Tampereen yliopisto

Sami Aalto

ESISELVITYS JSF-SOVELLUKSEN UUDISTAMISESTA YHDEN SIVUN SOVELLUKSEN TEKNOLOGIOILLA

Diplomityö
Tietotekniikan tiedekunta
Tarkastajat: Terhi Kilamo, Kari Systä
Maaliskuu 2021

TIIVISTELMÄ

Sami Aalto: Esiselvitys JSF-sovelluksen uudistamisesta yhden sivun sovelluksen teknologioilla
Diplomityö
Tampereen yliopisto
Tietotekniikka, DI
Maaliskuu 2021

Tässä työssä tehdään esiselvitystä sille, miten JavaServer Faces (JSF) -pohjaisen järjestelmän käyttöliittymän saa päivitettyä käyttämään yhden sivun sovelluksen teknologioita. Migraatiokeinona tutkitaan erityisesti vaiheistettua migraatiota, jossa vanha ja uusi teknologia muodostavat migraation aikana hybridijärjestelmän. Hybridijärjestelmässä osa sivuista käyttää vanhaa ja osa uutta teknologiaa.

Esiselvityksen tavoitteena on tutkia, miten JSF-toteutus voidaan siirtää yhden sivun sovelluksen teknologioille. Integraatiossa tutkitaan sekä eri teknologian sivujen välistä integraatiota että kommunikaatiota Java EE -palvelimen kanssa.

Esiselvityksessä migraatiokeinoja tutkitaan käytännönläheisesti kahden prototyypin avulla. Prototyypit toteutetaan eri teknologioilla ja eri integraatiolähestymisillä. Ensimmäinen prototyyppi toteutetaan Angularilla siten, että Angular-alijärjestelmä on itsenäinen yhden sivun sovellus. Toinen prototyyppi toteutetaan Reactilla siten, että jokaisella sivulla on oma React-sovellus, joka alustetaan JSF-sivun sisällä.

Prototyypeissä toteutetaan pohja migraation tekemiselle ja siirretään yhden näkymän toiminnallisuus vanhasta järjestelmästä. Lisäksi tarkastellaan yleisellä tasolla, miten JSF-toiminnallisuuden voi siirtää uusille teknologioille ja mitä osia tästä voidaan automatisoida.

Prototyyppien teon aikana löydetään keinoja JSF-logiikan siirtämiseksi ja kehitetään automatisointityökaluja, jotka tekevät osan näkymäkoodin koodimuunnoksista automaattisesti. Kehitellyt työkalut koetaan hyväksi pohjaksi migraatioiden tekemiselle.

Prototyyppien molemmat teknologiat ja integrointimenetelmät koetaan mahdollisiksi migraatioratkaisuiksi. Angular on ominaisuuksiltaan ja rakenteeltaan hieman lähempänä JSF-teknologiaa. Angular-prototyypin lähestymisen koetaan soveltuvan hieman paremmin laajaan migraatioon, jossa koko järjestelmä halutaan vaihtaa yhden sivun sovellukseksi. Se kuitenkin vaatii enemmän työtä varsinkin migraation alkuaikoina. React puolestaan koetaan soveltuvan paremmin pienemmän tason migraatioon, jossa uutta teknologiaa halutaan vain järjestelmän tiettyihin osiin. Vaikka React ei suoraan tarjoa kaikkia identtisiä ominaisuuksia kuin JSF, ne pystyy toteuttamaan itse tai apukirjastojen avulla.

Avainsanat: JSF, SPA, Yhden sivun sovellus, Ohjelmistomigraatio

ABSTRACT

Sami Aalto: Preliminary study on modernisation of JSF application with single-page application technologies
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
March 2021

In this thesis migration methods are investigated for updating a JavaServer Faces (JSF) based system with Single-page application (SPA) technologies. One particular migration method under investigation is phased interoperability migration where old and new technologies form a hybrid system. During migration some of the pages in the system use old technologies and some use the new ones.

The goal of the study is to investigate how JSF application can be migrated to SPA technologies and how the technologies can be integrated with each other. Both the integration between pages of different technologies and the integration betwen single-page application pages and the Java EE backend are investigated.

Migration methods are investigated on practical level with two prototypes. Prototypes use different technologies and different integration approaches. First prototype is implemented with Angular and uses Angular as mostly independent single-page application. The second prototype uses React and instead for each page has individual React applications which are initialised inside of JSF pages.

In the prototypes the migration foundations are implemented and a single page from the inspected system is migrated to the new technology. In addition, it is analysed on more general level how JSF functionality can be migrated into the new technologies and which parts of it can be automatised.

During the study both prototypes are implemented successfully. Methods for JSF migration are found and helpful automatisation tools are implemented for automatising some of the code transformations. These tools are helpful in reducing manual work during the migration.

At the end of the study both technologies and integration methods are deemed to be possible solutions for the migration. In terms of functionality, Angular is slightly closer to JSF and seems to be slightly better suited for a large-scale migration, which attempts to update the entire system. It however requires more work in the early stages of migration. React on the other hand seems to be better suited for a smaller scale migration, where only parts of the application are updated. Even though React does not directly provide all the same features as JSF, they can be easily implemented.

Keywords: JSF, SPA, Single page application, Software migration

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

camelCase	Nimeämiskäytäntö, jossa monisanaisissa nimissä sanat erotetaan toisistaan isoilla alkukirjaimilla.
CORS	Cross-origin resource sharing. Mekanismi, joka sallii web-resurssien pyytämisen muilta domaineilta.
CSS	Cascading Style Sheets. Käytetään WWW-sivustojen tyylillisen ulkoasun määrittämiseen.
CSS-in-JS	CSS-tyylisääntöjen määrittäminen JavaScript-koodin kautta.
Dekoraattori	Dekoraattorit (decorator) ovat funktioita, jotka täydentävät olion käyttäytymistä.
DOM	Dokumenttioliomalli (Document object model). Kuvaa esimerkiksi HTML-kaltaisia rakenteita oliopohjaisena puurakenteena, jota voi hallinnoida ohjelmointirajapintojen kautta.
ECMAScript	JavaScript-ohjelmointikielen standardi, johon JavaScript-toteutukset pohjautuvat.
HTML	Hypertext Markup Language. WWW-sivustojen sisällön määrittämisessä käytetty kuvauskieli.
Iframe	Selain-komponentti, jolla voi sisällyttää web-sivun sisällön toisen sivun sisälle.
Java EE	Java Platfom, Enterprise Edition. Sovelluskehysalusta Java-sovelluksien tekemiseen.
Java Servlet	Java EE -standardin mukainen sovelluskomponentti, joka käsittelee palvelimen saamia kyselyitä.
JavaBean	Uudelleen käytettävä Java-luokka, jota voi käyttää tietojen käsittelyyn.
JavaScript	Web-ohjelmoinnissa käytetty ohjelmointikieli, jolla voi toteuttaa selainsovelluksien toimintalogiikkaa. Nykyisin käytössä myös muualla, kuten palvelinohjelmissa.
JAX-RS	Java API for RESTful Web Services. Java EE -rajapintastandardi.
JAXB	Java Architecture for XML Binding. Tarjoaa työkalut XML- ja Java-luokkien välisiin muunnoksiin.

JMS	Java Messaging Service. XML-pohjainen viestijono.
JSF	JavaServer Faces. Java-pohjainen spesifikaatio komponenttipohjaisten käyttöliittymien toteutukseen.
JSON	JavaScript Object Notation. Tekstimuotoinen tiedonvälitysstandardi.
JSX	JavaScriptiä täydentävä XML-kaltainen syntaksi, joka voidaan kääntää tavalliseksi JavaScriptiksi. Käytetään esimerkiksi React-sovelluksissa.
Kaksisuuntainen sitominen	Two-way binding. Mekanismi, joka synkronoi kahteen arvoon tulevat muutokset keskenään molempiin suuntiin. Käytetään tyypillisesti mallin ja näkymän tilan synkronoinnissa.
Monen sivun sovellus	Multi-page application. Perinteinen web-sovellus, jossa jokaisella sivulla on oma html-tiedosto.
MVVM	ModelView-View-Model-arkkitehtuuri. Arkkitehtuurimalli, jossa arkkitehtuuri jakautuu näkymämalliin, näkymään ja malliin.
Nginx	Avoimen lähdekoodin WWW-palvelin, jota voi käyttää muun muassa reverse proxyna.
React hook	Funktio, jonka ominaisuuksia on täydennetty React-rajapintojen kautta. Mahdollistavat muun muassa tilan säilyttämisen ilman JavaScript-luokkia.
RegEx	Regular expression, säännöllinen lauseke. Lauseke, jonka avulla on mahdollista muun muassa löytää ja korvata halutun säännön mukaisia merkkijonoja tekstistä.
Reverse proxy	Välityspalvelin, jonka tehtävänä on välittää saamiaan kyselyitä oikeille palveluille.
Riippuvuusinjektio	Dependency injection. Mekanismi, jonka avulla on mahdollista saada tarvittava riippuvuus tietämättä. miten se luodaan.
SCSS	Sassy CSS. CSS-kieltä täydentävä kieli, joka lisää kieleen uusia ominaisuuksia. Kääntyy tavalliseksi CSS-koodiksi.
Singleton	Suunnittelumalli, jolla varmistetaan, että oliolla on sovelluksessa vain yksi instanssi.
SOA	Palvelukeskeinen arkkitehtuuri (Service Oriented Architecture).

SPA	Yhden sivun sovellus (Single-page application). Tapa toteuttaa web-sovelluksia, jossa selaimen JavaScript-logiikka hoitaa sivujen päivityksen dynaamisesti ilman, että sivua tarvitsee ladata kokonaan uudestaan.
TypeScript	JavaScript-kieleen pohjautuva ohjelmointikieli, joka tarjoaa vahvemman tyypityksen. Kääntyy tavalliseksi JavaScript-koodiksi.
WebLogic	Oraclen Java EE -sovelluspalvelin.
Webpack	JavaScript-paketointikirjasto.
XHTML	Extensible Hypertext Markup Language. XML-muotoinen HTML-tiedostoformaatti.
XML	Extensible Markup Language. Tekstimuotoinen tiedonvälitysstandardi.

1 JOHDANTO

Web-kehitys on viimeisen vuosikymmenen aikana kehittynyt merkittävää tahtia. Yksi suurimmista muutoksista on logiikan siirtyminen palvelimelta selaimiin. Tämän muutoksen myötä useat teknologiat ovat jääneet kehityksessä jälkeen, ja uusia on tullut paikalle. Monet vanhat järjestelmät käyttävät edelleen vanhoja teknologioita, mikä osaltaan voi hankaloittaa ylläpidettävyyttä sekä uutta kehitystyötä.

Vanhojen järjestelmien teknologioiden uudistaminen ei ole yksinkertainen tehtävä. Uudistamisessa tehtävä ohjelmistomigraatio riippuu vahvasti kohdejärjestelmästä, sen alkuperäisestä teknologiasta ja uudesta kohdeteknologiasta. Vaikka ohjelmistomigraatioista löytyy monenlaista tutkimusta, migraatioille ei yleensä löydy täysin valmiita ratkaisuja. Ohjelmistomigraation onnistumiseksi on hyvä tutkia ja testata migraation teknisiä ratkaisuja.

Tässä diplomityössä tehdään esiselvitystä Gofore Oyj:n asiakasprojektiin Suomen sosiaali- ja terveysministeriölle. Projektissa halutaan päivittää Java EE -pohjaisen järjestelmän Java ServerFaces (JSF) -käyttöliittymäteknologia modernimmaksi. Työn tutkimuskysymys on, miten tämänlainen uudistus voidaan tehdä hyödyntäen yhden sivun sovelluksien teknologioita (Single-page application, SPA). Tarkastelussa tutkitaan Angularia ja Reactia teknologiavaihtoehtoina. Toteutusta testataan käytännönläheisesti prototyypeillä. Tutkinnassa korostetaan erityisesti miten uuden teknologian saa integroitua vanhan järjestelmän kanssa ja miten vanhan toiminnallisuuden saa siirrettyä uudelle teknologialle. Lisäksi tutkitaan lyhyesti, mitä osia ohjelmistomigraatiosta voisi automatisoida. Vaikka tarkastelu tehdään tiettyä sovellusta silmällä pitäen, tehtyjä havaintoja voi hyödyntää muissa vastaavissa järjestelmissä.

Diplomityön rakenne on seuraavanlainen. Luvussa 2 tutustutaan järjestelmään, johon päivityksen esiselvitys tehdään. Samalla tutustutaan järjestelmän rakenteeseen, JavaServer Facesin toimintamalliin ja uudistuksen tavoitteisiin. Luvussa 3 tutustutaan aikaisempiin tapauksiin, joissa JSF-migraatioita tai -integraatioita on tehty. Luvussa 4 tutustutaan teoreettisella tasolla ohjelmistomigraatiomenetelmiin. Luvussa 5 tutustutaan yhden sivun sovelluksen teknologioihin sekä Java- ja TypeScript-ohjelmointikielien eroavaisuuksiin. Luvussa 6 mennään syvemmälle mahdollisiin menetelmiin, joilla yhden sivun sovelluksen teknologiat voi integroida olemassa olevan JSF-järjestelmän kanssa. Luvuissa 7, 8 ja 9 suunnitellaan ja toteutetaan kaksi prototyyppiä. Luvussa 10 koostetaan prototypoinnin tulokset ja tehdään analyysia ratkaisujen välillä. Lopuksi luvussa 11 tehdään yhteenveto työn tuloksista.

2 JÄRJESTELMÄ

Diplomityössä tarkasteltava sovellus on datapainotteinen tiedonhallintajärjestelmä. Sovelluksen pääominaisuuksiin kuuluvat tietojen syöttö, haku ja dokumenttien generointi. Järjestelmä on laaja ja käsittelee paikoitellen monimutkaisia suhteita syötettyjen tietojen välillä.

Järjestelmässä on useita käyttäjärooleja, jotka käyttävät järjestelmää hieman eri tarkoituksiin. Jotkin ominaisuudet ovat vain tiettyjen käyttäjäryhmien käytössä. Järjestelmään pääsee vain suljetusta verkosta, joten sen käyttäjäryhmä on hyvin rajattu.

Järjestelmä käyttää Java EE -alustaa. Sovelluspalvelimena on Oracle WebLogic Server ja käyttöliittymäteknologiana on JavaServer Faces (JSF). JSF-teknologia on ajan myötä tullut hankalammaksi ylläpitää ja se tuo mukanaan aikansa teknisiä rajoitteita, minkä vuoksi tutkitaan vaihtoehtoja sen vaihtamiseksi johonkin uudempaan.

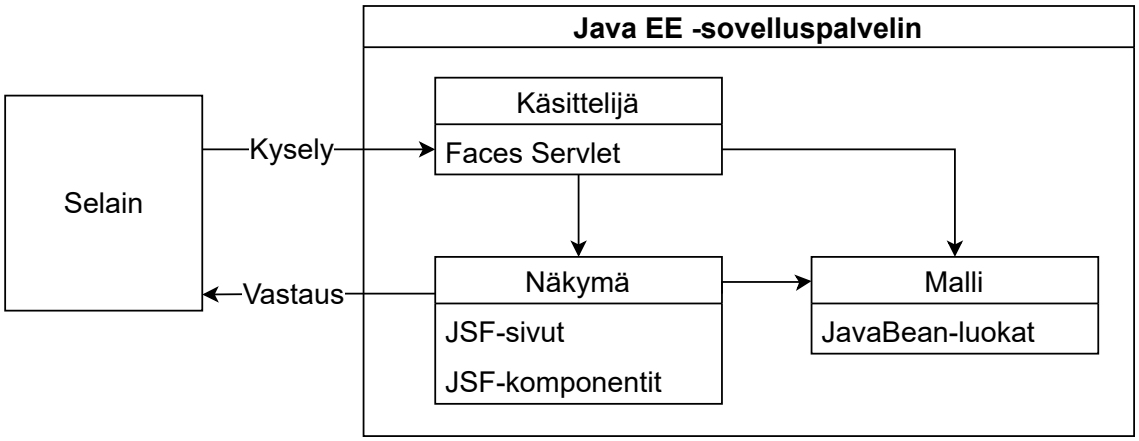
Luvussa 2.1 käydään yleisellä tasolla läpi JSF:n toimintaperiaatteita. Luvussa 2.2 katsotaan tarkasteltavan järjestelmän rakennetta korkealla tasolla ja luvussa 2.3 tutustutaan astetta syvemmälle esityskerroksen toimintaan. Lopulta luvussa 2.4 katsotaan ohjelmistomigraation tavoitteita tämän järjestelmän näkökulmasta.

2.1 JavaServer Faces -sovelluskehys

JavaServer Faces on Javan web-sovelluksissa yleisesti käytetty sovelluskehys. Sen alkuperäinen määrittely julkaistiin vuonna 2004 ja se siirtyi vuonna 2006 osaksi Java EE -standardia. [1]

JSF on vastuussa käyttöliittymän esityksestä sekä käyttöliittymälogiikasta. Kuten monet nykypäivän sovelluskehykset, se on komponenttipohjainen. Toisin sanoen JSF:lle on mahdollista tehdä helposti uudelleen käytettäviä käyttöliittymäkomponentteja. JSF:lle löytyy useita valmiita käyttöliittymäkirjastoja, kuten PrimeFaces ja OmniFaces [2, 3].

Nykypäivän yhden sivun sovelluksien sovelluskehyksistä poiketen JSF-logiikka on pääosin palvelimen puolella. Kaikki näkymät renderöidään palvelimella. Tästä huolimatta JSF:llä voidaan toteuttaa interaktiivisia näkymiä hyödyntämällä AJAX-kyselyitä. Kun käyttöliittymässä tapahtuu tapahtuma, kuten napin painallus, tieto tapahtumasta välittyy palvelimelle AJAX-kyselyllä. Palvelin vastaanottaa tapahtuman, päivittää käyttöliittymän mallia tarvittaessa, renderöi käyttöliittymän HTML-koodin ja antaa vastauksena muutokset. Selaimen käyttöliittymä osaa tämän perusteella päivittää tiedot ajan tasalle.



Kuva 2.1. *JSF-arkkitehtuurin MVC-esitys. [4]*

JSF käyttää MVC-arkkitehtuurimallia, jossa arkkitehtuuri jakautuu malliin, näkymään ja käsittelijään (Model, View, Controller) [4]. JSF:n MVC-toteutus on havainnollistettu kuvassa 2.1.

Sovelluksen käsittelijänä on Faces Servlet, joka vastaa sovelluksen ajamisesta ja kytkee näkymän ja mallin keskenään. Faces servlet muun muassa hoitaa tulevien HTTP-kyselyjen käsittelyn. Näkymä koostuu JSF-sivuista, jotka voivat koostua JSF-komponenteista. Sivujen sisältö esitetään XHTML-pohjaisella mallinekielellä. Sovelluksen varsinainen toimintalogiikka on JavaBean-luokissa, jotka toimivat sovelluksen mallina. JavaBeanit ovat Java-luokkia, joille sovelluskehys tarjoaa lisäominaisuuksia, kuten esimerkiksi tuen riippuvuusinjektiolle. JSF-sivujen XHTML-tiedostoissa voidaan viitata JavaBean-luokkien rajapintoihin dynaamisen toiminnallisuuden mahdollistamiseksi. [4, 5]

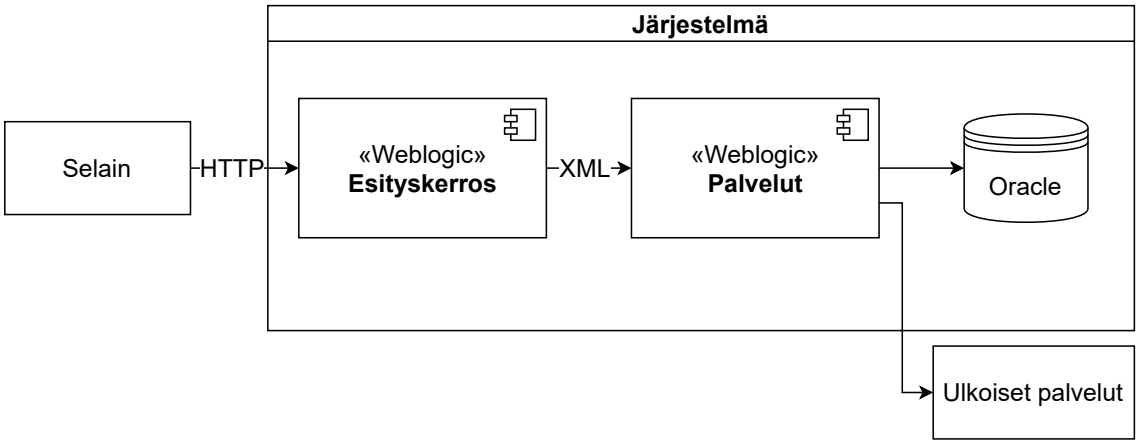
2.2 Korkean tason arkkitehtuuri

Tarkasteltava järjestelmä käyttää palvelukeskeistä arkkitehtuuria (Service Oriented Architecture, SOA). Järjestelmä jakautuu tasoittain kolmeen osaan: esityskerros (käyttöliittymä), palvelut (liiketoimintalogiikka) ja Oracle-tietokanta (tietovarasto). Korkean tason arkkitehtuuri on havainnollistettu kuvassa 2.2. Nuolilla näytetään komponenttien väliset riippuvuudet.

Esityskerros, palvelut ja Oracle-tietokanta ovat erotettuja siten, että ne voivat sijaita fyysisesti eri palvelimilla. Sekä esityskerros että palvelut-komponentti ovat WebLogic-sovelluspalvelimella ajettavia Java-sovelluksia.

Palvelut-komponentti sisältää sovelluksen olennaisimman liiketoimintalogiikan. Tähän sisältyvät muun muassa toimenpiteiden validointi, tietokantaoperaatiot, kommunikaatio ulkoisten palvelujen kanssa ja dokumenttien generointi. Koska ohjelmistomigraatio keskittyy käyttöliittymään, palvelut-komponentti voidaan tämän työn osalta tulkita mustana laatikkona.

Esityskerros on vastuussa käyttöliittymälogiikasta. Esityskerros toimii loppukäyttäjien nä-



Kuva 2.2. *Kuvaaja järjestelmän korkean tason arkkitehtuurista.*

kökulmasta web-palvelimena, joka tarjoaa järjestelmän staattiset ja dynaamiset tiedostot sekä muut mahdolliset rajapinnat. Tämän lisäksi esityskerros on vastuussa käyttäjien autentikoinnista. Esityskerroksella ei ole suoraa pääsyä tietokantaan tai ulkoisiin palveluihin, minkä vuoksi sen pitää tehdä kaikki toimenpiteet palvelut-komponentin rajapintojen kautta.

Esityskerros ja palvelut-komponentti kommunikoivat Java Messaging Service -viestijonojen avulla (JMS). Viestit ovat XML-muotoisia ja ne generoidaan JAXB-luokkien avulla (Java Architecture for XML Binding). JAXB:ssa XML-viesteille määritellään XML-muotoinen malli, jonka pohjalta generoidaan automaattisesti vastaavat Java-luokat. Java-luokat voidaan automaattisesti muuntaa XML-muotoon ja toisin päin. [6] Yksi XML-viesti voi viitata muihin XML-määrittelyihin, mikä mahdollistaa isompien viestien pilkkomisen uudelleen käytettäviksi komponenteiksi.

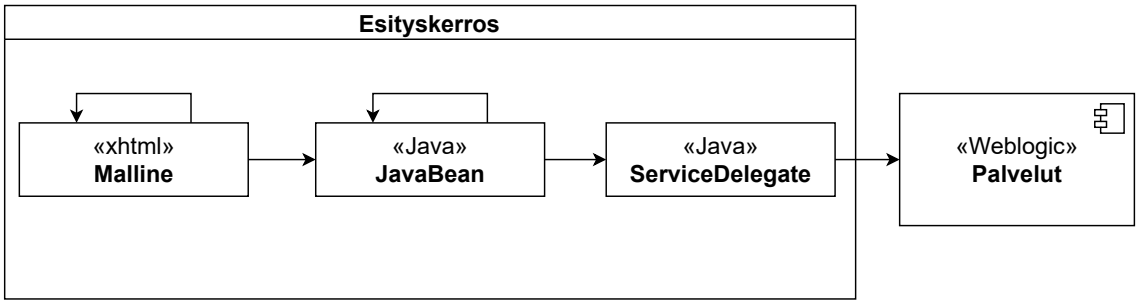
2.3 Esityskerroksen arkkitehtuuri

Esityskerros on pääosin vastuussa sivujen renderöinnistä. Tämän lisäksi esityskerros on vastuussa käyttäjäistunnon tilanhallinnasta, validoinnista ja kommunikaatiosta palvelut-komponentin kanssa.

Järjestelmän näkymien kooditoteutuksilla on säännölliset perusrakenteet. Kuvaajassa 2.3 näytetään olennaisimmat komponentit yksittäisen sivun renderöinnin näkökulmasta. Näytettyjen komponenttien lisäksi järjestelmässä on myös monia muita apuluokkia, mutta näytetyt esiintyvät jokaisen näkymän logiikassa ja kuvaavat hyvin kommunikaation suuntaa.

Jokaisen näkymän sisältö määrätään JSF-tiedostoissa, jotka tässä järjestelmässä ovat XHTML-pohjaisia mallinetiedostoja. Yksi tiedosto voi viitata muihin tiedostoihin. Esimerkiksi monet yleisesti käytetyt komponentit on määritelty omissa tiedostoissaan.

Jokainen JSF-tiedosto voi viitata yhteen tai useampaan JavaBean-luokkaan. Ne toimivat tietomalleina JSF-tiedostoille ja myös sisältävät kaiken esityskerroksen toimintalogiikan.



Kuva 2.3. *Kaavio esityskerroksen yleisimpien komponenttien kommunikaatiosta yhden näkymän näkökulmasta.*

Koska suuri osa liiketoimintalogiikasta on palvelut-komponentissa, tämä on pääosin validointilogiikkaa ja dataan liittyvää esi- tai jälkikäsittelyä.

JavaBean-luokkien olioilla voi olla eri mittaisia elinkaaria. Jotkin kestävät vain yksittäisen näkymän ajan, kun taas toiset voivat kestää koko käyttäjäistunnon. Järjestelmässä yleisimpiä ovat yksittäisen näkymän tilaa ylläpitävät ViewScoped JavaBeanit ja käyttäjäistunnon mittaiset SessionScoped JavaBeanit. Koko sovelluksen käynnissäolon aikaisia ApplicationScoped JavaBeaneja käytetään välillä välimuistitarkoituksiin. Niitä esimerkiksi käytetään yleisten tietokantahakujen optimointiin, joiden tulokset eivät yleensä muutu.

ServiceDelegate-luokat toimivat siltana palvelut-komponentin rajapintoihin. Jokainen ServiceDelegate-luokka vastaa tietyn vastuualueen rajapinnoista. Esimerkiksi EmployerServiceDelegate-luokka on vastuussa järjestelmään tallennettuihin työnantajiin liittyvistä rajapinnoista. ServiceDelegate-luokat ovat logiikaltaan yleensä yksinkertaisia ja toimivat vain abstraktiokerroksena palvelut-logiikan edessä. Kaikki palvelut-komponenttiin menevä liikenne menee ServiceDelegate-luokkien kautta. JavaBean-luokat saavat pääsyn ServiceDelegate-luokkiin riippuvuusinjektiolla.

2.4 Ohjelmistomigraation tavoitteet ja vaatimukset

Järjestelmä on hieman yli 10 vuotta vanha, ja tuona aikana web-teknologiat ovat kehittyneet merkittävästi. JSF on ollut aikoinaan suosittu menetelmä sovelluksien tekemiseen, mutta nykypäivänä sen suosio on merkittävästi pienempi verrattuna suosituimpiin www-teknologioihin.

Järjestelmää halutaan ylläpitää pitkälle tulevaisuuteen, minkä takia halutaan tutkia mahdollisuutta päivittää järjestelmän käyttöliittymäteknologia uudempaan. Isoimmat motivaatiot ovat järjestelmän ylläpidon helpottaminen, ylläpidettävyyden varmuus tulevaisuuteen ja teknisen velan vähentäminen. Teknologian uudistamisella voidaan näiden lisäksi saavuttaa muitakin etuja.

Koska JSF-logiikka tapahtuu pääosin palvelimen puolella, interaktiivisuuden toteuttaminen ei onnistu yhtä helposti. Interaktiivisuus on myös hitaampaa, koska käyttöliittymän päivitys vaatii kommunikaatiota palvelimen kanssa. Koska käyttöliittymän tila pidetään ko-

konaisuudessaan palvelimen käyttäjäistunnossa, aktiivisten ikkunoiden lukumäärää käyttäjää kohden on pitänyt rajoittaa. Logiikan siirtäminen selaimen puolelle nykyaikaisempien käytäntöjen mukaisesti auttaisi näissä ongelmissa.

JSF:n testaus on vaikeampaa, koska sen logiikkaan ei pääse kooditasolla helposti käsiksi ja ajo vaatii koko sovelluspalvelimen. Tämän vuoksi käyttöliittymälogiikan automaattinen testaus on jäänyt vähemmälle verrattuna muihin sovelluksen osiin. Nykyaikaisemmissa käyttöliittymäkirjastoissa testien toteutus on helpompaa, ja ne usein tarjoavat valmiita työkaluja testaamista varten [7, 8].

Järjestelmän toiminnallisuus pyritään pitämään samanlaisena kuin aikaisemmin, mutta siihen voidaan tehdä joitain käyttökokemukseen liittyviä parannuksia. Esimerkiksi mobiilikäyttö on asia, jota halutaan tutkia migraation yhteydessä. Tähän ei mennä tämän syvällisemmin tässä diplomityössä, sillä mobiilikäyttö ei ole suoraan teknologiavaihdosta riippuvainen.

