending *ngFor in Angular to support “for…in” [viitattu 28.02.2020]. Medium. Saatavissa: https://medium.com/@jsayol/having-fun-with-angular-extending-ngfor-to-support-for-in-f30c724967ed

Juri Strumpflohner, 2018. Better Redirects in Angular Route Guards [viitattu 06.04.2020]. Juristr. Saatavissa: https://juristr.com/blog/2018/11/better-route-guard-redirects/

Kalvi Group, 2020. Web Development Trends 2020 [viitattu 18.04.2020]. Medium. Saatavissa: https://medium.com/@seo.kalvigroup/web-development-trends-2020-9175e6aa9c39

Lars Nielsen, 2018. Model-View-Presenter with Angular [viitattu 28.02.2020]. Medium. Saatavissa: https://medium.com/angular-in-depth/model-view-presenter-with-angular-3a4dbffe49bb

Microsoft Corporation 2019. TypeScript – JavaScript that scales [viitattu 21.11.2019]. Microsoft. Saatavissa: https://www.typescriptlang.org/

Rik de Vos, 2019. 5 Tips & Best Practices to Organize your Angular Project [viitattu 21.11.2019]. Medium. Saatavissa: https://medium.com/dev-jam/5-tips-best-practices-to-organize-your-angular-project-e900db08702e

Saul McLeod, 2009. Short Term Memory [viitattu 23.03.2020]. SimplyPsychology. Saatavissa: https://www.simplypsychology.org/short-term-memory.html

Todd Motto, 2017. A deep dive on Angular decorators [viitattu 23.03.2020]. Ultimate Courses. Saatavissa: https://ultimatecourses.com/blog/angular-decorators#angular-decorators

Tutorials Point 2020. Component-based architecture [viitattu 26.02.2020]. Tutorials Point. Saatavissa: https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm

William Miller, 2019. Testing automation is easier with modular programming [viitattu 04.04.2020]. iBeta. Saatavissa: https://www.ibeta.com/testing-automation-is-easier-with-modular-programming/

Yakov, F. & Anton, M. 2018. Angular Development with TypeScript Second Edition. Stamford: Manning Publications Co.