jamk.fi

JavaScriptistä TypeScriptiin

Frontend-ohjelmistokehityksen tehokkuuden parantaminen

Iiro Sarkki

Opinnäytetyö
Joulukuu 2020
Tietojenkäsittely ja tietoliikenne
Insinööri (AMK), tieto- ja viestintätekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Sarkki, Iiro	Opinnäytetyö, AMK	Joulukuu 2020
	Sivumäärä 30	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
JavaScriptistä TypeScriptiin
Frontend-ohjelmistokehityksen tehokkuuden parantaminen

Tutkinto-ohjelma
Tieto- ja viestintätekniikan tutkinto-ohjelma

Työn ohjaaja(t)
Raija Hämäläinen, Kari Niemi

Toimeksiantaja(t)
Probis Solutions Oy

Tiivistelmä

Probis Solutions Oy -yrityksen frontend-ohjelmistokehitys on jo pitkään toteutettu JavaScript-kielellä. Vaikka kieli sopii loistavasti yrityksen tarpeisiin, siinä on ilmennyt toistuvasti samat heikkoudet vuodesta toiseen. Esimerkiksi koodin virheenkorjaus, suurempien ohjelmakokonaisuuksien rakenteiden hallinta ja jatkokehityksen kömpelyys aiheuttavat päänvaivaa jokapäiväisessä työskentelyssä.

Probis halusi tutkittavan, voisiko yrityksen frontend-ohjelmistokehitystä tehostaa vaihtamalla JavaScriptistä TypeScriptiin, joka tiedettävästi tarjoaa ratkaisuja juuri edellä mainittuihin hankaluuksiin. Tavoitteena oli saada vahvistusta siitä, voisivatko TypeScriptin tuomat hyödyt helpottaa ja tehostaa ohjelmoijien työskentelyä. Toteutustavaksi valittiin yritykselle tyypillisen uuden websovelluksen tekeminen JavaScriptin sijaan TypeScript-ohjelmointikielellä siten, että muut sovelluskehykset ja toimintaperiaatteet säilytettiin entisellään. Työ toteutettiin laadullisena vertailevana tutkimuksena.

Ohjelmointityö onnistui odotetusti hyvin, ja uuden ohjelmointikielen implementointikaan ei tuottanut liikaa päänvaivaa. TypeScriptin jyrkähköstä oppimiskaaresta huolimatta työ vahvisti ennakko-odotuksia kielen tuomista hyödyistä ja ratkaisuista. Sovelluksen suunnittelutyö ja rakenteiden luominen tyypitysten kautta oli eniten aikaa vievää, mutta säästi aikaa projektin muissa vaiheissa sekä potentiaalisesti myös jatkokehityksessä. Toisaalta aikaa kului, sillä juuri tyypitys on keskeisin ero TypeScriptin ja JavaScriptin välillä ja täten uusi opeteltava asia, jota ei aikaisemmin JavaScriptiä käytettäessä voinut myöskään toteuttaa yhtä kattavasti - jos ollenkaan. Tutkimuksen myötä yritys sai varmuutta uuden tekniikan lisäämisestä repertuaariinsa, ja näin tehden pidemmällä aikavälillä luultavasti tehostaa yritystoimintaansa.

Avainsanat (asiasanat)

TypeScript, JavaScript, tyypittäminen, ohjelmistokehitys

jamk.fi

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Sarkki, Iiro	Bachelor's thesis	December 2020
		Language of publication: Finnish
	Number of pages 30	Permission for web publication: x

Title of publication

From JavaScript to TypeScript

Enhancing the efficiency of frontend application development

Degree programme

Information and Communication Technology

Supervisor(s)

Raija Hämäläinen, Kari Niemi

Assigned by

Probis Solutions Oy

Abstract

The frontend development at Probis Solutions Oy has been produced by utilizing JavaScript as the main programming language. Although JavaScript has proven to be very fitting for Probis' needs, over the years it has displayed a few persistent weaknesses that slow down the effectiveness of coding. Debugging, the structure handling of large-scale apps and somewhat cumbersome further development are but a few of these weaknesses that strain the workflow of the day to day. Probis wanted it to be researched that whether changing from JavaScript to TypeScript would enhance overall effectiveness of software development and tackle some of the aforementioned weaknesses. The goal was to gain confirmation of the known benefits of TypeScript that will ease and enhance the workflow of Probis' programmers. The chosen method of implementation was to develop a new app typical of Probis' past software products utilizing the same frameworks and principles with the exception that TypeScript would be used instead of JavaScript. The study was conducted using a qualitative research method. As expected, the development of the new app progressed well, and implementing TypeScript didn't cause any drastic complications. Despite TypeScript's steep learning curve, the study reinforced the expectations of the benefits TypeScript could bring. The planning and building of software structures were the most time-consuming but in return it saved a lot of time in the later phases of development – and assumably of further development stages as well. On the other hand, a lot of time was also invested to typing since it is the key difference between JavaScript and TypeScript, and it was mostly a new feature to learn alongside the development process. The study gave Probis Solutions the confidence to add TypeScript into their repertoire and in so doing most probably enhanced the efficiency of their front-end development.

Keywords/tags (subjects)

TypeScript, JavaScript, typing, software development

Miscellaneous (Confidential information)

Sisältö

Termit **3**

1 Johdanto **4**

1.1 Toimeksiantaja ja työn taustat 4

1.2 Tutkimusmenetelmä ja -kysymykset 5

2 Tyypittäminen **5**

3 JavaScript **7**

3.1 Mikä on JavaScript? 7

3.2 Vahvuudet ja heikkoudet 8

4 TypeScript **9**

4.1 Mikä on TypeScript? 9

4.2 Tyyppien rakenteista 10

4.3 Vahvuudet ja heikkoudet 12

5 JavaScriptin ja TypeScriptin erot ja yhtäläisyydet **13**

6 Ohjelmointityö **13**

6.1 Työn määrittely ja suunnittelu 13

6.2 Työn aloitus 16

6.3 Työn eteneminen ja havainnot 18

6.4 Työn viimeistely 23

7 Tutkimustulokset **24**

8 Pohdinta ja johtopäätökset **26**

Lähteet **29**

Kuvio

Kuvio 1. Staattisesti tyypitetty koodiesimerkki....6
Kuvio 2. Dynaamisesti tyypitetty koodiesimerkki....6
Kuvio 3. Esimerkki tyyppiyhdisteestä....11
Kuvio 4. Esimerkki tyyppiyhdisteestä....11
Kuvio 5. Esimerkki tyyppiyleistyksestä....12
Kuvio 6. Ohjelman käyttöliittymänäkymä....15
Kuvio 7. Ohjelman käyttöliittymänäkymä....16
Kuvio 8. Ohjelmassa käytettävien olioiden sisältöjen määrittely....17
Kuvio 9. Ohjelmassa käytettävien liikuteltavien datasisältöjen määrittely....18
Kuvio 10. Esimerkki IDE:n muuttujan ennakoimisesta....19
Kuvio 11. Komponentin datan tyypitys....21
Kuvio 12. Tyypitetyt frontend API-funktiot....22
Kuvio 13. Frontendin API:n hallintafunktioita....23

Termit

Backend eli taustasovellus

Sovellus, joka ei näy käyttäjälle. Verkkokehityksessä esimerkiksi kontrolleri ja tietokantojen hallinta ovat osa taustasovellusta.

Compiler eli ohjelmointikielen kääntäjä

Sovellus, joka kääntää ihmisen ymmärtämän ohjelmointikielisen lähdekoodin tietokoneen suorittimen ymmärtämään muotoon.

CSS

Lyhenne sanoista Cascading Style Sheets. Sisältää verkkosivun ulkoasun tyyliohjeet.

Frontend eli edustasovellus

Käyttäjälle näkyvä sovelluksen osa. Esimerkiksi käyttöliittymät ovat iso osa edustasovellusta.

Framework eli sovelluskehys

Runko, jonka päälle uusi ohjelma rakennetaan. Koostuu usein useista yhteensopivista kirjastoista.

HTML

Lyhenne sanoista Hypertext Markup Language. Kieli, jolla verkkosivut kirjoitetaan.

IDE eli ohjelmointiympäristö

Lyhenne sanoista Integrated Development Environment. Ohjelma tai joukko ohjelmia, joilla ohjelmoija kirjoittaa ohjelmistokoodia.

Olio

Olio-ohjelmoinnin perusyksikkö, joka sisältää joukon tietoa ja toiminnallisuutta. Käytetään kuvastamaan abstraktin tai reaalimaailman käsitteen ilmentymää ohjelmistossa.

1 Johdanto

1.1 Toimeksiantaja ja työn taustat

Ohjelmointityönä tehtiin taloushallinto-ohjelma, jolla luodaan vyörytyssääntöjä resurssoinnin automatisointia varten.