Uusina teknologiavaihtoehtoina tutkitaan erityisesti suosioon tulleita yhden sivun sovelluksen teknologioita, joissa logiikka siirtyy palvelimelta enemmän selaimen puolelle. Tämä on lähestymisenä erilainen, minkä vuoksi se vaatii syvempää tutkintaa. Ohjelmointikielenä näissä on tyypillisesti JavaScript. Koska järjestelmässä toiminnan oikeellisuus on erityisen tärkeää, JavaScriptin sijaan uudistuksessa tulisi käyttää sen vahvemmin tyypitettyä versiota, TypeScriptiä.

Teknologiauudistuksen ohjelmistomigraatio pitää pystyä tekemään osissa järjestelmän laajuuden vuoksi. Uudistuksessa ei myöskään ole vielä sitouduttu uudistuksen laajuuteen. Uudistus saatetaan tehdä vain joihinkin yleisimpiin komponentteihin, tai aluksi vain uusiin ominaisuuksiin. Lopullisen uudistuksen laajuudesta riippumatta uudistuksen pitäisi tukea sekä vanhaa että uutta teknologiaa rinnakkain.

3 AIKAISEMPI TUTKIMUS

JSF-teknologian integroimiselle on entuudestaan toteutettu kirjastoja ja useampi järjestelmä on onnistuneesti siirtynyt uudempiin teknologioihin. Löydetyt kirjastot kuitenkin olivat vanhentuneita, eivätkä sellaisenaan ole käyttökelpoisia. Muutamasta projektista löytyy joitain toteutusyksityiskohtia, mutta tarkempaa kirjallisuutta aiheesta ei löydy. Niistä voidaan silti oppia jotain.

Tässä luvussa tutustutaan ensin lyhyesti kahteen vanhentuneeseen JSF-integraatiokirjastoon ja katsotaan, miten ne aikoinaan yrittivät ratkaista integraatioon liittyviä ongelmia. Tämän jälkeen katsotaan kahta nykyaikaisempaa projektia, jotka ovat onnistuneesti saaneet päivitettyä vanhat JSF-järjestelmänsä.

3.1 AngularFaces

Yksi kirjastoista JSF:n integroimiseen yhden sivun sovelluksen teknologian kanssa on vuonna 2013 julkaistu AngularFaces. Tämä kirjasto yhdistää JSF-teknologian ja AngularJS-teknologian siten, että niitä voidaan käyttää yhdessä. [9]

Ensi silmäyksellä kirjasto vaikuttaa lupaavalta, mutta siinä on useita ongelmia. Kirjaston isoin ongelma on, että se on tehty Angularin vanhemmalle AngularJS-versiolle. AngularJS on merkittävästi erilainen kuin uudemmat versiot eikä enää saa merkittäviä päivityksiä. Kirjastoa ei enää ylläpidetä eikä siihen ole tullut päivityksiä vuoden 2016 jälkeen [9, 10].

AngularFaces ei yritä poistaa JSF-logiikkaa, vaan sen sijaan integroi sen AngularJS-koodin kanssa ja tarjoaa mekanismeja, joilla ne voivat kommunikoida keskenään [9]. Tämä tuo syvän riippuvuussuhteen teknologioiden välille, mitä olisi hyvä välttää. Teknologiakohtaisen kommunikaatiomenetelmän sijaan olisi parempi hyödyntää teknologiariippumatonta yksisuuntaista kommunikaatiota, kuten HTTP-rajapintaa.

Näistä syistä AngularFaces ja sen käyttämät menetelmät eivät ole soveltuvia tähän projektiin.

3.2 AngularBeans

AngularBeans on vastaavanlainen kuin AngularFaces, mutta hieman yksinkertaisempi. Siinä AngularJS on vastuussa kokonaan käyttöliittymästä, eikä JSF-käyttöliittymäpohjia

tarvita ollenkaan. Se vain tarjoaa rajapinnat, joiden avulla JavaScript voi kommunikoida JavaBean-luokkien kanssa [11]. Tämä mahdollistaisi palvelinpään logiikan uudelleen käyttämisen.

Vaikka AngularBeans ei sido teknologioita yhteen yhtä tiukasti kuin AngularFaces, sillä on monet samoista ongelmista. AngularBeans tukee vain AngularJS-versiota eikä ole saanut päivityksiä vuoden 2018 jälkeen [12]. Kirjasto ei myöskään näytä olevan kovin käytetty, eikä sille löydy kattavaa dokumentaatiota.

Yksinkertaistettuna AngularBeans toimii siten, että se generoi JavaBean-luokista JavaScript-luokkia, jotka AngularJS pystyy injektoimaan riippuvuusinjektiolla. Generoidun luokan funktiot kommunikoivat oikean JavaBean-luokan kanssa ja siten tarjoavat rajapinnan selaimen ja palvelimen välillä. [11]

AngularBeansin menetelmä voisi olla lupaava vaihtoehto, jos sitä pystyisi käyttämään. Se kuitenkin toimii pelkästään AngularJS-kirjastolle, eikä sitä todennäköisesti saa helposti mukautettua uudempaan. Vastaavan toiminnallisuuden toteutus saattaisi olla työläs tehtävä.

3.3 Esimerkkitapaus: Bandwidthin React-migraatio

Nick Bragon kertoo blogissaan Bandwidth-sivuston käyttöliittymän onnistuneesta migraatiosta, jossa siirryttiin JSF:stä React-kirjastoon [13]. Blogi ei mene syvälle kaikkiin teknisiin ratkaisuihin, mutta niistä on kerrottu korkealla tasolla.

Bandwidth käyttää Reactia perinteisenä yhden sivun sovelluksena, joka jaetaan NodeJS-palvelimelta, erikseen JSF:n Java-palvelimesta. Perustelu NodeJS-palvelimen käytölle on sen työkalujen helppokäyttöisyys. Reititys Java- ja NodeJS-palvelimien välillä tehdään Apachen reverse proxylla, jonka avulla siirtymät eri teknologioiden välillä tapahtuvat käyttäjänäkökulmasta sulavasti. [13]

Toisin kuin JSF:llä, Reactilla ei pysty tekemään tietokantaoperaatioita suoraan. Näitä varten toteutettiin Java-palvelimelle rajapinnat, joilla tietoihin pääsee käsiksi. Näin pystytään hyödyntämään aikaisempaa toteutusta ja rakenteesta tulee modulaarisempi. Tilanhallintaan käytetään Redux-kirjastoa. [13]

Teknologiamigraation koettiin parantaneen kehitystehokkuutta merkittävästi. Erityisesti testattavuus ja tilanhallinta koettiin helpommaksi. Blogissa lisäksi korostettiin, että sovelluksen päivitys pienissä osissa helpottaa prosessia merkittävästi. [13]

3.4 Esimerkkitapaus: Gofore Oyj:n Vue-integraatio

Gofore Oyj:llä on ollut JavaServer Pages -järjestelmä (JSP), johon on myöhemmin otettu mukaan Vue-teknologiaa. JSF on rakennettu JSP-teknologian päälle [14, luku 1.3.4], joten ne ovat riittävän samankaltaisia tähän vertailuun.

Projektista ei ole julkista kirjallisuutta, mutta diplomityön aikana käytiin lyhyt haastattelu projektin sovelluskehittäjän kanssa. Projektissa tavoitteena oli ottaa Vue käyttöön sovelluksen uusissa näkymissä. Projektissa ei yritetty siirtää aikaisempaa toiminnallisuutta.

Projektin integraatiossa Vue-komponentit alustetaan JSP-tiedostojen sisällä. Käyttäjät tulevat JSP-sivulle kuten tavallisesti, mutta sen sisälle renderöidään Vue-komponentti. Sivun Vue-komponentti on vastuussa pääsisällön renderöinnistä. Kaikille sivuille yhteiset komponentit, kuten navigaatiopalkki, voidaan edelleen renderöidä JSP-teknologian avulla.

Tässä lähestymisessä Java EE -sovelluspalvelin on vastuussa reitityksestä kuten ennenkin. Sivusto on käytännössä edelleen monen sivun sovellus (multi-page application), mutta sen yksittäisillä sivuilla on Vue-koodia. Vuen ei tämän vuoksi tarvitse hoitaa reititystä. Kyseisessä projektissa ei myöskään ollut tarvetta tilanhallinnalle, joten Vuen ei tarvinnut huolehtia siitä.

Tässä lähestymisessä hyvä puoli on, että ero JSP:lla ja Vuella tehtyjen sivujen välillä voi olla täysin läpinäkyvä loppukäyttäjille, sillä url-polut ja sivun päärakenne pysyvät samoina. Vaikka Vue-komponentti alustetaan JSP-tiedostossa, niiden välillä ei tarvitse olla vahvaa riippuvuussuhdetta. Tämä voisi myöhemmin mahdollistaa niiden erotuksen kokonaan, jos sille olisi tarvetta.

Lähestymisen mahdolliset haitat liittyvät tehokkuuteen. Jos Vueta ei käytetä yhden sivun sovelluksen teknologiana, voidaan menettää tiettyjä tehokkuushyötyjä, joita käsitellään tarkemmin luvussa 5.1.1. Lisäksi renderöinnissä on kaksi kierrosta: ensin JSP:n renderöinti ja sitten Vuen renderöinti. JSP:n renderöinti tässä tapauksessa on hieman yksinkertaisempi, mutta on todennäköisesti hieman raskaampaa kuin staattisen HTML-tiedoston jakaminen. Kyseisessä projektissa ei havaittu merkittävää hitautta, minkä vuoksi tätä ei koettu ongelmaksi.

4 OHJELMISTOMIGRAATIO

Ohjelmistomigraatio on prosessi, jossa ohjelmisto siirretään uudelle alustalle tai uuteen teknologiaan. Migraatio on osa ohjelmiston ylläpitoa. Ohjelmistomigraatioiden tavoitteena on muun muassa parantaa järjestelmän ylläpidettävyyttä ja soveltuvuutta uusiin vaatimuksiin. Ohjelmistomigraatio tyypillisesti tehdään järjestelmiin, jotka ovat aktiivisessa käytössä. [15]

Toisin kuin ohjelmiston uudelleen toteutuksessa (redevelopment), ohjelmistomigraatiossa pyritään hyödyntämään vanhaa toteutusta mahdollisimman paljon. Muunnetun järjestelmän pitäisi käyttäytyä mahdollisimman samalla tavalla kuin vanha järjestelmä ja aiheuttaa mahdollisimman vähän häiriöitä jo käytössä olevaan järjestelmään. Migraatio voi olla monimutkaisempaa kuin järjestelmän toteutus uudestaan, mutta se on tyypillisesti vähemmän riskialtista ja halvempaa. [15]

Ohjelmistomigraation toteutus voi olla hyvin erilaista riippuen järjestelmästä ja käytetyistä teknologioista, eikä sitä tyypillisesti voi kokonaan automatisoida. Ei siis ole olemassa yhtä täsmällistä mallia, jolla minkä tahansa järjestelmän voisi siirtää uudelle teknologialle. Onnistunut ohjelmistomigraatio edellyttää vanhan järjestelmän syvää ymmärtämistä. [15]

4.1 Migraatiostrategia

Demeyeter et al. esittää kirjassa Object oriented reengineering patterns [16], että ohjelmistomigraatio on hyvä tehdä pienissä osissa. Vaiheistetussa migraatiossa (Phased interoperability) vanha järjestelmä siirretään osissa uuteen järjestelmään [17, s. 42–43]. Vanha järjestelmä pysyy koko migraation ajan käytössä. [16, s. 147–170]

Kun migraatio tehdään pienissä osissa, saadaan parempi varmuus järjestelmän toiminnasta. Pieniä osia on helpompi testata ja virheet havaitaan nopeammin. Toteutus on ketterämpää ja mahdollisiin vaatimusmuutoksiin voidaan reagoida nopeammin. [16, s. 147–170] Jos migraatio tehtäisiin täysin erillisenä sovelluksena, kehittäjien pitäisi ylläpitää järjestelmästä kahta erillistä versiota. Jos järjestelmä on laaja ja se saa migraation lisäksi muuta kehitystä, mahdolliset muutokset pitää mahdollisesti tehdä molempiin järjestelmiin. Tämä voisi vaatia moninkertaisen työmäärän ja lisäisi virheiden riskiä.

Vaiheittaisessa migraatiossa uusi ja vanha teknologia muodostavat hybridijärjestelmän, jossa on käytössä sekä vanhaa että uutta teknologiaa. Eräs haaste on näiden välinen kommunikaatio. Tarkka mekanismi riippuu teknologioista, mutta kommunikointia varten

voi olla tarpeen tehdä uusia rajapintoja, jotka käärivät vanhan järjestelmän toimintoja. Näiden abstraktiotason pitää olla käyttötarkoitukseen soveltuvia. Liian matalat abstraktiotasot voivat tehdä uuden järjestelmän liian riippuvaiseksi vanhan järjestelmän abstraktioista. [16, s. 147–170] [17, s. 42–43]

Migraatiossa kannattaa ensin toteuttaa käyttäjien näkökulmasta tärkeimmät tai eniten lisäarvoa tuovat ominaisuudet. Koska muutokset tulevat osaksi vanhaa järjestelmää, toteutuksen olisi hyvä olla mahdollisimman tutun näköinen. Jos esimerkiksi käyttöliittymä on merkittävästi erilainen kuin ennen, käyttäjien voi olla hankalampi sopeutua siihen. [16, s. 147–170]

Toteutuksen aikana voi olla hyödyllistä vertailla uutta toteutusta vanhempaan versioon, jotta voidaan paremmin varmentua toiminnan samankaltaisuudesta. Tämän jälkeen on kuitenkin tärkeää, että vanha toteutus poistetaan. Jos vanha toteutus jää, siitä tulee teknistä velkaa, mikä hankaloittaa sekä järjestelmän ylläpitoa että uusien ominaisuuksien lisäämistä. [16, s. 147–170]

Ennen varsinaisen migraation aloitusta on hyvä tehdä prototyyppi, jolla testataan isompia arkkitehtuurisia muutoksia. Prototyypissä kannattaa miettiä erityisesti isoimpia teknisiä riskejä. Prototyypeillä on helppo testata teknisiä ratkaisuja ennen varsinaista toteutusta. [16, s. 147–170]

4.2 Uudelleensuunnittelu

Uudelleensuunnittelulle (reengineering) löytyy kirjallisuudesta useita määritelmiä. Eräs yleisesti käytetty määritelmä ohjelmistojen uudelleensuunnittelulle on, että se on prosessi, jossa tutkitaan kohdejärjestelmää ja muunnetaan se haluttuun muotoon. Uudelleensuunnittelu voidaan jakaa kolmeen osaan: käänteinen suunnittelu (reverse engineering), muuntaminen (transformation) ja etenevä ohjelmistosuunnittelu (forward engineering). [17, s. 30–39]

Käänteinen suunnittelu on uudelleensuunnittelun ensimmäinen vaihe. Sen tavoitteena on kerätä tietoa järjestelmästä. Tähän sisältyvät järjestelmän rakenteen, käyttäytymisen ja vuorovaikutussuhteiden tarkastelu. Eräs vaiheen tavoitteista on saada korkeamman abstraktiotason tietoa, jota voidaan hyödyntää muissa vaiheissa. [17, s. 30–39]

Muunnosvaiheessa muunnetaan käänteisestä suunnittelusta saadut korkean abstraktiotason mallit uudelle arkkitehtuurille. Tässä vaiheessa esimerkiksi löydetään vanhoille käsitteille vastineet uudesta arkkitehtuurista. Muunnoksia voidaan tehdä arkkitehtuurisella tasolla, funktiotasolla sekä kooditasolla. Muunnos voidaan tehdä millä tahansa näistä tasoista. [17, s. 30–39]

Etenevässä ohjelmistosuunnittelussa muunnosvaiheen korkeamman tason abstraktiot toteutetaan konkreettisena ohjelmakoodina. Siinä missä käänteinen suunnittelu muuntaa alemman tason abstraktioita korkeamman tason abstraktioiksi, etenevä ohjelmistosuunnittelu muuntaa korkeamman tason abstraktioita alemman tason abstraktioiksi. Etenevä

ohjelmistosuunnittelu vastaa perinteistä ohjelmistokehitystä. [17, s. 30–39]

5 TEKNOLOGIAT JA OHJELMOINTIKIELET

Tässä luvussa tutututaan migraatiossa käytettäviin teknologioihin ja ohjelmointikieliin. Luvussa 5.1 tutustutaan yhden sivun sovelluksiin ja työn prototyypeissä käytettyihin teknologioihin, Angulariin ja Reactiin. Luvussa 5.2 katsotaan Javan ja TypeScriptin välisiä eroavaisuuksia migraation näkökulmasta.

5.1 Teknologiavaihtoehdot

Tässä luvussa tutustutaan tarkemmin kahteen ohjelmistomigraation mahdolliseen teknologiavaihtoehtoon. Luvussa 5.1.1 tutustutaan yleisellä tasolla yhden sivun sovelluksien toimintaperiaatteisiin. Luvuissa 5.1.2 ja 5.1.3 tutustutaan kahteen yleisimpään yhden sivun sovelluksen kirjastoon: React ja Angular [18].

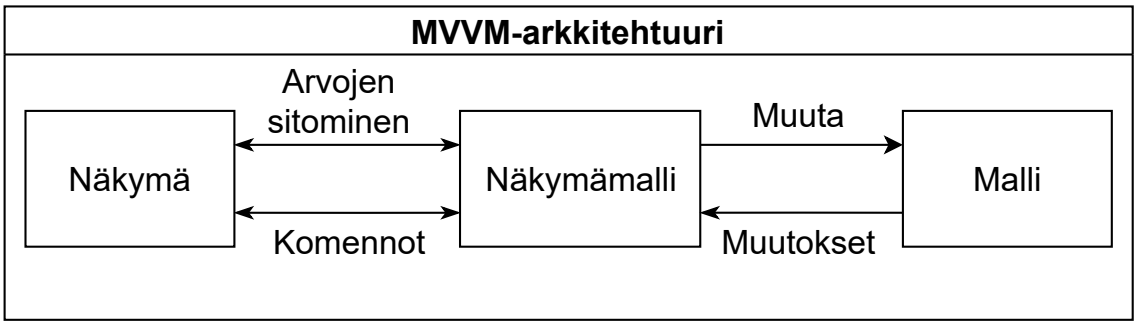
5.1.1 Yhden sivun sovellukset yleisesti

Yhden sivun sovellukset (SPA, Single-page application) ovat viimeisen vuosikymmenen aikana yleistynyt lähestymistapa web-sovelluksien tekemiseen. React, Angular ja Vue ovat nykyisin suosituimmat kirjastot yhden sivun sovelluksille [19].

Toisin kuin monissa perinteisemmissä monen sivun sovelluksissa, sovelluksen päälogiikka on palvelimen sijaan selaimessa. Selaimen JavaScript on vastuussa sivun sisällön hallinnasta. Tähän sisältyy muun muassa sisällön renderöinti, sivunavigaatiot ja interaktiivisuuden toteutus. [20]

Yhden sivun sovelluksien tarvitsee ladata palvelimelta vain HTML-tiedosto ja tarvittavat lähdekooditiedostot. Jos käyttäjä navigoi sovelluksen sisällä toiselle sivulle tai sovelluksen tila muuttuu, JavaScript renderöi vain sivun muuttuneet osat uudestaan. [20] Tämä voi nopeuttaa sivunavigaatioita, sillä koko sivua ei tarvitse ladata palvelimelta uudestaan. [21] Käänteisesti sivuston ensimmäinen latauskerta voi olla hieman hitaampi, sillä selaimen täytyy odottaa JavaScriptin lataamista.

Koska yhden sivun sovelluksien logiikka on selaimen puolella, ne eivät pääse suoraan käsiksi palvelimen ominaisuuksiin. Selaimilla ei esimerkiksi ole suoraa yhteyttä tietokantaan, mikä on JSF:llä mahdollista. Jos selaimen tarvitsee käyttää palvelinpään ominaisuuksia, niille perinteisesti toteutetaan kommunikaation mahdollistavat HTTP-rajapinnat. [20]



Kuva 5.1. *Kaavio MVVM-arkkitehtuurista. [22]*

Yhden sivun sovellukset ovat tyypillisesti komponenttipohjaisia. Sekä React, Angular että Vue tukevat omien HTML-elementtien kaltaisten komponenttien tekoa, joihin voi liittää omaa logiikkaa. Komponentit ovat uudelleen käytettäviä, ja isommat komponentit muodostetaan pienempiä komponentteja käyttäen.

Yhden sivun sovelluksen sovelluskehykset usein hyödyntävät MVVM-arkkitehtuurimallia (ModelView-View-Model), jossa arkkitehtuuri jakautuu näkymämalliin, näkymään ja malliin [23]. Toisin kuin JSF:n MVC-mallissa, tässä näkymä ja malli on erotettu toisistaan näkymämallilla [20, luku 4]. Kuvassa 5.1 on kaavio MVVM-arkkitehtuurista.

Näkymämallit ovat yhden sivun sovelluksissa usein komponenttikohtaisia. Ne kuvaavat näkymäkomponentin omaa tilaa, ja hoitavat tarvittavat kytkökset malliin. Malli sisältää bisneslogiikan ja datan. [20, luku 4] Toisin kuin näkymämallissa, JSF:n JavaBeanit eivät välttämättä ole komponentti- tai edes näkymäkohtaisia.

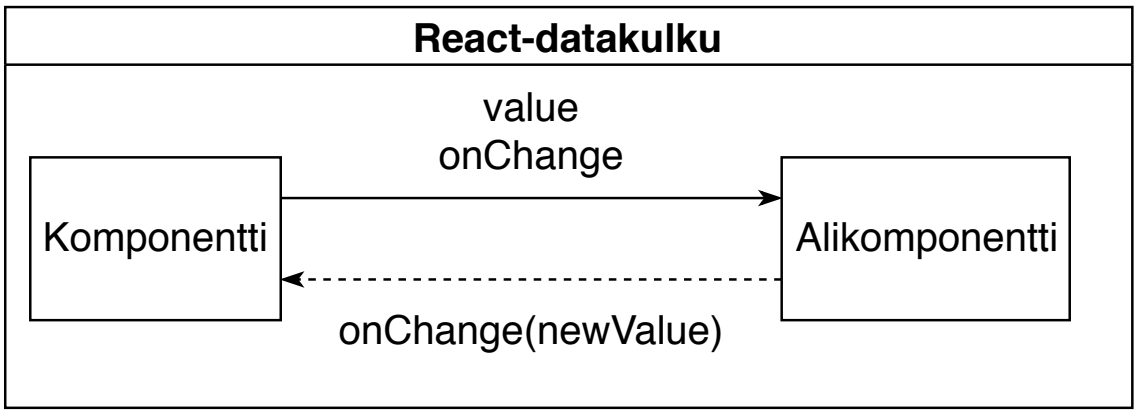
5.1.2 React

React on vuonna 2013 julkaistu Facebookin kehittämä käyttöliittymäkirjasto [24]. Reactilla on suuri yhteisö ja se saa aktiivisesti päivityksiä [24]. Se on vuoden 2019 suosituin yhden sivun sovelluksen kirjasto [19].

React on pelkkä käyttöliittymäkirjasto, minkä takia se ei tarjoa kaikkia ominaisuuksia, joita useimmat sovelluskehykset antavat. Esimerkiksi reititys ja tilanhallinta on usein tarpeen tehdä ulkoisilla kirjastoilla. Näille eräitä yleisiä vaihtoehtoja ovat React router ja Redux.

React käyttää oletuksena JavaScriptiä, mutta tukee myös TypeScriptiä [24]. Perinteisen Java- tai TypeScriptin lisäksi React perinteisesti käyttää JSX-syntaksia. JSX on JavaScript-laajennos, joka mahdollistaa XML-kaltaisen syntaksin käytön näkymäkoodin kirjoittamiseksi. Se ei ole suoraan yhteensopiva XML:n tai HTML:n kanssa, mutta on samankaltainen. JSX muunnetaan esimerkiksi Babel-kirjastolla tavalliseksi JavaScriptiksi. [25, 26]

React on perusluonteeltaan komponenttipohjainen. Kaikki React-sovellukset ovat yksinkertaisia React-komponentteja, jotka puolestaan voivat koostua alikomponenteista. Komponenteilla voi olla oma tila, tai ne voivat saada arvoja hierarkian ylemmiltä komponenteilta. [24]



Kuva 5.2. *Kaavio Reactin datan kulkusuunnasta ja muuttamisesta alikomponenteissa.*

Reactissa datan kulkusuunta on aina ylhäältä alaspäin, jossa ylimpänä on sovelluksen juurikomponentti. Toisin kuin JSF, React ei suoraan tue kaksisuuntaista arvojen sitomista (two-way binding), jossa arvon muuttuminen toisessa aina muuttaa sen myös toiseen. Tämän rajoitteen pystyy kiertämään välittämällä alikomponenteille ylemmän komponentin tilaa muuttavia funktioita. Tätä on havainnollistettu kuvassa 5.2. Kuvassa `onChange` on Reactilla generoitu funktio, joka muuttaa komponentin `value`-muuttujaa. Alikomponentti voi sen kautta muuttaa yläkomponentin tilaa.

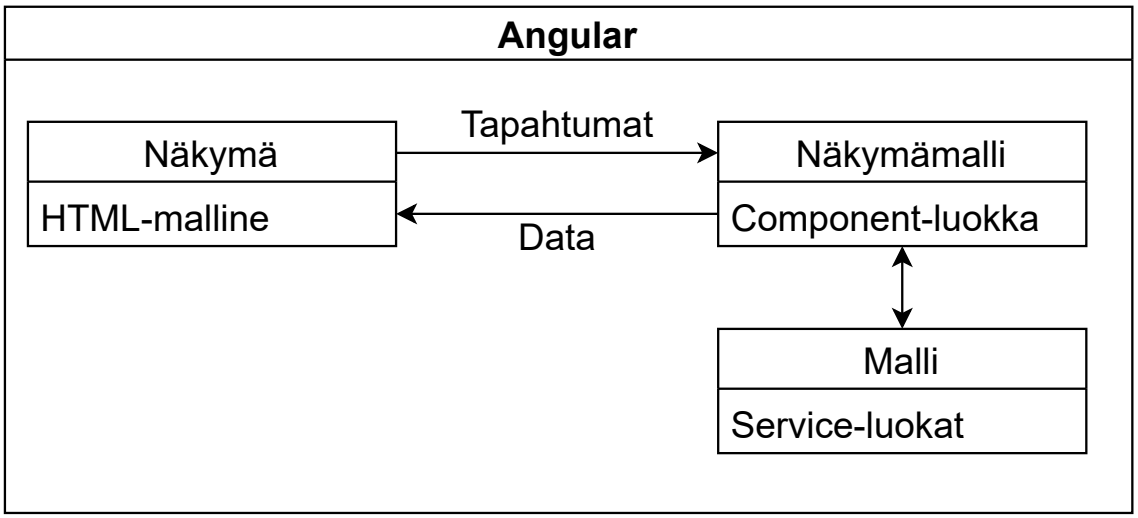
Monista muista yhden sivun sovelluksen kirjastoista poiketen Reactissa näkymien mallinekoodi, JSX, on suoraan osana JavaScript-koodia. JSX kääntyy tavallisiksi JavaScript-olioiksi, minkä vuoksi JSX-elementtejä voidaan sijoittaa muuttujiin ja muutenkin käyttää kuten muita JavaScriptin arvoja. [26]

Komponenttien sisäiset tilat täytyy aina päivittää Reactin `setState`-funktiolla tai sen vastineella, jotta React osaa päivittää käyttöliittymän. Jos tilaa yrittää muuttaa muuta kautta, React ei välttämättä tunnista muutosta. [24] Tämän vuoksi React hyödyntää usein enemmän funktionaalisen ohjelmoinnin menetelmiä kuin olio-ohjelmointia.

Versiossa 16.8 React sai hook-ominaisuuden, jonka avulla on helpompi tehdä uudelleen käytettävää logiikkaa React-komponenteista erillään [27]. Hookit ovat yksinkertaisia funktioita, jotka voivat pitää oman tilan ja toteuttaa React-komponenttien muut tarpeelliset ominaisuudet. Ne on tarkoitettu vaihtoehtona Reactin luokkapohjaisille komponenteille. [28] Facebook uskoo hookien olevan Reactin tulevaisuus ja suosittelee niiden käyttöä uusissa projekteissa [29].

5.1.3 Angular

AngularJS on vuonna 2010 julkaistu Googlen kehittämä sovelluskehys [30]. Vuonna 2016 julkaistiin AngularJS:n versio 2, joka ei ole yhteensopiva aikaisempien versioiden kanssa. Version 2 jälkeen sovelluskehyksen nimeksi tuli Angular. [31] Angular on tämän jälkeen saanut useita pienempiä päivityksiä niin ominaisuuksiin kuin suorituskykyyn. Angularin uusin versio on 10 ja Angular saa säännöllisiä toiminnallisia päivityksiä noin puolen vuo-



Kuva 5.3. *Angularin MVVM-arkkitehtuuri.*

den välein [32].

Toisin kuin React, Angular on täysi sovelluskehys. Angularin päätoteutus antaa muun muassa valmiita reititys-, http-, validointi-, testaus- ja animaatiomoduuleja [33]. Koska kaikki tärkeimmät ominaisuudet tulevat valmiina, ulkoisia kirjastoja ei välttämättä tarvita. Angularille kuitenkin löytyy myös laajasti muiden osapuolien ja yhteisön tekemiä kirjastoja.

Muista sovelluskehyksistä poiketen Angular käyttää oletuksena TypeScript-ohjelmointikieltä. Tämä mahdollistaa tiukemmat tyyppitarkistukset, mikä voi auttaa virheiden löytämisessä. Angular tukee tyyppitarkastuksia myös sen HTML-mallinetiedostoissa [34].

Angularin arkkitehtuuri noudattaa MVVM-arkkitehtuurimallia (ModelView-View-Model). Kuvassa 5.3 näytetään, miten tämä näkyy Angularin arkkitehtuurissa. Angular tukee näkymän ja näkymämallin välillä kaksisuuntaista arvojen sitomista (two-way binding), minkä ansiota muutokset yhdessä näkyvät automaattisesti myös toisessa. Tätä kautta muutokset voivat myös tulla malliin asti. Angularilla on oma muutoksien tunnistusmekanismi, minkä ansiota näkymämallin arvoja ei tarvitse asettaa erillisellä mekanismilla kuten Reactissa. [33] Tämän ansiota Angular tukee perinteistä olio-ohjelmointiparadigmaa hyvin.

