Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Sami Varis

Ohjelmistokehityksen nopeuttaminen Angular-ohjelmistokehyksen avulla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

6.11.2019

Tekijä Otsikko Sivumäärä Aika	Sami Varis Ohjelmistokehityksen nopeuttaminen Angular-ohjelmistokehyksen avulla 48 sivua 6.11.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Mediatekniikka
Ohjaaja	Yliopettaja Kari Aaltonen

Opinnäytetyössä tutkittiin ohjelmistokehityksen nopeuttamista ja sitä, miten Angular-ohjelmistokehys voi nopeuttaa web-ympäristössä tapahtuvaa ohjelmistokehitystä.

Ohjelmistokehityksen nopeuttaminen on usein kovasti tavoiteltua. Työssä tutkittiin ohjelmistokehityksen nopeuttamisen eri tapoja ja sitä, millä tavoilla ohjelmistokehityksen nopeutta voidaan mitata. Nopeuttamisesta saatuja hyötyjä tutkittiin myös yritystasolla, ja tutkittiin, millaisia hyötyjä yritykset voivat saada lisäämällä ohjelmistokehityksen nopeutta. Samalla tutkittiin Angular-ohjelmistokehystä ja sen toiminnallisuuksia. Tutkimus pohjautui verkkolähteisiin ja Angular-ohjelmistokehyksen dokumentaatioon.

Työssä huomattiin, että tapoja ohjelmistokehittämisen nopeuden mittaamiselle ja tavoille on useita, niin henkilöstön kuin myös teknisen puolen osalta. Oikein tehtynä yrityksen kannalta ohjelmistokehittämisen nopeuttamisella on positiivisia vaikutuksia niin yrityksen tulokseen kuin myös henkilöstön tehokkuuteen ja hyvinvointiin.

Insinöörityössä luotiin Parkkipaikat-niminen ohjelmisto Angular-ohjelmistokehyksellä web-ympäristöön. Sen tarkoituksena on auttaa pysäköintipaikkojen löytämisestä Helsingissä.

Työssä tehtiin havaintoja, miten Angular-ohjelmistokehys voi nopeuttaa ohjelmistokehitystä web-ympäristössä. Parkkipaikat-ohjelmiston valmistuttua todettiin, että Angular-ohjelmistokehys oli nopeuttanut Parkkipaikat-ohjelmiston ohjelmistokehitystä huomattavasti.

Avainsanat	Angular, ohjelmistokehitys

Author	Sami Varis
Title	Accelerating software development by using Angular-framework
Number of Pages	48 pages
Date	6 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Media Technology
Instructor	Kari Aaltonen, Principal Lecturer

This thesis investigated acceleration of software development and how Angular-framework can accelerate software development in the web environment.

Speeding up software development is often highly sought after. The theory section explored different ways to speed up software development and how to measure software development speed. The benefits of acceleration were also studied at the enterprise level and what benefits companies can gain by increasing the speed of software development. At the same time, the Angular-framework and its functionalities were studied. The research was based on online materials and documentation of the Angular-framework. It was stated that there are many ways to do and measure software development acceleration. If done right, accelerating software development will have a positive outcome on the company's performance as wall as on the efficiency and well-being of its employees.

The practical part was to create a software called Parkkipaikat with the Angular-framework for a web environment designed to make it easier for people to find parking places in Helsinki.

Based on the practical part, observations were made on how the Angular software framework can accelerate software development in the web environment. After the completion of the Parkkipaikat-software, it was found that the Angular software framework had significantly accelerated the development of the Parkkipaikat-software.

Keywords	Angular, software development

Sisällys

Lyhenteet

SPA	Single Page Application. Yhden sivun ohjelmisto, joka lataa kaiken tiedon palvelimelta ensimmäisellä latauskerralla.
HTTP	Hypertext Transfer Protocol. Protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
CSS	Cascading Style Sheets. Tyylitiedosto, joka pitää sisällään ohjeita, kuinka WWW-dokumentti pitää esittää.
SCSS	Syntactically Awesome Style Sheets. CSS-esiprosessori, jonka avulla voidaan kirjottaa helpommin hallittavia tyylitiedostoja.
CLI	Command Line Interface. Komentorivikäyttöliittymä, jonka avulla voidaan antaa komentoja ohjelmalle.
NPM	Node Package Manager. Paketinhallintajärjestelmä, jonka avulla voidaan asentaa koodikirjastoja ohjelmistoon.
E2E	End To End. Testaustapa, jossa koko ohjelmiston toiminta testataan käyttöliittymästä aina tietokantaan saakka.
LTS	Long Term Support. Tuotteen pitkäaikainen tuki, jossa tietokoneohjelmiston vakaata versiota tuetaan pitkään esimerkiksi tietoturvapäivityksillä.
DOM	Document Object Model. Dokumenttioliomalli, joka kuvaa dokumentin rakennetta.

1 Johdanto

Web-ympäristöön on viime vuosina ilmestynyt todella paljon ohjelmistokehyksiä. Niiden tarkoitus on nopeuttaa ohjelmistojen kehittämistä ja tehdä niiden hallitsemisesta mahdollisimman helppoa. Ohjelmointikielet eivät itsessään tarjoa kehittäjille toiminnallisuuksia, jotka nopeuttaisivat kehitystyötä, joten tätä varten on kehitetty ohjelmistokehyksiä, jotka sisältävät kirjastoja ja valmiita toiminnallisuuksia, joita sovelluskehittäjät pystyvät päivittäisessä työssään hyödyntämään. Ohjelmistokehyksiä pidetään yleensä hyvänä, sillä ne mahdollistavat osaaville koodaajille nopean tavan kehittää sovelluksia ja nuorille kehittäjille oivan tavan päästä mukaan ohjelmistokehityksen maailmaan.

Angular on TypeScript-kielellä kirjoitettava ohjelmistokehys, joka on julkaistu vuonna 2010. TypeScript on JavaScript-kielen varaan rakennettu ohjelmointikieli, joka tuo JavaScriptiin paljon uusia ominaisuuksia. Merkittävin näistä ominaisuuksista on vahva tyypitys, josta TypeScript saa myös nimensä. Angular on selainpuolen sovelluskehys, jolla voidaan kehittää yksisivuisia web-sovelluksia. Se tarjoaa monia sovelluskehitystä nopeuttavia työkaluja, kuten reitityksen, lomakkeet ja Hypertext Transport Protocol -kutsut eli HTTP-kutsut.

Insinöörityön tarkoituksena on perehtyä erilaisiin tapoihin, joilla ohjelmistokehitystä voidaan nopeuttaa, ja siihen, mitä hyötyä ohjelmistokehityksen nopeuttamisesta on yrityksille. Työssä tutustutaan myös tarkemmin Angular-ohjelmistokehykseen ja sen käyttöönottoon. Samalla tutustun Angularin perustoiminnallisuuksiin ja sen tarjoamiin sovelluskehitystä nopeuttaviin työkaluihin. Tutkitaan myös, kuinka Angular-ohjelmistokehys nopeuttaa ohjelmistokehitystä ja mitä muita hyötyjä Angular-ohjelmistokehyksen käyttäminen ohjelmistokehitysprojekteissa tuo. Työssä kehitetään myös web-sovellus Angular-ohjelmistokehystä käyttämällä, ja samalla tutkitaan, vaikuttaako Angular-ohjelmistokehys positiivisesti ohjelmiston kehittämisen nopeuteen.

2 Ohjelmistokehitys ja sen nopeuttaminen

Ohjelmistokehitys eli ohjelmistotuotanto on prosessi, jossa kehitetään ohjelmisto, jonka avulla tietokone saadaan tekemään jotain haluttua (1). Ohjelmistokehitys on siis todella laaja käsite. Tässä opinnäytetyössä perehdyn ohjelmistokehityksen nopeuttamiseen ja siihen, mitä tavoitteita, mittaustapoja, nopeutustapoja ja hyötyjä ohjelmistokehityksen nopeuttamisella on. Työssä tutkin myös, miten web-ympäristössä tapahtuvaa ohjelmistokehitystä voidaan nopeuttaa Angular-ohjelmistokehyksen avulla ja minkälaisilla tavoilla Angular ohjelmistokehitystä nopeuttaa.

2.1 Ohjelmistokehitys

Ohjelmistokehityksen elinkaari koostuu kahdesta pääosasta, kehityksestä ja ylläpidosta. Kehitykseen kuuluu vaatimusmäärittely, toiminnallinen määrittely, arkkitehtuurisuunnittelu, toteutus, julkistus ja käyttöönotto. Seuraavaksi tutustutaan näihin ohjelmistokehityksen osa-alueisiin lyhyesti.

Ohjelmistokehitys alkaa vaatimusanalyysilla, jonka tarkoituksena on selvittää ohjelmistolle asetetut tai asetettavat tavoitteet sellaisella tarkkuudella, että niiden avulla voidaan lopulta kehittää haluttu ohjelmisto. Jos ohjelmistokehitykseen käytetty malli on lineaarinen, on vaatimusanalyysin tarkoitus tuottaa täydellinen lista vaatimuksista. Jos taas malli on iteratiivinen, antaa vaatimusanalyysi vaatimuksen aina kyseiselle iteraatiokierrokselle. (2.) Vaatimusanalyysi kerää vaatimuksia seuraavista vaatimustyypeistä:

- Käyttäjän vaatimukset

  Käyttäjän vaatimukset ovat vaatimuksia ohjelmiston tarjoamille palveluille ja itse ohjelmistolle asetettavat rajoitukset. Vaatimukset kuvataan luonnollisella kielellä ja kaavioilla. (2.)

- Järjestelmävaatimukset

  Järjestelmävaatimukset puolestaan kuvaavat ohjelmiston palvelut ja ohjelmistoon kohdistuvat rajoitukset yksityiskohtaisesti. Kaikki järjestelmävaatimukset

kuvataan samalla formaatilla, ja järjestelmävaatimuksia saatetaan myös käyttää asiakkaan ja ohjelmistoyrityksen välillä. (2.)

- Ohjelmiston määrittelykuvaukset

  Määrittelykuvaukset ovat yksityiskohtaisia ja kattavia kuvauksia ohjelmistosta. Niiden kuvauksissa käytetään standardoituja menetelmiä, kaavioita ja strukturoitua kuvausta ohjelmistosta, ja samalla tämä toimii ohjelmiston suunnittelun esiasteena. (2.)

Vaatimusmäärittelyn jälkeen alkaa järjestelmäsuunnittelu, jossa tarkastellaan itse järjestelmien välistä työnjakoa, ja tässä vaiheessa voidaan esimerkiksi tutkia, mitä laitteistoja tai ohjelmistoja kehitettävä ohjelmisto tarvitsee. Samalla voidaan päättää itse ohjelmistoista ja esimerkiksi käytettävistä ohjelmointikielistä. (2.)

Järjestelmäsuunnittelua seuraa ohjelmistosuunnittelu, joka on prosessi, jossa määritellään ja tarkastellaan arkkitehtuuria, moduuleita, rajapintoja ja itse tietoa, jolla ohjelmisto pystyy täyttämään vaatimusmäärittelyn tavoitteet (3). Ohjelmistosuunnittelu koostuu kahdesta vaiheesta, joista toinen on toiminnallinen määrittely ja toinen tekninen määrittely. Toiminnallisessa määrittelyssä kuvataan kaikki ohjelmiston toiminnot ja se, miten ohjelmisto on liitetty järjestelmän ulkopuolelle. Tuloksena saadaan määrittelydokumentti, joka kuvaa koko ohjelmiston toimintaa ja sen käyttämistä. Määrittelydokumentti ei ota kantaa siihen, millä tavalla ohjelmisto tulisi toteuttaa. Toiminnallisen määrittelyn vastakohta on tekninen määrittely, joka suoritetaan toiminnallisen määrittelyn jälkeen. Tässä vaiheessa ei enää tehdä päätöksiä ohjelmiston toiminnallisuuksista, vaan teknisen määrittelyn tavoitteena on puhtaasti kuvata ohjelmiston tekninen arkkitehtuuri mahdollisimman tarkasti. Tämä sisältää päätökset esimerkiksi ohjelmistossa käytettävistä ohjelmointikielistä, kirjastoista, rakenteesta tai rajapinnoista.

Ohjelmiston määrittelyn valmistuttua ohjelmiston toteuttaminen voidaan aloittaa. Toteuttamisvaihe koostuu itse ohjelmiston koodin kirjoittamisesta ja samalla ohjelmadokumenttien laadinnasta sekä yksikkötestauksesta. Itse ohjelmointiin liittyy koodin kirjoittamisen lisäksi niin ohjelmiston sisäisten liitäntöjen kuin tietokantarakenteiden suunnittelu ja ohjelmiston kääntäminen. (4.) Toteuttamisvaiheeseen kuluva aika riippuu hyvin usein siitä, kuinka hyvin toteutusta edeltävä määrittely on tehty ja tarvittavat oheiskomponentit kuten

kuvat ja äänet on tuotettu valmiiksi. Mikäli edellä mainitut asiat on toteutettu hyvin ja valmiina käytettäväksi, vie toteutus yleensä vain noin 10–20 % koko ohjelmistokehitykseen kuluvasta ajasta. Toteutuksen jälkeen tuloksena on suoritettava ohjelmisto, jossa kuitenkin on yleensä suhteellisen paljon toiminnallisia virheitä.

Ohjelmiston tuotannon valmistuttua on viimeisenä vaiheena ohjelmiston kehityksessä testaaminen. Testaaminen pyrkii löytämään ohjelmistosta virheitä. Virheet voivat johtua esimerkiksi ohjelmiston komponenttien välisestä odottamattomasta vuorovaikutuksesta tai esimerkiksi ongelmasta ohjelmiston rajapintojen kanssa (5, s. 17). Vaikka ohjelmiston kehitys ja tuotanto olisi todella huolellista, löytyy virheitä siitä riippumatta kaikista ohjelmistoista. Tämän takia ohjelmiston laatuun tulee kiinnittää huomiota koko ohjelmistokehityksen ajan, ja esimerkiksi yrityksille on todella tärkeää löytää nämä virheet itse eikä jättää virheiden löytymistä asiakkaalle, joka sitten kertoo huonoista kokemuksistaan muille.

Testauksen jälkeen, kun virheet on löydetty ja korjattu on julkaisemisen aika. Julkaisulla tarkoitetaan tapahtumaa, jossa valmis ohjelmisto toimii lähes virheettömästi, ja se julkaistaan ja ohjelmistosta tiedotetaan käyttäjille. Ohjelmiston julkaisemisessa on kuitenkin myös aina riskit mukana, kuten Kari Kakkonen kirjoittaa blogissaan ”Leikkaussalissa tai julkaisemassa ohjelmistoa? – Punnitse ensin riskit!”. Riskinä on aina, että julkaisun jälkeen ohjelmisto ei toimikaan oikealla tavalla esimerkiksi sinne jääneiden virheiden takia tai ohjelmisto saattaa olla haavoittuvainen tietomurrolle. Julkaisemisessa on hyvä pitää mielessä, että ohjelmistoa ei aina tarvitse julkaista yhtenä isona kokonaisuutena vaan useimmiten ohjelmisto voidaan julkaista pienemmissä osissa ja näin myös vähentää julkaisemisen mukanaan tuomaa riskiä. (7.)