Toimeksiantaja oli vuonna 2003 Jämsässä perustettu keskisuomalainen pieni ohjelmointitalo Probis Solutions Oy. Probis on erikoistunut suurien ohjelmistokokonaisuuksien räätälöityihin toteutuksiin yritysasiakkaille ja työllistää tällä hetkellä kymmenen henkilöä. Yrityksellä on toimipisteet sekä Jyväskylässä että Tampereella, mutta kaikki työ toteutetaan käytännössä Jyväskylässä. (Probis n.d.)

Probis Solutions Oy on kiinnostunut laajentamaan JavaScriptistä TypeScriptin käyttöön, koska se saattaisi tehostaa yritystoimintaa. Yritys on pieni, ja tästä syystä yksittäiselle työntekijälle tulee helposti vastuu suuristakin kokonaisuuksista. Jos sama työntekijä on vastuussa lähes kokonaan yksittäisen ohjelmointityön läpiviennistä, se nostaa myös äkkiä inhimillisten virheiden määrää. Yrityksessä ei myöskään ole erikseen osoitettuja testaajia, joten vastuu työn jäljestä kasvaa entisestään. TypeScriptillä olisi mahdollisesti potentiaalia parantaa työjäljen laatua, koska oikein käytettynä sen sisäänrakennetut periaatteet ja säännöstöt takaavat, että työntekijä huomaa eksplisiittisimmätkin virheet helpommin reaaliajassa eikä vasta jälkikäteen esimerkiksi testausvaiheessa. Tämän seurauksena virheet eivät pääse kumuloitumaan (aikaisemmasta huomaamattomasta virheestä johtuvat uudet virheet) ja työn, eli tässä tapauksessa koodauksen, jälki pysyy kautta linjan koherentimpana ja soljuvampana. Itse ohjelmointityö tehtiin Probis Solutions Oy:n asiakkaan tarpeesta.

Vallitseva mielipide tuntuu olevan, että TypeScriptin käyttäminen JavaScriptin sijaan olisi tehokas tapa parantaa työnlaatua, ja yhä useampi suuri yritys vannoo TypeScriptin nimeen (Ghate 2017; Ramos 2020), joten tutkimus antaa hyvän tilaisuuden ymmärtää ilmiön syitä ja potentiaalisesti vahvistaa sitä.

1.2 Tutkimusmenetelmä ja -kysymykset

Tutkimuksessa käytettiin pääosin laadullista vertailevaa menetelmää, sillä tutkimus koostui uudella tekniikalla eli TypeScriptillä toteutetun ohjelmistokehityksen vertaamisesta vanhaan vastaavaan, mutta JavaScriptillä toteutettuun, ohjelmistokehitysasetelmaan. Laadullisella eli kvalitatiivisella tutkimuksella tarkoitetaan asian, tässä tapauksessa tekniikan, kokonaisvaltaista tutkimista asialle ominaisessa ympäristössä (Bister 2019, 33). Tutkimuksessa sivuttiin myös määrällistä menetelmää siltä osin, että siinä pyrittiin arvioimaan karkeasti sitä kehystä, missä TypeScriptin käyttäminen tulee JavaScriptiä paremmaksi vaihtoehdoksi työnjäljen ja työtehokkuuden kannalta. Määrällisen eli kvantitatiivisen menetelmän vaatimukset kuitenkin täyttyvät hyvin löyhästi, sillä menetelmä vaatisi tutkimuksessa käytettyjen aistihavaintojen ja loogisen päättelyn lisäksi statistiikkaan pohjautuvaa analysointia aineiston sekä tulosten merkittävyyden osalta (Bister 2019, 38).

Tutkimuksella toivottiin löydettävän vastaukset seuraaviin kysymyksiin:

1. Voiko JavaScriptistä TypeScriptiin vaihtamalla tehostaa työn laatua ja/tai nopeutta ja jos, niin mitkä tekijät vaikuttavat vaihtamisen kannattavuuteen?
2. Tuleeko koodista luettavampaa, ja mitkä ovat vaikutukset jatkokehitykseen?

2 Tyypittäminen

Tyypittäminen on tarkasteltavien ohjelmointikielien, JavaScriptin ja TypeScriptin, suurin yksittäinen ero. Tyypittämisellä selitetään datan sisältöä ja sitä, minkälaista manipulaatiota datalle saa tehdä. Tyypittäminen jaetaan yleensä karkeasti kahteen kategoriaan: staattisesti tyypitettyihin ja dynaamisesti tyypitettyihin.

Staattisesti tyypitetyissä ohjelmointikielissä muuttujan tyyppi pitää ilmoittaa (declare) etukäteen, ennen kuin muuttujaa voi käyttää. Muuttujaa voi kuitenkin käyttää jo, ennen kuin se on määritelty (define), eli ennen kuin sille on asetettu arvoja. Muuttuja ei saa sisältää muun tyyppistä dataa, kuin mitä sille on ilmoitettu. (Singh Rao 2018.)

Kuviossa 1 on staattisesti tyypitetty koodiesimerkki.

```
241  /*Staattisesti tyypitetty*/
242
243  var taulukko: Array<number> //Ilmoitetaan muuttujan tyypiksi taulukko,
244                              //joka sisältää numeroita
245
246  taulukko.toString() //Muuttuja löytää taulukon metodit (kuten toString),
247                      //vaikka sille ei ole vielä annettu arvoja
248
249  taulukko = [1]      //Ilmoitetun tyypin mukainen määrittely onnistuu,
250  taulukko = 'teksti' //mutta kun antaa väärän tyyppisen arvon, tulee virhe
251                      //"Type "teksti" is not assignable to type number[]"
```

Kuvio 1. Staattisesti tyypitetty koodiesimerkki

Dynaamisesti tyypitetyissä ohjelmointikielissä asia taas on päinvastoin: muuttuja pitää määritellä ennen sen käyttämistä, mutta tyyppideklaraatiota ei tarvita, sillä muuttujan tyyppi määräytyy sille annettujen arvojen perusteella. Dynaamisesti tyypitetyn muuttujan voi myös yleensä määritellä uudestaan koska vain. (Singh Rao 2018.)

Kuviossa 2 on esitetty dynaamisen tyypityksen esimerkki.

```
15
16  /*Dynaamisesti tyypitetty*/
17
18  var taulukko = [] //Määritetään taulukoksi, joten tyyppi määräytyy automaattisesti
19
20  taulukko.push(1) //Taulukon (array) metodit, kuten push(), toimivat kivasti
21
22  taulukko = 'kikka!' //toisaalta muuttujan voi lennosta vaihtaakin stringiksi
23
```

Kuvio 2. Dynaamisesti tyypitetty koodiesimerkki

3 JavaScript

3.1 Mikä on JavaScript?

JavaScript on olio-ohjelmointikieli, jonka avulla esimerkiksi verkkosivulle voidaan implementoida monimutkaisia ominaisuuksia ja dynaamista sisältöä kuten karttoja,

videoita tai pelejä. Standardissa web-kehityshierarkiassa se on kolmas kerros HTML:n ja CSS:n jälkeen. Kuten kielen nimi jo kertoo, JavaScriptillä tehdyt ohjelmat ovat skriptejä. Skripti on joukko ohjeita koodatussa muodossa, ja skriptin tarkoitus on muuttaa esimerkiksi verkkosivua määrätyllä tavalla. JavaScript-koodia ei tarvitse koota millään tavalla, koska selaimissa on JavaScript-kääntäjä, joka hoitaa koodin tulkinnan automaattisesti. (Learn JavaScript Tutorial n.d.)

JavaScript on dynaamisesti tyypitetty kieli, eli muuttujaan voi alustaa minkä vain tyyppistä dataa, olipa se sitten esimerkiksi tekstiä (string), kokonaislukuja (integer) tai taulukko (array) (JavaScript data types and structures n.d.).

JavaScriptille on kehitetty paljon sovelluskehyksiä eli *framework*eja. *Framework* on kokoelma valmiita koodikirjastoja, joita kehittäjät voivat käyttää helpottamaan oman työnsä tekoa. Ohjelmistokehityksessä toistuvat usein samat käytänteet sekä ongelmat, joten on ollut luontevaa luoda valmiita sovelluskehyksiä, joilla pyritään nivomaan yhteen parhaat toimintatavat. JavaScriptiä käytetään sekä servereiden että käyttöliittymien ohjelmointiin, ja molemmille puolille löytyy nykyään myös lukuisia *framework*eja.