Angular-sovellukset koostuvat pääosin komponenteista. Jokainen komponentti koostuu kolmesta osasta: HTML-malline, CSS ja Component-luokka. Component-luokat ovat TypeScript-luokkia, jotka määräävät ominaisuudet, joita mallineet voivat käyttää. HTML-mallineet käyttävät HTML-koodia, jota on laajennettu Angular-syntaksilla. [33]

Angularissa bisneslogiikka perinteisesti on Service-luokkien sisällä. Component-luokat ja palvelut voivat injektoida Service-luokkien instansseja riippuvuusinjektion avulla. [33]

5.2 Javan ja TypeScriptin eroavaisuudet

Java ja TypeScript ensisilmäyksellä vaikuttavat hyvin samankaltaisilta kieliltä. Molemmat tukevat vahvaa tyypitystä ja olio-ohjelmointia. Niissä kuitenkin on monia eroavaisuuksia.

Tässä luvussa tutustutaan tarkemmin Javan ja TypeScriptin erilaisuuksiin ja miten ne vaikuttavat migraatioon niiden välillä. Luvussa 5.2.1 vertaillaan yleisellä tasolla kielien tyyppijärjestelmien eroja. Luvuissa 5.2.2 ja 5.2.3 tutkitaan ohjelmointikielien primitiivi- ja enum-arvojen eroja. Luvussa 5.2.4 tutkitaan mitä eroja TypeScriptin asynkroninen luonne tuo. Lopulta luvussa 5.2.5 otetaan yhteenveto eroista migraation näkökulmasta.

5.2.1 Tyyppijärjestelmä

Java käyttää nimellistä tyyppijärjestelmää (nominal type system). Javassa millä tahansa arvolla voi olla yksi kolmesta mahdollisesta tyypistä: null, primitiiviarvo tai tunnetun luokan instanssi. Jokainen Javassa oleva olio kuuluu aina johonkin luokkaan (class). Jos muuttujalle on annettu jokin luokka tyypiksi, sille annetun arvon täytyy kuulua luokkaan, joka joko on kyseinen luokka tai periytyy siitä. Luokat ovat konkreettisia käsitteitä, jotka ovat myös ajon aikana olemassa. Vastaavasti jos muuttujan tyyppinä on rajapintaluokka (interface), sille annetun arvon luokan täytyy eksplisiittisesti toteuttaa se. Ei siis riitä, että arvolla sattuu olemaan rajapintaluokan mukaiset ominaisuudet. [35]

TypeScript puolestaan käyttää rakenteellista tyyppijärjestelmää. Sen sijaan että TypeScript katsoisi mistä luokasta tai rajapintaluokasta arvo on periytynyt, se tarkastelee arvon ominaisuuksia. Jos järjestelmässä on kaksi rakenteellisesti identtistä luokkaa, tyyppijärjestelmä tulkitsee molempien tyypit samoiksi ilman rajapinta- tai aliluokkien käyttöä. TypeScriptillä on myös mahdollista luoda olioita ilman luokkamäärittelyä, toisin kuin Javassa. [35]

TypeScript kääntyy tavalliseksi JavaScriptiksi. Koska JavaScript sallii useiden erilaisten arvojen sijoittamisen samaan muuttujaan, myös TypeScript sallii tämän. TypeScriptin tyyppijärjestelmässä tyypit eivät välttämättä kuvaa yksittäistä tyyppiä, vaan joukkoa mahdollisista tyypeistä. Muuttujalle voi esimerkiksi määrittää, että se on joko numero tai merkkijono. [35] TypeScript Javasta poiketen tarjoaa monipuolisia tyyppioperaattoreita, kuten tyyppien leikkaus- ja unionioperaatiot [36]. Näillä voidaan muodostaa monimutkaisiakin tyyppimäärityksiä.

Sekä Java että TypeScript tukevat tyyppien muunnosta toisiksi tyypeiksi. Jos tyyppi esimerkiksi on geneerinen, se voidaan muuntaa täsmällisemmäksi tyypiksi. Erona on, että Javalla tyypin tiukentaminen tarkistaa tyypin ajonaikana, jotta se varmasti on oikeaa tyyppiä. TypeScript ei tee mitään ajonaikaisia tarkistuksia. Jos tyyppi ei täsmääkään, se voi johtaa virheisiin muualla ohjelmassa. TypeScript voi vaatia hieman enemmän tarkkuutta siitä, että tyypit ovat oikeasti sitä, mitä on määritelty. TypeScriptin strict-asetukset auttavat välttämään tällaisia tilanteita, mutta eivät kokonaan estä sitä [37].

TypeScript tukee kaikkia yleisimpiä olio-ohjelmoinnin ominaisuuksia, kuten periyttäminen, rajapintaluokat ja staattiset metodit. [35] TypeScript ei edellytä olio-ohjelmoinnin käyttöä yhtä vahvasti kuin Java, mutta pystyy tekemään useimmat samat asiat.

5.2.2 Primitiiviarvot

Java ja TypeScript käyttäväy erilaisia primitiiviarvoja. Javalla esimerkiksi on kuusi erilaista primitiivityyppiä numeroille: byte, short, int, long, float ja double [38]. TypeScriptillä vastaavasti on käytössä number ja bigint, joista number on yleisemmin käytetty. TypeScriptin number-tyyppi on 64-bittinen liukukuku, joka vastaa Javan double-tyyppiä [39, s. 8].

Double sisältää suurimman osan muiden numerotyyppien arvoista. Hyvin suuria kokonaislukuja se ei pysty esittämään yhtä tarkasti kuin long, mutta tarkkuus riittää useimpiin tilanteisiin. Jos on tarvetta käsitellä hyvin suuria kokonaislukuja tarkasti, bigint-tyyppiä voi käyttää numberin sijaan.

5.2.3 Enum-arvot

Enum-arvot poikkeavat hieman Javan ja TypeScriptin välillä. TypeScriptissä enumit ovat aina primitiiviarvoja ja ovat suoraan korvattavissa niiden vastaavilla arvoilla. JavaScriptissä enum-käsitettä ei ole ollenkaan. Javassa puolestaan enumeilla voi olla luokkien ominaispiirteitä, kuten ominaisuuksia ja metodeja.

Jos Javan enumilla on vain yksittäinen ominaisuus, joka on primitiivi, se vastaa hyvin läheisesti TypeScriptin enumia. Jos enumilla on useita ominaisuuksia tai metodeja, ne pitää korvata TypeScriptissä joko luokilla tai tavallisilla olioilla.

5.2.4 Asynkronisuus

Java ja TypeScript usein käsittelevät asynkronisuutta eri tavalla. TypeScript käyttää tapahtumasilmukka-arkkitehtuuria (event loop), jossa erilaiset tapahtumat voivat käynnistää rinnakkaisia (parallel) koodisuorituksia. Tyypillisiä tapahtumia ovat esimerkiksi käyttäjien vuorovaikutus ja HTTP-kyselyjen tuloksien odottaminen. [40, luku 1]

TypeScriptin päälogiikka ajetaan vain yhdellä säikeellä, minkä vuoksi voidaan olla varmoja, että samoja muuttujia ei esimerkiksi muuteta täysin samaan aikaan. Asynkronisten operaatioiden suoritusjärjestys puolestaan ei ole yhtä varmaa. Jos käyttäjä esimerkiksi aktivoi kaksi hakua samaan aikaan eri ehdoilla, ne eivät välttämättä pääty käynnistysjärjestyksessä. Ensimmäinen haku voisi tällöin ylikirjoittaa uudemman, jos tätä ei oteta huomioon. [40, luku 1]

Java-pohjaisilla palvelimilla, kuten WebLogicilla, rinnakkaisuus toteutetaan perinteisesti rinnakkain (concurrent) ajettavilla säikeillä. Yksi säie käsittelee usein yhden käyttäjän kyselyä kerrallaan. [14, luku 17.12] Tarkasteltavassa järjestelmässä säikeet harvoin jaka-

vat yhteistä muutettavaa muistia ja Javan asynkronisia ominaisuuksia käytetään harvoin, minkä vuoksi merkittäviä rinnakkaisuusongelmia ei ole. Koodin näkökulmasta logiikka on synkronista.

TypeScriptillä pitkäkestoiset operaatiot, kuten HTTP-kyselyt, tapahtuvat taustalla erillisellä säikeellä. Kun operaatio päättyy, operaation jälkeinen toimenpide lisätään tapahtumasilmukan suoritettavaksi. Muu TypeScript-koodi operaation ympärillä ei odota operaation päättymistä ennen jatkamista. [40, luku 1]

TypeScriptin asynkroniset funktiot saa simuloimaan synkronisen koodin käyttäytymistä ECMAScript 2016 yhteydessä lisätyn async/await-syntaksin myötä. Syntaksin avulla on helpompaa kirjoittaa koodia, joka odottaa asynkronisen koodin suoritusta. Tämä ei kuitenkaan estä koodisuoritusta muualla, joten mahdolliset rinnakkaisuusongelmat pitää silti huomioida. [40, luku 4]

5.2.5 Yhteenveto

Edellisissä luvuissa käsiteltiin Javan ja TypeScriptin välisiä eroja ja samankaltaisuuksia. Suurin osa eroista kohdistuu ohjelmointikielien tyyppijärjestelmien toteutuksiin. Kaiken kaikkiaan TypeScript on tyypitykseltään hieman joustavampi kuin Java.

Kulissien takana olevista eroista huolimatta suurimmalle osalle Javan ominaisuuksista löytyy vastine TypeScriptillä, joka toimii vastaavalla tai lähes vastaavalla tavalla. Koska TypeScript tukee olio-ohjelmoinnin yleisimpiä piirteitä, Javan hierarkioita voidaan käyttää myös TypeScriptillä. Riittävän samankaltaisuuden vuoksi suuren osan Java-koodista pystyy kirjoittamaan vastaavaksi TypeScript-koodiksi.

Yksi isoimmista eroista kielien välillä on TypeScriptin käyttämä tapahtumasilmukka, joka tuo koodiin rinnakkaisuuden ominaispiirteitä. Rinnakkaisuuteen täytyy kiinnittää hieman enemmän huomiota logiikkaa siirrettäessä.

6 JÄRJESTELMÄINTEGRAATIO

Kun migraatio tehdään vaiheistetusti, järjestelmä siirtyy uudelle teknologialle pienissä osissa. Uusi ja vanha teknologia muodostavat hybridijärjestelmän, jonka on tarkoitus toimia käyttäjänäkökulmasta yhtenä järjestelmänä. Integraation voi jakaa kahteen osaan: käyttöliittymien integraatio ja integraatio palvelinlogiikan kanssa.

Koska migraatio kohdistuu pelkästään käyttöliittymään, migraation osalta itsenäiset näkymät tai näkymäkokonaisuudet ovat sopiva tapa pilkkoa migraatiota pienempiin osiin. Osa näkymistä käyttää uutta teknologiaa ja osa vanhaa. Luvussa 6.1 katsotaan, miten JSF:n ja yhden sivun sovelluksien käyttöliittymät voi integroida keskenään saman hybridijärjestelmän alle.

Palvelimen varsinainen liiketoimintalogiikka ja tietokantaoperaatiot eivät ole osa migraatiota, vaan ne pysyvät samoina. Selaimessa ajettavilla yhden sivun sovelluksilla ei ole yhtä suoraa pääsyä palvelimen ominaisuuksiin kuin JSF:llä, minkä vuoksi täytyy toteuttaa kommunikaatiokerros järjestelmien välille. Tähän tutustutaan tarkemmin luvussa 6.2.

6.1 Käyttöliittymäkerroksien integraatio

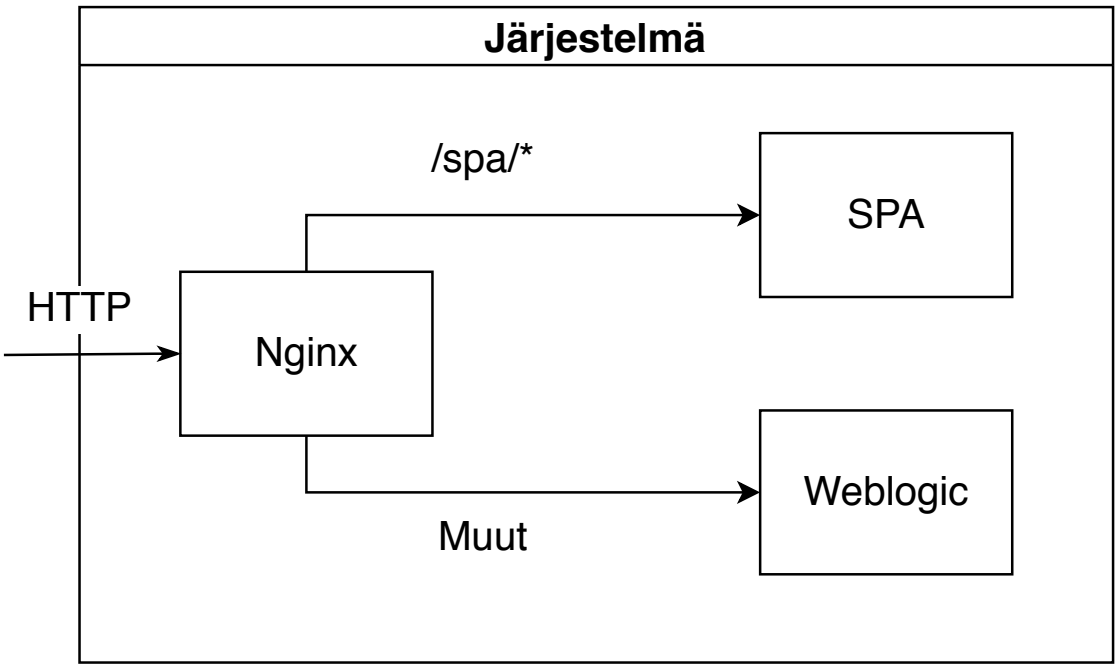
Tutkittavassa migraatiolähestymisessä käyttöliittymä käyttää sekä uutta että vanhaa teknologiaa migraation aikana. Koska käyttöliittymän yksittäiset sivut ovat pääosin itsenäisiä, niiden ei välttämättä tarvitse käyttää samaa teknologiaa.

Käyttöliittymäkerroksien integraatiossa on kaksi huomioitavaa asiaa: teknologioiden integraatio yhden sivun sisällä ja reititys eri sivujen välillä. Yksi sivu voi käyttää joko toista teknologioista tai molempia. Tämä vaikuttaa osaltaan siihen, miten reititys sivujen välillä toteutuu.

Luvussa 6.1.1 tutkitaan integraatiota tilanteessa, jossa järjestelmän JSF- ja SPA-toteutukset toteutetaan itsenäisinä alijärjestelminä. Luvuissa 6.1.2 ja 6.1.3 puolestaan tutkitaan tapoja, joilla saman sivun sisällä voi käyttää molempia teknologioita.

6.1.1 Itsenäiset toteutukset

JSF:n ja yhden sivun sovelluksen teknologian järjestelmät voidaan pitää lähes erillisinä järjestelminä siten, että yksittäisellä sivulla on vain jompaa kumpaa teknologiaa. Käyttöliittymätasolla järjestelmillä ei ole tiukkoja keskeisiä riippuvuuksia. Vain palvelinrajapinnat



Kuva 6.1. *Kaavio erillisten käyttöliittymäsovelluksien integraatiosta.*

ovat yhteiset.

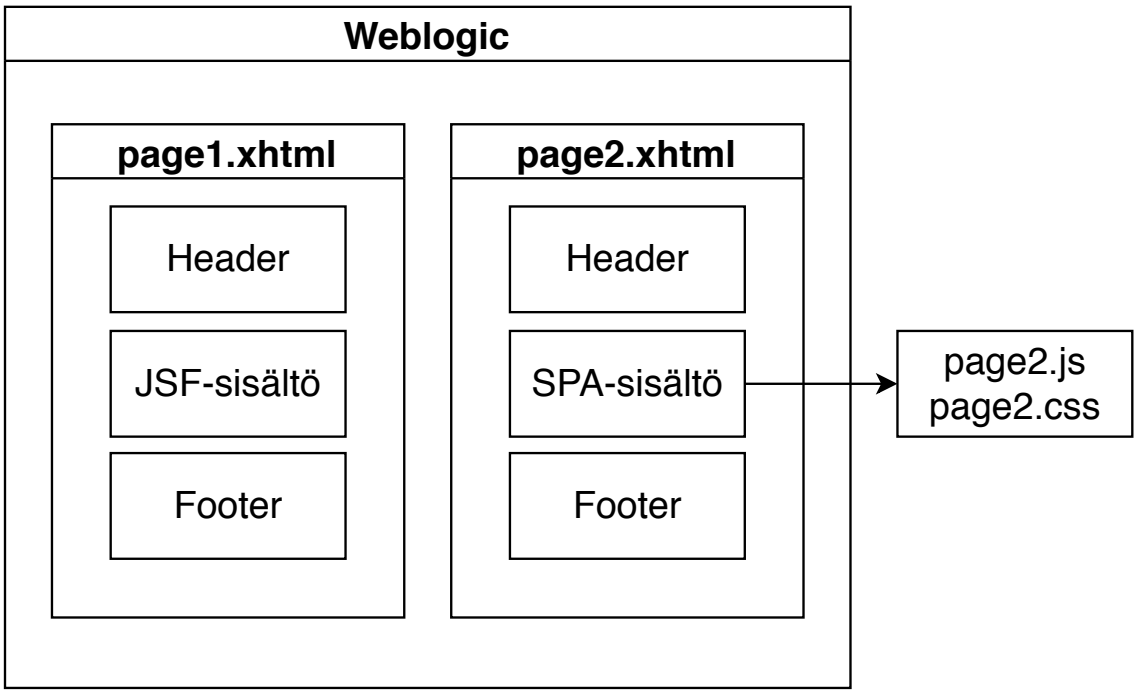
Järjestelmät voi yhdistää toisiinsa niiden edessä olevalla proxy-palvelimella, joka reitin perusteella päättelee kumpaan järjestelmään sivu kuuluu ja välittää kyselyn oikein. Esimerkiksi Nginx-palvelin, mikä on järjestelmässä muutenkin käytössä, soveltuu tähän tarkoitukseen. Molemmat järjestelmät voivat hoitaa omien reittiensä reitityksen. Navigaatio järjestelmien välillä onnistuu tavallisilla linkeillä. Kuvassa 6.1 on kaavio hybridijärjestelmän rakenteesta. Esimerkissä kaikki `/spa`-polun takana olevat sivut ovat yhden sivun sovelluksen sivuja, joille Nginx jakaa sovelluksen `index.html`-tiedoston. Muut polut menevät WebLogic-palvelimelle käsiteltäväksi.

Koska sovellukset ovat erilliset, uuden teknologian täytyy alusta lähtien toteuttaa kaikki käyttöliittymän perusominaisuudet. Tähän sisältyy muun muassa auktorisointi, reitityslogiikka ja sovellusrunko. Jos sivujen halutaan sopivan täydellisesti vanhan järjestelmän kanssa, sovellusrungon täytyy visuaalisesti täsmätä vanhaa toteutusta. Näistä syistä uuden teknologian sovelluksella todennäköisesti on paljon duplikaattilogiikkaa vanhan järjestelmän kanssa.

Etuna lähestymisessä on, että voidaan hyödyntää kaikkia yhden sivun sovelluksien etuja, kuten nopeampi navigaatio sivujen välillä. Lisäksi kun lopulta siirretään kaikki sivut uudelle teknologialle, erillistä migraatiovaihetta ei enää tarvita migraation viimeistelemiseksi.

6.1.2 SPA-toteutus JSF-sivun sisällä

Kevyempi ratkaisu uuden teknologian käyttöönottoon on, että se otetaan käyttöön valmiin sovellusrungon sisälle. Yhden sivun sovellukset ovat pääasiallisesti JavaScript-skriptejä,



Kuva 6.2. *Kaavio integraatiosta, jossa SPA-sovellus on JSF-sivun sisällä.*

jotka renderöivät sisältönsä yksinkertaiseen HTML-elementtiin.

Tässä lähestymisessä WebLogicin JSF-sivu toimisi säiliönä, joka hoitaa reitityksen, näkymätason autentikaation ja auktorisoinnin sekä sovellusrungon. Sovellusrungossa voi olla sivujen yhteiset elementit, kuten navigaatiopalkki. Sivun pääsisältö generoitaisiin yhden sivun sovelluksen teknologialla olevalla skriptillä. Kuvassa 6.2 on kaavio tämän mukaisesta rakenteesta. JSF-sivun näkökulmasta SPA-sisältö on pelkästään viittaus skriptiin ja mahdolliseen tyylitiedostoon, joka renderöi SPA-sisällön haluttuun komponenttiin.

Lähestymisen etuina on, että uuden teknologian ei tarvitse heti toteuttaa kaikkia perusominaisuuksia, ja osan sivun ominaisuuksista voi toteuttaa myös JSF:llä. Lisäksi tämä lähestyminen ei vaadi sivujen url-polkujen muuttamista, kuten erillisemmillä sovelluksilla todennäköisesti tarvitsisi. Kaikki linkit toimivat samalla tavalla, eikä JSF:n tarvitse tietää kumpi on sivun pääteknologia.

Haittapuolena on, että lähestymisellä ei voi hyödyntää kaikkia yhden sivun sovelluksien hyötyjä. Sivu pitää ladata uudestaan navigaatioiden välissä, minkä takia sivunavigaatiot eivät ole yhtä nopeita. Yhden sivun sovelluksien teknologioita ei yleensä käytetä tällä tavalla, minkä takia niiden rakennusprosessi voi myös olla monimutkaisempi. Jos JSF halutaan lopulta poistaa kokonaan, migraation lopussa vaaditaan enemmän työtä sovellusrungon siirtämisessä uudelle teknologialle.

6.1.3 JSF-toteutus SPA:n sisällä

Jos alijärjestelmät halutaan toteuttaa pääosin itsenäisinä sovelluksinaan, mutta joitain osia on hankala siirtää, selaimien iframet ovat eräs mahdollinen ratkaisu. Jos halutulle

JSF-komponentille toteutetaan minimaalinen sivu, yhden sivun sovellus voi käyttää sitä iframen kautta.

Iframeissa on teknisiä rajoituksia, minkä vuoksi esimerkiksi kommunikaatio iframen kanssa voi olla hankala toteuttaa. Se voi kuitenkin olla yksinkertainen ratkaisu, jos halutaan siirtää jokin monimutkaisempi JSF-ominaisuus, joka ei vaadi kommunikaatiota muun käyttöliittymälogiikan kanssa.

Tämä lähestyminen ei skaalaudu kokonaisille sivuille kovin helposti. Se ei ole kovin hyvä vaihtoehto edellisten käsiteltyjen sijaan, mutta voi olla hyödyllinen työkalu niiden lisäksi.

6.2 Palvelinlogiikan integraatio

Yhden sivun sovelluksilla ei ole suoraa pääsyä tietokantaan, minkä vuoksi sille täytyy toteuttaa kommunikaatiorajapinta palvelimen palveluiden kanssa. Luvussa 6.2.1 katsotaan, miten rajapinnan toteutus onnistuu palvelimen puolella. Luvussa 6.2.2 puolestaan tutkitaan, miten selaimen puolella rajapintaa voidaan käyttää.

6.2.1 Rajapintakommunikaatio (palvelin)

Kuten luvussa 2.2 käsiteltiin, suuri osa kohdejärjestelmän liiketoimintalogiikasta on Palvelut-komponentissa. Tähän sisältyvät kaikki tietokantaoperaatiot. Palvelut-komponentti tarjoaa teoriassa valmiit rajapinnat useimpiin toimenpiteisiin, joita tarvitaan. Tässä kuitenkin on muutama ongelmaa. Ensimmäinen on, että Palvelut ei tarjoa HTTP-rajapintaa, johon selain pääsisi helposti käsiksi. Toinen on, että sitä ei ole suunniteltu suoraan käytettäväksi. Se on alkuperäisessä määrittelyssä suunniteltu käyttöliittymälogiikasta selvästi erilliseksi komponentiksi, minkä ei tarvitse tietää mitään esimerkiksi käyttäjäistunnosta. Se ei myöskään tee loppukäyttäjien autentikaatiota, sillä autentikaatio tehdään Esityskerros-komponentilla. Vastuujaollisesti se on järkevää pitää tästä erillään.

Näistä syistä on parempi toteuttaa rajapinta Esityskerros-komponentille, joka voi puolestaan välittää kyselyn eteenpäin Palvelut-komponentille. Esityskerros voi tämän myötä toimia ylimääräisenä abstraktiokerroksena, mikä voi olla hyödyllistä, jos jokin palvelurajapinta ei suoraan ole käyttöliittymälle soveltuva. Lisäksi pystytään uudelleen käyttämään Esityskerros-komponentin Java-logiikkaa helpommin.

WebLogic-sovelluspalvelin tarjoaa JAX-RS-toteutuksen, jonka avulla on mahdollista toteuttaa HTTP-rajapintoja. Rajapinta tukee muun muassa XML- ja JSON-tietomuotoja, joista JSON on JavaScriptin näkökulmasta helppokäyttöisempi. Muunnos Java-luokkien ja JSON-muodon välillä tapahtuu automaattisesti, joten ei ole tarvetta tehdä niiden välisiä muunnoksia itse. [14, luku 29]

JAX-RS toteuttaa rajapinnat yksinkertaisina Java-luokkina, joissa kyselyihin liittyvät yksityiskohdat määritetään dekoraattorien avulla. Dekoraattoreilla esimerkiksi määrätään, minkä url-polun kyselyitä luokan metodi käsittelee. Rajapinnat voivat injektoida muita

Java-luokkia kuten JavaBeanit ja siten uudelleen käyttää aikaisempaa Java-toiminnallisuutta. [14, luku 29]

Toteutetut rajapinnat voivat käyttää täysin samoja käyttäjäistuntoja kuin JSF, joten autentikaatio on jo valmiiksi tehty. Auktorisointi tapahtuu tässä järjestelmässä aina viimeistään Palvelut-komponentin puolella. Roolivaatimuksia voidaan asettaa myös esityskerroksen rajapintoihin dekoraattorien avulla.

Koska Palvelut-rajapinnat on alun perin suunniteltu käyttöliittymäkerrokselle hyödyllisiksi, suurin osa voidaan suoraviivaisesti muuttaa vastaaviksi JSON-rajapinnoiksi. Koska serviceDelegate-luokat abstrahoivat nämä rajapinnat, JSON-rajapintatoteutus voidaan tehdä pienellä työllä. Rajapintoja voidaan kuitenkin tehdä myös muulle Esityskerroksen logiikalle ja niitä voidaan muuntaa paremmin selainsovelluksen tarpeille soveltuviksi.

6.2.2 Rajapintakommunikaatio (selain)

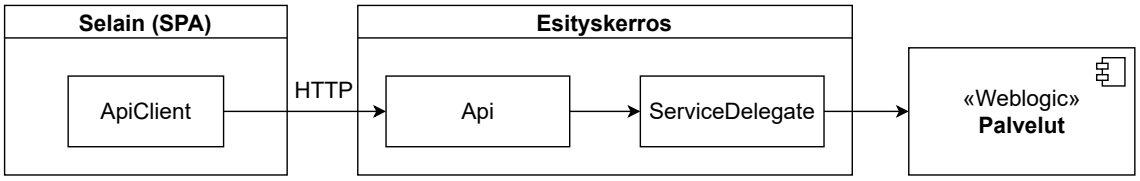
Luvussa 6.2.1 esitettiin menetelmä tehdä yksinkertaisia HTTP-rajapintoja, joiden kautta pääsee liiketoimintalogiikan palveluihin käsiksi. Koska rajapinta tukee HTTP:tä ja hyväksyy JSON-muotoista sisään- ja ulostuloa, sitä on helppo käyttää selaimesta yksinkertaisilla HTTP-kyselyillä.

Toisin kuin Esityskerroksen ja Palvelut-komponentin välisessä XML-kommunikaatiossa, selaimella ei ole vahvasti määriteltyä mallia, joka suoraan validoisi JSON-syötteen oikean muodon. Sekä JSON-syötteelle että JSON-ulostulolle on niitä vastaavat Java-luokat, joilla JSON-muunnokset tehdään automaattisesti. TypeScriptille voidaan näiden pohjalta tehdä vastaavat tyyppimääritykset. Jos tyyppimääritykset ovat oikeat ja riittävän tarkat, TypeScriptin staattinen analyysi voi huomauttaa mahdollisista virheistä jo ennen kuin koodia tarvitsee ajaa. Tyyppimääritykset täytyy tehdä sekä rajapintojen sisääntuloille että ulostuloille.

Java-luokkia vastaavien TypeScript-määrityksien teko käsin voi olla riskialtista ja aikaa vievää. Jos tyypeissä on kirjoitusvirhe, virhe voidaan havaita vasta ajon aikana. Jotkin virheet voivat pysyä kokonaan piilossa, jos ne eivät aiheuta koodissa poikkeuksia. Lisäksi voi tulla tilanteita, joissa rajapintamuutos päivitetään yhteen paikkaan, mutta ei toisiin. Tämän vuoksi tarvitaan varmempi ratkaisu tyypityksien generointiin.

TypeScript-generator-kirjasto on valmis ratkaisu tähän ongelmaan. TypeScript-generator pystyy automaattisesti löytämään JAX-RS-rajapintojen käyttämät Java-luokat ja generoimaan niitä vastaavat TypeScript-tyypitykset. Tämä tehdään sekä rajapintojen sisään- että ulostuloille. Tämän lisäksi kirjasto osaa generoida TypeScriptille client-luokat, jotka toteuttavat funktiot rajapintakyselyiden tekemiselle. Tämä varmistaa, että oikeiden tyypityksien lisäksi oikeita tyypityksiä käytetään oikeisiin rajapintoihin. [41]

Kuvan 6.3 kaaviossa on havainnollistettu, miten selaimen SPA-sovellus kommunikoi palvelimen palvelujen kanssa. Kaavio on yksittäisen rajapinnan näkökulmasta. Esimerkiksi työnantajatiedoille voi olla oma rajapinta, jolloin sille on oma ApiClient, Api ja ServiceDe-



Kuva 6.3. *Kaavio selaimen ja palvelujen välisestä kommunikaatiosta.*

legate. ServiceDelegate ja Palvelut ovat suoraan vanhasta järjestelmästä, eivätkä vaadi muutoksia. Esityskerroksen Api on uusi lisäys, jonka pohjalta generoidaan automaattisesti sitä vastaava ApiClient-luokka TypeScriptille.