Lopuksi ohjelmisto vaatii jonkinlaista ylläpitoa. Tämä tarkoittaa esimerkiksi web-ympäristössä huolehtimista siitä, että ohjelmisto on aina saatavilla ja ajan tasalla (8). Ylläpitoon kuuluu myös ohjelmiston virheistä raportoiminen tuotannolle ja virheiden korjaaminen. Samalla ylläpidossa täytyy huolehtia jatkuvasti siitä, että asiakkaat ovat tyytyväisiä tuotettuun ohjelmistoon.

2.2 Ohjelmistokehityksen nopeuttamisen tavoitteet

Ohjelmistokehityksen nopeuttamiseen voi olla useita syitä, ja siihen on myös monia erilaisia tapoja. Seuraavaksi käsittelen ohjelmistokehityksen nopeuttamisen yleisiä tavoitteita.

Sellaisilla teknologian markkinoilla, joilla tuotteet vanhentuvat nopeasti, voi ohjelmistokehityksen nopeus olla hyvinkin merkittävää, sillä silloin nopeus vaikuttaa suoraan siihen, kuinka usein ohjelmistoja saadaan ulos ja kuinka nopeasti vanhentuneiden tuotteiden tilalle saadaan uusia. Markkinoilla, joilla tuotteiden julkaiseminen osuu tiettyyn aikaikkunaan, kuten kausi- tai lomakohtaiset tuotteet, on kehityksen nopeudella merkitystä siinä, että näihin aikaikkunoihin osutaan. On kuitenkin hyvä huomata, että tällä sektorilla usein ohjelmiston kehityksen ennustettavuus on tärkeämpää kuin pelkkä nopeus. (6, s. 176.)

Ohjelmistokehityksen nopeuttaminen voi auttaa myös siinä, kuinka paljon ehditään tehdä muutoksia ohjelmistoon ilman että ohjelmistolle asetettu julkaisupäivämäärä kärsii. Samoin tämä voi auttaa virheiden korjauksessa. Kun ohjelmisto saadaan nopeammin valmiiksi, jää itse testaamiselle ja virheiden löytämiselle sekä korjaamiselle enemmän aikaa.

Ohjelmistokehityksen nopeuttamisen tavoitteet voivat kuitenkin usein myös olla pelkästään taloudellisia ja tuloksellisia. Voidaan helposti ajatella, että ohjelmistojen kehittämisen nopeuttaminen auttaa saamaan uusia tuotteita markkinoille, mikä auttaisi yrityksen taloustilannetta. Kuitenkin jos ohjelmiston kehityksen nopeuttamista ei tehdä huolellisesti, voi tämä johtaa huonompilaatuisiin tuotteisiin. Tällainen ajattelutapa on usein hyvin lyhytnäköistä, eikä itse liiketoimintaprosessien parantamiseen ole pitkällä aikavälillä kiinnostusta. (6, s. 176.)

Yksi ohjelmistokehityksen nopeuttamisen tavoite on myös yksinkertaisesti tuottavuuden parantaminen. Tämä on myös itselleni töissä tullut useimmiten vastaan, ja tästä minulla on siksi myös omakohtaista kokemusta. Useimmiten tuottavuuden parantamisella koetetaan saada joko ohjelmistoja useammin ulos tai suurissa ohjelmistoprojekteissa uusia toiminnallisuuksia useammin ulos tuotantoon. Tuottavuuden parantamisella voidaan

myös tavoitella kustannuksien vähentämistä. (6, s. 173.) Omiin kokemuksiini perustuen on ohjelmistokehityksen nopeuttaminen ja näin tuottavuuden parantaminen oiva tapa, kunhan ohjelmistokehityksen nopeuttaminen tehdään järkevästi ja kun sitä jatkuvasti tutkitaan ja tarvittaessa nopeutettua tapaa muutetaan.

2.3 Ohjelmistokehityksen nopeuden mittaaminen

Ohjelmistokehityksen nopeutta voidaan mitata monella eri tavalla. Oikean mittaustavan valinta voi olla suhteellisen vaikea asia. Tähän kannattaa käyttää tarvittava määrä aikaa. Mittaukset pitäisi ainoastaan suunnitella vastaamaan joihinkin liiketoiminnan kysymyksiin. Tällaisten kysymyksien ei kuitenkaan koskaan pitäisi olla sellaisia, joissa tarkastellaan sellaista asiaa, jolla ei oikeasti ole ohjelmistokehityksessä merkitystä kuten ”Kuinka monta KLOC (Kilo lines of code) eli tuhatta riviä koodia tällä hetkellä ohjelmistossa on” (9; 10). Omasta kokemuksestani voin sanoa, että ohjelmiston nopeuden mittaamista on todella vaikeaa tehdä, jos katsoo pelkästään numeroita, sillä ne ovat usein vain todella pieni osa totuutta. Seuraavaksi tutustutaan muutamaan nopeuden mittaamisen tapaan.

- Läpimenoaika

  Läpimenoaika kuvaa sitä, kuinka kauan idealla kestää siirtyä tuotantoon. Läpimenoaikaan voidaan vaikuttaa esimerkiksi yksinkertaistamalla päätöksentekoa ja lyhentämällä aikaa, joka kuluu pelkästään odotteluun. Lyhyt läpimenoaika on myös todella hyvä tapa olla reaktiivisempi asiakkaita kohtaan. Läpimenoaika sisältää sykliajan. (9.)

- Sykliaika

  Sykliaika kertoo, kuinka kauan kestää muutoksen tekeminen ohjelmistoon ja toimittaa tehty muutos tuotantoon. Sykliaika voi vaihdella tiimien mukaan minuuttien ja kuukausien välillä. (9.)

- Tiimin nopeus

  Tiimin nopeutta mitataan sillä, kuinka monta ”yksikköä” tiimi on onnistunut yhden iteraation aikana suorittamaan. Tällaisia numeroita pitäisi välttää nopeuden mittaamisessa, mutta tämän nopeuden tietäminen voi auttaa suuresti seuraavien iteraatioiden suunnittelemisessa. Tiimien nopeuden mittaaminen ei muuten kannata, sillä se perustuu ei-objektiivisiin arvioihin. Tämä saattaa johtaa virhearvioihin siitä, kuinka hyvin tiimi suoriutuu. (9.)

- Avoin ja suljettu vauhti

  Tämä nopeuden mittaamisen tapa kertoo, kuinka monta avointa tuotannon tehtävää raportoitiin ja suljettiin tietyn ajanjakson aikana. Tässä nopeuden mittarissa on tärkeää ymmärtää, että itse yleinen kehityksen suunta on tärkeämpää kuin tietyt numerot. (9.)

Nopeuden mittaaminen ohjelmistokehityksessä ei ole helppoa. Tämän takia onkin todella hyvä muistaa, että vaikka tulokset, joita mainituilla tavoilla saadaan, vaikuttaisivat hälyttäviltä, ei pelkästään näiden tulosten perusteella pidä vetää johtopäätöksiä siitä miksi tulokset näyttävät siltä. Tärkeää on aina kuulla kehitystiimin kommentit ja mietteet tuloksista sekä antaa tiimin pohtia, onko tavassa, jolla ohjelmistokehitystä tehdään, jotain parannettavaa. (9.)

2.4 Ohjelmistokehityksen nopeuttamiseen tavat

Paine ohjelmistokehityksen nopeuttamiseen voi olla hyvinkin kova, ja siinä voi myös helposti tehdä virheitä, jotka lopulta vain hidastavat ohjelmistokehitystä. Yrityksissä käydään paljon keskusteluja siitä, kuinka koodaajien pitäisi saada lyhyemmässä ajassa enemmän tehtyä. Yritykset tietenkin pitävät tästä ajatusmaailmasta, mutta yleensä tällainen ajattelu johtaa vain huonoihin seurauksiin ja kompromisseihin. Teknologia-alan yrityksissä yhdessä työpaikassa viihdytään suhteellisen lyhyen aikaa ja tiimien vaihtuvuus on suurta. Tämä johtuu esimerkiksi jatkuvista määräajoista, liian suurista odotuksista ja työviikoista, joista saattaa jäädä työntekijälle tekemistä vielä viikonlopuksikin. (11.)

Paine nopeuttamiseen tuskin koskaan teknologian alalta katoaa, mutta yrityksien ja tiimien tulisi kiinnittää enemmän huomiota omien työntekijöiden hyvinvointiin. Tähän ei riitä pelkästään se, että työntekijöillä on tarpeeksi vapaa-aikaa, vaan tarvitaan myös jatkuvaa kehitystä itse työpaikalla. Liian kova tahti ei loppujen lopuksi ole kestävää, sillä se johtaa tiimin nopeaan vaihtuvuuteen. Lisäksi uusia tekijöitä voi olla vaikea saada tiimiin, jos yrityksellä on sellainen maine, että tiimin vaihtuvuus on suurta. (11.) Seuraavaksi käyn läpi muutamia tapoja, joilla ohjelmistokehityksen nopeutta voi parantaa ilman, että työntekijät palavat loppuun.

Ensimmäinen asia, jolla ohjelmistokehityksen nopeutta voidaan parantaa, on palkata lisää työntekijöitä. Tämä kannattaa kuitenkin tehdä harkitusti, sillä uusien työntekijöiden palkkaus ei heti nopeuta ohjelmistokehitystä. Valintoja on tehtävä sen välillä, palkataanko osaavampia tekijöitä tiimiin, jotka pystyvät tekemään enemmän ja paremmin lyhyessä ajassa, vai palkataanko nuorempia uran alkuvaiheessa olevia tekijöitä. Uusien työntekijöiden palkkaus lopulta nostaa ohjelmistokehityksen nopeutta, mutta usein se aluksi vain laskee sitä. Uudet työntekijät täytyy kouluttaa, ja heidän pitää päästä sisälle omiin tehtäviinsä. Usein teknologian alalla uudelle työntekijälle myös nimetään ohjaaja, jolloin myös tämän ohjaajan aika ohjelmiston kehittämisessä vähenee. Samoin mitä enemmän työntekijöitä, sitä enemmän on koordinointia, mistä on pidettävä huolta. Jos tiimin koordinaatio ei toimi oikein, voi kehitykseen käytettävä aika vähentyä, kun joudutaan käyttämään aikaa huonosti järjestettyihin tapaamisiin ja aktiviteetteihin, jotka eivät suoraan nopeuta ohjelmistokehitystä. Hyvä tapa on esimerkiksi jakaa tiimi pienempiin osiin, jolloin jokaisella tiimillä on omat tehtävänsä ja yhden tiimin hidastuminen ei vaikuta niin suorasti toisen tiimin toimintaan ja kehitys on jatkuvasti käynnissä. (11.)

Aina ei kuitenkaan ole mahdollista tai kannattavaa palkata uusia työntekijöitä nopeuttaakseen kehitystä. Nopeutta voidaan hakea antamalla nykyisille työntekijöille mahdollisuus kouluttaa itseänsä ja näin parantaa omaa osaamistasoaan. Mitä parempia työntekijät ovat työssään, sitä nopeammin he onnistuvat ratkaisemaan ongelmia ja he pystyvät luomaan sellaisia ratkaisuja, jotka eivät ole monimutkaisia, ja näin myös tulevaisuudessa ohjelmiston kehittäminen on nopeampaa. Samalla kun ohjelmistokehityksen nopeus nousee, taidokkaammat työntekijät myös mahdollistavat parempilaatuisen tuotteen. (11.)

Suuri ongelma ohjelmistokehityksessä on työ, joka joudutaan tekemään uudelleen. Syitä tähän voi olla monia, mutta yleisimpiä kuitenkin ovat virheet ohjelmistossa ja epäselvät

vaatimukset (11). Ohjelmistokehitystä voidaan nopeuttaa siten, että kiinnitetään enemmän huomiota virheisiin ja siihen, että ne löydetään ajoissa. Kun virhe löydetään ajoissa, on useimmiten myös joku tiimistä tehnyt työtä sen ohjelmiston osan kanssa, josta virhe löytyi. Tämä taas nopeuttaa virheen korjaamista, kun koodi, joka virheen aiheuttaa, on työntekijän muistissa. Myös epäselviin vaatimuksiin on syytä kiinnittää huomiota ja parantaa vaatimuksien kuvauksia. Epäselvät vaatimukset johtavat turhaan työhön, joka joudutaan tekemään uudelleen, jotta alkuperäinen vaatimus saadaan täytettyä. (11.)

Viimeinen tapa, jonka tässä opinnäytetyössä käyn läpi, on palautteen pyytäminen asiakkaalta. Usein on vaikeaa tietää tarkasti, mitä asiakas haluaa, ja jos asiakkaalta ei jatkuvasti kysytä, mitä mieltä hän on tuotteesta, saatetaan aikaa kuluttaa sellaisiin ohjelmiston osiin, joita lopulta kukaan ei käytä. Kommunikointi asiakkaan kanssa vähentää arvailua ja nopeuttaa tuotteen valmistumista. Vaikka ohjelmisto ei olisikaan sellaisessa vaiheessa, jossa sitä voidaan asiakkaalle esitellä, voidaan kommentteja ohjelmistosta pyytää esimerkiksi prototyyppien, piirrosten ja kuvien avulla. (11.)

Teknisillä ratkaisuilla voidaan myös vaikuttaa ohjelmistokehityksen nopeuteen. Kehitystä voidaan nopeuttaa esimerkiksi käyttämällä ohjelmistokehyksiä, kirjastoja ja muita ohjelmistokehityksen työkaluja (12). On tärkeää tunnistaa, mitkä näistä työkaluista ovat oikeat projektiin, eikä niitä pitäisi käyttää vain siksi, että ne ovat olemassa. Hyvin valitut työkalut saattavat nopeuttaa kehitystyötä huomattavasti, kun kaikkea ei tarvitse kehittää itse alusta -asti. Tekniset ratkaisut eivät rajoitu pelkästään eri työkaluihin. Tekniset ratkaisut vaikuttavat aivan koodin tasolla asti, ja nopeutta kehitykseen voidaan hakea esimerkiksi kirjoittamalla paljon hyviä testejä, joilla mahdolliset virheet ohjelmistosta saadaan helposti kiinni. Kannattaa myös selvittää, olisiko esimerkiksi testivetoinen tai käyttäytymislähtoinen ohjelmointitapa ohjelmiston kehityksessä hyödyksi. Näistä ensimmäinen voi nopeuttaa kehitystä esimerkiksi siten, että se auttaa kehittäjiä ymmärtämään paremmin kirjoittamaansa koodia ja ehkäisee virheitä. (13.)

2.5 Ohjelmistokehityksen nopeuttamisen hyödyt yrityksessä

Ohjelmistokehityksen oikeanlaisesta nopeuttamisesta saatavat hyödyt ovat hyvin monenlaisia. Hyvin toteutettu nopeuttaminen näkyy esimerkiksi yrityksen parantuneessa tuloksessa. Nopeutettu ohjelmistokehitys oikein tehtynä tuottaa nopeammin uusia tuotteita

markkinoille tai uusia ominaisuuksia jo olemassa oleviin ohjelmistoihin. Mahdollisissa tapauksissa, joissa nopeutta on haettu myös ohjelmiston virheiden ehkäisemisellä ja niiden aikaisin löytymisellä, voi ohjelmistokehityksen nopeutus myös vaikuttaa positiivisesti ohjelmiston laatuun.