Käyttäjäpään, eli käyttöliittymäkehitykseen tarkoitetuista, *framework*eista on helppo nostaa esiin kaksi eniten käytettyä: AngularJS ja React.JS. AngularJS on Googlen vuonna 2016 julkaisema avoimen lähdekoodin *framework*, joka tukee moitteettomasti perinteistä Model View Controller -mallia eli MVC-mallia (What is AngularJS n.d.). React.JS on Facebookin kehittämä ja ylläpitämä kokoelma JavaScript-kirjastoja, ja perinteisessä MVC-mallissa se edustaa yleensä lähinnä View-osaa eli käyttöliittymän näkymää (ReactJS Overview n.d.). Toisin kuin AngularJS se vaatii toimiakseen eräänlaisen moduuliniputtajan (bundler) kuten Webpackin. Webpack käy järjestelmällisesti läpi ohjelman ja niputtaa kaikki moduulit ja riippuvuudet, kuten kuvat ja fontit, yhteen nippuun ja toimittaa ne selaimelle (Concepts n.d.).

3.2 Vahvuudet ja heikkoudet

JavaScriptistä löytyy monia vahvuuksia, eikä siis ole ihme, että se on yksi maailman käytetyimmistä ohjelmointikielistä. JavaScriptin oppimiskäyrä on helpohko, ja sen opetteluun löytyy ohjeita ja tutoriaaleja erittäin paljon. JavaScript on helppo ja nopea ottaa käyttöön siksi, että sille löytyy tuki kaikista selaimista ja alustoista ja kieli on kirjoitusasultaan selkeää ja intuitiivista. JavaScriptiä voi ryhtyä ohjelmoimaan esimerkiksi tavallisella tekstinkäsittelyohjelmalla.

JavaScriptin käyttö on dynaamisen tyypityksensä vuoksi hyvin nopeaa ja vaivatonta, joten se luo myös suuren määrän vastuuta ohjelmoijan tarkkuuden suhteen. Tästä syystä JavaScriptin heikkoudet käyvät ilmi sitä paremmin, mitä suurempi ja monimutkaisempi ohjelmakokonaisuus on kyseessä.

Koska JavaScriptiä ei tarvitse koota ennen suorittamista, ilman kehittynyttä ohjelmointiympäristöä (IDE) JavaScriptin virheet myös käyvät ilmi vasta koodia ajettaessa, mikä hidastaa virheenseulontaa entisestään. Kaikki virheet eivät välttämättä myöskään ole sellaisia, joista selaimen kääntäjä edes osaisi ilmoittaa, vaan ohjelma ei vain toimi odotetulla tavalla ja käyttäjä joutuu itse jäljittämään virheen paikan. Joskus virheilmoitus on myös harhaanjohtava tai turhan abstrakti, sillä kääntäjä saattaa ilmoittaa virheeksi jonkin yleisen poikkeuksen tai takertua johonkin oikean virheen aiheuttamaan muuhun seuraukseen.

4 TypeScript

4.1 Mikä on TypeScript?

TypeScript on Microsoftin kehittämä JavaScriptin laajennus (superset), joka sisältää kaikki samat ominaisuudet kuin JavaScript ja lisää siihen staattiset tyypitykset. Toisin kuin JavaScript TypeScript pitää koota (compile) ennen suorittamista. Kokoamisen lopputulos on puhdasta JavaScriptiä, ja tästä syystä TypeScript myös toimii yhtä kat-

tavasti kaikilla samoilla alustoilla kuin JavaScriptkin. Selaimet ja niiden versiot eivät vaikuta kääntämiseen millään tavalla, sillä kääntäminen tapahtuu palvelimilla. Kokoamisen aikana tarkistetaan tyypitykset ja muu koodin eheys, mutta nämä eivät vaikuta käytännössä lopputulokseen, eli koodi kootaan, vaikka virheitä olisikin. (TypeScript – Overview n.d.)

Jos koodatessa käyttää hyvää TypeScriptiä ymmärtävää *IDE:tä*, kuten tutkimuksen ohjelmointityössä käytetty ohjelma PhpStorm, osaa *IDE* analysoida kirjoitettua koodia reaaliajassa, ja ohjelmoija näkee virheet jo ennen kokoamisvaihetta (PhpStorm: The Lightning-Smart IDE for PHP Programming by JetBrains n.d.).

TypeScript on siis staattisesti tyypitetty kieli, mutta koska siihen on implementoitu tyypin päättely (type inference), sen tyyppideklaraatiot ovat valinnaisia, eli TypeScript asettaa tyypin muuttujalle annetun arvon perusteella (TypeScript: Handbook – TypeScript For JavaScript Programmers n.d.), mutta staattisen tyypityksen mukaisesti kyseinen muuttuja ei enää hyväksy muun tyyppisiä arvoja. Tosiasiassa TypeScript on siis tavallaan sekä dynaamisesti että staattisesti tyypitetty kieli. TypeScript antaakin melko monipuoliset koodinkokoajan säätömahdollisuudet tyypityksen tiukkuuden suhteen.

TypeScriptin virheentarkistusasetusten päälle/pois -valinnat

- *noImplicitAny*
- *noImplicitThis*
- *strictBindCallApply*
- *strictNullChecks*
- *strictFunctionTypes*
- *strictPropertyInitialization*
- *alwaysStrict*
- *strict*

NoImplicitAny antaa virheen, jos koodin lausekkeissa tai tyyppideklaraatioissa on johdettuja *any*-tyypin määrityksiä. *NoImplicitThis* antaa virheen, jos koodissa *this*-lausekkeissa on implisiittisiä *any*-tyypin määrityksiä (*This* viittaa tyypillisesti sillä hetkellä käsiteltävän luokan objektiin). *StrictBindCallApply* asettaa funktioiden *bind*-, *call*- ja

apply-metodien tyyppitarkistukset ehdottomiksi. *StrictNullChecks*-asetuksen ollessa päällä, *null*- tai *undefined* -arvoja ei hyväksytä, ellei niitä ole erikseen tyyppideklaraatiossa sallittu. *StrictFunctionTypes* poistaa kaksimuotoisten parametrien tarkistuksen funktiotyyppien osalta. *StrictPropertyInitialization* varmistaa, että luokan ominaisuuksille on joko määrätty oletusarvot tai ne asetetaan luokkaobjektia luodessa. *AlwaysStrict* parsii koodin JavaScriptin *strict*-moodissa ja lisää “use strict” -määritteen koottuun koodiin. *Strict* on ankarin tiukkuusvaatimus, joka käytännössä kattaa kaikki yllä olevat. (Rippon 2020.)

4.2 Tyyppien rakenteista

Yhdiste (Union)

Tyyppideklaraatiossa ei ole välttämätöntä rajata tyyppimääritelmää vain yhteen tyyppiin. Tyyppiyhdiste on myös helppokäyttöinen, sillä syntaksiksi kelpaa tolppamerkki |, jota käytetään ohjelmoinnissa muutenkin logiikan määrittelyssä TAI-erottajana (TypeScript: Handbook – TypeScript For JavaScript Programmers n.d.). Kuviossa 3 *min* ja *max* on määritelty yhdisteellä siten, että *min* saa olla joko tyyppiä *string* TAI tyyppiä *null* ja *max* tyyppiä *string* TAI *number*.

```
min: string | null;
max: string | number;
```

Kuvio 3. Esimerkki tyyppiyhdisteestä

Yhdisteellä voidaan luoda myös eksplisiittisempiä tyyppisääntöjä. Kuviossa 4 on ilmoitettu, että *value*-arvoksi saa antaa numeron väliltä 1-5.

```
value: 1 | 2 | 3 | 4 | 5;
```

Kuvio 4. Esimerkki tyyppiyhdisteestä

Yleistykset (Generics)

TypeScriptissä tyypit voidaan ilmoittaa joko yleistäen tai eristäen. Esimerkiksi tyypiksi voitaisiin antaa yksinkertaisesti *List*, joka tarkoittaa sitä, että muuttujan tyyppi on lista, joka voi sisältää mitä tahansa, kunhan se on varmasti tyyppiä *List*. Toisaalta voidaan määritellä, että tyyppi on *List*, joka saa sisältää vain tietynlaista dataa, eli eristetään listan sisältö. (TypeScript: Handbook – TypeScript For JavaScript Programmers n.d.)

Kuviossa 5 näkyvä *items1* on tyypiltään yleinen lista eli lista, joka saa sisältää mitä arvoja tahansa. *Items2* on identtinen *items1*:n kanssa, sillä syntaksi sallii kumman vain kirjoitusasun. *Items3*-tyyppi sen sijaan on eristetty listaksi, joka saa sisältää ainoastaan numeroita.

```
items1: List;
items2: List<any>;

items3: List<number>;
```

Kuvio 5. Esimerkki tyyppiyleistyksestä

4.3 Vahvuudet ja heikkoudet

TypeScriptin lyömättömiä vahvuuksia ovat kielen tyypityksen ja reaaliaikaisen koodianalysoinnin tuomat ylläpidettävyys ja luotettavuus. Lisäksi TypeScript ymmärtää JavaScriptiä, minkä seurauksena tyyppien määritteleminen on hyvin monimuotoista. Suurimpina heikkouksina taas mainittakoon jyrkähkö opettelukaari sekä aikaa vievä työn alustus- ja suunnitteluvaihe.