Rajapintojen TypeScript-määrittelyt generoidaan aina Java-sovelluksen rakentamisen yhteydessä, jotta ne varmasti ovat ajan tasalla. Jos rajapintoihin tulee muutoksia, TypeScriptin tyyppitarkastuksien pitäisi useimmissa tilanteissa varoittaa niistä.

Liitteessä A on esimerkki erään rajapinnan palauttamasta Java-luokasta ja siitä generoiduista TypeScript-määrityksistä. Tyyppigenerointi luo myös kaikki viitatut tyypit. Molemmista koodiesimerkeistä on poistettu attribuuttikenttiä esimerkkien lyhentämiseksi.

Tyyppimäärityksissä täytyy varmistaa, että TypeScript-tyyppigeneraatio tulkitsee Java-luokat vastaavasti kuin JAX-RS muuntaa ne JSON-muotoon. Jos näin ei ole, generoidut tyypit eivät välttämättä ole täysin samassa muodossa kuin JSON, jota rajapinta odottaa. Tämän varmistamiseksi sekä TypeScript-generator että JAX-RS muunnetaan käyttämään Jackson JSON Parser -kirjastoa. Jackson valittiin JAX-RS:n ensisijaisen MOXY-kirjaston sijaan siksi, että se kokeilujen perusteella validoi JSON-rakenteen JAX-RS-rajapinnoissa paremmin. Vaikka vääränlainen JSON-olio tulisikin rajapinnalle, siitä saadaan varoitus.

7 PROTOTYYPPISUUNNITELMA

Diplomityössä toteutetaan kaksi prototyyppiä, jotka käyttävät eri teknologioita ja integraatiolähestymisiä. Teknologioiden osalta tutkintaan otetaan Angular ja React, joita käsiteltiin luvuissa 5.1.2 ja 5.1.3. Integraatiolähestymisissä tutkintaan otetaan luvuissa 6.1.1 ja 6.1.2 käsitellyt lähestymiset. Näistä ensimmäisessä tehdään JSF:stä täysin erillinen käyttöliittymäsovellus, kun taas toisessa uusi teknologia laitetaan JSF-sivun sisälle.

Angular on täysi sovelluskehys, joka valmiiksi antaa paljon työkaluja itsenäisen sovelluksen tekemiseen. Tämän vuoksi se soveltuu hyvin luvun 6.1.1 lähestymistavan testaamiseksi. Sovellus on itse vastuussa muun muassa sovellusrungon toteutuksesta ja reitityksestä.

React pienempänä kirjastona soveltuu luvun 6.1.2 lähestymiseen, koska sillä on helpompi tehdä yksittäisiä komponentteja, joita käytetään osana toista sovellusta. Vaikka tämä olisi teoriassa mahdollista myös Angularilla, React on tähän tarkoitukseen yksinkertaisempi.

Molemmissa prototyypeissä toteutetaan integraation vaatima taustatyö, jotta uutta ja vanhaa teknologiaa voi käyttää samassa järjestelmässä. Lisäksi molempiin toteutetaan konkreettisena esimerkkinä tarkasteltavasta järjestelmästä hakunäkymä, jonka avulla voi hakea järjestelmään tallennettuja työnantajia erilaisilla hakuehdoilla. Hakunäkymässä on monia dynaamisia ominaisuuksia ja haut ovat yksiä järjestelmän pääominaisuuksista, minkä vuoksi se soveltuu prototyypin testikohteeksi.

Prototyyppien tavoitteena on selvittää mahdollisia teknisiä haasteita ja löytää niihin ratkaisuja. Erityisesti keskitytään siihen, miten toiminnallisuuden saa siirrettyä vanhasta järjestelmästä ilman, että toiminta muuttuu merkittävästi. Aihetta tutkitaan uudelleensuunnittelun keinoin. Mahdollisia automatisointimahdollisuuksia myös tutkitaan.

Lopullisten prototyyppien ei tarvitse olla tuotantokelpoisia sovelluksia, vaikka prototyyppeihin toteutetaan oikea toiminnallisuus. Tavoitteena kuitenkin on, että yleisten ongelmien teknisiä ratkaisuja voi suoraan hyödyntää myös lopullisen migraation toteutuksessa.

8 ANGULAR-PROTOTYYPPI

Angular-prototyypissä tutkitaan lähestymistä, jossa JSF-järjestelmän rinnalle laitetaan itsenäinen Angular-sovellus. Yhdellä sivulla on joko JSF- tai Angular-teknologiaa, mutta ei molempia. Navigaatio näiden välillä tehdään eri muotoisilla poluilla. Angular voi hyödyntää jotain samaa Java-logiikkaa kuin JSF luvussa 6.2 käsiteltyjen HTTP-rajapintojen kautta, mutta muuta yhteistä niillä ei ole.

Luvuissa 8.1 ja 8.2 käydään läpi, miten prototyypin Angular-sovellus alustetaan ja miten se liitetään vanhaan järjestelmään. Luvussa 8.3 tutkitaan, miten JSF-tiedostojen näkymäkuvaukset siirretään vastaaviksi Angular-näkymiksi. Luvussa 8.4 tutkitaan, miten näkymämigraatiota voisi helpottaa automatisointityökaluilla. Luvuissa 8.5 ja 8.6 vastaavasti tutkitaan, miten JSF:n Java-toimintalogiikka voidaan siirtää Angularille ja pystyykö tätä helpottamaan automatisoinnin avulla.

8.1 Alkutoimenpiteet

Angularin projektipohjan tekemiseen käytetään Angularin virallista komentorivityökalua [42], joka automaattisesti tekee kaikki Angularin vaatimat määrittelytiedostot ja yksinkertaisen projektipohjan. Angularin komentorivityökalu tarjoaa paljon hyödyllisiä työkaluja. Työkalun ominaisuuksiin kuuluvat muun muassa koodin staattinen analyysi, komponenttipohjien automaattinen generointi ja Angular-versioiden automaattinen päivitys

Komentorivityökalu käyttää oletuksena Angularin uusinta versiota, joka prototyypin kehittämisen aikaan on versio 10. Komentorivityökalu ottaa useita argumentteja, joilla voi määrittää projektiin liittyviä asetuksia [43]. Näillä projektiin otetaan mukaan reititysmoduuli reitityksen määrittämiseksi, ”strict”-asetus tiukempia tyyppitarkastuksia varten ja SCSS tyylitiedostojen tyypiksi.

”strict”-asetuksen avulla TypeScript ja Angular tekevät tiukempia tyyppitarkastuksia, mikä auttaa mahdollisten virheiden löytämisessä. Tämä koskee sekä TypeScript-koodia että Angularin mallinetiedostoja. Tiukemmat tyypitykset auttavat myös staattista analyysia, joka auttaa Angularin automatisoitujen päivityksien refaktorointeja. [37]

SCSS (Sassy CSS) on CSS-laajennos, joka kääntyy tavalliseksi CSS-koodiksi. Se valitaan perinteisemmän CSS:n sijaan sen kattavampien ominaisuuksien vuoksi. Se on täysin yhteen sopiva tavallisen CSS:n kanssa, joten siirtymä niiden välillä on yksinkertainen. [44, 45]

Generoitu projektipohja käyttää oletuksena TSLint-työkalua koodityylin tarkasteluun ja staattiseen analyysiin. Oletusarvot ovat pääosin hyvät ja auttavat mahdollisten virheiden löytämisessä. TSLint ei nykypäivänä enää ole suositus, sillä TSLint-kehitys on siirtymässä ESLintin puolelle eikä tulevaisuudessa saa uusimpia ominaisuuksia [46]. Angularin staattisen analyysin työkalu vaihtuu tulevaisuudessa ESLintiksi [47], mutta tämä ei ole vielä tapahtunut. Staattisen koodianalyysityökalun pystyy vaihtamaan suoraviivaisesti, mutta tämän prototyypin osalta TSLint on riittävä.

8.2 Reititys sovelluksien välillä

Migraation aikana on kaksi erillistä käyttöliittymäsovellusta: Esityskerroksen JSF ja Angular. Migraatiossa näkymiä siirretään asteittain JSF-sovelluksesta Angular-sovellukseen. Loppukäyttäjien näkökulmasta järjestelmän pitäisi näyttää yhdeltä sovellukselta. Tämän vuoksi tarvitaan keino, jolla voidaan läpinäkyvästi siirtyä JSF- ja Angular-sivujen välillä.

Luvussa 6.1.1 tähän ongelmaan esitettiin ratkaisuna Nginx-palvelimen proxy-ominaisuuksia. Proxy-ominaisuuksilla voidaan polun mukaan reitittää kyselyitä eri kohteisiin. Tämä lähestyminen soveltuu tähän käyttötarkoitukseen, sillä järjestelmä käyttää tuotannossa Nginxiä WebLogic-sovelluspalvelimen edessä. Koska Nginx on myös WWW-palvelin, se voi hoitaa Angularin staattisten tiedostojen isännöinnin. Koska Angular on samalla isännällä kuin WebLogic, sovelluksien välisessä HTTP-kommunikaatiossa ei myöskään ole CORS-ongelmia.

Yksinkertaisin ratkaisu Nginx-reitityksien määrittelyyn on, että kaikki tietyn polun takana olevat reitit menevät Angular-sovellukselle ja muut WebLogic-sovelluspalvelimelle. Navigaatio voidaan tehdä yksinkertaisilla linkeillä. Kun näkymä siirretään JSF-sovelluksesta Angulariin, kaikki kyseiseen näkymään viittaavat linkit ja navigaatiot pitää muuttaa käyttämään uutta Angular-polkua. Teoriassa polut voisi pitää samoina tekemällä reitityksestä näkymäkohtaisesti, mutta tämä vaatisi hieman enemmän työtä.

Angular oletuksena olettaa, että sen reitit alkavat juuripolusta `/`. Jotta Angularin sisäinen reititys toimii oikein, sille pitää antaa oikea juuripolku asettamalla se HTML-tiedoston `<base href="...">`-elementtiin. [48]

Lokaalissa kehityksessä Nginx ei ole yhtä optimaalinen, sillä siinä ei voi helposti hyödyntää Angular-kehityspalvelimen hot reload -ominaisuuksia, joilla koodimuutokset päivittyvät avoinna olevaan sovellukseen automaattisesti. Angularin ja WebLogicin isännän pitää olla sama, jotta HTTP-kommunikaatio niiden välillä toimii oikein. Angularin kehityspalvelin tukee proxyjen määrittelyä, joten sen avulla on mahdollista reitittää kyselyitä WebLogicille [49]. Tätä voidaan käyttää sekä sivunavigaatioiden että HTTP-kyselyjen reitittämiseen.

Ohjelmassa 8.1 on yksinkertainen proxy-määrittely Angular-kehityspalvelimelle. Se välittää kaikki Angularin juuripolusta `/angular-client` poikkeavat kyselyt WebLogicin porttiin 8002. Koska halutaan reitittää kaikki paitsi Angularin omat polut, proxy on oletuksena kaikille kyselyillä, mutta `bypass`-funktio ohittaa sen Angular-kyselyille.

```
const PROXY_CONFIG = {
  "/": {
    // Oletuksena kaikki kyselyt proxytaan Weblogicin porttiin
    "target": "https://localhost:8002",
    "secure": false,
    "logLevel": "debug",
    "changeOrigin": true,
    "bypass": function (req, res, proxyOptions) {
      if (req.url.startsWith("/angular-client")) {
        // Polku on Angularin, joten ohitetaan proxy
        return req.url;
      }
    }
  }
}
module.exports = PROXY_CONFIG;
```

Ohjelma 8.1. *Angular-kehityspalvelimen proxy-määrittely, joka reitittää kaikki Angular-sivujen ulkopuoliset polut WebLogic-sovelluspalvelimen porttiin.*

Vaihtoehtoinen ratkaisu olisi, että WebLogic-sovelluspalvelin hoitaa myös Angular-tiedostojen isännöinnin ja reitityksen. Eräs tämän etu olisi, että käyttäjien autentikaatio voidaan tehdä jo ennen kuin käyttäjä pääsee Angularin HTML-sivulle. Tämän toteutus kuitenkin näyttäisi edellyttävän oman Java Servletin määrittelyä eikä suoraan tukisi Angular-sovelluspalvelimen kehittäjäystävällisempiä ominaisuuksia.

8.3 Näkymien migraatio

Sekä Angularissa että JSF:ssä näkymät toteutetaan HTML-kaltaisella mallinekielellä. JSF-sovelluksissa nämä ovat XHTML-tiedostoja ja Angularilla HTML-tiedostoja. Molemmat tukevat omia HTML-tagimäärittelyjä, ja voivat vaikuttaa niiden käyttäytymiseen attribuuteilla. Molemmissa mallineet voivat dynaamisesti viitata joko TypeScript- tai Java-koodiin. Molemmat loppujen lopuksi renderöidään HTML-koodiksi.

Mallinekielien samankaltaisuuksien vuoksi suurimman osan JSF-mallineista pystyy muuntamaan vastaviksi Angular-mallineiksi. Näiden välillä on kuitenkin joitain eroja, jotka täytyy ottaa huomioon.

8.3.1 Tagimäärittelyt

Vaikka JSF-mallineissa voi hyödyntää tavallisia HTML-tageja, JSF usein käyttää niiden sijaan omia tagejaan. Suurin osa näistä täsmää yksi yhteen vastaavien HTML-tagien kanssa, kuten taulukossa 8.1 on havainnollistettu. Attribuuttinimetkin voivat olla hieman erilaisia, mutta useimmille on täysin vastaavat HTML-versiot. Tagien vastaavat HTML-versiot voi selvittää joko tutkimalla JSF-sovellusta selaimien kehittäjätyökalujen inspect-ominaisuudella tai katsomalla niiden dokumentaatiota [14, 50].

Eräs syy sille miksi JSF käyttää omia tagejaan on, että niillä on tuki dynaamisemmille

Taulukko 8.1. *Esimerkkejä JSF-tageista ja niiden vastaavista HTML-versioista.*

JSF	HTML
`<h:inputText/>`	`<input type="text">`

`<h:outputText value="Test" styleClass="text"/>`	`<span class="text">`
	`  Test`
	`</span>`

`<h:selectOneMenu>`
`  <f:selectItem itemValue="1" itemLabel="Value 1"/>`
`  <f:selectItem itemValue="2" itemLabel="Value 2"/>`
`</h:selectOneMenu>`
	`<select>`
	`  <option value="1">`
	`  Value 1`
	`  </option>`
	`  <option value="2">`
	`  Value 2`
	`  </option>`
	`</select>`

ominaisuuksille. Esimerkiksi `rendered`-attribuuttin avulla elementti voidaan näyttää jonkin ehdon perusteella. Angular puolestaan pystyy täydentämään HTML-elementtien ominaisuuksia ilman, että sen tarvitsee tehdä niille omia elementtejään. Tämä tehdään Angularin direktiivien avulla, jotka syntaksiltaan vastaavat HTML-attribuutteja. Esimerkiksi Angularin mukana tuleva valmis direktiivi `*ngIf` toimii vastaavasti kuin JSF:n `rendered`. Tämän vuoksi Angular-mallineissa JSF-tagit voi useimmiten korvata suoraviivaisesti niiden vastaavilla HTML-versioilla.

Kaikille JSF-tageille ei ole yhtä yksinkertaista HTML-vastinetta. Tästä eräs esimerkki on `selectManyCheckbox`, joka renderöi HTML-taulukon ja sen sisälle useita `checkbox`-elementtejä [50]. Tämänkin voisi siirtää Angular-mallineeseen vastaavana HTML-koodina, mutta siihen liittyvä logiikka vaatisi enemmän työtä vastaavaan toiminnallisuuteen. Komponenttia käytetään useammassa paikassa, joten työ pitäisi tehdä moneen paikkaan. Tällaisissa tilanteissa voidaan tehdä JSF-tagin toiminnallisuutta vastaava Angular-komponentti. Kun komponentti on yhden kerran tehty, sitä on helppo käyttää.

Tarkasteltavassa järjestelmässä käytetään myös PrimeFaces-komponenttikirjastoa. Tälle on olemassa Angular-versio, jolla on vastineet suurelle osalle komponenteista [51]. Komponenteissa on joitain pieniä eroja, mutta niitä voidaan silti hyödyntää migraatiossa.

8.3.2 Dynaamisuus ja interaktiivisuus

Sekä JSF- että Angular-mallineet tukevat dynaamisuutta ja vuorovaikutusta sovelluksen tilan kanssa. JSF-mallineissa voidaan viitata Java-koodiin, ja Angular-mallineissa vas-

taavasti TypeScript-koodiin. Näkymä voi päivittyä tilan muutoksien myötä. Käyttöliittymän tapahtumat voivat aktivoida funktiokutsuja, jotka puolestaan voivat muuttaa sovelluksen tilaa tai esimerkiksi aktivoida tietokantaoperaatioita.

JSF ja Angular toteuttavat näkymän ja toimintalogiikan välisen vuorovaikutuksen vastaavalla tavalla, mutta hieman eri syntaksilla. Ohjelmassa 8.2 on esimerkki yksinkertaisesta JSF-lomakkeesta. Ohjelmassa 8.3 on vastaava esimerkki Angularilla.

```
<h:form id="exampleForm">
  <h:inputText value="#{exampleBean.value}" />
  <h:commandButton value="Submit"
                   action="#{exampleBean.submit()}">
    <f:ajax render="exampleForm result"/>
  </h:commandButton>
</h:form>
<h:outputText id="result"
              styleClass="#{exampleBean.isError() ? 'error': 'result'}"
              rendered="not empty result"
              value="#{exampleBean.result"/>
```

Ohjelma 8.2. *Esimerkki yksinkertaisesta JSF-lomakkeesta.*

```
<form id="exampleForm>
  <input [(ngModel)]="value" />
  <button (click)="submit()">Submit</button>
</form>
<span id="result"
      [class]="isError() ? 'error' : 'result'"
      *ngIf="result">
  {{result}}
</span>
```

Ohjelma 8.3. *Esimerkki yksinkertaisesta Angular-lomakkeesta.*

JSF-mallineiden käyttämässä EL-syntaksissa dynaamisiin arvoihin voidaan viitata joko `${lauseke}`-syntaksilla tai `#{lauseke}`-syntaksilla. Näistä `#{lauseke}` on JSF:n kanssa tyypillisemmin käytetty [14]. Tarkasteltavassa järjestelmässä käytetään pääosin `#{lauseke}`-syntaksia, joten tässä ei mennä sen syvällisemmin näiden eroavaisuuksiin.

Dynaamisten arvojen syntaksia voi käyttää monessa asiayhteydessä. Esimerkin `inputText`-tagin tapauksessa välitetään kaksisuuntainen arvoviittaus, jossa syöttökentän muutokset näkyvät `value`-arvossa ja toisin päin. `commandButton`-tagin tapauksessa EL-syntaksilla välitetään metodikutsu, jota kutsutaan nappia painettaessa. `outputText`-kentän tapauksessa määritetään dynaamisesti, näytetäänkö komponenttia, ja mitä CSS-luokkaa sillä käytetään.

Angularissa käytetään eri asiayhteyksissä hieman erilaista syntaksia. Yksinkertainen interpolaatio tehdään `{{lauseke}}`-syntaksilla, kuten `span`-elementissä näkyy. Interpolaation lisäksi dynaamisuutta voi olla tagien attribuuteissa. Näissä dynaamisuus on jaoteltu

erikseen komponenttien sisään- ja ulostuloille. Sisääntulot ovat komponentin vastaanottamia arvoja, ja ne merkitään `[attribuutti]="lauseke"`-syntaksilla. Ulostulot, kuten esimerkiksi klikkaustapahtuma, merkitään `(attribuutti)="lauseke"`-syntaksilla. Jos komponentti halutaan aina pitää synkronoituna arvon kanssa, se voidaan tehdä `[(attribuutti)]="lauseke"`-syntaksilla. Tavallisille HTML:n `input`-elementeille kaksisuuntainen arvojen sitominen tehdään Angularin antaman `ngModel`-direktiivin kautta. Hieman erilaisesta syntaksista huolimatta nämä käyttäytyvät hyvin vastaavasti kuin EL-syntaksissa. [33]

JSF- ja Angular-mallineiden syntakseissa on pieniä eroja loogisten operaattorien suhteen. JSF esimerkiksi voi käyttää `&&`- ja `||`- operaattorien sijaan `and`- ja `or`-avainsanoja [14]. Kaikki yleiset loogiset ja matemaattiset operaatiot löytyvät molemmista, joten logiikka ei vaadi suurempia muutoksia näiden välillä.

Eräs ero JSF:n ja Angularin dynaamisuuden välillä on tietojen synkronoinnissa. JSF:n tapauksessa toimintalogiikka on pääosin palvelimen puolella. Tämän vuoksi muutokset selaimen käyttöliittymän ja palvelimen välillä pitää välittää Ajax-kyselyillä. Operaatioiden pitää erikseen kertoa, mitkä komponentit täytyy renderöidä uudestaan. Esimerkkiohjelmassa 8.2 `actionButton`-napin toiminnolle kerrotaan erikseen, että `exampleForm`- ja `result`-elementit renderöidään toiminnon suorituksen jälkeen.

Angularilla käyttöliittymän tila ja toimintalogiikka ovat lokaalisti selaimessa, minkä vuoksi ei tarvita JSF:n `<f:ajax/>`-tageja. Angularissa on automaattinen muutoksien tunnistus, joka automaattisesti osaa päätellä mitä käyttöliittymässä pitää päivittää tilan muuttuessa [33]. Kehittäjän ei tarvitse erikseen kertoa, mitä käyttöliittymässä pitää päivittää.

8.3.3 Tyylimäärittelyt

JSF ja Angular käyttävät CSS-kieltä tyylimäärittelyiden tekemiseen. Molemmissa määrittelyt tyypillisesti ovat omissa tiedostoissaan. Jos Angular-version lopullisen HTML-ulostulon rakenne on vastaava kuin JSF:n kanssa, samojen tyylimäärittelyiden pitäisi toimia sellaisenaan. Tyylimäärittelyiden suositellussa sijoituksessa on kuitenkin joitain eroja.

JSF:ssä CSS-määrittelyt ovat perinteisiä CSS-tiedostoja, jotka ovat globaalisti voimassa koko sivulle. Angularillekin voi määrittää sivun näkökulmasta globaaleja tyylisääntöjä, mutta se ei ole yhtä suositeltavaa.

Angularilla on mahdollista toteuttaa komponenttikohtaisia CSS-sääntöjä. Toisin kuin perinteisissä CSS-tiedostoissa, komponenttikohtaisten sääntöjen näkyvyysalue on rajattu kyseiseen komponenttiin. Tässä etuna on, että komponenttiin liittyvät säännöt löytyvät helpommin eikä tarvitse etsiä missä muualla samoja sääntöjä käytetään. [52] Tämä on Angularin virallinen suositus [53].

Tarkasteltavassa järjestelmässä suurin osa tyylisäännöistä on määritelty yhdessä globaalissa CSS-tiedostossa. Nopein tapa siirtää tyylit on ottaa tämä käyttöön myös Angular-versiossa, mutta se voi hankaloittaa ylläpitoa pidemmällä aikavälillä. Tämä voidaan ratkoa kopioimalla tyylisääntöjä komponenttikohtaisiin tyylitiedostoihin sitä mukaan, kun kompo-

nentteja tehdään.

Varmin tapa löytää kaikki komponentin käyttämät tyylit on katsoa selaimen kehittäjätyökalujen inspect-työkalulla mitä sääntöjä komponentin alielementit käyttävät. Inspect-työkalua voidaan myös käyttää varmistamaan, että kaikki tyylimäärittelyt on siirretty oikein. Tämän automatisaatioon palataan luvussa 8.4.

Eräs ero JSF- ja Angular-komponenttien välillä on, että Angular-komponentin juuritagi on osana komponenttipuuta. Jos Angular-komponentin nimi esimerkiksi on "example-form", selaimen DOM-rakenteeseen tulee `<example-form>`-elementti. Juuritageilla ei oletuksena ole mitään tyylimäärityksiä, joten niillä ei yleensä ole vaikutusta. Tämä saattaa aiheuttaa joitain pieniä eroja joissain tyylisäännöissä, mikä on hyvä huomioida.

Tarkasteltavassa järjestelmässä käytetty PrimeFaces-kirjaston versio käyttää hieman erilaisia komponenttirakenteita kuin sen vastaava Angular-versio. Komponentit esimerkiksi käyttävät erilaisia CSS-luokkanimiä ja oletustyylejä. Tämän vuoksi järjestelmän PrimeFaces-komponentteja varten tehdyt CSS-säännöt eivät sellaisinaan toimi, vaan pitää toteuttaa uudestaan.

8.3.4 Komponenttimäärittelyt

JSF ja Angular tukevat omien komponenttien määrittelyjä, joita voi käyttää kuten tavallisia HTML-elementtejä. JSF:llä on kaksi tapaa tehdä omia komponentteja: laajentamalla `UIComponent`-Java-luokkaa, tai hyödyntämällä composite-tagikirjastoa [14, luku 15]. Tarkasteltava järjestelmä käyttää vain jälkimmäistä, joten ensimmäistä vaihtoehtoa ei tässä tarkastella sen enempää.

composite-tagikirjaston avulla komponentit voidaan määritellä XHTML-tiedoston sisällä. Tiedostoon määritellään sekä komponentin rajapinta että sen sisältö. Liitteen B ohjelmassa B.1 on esimerkki yksinkertaisesta lomakekomponentista. Komponentti saa viittauksen syötekentän arvoon, syötekentän `label`-elementin tekstin ja toiminnon, joka ajetaan nappia painettaessa. Rajapinnan jokaiselle attribuutille voi määrittää arvon tyypin ja pakollisuuden, mutta niitä ei ole pakko antaa.

Angularissa komponenttien määrittely perinteisesti jakautuu kolmeen tiedostoon: HTML-tiedosto, CSS-tiedosto ja komponentin TypeScript-tiedosto [33]. Käsitteellisesti Angularin HTML-tiedosto vastaa JSF-tiedoston `composite:implementation`-osuutta. Rajapinnat määritellään Angularissa komponentin TypeScript-luokassa, mikä vastaa pitkälti JSF:n `composite:interface`-määrittelyitä. Erona on, että Angularissa täytyy erikseen määrittää, mitkä ovat sisään- ja mitkä ovat ulosmeneviä arvoja.

Liitteen B ohjelmissa B.2 ja B.3 on vastaava Angular-esimerkki ohjelman B.1 lomakekomponentille. Ensimmäisessä on komponentin TypeScript-tiedosto ja toisessa sen HTML. TypeScript-tiedostossa määritellään JSF-attribuutteja vastaavat ominaisuudet. Eräs ero on, että `value`-argumentille täytyy määrittää sekä `Input` että `Output`, jotta arvon muuttuminen komponentin sisällä voi päivittää myös alkuperäisen arvon. Angularissa kompo-

nentin `Input`-kentille ei voi asettaa pakollisuutta, mikä tässä on kierretty antamalla niille oletusarvot. Jos pakollisuutta haluaa erikseen varmistaa, se pitää toteuttaa itse.

Angular-komponenteissa ei voi viitata globaaleihin muuttujiin kuten JSF-esimerkissä `messageBean`. Kaikkien viitattujen arvojen täytyy olla määritelty komponenttiluokkaan. Tässä esimerkissä `messageBean` on korvattu `messageServicellä`, jonka TypeScript-komponenttiluokka saa parametrina Angularin riippuvuusinjektion kautta.

Eräs komponenteissa huomioitava asia on, miten JSF ja Angular käsittelevät id-arvot. JSF:ssä id-arvo ei välttämättä ole selaimessa se, mitä XHTML-tiedostossa on annettu. Sen sijaan JSF generoi elementeille dynaamisesti id-arvot, joka on yhdistelmä tiettyjen säiliöelementtien id-arvoista ja elementin omasta id-arvosta. Jos esimerkiksi ohjelman B.1 form-elementille annettaisiin id `simpleForm`, `input`-elementin id:n loppuosa olisi `simpleForm:input`. Jos komponenttia käytetään toisen säiliöelementin sisällä, sen id tulisi myös sen osaksi. Koska tässä `form`-elementille ei annettu id:tä, sille generoidaan uniikki tunniste automaattisesti [14, luku 10.2.1.1].

Angularin id-arvoilla ei ole tällaista käyttäytymistä. Koska komponenttien id-arvojen pitäisi olla ainutkertaisia saman sivun sisällä, saman sivun sisällä useaan kertaan käytetyissä komponenteissa ei saisi olla staattisia id-arvoja. Tämän takia ohjelman B.2 esimerkissä komponentti ottaa vastaan myös id-arvon.

JSF-mallineissa id-arvoja tarvitaan usein, koska niitä käytetään sivujen Ajax-päivityksissä kuten ohjelmassa 8.2 näytettiin. Angularissa id-arvoja tarvitaan harvemmin, joten suurimmassa osassa tapauksia id-arvot voidaan poistaa kokonaan. SimpleForm-esimerkissä id-arvoa tarvitaan `label`- ja `input`-elementtien kytkemiseksi toisiinsa, joten tässä tilanteessa se oli tarpeen.

Syntaksieroavaisuuksista huolimatta Angular- ja JSF-komponentit voivat tehdä hyvin samankaltaisia asioita, joten migraatiossa omat JSF-komponentit voi useimmiten korvata suoraviivaisesti Angular-komponenteilla.