Samalla kun ohjelmistoja saadaan useammin ulos tai uusia ominaisuuksia vanhoihin sovelluksiin, ovat myös sovelluksien käyttäjät tyytyväisempiä, ja näin saadaan nostettua yrityksen ja yrityksen ohjelmistojen mainetta. Mahdollisesti vielä, jos ohjelmistokehitysprosessin aikana on oltu paljon yhteydessä asiakkaaseen ja sen kanssa on tehty jatkuvaa kehitystyötä läpi prosessin, on myös asiakaskokemus parempi.

Ohjelmistokehityksen kaikki hyödyt eivät näy vain yrityksen tuloksessa tai ohjelmiston laadun parantumisena. Hyvin toteutetun prosessin jälkeen parannusta näkyy myös työntekijöiden hyvinvoinnissa ja heidän motivaatiossaan ja osaamisessaan. Kun kehittäjillä ei ole jatkuvaa korkea painetta, todella tiukkoja määräaikoja, suuria odotuksia eikä viikonlopulle valuvia työviikkoja, on työntekijöiden jaksaminen ja heidän tekemänsä työn laatu parempaa, jolloin myös ohjelmiston laatu paranee. Myös mahdollinen työntekijöiden kouluttaminen ja kehityksen tukeminen tuottaa pitkällä aikavälillä yritykselle niin parempilaatuista ohjelmistoa kuin myös taitavamman kehittäjän, joista jo tälläkin hetkellä Suomessa on pula ja joiden palkkaaminen on todella vaikeaa. (14.) Yritykselle on nykyään tärkeää, että kehittäjillä on mukava olla töissä ja yrityksessä tekijöiden vaihtuvuus pientä. Varsinkin nykyään, kun uudet juuri valmistumassa olevat kehittäjät otetaan suoraan töihin jo koulunpenkiltä (15) on tällainen maine yritykselle hyvästä ja helpottaa uusien kehittäjien palkkaamista yritykseen.

3 Angular-ohjelmistokehyksen perusteet

Angular on Googlen vuonna 2010 julkaisema ja ylläpitämä TypeScript-kielellä kirjoitettava ohjelmistokehys (29), jonka avulla voidaan kehittää yksisivuisia sovelluksia. Angular tai usein myös Angular 2 -nimellä tunnettu ohjelmistokehys on kokonaan uudelleen kirjoitettu ohjelmistokehys, joka korvasi aikaisemman AngularJS-ohjelmistokehyksen. Tämän uudelleen kirjoitetun Angular-version kerrottiin olevan kehityksessä jo vuonna 2014. Vuonna 2015 se siirtyi beetavaiheeseen, ja viimein vuonna 2016 Angular julkaistiin. Vaikka vanhan AngularJS-version kokonaan uudelleenkirjoitus aiheutti kehittäjissä

vastustusta, on Angular tämän jälkeen kasvattanut käyttäjäkuntaansa huomattavasti. (29.) Uusin Angularin versio tämän opinnäytetyön kirjoitushetkellä on versio 8, ja tässä työssä tutustutaan Angulariin tätä versiota käyttämällä. Nykyään Angular julkaisee uuden version aina puolen vuoden välein, ja vanhempia versioita tuetaan 18 kuukautta niiden julkaisupäivämäärästä.

3.1 Yhden sivun ohjelmisto

Yhden sivun ohjelmisto tai paremmin tunnettuna SPA (Single Page Application) on verkossa toimiva ohjelmisto tai verkkosivu, joka lataa kaiken tarvittavan tiedon palvelimelta kerralla (20). Tämän jälkeen ohjelmisto dynaamisesti muokkaa näyttämäänsä sivua sen sijaan, että uusi sivu ladattaisiin aina palvelimelta. SPA-sivustoja on voitu kirjoittaa jo pitkään, ja termiä SPA käytettiin ensimmäisiä kertoja vuonna 2002. Kuitenkin vasta nykyaikaiset SPA-ohjelmistokehykset ovat tuoneet SPA-sivustot laajemmin käyttöön. Tämä johtuu siitä, että nämä ohjelmistokehykset vähentävät huomattavasti kehittäjiltä vaadittua työtä, joka kuluu SPA-sivuston rakentamiseen. Vaikkakin SPA-sivuston ensimmäinen lataus on yleisesti hitaampi kuin tavallisen verkkosivuston lataus, mahdollistaa SPA paljon käyttäjäystävällisemmän kokemuksen. Tämä johtuu siitä, että ohjelmisto pystyy välttämään käyttökokemuksen keskeytymisen sivujen siirtymisen välillä. SPA-ohjelmistossa kaikki tarvittavat tiedostot, kuten HTML-, CSS- ja JavaScript-tiedostot ladataan palvelimelta heti ensimmäisellä sivuston latauskerralla. Tämän jälkeen käyttäjälle näytettyä näkymää muokataan käyttäjän tekemien toimintojen perusteella. Tämä mahdollistaa sen, että itse sivusto ei koskaan joudu latautumaan uudelleen. Kun sivusto tarvitsee lisää tietoa palvelimelta, pyydetään tieto taustalla yleensä HTTP-pyynnöillä halutusta rajapinnasta. (20.)

3.2 Angular-komentorivin käyttöliittymä, Node.js ja NPM

Angular CLI on virallinen komentorivin käyttöliittymä. Sen käyttäminen on todella suositeltavaa, sillä Angular-ohjelmistot ovat melko tarkkoja ohjelmiston käynnistämisen kulusta. Tämä käsittää niin tiedostojen muuntamista oikeaan muotoon, ennen kuin ne voidaan näyttää selaimessa, kuin myös tiedostojen optimoimista. Angular CLI tekee tämän kaiken, jolloin myös varmasti saadaan tehokkaasti toimiva Angular-ohjelmisto käyttöön.

Ennen kuin voidaan aloittaa Angular CLI:n käyttö, vaatii Angular CLI muutaman edellytyksen. Nämä vaatimukset ovat Node.js-versio 10.9.0 tai myöhempi ja NPM-paketinhallintajärjestelmä.

Node.js on JavaScript-ympäristö, jolla voidaan suorittaa koodia palvelimella. Angular tarvitsee Node.js-ympäristön käyttöönsä, sillä taustalla Angular CLI muuntaa ja optimoi Angular-projektin. NPM taas tarkoittaa kahta asiaa. NPM on maailman suurin sovelluskirjasto, joka sisältää yli 800 000 koodikirjastoa. Se on myös näiden sovelluskirjastojen hallinnointiohjelma ja asentaja. Angular tarvitsee siis NPM:n itse Angularin vaatimuksien asentamiseen, mutta Angular-kehityksessä NPM-asentajaa käytetään todella usein, koska se myös mahdollistaa koodikirjastojen lataamisen Angular-sovellukseen ja näin ollen vähentää tiettyihin ominaisuuksiin kuluvaa kehitysaikaa.

Molemmat vaatimukset voidaan asentaa lataamalla Node.js-asennusohjelmisto osoitteesta https://nodejs.org/. Tältä sivulta ladataan viimeisin LTS-versio, joka kirjoitushetkellä on 12.13.0. Tämän jälkeen voidaan suorittaa Node.js asennusohjelma, ja näin kumpikin Node.js ja NPM, ovat valmiita käyttöön.

3.3 Uuden Angular-projektin luominen

Ennen uuden projektin luomista täytyy Angular CLI asentaa. Se voidaan asentaa suorittamalla seuraava komento:

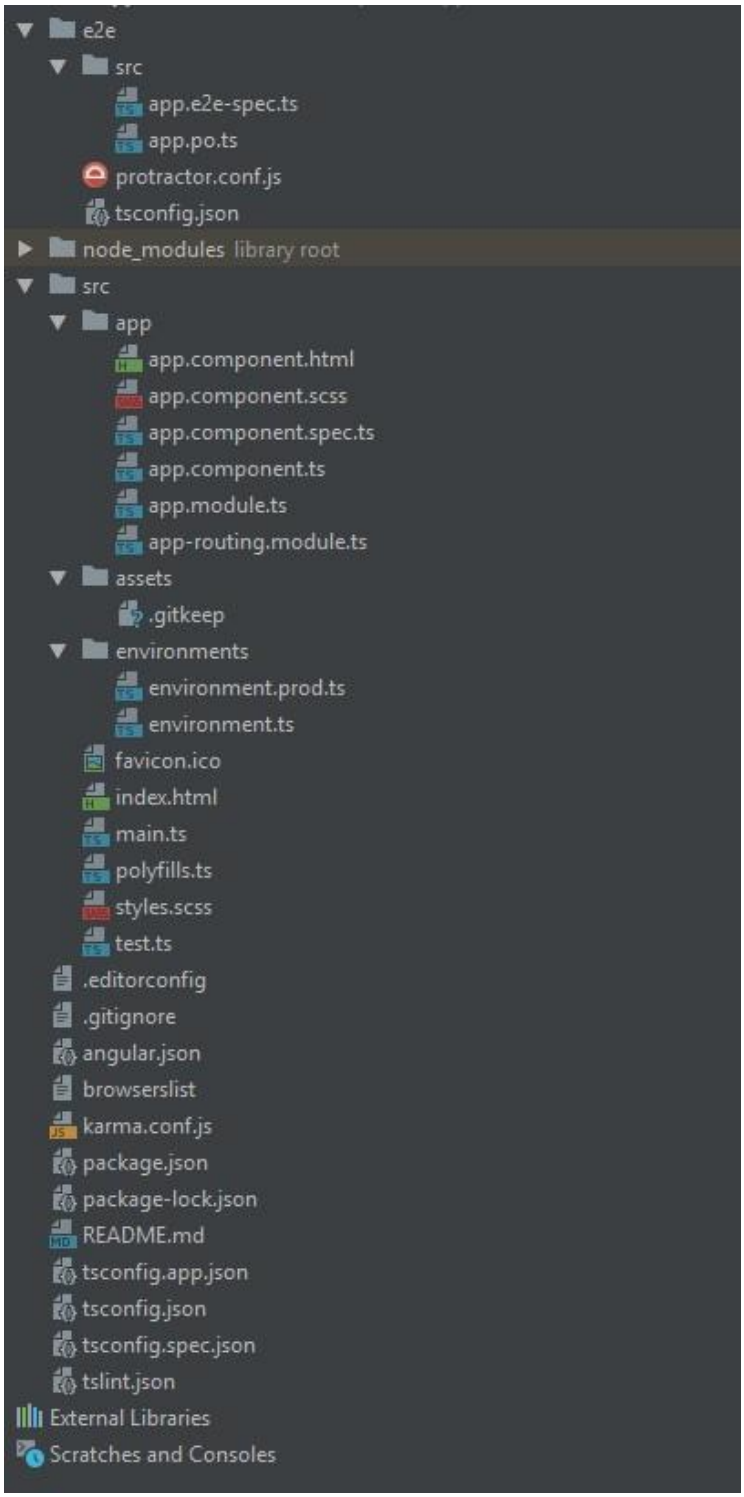
```
npm install -g @angular/cli@latest
```

Tämä komento asentaa Angular CLI:n NPM:n avulla. Komennossa oleva -g kertoo asennukselle, että se asennetaan tietokoneelle globaalisti, jolloin sitä voidaan käyttää mistä tahansa tietokoneella. @angular/cli on asennettavan paketin nimi, ja @latest kertoo asennusohjelmalle, että Angular CLI:stä halutaan asentaa viimeisin versio.

Uusi Angular-projekti luodaan suorittamalla seuraava komento komentorivillä, jossa <projektin-nimi> korvataan halutulla projektin nimellä. Tämä luo projektin kansion siihen sijaintiin, mikä komentorivillä on valittuna.

```
ng new <projektin-nimi> --style=scss –routing
```

Huomattavaa edellä näkyvässä komennossa ovat liput –style ja –routing. Kumpikin näistä lipuista on täysin vapaaehtoisia. --style=scss kertoo, että uudessa projektissa halutaan käyttää CSS-esiprosessoria SCSS. SCSS-esiprosessori tekee CSS:n kirjoittamisesta ja lukemisesta huomattavasti helpompaa. –routing-lippu taas tarkoittaa, että projektissa halutaan käyttää reititystä, ja tämä lippu laittaakin Angular CLI:n luomaan tarpeelliset tiedostot reitityksen käyttämiseksi. Tämän jälkeen Angular-projekti on luotu ja valmis muokattavaksi. Uuden Angular projektin tiedostorakenne on kuvan 1 mukainen.



Kuva 1. Angular-ohjelmiston tiedostorakenne.

3.4 Typescript-ohjelmointikieli

Angular-ohjelmistoja kirjoitetaan TypeScript-ohjelmointikielellä. TypeScript on JavaScript-kielen varaan rakennettu ohjelmointikieli, joka tuo JavaScriptiin paljon uusia ominaisuuksia (18). Merkittävin näistä ominaisuuksista on vahva tyypitys, josta TypeScript saa myös nimensä. TypeScript siis mahdollistaa muuttujien, esimerkiksi ikämuuttujan, tyypityksen. Tämä auttaa ohjelmoijaa tietämään, mitä tyyppiä hänen käyttämänsä muuttuja on, eikä hänen näin ollen tarvitse aina palata tarkistamaan muuttujan tyyppiä, jos se on sattunut unohtumaan. Samalla TypeScript nopeuttaa ohjelmoijan työtä ja helpottaa tuotetun koodin ylläpitämistä (19). TypeScriptiä ei kuitenkaan voida suorittaa selaimessa, joten kun Angular-projekti suoritetaan selaimessa, Angular CLI muuntaa TypeScriptin JavaScriptiksi.

3.5 Projektin tiedostorakenne

Seuraavat kansiot ja tiedostot ovat Angular-projektin ylätason tiedostoja ja kansioita:

- E2E-kansio

  E2E-kansio on tarkoitettu E2E-testien kirjoittamista varten. Angular CLI luo kansioon myös valmiiksi muutaman valmiin E2E-testin. (17.)

- node_modules-kansio

  Node_modules-kansio on tarkoitettu NPM-kirjastojen tallennuspaikaksi. Kaikki projektissa käytettävät kirjastot voidaan asentamisen jälkeen sisällyttää projektiin tästä kansiosta. (17.)

- src-kansio

  Src-kansio on kansio, jossa ohjelmiston lähdekoodi sijaitsee. Kansiossa on app-kansio, jossa ohjelmiston juurikomponentti sijaitsee. Lisäksi kansiossa on assets-kansio, joka on tarkoitettu kaikille ohjelmiston resursseille, kuten kuville (17).

Kansiossa on myös environments-kansio, joka pitää sisällään Angular-ohjelmiston ympäristön konfiguraatiotiedostot. Environments-kansioon voidaan tarvittaessa myös kirjoittaa omia konfiguraatiotiedostoja. Lisäksi src-kansiossa on kolme tiedostoa. Index.html on tiedosto, jonka selain renderöi, kun sivusto ladataan. Styles.scss-tiedosto on tyylitiedosto, johon voidaan kirjoittaa ohjelmistossa globaalisti käytettävät tyylit. Main.ts-tiedosto toimii ohjelmiston alkupisteenä, joka kääntää AppModuulin ja renredöi tarvittavan näkymän selaimeen.