TypeScriptin tyypitykset luovat hyvin dynaamisen, mutta järjestelmällisen, koodausympäristön, mikä parantaa työn hallittavuutta ja vähentää virheitä. Toisaalta koska koodaaja voi vapaasti itse päättää, kuinka tiukasti tyyppejä ja koodin eheyttä vahditaan, se asettaa paljon lisäpainoa ja -työtä ohjelmiston struktuurin suunnitteluun ja työnkulun ennakointiin - ainakin TypeScriptin ensimmäisillä käyttöönottokerroilla.

Selkeä heikkous JavaScriptiin verrattaessa on myös koodin kokoamisen välttämättömyys. Vaikka yksittäinen kokoaminen kestää mitättömältä tuntuvan ajan, se kuitenkin kumuloituu esimerkiksi työpäivän aikana jo melko suureksi, sillä koodia ajetaan yleensä useita kertoja ohjelmaa kirjoitettaessa.

5 JavaScriptin ja TypeScriptin erot ja yhtäläisyydet

Toisaalta kielien vertailu tuntuu hassulta, koska TypeScript on periaatteessa vain JavaScriptin kehittyneempi muoto ja teknisesti ottaen kaikki, mikä voidaan toteuttaa TypeScriptillä, voitaisiin toteuttaa pelkästään JavaScriptillä. Tästä syystä vertailun tarve syntyykin työn tehokkuuden, koodin ylläpidon ja jatkokehityksen optimoimisesta.

JavaScript on suunniteltu erityisesti pienten ohjelmien ja komponenttien tekoon. Tätä ajatusta tukee kielen yksinkertaisuus, ketteryys ja dynaamisuus. Nykyään kuitenkin JavaScript on kasvanut ja yleistynyt niin paljon, että sen pohjalta tehdään suuriakin monimuotoisia ohjelmistokokonaisuuksia, ja sen myötä kielen ominaisuudet paljastavat myös paremmin heikkoutensa. Erityisesti tyypittämättömyys luo hankalia virhetilanteita suurissa kokonaisuuksissa, sillä virheenseulonta hankaloituu, kun muut-

tujien haluttu tyyppi ei ole etukäteen tiedossa. Täten koodaaja joutuu kuormittamaan työmuistiaan tai käyttämään turhaa työaikaa selvittääkseen muuttujan tyypin ja myös inhimillisen erheen todennäköisyys kasvaa.

JavaScript-standardin (ECMAScript) kuudennessa versiossa esiteltiin JavaScriptin luokat, joita implementoimalla saadaan luotua hieman parempaa struktuuria frontend-applikaatioille, mutta samoja luokkarakenteita voidaan hyödyntää myös TypeScriptissä, eivätkä ne JavaScriptissä tarjoa tarpeeksi kilpailukykyä TypeScriptin tuomia etuja vastaan.

6 Ohjelmointityö

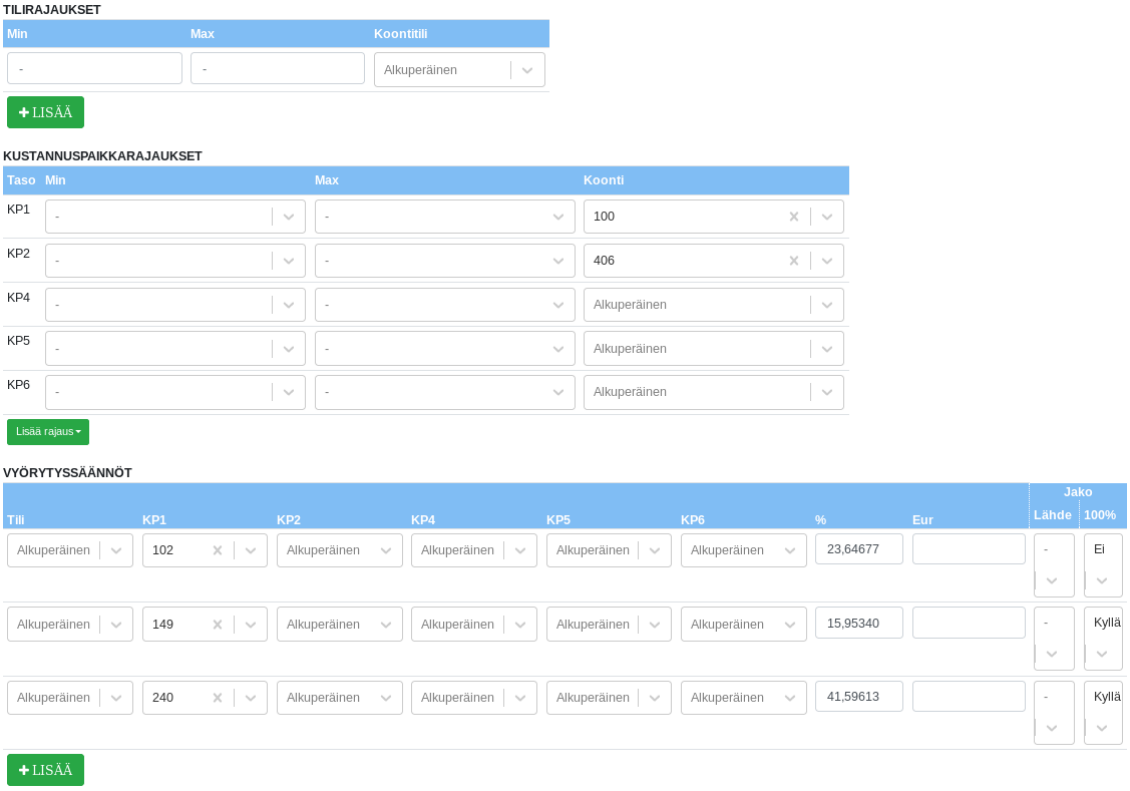
6.1 Työn määrittely ja suunnittelu

Koska yrityksen tarve tutkimukselle ei tullut ohjelmasta, vaan uuteen tekniikkaan siirtymisen potentiaalisten hyötyjen tutkimisesta, prosessi alkoi sopivan ohjelmointikokonaisuuden löytämisestä. Ohjelmointityöksi valittiin työ, joka vastaa ominaisuuksiltaan ja tekniikoiltaan mahdollisimman paljon niitä, mitä opinnäytetyön tekijän päivätyössä on eniten käytetty viimeisen viiden vuoden aikana. Tämä ratkaisu takaisi luotettavimman vertailukohdan JavaScriptin ja TypeScriptin käytöstä syntyviin eroihin, sillä näin tehden ainoa muuttunut tekijä on ohjelmointikieli. Käyttöliittymän tekemiseen käytettiin siis ennestään tuttua ReactJS *framework*ia, joka toimi suoraan myös TypeScriptin kanssa.

Tuleva ohjelmisto on taloushallinnon resurssoinnin automatisointijärjestelmän komponentti. Ohjelmistolla luodaan vyörytyssääntöjä, joiden pohjalta yrityksen eri osastojen tai kustannuspaikkojen kuluja jaetaan muihin sektoreihin. Nykyjärjestelmässä vyörytykset hoidetaan manuaalisesti, joten ohjelmistolla pyritään tehostamaan ja helpottamaan taloushallinnon työntekijöiden työtä.

Ohjelmaan tarvittiin käyttöliittymä, jossa pystyy luomaan sääntöjä siitä, miten tietyn yrityksen kulut voidaan jakaa laskutettaviin sektoreihin. Kulujen jako eli vyörytys suo-

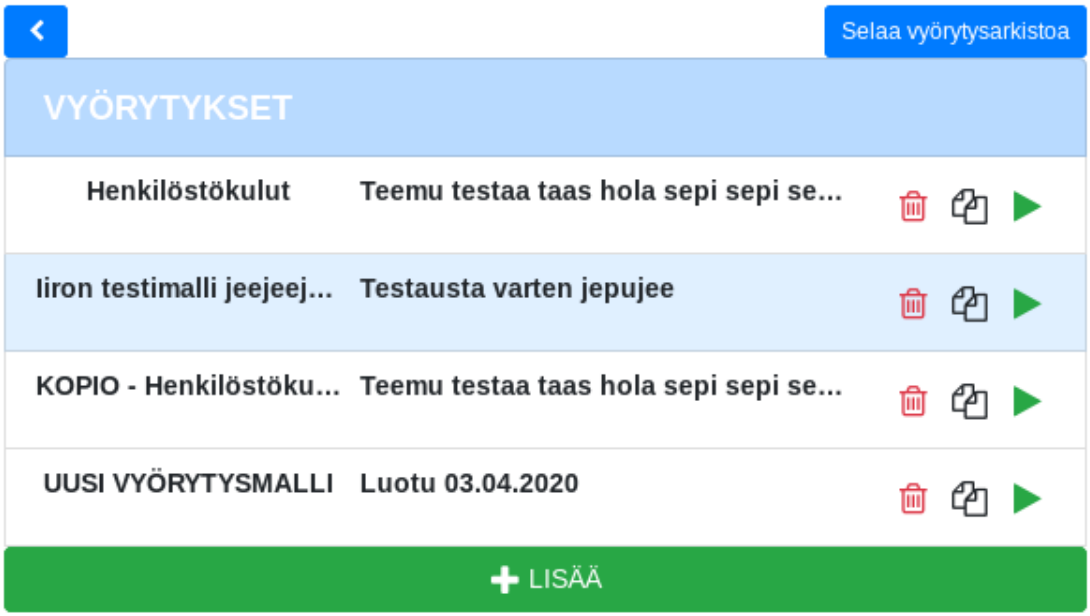
ritettaisiin erillisellä järjestelmän taustakomponentilla joko ajastetusti automaattisesti tai manuaalisesti käyttäjän toimesta. Tätä vyörytyksen suoritusta varten tarvittiin määritelmät tilien ja kustannuspaikkojen rajauksista ja näitä rajauksia taas vuorostaan sovelletaan vyörytyssääntöjen määrittelemiseen (ks. kuvio 6).