8.3.5 Validointi ja arvomuunnokset

JSF tarjoaa validointi- ja arvomuunnostageja, joilla voidaan validoida tai muuntaa syötekenttien arvoja. Näistä eräitä esimerkkejä ovat `validateLength` ja `convertNumber`. Omien validaattorien ja arvomuuntimien teko on myös mahdollista. Ohjelmassa 8.4 on yksinkertainen esimerkki oman validaattorin ja arvomuuntimen käyttämisestä sekä virheviestin näyttämisestä.

Angular ei täydennä elementtien ominaisuuksia tageilla, vaan käyttää sen sijaan direktiivejä. Angular tukee suoraan omien validaattoridirektiivien toteutusta [54]. Arvomuunnoksille Angular ei tarjoa yhtä suoraa pohjaa, mutta niillekin pystyy tekemään vastaavan toiminnallisuuden direktiiveillä [55]. Vaihtoehtoisesti arvomuunnokset voi tehdä toimintalogiikan puolella tavallisilla funktioilla. Ohjelmassa 8.5 on vastaava esimerkki Angularilla kuin ohjelmassa 8.4.

```
1 <h:inputText id="nameInput" value="#{exampleBean.name}">
2     <f:validator validatorId="com.myapp.NameValidator" />
3     <f:converter converterId="com.myapp.NameConverter" />
4 </h:inputText>
5 <h:message for="nameInput" class="error" />
```

Ohjelma 8.4. *Validointi- ja arvomuunnosesimerkki JSF:llä.*

```
1 <!-- Kentän malli tallennetaan nameInput-muuttujaan -->
2 <input #nameInput="ngModel"
3         [(ngModel)]="name"
4         myAppNameValidator
5         myAppNameConverter />
6 <div *ngIf="nameInput.invalid && nameField.errors.name" class="error">
7     {{nameInput.errors.name.errorMessage}}
8 </div>
```

Ohjelma 8.5. *Validointi- ja arvomuunnosesimerkki Angularilla.*

Syntaksit ovat hieman erilaiset, mutta Angularin direktiiveillä pystyy toteuttamaan vastaavat toiminnallisuudet kuin JSF:n validaattoreilla ja arvomuuntimilla.

8.4 Näkymien migraation automatisointi

Kuten edeltävissä luvuissa on käsitelty, useimmille JSF-näkymien ominaisuuksille löytyy vastaavat Angular-versiot. Ne ovat myös rakenteiltaan hyvin samankaltaisia. Vastaavista rakenteistaan huolimatta ne kuitenkin käyttävät erilaisia taginimiä, attribuuttinimiä ja paikoitellen erilaista syntaksia. Rakenteesta voi ottaa mallia vastaavaa Angular-koodia kirjoittaessa, mutta varsinainen koodi pitää käytännössä kirjoittaa tyhjästä.

Tässä luvussa katsotaan kolmea kokeilua, joita prototyypin toteutuksen aikana tutkittiin näkymälogiikan siirron helpottamiseksi. Kokeilujen tavoitteena on vähentää manuaalista kirjoitustyötä. Prosessin osittainen automatisointi voisi merkittävästi helpottaa laajan järjestelmän siirtotyötä. Tässä vaiheessa aihetta tarkastellaan erityisesti mallinetiedostojen näkökulmasta.

8.4.1 Tapa 1: Generoidun HTML:n hyödyntäminen

Ensimmäisessä yrityksessä yritettiin lähdekoodin sijaan käyttää JSF:n generoimaa HTML-koodia. Koodin saa helposti esille selaimen kehittäjätyökalujen inspect-ominaisuudella. Generoitu HTML-koodi on paljon lähempänä Angular-koodia oikeiden taginimien vuoksi. Ongelmana tässä lähestymisessä on, että lähes kaikki näkymään liittyvä logiikka ja dynaamiset arvomääritykset katoavat generoinnissa. Nämä voi katsoa JSF-koodista ja laittaa manuaalisesti HTML-tiedostoon, mutta tässä voi helposti tulla virheitä.

Generoidussa HTML-koodissa on myös JSF:n omia metatietoja, jotka pitäisi poistaa manuaalisesti. Generoidut satunnaiset id-arvot ja tapahtumakäsittelijät ovat näistä eräitä esi-

merkkejä. Generoitua HTML-koodia voisi teoriassa käyttää nopeisiin prototyyppikokeiluihin, joissa oikeaa toiminnallisuutta ei tarvita. Lopullisen toteutuksen työkaluksi se ei kuitenkaan sovellu edellä mainituista syistä, ellei sisältö ole hyvin staattista.

8.4.2 Tapa 2: Taginimien korvaus RegExin avulla

Toisessa yrityksessä tehtiin oma skripti, joka RegExin (Regular Expressions) avulla muuntaa JSF-lähdekooditiedoston taginimiä vastaaviksi HTML-tageiksi. Koska muunnokset ovat pääosin säännöllisiä ja useimmille tageille on yksikäsitteinen HTML-tagi. Tämä antoi parempia tuloksia, mutta ei riittävästi.

Taginimien lisäksi elementeillä on eroavaisuuksia attribuuttien suhteen. Jotkin tagimuunnokset voivat myös olla riippuvaisia tagin isäntäelementistä tai sen attribuuteista. Attribuuttien ja monimutkaisempien tapauksien käsittely RegExillä olisi hankalaa ja virhealtista.

8.4.3 Tapa 3: XML-puurakenteen muuntaminen

Edellisessä luvussa käsitelty RegEx ei ollut riittävän joustava, joten tutkittiin tapaa tutkia rakennetta syvemmällä tasolla. JSF käyttää XHTML-koodia, joka puolestaan on XML-muodossa. Koodin pystyy siis lukemaan XML:nä. Tällä ajatuksella toteutettiin TypeScript-skripti, joka pystyy lukemaan lähdekoodin XML:nä ja muuntamaan sen Angularia vastaavaksi koodiksi. Skripti on korkealla tasolla seuraavanlainen:

1. Lue argumentissa annetun tiedoston sisältö.
2. Muunna tiedoston XML JavaScript-oliksi. Näin saadaan XML puumaisena rakenteena.
3. Muunna XML-olion taginimi ja attribuutit niiden HTML-vastineeksi. Jokaiselle taginimelle on omat muunnossäännöt.
4. Muunna XML-olion alielementit rekursiivisesti (palaa kohtaan 3).
5. Palauta XML-olio XML-koodiksi ja kirjoita tiedostoon.

XML-muunnoksissa käytettiin xml-js-kirjastoa [56], joka pystyy muuntamaan XML-koodin JavaScript-olioksi ja toisin päin. Toisin kuin muut kokeillut JavaScriptin XML-kirjastot, xml-js säilyttää elementtien järjestyksen. Jos kirjaston muuntaman XML-koodin antaa sille takaisin, se antaa sen vastaavana kuin alkuperäisen koodin. Tämän vuoksi se soveltuu myös XHTML-koodin lukemiseen ja generointiin.

Skripti hyödyntää rekursiota koko dokumenttipuun muuntamiseksi. Rekursiofunktio katsoo elementin taginimen perusteella millaisia muunnoksia sille pitää tehdä. Vastaava tehdään rekursiivisesti elementin alielementeille. Ohjelmassa 8.6 on rekursiivisen funktion runko ja esimerkkinä `selectBooleanCheckbox`- ja `outputLabel`-elementtien muunnoksien määrittelyt.

```
function parseElement(element: Element): Element | Element[] | null {
  if (element.type !== 'element') {
    return element;
  }
  const { name } = element;
  switch (name) {
    // Esimerkki:
    // <h:outputLabel value="Otsikko" />
    // <label>Otsikko</label>
    case 'h:outputLabel':
      return parseElementBase(element, {
        name: 'label',
        innerTextAttribute: 'value',
        attributeMappings: {
          ...defaultAttributeMappings,
          // null-attribuutit poistetaan
          value: null
        }
      });
     // Esimerkki:
     // <h:selectBooleanCheckbox value="#{exampleBean.value}" />
     // <input [(ngModel)]="exampleBean.value" type="checkbox" />
    case 'h:selectBooleanCheckbox':
      return parseElementBase(element, {
        name: 'input',
        attributeMappings: {
          ..defaultAttributeMappings,
          value: '[(ngModel)]'
        },
        additionalAttributes: {
          type: 'checkbox'
        }
      });
    // ...
  }
  // Oletus: Vain perusmuunnokset
  return parseElementBase(element, {})
}
```

Ohjelma 8.6. *Muunnosmääritykset label- ja checkbox-elementeille.*

Suurin osa elementeistä käyttää hyvin samankaltaisia muunnoksia. `parseElementBase` on apufunktio, joka ajaa yhteiset sekä argumenttina määritellyt muunnokset. Tämän lisäksi se kutsuu `parseElement`-funktiota kaikille alielementeille, toteuttaen rekursion. Funktiolle annetut yleisimmät muutokset ovat seuraavat:

- Taginimen vaihto
- Attribuuttinimien vaihto
- Attribuuttien poisto
- Attribuuttien lisäys
- Attribuuttiarvon siirto elementin sisälle

Taginimen vaihto on muunnoksista yleisin, joka on käytössä lähes kaikille elementeille.

Esimerkiksi `inputCheckbox`-elementille vastaava nimi on `input` ja `outputLabel`-elementille `label`. Taginimien vastaavuuksia on tarkemmin käsitelty luvussa 8.3.1.

Attribuuttinimien vaihdossa vaihdetaan JSF-attribuutin nimi vastaavaan Angular-vastineeseen. Esimerkiksi `inputText`-elementin `value`-attribuutin Angular-vastine on `[(ngModel)]`. Jotkin yleiset attribuuttinimimuunnokset on määritelty kaikille oletuksena. Näistä eräitä esimerkkejä ovat `rendered` => `*ngIf`, `styleClass` => `class` ja `onclick` => `(click)`.

Joissain tilanteissa JSF-käyttää attribuutteja, joita Angular ei tarvitse. Vastaavasti Angular voi tarvita attribuutteja, joita JSF ei käytä. Esimerkiksi PrimeFaces-kirjaston `commandButton`-komponentin `update`-attribuuttia käytetään elementtien päivittämiseen Ajaxilla, mitä Angularissa ei tarvitse tehdä. Käänteisesti HTML:ssä `checkbox`-elementit ovat `input`-elementtejä, joilla on `type`-attribuutin arvona `checkbox`. JSF käyttää `checkbox`-elementeille erillistä `selectBooleanCheckbox`-elementtiä, jolla `type`-attribuuttia ei tarvitse olla ollenkaan.

Jotkin JSF-elementit käyttävät hieman erilaista rakennetta kuin niiden vastaavat HTML-versiot. Esimerkiksi `outputLabel`-elementissä tekstisisältö määritellään `value`-attribuutilla, kun taas HTML:n `label`-elementillä tekstisisältö on elementin sisällä.

Edellä mainittujen muunnoksien lisäksi joitain muunnoksia tehdään kaikille elementeille. Eräs esimerkki on dynaamisten arvojen syntaksien muuntaminen. Dynaamisten arvojen syntakseja on käsitelty luvussa 8.3.2. Jos attribuuttiarvossa käytetään JSF:n dynaamisten arvojen `#{}`-syntaksia, skripti muuttaa attribuuttinimen muotoon `[attribuutti]` lisäämällä attribuuttinimen ympärille hakasulkeet. Tämä ei ole kaikille attribuuteille oikea syntaksi, sillä joillain sen kuuluu olla `(attribuutti)` tai `[(attribuutti)]`. Tiedetyille tapauksille tämä voidaan korjata skriptin attribuuttimuunnoksilla, joita edellä käsiteltiin. Väärät sulkeet tulevat vastaan viimeistään silloin, kun kehittäjä käy koodia läpi. Tekstin joukossa olevat dynaamisten arvojen `#{}`-syntaksit vastaavasti korvataan Angularin `{{}}`-syntaksilla.

Jotkin elementit ovat monimutkaisempia, eivätkä edellä mainitut yksinkertaiset muunnokset riitä. Esimerkiksi luvussa 8.3.5 käsitellyille validaattoreille ei löydy suoria HTML-tagivastaavuuksia, ja joillekin elementeille muunnokset riippuvat asiayhteydestä. Tällaisille tapauksille on lisätty omaa logiikkaa, joka tekee elementille tai sen alielementeille tarvittavia monimutkaisempia muutoksia.

Koska kaikille elementeille pitää tehdä tapauskohtaiset muunnosmäärittelyt, skripti ei välttämättä kata kaikkia JSF-elementtejä. Tällaisissa tapauksissa skripti jättää ne generoituun skriptiin. Angular ei tunnista JSF-tageja, minkä seurauksena niistä tulee kehittäjälle varoituksia. Kehittäjä voi näin manuaalisesti käydä nämä tapaukset läpi ja päättää miten niiden kanssa tehdään. Jos tapaus on riittävän yleinen, kehittäjä voi täydentää muunnosskriptiä käsittelemään myös sen.

Ohjelmissa C.1 ja C.2 on laajempi esimerkki skriptin tekemistä muunnoksista. Ensimmäisessä on alkuperäinen JSF:n XHTML-tiedosto, ja toisessa siitä generoitu HTML-tiedosto.

Esimerkeistä on otettu pois tiedoston alussa olevat HTML-määrittelyt esimerkkien lyhentämiseksi.

Ohjelmaesimerkissä kaikki JSF-koodi on saatu muunnettua vastaavaksi Angular-koodiksi. Osan JSF-koodista voi poistaa kokonaan, kuten `f:ajax`-elementit ja `update`-attribuutit. Tekstikentän `validateLength`-validaattorin pystyi poistamaan, sillä Angular tekee vastaavan suoraan `maxlength`-attribuutin kautta. `NameConverter` on oma muuntajatoteutus tekstille, jolle pystyi tekemään vastaavan Angular-direktiivin. Tämän myötä se siirtyy alielementistä tekstikentän attribuutiksi. `ui:repeat`-elementille ei ole suoraa HTML-vastinetta, mutta tässä tilanteessa vastaava toiminnallisuus saadaan toistettua laittamalla vastaava `ngFor`-direktiivi sen ainoalle alielementille. `gof:help` on itse tehty JSF-komponentti, jolle on toteutettu Angularia varten vastaava `gof-help`-komponentti.

Jos ohjelman Angular-komponentille määriteltäisiin vastaavat `exampleBean`- ja `i18n`-ominaisuudet, näkymäkoodi ei vaatisi tässä yksinkertaisessa tapauksessa ollenkaan muutoksia. Kaikki tapaukset eivät kuitenkaan ole yhtä yksinkertaisia, minkä vuoksi on joitain muunnoksia, joita ei voi täysin automatisoida. Automatisointi on myös rajoitettu tapauksiin, jotka on erikseen määritelty. Jos tulee vastaan elementti tai elementin erikoiskäyttö, jota ei ole skriptiin erikseen määrätty, kehittäjän täytyy ratkaista tilanne manuaalisesti.

Ohjelman C.2 CSS-analyysiin palataan tarkemmin seuraavassa luvussa.

8.4.4 CSS-sääntöjen poiminta

Luvussa 8.3.3 käsiteltiin JSF:n ja Angularin erilaisia lähestymistapoja CSS-tyylisääntöihin. JSF perinteisemmin käyttää yhteisiä CSS-tiedostoja, kun taas Angularissa suositaan komponenttikohtaisia tyylimäärittelyitä. Erityisesti tarkasteltavassa järjestelmässä suurin osa tyylimäärittelyistä on tehty yhdessä suuressa tiedostossa, joka kattaa koko sovelluksen. Vaikka Angular voisi myös käyttää isoa tiedostoa, se ei olisi sen parhaiden käytäntöjen mukaista.

Komponenttikohtaisten tyylisääntöjen poiminta manuaalisesti voi olla työlästä, sillä kaikki säännöt eivät ole valmiiksi ryhmiteltyjä. Tämän vuoksi otetaan tutkintaan, miten tätä pystyisi automatisoimaan.

Luvussa 8.4.3 saatiin muunnettua JSF:n XHTML-koodi puurakenteeksi, jota on helpompi käsitellä. Tätä kautta voi helposti kerätä kaikki tiedoston käyttämät `class`- ja `id`-määrittelyt rekursiivisesti iteroimalla.

CSS:n säännöllisen rakenteen vuoksi on mahdollista lukea kaikki CSS-tiedoston tyylisäännöt ja erottaa niistä valitsijat (selector), kuten `class` ja `id`. CSS-tiedostosta saatiin muodostettua hakurakenne, jossa avaimina ovat `class`- ja `id`-arvot. Avaimien arvoina ovat kaikki CSS-säännöt, joissa ne ovat mukana.

Kun yhdistää JSF-tiedostosta löydetyt `class`- ja `id`-arvot ja CSS-tiedostosta saadut valitsin–säännöt-parit, voidaan saada kaikki tiedoston käyttämät CSS-säännöt, joissa on mukana

`class` tai `id`. Tämä toiminnallisuus lisättiin osaksi luvun 8.4.3 skriptiä, jonka ulostulosta on esimerkki ohjelmassa C.2. Kaikki löydetyt tyylisäännöt lisätään generoidun HTML-tiedoston loppuun kommentissa.

Skripti ei löydä sääntöjä, joissa ei viitata `class`- tai `id`-arvoon. Mukaan ei siis tule pelkästään HTML-elementin nimeen viittaavia sääntöjä, kuten `h1`. Tällaiset säännöt kuitenkin ovat harvinaisempia, ja ne voi olla sopivampi määrittää globaalissa tyylitiedostossa. Jos näihin liittyviä visuaalisia eroja tulee vastaan, tilanteeseen liittyvät säännöt voi löytää esimerkiksi selaimen kehittäjätyökalujen avulla.

8.4.5 Yhteenveto

Tässä luvussa tutkittiin keinoja, joilla JSF:n näkymäkoodin muuntamista Angularin vastaavaksi koodiksi saisi automatisoitua ainakin osittain. Vaihtoehtoina tutkittiin aluksi JSF:n generoiman HTML-koodin tarkastelua ja RegExin hyödyntämistä, mutta näitä ei koettu tarpeeksi tehokkaiksi keinoiksi tähän tarkoitukseen.

Automatisaatiolle saatiin toteutettua skripti, joka JSF-tiedostojen XML-rakennetta tutkimalla pystyy rekursiivisesti muuntamaan JSF-elementit niitä vastaaviksi Angular-elementeiksi. Toteutetulla skriptillä pystytään helposti tekemään yleisimpiä muunnoksia, ja tarvittaessa myös monimutkaisempia rakenteellisia muutoksia. Kaikki mahdolliset tapaukset täytyy määritellä skriptille erikseen, minkä vuoksi skriptiä tarvitsee täydentää sitä mukaan, kun migraatiossa tulee uusia tilanteita vastaan.

Toteutetulla automaatioskriptillä pystyy HTML-koodigeneroinnin lisäksi löytämään useimmat tiedoston käyttämistä CSS-säännöistä. Tämä auttaa erottamaan tyylisäännöt komponenttikohtaisesti.

Näkymien migraatio tulee skriptistä huolimatta vaatimaan manuaalista tarkastelua, sillä kaikkea ei voi automatisoida täydellisesti. Toteutettu skripti on kuitenkin hyödyllinen työkalu, jonka pitäisi helpottaa ja nopeuttaa migraation tekemistä merkittävästi.

8.5 Toimintalogiikan migraatio

Tässä luvussa käsitellään, miten JSF:n toimintalogiikka voidaan muuntaa vastaavaksi Angular-koodiksi. Toimintalogiikaksi tässä tulkitaan kaikki Javalla tai TypeScriptillä kirjoitettu koodi. Mallinetiedostojen muuntamista käsiteltiin luvussa 8.3.

Luvussa 8.5.1 käydään läpi JSF:n ja Angularin käsitteitä ja katsotaan, miten ne liittyvät toisiinsa. Luvussa 8.5.2 katsotaan eroja JSF:n ja Angularin tilanhallinnassa ja miten vastaavan toiminnallisuuden voi tehdä Angularilla. Luvuissa 8.5.3 ja 8.5.4 katsotaan lyhyesti ratkaisuja autentikaatioon ja virheidenkäsittelyyn.

8.5.1 Käsitteiden mallinnus

JSF ja Angular käyttävät erilaisia käsitteitä toiminnallisuuksien kuvaamiseen. Useimmille käsitteille löytyy vastineet, mitä voidaan hyödyntää migraation toteutuksessa.

Taulukossa 8.2 luetellaan JSF:n JavaBean-luokkatyypit ja niitä vastaavat Angular-käsitteet. JavaBeanit on ryhmitelty niiden Scope-määrittelyjen mukaan. Käsitteiden liittämisessä on huomioitu sekä tilan näkyvyysalue että käyttötarkoitus. Kaikille käsitteille ei ole yksikäsitteistä vastinetta, minkä vuoksi paras vaihtoehto voi riippua tilanteesta.

Taulukossa 8.3 on lueteltu muita JSF- ja Angular-käsitteitä ja miten ne liittyvät toisiinsa. Joitain käsitteistä on jo hieman käsitelty edeltävissä luvuissa. Angular-moduulit ovat yksi käsite, jolle ei ole suoraa vastinetta JSF-puolella. Javan pakkaukset (package) ovat käsitteellisesti lähellä, mutta eivät käytön osalta käyttäydy vastaavasti. Moduulijako vaikuttaa siihen, miten isoissa osissa selain joutuu lataamaan Angular-sovelluksen lähdekoodia.

Suurimmalle osalla Java-luokista löytyy käsitteellisesti Angularin vastaava Java-/TypeScript-luokka. Kuten luvussa 5.2 käsiteltiin, ohjelmointikielien samankaltaisuuksien ja olio-ohjelmointituen vuoksi luokkien Java-toteutukset pystyy siirtämään vastaaviksi TypeScript-toteutuksiksi.

8.5.2 Tilanhallinta

Angular ja JSF käyttävät erilaisia mekanismeja sovelluksen tilanhallintaan. JSF:ssä sovelluksen tila säilötään JavaBean-luokissa, joiden elinaika määrätään erilaisilla Scope-dekoraattoreilla. Angularissa tila usein rajoittuu joko yksittäisen komponentin elinaikaan, tai globaaliin singleton service -instanssiin. Taulukossa 8.2 esitettiin, miten erilaiset Bean-luokat voidaan käsitteellisesti kytkeä Angular-luokkiin.

Toisin kuin palvelimella ajossa oleva JSF-sovellus, Angular menettää kaiken tilansa, jos käyttäjä poistuu sovelluksesta. Migraation aikana osa sivuista on toteutettu JSF:llä ja osa Angularilla, minkä vuoksi käyttäjä voi joutua usein poistumaan Angular-sovelluksesta. Tila hukataan myös, jos sivu avataan uuteen välilehteen. Angular tarvitsee tavallisten JavaScript-luokkien lisäksi pysyvämmän tilanhallintaratkaisun.

Selaimessa tähän sopiva kohde on session storage, johon pystyy tallentamaan muun muassa JSON-merkkijonoja. Session storage muistaa tiedot selaimen sulkemiseen asti [58]. Tämä on hyvä vastine esimerkiksi SessionScoped beanien tilanhallinnalle. Toinen vaihtoehto on local storage, joka säilyy myös selaimen sulkemisen jälkeen.

Jos JSF- ja Angular-sivujen täytyy käyttää yhteistä tilaa, tämän voi toteuttaa tekemällä SessionScoped JavaBeanin eteen JSON-rajapinnan. Angular-toteutuksen täytyy huolehtia tietojen ajantasaisuudesta ja tarvittaessa päivittää istunnon tietoja rajapinnan kautta. Kun JSF-sivuja on saatu riittävästi siirrettyä Angularille, toteutuksen voi vaihtaa session storageen.

Taulukko 8.2. *JSF:n JavaBean-luokkien vastaavat käsitteet Angularilla.*

JSF	Angular	Kuvaus
@Application-Scoped bean	JS-moduuli, JSON-rajapinta	ApplicationScoped beaneja käytetään perinteisesti joko yhteisenä välimuistina (esim. tietokannasta haetut harvoin muuttuvat arvot) tai kokoelmana apufunktioita, joita voi hyödyntää mallineissa. Apufunktiot voivat toimia tavallisena JS-moduulina. Jos ”välimuisti” ei ole suuruudeltaan iso ja on hyvin staattinen, se voi myös olla tavallisessa JS-moduulissa. Jos beanin sisältö on suurempi, siihen voidaan antaa pääsy JSON-rajapinnan kautta.
@SessionScoped bean	Angular service + session storage, JSON-rajapinta	Sisältää käyttäjäistunnon tilan. Jos tilan pitää olla yhteinen sekä JSF- että Angular-toteutuksille, tila täytyy pitää SessionScoped beanissa. Angular pääsee tilaan käsiksi tällöin JSON-rajapintojen kautta.
@ViewScoped bean (kertakäyttöinen)	Angular-komponentti	Sisältää näkymän tai komponentin tilan ja mallineen tarvitsemat ominaisuudet.
@ViewScoped bean (monikäyttöinen)	Angular service	JSF sallii saman beanin käytön useassa erillisessä näkymässä ja/tai komponentissa. Angularin servicet toimivat vastaavasti. Jos beanilla on oma tila jota ei kuulu jakaa näkymien välillä, servicen voi myös alustaa komponenttikohtaisesti. Tällöin sen ei tarvitse olla singleton.
@RequestScoped bean	Angular service, Angular-komponentti, JS-moduuli	Beanin geneerisyyden mukaan logiikka voi sijoittua service-luokkaan, Angular-komponenttiin tai JS-moduuliin. Angularin vastineet eivät automaattisesti resetoi tilaansa kuten RequestScoped, mikä pitää mahdollisesti huomioida.
@Conversation-Scoped bean	Angular service	Jos toisiinsa liittyvät näkymät voidaan toteuttaa saman isäntänäkymän alle, Angular service voidaan luoda isäntänäkymän tasolla. Tällöin alinäkymät voivat käyttää samaa service-instanssia. Instanssi generoidaan uudestaan, jos isäntänäkymästä poistutaan ja palataan takaisin. Jos beania käyttävät näkymät ovat navigaatiohierarkiassa erillisempiä, singleton serviceä voi myös hyödyntää. Tällöin sen tila pitää resetoida manuaalisesti uuden ”conversationin” alkaessa.

Taulukko 8.3. *Vastaavat käsitteet JSF:n ja Angularin välillä.*

JSF	Angular	Kuvaus
ServiceDelegate-luokka	JSON-rajapinta + client + service	Palvelurajapintoihin päästään käsinksi JSON-rajapintojen kautta, jotka käyttävät alkuperäisiä ServiceDelegate-toteutuksia. Client-luokat generoidaan automaattisesti, ja servicet toimivat niiden edessä abstraktiokerroksena. Käsitelty tarkemmin luvussa 6.2.
JSF-validaattori	Angular-direktiivi	Yleisimmät validaattoridirektiivit tulevat Angularin mukana [54]. Loput voi toteuttaa itse.
FacesConverter	Angular-direktiivi, tavallinen funktio	Angular ei suoraan tarjoa arvoja muuntavia direktiivejä, mutta niitä voi toteuttaa itse [55]. Vaihtoehtoisesti arvoja voi muuntaa tavallisilla funktioilla kooditasolla.
Java-apuluokka	JS-luokka, JS-moduuli	Tilalliset apuluokat voi toteuttaa vastaavina JavaScript-luokkina. Jos apuluokka on tilaton, sen voi toteuttaa myös tavallisena moduulina.
-	Angular-moduuli	Ei suoraa vastinetta. Käytetään Angularissa kokonaisuuksien pilkkomiseen ja riippuvuuksien määrittämiseen.
Reititys: faces-config.xml Polut hakemistorakenteesta	RoutingModule [57]	Angularissa ei määrätä erikseen mihin näkymiin mistäkin näkymästä voi päästä, eivätkä polut riipu suoraan hakemistorakenteesta. Navigaatiosäännöt määritellään pelkästään url-polkujen perusteella.

8.5.3 Autentikaatio

WebLogic-sovelluspalvelin tarjoaa roolipohjaisen autentikaatiomekanismin, jolla käyttäjältä voi vaatia tiettyä roolia sivulle pääsemiseksi. Rooleja pystyy tarkastelemaan myös kooditasolla.

Angularissa vastaavan reittipohjaisen autentikaation voi toteuttaa Angularin `CanActivate`-rajapinnalla, joilla voi varmistaa käyttäjän oikeudet mielivaltaisilla ehdoilla. Angular ei suoraan tiedä käyttäjän sessiotietoja, joten ne pitää hakea WebLogic-palvelimelta JSON-rajapinnan kautta. Haettuja roolitietoja voi vastaavasti käyttää kooditason autentikaatioon.

Iso ero Angularin ja WebLogicin autentikaation välillä on, että Angularin autentikaatioon ei voi yksinään luottaa tietoturvanäkökulmasta. Koska kaikki Angular-koodi ajetaan selaimessa, käyttäjät voivat koodia manipuloimalla kiertää käyttöliittymän rajoitukset. Esimerkiksi napin piillottaminen käyttöliittymästä Angularilla ei ole riittävä keino varmistamaan, että käyttäjä ei voisi sitä kiertää. Kaikki autentikaatiota vaativat toimenpiteet täytyy aina varmistaa myös palvelimen puolella.

8.5.4 Lokitus ja virheidenhallinta

Jos WebLogic-sovelluspalvelimen Java-koodissa tapahtuu poikkeus tai muu virhetilanne, se voidaan helposti kaapata ja kirjoittaa lokitiedostoon. Tämä on hyödyllinen työkalu virhetilanteiden selvittelyssä.

Angular on selaimessa ajettava sovellus, minkä vuoksi virheilmoitukset eivät automaattisesti näy palvelimella. Virhetilanteiden tutkimisen helpottamiseksi palvelimelle voidaan lisätä JSON-rajapinta, johon selain voi lähettää kaapattuja virheilmoituksia.

8.6 Toimintalogiikan migraation automatisointi

Prototyypin toteutuksen aikana tutkittiin vaihtoehtoja, joilla Java-koodin saisi hieman helpommin muunnettua vastaavaksi TypeScript-koodiksi. Ohjelmointikielet ovat samankaltaisia, minkä vuoksi se teoriassa voisi olla mahdollista.