- package.json-tiedosto

  Package.json on tiedosto, johon on tallennettu kaikki ohjelmiston käyttämät ja NPM:n kautta asennetut kirjastot (17). Tämä tiedosto mahdollistaa sen, että esimerkiksi node_modules-kansiota ei tarvitse tallentaa versionhallintaan, koska tämä tiedosto pitää huolen siitä, että halutut paketit asentuvat aina samalla tavalla ja samalla versiolla.

3.6 Projektin kehityspalvelimen käynnistäminen

Kun Angular-ohjelmisto on saatu luotua, se voidaan käynnistää seuraavalla komennolla:

```
ng serve
```

Tämä käynnistää Angular-ohjelmiston oletusportissa 4 200 ja avaa ohjelmiston tietokoneen oletusselaimessa. Edellä mainitulle komennolle voi myös antaa lippuja, joilla ohjelmiston käynnistämiseen voidaan vaikuttaa. Näistä useimmiten käytetyt ovat liput --port ja -c. –port-lippu mahdollistaa sen portin vaihtamisen, jossa ohjelmisto on nähtävillä. -c-lippu puolestaan mahdollistaa käytettävän konfiguraatiotiedoston valitsemisen. Näitä lippuja käyttämällä voisi käynnistämiskomento olla esimerkiksi seuraavanlainen:

```
ng serve -c local --port=5500
```

4 Angular-ohjelmistokehyksen arkkitehtuuri ja ominaisuudet

Tässä luvussa käydään läpi Angular-ohjelmistokehyksen arkkitehtuuria ja tutustutaan tarkemmin Angular-ohjelmistokehyksen kehittäjille tarjoamiin työkaluihin.

4.1 Moduulit

Angularin moduulit ovat tapa niputtaa Angularin muita rakennuspalikoita yhteen, kuten komponentteja, direktiivejä, palveluita ja putkia (22). Moduulit pitävät huolen siitä, että Angular-projekti tietää esimerkiksi edellä mainittujen palikoiden olemassaolosta. Angular ei automaattisesti skannaa projektin tiedostorakennetta ja etsi kaikkia mahdollisia rakennuspalikoita, ja siksi Angularille pitää kertoa moduulissa, mitkä palikat ovat saatavilla ja missä. Toisin sanoen moduulit auttavat Angular-ohjelmistoa ymmärtämään ja analysoimaan projektin rakennetta. Jokaisessa Angular-projektissa pitää siis olla vähintäänkin yksi moduuli (22). Angular CLI luo tällaisen moduulin nimeltä app.module.ts, kun CLI luo uuden projektin. Moduuleita voi kuitenkin olla myös useita, ja useimmiten laajemmissa ohjelmistoissa se on jopa suotavaa. Usean moduulin olemassa oleminen mahdollistaa sen, että kaikki ohjelmiston rakennuspalikat eivät ole samassa moduulissa, ja tällä voidaan nopeuttaa ohjelmiston toimintaa ja tehdä siitä helpommin hallittava. (21)

Angular myös itsessään tarjoaa joitain moduuleita valmiina. Esimerkki tällaisesta moduulista on lomakemoduuli, joka voidaan tuoda omaan moduuliin, minkä jälkeen se on käytettävissä ohjelmistossa. Tällä pystytään estämään se, ettei ohjelmoitsijan tarvitse itse tuoda ohjelmistoon useita eri direktiivejä ja muita rakennuspalikoita, kun ne voidaan tuoda kaikki kerralla moduulin mukana. (21.)

Tärkeää on kuitenkin muistaa, että Angular-moduulit eivät keskustele keskenään millään tavalla. Tämän takia, jos Angular-moduulissa haluaa käyttää toista moduulia, pitää tämä moduuli tuoda siihen moduuliin, jossa sitä halutaan käyttää. Jos ohjelmistossa on useita palikoita, joita käytetään melkein jokaisessa ohjelmiston osassa, kannattaa luoda näistä palikoita oma moduuli. Tämä helpottaa myös ohjelmiston ylläpidettävyyttä, kun sen sijaan, että jokainen tarvittava rakennuspalikka tuotaisiin moduuliin erikseen, voidaan moduuliin vain tuoda toinen moduuli, jossa nämä kaikki rakennuspalikat ovat valmiina. (21.)

4.2 Komponentit

Komponentit ovat Angular-ohjelmiston tärkein osa, ja koko ohjelmisto koostuu pääosin komponenteista. Angular CLI:n luodessa projektin luo CLI app.component.ts komponentin, joka on Angular-ohjelmiston juurikomponentti, jonka päälle myöhemmin rakennetaan muut komponentit. (21.)

Hyviä esimerkkejä ohjelmiston osista, jotka usein luodaan omiksi komponenteiksensa, ovat ohjelmiston ylä- ja alapalkit. Samoin myös sivupalkki tai esimerkiksi kirjautumisnäkymä voisi olla oma komponenttinsa. Myös tätä pienemmät tai suuremmat toiminnallisuudet tai ohjelmiston palaset voivat olla omia komponenttejansa, mutta niiden ei ole pakko olla. Komponentit ovat todella hyödyllisiä, sillä ne mahdollistavat tietyn ohjelmiston palasen uudelleenkäyttämisen. (23.) Tällä tavalla esimerkiksi sivuston yläpalkkia ei tarvitse aina kirjoittaa uudelleen, vaan kun se on luotu omaksi komponentiksensa, voidaan sitä helposti käyttää joka puolella ohjelmistoa (21).

Jokaisella komponentilla on oma HTML-, tyyli- ja TypeScript-tiedosto. Tämä myös mahdollistaa sen, että jokaisella komponentilla on oma logiikkansa. Tämä helpottaa kehittäjän työtä ja ennaltaehkäisee mahdollisia konflikteja koodissa. Samalla kun logiikkaa voidaan käyttää joka puolella ohjelmistoa ja logiikka ei tarvitse massiivisia HTML- tai TypeScript-tiedostoja, vähentävät komponentit tälläkin tavalla kehittäjän työtä ja tekevät ohjelmiston rakenteesta paljon ylläpidettävämmän ja siistimmän. (21.)

Uusi Angular-komponentti voidaan luoda seuraavanlaisella komennolla:

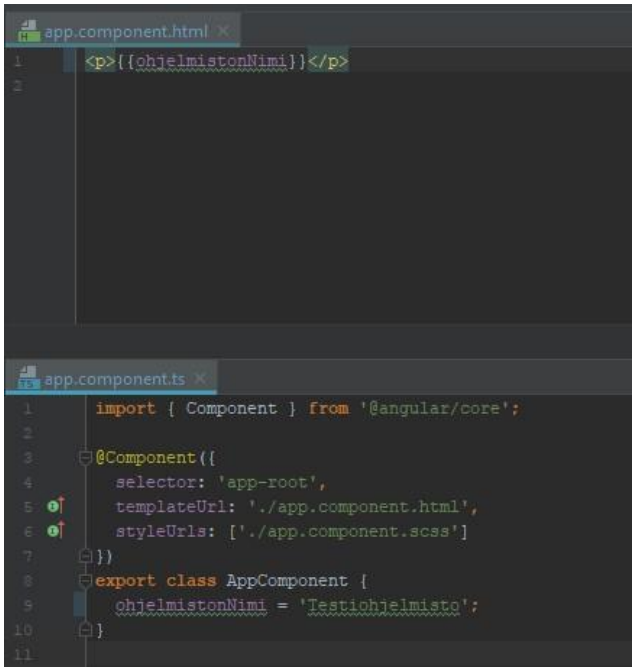
```
ng generate component <komponentin-nimi>
```

Komennossa <komponentin-nimi> korvataan sillä nimellä, mikä komponentille halutaan antaa. Tämä komento luo ohjelmiston tiedostorakenteessa kansioon app uuden kansion annetulla komponentin nimellä, ja tähän uuteen kansioon komento luo kaikki komponentin tarvitsemat tiedostot. Näitä tiedostoja ovat HTML-, tyyli-, TypeScript- ja testitiedostot. Lisäksi Angular CLI lisää luodun komponentin juurimoduuliin (app.module.ts), jotta Angular-ohjelmisto tietää uuden komponentin olemassaolosta. (21.)

4.3 Datakytkentä

Yksi Angular-ohjelmiston tärkeimpiä ominaisuuksia on myös datakytkentä, joka mahdollistaa kommunikaation TypeScript- ja HTML-tiedostojen välillä. Tämä myös mahdollistaa sen, että itse TypeScript-puolella voidaan tehdä esimerkiksi laskutoimituksia tai hakea tietoa vaikka rajapinnasta ja näyttää se heti sen jälkeen käyttäjälle ohjelmistossa. Datakytkentä on ratkaisu siihen, miten TypeScript- ja HTML-tiedostot pystyvät keskustelemaan Angular-ohjelmistossa. (21.)

Merkkijonon interpolointi eli string interpolation on yksi datakytkennän tyypeistä (24). String interpolation toimii siten, että HTML-tiedoston puolella haluttuun kohtaan lisätään kaksoisaaltosulkeet ja tämän sisään haluttu arvo TypeScript-tiedostosta kuten kuvassa 2. Tämän jälkeen Angular näyttää halutun arvon itse HTML-tiedostossa. Ainoat vaatimukset string interpolationille ovat, että arvo, joka string iterpolationin sisään annetaan, on pystyttävä lopulta muuttamaan merkkijonoksi. Esimerkiksi listaa tai objektia ei suoraan voi syöttää string interpolationiin, mutta edellä mainitut voidaan näyttää string interpolationissa esimerkiksi JSON-putkella, joka muuttaa halutun arvon merkkijonoksi. Samoin string interpolationin sisään ei voi antaa useamman rivin pituisia koodeja. (21.)



Kuva 2. String interpolation Angular-ohjelmistossa.

Ominaisuuden sidonta eli property binding on toinen Angular-ohjelmiston datakytkentätyypeistä. Property bindingilla voidaan dynaamisesti muuttaa HTML-elementtien ominaisuuksia. Hyvä esimerkki tällaisesta ominaisuudesta on painikkeen disabled ominaisuus, joka määrittelee, onko painike poistettu käytöstä. Property bindingia voidaan käyttää melkein kaikkiin HTML-elementteihin ja näin muokata elementin ominaisuuksia. Tämä onnistuu laittamalla elementin ominaisuuden hakasulkeiden väliin. Property binding hyväksyy erilaisia arvoja riippuen itse elementin ominaisuudesta, jota ollaan muokkaamassa. Näitä arvoja voivat olla esimerkiksi totuusarvo kuvan 3 mukaisesti mutta myös ihan pelkkä merkkijono.

app.component.html

```
<button [disabled]="painikePoisKaytosta"></button>
```

app.component.ts

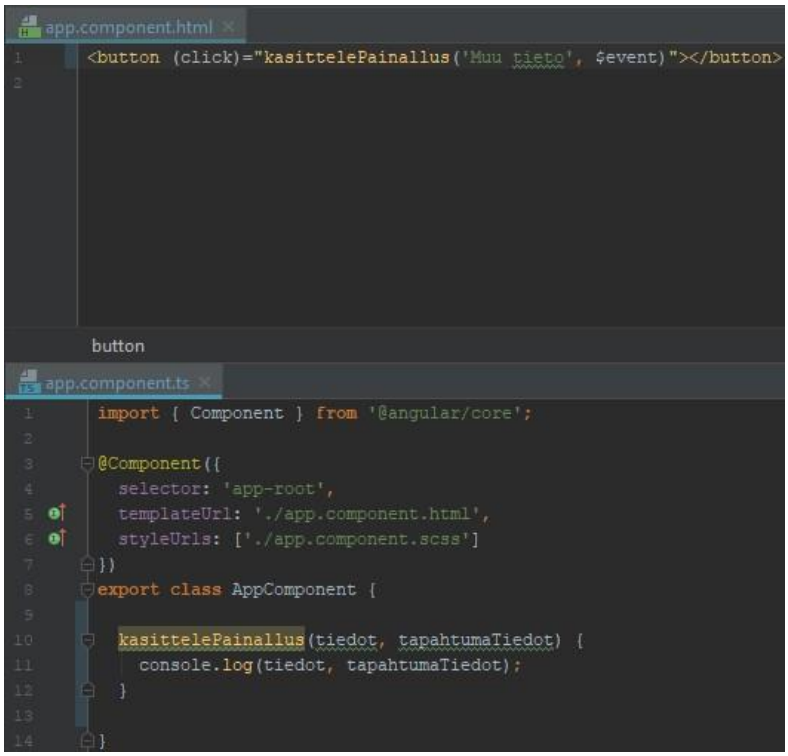
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  painikePoisKaytosta = true;
}
```

Kuva 3. Painikkeen poistaminen käytöstä ominaisuuden sidonnan avulla.

Property bindingia voidaan käyttää myös omiin direktiiveihin tai komponentteihin. Näin voidaan esimerkiksi siirtää tietoa kahden eri komponentin välillä. Tämä toimii samalla tavalla kuin HTML-elementin property binding. Ainoa ero on, että tietoa vastaanottava puoli täytyy valmistella sitä varten. Tämä onnistuu @Input() decoratorilla. Tällöin haluttu TypeScript-muuttuja on muidenkin ohjelmiston osien saatavilla ja kyseiseen muuttujaan voidaan kytkeytyä.

Myös toiseen suuntaan suuntautuva datakytkentä on usein tarpeellista. Tällöin halutaan reagoida useimmiten johonkin käyttäjän tekemään asiaan, ja tätä kutsutaan tapahtumakytkennäksi. Useimmiten käytettyjä datakytkentöjä ovat klikkaus-, muutos- ja

syöttötapahtumat. Esimerkiksi klikkaustapahtumassa voidaan haluta suorittaa koodia, kun käyttäjä klikkaa painiketta. Tapahtumakytkentä ottaa sisäänsä koodinpätkän, joka halutaan suorittaa tapahtuman tapahtuessa kuten kuvassa 4. Tämän koodin sisällä voidaan myös käyttää Angularin varattua avainsanaa $event. Sillä voidaan lähettää kaikki tapahtuman tiedot eteenpäin prosessoitavaksi. Tämä tieto useimmiten sisältää kaiken HTML-tapahtuman antamat tiedot.



Kuva 4. Tapahtumakytkentä, jossa tulostetaan funktiolle annetut tiedot, joita ovat teksti ”Muu tieto” ja painallustapahtuman tiedot.

Viimeisimpänä mutta mielestäni Angularissa tärkeimpänä datakytkentänä on kaksisuuntainen datakytkentä, josta Angular on myös tullut hyvin tunnetuksi. Se yhdistää niin tapahtumakytkennän kuin ominaisuuskytkennän. Kaksisuuntaisen datakytkennän tunnistaa koodissa seuraavanlaisesta direktiivistä:

```
[(ngModel)]
```

Kaksisuuntaisessa datakytkennässä siis, kuten myös edellä olevan esimerkin suluista voi huomata, yhdistyy tapahtuma ja ominaisuuskytkentä. Kaksisuuntaisessa datakytkennässä TypeScript-muuttuja asetetaan direktiivin sisään, kuten kuvassa 5.

Kaksisuuntainen datakytkentä mahdollistaa sen, että kumpikin, itse sivupohja ja TypeScript-tiedosto, voivat muokata samaa muuttujaa ja olla tietoisia tämän muutoksista. Esimerkiksi tekstikenttä voi muuttaa muuttujaa koodin puolella, mutta samalla jos koodi muuttaa muuttujan arvoa, muuttuu arvo tekstikentässä.

app.component.html

```
<input type="text" [(ngModel)]="nimi">
```

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  nimi = 'Matti Meikäläinen';

}
```

Kuva 5. Kaksisuuntainen datakytkentä. Tekstinsyöttökenttä muuttaa TypeScript-tiedoston nimi-muuttujaa ja näyttää muutokset HTML-tiedostossa, jos jokin ulkopuolinen koodi muuttaa nimen arvoa.