Kuvio 6. Ohjelman käyttöliittymänäkymä

Vaikka käyttöliittymä näyttää melko yksinkertaiselta, ohjelmassa kuitenkin liikkuu paljon pientä dataa, etenkin *ReactJS:n* tilassa, joten TypeScriptin tyypitykset olivat tervetulleita.

Lisäksi tarvittiin luotujen vyörytysten listaamiselle käyttöliittymä, jonka kautta onnistuisivat myös vyörytysten manuaalinen suorittaminen sekä muokkaus- ja poistotoiminnot (ks. kuvio 7).



Kuvio 7. Ohjelman käyttöliittymänäkymä

6.2 Työn aloitus

TypeScript-ohjelmointi lähti kankeasti käyntiin, sillä kielen syntaksi ja tyyppitarkistukset sekä tarkistusmääritykset ovat suurin eroavaisuus JavaScriptistä. Yllättävän monta asiaa piti tehdä eri tavalla kuin aikaisemmin JavaScriptillä, ja erityisesti työn pohjustus vaatii enemmän painotusta, sillä TypeScript tuo mukanaan todella paljon erilaisia valinnaisia tyypitysmahdollisuuksia. Tästä syystä alkoi todella nopeasti tuntumaan siltä, että pitää tehdä selkeitä rajauksia siihen, kuinka kattavasti ja syvällisesti tyypityksiä haluaa ja kannattaa alkaa tekemään.

Tyypitysjaotteluja päädyttiin tekemään pääasiassa karkeasti kaksi isompaa käyttäen TypeScriptin *interface*ja. Ensimmäiseksi päätettiin luoda tyyppikuvaukset olioista (ks. kuvio 8), joita ohjelmassa käytetään. Toiseksi määriteltiin ohjelmassa liikuteltavien datakokonaisuuksien sisällöt (ks. kuvio 9), jotka sisältävät muun muassa edellä mainittuja oliotyyppejä.

```
export interface EntityType {
    id: number;
}

export interface AllocationSource extends EntityType {
    accountExpression: string;
    name: string;
}

export interface AllocationAccount extends EntityType {
    min: string | null;
    max: string | null;
    elimination: string;
    template: number;
}

export interface AllocationCostCentre extends EntityType {
    dimension: number;
    min: string | null;
    max: string | null;
    elimination: string;
    template: number;
}

export interface AllocationRule extends EntityType {
    source: number | null;
    percentage: number | null;
    fixedValue: number | null;
    sourceTotal: boolean;
    template: number;
    account: string | null;
    costCentres: string;
}

export interface AllocationTemplate extends EntityType {
    name: string;
    description: string;
}

export interface Allocation extends EntityType {
    company: number;
    excludedVoucherGroups: number[] | null;
    excludedVoucherTypes: number[] | null;
    startDate: Date;
    endDate: Date;
    targetDate: Date;
    template: number;
    logPerson?: number;
    logTime?: Date;
    logTemplate?: string;
    vouchers?: number[];
}
```

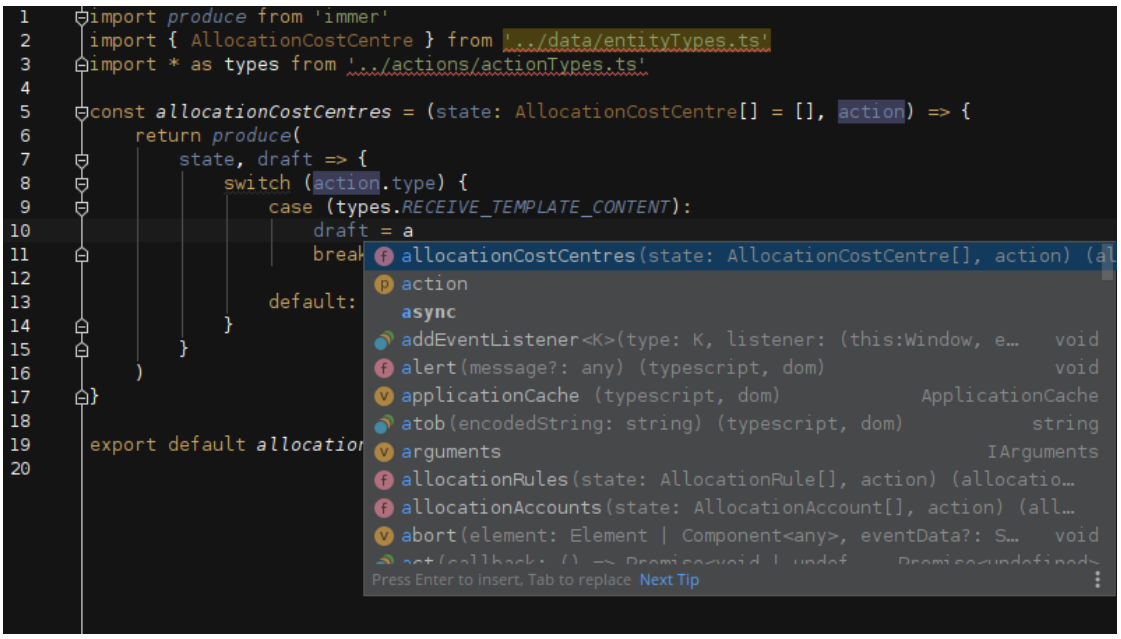
Kuvio 8. Ohjelmassa käytettävien olioiden sisältöjen määrittely

```typescript
import {
    Allocation,
    AllocationAccount,
    AllocationCostCentre,
    AllocationRule,
    AllocationTemplate,
    EntityType,
    PaginationInfo
} from './entityTypes'

export interface TemplateContent {
    allocationTemplate?: AllocationTemplate;
    allocations: Allocation[];
    info: PaginationInfo;
    allocationAccounts: AllocationAccount[];
    allocationCostCentres: AllocationCostCentre[];
    allocationRules: AllocationRule[];
}

export interface NewTemplateContent {
    allocationTemplate: AllocationTemplate;
    allocationAccounts: AllocationAccount[];
    allocationCostCentres: AllocationCostCentre[];
}

export interface AllocationOptionsContent {
    voucherGroups: Array<EntityType & Record<string, any>>;
    voucherTypes: Array<EntityType & Record<string, any>>;
}

export interface RunAllocationContent {
    voucherIds: number[];
    allocations: Allocation[];
}

```

Kuvio 9. Ohjelmassa käytettävien liikuteltavien datasisältöjen määrittely

6.3 Työn eteneminen ja havainnot

Parinkymmenen tunnin TypeScript-ohjelmoinnin jälkeen alkavat kielen tuomat hyödyt näkymään selkeästi. Ohjelmointi on koodinkirjoituksesta lähtien vikkelämpää, sillä kun on ensin käyttänyt hieman aikaa ja tarkkuutta datatyyppien kuvauksiin, *IDE* osasi jo ennakoida haettuja (ks. kuvio 10) muuttujia, ja näin ollen säästyy paitsi turhalta kirjoittamiselta, myös ylimääräiseltä työmuistin kuormitukselta.