Eräs lupaava ratkaisu oli JSWeet-kirjasto, jonka avulla on mahdollista tehdä Java-koodia, joka käännetään TypeScript-koodiksi. Kirjastoa ei ole tarkoitettu olemassa olevien Java-järjestelmien kääntämistä TypeScriptille, mutta se ehkä pystyisi toimimaan hyvänä pohjana.

Kirjastolla saatiin muunnettua koko järjestelmän Java-koodi TypeScript-koodiksi, mutta siinä on muutamia ongelmia. Kirjasto ei osaa muuttaa useita kirjastoja TypeScriptiksi, minkä vuoksi suurta osaa koodista ei voi hyödyntää. Toinen ongelma on, että sen generoima TypeScript ei ole helppolukuista. Kirjasto generoi koodia, joka simuloi Javan vastaavaa toiminnallisuutta, mutta se ei hyödynnä TypeScriptin omia ominaisuuksia ideaalisesti.

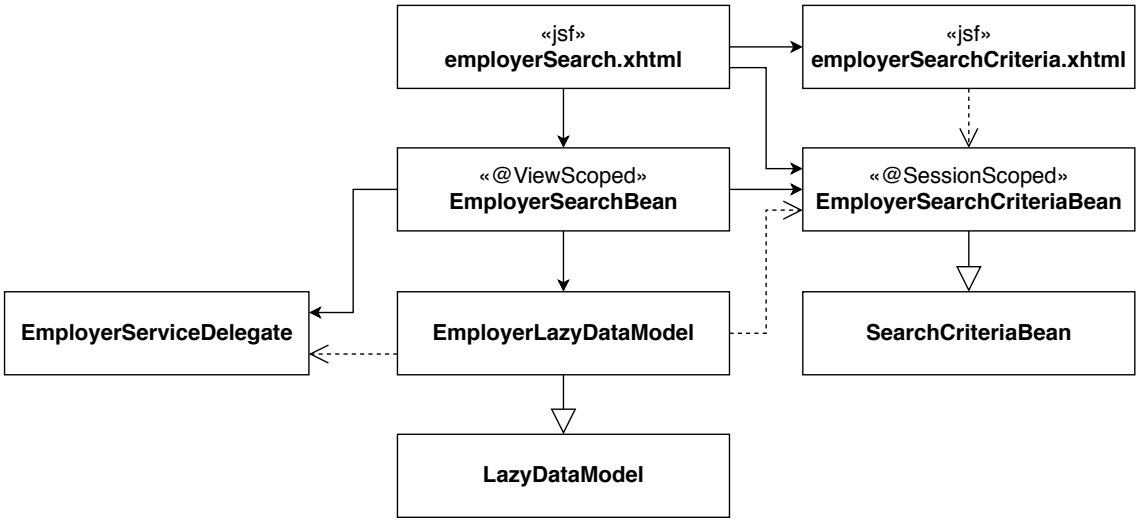
Vaikka JSWeet-kirjaston generoimaa koodia voi teoriassa hyödyntää jonkinlaisena pohjana, sen siistiminen käytännössä vaatisi enemmän työtä kuin optimaalisen koodin kirjoitus itse. Järjestelmän ylläpidettävyyden parantaminen on yksi päätavoitteista, minkä vuoksi JSWeetin generoima koodi ei ole riittävän hyvä tähän käyttötarkoitukseen.

Luvussa 6.2 käsitelty rajapintakoodin ja -tyypityksien automaattinen generointi toimii hyvin, mutta se ei toimi muun toimintalogiikan generointiin. Projektin aikana ei löydetty parempia vaihtoehtoja toimintalogiikan generoinnin automatisoinniksi.

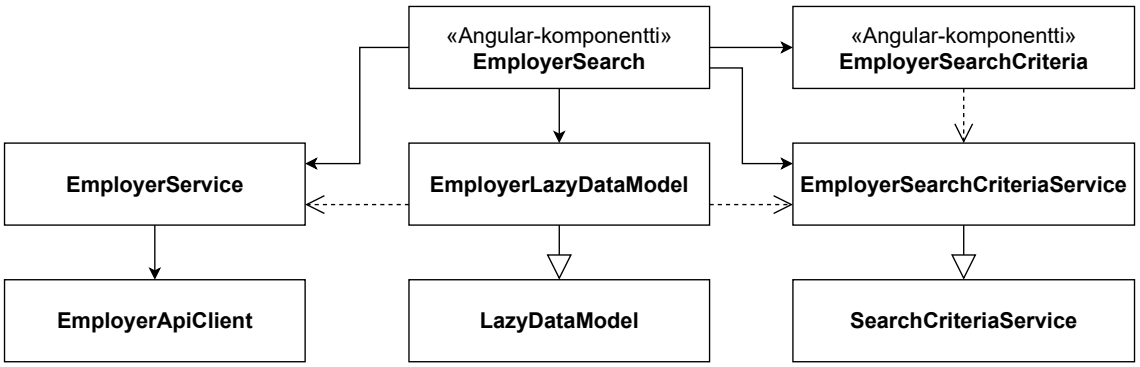
8.7 Esimerkkitapaus: Hakunäkymän migraatio

Prototyypissä siirrettiin onnistuneesti tarkasteltavasta järjestelmästä yksi näkymä Angulariksi. Prototyyppiin toteutettiin valmis sovellusrunko ja tarvittavat toimenpiteet sen integroimiseen JSF-sovelluksen kanssa.

Näkymämigraatiota varten toteutettu skripti toimii hyvin Angularin mallinetiedostojen generoinnissa. Skriptiä saatiin helposti täydennettyä sitä mukaan, kun näkymissä tuli uusia tapauksia vastaan.



Kuva 8.1. *Kaavio JSF-toteutuksen hakunäkymän toteutusrakenteesta.*



Kuva 8.2. *Kaavio Angular-toteutuksen hakunäkymän toteutusrakenteesta.*

Isoin esimerkki, jossa näkymämigraatioskriptiä testattiin, oli hakunäkymän hakulomake. Lomake sisältää muun muassa erilaisia tekstikenttiä, monivalintakenttiä, ehtolausekkeita, automaattisia täydennyskenttiä, validointia ja omia alikomponentteja. Lomakekomponentille generoidussa tiedostossa on 88 HTML/Angular-elementtiä, joilla on yhteensä 203 attribuuttia. HTML-elementeistä 8 piti korvata toisella tagilla tai poistaa manuaalisesti. Jos muutoksia muuttuja- ja funktionimiin ei lasketa, attribuuteista vain 26 vaati manuaalisia muutoksia. Toisin sanoen HTML-elementtinimistä noin 91 % oli oikein, ja attribuuteista noin 87 % ei vaatinut muita muutoksia kuin muuttujanimien korjaamisen.

Tulokset eivät ole yhtä hyvät kaikilla näkymillä, mutta tulokset ovat vastaavan kaltaiset myös muilla testatuilla komponenteilla. Yksinkertaisimmat komponentit eivät vaatineet mitään manuaalisia muutoksia. Manuaalista työtä tuli eniten esiin PrimeFaces-kirjaston komponenttien kanssa, joiden rajapinnat ovat Angularille hieman erilaiset. Nämä vaativat muutoksia myös TypeScript-koodissa. Vaikka toteutettu apuskripti ei poista manuaalista työtä kokonaan, se on paljon nopeampaa kuin vastaavan koodin kirjoittaminen tyhjästä.

Kuvassa 8.1 on alkuperäisen hakunäkymän toteutusrakenne ja kuvassa 8.2 on sitä vastaava Angular-toteutus. JSF-kaaviossa ylimmät komponentit ovat JSF:n XHTML-tiedostoja ja loput Java-luokkia. Angularin kaaviossa Angular-komponentteihin on sisällytetty

sekä komponentin HTML-tiedosto että TypeScript-luokka. Muut ovat tavallisia TypeScript-luokkia. Molemmissa nuolilla esitetään käsitteiden välisiä riippuvuuksia. Katkoviivoilla esitetään riippuvuudet, jotka saadaan ylemmän tason käsitteeltä. Esimerkiksi Angular-kaaviossa `EmployerLazyDataModel` saa `EmployerService`-instanssin `EmployerSearch`-komponentilta. Molemmista kaavioista on poistettu muutamia abstraktiokerroksia ja riippuvuuksia, jotka eivät tämän tarkastelun osalta ole olennaisia.

Angular-toteutuksessa `EmployerSearchBean` on siirtynyt osaksi `EmployerSearch`-komponenttia, sillä sen logiikkaa ei käytetä järjestelmässä muualla. `EmployerSearchCriteriaService` puolestaan on pidetty omana luokkanaan, sillä siihen on riippuvuuksia muualtakin kuin `EmployerSearchCriteria`-komponentista. Kuten kaaviosta näkee, luokkatasolla Angular-toteutuksen rakenne on hyvin symmetrinen JSF-toteutuksen kanssa.

8.8 Yhteenveto

Toteutetussa Angular-prototyypissä saatiin selvitettyä useita teknisiä haasteita ja niihin käytännöllisiä ratkaisuja. Huomiota kiinnitettiin Angular-sovelluksen alustukseen, sen integraatioon vanhan JSF-järjestelmän kanssa, miten näkymien mallinetiedostot voi siirtää ja miten vanha toimintalogiikka saadaan siirrettyä Angularille.

Näkymien migraatiossa tutkittiin JSF:n ja Angularin mallinekielien samankaltaisuuksia ja eroja. Mallinemigraatiota varten saatiin toteutettua apuskripti, joka kokeilujen perusteella pystyy siirtämään suurimman osan JSF:n mallinekoodista vastaavaksi Angular-koodiksi.

Toimintalogiikan migraatiossa tutkittiin miten teknologioiden eri käsitteet täsmäävät keskenään. Näiden lisäksi katsottiin lyhyesti tilanhallinnan, autentikaation ja virheidenhallinnan eroavia toimintamalleja teknologioiden välillä. JSON-rajapinnoille saatiin generoitua automaattisesti client-toteutukset ja tyypitykset, mutta muulle toimintalogiikalle ei saatu automaattisesti generoitua vastaavaa käyttökelpoista TypeScript-koodia.

9 REACT-PROTOTYYPPI

React-prototyypissä tutkitaan lähestymistä, jossa React-toteutus laitetaan JSF-sivun sisälle. Tätä lähestymistä käsiteltiin yleisemmällä tasolla luvussa 6.1.2.

Koska React on JSF-sovelluksen sisällä, sen vastuualue on hieman pienempi. JSF on edelleen vastuussa sovellusrungosta, johon sisältyy sivujen yhteiset elementit, alustava auktorisointi ja navigaatio.

Luvuissa 9.1 ja 9.2 käydään läpi, miten prototyypin React-sovellus alustetaan ja miten se liitetään vanhaan järjestelmään. Luvussa 9.3 tutkitaan, miten JSF-näkymätiedostot saadaan muunnettua vastaavaksi React-koodiksi. Luvussa 9.4 jatketaan näkymämigraation tutkimista automatisaation näkökulmasta. Luvussa 9.5 tutkitaan, miten Java-logiikka voidaan siirtää Reactille. Luvussa 9.6 katsotaan prototyypissä tehtyä käytännön esimerkkiä ja lopuksi luvussa 9.7 otetaan yhteenveto havainnoista.

9.1 Alkutoimenpiteet

Prototyypissä otettiin lähtökohdaksi Create React App -kirjaston generoima pohja. Kirjasto antaa useita valmiita työkaluja, ja on Reactin virallinen suositus [59]. Tämä kuitenkin osoittautui toimimattomaksi ratkaisuksi.

Koska tutkittavassa lähestymisessä React-koodi on JSF-sivun sisällä, jokaisen sivun tarvitsee olla oma React-sovellus. Toisin sanoen jokainen sivu tarvitsee oman JavaScript-tiedoston, joka määrittää kyseisen sivun React-sisällön. Create React App -kirjaston generoima pohja olettaa järjestelmän yhden sivun sovellukseksi, minkä seurauksena sillä on vain yksi ulostulo. Kirjasto antaa rajoitetusti konfigurointivaihtoehtoja, minkä takia se ei suoraan riitä migraation tarkoituksiin.

Create React App antaa mahdollisuuden kopioida kaikki sen Webpack-määritykset ja riippuvuudet omaan projektiin, jolloin niitä voi muokata mielivaltaisesti. Tämä mahdollistaa omien määrityksien tekemisen, mutta myös siirtää ylläpidollisen vastuun kehittäjälle. Siirtoa ei voi peruuttaa, eikä kirjaston päivityksiä voi automaattisesti enää saada. [60, s. 7–8]

Create React App -kirjaston Webpack-määrittelyssä on migraationäkökulmasta kolme ongelmaa: tiedostojen generointi kehitystilassa, generoitujen tiedostojen määrä ja generoitujen tiedostojen nimet.

Koska JSF toimii sovelluksen runkona, sen pitää päästä käsiksi React-koodista generoituihin tiedostoihin. Oletusarvoinen kehityspalvelin katsoo tiedostomuutoksia ja päivittää muutokset selaimeen, mutta ei generoi fyysisiä tiedostoja levylle. Oletusarvoinen kehityspalvelin pitää vaihtaa Webpackin ”watch”-tilaan, joka generoi tiedostot automaattisesti muutoksien yhteydessä.

Oletusarvoisessa Webpack-määrittelyssä sovelluksella on vain yksi alkutiedosto (entry point), minkä seurauksena se generoi vain yhden ulostulon. Koska jokaisella näkymällä on oma React-sovelluksensa, jokaisella pitää olla oma alkutiedosto. Hakemistorakenteessa jokaiselle näkymälle on oma TypeScript-tiedosto saman kansion alla. Webpack-määrittelyssä voi katsoa kaikki halutun kansion tiedostot ja käyttää niitä rakentamisen alkupisteinä.

Webpack-määrittely oletusarvoisesti generoi tiedostoille nimet, joita ei voi suoraan liittää alkuperäisiin tiedostonimiin. Esimerkiksi ”myPage.ts” voi saada nimen ”1.0g2hml.js”, jossa ensimmäinen on kasvava numero ja toinen sen tiivistetunniste. Jotta JSF voi viitata lähdekooditiedostoon, nimen pitää joko olla staattinen tai muuten pääteltävissä. Asetuksia säätämällä nimen voi laittaa esimerkiksi muotoon ”myPage.js” tai ”myPage.0g2hml.js”. Ensimmäinen on helpompi ratkaisu, mutta ei mahdollista yhtä tehokasta välimuistin käyttöä. [61, luku 7] Toinen ratkaisu edellyttää, että JSF-toteutuksessa selvitetään tiedoston koko nimi esimerkiksi vertaamalla staattista osaa kaikkiin kansion tiedostoihin.

Koska Webpack-määrittely vaatii paljon rakenteellisia muutoksia ja kaikkia Create React App -ominaisuuksia ei tarvita, prototyypissä toteutettiin oma Webpack-määrittely tyhjältä pohjalta, johon kopioitiin osia Create React App -kirjaston määrityksistä. Lopullisen migraation toteutuksessa voi olla tarpeen pohtia, pitäisikö toteutuksen olla enemmän Create React App -kirjaston mukainen vai ei.

9.2 Reititys sovelluksien välillä

Koska kaikki sovelluksen sivut ovat edelleen JSF-sivuja, JSF-toteutuksen näkökulmasta reititys toimii täysin samalla tavalla kuin aikaisemminkin. Kaikkien sivujen polut pysyvät samoina, minkä vuoksi asiat toimivat kuten ennenkin. JSF-toteutuksen ei tarvitse tietää, onko viitattu sivu JSF- vai React-pohjainen.

Reactia käyttävien JSF-sivujen tarvitsee kuitenkin tietää niitä vastaavien React-kooditiedostojen polut. React-lähdekooditiedostot voidaan isännöidä staattisina tiedostoina, joihin voidaan viitata absoluuttisilla poluilla. Esimerkiksi Nginx-palvelinta voi käyttää isännöinnissä. Jos tiedostonimissä on mukana tiivistetunniste, tiedoston koko nimen päättelyyn vaaditaan hieman lisälogiikkaa.

Ohjelmassa 9.1 on yksinkertainen esimerkki siitä, miten JSF-sivu voi viitata React-toteukseen. Esimerkin `ReactBean` sisältää logiikan, joka osaa etsiä hakemistorakenteesta tiedoston koko polun. Esimerkiksi JavaScript-resurssin polku voisi olla `/react-resource/-examplePage.205199ab45963f6a62ec.js`.

```
1  <!-- Tyylisäännöt -->
2  <link href="${reactBean.getCssPath('vendor')}"
3        type="text/css"
4        rel="stylesheet"/>
5  <link href="${reactBean.getCssPath('examplePage')}"
6        type="text/css"
7        rel="stylesheet"/>
8
9  <!-- React-sovellus tulee root-elementin sisälle -->
10 <div id="root"></div>
11 <!--
12     Vendor sisältää sivujen yhteiset riippuvuudet ja
13     examplePage tietyn sivun logiikan.
14 -->
15 <script src="${reactBean.getJavaScriptPath('vendor')}"
16         type="application/javascript"></script>
17 <script src="${reactBean.getJavaScriptPath('examplePage')}"
18         type="application/javascript"></script>
```

Ohjelma 9.1. *Esimerkki React-koodiin viittaamisesta JSF:n XHTML-tiedostossa.*

React-koodissa kytkennät JSF-sivuille voidaan tehdä tavallisilla linkeillä. Reactinkaan ei tarvitse tietää, onko linkin näkymä kokonaan JSF-sivu vai JSF:n ja Reactin hybridi.

9.3 Näkymien migraatio

Reactin JSX on syntaksiltaan samankaltainen kuin HTML, mutta kulissien takana JSX-elementit ovat tavallisia JavaScript-olioita. Suuri osa luvussa 8.3 käsitellyistä Angularin ja JSF:n välisistä havainnoista pätee myös Reactin kanssa, mutta niissä on myös joitain eroavaisuuksia.

9.3.1 Tagimäärittelyt

React tukee Angularin tavoin sekä tavallisia HTML-elementtejä että omia komponenttimäärityksiään. Yksinkertaisissa tapauksissa JSF-elementit muuntuvat vastaaviksi HTML-elementeiksi täysin vastaavasti kuin Angularilla, mitä on tarkemmin käsitelty luvussa 8.3.1. Vastaavasti JSF-komponenteille, joille ei ole suoraa HTML-vastinetta, voidaan tehdä omat React-komponentit.

Perinteisissä HTML-elementeissä eräs ero on, että React ei salli JavaScriptin varattuja sanoja elementtien attribuuteissa. Esimerkiksi `class`-attribuutti on Reactissa `className`. Toinen ero attribuuttinimissä on, että React käyttää camelCase-nimeämistä attribuuttinimissä. Esimerkiksi HTML:n `onclick`-attribuutti on Reactissa `onClick`. [24]

Tarkasteltavassa järjestelmässä käytetylle PrimeFaces-kirjastolle on olemassa React-versio, kuten Angularilla [62].

```
1  function ExampleForm() {
2    const [value, setValue] = useState('');
3    const [submit, result, isError] = useExample();
4    return (
5      <form id="exampleForm">
6        <input value={value} onChange={(e) => setValue(e.target.value)} />
7        <button onClick={() => submit(value)}>Submit</button>
8      </form>
9      {result && (
10       <span id="result"
11             className={isError() ? 'error' : 'result'}>
12         {result}
13       </span>
14     )}
15   );
16 }
```

Ohjelma 9.2. *Esimerkki yksinkertaisesta React-lomakkeesta.*

9.3.2 Dynaamisuus ja interaktiivisuus

Angularista ja JSF:stä poiketen React ei tue kaksisuuntaista arvojen sitomista. Jos esimerkiksi elementille annetaan arvona olion attribuutti, se ei voi muuttaa sitä suoraan. Sen sijaan Reactin pitää erikseen välittää alikomponentille funktio, joka voi tehdä muutoksen ylemmän tason komponentissa.

Reactissa dynaamiset lausekkeet toteutetaan `{lauseke}`-syntaksilla. Syntaksia voi käyttää muun muassa React-elementtien ominaisuuksien määrittämiseen tai arvojen interpolointiin JSX-syntaksin keskellä. Angularista ja JSF:stä poiketen React ei lisää tavallisiin HTML-elementteihin lisäattribuutteja, joiden avulla esimerkiksi voisi määrätä elementin näkymistä tai iteroida elementtejä. Sen sijaan ehtolausekkeet tehdään tavallisella JavaScriptillä, sillä JSX on kulissien takana vain JavaScriptiä. Ohjelmassa 9.2 on esimerkki yksinkertaisesta React-lomakkeesta, joka vastaa ohjelman 8.2 JSF-esimerkkiä.

Koska React ei erota JSX-koodia muusta komponenttitoteutuksesta, se ei tarvitse erillistä näkymämallia kuten Angular. JSX voi käyttää kaikkia muuttujia ja funktioita, joita komponenttifunktion näkyvyysalueella on saatavilla.

9.3.3 Tyylimäärittelyt

React tarjoaa useita tapoja määrittää CSS-tyylisääntöjä. Tyylisäännöt voivat olla muun muassa globaaleja tiedostoja, `import`-syntaksilla tuotuja CSS-tiedostoja, `import`-syntaksilla tuotuja CSS-moduuleja tai JavaScriptillä määriteltyjä tyylisääntöjä ”CSS-in-JS”-lähestymisellä. [63, 64, 65]

`import`-syntaksiin pohjautuvat lähestymiset voivat käyttää esimerkiksi Webpackia koostamaan kaikki viitatut CSS-tiedostot yhdeksi tiedostoksi. CSS-moduulit tässä tapauksessa voivat toteuttaa vastaavan käyttäytymisen kuin Angular, jossa sääntöjen näkyvyysaluet-

ta voidaan rajata. Erona on, että Reactissa täytyy erikseen tuoda CSS-tiedoston luokkamäärittelyt ja asettaa ne halutulle komponentille. CSS import -syntaksi mahdollistaa CSS-logiikan pilkkomisen pienemmiksi osiksi, mikä voi auttaa komponenttiin liittyvien tyylimäärityksien löytämisessä. [63, 64]

JSF-tyylimäärityksiä voidaan siirtää komponenttipohjaisiin CSS-tiedostoihin vastaavasti kuin Angularilla, mitä on tarkemmin käsitelty luvussa 8.3.3. Eräs ero näiden välillä on, että Reactin omat elementit eivät näy sivun komponenttipuussa ollenkaan. Reactin generoima HTML-rakenne voi siis olla hieman lähempänä alkuperäistä kuin Angularilla.

”CSS-in-JS”-lähestymistä ei ole tämän järjestelmän osalta pidetty parhaana vaihtoehtona, sillä sen toteutuksissa on enemmän vaihtelua kuin yksinkertaisissa CSS- tai SCSS-tiedostoissa. Pelkästään ”CSS-in-JS”:n oliopohjaisille toteutuksille on yli 42 erilaista kirjastovaihtoehtoa [65]. Ylläpidon näkökulmasta tavalliset tyylitiedostot ovat paremmin standardoituja ja siten tämän järjestelmän tarpeisiin soveltuvampia.

9.3.4 Komponenttimäärittelyt

React tarjoaa kaksi eri tapaa komponenttien määrittämiseen: luokkapohjaiset komponentit ja funktiopohjaiset komponentit. Komponenttien käytössä ei tarvitse tietää kummalla tavalla komponentti on tehty, joten molempien tapojen käyttö on mahdollista. Molemmilla tavoilla pystyy React hookien myötä tekemään vastaavaa toiminnallisuutta, mutta vain funktionaalisilla komponenteilla pystyy käyttämään hook-funktioita. [24] Tässä työssä keskitytään funktionaalisiin komponentteihin, sillä ne ovat hookien ansiota hieman joustavampia.

Toisin kuin Angular, React ei erota komponentin koodia HTML-koodiksi ja toimintalogiikaksi. React-komponentti on yhdessä tiedostossa, ja siinä voi sekoittaa HTML-kaltaista JSX-syntaksia ja JavaScript-koodia mielivaltaisesti. React-komponentit ovat yksinkertaisia JavaScript-funktioita, jotka palauttavat JSX-syntaksilla muodostetun arvon.

Ohjelmassa B.4 on vastaava React-esimerkki ohjelman B.1 JSF-komponentille. `SimpleFormProps`-tyyppimäärittelyllä voidaan määrittää komponentin saamat argumentit. Tämä vastaa JSF-komponentin `interface`-määrittelyä. Tässä tapauksessa komponentin rakenne on suurelta osin vastaava kuin luvun 8.3.4 Angular-esimerkillä.

Angularista ja JSF:stä poiketen React-komponentit eivät voi saada attribuuttien arvoina lausekkeita. Attribuuttien arvojen pitää olla konkreettisia JavaScript-arvoja, kuten funktioita. Lausekkeet voi kuitenkin helposti muuntaa funktioiksi, kuten esimerkkiohjelman `input`-komponentissa on tehty.

React-komponentit eivät tue kaksisuuntaista arvojen sitomista, minkä takia React-komponentille pitää näissä tilanteissa antaa erikseen saatu arvo ja funktio, joka muuttaa saadun arvon. Näiden määritykset pitää tehdä isäntäkomponentissa.

Yksityiskohtien eroavaisuuksista huolimatta React-komponentit pystyvät toteuttamaan vastaavat ominaisuudet kuin JSF-komponentit, joten migraatiossa JSF-komponentit voidaan

```
1  import React from 'react';
2  import { useState } from '@hookstate/core';
3  import { useTextField } from '../hooks/useTextField';
4  import { nameConverter } from '../converters/nameConverter';
5  import { nameValidator } from '../validators/nameValidator';
6
7  function ValidationExample() {
8    const nameState = useState('');
9    const nameField = useTextField(nameState, {
10     converter: nameConverter,
11     // Validaattoreita voi olla useampia
12     validators: [nameValidator],
13   });
14
15   const errors = nameField.errors;
16
17   return (
18     <>
19       <input {...nameField.inputBinding} />
20       {errors.map((error) => (
21         <div className="error">{error.message}</div>
22       ))}
23     </>
24   );
25 }
```

Ohjelma 9.3. *Validointi- ja arvomuunnosesimerkki Reactilla.*

korvata React-komponenteilla.

9.3.5 Validointi ja arvomuunnokset

Reactilla ei ole sisäänrakennettua validointi- tai arvomuunnosmekanismia. React ei myöskään salli tavallisten HTML-elementtien ominaisuuksien laajentamista kuten Angularin direktiiveillä voi. Tämän vuoksi validaatio- ja muunnoslogiikka pitää toteuttaa erikseen JavaScript-koodilla.

Esimerkiksi React hookien avulla on mahdollista toteuttaa uudelleen käytettävää logiikkaa, jolla tämän voi tehdä helpommin. Prototyypin aikana tätä varten on toteutettu yleiskäytettävä `useTextField`-hook, jolle voi helposti antaa halutut validointi- ja muunnossäännöt. Ohjelmassa 9.3 on esimerkki `useTextField`-hookin käytöstä, joka toiminnallisuudeltaan vastaa ohjelman 8.4 JSF-esimerkkiä.

`useTextField`-hookin toteutus löytyy liitteestä D. `useTextField` pystyy validoinnin lisäksi yksinkertaistamaan arvojen kaksisuuntaisen sitomisen toteutusta, jossa kentän muuttaminen päivittää myös alkuperäistä arvoa. Vastaavat hook-funktiot on toteutettu myös muille kenttätyypeille, kuten `checkbox`-elementeille. Hookissa käytetään Reactin oman `useState`-funktion sijaan hookstate-kirjaston versiota [66], mutta vastaavan toteutuksen voisi tehdä myös Reactin versiolla. Luvussa 9.5 palataan tarkemmin siihen, miksi tässä käytetään eri kirjastoa.

```
1  /*
2    JSF:
3    <div>
4      <h:outputText styleClass="exampleText"
5                    rendered="#{exampleBean.visible}"
6                    value="#{exampleBean.name}"/>
7    </div>
8  */
9  function RenderedExample() {
10     return (
11       <div>
12         {exampleBean.visible && (
13           <span className="exampleText">{exampleBean.name}</span>
14         )}
15       </div>
16     );
17 }
```

Ohjelma 9.4. *Esimerkki rendered-ehdon generoinnista Reactille.*

Validaation ja arvomuunnoksien toteutus tehdään Reactissa eri tavalla, mutta JSF-toteutuksen voi siitä huolimatta siirtää suoraviivaisesti pohjatyön tekemisen jälkeen.

9.4 Näkymien migraation automatisointi

Reactin näkymäkoodimigraatiossa otetaan pohjaksi luvussa 8.4 käsitelty migraatioskripti, joka muuntaa JSF:n XHTML-koodin vastaavaksi Angularin HTML-koodiksi. Koska suuri osa JSF:n ja HTML:n välisistä muunnoksista menee vastaavasti Reactilla, suurta osaa toteutuksesta voidaan käyttää suoraan. Reactin JSX on riittävän lähellä HTML-syntaksia, että skriptin muunnoksia voidaan hyödyntää. Komponentti- ja attribuuttinimissä on joitain eroja verrattuna Angulariin, mutta näiden vaihtaminen skriptiin onnistuu suoraviivaisesti.

JSX-syntaksilla on muutamia eroja verrattuna XML-muotoon, jota käytetty xml-js-kirjasto antaa. JSX-syntaksissa dynaamiset attribuuttiarvot merkitään `{}`-sulkeilla, mutta ilman arvon ympärillä olevia lainausmerkkejä. Kirjasto ei tällaista ulostuloa mahdollista itsessään. Generoitua XML-ulostuloa voidaan kuitenkin käsitellä merkkijonona, jolloin kyseiset tapaukset voidaan RegEx-korvauksilla muuntaa oikeaan muotoon.