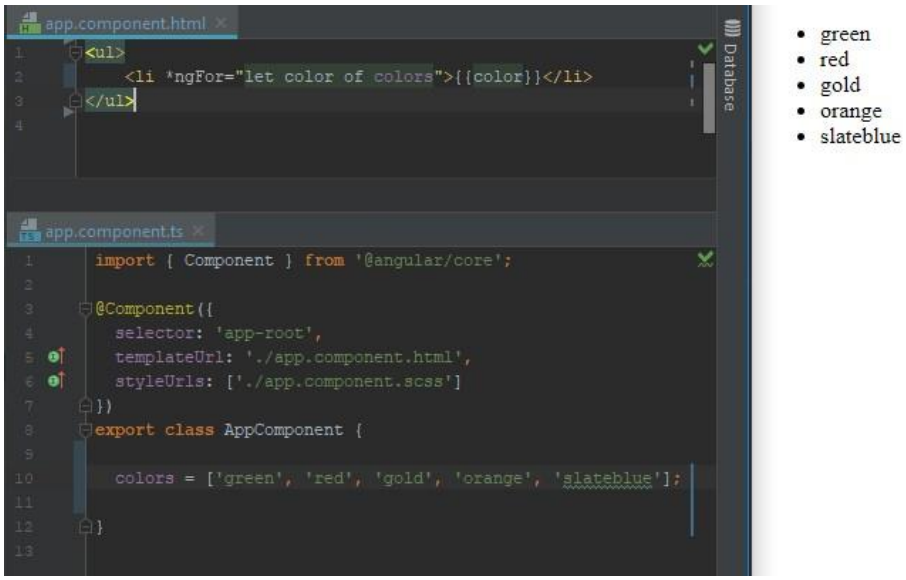
4.4 Direktiivit

Direktiivit ovat tapoja antaa DOM:lle ohjeita (25). Myös komponentit ovat direktiivejä, mutta komponenteilla on myös sivupohja. Komponentit siis myös ovat direktiivejä, ja niissä direktiivi auttaa DOM:a tietämään, mihin kohtaan DOM:ssa komponentti halutaan tulostaa. Direktiivejä on niin rakenteellisia kuin myös ominaisuudellisia. Direktiivejä on valmiiksi Angularissa, mutta niitä voi myös luoda itse lisää, ja direktiivejä käytetäänkin melkein jokaisessa Angular-projektissa. (21.)

Usein ohjelmistossa on tarpeen näyttää tiettyä dataa ehdollisesti. Esimerkiksi jos käyttäjä on painanut rekisteröitymispainiketta, halutaan käyttäjälle näyttää rekisteröitymislomake, mutta muuten kirjautumislomake. Tätä varten Angularissa on direktiivi *ngIf. Se on rakenteellinen direktiivi, ja sen tunnistaa tähtimerkistä itse direktiivin nimen edessä.

Tähtimerkki on Angularille vain lisätietoa direktiivistä, ja itse direktiivin nimi on ngIf, mutta on tärkeää muistaa, että tähtimerkki on pakollinen, muuten direktiivi ei toimi. ngIf-direktiiville voi antaa lainausmerkkien väliin ehdon, joka palauttaa totuusarvon. Tähän toimii esimerkiksi viittaus TypeScript-totuusarvomuuttujaan tai funktioon, joka palauttaa vastauksena totuusarvon. On myös tärkeää huomata, että tämä direktiivi ei vain piilota elementtiä DOM:sta vaan se poistaa tai lisää elementin DOM:iin (26). Tämän takia elementit, jotka on *ngIf-direktiivillä poistettu DOM:sta, ovat myös muuten TypeScriptin tavoittamattomissa eikä niitä voida kutsua. (21.)

Toinen todella usein käytetty rakenteellinen direktiivi on *ngFor. Tämä direktiivi mahdollistaa haluttujen elementtien toistamisen sivupohjassa (26), kuten kuvassa 6, jossa TypeScript-lista nimeltä colors iteroidaan sivupohjassa ja jokainen listan sisältämä elementti tulostetaan sivulle.

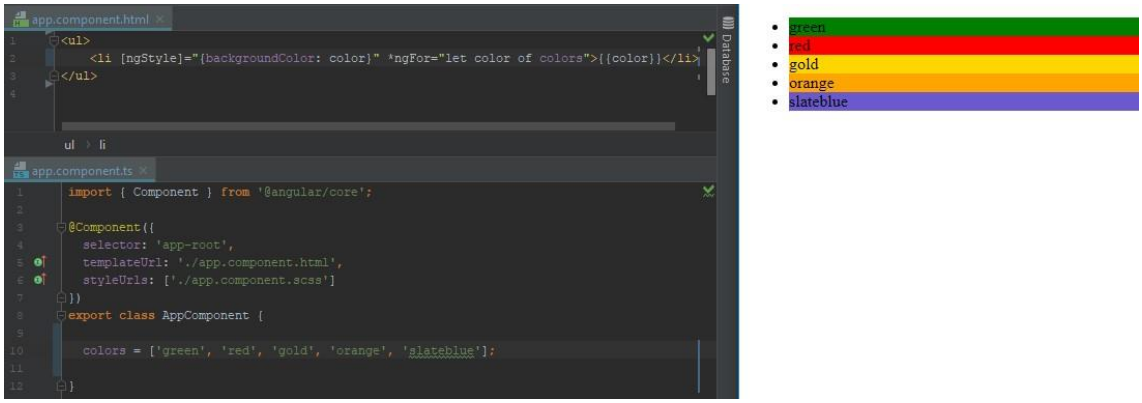


Kuva 6. ngFor-direktiivi Angular-ohjelmistossa.

Toinen sisäänrakennettu direktiivityyppi Angularissa on ominaisuudellinen direktiivi. Nämä saavat nimensä siitä, että ne näyttävät melkein kuin HTML-ominaisuuksilta, mutta näissä direktiiveissä ei ole tähtimerkkiä, eivätkä ne lisää tai poista elementtejä sivupohjasta vaan vain muokkaavat niitä. (21.)

Tällainen direktiivi on esimerkiksi ngStyle. ngStyle on sisäänrakennettu direktiivi. Itsessään ngStyle-direktiivi ei tee mitään, vaan se vaatii konfigurointia toimiakseen. Tämä konfigurointi tehdään direktiivin lainausmerkkien sisällä, ja siihen käytetään apuna

ominaisuuden sidontaa. Toisin kuin rakenteellisia direktiivejä, ominaisuudellisia direktiivejä voi olla useampia yhdessä elementissä. Myös rakenteellinen ja ominaisuudelliset direktiivit voivat olla samassa elementissä, kuten kuvassa 7.



Kuva 7. ngStyle-direktiivi, joka antaa li-elementille taustavärin.

Toisin kuin kuvassa 7, ngStyle-direktiivi voidaan myös sitoa funktioon, joka palauttaa sopivan arvon. Tässä on hyvä huomata, että itse hakasulkeet eivät kuulu direktiiviin. Hakasulkeet aiheuttavat sen, että ei käskytetä suoraan direktiiviä vaan sidotaan suoraan direktiivin ngStyle-ominaisuuteen, joka on samanniminen eli ngStyle. ngStyle odottaa JavaScript-objektia, joka pitää sisällään avain-arvopareja, joissa avain on CSS-tyylin nimi ja arvo haluttu arvo.

Samanlainen kuin ngStyle on ngClass. Ainoa ero ngClass-direktiivissä on se, että kun ngStyle nimensä mukaisesti antaa elementille tyylejä, antaa ngClass luokkia. Seuraavassa on esimerkki ngClass-direktiivin käytöstä tilanteessa, jossa totuusarvo muuttaa painikkeen active luokkaa:

```
<button [ngClass]="'active': true">Painike</button>
```

4.5 Palvelut ja riippuvuusinjektiot

Palvelut ovat Angularissa todella kätevä tapa jakaa tietoa komponenttien välillä sekä vähentää koodin toistamista (27). Samalla usein palvelut toimivat Angular-ohjelmistossa paikkana, johon voidaan hyvin tallentaa tietoa ohjelman suorituksen ajaksi. Palvelut siis auttavat komponentteja jakamaan dataa ja keskittämään koodia niihin. Aikaisemmin

tässä työssä tutustuttiin datakytkentään ja siihen, miten se mahdollistaa komponenttien välisen keskustelun. Datakytkentä voi kuitenkin isommissa ohjelmistoissa nopeasti mennä todella sekavaksi, ja palvelut myös vastaavat tähän ongelmaan. (21.)

Palvelu voidaan luoda niin manuaalisesti kuin myös automaattisesti. Manuaalisesti luotaessa palvelulle täytyy ensin etsiä jokin hyvä paikka. Se voi olla esimerkiksi suoraan app-kansioon tai app-kansioon voi luoda palveluille kokonaan oman kansion. Palvelua ei tuoda koodiin normaalisti import- ja new-avainsanojen avulla vaan Angular antaa tähän paljon paremman tavan, jota kutsutaan riippuvuusinjektioksi. (21.)

Riippuvuudella Angular-ohjelmistossa tarkoitetaan sitä, että jokin TypeScript-luokka on riippuvainen toisesta TypeScript-luokasta eli vaatii sen toimiakseen. Esimerkiksi kirjautumiskomponentti voisi olla riippuvainen käyttäjäpalvelusta. Riippuvuusinjektori taas yksinkertaisesti injektoi riippuvuuden haluttuun paikkaan automaattisesti. Edellisessä esimerkissä käytetty käyttäjäpalvelu siis injektoitaisiin kirjautumiskomponenttiin. Pelkästään tämän palvelun luominen ei kuitenkaan riitä, vaan Angularille täytyy myös kertoa, miten palvelu voidaan tarjota sitä tarvitseville paikoille. Tämä voidaan helposti tehdä joko lisäämällä palvelu providers-listaan joko komponenttitasolla tai moduulitasolla esimerkiksi AppModulessa. Angular 6:ssa ja sitä myöhemmissä versioissa Angularia voidaan ohjeistaa palvelun luomisessa myös itse palvelussa seuraavalla tavalla:

```
@Injectable({providedIn: 'root'})
```

(21; 27).

Jotta Angular tietäisi, mitkä komponentit tarvitsevat tietyn palvelun, se täytyy kertoa ohjelmistolle luokan rakentajassa eli constructorissa. Tämä tapahtuu helposti seuraavan koodiesimerkin mukaan:

```
constructor(private userService: UserService);
```

(27).

Angularin riippuvuusinjektori on hierarkkinen injektori. Tämä tarkoittaa sitä, että kun palvelu tarjotaan jossain komponentissa, kaikki tämän komponentin lapsikomponentit ja

niiden lapsikomponentit saavat saman instanssin palvelusta, ellei palvelua ole uudelleen lisätty lapsikomponentissa providers-listaan. Tämä tarkoittaa, että jos palvelu lisätään AppModulessa providers-listaan, on se lisätty korkeimmalla mahdollisella tasolla ja tällöin kaikki ohjelmiston muut komponentit saavat saman instanssin palvelusta. Alin mahdollinen taso palvelun tarjoamiselle on komponentti, jolla ei ole lapsikomponentteja. Tällöin kyseinen komponentti saa täysin oman instanssin palvelusta. On myös hyvä huomata, että jos palvelu tarjotaan Angular 6 -version mukanaan tuoman tavan mukaisesti, silloin palvelu tarjotaan ylimmällä mahdollisella tasolla. Tämä mahdollistaa palveluiden käytön joko siten, että kaikilla ohjelmiston osilla on käytössä sama instanssi ja näin kaikki sama tieto, mitä palveluun on tallennettu, tai toinen mahdollisuus on se, että eri ohjelmiston osilla on käytössä omat instanssit palvelusta ja näin ollen näillä ohjelmiston osilla on tieto tallennettuna omaan palvelun instanssiin eikä muilla osilla, joilla on oma instanssi, ole mitään pääsyä käsiksi instanssin tietoihin. (21.)

Palvelu voidaan myös luoda helposti Angular CLI:n avulla alla olevalla komennolla. Komennossa <palvelu-nimi> on korvattu sillä nimellä, joka palvelulle halutaan antaa.

```
ng g s services/<palvelu-nimi>
```

Kuvassa 8 on palvelu, jonka automaattinen palvelun luominen Angularissa luo.

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

@Injectable({
  providedIn: 'root'
})
export class PalveluService {

  constructor() { }
}
```

Kuva 8. Palvelu, jonka automaattinen palvelun luominen luo.

4.6 Reititys

Vaikka Angular on yksisivuinen ohjelmisto, on yleensä haluttua, että käyttäjälle sivuston toiminta näkyy niin sanotusti tavallisena sivustona, ja käyttäjästä tuntuu, että hän

vaihtaisi sivujen välillä, vaikka näin ei todellisuudessa olisikaan. Angular tietenkin mahdollistaa tällaisen toiminnan, jossa sivu ei päivity mutta osoite osoitepalkissa päivittyy ja samalla JavaScript muuttaa sivuston DOM:a usein paljonkin. Tässä luvussa käyn läpi Angularin reitityksen toimintaa ja esittelen muutamia koodiesimerkkejä.

Angularin reittien määrittäminen ohjelmistoon alkaa itselläni aina ensin itse reittien luomisella. Pienemmissä ohjelmistoissa reitit voi hyvin määritellä AppModulessa, mutta suuremmissa ohjelmistoissa, joissa on paljon eri reittejä tai reitityslogiikka on monimutkaisempi, ne on hyvä jakaa omaan tiedostoonsa tai tarvittaessa jopa useampaan omaan tiedostoon. Tässä luvussa tutustutaan kuitenkin vain Angularin perusreititysominaisuuksiin ja esimerkeissä reitit on määritelty AppModulessa.

Reitit määritellään listana, joka on tyyppiä Routes, joka pitää sisällään JavaScript-objekteja, jotka ovat ennalta määrätyssä muodossa (28). Jokainen reitti tarvitsee parametrin path, joka kertoo Angularille, mitä selaimen osoitepalkissa pitäisi olla, jotta tämä reitti aktivoituisi. Tämän lisäksi reitille voidaan antaa component-parametri, joka kertoo Angularille, että kun tämä reitti aktivoidaan, tähän annettu komponentti pitää ladata käyttäjälle näkyville. (21.)

Tämä ei kuitenkaan pelkästään riitä, vaan Angular-ohjelmistolle pitää myös kertoa näiden reittien olemassaolosta. Tämä tehdään esimerkeissäni AppModulessa sen imports-listassa kuten kuvassa 9. Tähän listaan lisätään RouterModule, joka mahdollistaa reitityksen, ja määritetyt reitit annetaan RouterModulelle sen omalla forRoot-metodilla, joka rekisteröi reitit ohjelmistolle. (21.)

Nyt Angular tietää reitit ja mitä sen pitää näyttää käyttäjälle, kun jokin reitti aktivoituu. Kuitenkaan itse näkymä käyttäjälle ei vielä muutu, koska ohjelmistosta puuttuu tieto, missä kohdassa Angularin pitäisi haluttu komponentti näyttää. Angularin mukana tulee sisäänrakennettu direktiivi router-outlet. Sen avulla voidaan ohjelmistossa määritellä kohta, johon reitin määrittelemä komponentti tulostetaan. (21; 28.)

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {RouterModule, Routes} from '@angular/router';
import { KotiComponent } from './koti/koti.component';
import { KayttajaComponent } from './kayttaja/kayttaja.component';
import { KayttajatComponent } from './kayttajat/kayttajat.component';

const routes: Routes = [
  {path: '', component: KotiComponent},
  {path: 'kayttajat', component: KayttajatComponent},
  {path: 'kayttaja/:id', component: KayttajaComponent},
  {path: '**', pathMatch: 'full', redirectTo: ''}
];

@NgModule({
  declarations: [
    AppComponent,
    KotiComponent,
    KayttajaComponent,
    KayttajatComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    RouterModule.forRoot(routes),
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.html