Kuvio 10. Esimerkki IDE:n muuttujan ennakoimisesta

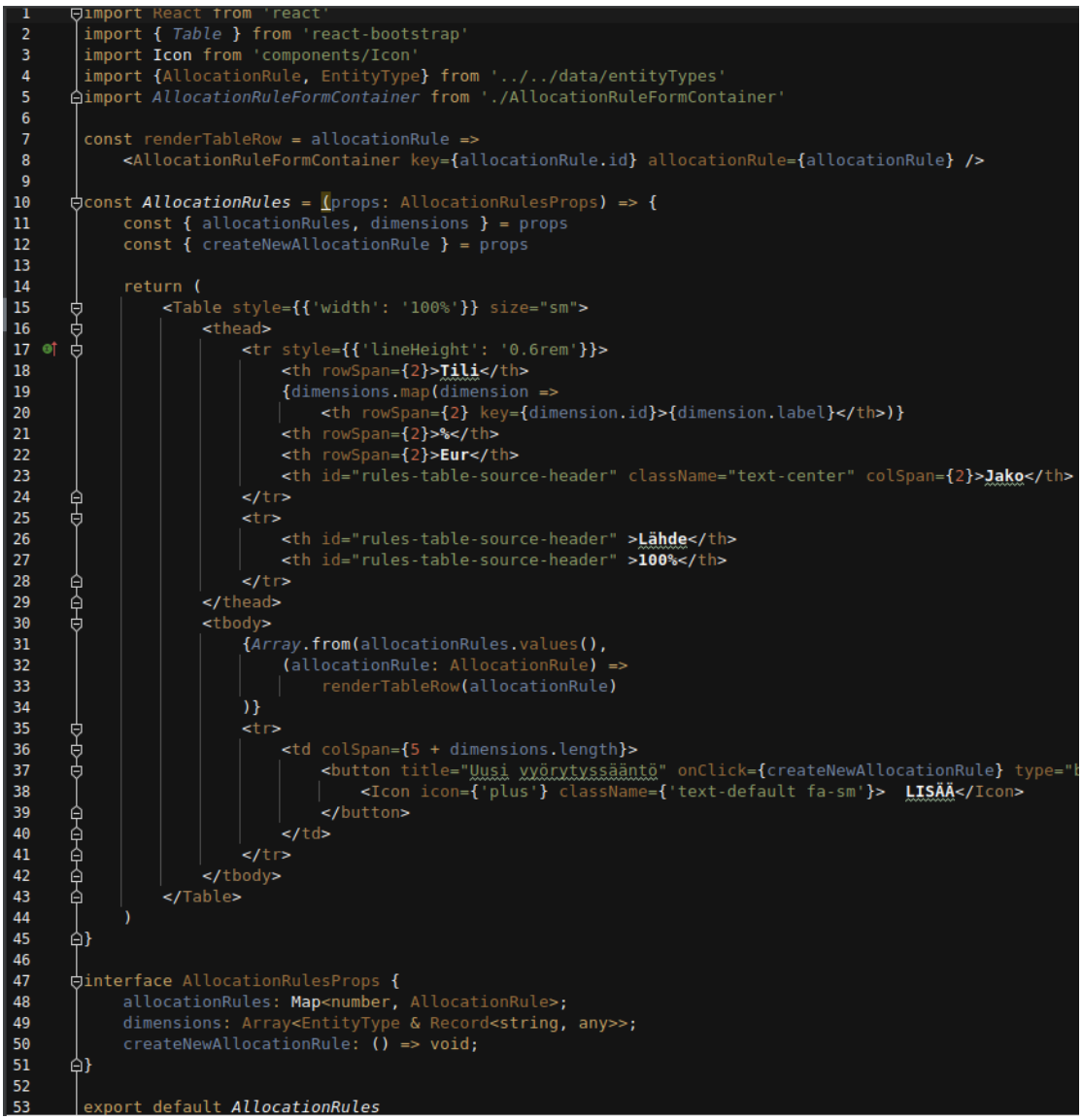
Komponentit

Reactin funktionaaliselle komponentille tuodaan dataa niin kutsutusta *container*ista, ja komponentti ei oletusarvoisesti tiedä mitään sille annettavan datan sisällöstä. Puhtaasti JavaScriptillä ohjelmoitaessa tästä aiheutuu pidemmän päälle potentiaalisesti suuriakin riskejä, ja parhaassakin tapauksessa se hidastaa työn tekoa ja hankaloittaa jatkokehitystä, sillä koodia kirjoittava tai tutkiva henkilö joutuu joko muistamaan ulkoa tai hyppimään eri tiedostojen välillä löytääkseen, minkä tyyppistä dataa komponenteissa liikkuu. Mitä suuremmaksi ohjelma kasvaa, sitä enemmän muistettavaa tai tarkistettavaa myös kertyy, ja inhimillisen erheen riski kasvaa. Tämä taas puolestaan johtaa helposti virheisiin.

TypeScriptin tyypityssyntaksi vapauttaa ohjelmoijan edellä mainitusta taakasta, sillä komponentille voidaan suoraan kertoa, mitä sille tuotu data sisältää. Kun komponentti tuntee käsiteltävän datan, se osaa ilmoittaa, jos ohjelmoija on tekemässä virhettä tai epäloogista toimenpidettä. Kun potentiaalisiin virheisiin puututaan tällä tavalla, se paitsi nopeuttaa työskentelyä sillä hetkellä myös estää virheen aiheuttamasta lumipalloefektiä. Lumipalloefektin seurauksena ohjelmoija saattaa huomata vir-

heen vasta myöhemmässä vaiheessa, kun virheellinen data on jo siirtynyt toiseen paikkaan ohjelmassa, ja siinä vaiheessa myös virheenseulonta ja -korjaaminen on monin kerroin työläämpää.

Kuviossa 11 on *AllocationRules*-niminen funktionaalinen komponentti, jolle on tuotu dataa yhdessä objektissa, ja se on nimellä *props*. TypeScriptin syntaksin mukaisesti sille on määritelty tyypiksi *AllocationRulesProps*, joka taas on muotoa interface. Kuvion alareunassa näkyy edellä mainitun interfacen sisältö, ja siitä käy ilmi, että data-objektin sisällä on kolme tietuetta: yksi Map, yksi array eli taulukko sekä yksi funktio. Map-tietue *allocationRules* sisältää aikaisemmin esiteltyjä olioita nimeltä *AllocationRule*, *dimensions*-niminen taulukko sisältää tietueita, jotka ovat sekä edellä kuvattua tyyppiä *EntityType* että tyyppiä *Record*. Nyt kun komponentti tietää sille tuotujen datojen tyypit, se osaa implisiittisesti tutkia tyyppien ominaisuuksia, kuten esimerkiksi Map-tyypin metodeja tai *AllocationRule*-olion parametreja. Näistä komponentille annetuista tiedoista viisastuneena komponentti osaa täten ennustaa ohjelmoijan koodinkirjoitusta ja varoittaa virheistä (ks. kuvio 10.)



```
import React from 'react'
import { Table } from 'react-bootstrap'
import Icon from 'components/Icon'
import {AllocationRule, EntityType} from '../../data/entityTypes'
import AllocationRuleFormContainer from './AllocationRuleFormContainer'

const renderTableRow = allocationRule =>
    <AllocationRuleFormContainer key={allocationRule.id} allocationRule={allocationRule} />

const AllocationRules = (props: AllocationRulesProps) => {
    const { allocationRules, dimensions } = props
    const { createNewAllocationRule } = props

    return (
        <Table style={{'width': '100%'}} size="sm">
            <thead>
                <tr style={{'lineHeight': '0.6rem'}}>
                    <th rowSpan={2}>Tili</th>
                    {dimensions.map(dimension =>
                        <th rowSpan={2} key={dimension.id}>{dimension.label}</th>)}
                    <th rowSpan={2}>%</th>
                    <th rowSpan={2}>Eur</th>
                    <th id="rules-table-source-header" className="text-center" colSpan={2}>Jako</th>
                </tr>
                <tr>
                    <th id="rules-table-source-header" >Lähde</th>
                    <th id="rules-table-source-header" >100%</th>
                </tr>
            </thead>
            <tbody>
                {Array.from(allocationRules.values(),
                    (allocationRule: AllocationRule) =>
                        renderTableRow(allocationRule)
                )}
                <tr>
                    <td colSpan={5 + dimensions.length}>
                        <button title="Uusi yvörytyssääntö" onClick={createNewAllocationRule} type="b
                            <Icon icon={'plus'} className={'text-default fa-sm'}>  LISÄÄ</Icon>
                        </button>
                    </td>
                </tr>
            </tbody>
        </Table>
    )
}

interface AllocationRulesProps {
    allocationRules: Map<number, AllocationRule>;
    dimensions: Array<EntityType & Record<string, any>>;
    createNewAllocationRule: () => void;
}

export default AllocationRules
```

Kuvio 11. Komponentin datan tyypitys

Rajapinnat

TypeScriptiä käyttämällä voidaan myös vaivattomasti pienentää riskiä siitä, että rajapinnan läpi ja mahdollisesti tietokantaan asti pääsisi virheellistä dataa. Ohjelman etu- ja takapään välissä liikkuva data on kuvattu API-tiedostossa, johon liikkuvan datan tarkoitetut tyypit voidaan kuvata samaan tapaan kuin komponenteissakin.