Kaikki JSF-attribuutit eivät muunnu React-attribuuteiksi yhtä suoraviivaisesti kuin Angularilla. Reactilla ei esimerkiksi ole suoraa vastinetta Angularin `ngIf`- ja `ngFor`-direktiiveille, minkä takia ne pitää toteuttaa JavaScript-koodina, jota laitetaan JSX:n sekaan. xmsl-js-kirjasto sallii tavallisen tekstin lisäämisen XML-elementtien sekaan, minkä avulla tämä voidaan mahdollistaa. Jos elementillä esimerkiksi on `rendered`-attribuutti, sille voidaan generoida kolmiosainen XML. Ohjelmassa 9.4 on tästä esimerkki. Esimerkissä rivit 12 ja 14 on generoitu tavallisina tekstielementteinä ja rivi 13 perinteisenä XML-elementtinä. Vastaavalla toimintalogiikalla voidaan toteuttaa silmukkarakenteita ja muuta logiikkaa, joka edellyttää JavaScriptin sisällyttämistä XML-rakenteen sisälle.

Reactissa näkymäkoodi ei ole yhtä erillään TypeScript-koodista kuin esimerkiksi Angu-

larissa. JSX-koodi on tavallisen TypeScript-funktion sisällä. Jos JSX käyttää omia tai kirjastosta saatuja React-komponentteja, ne pitää erikseen tuoda moduuleista `import`-lausekkeella. Automatisaation parantamiseksi skriptiä on täydennetty rakentamaan myös JSX-näkymäkoodin ympärillä oleva tiedostorakenne. Tähän sisältyy funktiomäärittely, käytettyjen komponenttien `import`-komennot, komponentin `props`-määrittely sekä syötekenttien hook-määritykset.

Ohjelmassa C.3 on ohjelman C.1 mukaisesta JSF-komponentista generoitu React-versio. Ohjelmakoodin ulkoasua on siistitty, mutta muuten migraatioskriptin tulokseen ei ole tehty muutoksia.

JSX-koodin generointi on tehty vastaavalla periaatteella kuin Angularilla, mitä on käsitelty tarkemmin luvussa 8.4. Erona aikaisempaan on, että tällä kertaa se myös kerää XML-rakenteesta metatietoja. Kun skripti käy XML-rakennetta rekursiivisesti läpi, se kirjaa niiden tietoja sitä mukaan talteen. Jos vastaan esimerkiksi tulee komponentti, joka tarvitsee `import`-komentoja, se kirjaa `import`-komennot talteen. Jos vastaan tulee `input`-elementti, se katsoo sen nimen, validointisäännöt ja tyypin, ja ottaa ne talteen hook-määrityksiä varten. Komponenttirajapinnan `props`-määritykset saadaan poimittua JSF:n `interface`-määrityksistä. Kun XML-rakenne on saatu kokonaan käsiteltyä ja muunnettua merkkijonoksi, kerätty metatieto muunnetaan TypeScript-koodiksi. Tämä yhdistetään generoidun JSX-koodin kanssa, jolloin saadaan koko React-komponentin määritys. Tiedoston loppuun tulee myös poimitut CSS-säännöt vastaavasti kuin luvussa 8.4.4 on tehty.

Generoitu ulostulo on melkein sellaisenaan validia React-koodia. Skripti ei osaa hoitaa oikein muun muassa JSF:n JavaBean-luokkien määrityksiä. Migraatioskriptiin liittyy vastaavat rajoitteet kuin Angularilla, eli se pystyy käsittelemään vain sellaisten komponenttien muunnokset, jotka sille on kerrottu.

9.5 Toimintalogiikan migraatio

Tässä luvussa käsitellään, miten JSF:n toimintalogiikka voidaan muuntaa vastaavaksi React-koodiksi. Toimintalogiikaksi tulkitaan erityisesti Java-ohjelmointikielellä toteutettu logiikka, eli JavaBeanit ja niiden käyttämät apuluokat. Mallinetiedostojen muuntamista käsiteltiin luvussa 9.3.

Luvussa 9.5.1 käydään läpi JSF:n ja Reactin käsitteitä ja katsotaan, miten ne liittyvät toisiinsa. Luvussa 9.5.2 katsotaan eroja JSF:n ja Reactin tilanhallinnassa ja miten vastaavan toiminnallisuuden voi tehdä Reactilla. Luvuissa 9.5.3 ja 9.5.4 katsotaan lyhyesti ratkaisuja autentikaatioon ja virheidenkäsittelyyn.

9.5.1 Käsitteiden mallinnus

JSF-sovelluksissa valtaosaan sovelluslogiikasta päästään käsiksi JavaBean-luokkien kautta. React-sovelluksien rakenteissa on enemmän vaihteluita, sillä React ei itsessään suosittele mitään tiettyä sovellusrakennetta. Prototyypissä on päädytty käyttämään hook-

Taulukko 9.1. *JSF:n JavaBean-luokkien vastaavat käsitteet Reactilla.*

JSF	React	Kuvaus
@Application-Scoped bean	JS-moduuli, JSON-rajapinta	ApplicationScoped beaneja käytetään perinteisesti joko yhteisenä välimuistina (esim. tietokannasta haetut harvoin muuttuvat arvot) tai kokoelmana apufunktioita, joita voi hyödyntää mallineissa. Apufunktiot voivat toimia tavallisena JS-moduulina. Jos ”välimuisti” ei ole suuruudeltaan iso ja on hyvin staattinen, se voi myös olla tavallisessa JS-moduulissa. Jos beanin sisältö on suurempi, siihen voidaan antaa pääsy JSON-rajapinnan kautta.
@SessionScoped bean	React hook, session storage, JSON-rajapinta	Sisältää käyttäjäistunnon tilan. Jos tilan pitää olla yhteinen sekä JSF- että React-toteutuksille, tila täytyy pitää SessionScoped beanissa. React pääsee tilaan käsiksi tällöin JSON-rajapintojen kautta. Jos tila on vain React-toteutuksessa, tilaa voidaan säilyttää session storagessa.
@ViewScoped bean (kertakäyttöinen)	React hook, React-komponentti	Sisältää näkymän tai komponentin tilan ja mallineen tarvitsemat ominaisuudet.
@ViewScoped bean (monikäyttöinen)	React hook	JSF sallii saman beanin käytön useassa erillisessä näkymässä ja/tai komponentissa. React hookeja voi käyttää vastaavasti.
@RequestScoped bean	React hook, React-komponentti, JS-moduuli	Beanin geneerisyyden mukaan logiikka voi sijoittua React hookiin, React-komponenttiin tai JS-moduuliin.
@Conversation-Scoped bean	React hook, session storage, JSON-rajapinta	Jos toisiinsa liittyvät näkymät voidaan toteuttaa saman isäntänäkymän alle, React voi säilyttää tilan itse. Muutoin tarvitaan joko JSON-rajapintaa tai session storagea.

pohjaista rakennetta, jossa komponenttien toimintalogiikka ja suuri osa tilasta säilötään Reactin hook-funktioissa.

Hook-funktiota on mahdollista käyttää vastaavasti kuin JavaBeaneja. Kuten JavaBeaneissa, sama komponentti voi käyttää useita hook-määrityksiä ja käyttää niiden tarjoamia tiloja ja funktioita. Erona on, että samasta hookista voi olla samalla sivulla useampi versio. Jos samaa hookia käytetään useassa eri komponentissa samalla sivulla, se pitää välittää isäntäkomponentilta alikomponenteille.

Taulukossa 8.2 luetellaan, miten erilaiset JavaBean-tyypit voidaan toteuttaa Reactilla. Jos samaa tilaa vaaditaan useilla eri sivuilla, se pitää tallentaa joko palvelimen JSON-

Taulukko 9.2. *Esimerkki Java-periytymisen muuntamisesta hook-rakenteeksi.*

```java
public class CounterBean {
  private Long value = 0;

  public Long getValue() {
    return value;
  }
  public void increment() {
    value += 1;
  }
}

public class DoubleCounterBean
    extends CounterBean {
  @Override
  public void increment() {
    super.increment();
    super.increment();
  }

  public String getName() {
    return "DoubleCounter";
  }
}
```

```javascript
function useCounterHook() {
  const [value, setValue] =
    useState(0);
  return {
    value,
    increment() {
      setValue(state => state+1);
    }
  }
}

function useDoubleCounterHook() {
  const counterHook =
    useCounterHook();
  return {
    ...counterHook,
    // Ylikirjoittaa
    counterHook-toteutuksen
    increment() {
      counterHook.increment();
      counterHook.increment();
    },
    name: 'DoubleCounter'
  }
}
```

rajapinnan takana oleviin JavaBeaneihin tai selaimen session storageen. Useimmissa tilanteissa JavaBeanin voi korvata hookeilla, mutta joissain tapauksissa toteutus voi olla myös suoraan React-komponentin määrityksessä tai tavallisessa JS-moduulissa.

Vaikka React hookeilla on vastaavanlaisia ominaisuuksia kuin JavaBeaneilla, ne ovat rakenteellisesti erilaisia. JavaBeanit ovat luokkia, kun taas React hookit ovat funktioita. Funktiot eivät esimerkiksi suoraan tue periytymistä. Hookit kuitenkin tukevat koostamista (composition), jolla on mahdollista toteuttaa periytymistä vastaavaa käyttäytymistä. Martin Fowler kirjassaan Refactoring esittää keinoja, joiden avulla periytymisen voi muuntaa koosterakenteeksi [67, s. 381–404]. Vastaavia menetelmiä voidaan hyödyntää JavaBean-luokkien periytymisrakenteen muuntamisessa vastaaviksi hook-funktioiksi. Taulukossa 9.2 on yksinkertainen esimerkki Javan periytymisrakenteen muuntamisesta hook-rakenteeksi.

Taulukossa 9.3 on muita JSF-käsitteitä ja niiden React-vastineita. Koska React on pelkkä kirjasto, se ei tarjoa yhtä paljon ominaisuuksia valmiina kuin JSF tai Angular. Vastaavien ominaisuuksien tekeminen itse ei kuitenkaan ole suuri työ, ja hookien ansiota samaa logiikkaa on helppo käyttää uudestaan. Esimerkiksi luvussa 9.3.5 esitetyllä `useTextField`-hookilla pystytään toteuttamaan vastaavat validaatio- ja arvomuunnosominaisuudet helposti. Reactille lisäksi löytyy monia kirjastoja, joita voi hyödyntää.

Taulukko 9.3. *Vastaavat käsitteet JSF:n ja Reactin välillä.*

JSF	React	Kuvaus
ServiceDelegate-luokka	JSON-rajapinta + client + hook	Palvelurajapintoihin päästään käsiksi JSON-rajapintojen kautta, jotka käyttävät alkuperäisiä ServiceDelegate-toteutuksia. Client-luokat generoidaan automaattisesti, ja hook-funktiot voivat toimia niiden edessä abstraktiokerroksena. Parempaa virheidenkäsittelyä ja lataushallintaa varten voidaan luoda geneeriset hook-funktiot.
JSF-validaattori	JS-funktio	React ei itsessään tarjoa validaatiologiikkaa, joten se pitää tehdä itse. Aihetta on käsitelty tarkemmin luvussa 9.3.5.
FacesConverter	JS-funktio	React ei itsessään tarjoa muuntajalogiikkaa, joten se pitää tehdä itse. Aihetta on käsitelty tarkemmin luvussa 9.3.5.
Java-apuluokka	React hook, JS-moduuli	Tilalliset apuluokat voi toteuttaa vastaavilla React hookeilla. Jos apuluokka on tilaton, sen voi toteuttaa myös tavallisena moduulina.
Reititys: faces-config.xml Polut hakemistorakenteesta	React router	Migraatio alkuvaiheissa reititys tapahtuu edelleen JSF:n kautta. Migraation lopussa voidaan siirtyä esimerkiksi React router -kirjastoon [68].

9.5.2 Tilanhallinta

React-järjestelmissä on useita mahdollisia käytäntöjä tilanhallinnalle. Reactilla on oma yksinkertainen tilanhallinta, mutta ulkoisten kirjastojen käyttö on yleistä. Näistä eräitä esimerkkejä ovat Redux, MobX ja Akita [69, 70, 71].

Reactin tilanhallintakirjastoille on tyypillistä, että ne säilyttävät globaalia tilaa, johon tarjotaan pääsy kaikkialta sovellusta. Tarkasteltavassa järjestelmässä tila on pääosin näkymäkohtainen, ja tiedon ajantasaisuus on erityisen tärkeää. Järjestelmän laajuuden ja datapainotteisuuden vuoksi koko järjestelmän pitäminen esimerkiksi Redux-tietovarastossa voi olla haastavaa.

Aikaisemmin Reactin oma tilanhallintajärjestelmä oli rajoittunut pelkästään komponenttien omaan tilaan, mikä hankaloitti tilallisen logiikan käyttämistä useassa eri paikassa. React hookien myötä tilaa voi käyttää komponenteista erillisissä funktioissa, mikä mahdollistaa logiikan uudelleen käytettävyyden. React hookit yhdessä Reactin Context-rajapinnan kanssa olivat vuoden 2020 State of Frontend -kyselyssä kaikista suosituin tilanhallintaratkaisu [72].

Prototyypissä tilanhallintaan käytetään tilallisia hook-funktioita. Lisäksi apuna on käytetty hookstate-kirjastoa [66], jonka avulla on helpompi luoda oliorakenteiden kentille rekursiivisesti Reactin setter-funktiot. Tämä yksinkertaistaa kaksisuuntaisten arvojen sitomisen toteutusta lomakkeille, mikä on yksi järjestelmän yleisimmistä käyttökohteista. Tä-

män käytöstä on esimerkki ohjelmassa 9.3.

Järjestelmän globaaliin tilaan voidaan käyttää esimerkiksi Reactin Contextia, jonka avulla alempana hierarkiassa päästään käsiksi ylemmän hierarkian tietoihin [24]. Sivujen välisiin tiloihin voidaan käyttää esimerkiksi selaimen session storagea tai JSON-rajapinnan takana olevaa SessionScoped beania kuten Angularilla. Tätä on käsitelty tarkemmin luvussa 8.5.2.

9.5.3 Autentikaatio

Migraatiovaiheessa React-sovellukset ovat JSF-sivujen sisällä, minkä takia näkymäpääsyn auktorisoinnissa voidaan käyttää vanhaa JSF-logiikkaa. Sivun sisäisiä oikeustarkasteluja varten pitää hakea käyttäjän tiedot JSON-rajapinnasta, kuten Angularilla.

Reactia koskevat samat rajoitteet kuin luvussa 8.5.3 Angularin kanssa, eli toteutuksessa ei voi luottaa pelkästään React-toteutuksen oikeustarkasteluihin. Oikeudet pitää aina tarkistaa myös palvelimen logiikassa.

9.5.4 Lokitus ja virheidenhallinta

Kuten Angularilla (luku 8.5.4), myös Reactille pitää luoda JSON-rajapinta, jonka kautta mahdolliset virheet voidaan välittää palvelimelle.

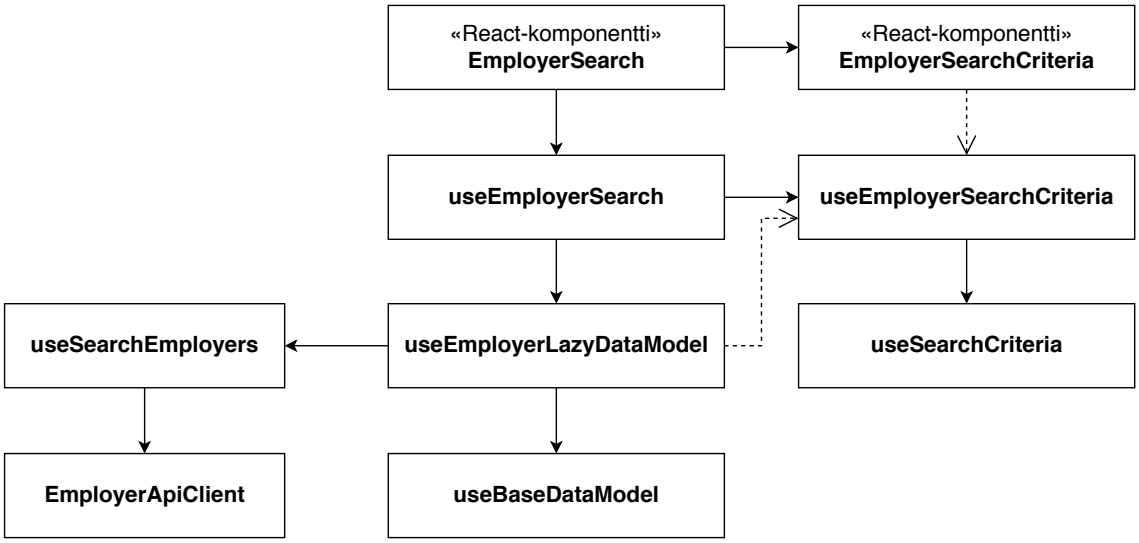
React sai versiossa 16 tuen odottamattomien virheiden kaappaamiselle [24], mitä voi hyödyntää virheiden käsittelyssä.

9.6 Esimerkkitapaus: Hakunäkymän migraatio

Prototyypissä saatiin onnistuneesti siirrettyä tarkasteltavasta järjestelmästä sama näkymä kuin Angularin prototyypissä. Kuten Angular-prototyypissä, prototyyppiin toteutettiin tarvittavat toimenpiteet JSF-integraatiolle sekä kaikki tarkasteltavan sivun perustoiminnot.

Luvussa 9.3 käsitelty migraatioskripti toimi hyvin JSF:n XHTML-koodin siirtämisessä vastaavaksi JSX-koodiksi. Toteutettu migraatioskripti vietiin tässä prototyypissä hieman pidemmälle kuin Angularilla, sillä se automatisoi myös pienen osan komponentin TypeScript-koodin generoinnista. Automatisaatio ei yllä varsinaisen toimintalogiikan siirtoon, mutta on migraatiossa suuri apu.

Isoin migraatioskriptin testikohde oli hakunäkymän hakulomake, johon sisältyy monenlaisia kenttiä ja erinäistä ehtologiikkaa. Sille generoidussa JSX-koodissa on 86 JSX-elementtiä ja niillä yhteensä 170 attribuuttia. Elementeistä 6 piti joko korvata toisella elementillä tai poistaa manuaalisesti. Attribuuteista noin 40 vaati manuaalisia muutoksia, jos yksinkertaisia muuttujanimien muutoksia ei lasketa. Toisin sanoen noin 93 % elementtinimistä oli oikein ja attribuuteista noin 76 %. Näiden lisäksi komponentin TypeScript-koodi



Kuva 9.1. *Kaavio hakunäkymän React-toteutuksen toteutusrakenteesta.*

vaati pieniä manuaalisia muutoksia.

Suuri osa attribuuttien muutoksista liittyi monimutkaisempiin `Autocomplete`-komponentteihin, joissa suuri osa attribuuteista korvattiin kokonaan hook-viittauksella. Koska JSX sisältää hieman enemmän toimintalogiikkaa kuin Angularin mallinetiedostot, saadut luvut eivät ole suoraan vertailtavissa luvun 8.7 tuloksiin.

Muilla näkymillä saadut tulokset ovat vastaavat. Kaiken kaikkiaan toteutettu apuskripti on hyödyllinen työkalu, vaikka ei poista kaikkea manuaalista työtä.

Kuvassa 9.1 on hakunäkymän React-toteutuksen rakenne. Toteutuksessa kaavion 8.1 JavaBean-luokat on muunnettu luvun 9.5.1 mukaisesti Reactin hook-funktioiksi. Tämän muutoksen myötä aikaisemmat periytymisrakenteet ovat muuttuneet tavallisiksi riippuvuuksiksi. JSF-kaavion `EmployerServiceDelegate` on tässä korvautunut `useSearchEmployers` hookilla. Tämän hookin riippuvuus on hierarkiassa siirtynyt alemmalle tasolle, sillä se on yksinkertainen funktio, jonka voi suoraan tuoda JavaScript-moduulista.

Päärakenteeltaan React-toteutus on hyvin saman näköinen kuin alkuperäinen JSF-toteutus. Toimintalogiikan siirto kuitenkin vaati hieman enemmän työtä kuin Angularilla, sillä Java-luokkien siirto hook-funtioiksi ei ole yhtä suoraviivainen kuin Java-luokkien siirto TypeScript-luokiksi.

9.7 Yhteenveto

Prototyypissä saatiin selvitettyä ratkaisuja useisiin migraation mahdollisiin haasteisiin. Huomiota kiinnitettiin erityisesti React-sovelluksen integraatioon JSF-sivujen kanssa, XHTML-koodin puoliautomaattiseen migraatioon ja miten JSF-arkkitehtuurin voi siirtää Reactille.

Näkymien migraatiossa tutkittiin miten XHTML-koodin saa siirrettyä React-koodiksi. Vaikka React ei suoraan tue kaikkia JSF:n ominaisuuksia, vastaavaa ominaisuutta voidaan si-

muloida yleiskäyttöisillä hook-funktioilla. Migraatiota auttamaan kehitettiin apuskripti, joka pystyy generoimaan React-komponentille puolivalmiin JSX-koodin ja osan sen ympärillä olevasta TypeScript-koodista. Vaikka skriptin ulostulo vaatii manuaalista tarkastelua, se prototyypin kokeilujen perusteella toimii erinomaisena työkaluna JSF-näkymäkoodin siirtämisessä Reactille.

Toimintalogiikan migraatiossa tutkittiin eri käsitteiden vastaavuuksia ja miten JSF-arkkitehtuurin saa muunnettua vastaavaksi React-arkkitehtuuriksi. Hook-funktioilla voi simuloida JavaBean-ominaisuuksia, mikä mahdollistaa samankaltaisen yleisarkkitehtuurin. React-koodi ei kuitenkaan tue vastaavia olio-ohjelmoinnin mukaisia rakenteita, minkä vuoksi logiikka ei ole sellaisenaan siirrettävissä.

10 PROTOTYPOINNIN TULOKSET JA ANALYYSI

Molemmilla työssä tutkituilla lähestymisillä saatiin toteutettua toimivat prototyypit. Molemmissa prototyypeissä löydettiin toimivat mekanismit vanhan logiikan siirtämiseksi ja uuden ja vanhan teknologian integrointiin. Prototyyppien välisillä lähestymisillä on omat vahvuutensa ja heikkoutensa.

Seuraavissa luvuissa tehdään tarkempaa vertailua Angular- ja React-toteutuksien välillä.

10.1 Integraatiomenetelmät

Angular-prototyypin integraatiomenetelmässä uusi teknologia on vahvemmin erillään vanhasta järjestelmästä, minkä vuoksi sen pitää toteuttaa enemmän ominaisuuksia perusrakenteessaan. Järjestelmän Angular-alijärjestelmä hoitaa oman navigaationsa, minkä vuoksi JSF-sivujen vanhat linkit eivät enää toimi ja pitää päivittää paikkoihin, joissa niihin viitataan.

React-prototyypin integraatiomenetelmässä React-koodi integroidaan migraatiovaiheen aikana osaksi JSF-sivua. Koska JSF on edelleen vastuussa sovellusrungosta ja navigaatiologiikasta, Reactin vastuualue on pienempi. Reactin ja JSF:n välinen integraatio on läpinäkyvämpää, sillä sivujen välisissä linkeissä ei tarvitse tietää onko kohdetoteutus tehty JSF:llä vai Reactilla.

Integraatiolähestymisistä Angular-prototyyppi todennäköisesti tarvitsee migraation alkuvaiheissa enemmän työtä kuin React-prototyyppi. React kuitenkin vaatii migraation lopussa erillisen loppumigraatiovaiheen, jossa loputkin JSF-rakenteesta siirretään Reactille. Koska loppumigraatiovaihe vaikuttaa koko sovelluksen rakenteeseen, siihen liittyy enemmän riskejä. Kaikki sivut on testattava uudestaan toiminnallisuuden varmistamiseksi.

Koska Angular-prototyypin lähestymisessä sovellus toteutetaan alusta lähtien yhden sivun sovelluksien periaattein, se saa kaikki yhden sivun sovelluksien hyödyt. Angular-alijärjestelmän ylläpito migraation aikana voi olla raskaampaa verrattuna Reactiin, sillä alijärjestelmien välillä on enemmän päällekkäistä logiikkaa. Jos migraatio halutaan tehdä vain osittaisena, esimerkiksi vain osalle sivuista, React-prototyypin lähestyminen on todennäköisesti soveltuvampi.

Vaikka aihetta ei työssä tarkemmin tutkittu, prototyypeissä testatut integrointimenetelmät eivät ole vahvasti teknologiasidonnaisia. Esimerkiksi Angular-prototyypin käyttämää me-

netelmää on mahdollista käyttää myös Reactin kanssa, sillä proxy-palvelimen ei tarvitse tietää mitä yhden sivun sovelluksen teknologiaa sovellus oikeasti käyttää.

10.2 Näkymien migraatio

Molemmissa prototyypeissä saatiin suoraviivaisesti muunnettua JSF-näkymälogiikan XHTML-tiedostot vastaavaksi sHTML/JSX-koodiksi. Angular täytenä sovelluskehyksenä antaa hieman enemmän vastaavia JSF-ominaisuuksia verrattuna Reactiin. Näistä eräitä esimerkkejä ovat validaattorit ja attribuuttipohjaiset renderöintiehdot. Reactilla kuitenkin on mahdollista toteuttaa vastaavat ominaisuudet muilla keinoin. Angularin HTML on syntaksiltaan lähempänä XHTML-syntaksia, mutta Reactin JSX on puolestaan joustavampi.

XHTML-koodin migraation helpottamista varten saatiin toteutettua molemmille prototyypeille apuskriptit, joiden avulla suuri osa sen manuaalisesta migraatiotyöstä saadaan automatisoitua. Prototyyppien toteutuksen aikana skriptit koettiin hyödyllisiksi ja auttoivat muun muassa huolimattomuusvirheiden välttämisessä. Molemmissa prototyypeissä tarkastelluissa esimerkeissä yli 75 % generoiduista HTML/JSX-tageista ei vaatinut mitään manuaalisia muutoksia mahdollisten muuttujanimikorjauksien lisäksi. React-prototyypissä migraatioskripti vietiin hieman Angularia pidemmälle, sillä se rakentaa JSX-koodin lisäksi sen ympärillä olevat komponenttimäärittelyt. Tämä on teoriassa mahdollista myös Angularille pienellä lisätyöllä, mutta tämän työn aikana sitä ei tehty. Kaiken kaikkiaan apuskriptit toimivat molemmilla lähestymisillä erinomaisesti ja toimivat hyvinä pohjina migraatioita tehtäessä.

10.3 Toimintalogiikan migraatio

Java-toimintalogiikan migraatiossa on hieman enemmän eroavaisuuksia prototyyppien välillä. Korkealla tasolla löydettiin sekä Angularille että Reactille JSF-käsitteitä vastaavat käsitteet, joilla on mahdollista toteuttaa järjestelmä vastaavalla arkkitehtuurilla. Alemmalla tasolla erot kuitenkin ovat isommat.

Angular tukee JSF:n tavoin olio-ohjelmointia, minkä ansiota se voi käyttää vastaavia rakenteita. React puolestaan edellyttää, että kaikki tilaan tehtävät muutokset menevät Reactin setter-funktioiden kautta. Tämän takia React ei tue yhtä hyvin esimerkiksi olio-ohjelmoinnin periytymis- ja luokkamekanismeja, joissa luokkainstanssit yleensä vastaavat omasta tilastaan. Vastaavia ominaisuuksia pystytään kuitenkin simuloimaan React hookien avulla, joilla tilallista logiikkaa voidaan uudelleen käyttää hieman kuten olio-ohjelmoinnilla. Kooditasolla ohjelmat pitää rakentaa hieman erilaisesti, minkä vuoksi Java-koodin siirtäminen Reactille ei ole yhtä suoraviivaista kuin Angularille.

10.4 Ylläpidettävyys

Eräs migraation tärkeistä kriteereistä on ylläpidon mahdollistaminen pitkälle tulevaisuuteen. Sekä Angular että React ovat suosittuja yhden sivun sovelluksen kirjastoja, ja molemmat saavat aktiivisesti päivityksiä. React on näistä kahdesta suositumpi [72], mutta molempien tulevaisuudennäkymät vaikuttavat hyvältä.

Angular-sovelluskehys on perustoiminnallisuudeltaan pysynyt vakaana vuoden 2016 laajan versio 2 -päivityksen jälkeen. Angularissa on tämän jälkeen kiinnittetty erityistä huomiota siihen, että päivitykset uusiin versioihin onnistuvat helposti. Angular tarjoaa kattavat ohjeet versiopäivityksiin [73] ja työkaluja mahdollisten koodimuutoksien automatisoinniksi. Angularilla on selvä visio tulevaisuudelle sovelluskehyksen parantamiseksi [47].

React pienempänä kirjastona ei saa aikaisempaa toiminnallisuutta rikkovia päivityksiä yhtä usein kuin Angular ja yrittää yleisesti minimoida muutokset julkisiin rajapintoihin [74]. Yksi kirjaston isoimmista muutoksista on vuonna 2019 julkaistut React hookit, joita prototyypissäkin käytetään. Vaikka hookit eivät riko aikaisempaa React-koodia, niiden tarjoamat uudet ominaisuudet muuttavat React-koodin kirjoittamista merkittävästi. React hookit ovat saaneet paljon suosiota, mutta niiden suhteellinen uutuus voi olla riski lyhyellä aikavälillä.

10.5 Jatkotutkimus

Diplomityö on rajoittunut yksinkertaiseen esiselvitykseen, jonka prototyypeissä on päivitetty vain yksittäinen näkymä. Tämän vuoksi on mahdollista, että tehty tutkimus ei kata eri näkymien kaikkia mahdollisia vaatimuksia. Esimerkiksi sivujen väliset riippuvuudet ja sivusiirtymät voivat tuoda uusia haasteita laajemmassa migraatiossa. Toteutuksien skaalautuvuus voi vaatia lisätutkimusta.

Teknologioista tutkittiin vain Angularia ja Reactia, mutta ne eivät ole ainoat mahdolliset vaihtoehdot. Esimerkiksi Vue on myös yksi hyvä mahdollisuus. Tutkimuksen aikana julkaistun Vue 3 -version myötä Vue sai paremman TypeScript-tuen [75], mikä tekee siitä paremman ehdokkaan kuin tutkimuksen alussa. Vuella on monia vastaavia ominaisuuksia Angularilla ja Reactilla, minkä vuoksi monia työn havaintoja voisi myös käyttää mahdollisessa Vue-migraatiossa.