```
<router-outlet></router-outlet>
```

Kuva 9. Reititys, jossa vasemmalla AppModule, jossa määritelty reitit ja lisätty RouterModule reittien kanssa imports-listaan. Oikealla router-outlet-direktiivi, johon aktiivisen reitin komponentti tulostetaan.

Reittien välillä pitää pystyä ohjelmistossa liikkumaan, ja tähän Angularin mukana tulee direktiivi routerLink (28). RouterLink-direktiivi kertoo Angularille, että se elementti, johon direktiivi on liitetty, toimii linkkinä. Tällöin Angular osaa käsitellä tämän routerLink-elementin klikkauksen oikealla tavalla ja ohjata halutulle sivulle ilman, että sivusta latautuu uudelleen. RouterLink-direktiivi hyväksyy suoraan syötetyn tekstiarvon, mutta routerLinkin kanssa voidaan myös käyttää ominaisuudensidontaa. Ominaisuudensidontaa käyttämällä voidaan routerLinkille antaa esimerkiksi funktio, joka palauttaa halutun reitin, mutta myös lista, joka antaa enemmän hallintamahdollisuuksia reitin toimintaan ja siihen, miten Angular reitin käsittelee ja mahdollistaa monimutkaisempien reittien hallinnan helposti. Tässä opinnäytetyössä ja luvussa keskityn kuitenkin vain suoraan tekstinä annettaviin reitteihin ja reitityksiin.

RouterLink-direktiivillä voidaan ohjata käyttäjä uudelle sivulle käyttäjän niin halutessa, mutta joskus myös tarvitaan reititystä, kun esimerkiksi jokin asia koodissa on saatu päätökseen. Hyvä esimerkki tällaisesta tapauksesta on kirjautuminen, jossa sen jälkeen, kun käyttäjä on syöttänyt kirjautumistietonsa, täytyy koodin käydä varmistamassa, että syötetyt tiedot ovat oikein. Kun tällöin saadaan vastaus, että tiedot ovat oikein, täytyy

käyttäjä useimmiten ohjata jollekin toiselle sivuille. Tämä voidaan toteuttaa Angularissa helposti TypeScriptin puolella. Se tehdään injektoimalla Angularin reititin, ja tämän jälkeen reitittimen avulla voidaan ohjata käyttäjä uuteen reittiin reitittimen navigate-funktion avulla (21). Navigaatio ottaa parametriksensa listan, jonka ensimmäinen arvo on itse reitti. Tästä esimerkki on kuvassa 10.

```
constructor(private router: Router) { }

ngOnInit() {
}

navigoiKayttajatSivulle() {
  this.router.navigate( commands: ['/kayttajat']);
}
```

Kuva 10. Funktio, joka ohjaa käyttäjän /kayttajat-reittiin.

Viimeinen asia Angular-ohjelmistokehyksen reitityksestä, jonka käyn läpi tässä opinnäytetyössä, on parametrien antaminen reitille ja niiden hakeminen TypeScriptissä. Itselläni usein Angular-ohjelmistossa tulee tarve hakea tietoa rajapinnalta sen mukaan, mitä käyttäjä tekee ohjelmistossa. Hyvä esimerkki tällaisesta toiminnasta on se, että käyttäjä haluaa nähdä toisen käyttäjän tietoja. Tällöin omissa ohjelmistoissani minulla on reitti, joka ottaa parametrina halutun käyttäjän yksilöivän numeron eli ID:n. Tämä reitti ohjaa käyttäjän käyttäjäkomponenttiin ja hakee tarvittavan tiedon käyttäjästä reitille annetun ID:n perusteella. Tällaisesta reitistä on esimerkki kuvassa 9. Parametrit annetaan reitille yksinkertaisesti lisäämällä halutun parametrinimen eteen reitissä kaksoispiste, kuten on tehty kuvassa 9. Parametrin nimi voi olla mikä tahansa, ja tällä nimellä parametri voidaan myöhemmin ohjelmistossa hakea käyttöön, kuten kuvassa 11.

Reitin parametriin päästään käsiksi injektoimalla aktivoitu reitti. Tämä antaa pääsyn ladattuun reittiin, joka on JavaScript-objekti ja sisältää paljon metatietoa reitistä, kuten parametrit, jotka reitille on annettu. Kuvassa 11 haetaan ja tulostetaan konsoliin käyttäjä sen perusteella, mikä ID reitille kayttajat/:id on annettu.

```
constructor(
  private route: ActivatedRoute,
  private kayttajaService: KayttajaService
) { }

ngOnInit() {
  const id = this.route.snapshot.params.id;
  console.log(this.kayttajaService.haeKayttaja(id));
}
```

Kuva 11. Komponentti, joka ladattaessa hakee käyttäjän reitille annetun id-parametrin avulla.

5 Ohjelmistokehityksen nopeutuminen Angular-ohjelmistokehystä käyttämällä

Tässä luvussa käydään läpi, kuinka Angular-ohjelmistokehyksellä voidaan luoda web-ohjelmisto, joka näyttää pysäköintipaikat Helsingissä ja niistä lisätietoa. Samalla on tarkoitus tarkkailla, kuinka Angular-ohjelmistokehys nopeuttaa ohjelmiston luomista, ja lopuksi vielä tutkia tarkemmin, kuinka Angularin käyttö vaikutti projektin nopeuteen. Ohjelmiston tarkoituksena olisi tarjota kartta, josta ohjelmiston käyttäjät voisivat helposti nähdä heitä lähellä olevat pysäköintipaikat. Toteutettavan projektin nimi tässä opinnäytetyössä on Parkkipaikat. Parkkipaikat-sovellus saa tietonsa Helsingin kaupungin avoimesta rajapinnasta Parkkihubi Public API. Rajapinta ei ole täysin tarkka, sillä se ei pysty palauttamaan tietoa asukaspysäköinnistä, jolloin pysäköintipaikka, jolla vaikuttaisi rajapinnan mukaan olevan tilaa, onkin todellisuudessa täynnä (16). Ohjelmistossa on myös käytetty Bootstrap-kirjastoa, jolla saadaan helposti kehitettyä hyviä sivuston rakenteita ja jossa on paljon valmiina tyylejä eri HTML-elementeille.

Parkkipaikat koostuu kolmesta erillisestä sivusta. Ensimmäinen on etusivu, jonka käyttäjät näkevät, kun he tulevat sivustolle. Tämän sivun tarkoitus on vain toivottaa käyttäjä tervetulleeksi sivustolle ja ohjata hänet Parkkipaikat-sivulle. Parkkipaikat-sivu koostuu kartasta, jossa näytetään sata pysäköintipaikkaa käyttäjälle kartalla. Lisäksi sivulla on painikkeet, joiden avulla käyttäjä voi ladata seuraavat tai edelliset sata pysäköintipaikkaa kartalle. Viimeinen sivu on pysäköintipaikkasivu, joka näyttää tarkempaa tietoa käyttäjän valitsemasta pysäköintipaikasta.

5.1 Parkkipaikat-Angular-projektin luominen

Parkkipaikat-ohjelmiston tekeminen aloitettiin luomalla uusi Angular-projekti Angular CLI:n avulla suorittamalla seuraava komento:

```
ng new parkkipaikat --style=”scss” --routing
```

Tämä komento luo kuvan 1 mukaisen tiedostorakenteen. Style-lippu kertoo Angular CLI:lle, että tässä projektissa halutaan käyttää SCSS-esiprosessoria, ja routing-lippu kertoo, että projektissa tullaan käyttämään reititystä.

Tämän jälkeen projektiin asennettiin Bootstrap komentorivin avulla, jolla suoritettiin seuraava komento:

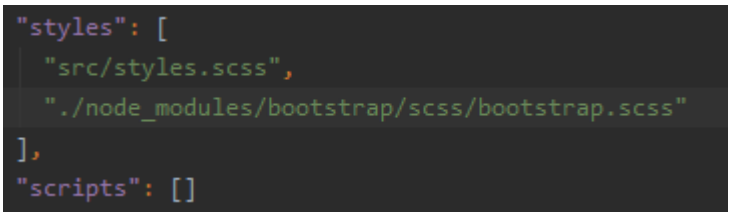
```
npm install --save bootstrap@latest @ng-bootstrap/ng-bootstrap@latest
```

Tämä komento asensi projektiin kaksi valmista kirjastoa, joista toinen on Bootstrap, jonka avulla saadaan Bootstrapin tyylit projektiin, ja toinen on ng-bootstrap, joka on kirjasto, jossa Bootstrapin komponentit on kirjoitettu paremmin Angulariin sopiviksi. Tämän jälkeen ng-bootstrap täytyy tuoda projektiin lisäämällä se app.module.ts-tiedostoon kuvan 12 mukaisesti.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    NgbModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Kuva 12. ng-bootstrap-kirjaston lisääminen app.module.ts-tiedostoon.

Tämän jälkeen vielä angular.json-tiedostoon lisätään polku Bootstrapin tyylitiedostoon, kuten kuvassa 13, jolloin ohjelmistossa saadaan Bootstrapin tyylit käyttöön.



Kuva 13. Bootstrap-tyylitiedoston polku angular.json-tiedostossa.

Tämän jälkeen vielä app.component.html-tiedostosta korvataan kaikki Angularin sinne itse lisäämä koodi router-outlet-direktiivillä. Tämän jälkeen app.component.ts-tiedostossa ei pitäisi olla mitään muuta kuin router-outlet-direktiivi, kuten seuraavassa koodiesimerkissä:

```
<router-outlet></router-outlet>
```

Tämän jälkeen kaikki tarvittava esiasennus on tehty Parkkipaikat-ohjelmistolle ja projektin kehityspalvelin voidaan käynnistää suorittamalla komentorivillä ng-serve-komento.

Omasta mielestäni jo tässä vaiheessa Angular-ohjelmistokehyksen hyödyt alkavat tulla hyvin esiin. Sen sijasta, että jokainen tiedosto pitäisi luoda itse käsin, tekee Angular tämän hetkessä itse. Samoin SCSS-esiprosessorin käyttäminen on Angularin avulla todella helppoa, kun Angular itse konfiguroi sen valmiiksi. Paljon enemmän aikaa kuluisi, jos SCSS pitäisi asentaa erikseen koneelle.

5.2 Ohjelmiston komponentit ja reititys

Parkkipaikat-ohjelmisto koostuu neljästä eri näkymästä. Jokaista näkymää varten luodaan oma komponentti, ja nämä näkymät ovat etusivu, parkkipaikat, parkkipaikka ja yläpalkki. Yläpalkki on ainoa komponentti, joka näkyy käyttäjälle joka sivulla. Muut komponentit ovat omia näkymiään, joiden välillä siirrytään Angularin reitityksen avulla. Kaikki komponentit luodaan alla olevalla komennolla, jossa <komponentin-nimi> on korvattu luotavan komponentin nimellä. Tässä projektissa käytin edellä mainitsemiani näkymien

nimiä. Vain yläpalkkikomponentin nimen vaihdoin sitä luotaessa ylapalkki-nimeen välttääkseni ä-kirjainta nimessä.

```
ng g c <komponentin-nimi>
```

Kun näkymät oli luotu, seuraavaksi konfiguroitiin reititys näiden komponenttien välillä. Koska tämä projekti on niin pieni ja reititys on todella yksinkertainen ja pieni, reititys toteutettiin app-routing.module.ts-tiedostossa. Kuvan 14 mukaisesti eri reittejä tuli neljä. Kuvan ensimmäinen reitti ohjaa tyhjän polun etusivulle. Parkkipaikat-polku taas ohjaa parkkipaikat-sivulle ja parkkipaikka-reitti ohjaa parkkipaikka-sivulle. Parkkipaikka-reitissä on myös parametri ID, jonka avulla parkkipaikka-sivulla haetaan oikean pysäköintipaikan tiedot. Viimeinen reitti on villikortti-reitti. Tämä reitti ohjaa kaikki polut, jotka eivät täsmänneet mihinkään edelliseen reittiin etusivulle. Esimerkiksi jos käyttäjä hakisi polkua parkkihallit, ohjaantuisi tämä etusivulle, koska ohjelmistossa ei ole omaa reittiä parkkihallit-polulle.

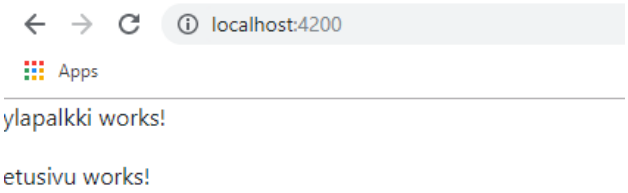
```
const routes: Routes = [
  { path: '', component: EtusivuComponent },
  { path: 'parkkipaikat', component: ParkkipaikatComponent },
  { path: 'parkkipaikka/:id', component: ParkkipaikkaComponent },
  { path: '**', pathMatch: 'full', redirectTo: '' }
];
```

Kuva 14. Parkkipaikat-ohjelmiston reitit.

Tässä vaiheessa, kun yläpalkki komponentti oli luotu, se lisättiin app.component.html-tiedostoon router-outlet-direktiivin yläpuolelle, jotta tämä komponentti näkyy jokaisella sivulla:

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

Tämän jälkeen ohjelmiston runko oli koossa ja selaimessa menemällä osoitteeseen localhost:4200 nähtiin ohjelmisto toiminnassa, kuten kuvassa 15.



Kuva 15. Parkkipaikat-ohjelmiston etusivu komponenttien ja reitityksen luonnin jälkeen.

5.3 Tiedon hakeminen palveluiden avulla rajapinnasta

Parkkipaikat-ohjelmisto hakee käyttämänsä tiedon Helsingin kaupungin avoimesta rajapinnasta nimeltä Parkkihubi Public API. Rajapintaan tehdään HTTP-kutsuja, jotka palauttavat vastauksena tietoa pysäköintipaikoista. Koska parkkipaikat ja parkkipaikkakomponentit kummatkin tarvitsevat tietoa rajapinnalta, tehtiin kaikki rajapintaa kutsuvat funktiot palveluun. Näin kaikki rajapintaa kutsuvat funktiot ovat samassa tiedostossa, mikä tekee niiden hallitsemisesta helpompaa ja silloin samaa koodia tarvitse kirjoittaa moneen paikkaan, kun komponentit voivat hyödyntää tätä palvelua. Kuvassa 16 oleva palvelu, jolle tässä projektissa annettiin nimeksi api, luotiin services-kansioon seuraavalla komennolla:

```
ng g s services/api
```

Rajapintakutsut tehdään HTTP-kutsuilla, ja tämä vaatii Angular-projektin pientä konfiguroimista, jotta Angular-ohjelmisto pystyy näitä kutsuja tekemään. app.module.ts-tiedoston imports-listaan lisättiin HttpClientModule, joka mahdollistaa näiden rajapintakutsujen toiminnan.

Ensiksi palveluun luotiin muuttuja, johon rajapinnan osoite tallennettiin. Tämän jälkeen HttpClient injektoitiin palvelun rakentajassa. Luotu palvelu käsittää kolme funktiota, joiden avulla tietoa rajapinnasta voidaan hakea. Funktiot palauttavat vastauksena HTTP-pyynnön, jonka avulla voidaan myöhemmin käsitellä komponentissa se pyyntö, joka funktiota kutsui.

Ensimmäinen funktio haeParkkipaikat hakee sadan pysäköintipaikan tiedot. Parkkihubi-rajapintaa voidaan ajatella kirjana, jonka yhdellä sivulla on aina tiedot sadasta pysäköintipaikasta. Rajapinnalle voidaankin lähettää parametrina sen sivun numero, jolta

halutaan pysäköintipaikkojen tiedot. Palvelun haeParkkipaikat funktio hyväksyy parametrina numeron, jonka avulla rajapintakutsulle voidaan kertoa, monenneltako sivulta tiedot halutaan. Parametri on kuitenkin vapaaehtoinen, ja jos sitä ei anneta funktion kutsulle, palauttaa rajapinta ensimmäiset sata pysäköintipaikkaa. Osoite, johon funktio kutsun tekee, on seuraavan esimerkin lainen, jossa <sivunumero> on muutettu haluttavan sivun numeroksi:

```
https://pubapi.parkkiopas.fi/public/v1/parking_area/?page=<sivunumero>
```

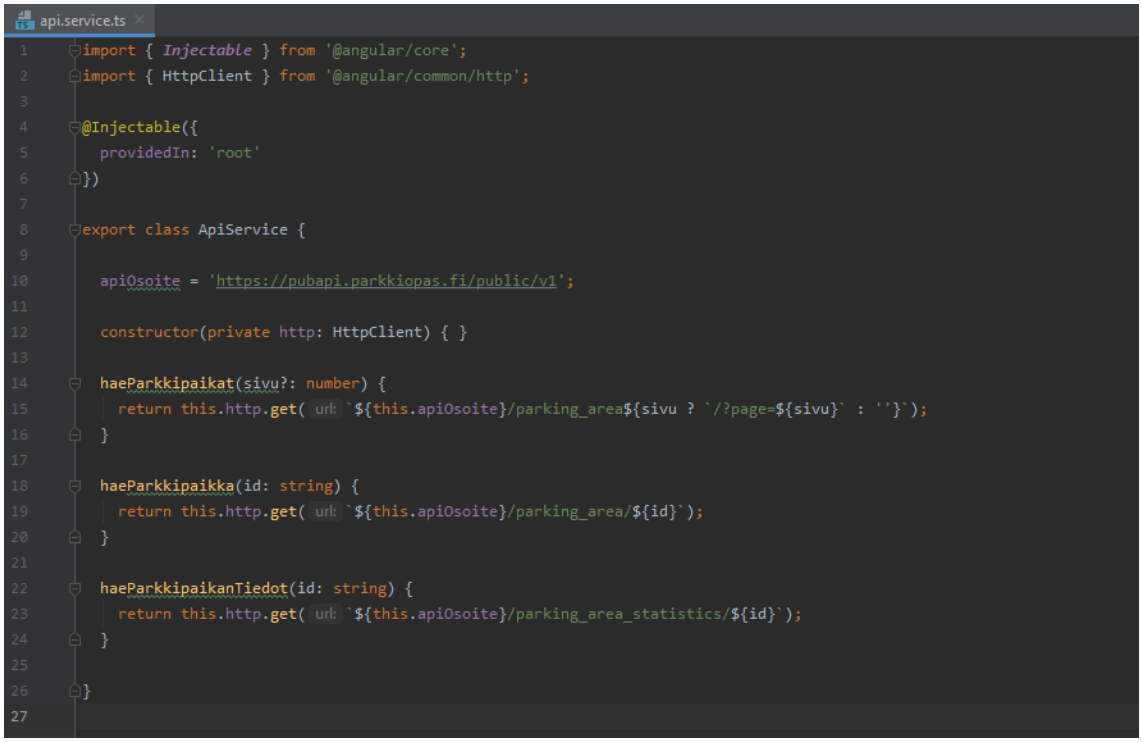
Toinen funktio haeParkkipaikka puolestaan hakee yhden parkkipaikan tiedot. Funktio ottaa parametrina parkkipaikan ID:n. Viimeinen funktio haeParkkipaikanTiedot on samanlainen kuin haeParkkipaikka-funktio. Ainoa ero on, että rajapintakutsu tehdään eri osoitteeseen. haeParkkipaikanTiedot-kutsu eroaa haeParkkipaikka-kutsusta siten että haeParkkipaikanTiedot-kutsu palauttaa mukanaan tiedon, kuinka monta ajoneuvoa pysäköintipaikalla on tällä hetkellä pysäköitynä.

haeParkkipaikka-funktio tekee kutsun seuraavaan osoitteeseen, jossa <id> on korvattu haetun pysäköintipaikan ID:llä:

```
https://pubapi.parkkiopas.fi/public/v1/parking_area/<id>
```

haeParkkipaikanTiedot tekee kutsun seuraavaan osoitteeseen, jossa <id> on korvattu haetun pysäköintipaikan ID:llä:

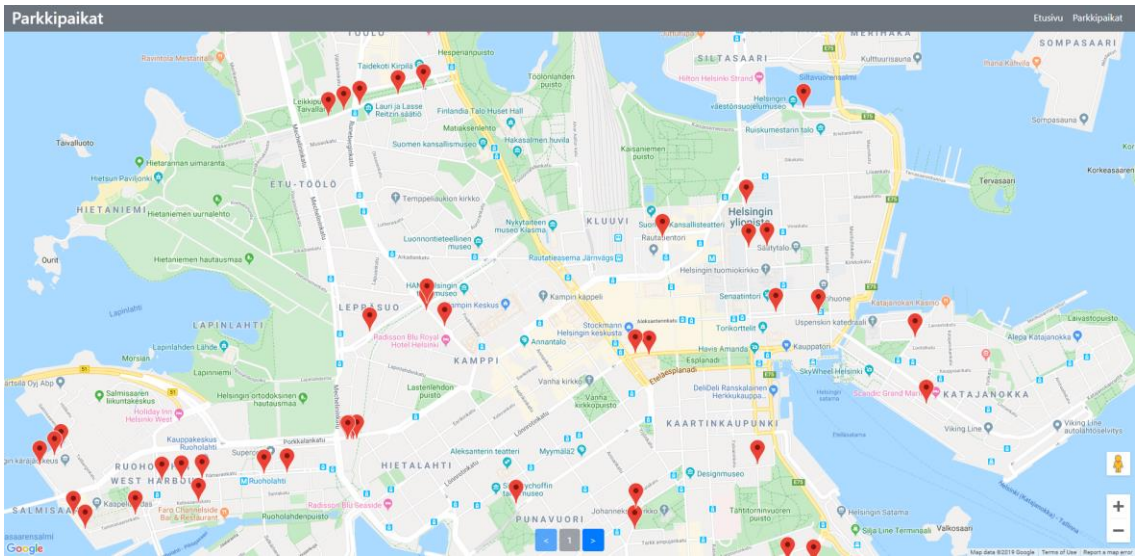
```
https://pubapi.parkkiopas.fi/public/v1/parking_area_statistics/<id>
```



Kuva 16. Api-palvelu, jonka avulla voidaan tehdä rajapintakutsuja.

5.4 Näkymien rakentaminen

Parkkipaikat-ohjelmistossa on kolme suurempaa näkymää ja yksi pienempi näkymä, joka on yläpalkki. Näkymät toteutettiin HTML:n, SCSS:n ja TypeScriptin avulla. Suurin ja tärkein näkymä ohjelmistossa on kuvassa 17 oleva parkkipaikat-näkymä, ja käyn tämän näkymän rakentamisen läpi. Muita näkymiä en tässä kohtaa käy läpi, sillä niiden rakentaminen oli hyvin samanlaista kuin parkkipaikat-näkymän rakentaminen.



Kuva 17. Parkkipaikat-näkymä.

Parkkipaikat-näkymä koostuu suuresta kartasta ja kartan alalaidassa olevista painikkeista, joiden avulla käyttäjä voi vaihtaa sivua ja näin ladata aina seuraavat tai edelliset sata pysäköintipaikkaa kartalle. Kartta toteutettiin Angular Google Maps (AGM) -kirjaston avulla. AGM asenettiin projektiin komentoriviltä suorittamalla seuraava komento:

```
npm install --save @agm/core
```

Tämän jälkeen app.module.ts-tiedostossa AgmCoreModule lisättiin imports-listaan. Jotta AGM pystyy käyttämään Googlen karttaa, vaati AGM Google Maps API -avaimen. Tämä API-avain annetaan imports-listassa olevalle AgmCodeModulelle kuvan 18 mukaisella tavalla.

```
imports: [
  BrowserModule,
  HttpClientModule,
  AppRoutingModule,
  AgmCoreModule.forRoot( lazyMapsAPILoaderConfig: {
    apiKey: 'google-maps-api-avain'
  }),
  NgbModule
```
Kuva 18. AgmCodeModulen lisääminen app.module.ts-tiedostoon.

Tämän jälkeen kuvassa 19 oleva parkkipaikat.component.ts-tiedosto voitiin rakentaa. Tämä tiedosto vastaa kaikesta tiedosta, mitä komponentti tarvitsee, ja on vastuussa myös näkymässä tapahtuvista toiminnoista. Tiedostossa on neljä muuttujaa:

- zoom

  Zoom-muuttuja vastaa, kuinka lähelle kartta on zoomattu, kun näkymä latautuu.

- koordinaatit

  Koordinaatit-muuttuja vastaa siitä, mihin kohtaan kartta keskittyy, kun näkymä latautuu. Koordinaateiksi on asetettu Helsingin keskusta.

- parkkipaikat

  Parkkipaikat-muuttuja on lista, johon rajapinnan vastauksen mukana tulleet pysäköintipaikat tallennetaan. Näkymä piirtää kartan merkit tämän listan perusteella.

- sivu

  Sivu-muuttujaan tallennetaan sen sivun numero, jota käyttäjä juuri sillä hetkellä katselee. Tämän muuttujan avulla myös rajapinnalle tehdään haku ja kerrotaan, miltä sivulta seuraavaksi pysäköintipaikat halutaan hakea. Muuttuja on asetettu aluksi nollaksi, jolloin kun näkymä latautuu, haetaan siihen ensimmäiset sata pysäköintipaikkaa.

Tiedostossa on myös neljä funktiota:

- ngOnInit

  ngOnInit on funktio, joka ajetaan aina, kun näkymä latautuu. Tässä funktiossa kutsutaan haeParkkipaikat-funktiota.

- avaaParkkipaikka

  Tämä funktio on vastuussa käyttäjän ohjaamisesta parkkipaikka-sivulle, kun käyttäjä on klikannut pysäköintipaikan merkkiä kartalla. Funktio käyttää apuna Angularin reititintä, jonka avulla käyttäjä ohjataan parkkipaikka-sivulle valitun pysäköintipaikan ID:n kanssa.

- sivunVaihto

  Funktion tarkoituksena on muuttaa sivu-muuttujaa käyttäjän painalluksen perusteella. Sivu-muuttujaan siis lisätään tai siitä vähennetään yksi riippuen siitä, haluaako käyttäjä mennä sivuissa eteenpäin vai taaksepäin. Tämän jälkeen funktio vielä kutsuu haeParkkipaikat-funktiota.

- haeParkkipaikat

  Tämä funktio kutsuu api-palvelun haeParkkipaikat-funktiota ja jää odottamaan vastausta rajapinnalta. Kun vastaus rajapinnalta saapuu, käsittelee funktio tämän vastauksen. Vastaus pitää sisällään features-listan, jonka funktio tallentaa parkkipaikat-muuttujaan. Tämän jälkeen, kun tieto on tallennettu parkkipaikat-muuttujaan, voidaan pysäköintipaikat näyttää käyttäjälle näkymässä. HaeParkkipaikat-funktio kutsuessaan api-palvelun haeParkkipaikat-funktiota antaa funktiolle parametrina sivu-muuttujan. Tämän avulla api-palvelun haeParkkipaikat-funktio voi pyytää oikean sivun rajapinnalta.

parkkipaikat.component.ts

```typescript
import { Component, OnInit } from '@angular/core';
import {ApiService} from '../services/api.service';
import {Router} from '@angular/router';

@Component({
  selector: 'app-parkkipaikat',
  templateUrl: './parkkipaikat.component.html',
  styleUrls: ['./parkkipaikat.component.scss']
})
export class ParkkipaikatComponent implements OnInit {

  zoom = 15;
  koordinaatit = { latitude: 60.169843, longitude: 24.937905 };
  parkkipaikat = [];
  sivu = 0;

  constructor(private apiService: ApiService, private router: Router) { }

  ngOnInit() {
    this.haeParkkipaikat();
  }

  avaaParkkipaikka(id) {
    this.router.navigate( commands: [`/parkkipaikka/${id}`]);
  }

  sivunVaihto(muutos) {
    this.sivu += muutos;
    this.haeParkkipaikat();
  }

  haeParkkipaikat() {
    this.apiService.haeParkkipaikat( sivu: this.sivu !== 0 ? this.sivu : null).subscribe( next: (data: any) => {
      this.parkkipaikat = data.features;
    });
  }

}
```

Kuva 19. Parkkipaikat-näkymän TypeScript-tiedosto, joka sisältää näkymän logiikan.

Näkymän logiikan valmistuttua voitiin seuraavaksi rakentaa näkymän HTML-koodi parkkipaikat.component.html-tiedostoon. Tiedosto on tavallista HTML-koodia Angular Google Mapsin elementtiä lukuun ottamatta. Angular Google Maps voidaan asettaa näkymään agm-map-elementillä. Tälle elementille annettiin ominaisuuden sidonnalla zoom-, latitude- ja longitude-ominaisuudet. Zoom-ominaisuuden sidonnalle annettiin TypeScript-tiedostosta zoom-muuttuja. Latitudelle annettiin TypeScript-tiedoston koordinaatit-muuttujan latitude-arvo sekä longitudelle koordinaatit-muuttujan longitude-arvo.

Agm-map-elementin sisään luotiin agm-marker-elementti, jotka ovat kartalla olevia merkkejä. ngFor-direktiivillä karttamerkkejä piirretään kartalle niin monta kuin TypeScript-tiedoston parkkipaikat-listassa on pysäköintipaikkoja. Myös agm-marker-elementille annettaan ominaisuuden sidonnalla latitude ja longitude, jotka saadaan pysäköintipaikan

tiedoista. Lisäksi agm-marker-elementille annettiin vielä markerClick-tapahtumakytkentä, jonka avulla käyttäjä ohjataan parkkipaikka-sivulle, kun hän klikkaa pysäköintipaikan merkkiä kartalla.

Kuvassa 20 parkkipaikat-näkymän HTML-tiedosto, jossa agm-map elementti ja markerClick-tapahtumakytkentä.

parkkipaikat.component.html

```
<div class="container-fluid">
  <div class="row">
    <div class="col-12 px-0">
      <agm-map [zoom]="zoom" [latitude]="koordinaatit.latitude" [longitude]="koordinaatit.longitude">
        <agm-marker
          *ngFor="let parkkipaikka of parkkipaikat"
          [latitude]="parkkipaikka.geometry.coordinates[0][0][0][1]"
          [longitude]="parkkipaikka.geometry.coordinates[0][0][0][0]"
          (markerClick)="avaaParkkipaikka(parkkipaikka.id)">
        </agm-marker>
      </agm-map>
    </div>
  </div>
</div>
<div id="painikkeet" class="d-flex justify-content-around align-items-center">
  <button [disabled]="sivu === 0" (click)="sivunVaihto( muutos: -1)" class="btn btn-primary"><</button>
  <button class="mx-1 btn btn-secondary disabled">{{sivu + 1}}</button>
  <button (click)="sivunVaihto( muutos: 1)" class="btn btn-primary">></button>
</div>
```

Kuva 20. Parkkipaikat-näkymän HTML-koodi.

Jotta kartta näkyy käyttäjälle ja täyttää koko ruudun, täytyi parkkipaikat.component.scss-tiedostossa agm-map-elementille lisätä vielä korkeustyyli kuvan 21 mukaisesti.

parkkipaikat.component.scss

Enable File Watcher to compile SCSS to CSS?