Kuviossa 12 on edellä mainittu API-tiedosto, ja siinä on esitelty kontrollerireitit, mihin dataa viedään. Näiden alla on *frontendin* käyttämiä funktioita, joita *React*-ohjelma

kutsuu, ja niiden vastaanottama data on tyypitetty, jotta voidaan varmistua siitä, minkälaista tietoa rajapinnan yli ollaan lähettämässä.

```
const _saveAccount = routeFetch('prowess_etoimisto_accounting_allocation_saveaccount', 'POST')
const _saveCostCentre = routeFetch('prowess_etoimisto_accounting_allocation_savecostcentre', 'POST')
const _saveRule = routeFetch('prowess_etoimisto_accounting_allocation_saverule', 'POST')
const _saveRunAllocation = routeFetch('prowess_etoimisto_accounting_allocation_run', 'POST')

export const copyToNewTemplate = data => _copyToNewTemplate({}, data)
export const createNewAccount = (template: number) => _createNewAccount({template})
export const createNewCostCentre = (template: number, dimension: number) => _createNewCostCentre({templ
export const createNewRule = (template: number) => _createNewRule({template})
export const createNewTemplate = () => _createNewTemplate()
export const deleteTemplate = (template: number) => _deleteTemplate({template})
export const deleteAccount = (account: number) => _deleteAccount({account})
export const deleteCostCentre = (costCentre: number) => _deleteCostCentre({costCentre})
export const deleteRule = (rule: number) => _deleteRule({rule})
export const fetchAllocations = (page: number, template: number | null) => _fetchAllocations({template,
export const fetchAllocationOptions = () => _fetchAllocationOptions({})
export const fetchAllocationVouchers = (allocation: number) => _fetchAllocationVouchers({allocation})
export const fetchTemplateContent = (template: number) => _fetchTemplateContent({template})
export const saveAccount = (account: AllocationAccount) => _saveAccount({}, account)
```

Kuvio 12. Tyypitetyt frontend API-funktiot

Kuviossa 13 näkyy *frontendin* käsittelijäfunktio *saveData*, joka ohjaa datan oikeaan väylään rajapinnassa. Esimerkiksi kuviossa 12 esitetty funktio *saveAccount* vaatii, että sille annetun muuttujan *account*-tyyppi on *AllocationAccount*, joten kuviossa 10 näkyvä funktiokutsu *api.saveAccount(action.allocationAccount)* osaa nyt varoittaa, jos sille annettu muuttuja on väärää tyyppiä.

```
function* saveData(action) {
    switch (action.type) {
        case types.SAVE_ALLOCATION_TEMPLATE:
            yield* apiWorker(
                () => api.saveTemplate(action.allocationTemplate),
                actions.receiveSaveAllocationTemplate)
            break
        case types.SAVE_ALLOCATION_ACCOUNT:
            yield* apiWorker(
                () => api.saveAccount(action.allocationAccount),
                actions.receiveSaveAllocationAccount)
            break
        case types.SAVE_ALLOCATION_COST_CENTRE:
            yield* apiWorker(
                () => api.saveCostCentre(action.allocationCostCentre),
                actions.receiveSaveAllocationCostCentre)
            break
        case types.SAVE_ALLOCATION_RULE:
            yield* apiWorker(
                () => api.saveRule(action.allocationRule),
                actions.receiveSaveAllocationRule)
            break
    }
}
```

Kuvio 13. Frontendin API:n hallintafunktioita

6.4 Työn viimeistely

Ohjelman ensimmäisen käyttökelpoisen version valmistuttua alkoi perinteinen koodinkatselmointi, eli kollega silmäilee koodin läpi ja merkitsee ylös löytämänsä virheet ja epäkohdat. TypeScriptin tuoma selkeys heijastui odotetusti myös katselmointiin, sillä koodia tuntematon henkilö näki selkeämmin ja nopeammin, mitä koodissa kuuluisi tapahtua, eikä muuttujien tyyppiä tarvinnut seurata tiedostosta toiseen. Toisin sanoen luottamus koodin eheyteen kasvoi, ja täten ohjelman viimeistelyn vaiheet myös sujuivat vikkelämmin eikä katselmoinnissa juurikaan ilmennyt huomautettavaa ainakaan niiltä osin, missä TypeScriptiä käytettiin.

7 Tutkimustulokset

TypeScriptin tyypitysten tuoma muuttujien ja olioiden ominaisuuksien päättely ja ennakointi auttaa keskittymään olennaiseen eli ohjelman rungon, komponenttien ja logiikan rakentamiseen. Muuttujien ennakointi ja tyypitys vähentävät huomattavasti datan liikuttelusta aiheutuvaa turhan työmuistin kuormitusta. JavaScriptillä koodatessa liikuteltavan datan tyypit pitää joko muistaa tai tarkistaa, mikä johtaa tiedostojen välillä hyppimiseen. Tämä taas rikkoo flow-tilaa ja vie turhaan työaikaa. TypeScriptin käyttäminen lähes poistaa tämän taakan, sillä ohjelmoija voi luottaa työkalujensa huolehtivan siitä. Toki vastuu työkalujen opettamisesta, esimerkiksi datamallien kuvauksilla ja tyyppideklaraatioilla, on edelleen ohjelmoijalla, mutta se pitää tehdä vain kerran hyvin, ja sen hyöty näkyy loppuun asti.

Myös pidemmän kantaman hyödyt ovat erinomaisia TypeScriptissa. Jatkokehityksen helpotukset ovat niistä selkeä osoitus, mutta myös yritysten vastuuta asiakkaiden tietoja kohtaan voidaan suoraan tehostaa, kun rajapintojen läpi kulkevalle datalle saadaan tyyppitarkistusten avulla vaivattomasti yksi suojakerros lisää tyypillisten kontrollerin validointien lisäksi.

Tutkimus vahvistaa laajalti esitettyä mielipidettä siitä, että TypeScript on loppujen lopuksi tehokkaampi kieli etenkin laajemmissa ohjelmistokokonaisuuksissa. Vaikka JavaScript on kätevämpi ja ketterämpi silloin, kun halutaan tehdä nopeasti pieniä komponentteja, sen verrattava käyttöhyöty pienenee huomattavasti, jos ohjelmoija hallitsee jo TypeScriptin ja TypeScriptille on olemassa tuki ja asetukset valmiina. Molemmissa ohjelmointikielissä on selvästi omat heikkoutensa ja vahvuutensa, mutta kokonaiskuva vaikuttaisi suosivan TypeScriptia.

Taulukossa 1 on vielä havainnollistettu vertailtavien ohjelmointikielien keskeisimmät erot.

Taulukko 1. JavaScriptin ja TypeScriptin keskeisimmät erot.

	JavaScript	**TypeScript**
Tuki ja käyttö	Tuettu lähestulkoon kaikissa selaimissa ja ympäristöissä. Työkaluksi riittää esimerkiksi tekstieditori.	Kääntyy JavaScriptiksi, joten tuettu käytännössä JavaScriptin lailla. Työkaluksi vaaditaan kehittynyt ohjelmointiympäristö.
Suorittaminen	Ei vaadi kääntämistä ja kaikista moderneista selaimista löytyy tulkki kielelle.	Kieli pitää kääntää ennen suorittamista, jolloin lopputulos on puhdasta JavaScriptiä.
Tyypittäminen	Dynaamisesti tyypitetty ja täten koodin eheyden ylläpito vaatii jatkuvaa huolellisuutta ohjelmoijalta.	Staattisesti tyypitetty; datan tyyppimääritykset vähentävät huomattavasti ohjelmoijan taakkaa koodin eheyden ylläpidon suhteen.
Virheenseulonta	Käytännössä melkein kaikki virheet huomataan vasta ohjelmaa suoritettaessa, mikä usein tekee virheenkorjauksesta hidasta ja hankalaa sekä altistaa virheiden lumipalloefektille.	Virheet huomataan jo kääntäjässä eli virheenseulonta on reaaliaikaista ja hyvin spesifioitua ja täten virheenkorjaus on nopeaa ja vaivatonta sekä estää virheiden kumuloitumista.

8 Pohdinta ja johtopäätökset

Ohjelmointikieli TypeScript oli minulle entuudestaan täysin tuntematon, joten jouduin opettelemaan sitä ohjelmointityön ohessa. Uskon, että tämä kuitenkin mahdollisti ennakkoluulottoman sekä tuoreen lähestymisen tekniikoiden vertailulle ja, koska kielet ovat kuitenkin lähes identtisiä periaatteiltaan, uuden opettelu ei kuitenkaan vienyt fokusta itse tutkimukselta.

Vaikka TypeScriptin tuomat tyypitysmahdollisuudet ovat hoidettavissa myös osittain hyödyntäen JavaScriptin luokkia sekä ulkoisilla kirjastoilla, kuten JavaScriptille tehdyn tyyppitarkistajan nimeltä *Flow* kanssa, TypeScript tuntuu silti huomattavasti selkeämmältä ja koherentimmalta vaihtoehdolta. Ylimääräiset kirjastot tuovat aina muka-

naan omat säätönsä ja saattavat aiheuttaa turhia ylimääräisiä ongelmia, joten miksi käyttää niitä, jos voi saada kaiken yhdessä paketissa TypeScriptin kanssa?