React-prototyypissä päädyttiin käyttämään hook-pohjaista tilanhallintaa, mikä koettiin soveltuvaksi uudelleen käytettävän näkymäkohtaisen tilan hallintaan. Reactille kuitenkin löytyy useita muita tilanhallintakirjastoja, jotka myös voisivat olla mahdollisia vaihtoehtoja.

Työn tutkimus painottuu erityisesti tekniseen toteutukseen, mutta ei ota vahvasti kantaa migraation muihin osuuksiin, kuten testaukseen. Kattava testaus on tärkeää migraation onnistumiseksi.

11 YHTEENVETO

Työssä tutkittiin keinoja, joilla JSF-järjestelmän pystyy siirtämään yhden sivun sovelluksien teknologioille. Migraatiostrategiana on tutkittu erityisesti vaiheistettua migraatiota, jossa järjestelmä päivitetään pienissä osissa. Migraation aikana vanha ja uusi teknologia muodostavat hybridijärjestelmän, jossa osa sivuista on toteutettu vanhalla ja osa uudella teknologialla.

Migraatiovaihtoehtojen tutkinnassa toteutettiin kaksi prototyyppiä, joilla testattiin eri teknologioita ja erilaisia integrointilähestymisiä. Prototyypeissä kiinnitettiin huomiota erityisesti siihen, miten vanha ja uusi teknologia voidaan integroida keskenään ja miten vanhan järjestelmän ominaisuudet voidaan siirtää uudelle teknologialle. Molemmissa prototyypeissä tehtiin toimiva pohja migraation tekemiselle sekä kokeiltiin yksittäisen näkymän muuntamista vanhasta järjestelmästä.

Ensimmäinen prototyyppi toteutettiin Angularilla. Tässä kokeiltiin integraatiolähestymistä, jossa Angular-alijärjestelmä toteutetaan itsenäisenä yhden sivun sovelluksena, joka ei ole riippuvainen JSF-logiikasta. Alijärjestelmät kytketään toisiinsa proxy-palvelimella ja riittävän yhtenäisellä ulkoasulla. Navigaatio alijärjestelmien välillä tapahtuu tavallisilla linkeillä.

Toinen prototyyppi toteutettiin Reactilla. Tässä kokeiltiin integrointilähestymistä, jossa jokainen sivu saa oman React-sovelluksen, joka alustetaan JSF-sivun sisällä. Lähestymisessä JSF toimii migraation aikana sovelluksen säiliönä, mutta sivun pääsisältö toteutetaan Reactilla.

Molemmille prototyypeille toteutettiin apuskriptit, joiden avulla on mahdollista generoida JSF:n XHTML-koodista automaattisesti Angularin ja Reactin vastaavat HTML/JSX-määrittelyt. Testatuissa tapauksissa yli 75 % generoiduista elementeistä ei vaatinut manuaalisia muutoksia mahdollisten muuttujanimikorjauksien lisäksi. Migraatiotyökalun koettiin toimivan hyvänä pohjana XHTML-tiedostojen migraatioille ja vähentävän manuaalista työtä merkittävästi. Tämän lisäksi käytettiin typescript-generator-kirjastoa, jonka avulla on mahdollista generoida rajapintamäärittelyt Java EE:n JAX RS -rajapinnoille, mikä helpottaa kommunikaatiota palvelimen kanssa.

Molemmissa prototyypeissä saatiin onnistuneesti siirrettyä tarkastelussa olleen näkymän toiminnallisuus uudelle teknologialle ja integroitua vanhan järjestelmän kanssa. Lisäksi tutkittiin yleisemmällä tasolla, miten JSF-ominaisuuksia voidaan siirtää uusille teknologioille. Molempien prototyyppien teknologiat ja lähestymiset ovat migraatiolle mahdollisia

ratkaisuja, mutta niillä on erilaiset vahvuudet.

Angular-sovelluskehys on lähempänä JSF:n ominaisuuksia. Angular-prototyypissä tutkittu lähestyminen mahdollistaa yhden sivun sovelluksien hyötyjen hyödyntämisen hieman paremmin, mutta vaatii erityisesti migraation alkuvaiheissa enemmän työtä. Itsenäisen Angular-sovelluksen ylläpito voi olla hieman työläämpää, kunnes kaikki päällekkäiset JSF-ominaisuudet on saatu siirrettyä.

React-kirjasto puolestaan on soveltuvampi pienempään migraatioon, jossa halutaan päivittää vain osa sovelluksesta. Se pystyy kevyemmin olemaan JSF-järjestelmän rinnalla. Vaikka React ei tue suoraan kaikkia JSF:n ominaisuuksia ja sen rakenne on hieman erilainen, vastaavat ominaisuudet on mahdollista toteuttaa itse tai apukirjastojen avulla. Jos JSF halutaan lopulta poistaa kokonaan, esitetty React-migraation toteutustapa vaatii migraation lopussa migraatiovaiheen, jossa sovellusrunko siirretään Reactille.

Angular ja React ovat työssä tutkitut teknologiat, mutta suuri osa työssä tehdyistä havainnoista pätee myös muihin yhden sivun sovelluksen teknologioihin. Kaiken kaikkiaan JSF-järjestelmän siirtäminen kummalle tahansa yhden sivun sovelluksen teknologialle näyttäisi onnistuvan työssä kuvatuilla menetelmillä. Lisätutkimusta voidaan kuitenkin vaatia, sillä kaikkia mahdollisia ongelmakohtia ei voi löytää yksinkertaisella prototyypillä.

LÄHTEET

[1] *Jakarta Server Faces*. URL: https://en.wikipedia.org/wiki/Jakarta_Server_Faces (viitattu 16. 08. 2020).

[2] *PrimeFaces*. URL: https://www.primefaces.org/ (viitattu 21. 11. 2020).

[3] *OmniFaces*. URL: https://omnifaces.org/ (viitattu 21. 11. 2020).

[4] *JSF - Architecture*. URL: https://www.tutorialspoint.com/jsf/jsf_architecture.htm (viitattu 16. 08. 2020).

[5] Leonard, A. *Mastering JavaServer Faces 2.2 : master the art of implementing user interfaces with JSF 2.2*. eng.

[6] Ed Ort, B. M. *Java Architecture for XML Binding (JAXB)*. Maaliskuu 2003. URL: https://www.oracle.com/technical-resources/articles/javase/jaxb.html (viitattu 26. 07. 2020).

[7] *Angular - Testing*. URL: https://angular.io/guide/testing (viitattu 08. 10. 2020).

[8] React. *React - Testing Recipes*. URL: https://reactjs.org/docs/testing-recipes.html (viitattu 08. 10. 2020).

[9] Rauh, S. *AngularFaces: AngularJS Puts JSF on Steroids*. 19. heinäkuuta 2013. URL: https://www.beyondjava.net/angularfaces-jsf-beyond-ajax (viitattu 02. 08. 2020).

[10] *GitHub - AngularFaces repository*. URL: https://github.com/stephanrauh/AngularFaces (viitattu 02. 08. 2020).

[11] bessemHmidi. *AngularBeans*. URL: http://bessemhmidi.github.io/AngularBeans/ (viitattu 02. 08. 2020).

[12] bessemHmidi. *GitHub - AngularBeans repository*. URL: https://github.com/bessemHmidi/AngularBeans (viitattu 02. 08. 2020).

[13] Bragdon, N. *Devstack download: migrating from JSF to ReactJS*. 12. syyskuuta 2016. URL: https://www.bandwidth.com/blog/devstack-download-migrating-from-jsf-to-reactjs/ (viitattu 04. 08. 2020).

[14] Jendrock, E., Cervera-Navarro, R., Evans, I., Haase, K. ja Markito, W. *The Java EE 7 Tutorial: Volume 1, Fifth Edition*. eng. 4. painos. Addison-Wesley Professional, 2014. ISBN: 0321994922.

[15] Bisbal, J., Lawless, D., Wu, B. ja Grimson, J. Legacy information systems: issues and directions. eng. *IEEE software* 16.5 (1999), 103–111. ISSN: 0740-7459.

[16] Demeyer, S. *Object-oriented reengineering patterns*. eng. The Morgan Kaufmann Series in Software Engineering and Programming. San Francisco: Morgan Kaufman Publishers. ISBN: 1-281-07175-7.

[17] Wagner, C. *Model-Driven Software Migration: A Methodology*. 2014.

[18] *Stack Overflow Trends*. URL: https://insights.stackoverflow.com/trends?tags=angular%5C%2Creactjs%5C%2Cjsf%5C%2Cvue.js (viitattu 21. 11. 2020).

[19] Greig, S. ja Benitte, R. *State of JS 2019 - Front End Frameworks*. 2019. URL: https://2019.stateofjs.com/front-end-frameworks/ (viitattu 22. 11. 2020).

[20] Fink, G. *Pro Single Page Application Development Using Backbone.js and ASP.NET*. eng. Berkeley, CA.

[21] Scott, E. A. *SPA Design and Architecture: Understanding single-page web applications*. eng. 1. painos. Manning Publications, 2015. ISBN: 1617292435.

[22] Chitrapu, S. MVVM Model View ViewModel Part - 1 (31. elokuuta 2012). URL: https://social.technet.microsoft.com/wiki/contents/articles/13347.mvvm-model-view-viewmodel-part-1.aspx (viitattu 22. 11. 2020).

[23] Klauzinski, P. ja Moore, J. *Mastering JavaScript Single Page Application Development*. Birmingham: Packt Publishing, Limited, 2016. ISBN: 1785881647.

[24] *React website*. URL: https://reactjs.org/ (viitattu 24. 09. 2020).

[25] *Draft: JSX Specification*. URL: https://facebook.github.io/jsx/ (viitattu 22. 11. 2020).

[26] *React - Intoducing JSX*. URL: https://reactjs.org/docs/introducing-jsx.html (viitattu 22. 11. 2020).

[27] Abramov, D. React v16.8: The One With Hooks (2. kesäkuuta 2019). URL: https://reactjs.org/blog/2019/02/06/react-v16.8.0.html (viitattu 22. 11. 2020).

[28] *React - Hooks FAQ*. URL: https://reactjs.org/docs/hooks-faq.html (viitattu 22. 11. 2020).

[29] Abramov, D. React v16.x Roadmap (27. marraskuuta 2018). URL: https://reactjs.org/blog/2018/11/27/react-16-roadmap.html (viitattu 22. 11. 2020).

[30] *angular.js releases*. URL: https://github.com/angular/angular.js/releases?after=v0.9.4 (viitattu 22. 11. 2020).

[31] Fluin, S. Branding Guidelines for Angular and AngularJS (27. tammikuuta 2017). URL: http://blog.angularjs.org/2017/01/branding-guidelines-for-angular-and.html (viitattu 22. 11. 2020).

[32] *Angular versioning and releases*. URL: https://angular.io/guide/releases (viitattu 22. 11. 2020).

[33] *Angular Docs*. URL: https://angular.io/docs (viitattu 22. 11. 2020).

[34] *Angular - Template type checking*. URL: https://angular.io/guide/template-typecheck (viitattu 20. 08. 2020).

[35] *The TypeScript Handbook - TypeScript for Java/C# Programmers*. URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-oop.html (viitattu 20. 09. 2020).

[36] Mikkonen, J. Statically typed programming languages in the JavaScript ecosystem: A type system perspective; Staattisesti tyypitetyt ohjelmointikielet JavaScript-ekosysteemissä: tyyppijärjestelmien näkökulma. en. G2 Pro gradu, diplomityö. 20. tammikuuta 2020, 37–40. URL: http://urn.fi/URN:NBN:fi:aalto-202001261829.

[37] *Angular - Strict mode*. URL: https://angular.io/guide/strict-mode (viitattu 20. 08. 2020).

[38] Reese, R. M. *Oracle certified associate, Java SE 7 programmer study guide*. eng. Birmingham.

[39] Antani, V. *Mastering JavaScript : explore and master modern JavaScript techniques in order to build large-scale web applications*. Packt Publishing, 2016. ISBN: 1-78528-628-5.

[40] Simpson, K. *You Don't Know JS: Async & Performance*. eng. 1. painos. Sebastopol: O'Reilly Media, Incorporated, 2015. ISBN: 9781491904220.

[41] *GitHub - typescript-generator*. URL: https://github.com/vojtechhabarta/typescript-generator (viitattu 19. 11. 2020).

[42] *Angular - CLI Overview and Command Reference*. URL: https://angular.io/cli (viitattu 20. 08. 2020).

[43] *Angular - CLI - ng new*. URL: https://angular.io/cli/new (viitattu 20. 08. 2020).

[44] *Sass: Syntactically Awesome Stylesheets*. URL: https://sass-lang.com/ (viitattu 20. 08. 2020).

[45] *Sass: Syntax*. URL: https://sass-lang.com/documentation/syntax (viitattu 20. 08. 2020).

[46] Adi, D., John, W., Robert, F. ja Stephanie, Y. TSLint in 2019 (19. helmikuuta 2019). URL: https://medium.com/palantir/tslint-in-2019-1a144c2317a9 (viitattu 20. 08. 2020).

[47] *Angular - Roadmap*. URL: https://angular.io/guide/roadmap (viitattu 20. 08. 2020).

[48] *Angular - Deployment - The base tag*. URL: https://angular.io/guide/deployment#the-base-tag (viitattu 20. 08. 2020).

[49] *Angular - Build - Proxying to a backend server*. URL: https://angular.io/guide/build#proxying-to-a-backend-server (viitattu 20. 08. 2020).

[50] *JSF 2.2 View declaration tag summary*. URL: https://docs.oracle.com/javaee/7/javaserver-faces-2-2/vdldocs-facelets/h/tld-summary.html (viitattu 30. 08. 2020).

[51] *PrimeFaces - PrimeNG*. URL: https://www.primefaces.org/primeng/ (viitattu 30. 08. 2020).

[52] *Angular - Component styles*. URL: https://angular.io/guide/component-styles (viitattu 30. 08. 2020).

[53] *Angular - Coding style guide*. URL: https://angular.io/guide/styleguide (viitattu 30. 08. 2020).

[54] *Angular - Validating form*. URL: https://angular.io/guide/form-validation (viitattu 01. 10. 2020).

[55] Zemke, F. Angular Model Converter (3. elokuuta 2019). URL: https://zemke.io/angular-model-converter.

[56] *GitHub - xml-js*. URL: https://github.com/nashwaan/xml-js (viitattu 03. 09. 2020).

[57] *Angular - Routing*. URL: https://angular.io/guide/router (viitattu 17. 09. 2020).

[58] *MDN Web Docs - Window.sessionStorage*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage (viitattu 01. 12. 2020).

[59] *React - Create a New React App*. URL: https://reactjs.org/docs/create-a-new-react-app.html (viitattu 23. 11. 2020).

[60] Valjakka, A. A Reengineering Framework for the Migration of a Legacy Front End. en. diplomityö. 17. joulukuuta 2019.

[61] Zammetti, F. W. *Modern full-stack development : using TypeScript, React, Node.js, Webpack, and Docker*. eng.

[62] *PrimeFaces - PrimeReact*. URL: https://www.primefaces.org/primereact/ (viitattu 24. 09. 2020).

[63] *Adding a stylesheet*. URL: https://create-react-app.dev/docs/adding-a-stylesheet (viitattu 01. 10. 2020).

[64] *Adding a CSS Modules stylesheets*. URL: https://create-react-app.dev/docs/adding-a-css-modules-stylesheet (viitattu 01. 10. 2020).

[65] *GitHub - CSS in Js*. URL: https://github.com/MicheleBertoli/css-in-js (viitattu 01. 10. 2020).

[66] *Hookstate*. URL: https://hookstate.js.org/ (viitattu 23. 11. 2020).

[67] Fowler, M. *Refactoring. Improving the Design of Existing Code*. 2. painos. Addison-Wesley Signature Series (Fowler). Addison-Wesley, 2018. ISBN: 978-0-13-475759-9.

[68] *React router*. URL: https://reactrouter.com/ (viitattu 05. 11. 2020).

[69] *Redux website*. URL: https://redux.js.org/ (viitattu 13. 02. 2021).

[70] *MobX website*. URL: https://mobx.js.org (viitattu 13. 02. 2021).

[71] *GitHub - Akita*. URL: https://github.com/datorama/akita (viitattu 13. 02. 2021).

[72] Schiemann, D. *State of Frontend 2020 Report - Frameworks*. URL: https://tsh.io/state-of-frontend/#frameworks (viitattu 23. 11. 2020).

[73] *Angular Update Gude*. URL: https://update.angular.io/ (viitattu 29. 11. 2020).

[74] *React - Design Principles*. URL: https://reactjs.org/docs/design-principles.html (viitattu 29. 11. 2020).

[75] You, E. Vue.js v3.0.0 One Piece Released (18. syyskuuta 2020). URL: https://github.com/vuejs/vue-next/releases/tag/v3.0.0 (viitattu 01. 12. 2020).

A ESIMERKKI TYYPPIGENEROINNISTA

```java
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "EmployerType", namespace = "...", propOrder = {
  "id",
  "yCode",
  "homeMunicipality",
  // ... Muut ominaisuudet
})
public class EmployerType implements Serializable
{
  @XmlElement(
    required = true, type = Long.class, nillable = true
  )
  protected Long id;
  @XmlElement(
    required = true, nillable = true
  )
  protected String yCode;
  @XmlElement(
    required = true, nillable = true
  )
  protected MunicipalityType homeMunicipality;
  // ... Muut ominaisuudet

  public Long getId() {
    return id;
  }

  public void setId(Long value) {
    this.id = value;
  }
  // ... Muut getter/setter-metodit
}
```

Ohjelma A.1. *JAXB:n XML-määrittelystä automaattisesti generoitu Java-luokka (EmployerType.java).*

```
1  export interface EmployerType extends Serializable {
2    id: number;
3    yCode: string;
4    homeMunicipality: MunicipalityType;
5    // ...
6  }
7
8  // Myös tyyppiriippuvuudet generoidaan
9  export interface MunicipalityType extends Serializable {
10   id: string;
11   name: string;
12   nameSwedish: string;
13   // ...
14 }
15
16 export interface Serializable {
17 }
```

Ohjelma A.2. *EmployerType.java-tiedostosta generoidut TypeScript-määrittelyt.*

B ESIMERKKEJÄ KOMPONENTTIEN MÄÄRITTELYSTÄ

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <html xmlns="http://www.w3.org/1999/xhtml"
6        xmlns:h="http://java.sun.com/jsf/html"
7        xmlns:composite="http://java.sun.com/jsf/composite">
8
9  <composite:interface>
10   <composite:attribute name="value"
11                        type="java.lang.String"
12                        required="true" />
13   <composite:attribute name="label"
14                        type="java.lang.String"
15                        default="Input:" />
16   <composite:attribute name="acton"
17                        required="true" />
18 </composite:interface>
19
20 <composite:implementation>
21   <h:form>
22     <h:outputLabel for="input" value="#{cc.attrs.label}"/>
23     <h:inputText id="input" value="#{cc.attrs.value}"/>
24     <h:commandButton action="#{cc.attrs.action}"
25                      value="#{messageBean.actionMessage}"/>
26   </h:form>
27 </composite:implementation>
28 </html>
```

Ohjelma B.1. *Yksinkertainen JSF-komponentti (simpleForm.xhtml).*

```
1  import { Component, Input, Output, EventEmitter } from '@angular/core';
2  import { MessageService } from './message.service.ts';
3
4  @Component({
5    selector: 'gof-simple-form',
6    templateUrl: './simple-form.component.html',
7    styleUrls: ['./simple-form.component.scss']
8  })
9  export class SimpleFormComponent {
10
11   @Input()
12   id: string = '';
13   @Input()
14   value: string = '';
15   @Output()
16   valueChange = new EventEmitter<string>();
17   @Input()
18   label: string = 'Input:';
19
20   @Output()
21   action = new EventEmitter<void>();
22
23   constructor(public messageService: MessageService) { }
24 }
```

Ohjelma B.2. *Angular-komponentin TypeScript-tiedosto (simple-form.component.ts).*

```
1  <form>
2    <label [attr.for]="id + '-input'">{{label}}</label>
3    <input [id]="id + '-input'"
4           [(ngModel)]="value"
5           (ngModelChange)="valueChange.emit(value)">
6    <button (click)="action.emit()">
7      {{messageService.actionMessage}}
8    </button>
9  </form>
```

Ohjelma B.3. *Angular-komponentin HTML-tiedosto (simple-form.component.html).*

```tsx
import React from 'react';
import { actionMessage } from '../common/messages';

type SimpleFormProps = {
  id: string;
  value: string;
  onChange: (value: string) => unknown;
  label?: string;
  onAction: () => unknown;
};

export function SimpleForm(props: SimpleFormProps) {
  // Jos label-arvoa ei ole annettu, käytetään oletusarvoa
  const label = props.label ?? 'Input:';
  const inputId = props.id + '-input';
  return (
    <form>
      <label htmlFor={inputId}>{label}</label>
      <input id={inputId}
             value={props.value}
             onChange={
               (event) => props.onChange(event.target.value)
             }
             />
      <button onClick={props.onAction}>{actionMessage}</button>
    </form>
  );
}
```

Ohjelma B.4. *React-komponentin TypeScript-tiedosto (SimpleForm.tsx).*

C ESIMERKKEJÄ NÄKYMÄKOODIN AUTOMAATTISESTA GENEROINNISTA

```
1  <h:form>
2    <h:outputLabel for="name"
3                   value="#{i18n.name}"
4                   styleClass="labelEdit"/>
5    <h:inputText id="name"
6                 value="#{exampleBean.name}"
7                 maxlength="100">
8      <f:converter converterId="com.myapp.converter.NameConverter"/>
9      <f:validateLength maximum="100"/>
10   </h:inputText>
11   <h:outputLabel for="countryCheckbox"
12                  value="#{i18n.hasCountry}"
13                  styleClass="labelEdit"/>
14   <h:selectBooleanCheckbox id="countryCheckbox"
15                            value="#{exampleBean.hasCountry}">
16     <p:ajax update="countryPanel"/>
17   </h:selectBooleanCheckbox>
18   <gof:help text="#{i18n.countryInfo}"/>
19   <h:panelGroup id="countryPanel" layout="block" rendered="#{
       exampleBean.hasCountry}">
20     <h:selectOneMenu value="#{exampleBean.country}">
21       <f:selectItem itemValue="#{null}"
22                     itemLabel="#{i18n.notSelected}"/>
23       <f:selectItems value="#{exampleBean.countryOptions}"
24                      var="country"
25                      itemLabel="#{i18n[country]}"
26                      itemValue="#{country}"/>
27     </h:selectOneMenu>
28   </h:panelGroup>
29   <p:commandButton value="#{i18n.add}"
30                    action="#{exampleBean.addEntity()}"
31                    update=":entityList"/>
32 </h:form>
33 <ul class="entityList">
34   <ui:repeat value="#{exampleBean.entities}" var="entity">
35     <li>
36       <h:outputText value="#{exampleBean.formatName(entity.name)}"/>
```

```
37          </li>
38      </ui:repeat>
39  </ul>
```

Ohjelma C.1. *Esimerkki yksinkertaisesta JSF-sivusta.*

```
<form>
  <label for="nameInput"
         class="labelEdit">
    {{i18n.name}}
  </label>
  <input id="nameInput"
         [(ngModel)]="exampleBean.name"
         maxlength="100"
         nameConverter/>
  <label for="countryCheckbox" class="labelEdit">
      {{i18n.hasCountry}}
  </label>
  <input id="countryCheckbox"
         [(ngModel)]="exampleBean.hasCountry"
         type="checkbox"/>
  <gof-help [text]="i18n.countryInfo"></gof-help>
  <div id="countryPanel" *ngIf="exampleBean.hasCountry">
    <select [(ngModel)]="exampleBean.country">
      <option [value]="null">{{i18n.notSelected}}</option>
      <option [value]="country"
              *ngFor="let country of exampleBean.countryOptions">
        {{i18n[country]}}
      </option>
    </select>
  </div>
  <p-button [label]="i18n.add" (click)="exampleBean.addEntity()">
  </p-button>
</form>
<ul class="entityList">
  <li *ngFor="let entity of exampleBean.entities">
      {{exampleBean.formatName(entity.name)}}
  </li>
</ul>
<!--CSS Analysis:

Classes and ids: .labelEdit, #nameInput, .inputTextEdit, .labelEdit, #
    countryCheckbox, #countryPanel, .entityList

Matching css rules:

 .labelEdit {
    width: 240px;
    display: block;
}

.entityList li {
  font-size: 16px;
}
...
-->
```

Ohjelma C.2. *Esimerkistä C.1 automaattisesti generoitu Angular-versio.*

```
1  import React from 'react';
2  import { Help } from 'src/components/common/Help';
3  import { Button } from 'primereact/button';
4  import {
5    useCheckboxField, useSelectField, useTextField
6  } from 'src/hooks/formHooks';
7  import { nameConverter } from 'src/coverters/nameConverter';
8  import { i18n } from 'src/i18n/i18n';
9
10 type ExampleProps = {};
11
12 export function Example(props: ExampleProps) {
13   const nameField = useTextField(exampleBean.name, {
14     maxLength: 100,
15     converter: nameConverter,
16   });
17   const hasCountryField = useCheckboxField(exampleBean.hasCountry);
18   const countryField = useSelectField(exampleBean.country);
19   return (
20     <>
21       <form>
22         <label htmlFor="name" className="labelEdit">
23           {i18n.common.name}
24         </label>
25         <input id="name" {...nameField.inputBinding} />
26         <label htmlFor="countryCheckbox" className="labelEdit">
27           {i18n.common.hasCountry}
28         </label>
29         <input id="countryCheckbox"
30                type="checkbox"
31                {...hasCountryField.inputBinding} />
32         <Help text={i18n.common.countryInfo} />
33         {exampleBean.hasCountry && (
34           <div id="countryPanel">
35             <select {...countryField.inputBinding}>
36               <option>{i18n.common.notSelected}</option>
37               {exampleBean.countryOptions.map((country) => (
38                 <option value={country}>{i18n[country]}</option>
39               ))}
40             </select>
41           </div>
42         )}
43
44         <Button label={i18n.common.add}
45                 onClick={() => exampleBean.addEntity()} />
46       </form>
47       <ul className="entityList">
48         {exampleBean.entities.map((entity) => (
```

```
49              <li>{exampleBean.formatName(entity.name)}</li>
50            ))}
51          </ul>
52        </>
53    );
54  }
55  /*
56  CSS Analysis:
57
58  Classes and ids: .labelEdit, #name, .labelEdit, #countryCheckbox, #
        countryPanel, .entityList
59
60  Matching css rules:
61
62   .labelEdit {
63      padding-top: 5px;
64      width: 240px;
65      display: block;
66      float: left;
67      clear: left;
68      margin-bottom: 10px;
69  }
70
71  .entityList li {
72    font-size: 16px;
73  }
74  ...
75  */
```

Ohjelma C.3. *Esimerkistä C.1 generoitu React-versio.*

D USETEXTFIELD-HOOKIN TOTEUTUS

```
1  import { State } from '@hookstate/core';
2  import { useState } from '@hookstate/core';
3  import { ChangeEvent, useMemo } from 'react';
4  import { ValidationError } from 'src/util/errors';
5
6  /**
7   * Luo tekstikenttämäärityksen, jolle voi asettaa esimerkiksi
8   * kentän validointi- ja muunnossääntöjä.
9   * Jos kentän saaman tilan arvo ei ole tyypiltään string, sille pitää
      antaa
10  * configissa converter, joka osaa muuntaa arvon molempiin suuntiin.
11  */
12 export function useTextField<T>(
13   state: State<T>, config: TextFieldConfig<T> = {}
14 ): TextField<T> {
15   const validators = useMemo(
16     () => config.validators || [],
17     [config.validators]
18   );
19   const converter = config.converter || (defaultConverter as
       FieldValueConverter<T, string>);
20   // Oma kopio arvosta, jotta se voidaan pitää erillään muunnetusta versiosta
21   const innerState = useState(converter.from(state.value));
22   // Virheet päivitetään vain, kun arvo tai validaattorit muuttuvat
23   const errors = useMemo(
24     () =>
25       validators
26         .map((validator) => validator(state.value))
27         .filter((value) => value !== undefined) as ValidationError[],
28     [validators, state.value]
29   );
30
31   const setValue = (value: string) => {
32     innerState.set(value);
33     state.set(converter.to(value));
34   };
35   const reset = () => setValue('');
36   const { maxLength, minLength, required } = config;
```

```
37    return {
38      value: state.value,
39      errors,
40      reset,
41      inputBinding: {
42        value: innerState.value,
43        onChange: (event: ChangeEvent<HTMLInputElement>) => setValue(
      event.target.value),
44        maxLength,
45        minLength,
46        required,
47      },
48    };
49  }
50
51  const defaultConverter = {
52    from<T>(value: T) {
53      return value;
54    },
55    to<T>(value: T) {
56      return value;
57    },
58  };
59
60  type FieldConfig<T, FieldValue = string> = {
61    validators?: Validator<T>[];
62    converter?: FieldValueConverter<T, FieldValue>;
63    required?: boolean;
64  };
65
66  type TextFieldConfig<T> = FieldConfig<T, string> & {
67    maxLength?: number;
68    minLength?: number;
69  };
70
71  type Validator<T> = (value: T) => ValidationError | undefined;
72
73  type FieldValueConverter<T, FieldValue = string> = {
74    from(value: T): FieldValue;
75    to(value: FieldValue): T;
76  };
77
78  type Field<T> = {
79    value: T;
80    errors: ValidationError[];
81    reset: () => void;
82  };
83
```

```
84  // inputBindingin kentät voi antaa suoraan input-elementille
85  // Toteuttaa myös kaksisuuntaisen arvojen sitomisen automaattisesti
86  type TextField<T> = Field<T> & {
87    inputBinding: {
88      value: string;
89      onChange: (event: ChangeEvent<HTMLInputElement>) => void;
90      minLength?: number;
91      maxLength?: number;
92      required?: boolean;
93    };
94  };
```

Ohjelma D.1. *useTextField-hookin toteutus.*