```
1   agm-map {
2     height: calc(100vh - 50px);
3   }
4
5   #painikkeet {
6     position: absolute;
7     bottom: 10px;
8     left: 50%;
9     transform: translateX(-50%);
10  }
11
```

Kuva 21. Parkkipaikat-näkymän tyylitiedosto.

5.5 Angular-ohjelmistokehyksen vaikutus ohjelmistokehityksen nopeuteen

Mielestäni Angular-ohjelmistokehys nopeutti Parkkipaikat-sovelluksen kehittämistä huomattavasti. Parkkipaikat-sovelluksen kehittäminen ilman ohjelmistokehystä olisi omasta mielestäni vienyt paljon enemmän aikaa, kun kehitystyössä olisi pitänyt luoda kaikki tiedostot ja niiden välillä kulkevan datan logiikka tyhjästä. Angular-ohjelmistokehys helpotti paljon kehitystyötä, kun uudet sivut ja komponentit saatiin luotua helposti komentoriviltä yhden komennon avulla.

Aluksi kehittämisen nopeuden kasvamisen huomasi heti projektin alussa, kun koko projekti oli konfiguroitu ja valmiina kehitettävänä pienessä ajassa verrattuna siihen, jos kaikkien sivujen HTML-, CSS- ja JavaScript-tiedostot olisi pitänyt liittää erikseen näihin sivuihin. Myös Angular-ohjelmistokehykseen sisäänrakennettu reititys vähensi mielestäni paljon sovelluksen reitityksen kehittämiseen kuluvaa aikaa, ja varsinkin Angular-ohjelmistokehyksen koin tässä todella käteväksi villikorttireitin takia. Sen avulla käyttäjä voitiin ohjata aina kätevästi ja helposti ohjelmiston kotisivulle, jos hän yrittää hakea olematonta sivua.

Näkymien rakentaminen Angular-ohjelmistokehyksen avulla oli myös todella helppoa ja mukavaa. Angularin datakytkentä nopeutti näkymien rakentamisessa kehitystyötä valtavasti. Kun tällainen logiikan ja näkymän välinen datakytkentä on olemassa, sen vähentää

mielestäni todella paljon kehitykseen kuluvaa aikaa, kun toiminnallisuutta ei tarvitse itse rakentaa tyhjästä. Samoin näkymien rakentamisessa oli kätevää se, että yhtä komponenttia voitiin käyttää jokaisella sivulla ja näin tätä komponenttia ei tarvinnut kirjoittaa moneen paikkaan erikseen vaan kerran kirjoitettua koodia pystyttiin tehokkaasti käyttämään ympäri sivustoa.

Mielestäni Angular-ohjelmistokehys myös paransi omalla kohdallani paljon kehitystyytyväisyyttä ja luottamustani tekemäni koodin laatuun. TypeScriptin vahva tyypitys oli myös kätevä. Sen avulla pystyi helposti välttämään pieniä virheitä, jotka huomaamatta jääneinä olisivat varmasti pidentäneet kehitykseen kuluvaa aikaa. Voin hyvin sanoa, että Parkkipaikat-projektissa Angular-ohjelmistokehys nopeutti projektia huomattavasti.

6 Yhteenveto

Opinnäytetyössä tutkittiin, miten ohjelmistokehitystä voidaan nopeuttaa ja miten Angular-ohjelmistokehys voi auttaa tässä tavoitteessa.

Aluksi selvitettiin, mikä ohjelmistokehitys on ja minkälaisia tavoitteita ohjelmistokehityksen nopeuttamisella yleensä on. Tämän jälkeen tutkittiin, miten ohjelmistokehityksen nopeutta voidaan mitata ja mitä eri tapoja ohjelmistokehityksen nopeuttamiselle on niin henkilöstön kuin myös tekniikan osalta. Nopeuden mittaamiselle ja nopeuttamiselle löydettiin useita eri tapoja.

Työssä huomattiin, että henkilöstöön panostaminen tuottaa todennäköisemmin parempia tuloksia ohjelmistokehityksen nopeuttamiseen kuin tekniikkaan panostaminen. Nopeuttamisen osalta lopuksi vielä tutkittiin ohjelmistokehityksen hyötyjä yrityksissä. Huomattiin, että oikein toteutettuna ohjelmistokehityksen nopeuttaminen ei ainoastaan auta yritystä tuomaan uusia tuotteita markkinoille useammin, vaan ohjelmistokehityksen nopeuttamisella on myös positiivinen vaikutus asiakastyytyväisyyteen, työntekijöiden hyvinvointiin ja ohjelmiston laatuun. Samalla yritys, jossa panostetaan ohjelmistokehityksen nopeuttamiseen oikeilla tavoilla kuten panostamalla henkilöstön osaamiseen ja hyvinvointiin tekee yrityksestä mielenkiintoisemman paikan uusille työntekijöille.

Tämän jälkeen siirryttiin tutkimaan Angular-ohjelmistokehystä, sen perusteita, arkkitehtuuria ja ominaisuuksia. Angular-osuudessa tutkittiin yhden sivun ohjelmistoja ja uuden yhden sivun ohjelmiston luomista Angular-ohjelmistokehyksellä. Samalla myös TypeScript-ohjelmointikieleen tutustuttiin. Työssä todettiin, että Angular-ohjelmistokehys on hyvä ohjelmistokehys sen kehittäjille tarjoamien työkalujen ansiosta.

Työssä myös luotiin Parkkipaikat-niminen yhden sivun sovellus Angular-ohjelmistokehystä käyttämällä. Parkkipaikat-sovellus koostuu kolmesta eri sivusta, ja ohjelmiston tarkoitus on helpottaa vapaiden pysäköintipaikkojen löytymistä Helsingissä. Ohjelmisto sai tarvitsemansa tiedon Parkkihubi Public API -rajapinnasta. Parkkipaikat-ohjelmiston luominen aloitettiin luomalla ensin Angular-projekti, minkä jälkeen projektiin luotiin sen tarvitsemat komponentit ja reititys. Tämän jälkeen projektiin tehtiin Angular-palvelu, jonka avulla tarvittava tieto saatiin haettua rajapinnalta HTML-kutsujen avulla. Sen jälkeen projektiin rakennettiin tarvittavat näkymät. Lopuksi tutkittiin Angular-ohjelmistokehyksen sopivuutta tällaisen projektin luomiseen ja sitä, oliko Angular-ohjelmistokehys vähentänyt kehittämiseen kuluvaa aikaa ja näin nopeuttanut ohjelmistokehitystyötä. Huomattiin, että Angular-ohjelmistokehys oli vähentänyt kehittämiseen kuluvaa aikaa huomattavasti ja näin nopeuttanut ohjelmiston kehitystyötä.

Työlle asetetut tavoitteet täyttyivät omasta mielestäni erinomaisesti. Sain hyvin selvitettyä ohjelmistokehityksen nopeuttamisen tavoitteita ja sitä, minkälaisia hyötyjä tämä tuo mukanaan. Angular-osiossa sain hyvin tutustuttua Angular-ohjelmistokehykseen ja sen toimintaan. Parkkipaikat-sovellus oli kehitetty Angular-ohjelmistokehyksellä, joka mielestäni sopii todella hyvin Parkkipaikat-sovelluksien tyyppiseen kehittämiseen, ja Angular-ohjelmistokehys nopeuttaa ohjelmistokehitystä sen verran paljon, että sitä kannattaa ainakin kokeilla.

Lähteet

1 Ohjelmistokehitys. Verkkoaineisto. ite wiki Oy. < https://www.ite-wiki.fi/opas/ohjelmistokehitys/ >. Luettu 8.10.2019.

2 Taina, Juha. 2013. Vaatimusanalyysi. Luentomoniste. Helsingin yliopisto.

3 System Design in Software Development. Verkkoaineisto. Medium. <https://medium.com/the-andela-way/system-design-in-software-development-f360ce6fcbb9>. Luettu 17.11.2019.

4 Immonen, Jarkko. 2002. Johdatus ohjelmistotuotantoon. Luentomoniste. Joensuun yliopisto.

5 Pollari, Jukka. 2014. Ohjelmistotestaus. Insinöörityö. Centria ammattikorkeakoulu. Theseus-tietokanta.

6 Kahn, Kenneth. 2013. The PDMA Handbook of New Product Development. Second Edition. New Jersey: John Wiley & Sons.

7 Kakkonen, Kari. 2019. Leikkaussalissa tai julkaisemassa ohjelmistoa? – Punnitse ensin riskit. Verkkoaineisto. Tivi. <https://www.tivi.fi/blogit/leikkaussalissa-tai-julkaisemassa-ohjelmistoa-punnitse-ensin-riskit/081d4993-4bf4-4dc9-80f0-89e7a9f23bbd>. 19.9.2019. Luettu 27.10.2019.

8 Spets, Toni. 2019. Ylläpito – Mitä ja miksi? Verkkoaineisto. Whitestone Oy. <https://www.whitestone.fi/blogi/yllapito-mita-ja-miksi>. 28.8.2019. Luettu 27.10.2019.

9 Lowe, Steven. 9 metrics that can make a difference to today's software development teams. Verkkoaineisto. TechBeacon. < https://techbeacon.com/app-dev-testing/9-metrics-can-make-difference-todays-software-development-teams >. Luettu 27.10.2019.

10 KLOC. Verkkoaineisto. Webopedia. <https://www.webopedia.com/TERM/K/KLOC.html>. Luettu 28.10.2019.

11 Anderson, Kelsie. 8 Ways to Crank Up Speed in Software Development. Verkkoaineisto. Targetprocess. <https://www.targetprocess.com/articles/speed-in-software-development/>. Luettu 28.10.2019.

12 Hossain, Akhtar. 2019. How to Speed up the Web Development Process. Verkkoaineisto. Codementor. <https://www.codementor.io/learn-development/speed-up-web-development-process>. 28.7.2019. Luettu 28.10.2019.

13 Ghahrai, Amir. 2018. Pros and Cons of Test Driven Development. Verkkoaineisto. Testing Excellence. <https://www.testingexcellence.com/pros-cons-test-driven-development/>. 3.12.2018. Luettu 29.10.2019.

14 Collin, Paula. 2019. Jopa 10 000 työpaikkaa koodareille, mutta tekijät puuttuvat – ”Vaatii kaikkien osapuolten aktivoitumista”. Verkkoaineisto. Yle. < https://yle.fi/uutiset/3-10669492>. Julkaistu 4.3.2019. Luettu 30.10.2019.

15 Hagelin, Heidi. 2019. Visma palkkaisi heti 50 koodaria, jos heitä olisi – nuoria osaajia kiskotaan töihin jo koulunpenkiltäkin. Verkkoaineisto. Taloussanomat. <https://www.is.fi/taloussanomat/yrittaja/art-2000005985601.html>. Julkaistu 4.2.2019. Luettu 30.10.2019.

16 Rajapinta Helsingin pysäköintipaikkojen käytöstä. Verkkoaineisto. Avoindata. <https://www.avoindata.fi/data/fi/dataset/rajapinta-helsingin-pysakointipaikkojen-kaytosta>. Luettu 2.11.2019.

17 Aggarwal, Lalit. 2019. Angular 8 – Understanding Directory Structure & Creating CRUD App. Verkkoaineisto. Overflowjs. <https://overflowjs.com/posts/Angular-8-Understanding-Directory-Structure-and-Creating-CRUD-App>. 8.7.2019. Luettu 5.11.2019.

18 Tarvainen, Jani. 2016. Mikä on TypeScript? Verkkoaineisto. Symfony. <https://symfony.fi/artikkeli/mika-on-typescript>. Julkaistu 30.7.2016. Luettu 7.8.2019.

19 Ankkala, Mikko. 2017. Vaihtamalla paranee – 3 hyvää syytä valita TypeScript. Verkkoaineisto. Sysart Oy. <https://sysart.fi/blog/2017/12/04/vaihtamalla-paranee-3-hyvaa-syyta-valita-typescript/>. Julkaistu 4.12.2017. Luettu 7.8.2019.

20 Mikä on Single Page App ja mihin sitä käytetään. Verkkoaineisto. City Dev Labs Oy. <https://citydevlabs.fi/single-page-app/>. Luettu 5.8.2019.

21 Schwarzmüller, Maximilian. 2019. Angular 8 – The Complete Guide (2019+ Edition). Verkkoaineisto. <https://www.udemy.com/course/the-complete-guide-to-angular-2/ >. Päivitetty 10.2019. Luettu 15.10.2019.

22 NgModules. Verkkoaineisto. Google. <https://angular.io/guide/ngmodules>. Luettu 6.11.2019.

23 Component. Verkkoaineisto. Google. <https://angular.io/api/core/Component>. Luettu 6.11.2019.

24 Displaying Data. Verkkoaineisto. Google. <https://angular.io/guide/displaying-data>. Luettu 6.11.2019.

25 Attribute Directives. Verkkoaineisto. Google. <https://angular.io/guide/attribute-directives>. Luettu 6.11.2019.

26 Structural Directives. Verkkoaineisto. Google. <https://angular.io/guide/structural-directives>. Luettu 6.11.2019.

27 Introduction to services and dependency injection. Verkkoaineisto. Google. <https://angular.io/guide/architecture-services>. Luettu 6.11.2019.

28 Routing & Navigation. Verkkoaineisto. Google. <https://angular.io/guide/router>. Luettu 6.11.2019.

29 Angular vs React vs Vue: Which Framework to Choose in 2019. Verkkoaineisto. codeinwp. < https://www.codeinwp.com/blog/angular-vs-vue-vs-react/ >. Luettu 17.11.2019.