Alussa esitettyihin kahteen tutkimuskysymykseen löytyivät myös melko selkeät vastaukset.

Voiko JavaScriptistä TypeScriptiin vaihtamalla tehostaa työn laatua ja/tai nopeutta ja jos, niin mitkä tekijät vaikuttavat vaihtamisen kannattavuuteen?

Ensimmäiseen kysymykseen vastaisin myöntävästi, sillä jo ensimmäinen TypeScriptillä koodaamani ohjelma osoitti, että kun koodin sisältö on tyypitettyä, sen hallitseminen ja hahmottaminen on helpompaa, ja täten myös työn laatu on alusta asti parempaa. Vaikka koodaaminen on datakuvausten vuoksi aluksi hieman tavallista hitaampaa, siihen käytetty aika maksaa itsensä takaisin, mitä pidemmälle työ etenee, sillä ilman datakuvauksien tuomaa ennakoitua syötettä aikaa kuluisi turhaan tarkisteluun. Näin ensimmäistä kertaa kieltä käyttäessä myös komponenttien ja funktioiden tyyppien merkitsemiset pidensivät ehkä marginaalisesti työn kestoa, mutta on helposti pääteltävissä, että tämä johtuu uuden asian opettelusta ja muuttuu nopeasti triviaaliksi vaikutukseksi.

Edellä mainittujen asioiden merkittävyyteen taas vaikuttaa ohjelman laajuus. Mitä suurempi ohjelmoitava kokonaisuus on, sitä levittäytyneempää on myös siinä liikkuva data. Monimuotoisen ja levittäytyneen datan käsittely taas kuormittaa ohjelmoijan työmuistia, mikä heikentää työn tehokkuutta. Toisin sanoen hyvin pienissä ohjelmissa TypeScriptin tuomat hyödyt eivät ole yhtä korostuneet ja selkeät, jolloin JavaScriptin käyttö saattaa myös olla suositeltavampaa.

Tuleeko koodista luettavampaa, ja mitkä ovat vaikutukset jatkokehitykseen?

Toisessa kysymyksessä päädyin myös enimmäkseen myöntävään lopputulokseen. Kun komponenteissa liikuteltava data ja sen tyypit ovat nähtävissä suoraan samasta paikasta, koodin seuraaminen on koherentimpaa ja soljuvampaa. Eri tiedostojen välillä hyppiminen vähenee huomattavasti, eikä ohjelman logiikkaa seuratessa tarvitse

muistaa läheskään yhtä montaa asiaa. Toisaalta tyypitykset tuovat mukanaan myös lisää koodirivejä, mikä on yleisesti ohjelmoinnissa minimoitava asia, joten sen suhteen JavaScript on minimalistisuutensa vuoksi kompaktimpaa ja tietyssä mielessä esteettisempää luettavaa. Sanoisin kuitenkin, että ylimääräiset koodirivit ovat hyvin oikeutettuja ja niiden tuomat hyvät puolet päihittävät huonot puolet.

TypeScriptin jatkokehitykseen tuomat vaikutukset ovat tässä vaiheessa vain tutkimuksesta johdettuja spekulaatioita, sillä tutkimusta varten tehtyä ohjelmaa ei ole vielä alettu jatkokehittämään, mutta ohjelmaa tehdessä ilmenneet vahvuudet voidaan käytännössä suoraan soveltaa myös jatkokehitykseen. Toisin sanoen koska koodi on luettavampaa ja koherentimpaa, on myös erittäin todennäköistä, että ohjelmaa ennestään tuntemattoman ohjelmoijan on helpompi ja ennen kaikkea nopeampi alkaa tekemään koodiin myös muutoksia. Suurin negatiivinen vaikutus jatkokehitykseen ilmenee siinä vaiheessa, jos ohjelmassa liikuteltavaan dataan tehdään muutoksia – etenkin tietokantatauluihin –, sillä se tarkoittaa sitä, että myös käyttöliittymässä kuvatut datamallit on päivitettävä. Tätä haittaa voidaan toisaalta myös ennakoivasti helpottaa jo ohjelman suunnitteluvaiheessa varmistamalla, että datana tyypit olisi kuvattu mahdollisimman keskitetysti. Ihanteellista olisikin, että kuvaukset olisivat vain yhdessä paikassa ja ohjelmassa viitataan aina niihin. Toinen negatiivinen vaikutus jatkokehitykseen tuntuisi muodostuvan, jos jatkokehitystä tekevä ohjelmoija ei tuntisi itse entuudestaan TypeScriptiä. Toisaalta tämä potentiaalinen heikkous on vastassa lähes kaikkien ohjelmointikielten kohdalla eikä sitä siksi voida oikeastaan kutsua TypeScriptin vaikutukseksi.

Ohjelmointi on ennen kaikkea henkisesti väsyttävää työtä, ja kaikki keinot, joilla sitä taakkaa pystytään pienentämään, ovat mielestäni hyödyntämisen arvoisia, sillä kuormittuneet aivot tekevät työstä epämielyttävää ja tehottomampaa. TypeScriptin käyttäminen tuo helpotusta moniin hyvin konkreettisiin sekä inhimillisiin ongelmiin, ja näin ollen voidaan keventää tätä taakkaa.

Lähteet

Bister, T. 2019. Tietojenkäsittelyn opinnäytetyö: Viittoja ja karttoja tutkimisen ja kehittämisen teille. Jyväskylän ammattikorkeakoulun julkaisuja 272. Viitattu 24.9.2020. http://ezproxy.jamk.fi:2048/login?url=https://www.booky.fi/lainaa/1144.

Concepts. N.d. Webpack.js.org-verkkosivusto. Viitattu 24.8.2020. https://webpack.js.org/concepts/.

Ghate, S. 2017. Why TypeScript Is Better Than JavaScript? Medium.com-verkkosivusto. Viitattu 21.10.2020. https://medium.com/udgama-engineering/why-typescript-is-better-than-javascript-b40bba96751f.

JavaScript data types and structures. N.d. Developer.mozilla.org-verkkosivusto. Viitattu 26.10.2020. https://developer.mozilla.org/en-S/docs/Web/JavaScript/Data_structures.

JavaScript Overview. N.d. W3schools.in-verkkosivusto. Viitattu 21.1.2020. https://www.w3schools.in/javascript-tutorial/overview/.

Learn JavaScript Tutorial. N.d. Javapoint.com-verkkosivusto. Viitattu 14.10.2020. https://www.javatpoint.com/javascript-tutorial.

PhpStorm: The Lightning-Smart IDE for PHP Programming by JetBrains. N.d. Jetbrains.com-verkkosivusto. Viitattu 13.10.2020. https://www.jetbrains.com/phpstorm/.

Probis. N.d. Probis.fi-verkkosivusto. Viitattu 28.2.2020. https://www.probis.fi/.

Ramos, D. 2020. Is TypeScript Better Than JavaScript? Acidtango.com-verkkosivusto. Viitattu 21.10.2020. https://acidtango.com/thelemoncrunch/is-typescript-better-than-javascript/.

ReactJS Overview. N.d. Wikitechy.com-verkkosivusto. Viitattu 24.8.2020. https://wikitechy.com/tutorials/react/what-is-react-js.

Rippon, C. 2020. Controlling Type Checking Strictness in TypeScript Carlrippon.com-verkkosivusto. Viitattu 20.10.2020. https://www.carlrippon.com/controlling-type-checking-strictness-in-typescript.

Singh Rao, V. 2018. Static Typed vs Dynamic Typed Programming Languages Technotification.com-verkkosivusto. Viitattu 5.10.2020. https://www.technotification.com/2018/09/static-vs-dynamic-typed-programming-languages.html.

TypeScript: Handbook – TypeScript for JavaScript Programmers. N.d. Typescriptlang.org-verkkosivusto. Viitattu 26.10.2020. https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.

TypeScript – Overview. N.d. Tutorialspoint.com-verkkosivusto. Viitattu 26.10.2020. https://www.tutorialspoint.com/typescript/typescript_overview.htm.

TypeScript vs JavaScript: What's the Difference? N.d. Guru99.com-verkkosivusto. Viitattu 27.1.2020. https://www.guru99.com/TypeScript-vs-javascript.html.

TypeScript: Why does TypeScript exist? N.d. Typescriptlang.org-verkkosivusto. Viitattu 12.8.2020. https://www.typescriptlang.org/why-create-typescript.

What is AngularJS. N.d. Javatpoint.com-verkkosivusto. Viitattu 21.8.2020. https://www.javatpoint.com/what-is-angularjs.