TAMPERE UNIVERSITY OF TECHNOLOGY

Antti Laitinen

DYNAAMISEN SYÖTTEENTARKISTUKSEN ONGELMAT

Diplomityö

Tarkastaja: Mikkonen, Tommi
Tarkastaja ja aiheet hyväksytty
Tieto- ja sähkötekniikan tiedekunnan tiedekuntaneuvostossa
7. Lokakuuta 2015

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
LAITINEN, ANTTI: Dynaamisen syötteentarkistuksen ongelmat
Diplomityö, 51 sivua
Tammikuu 2016
Pääaine: Ohjelmistotuotanto
Tarkastaja: Mikkonen, Tommi
Avainsanat: Dynaaminen, Validointi, Tiedon tarkistus, Web, ERP, Spring, JavaScript

Tiedon eheys ja virheettömyys ovat tärkeitä seikkoja lähes jokaisessa tietojärjestelmässä. Erityisesti näiden seikkojen tärkeys korostuu tuotannonohjausjärjestelmissä ja muissa liiketoiminnalle kriittisissä ohjelmistoissa, joissa talletettua tietoa käytetään esim. asiakastietojen hallintaan, toiminnan seurantaan tai laskutukseen. Tietojen eheys voidaan varmistaa tarkistamalla tieto palvelimella ennen sen tallentamista tietokantaan. Usein kuitenkin on vaatimuksena, että virheiden pitää näkyä jo tietoa muokattaessa, jolloin käyttäjä voi reagoida virheisiin heti (dynaaminen tarkistus). Jos tarkistettavat tiedot ja niihin liittyvät liiketoimintasäännöt ovat lisäksi monimutkaisia, päädytään herkästi hankaluuksiin esimerkiksi suorituskyvyn ja tarkistuskoodin kahdentumisen suhteen.

Diplomityössä toteutetaan yleiskäyttöinen ja dynaaminen syötteentarkistusjärjestelmä osana laajempaa web-pohjaista järjestelmää. Järjestelmän dynaamiseen syötteentarkistukseen liittyviin haasteisiin vastataan ja niihin esitetään ratkaisu. Työssä käydään läpi toteutukselle asetettuja reunaehtoja, käydään seikkaperäisesti läpi vaatimukset täyttävä syötteentarkistimen toteutus sekä arvioidaan ratkaisun onnistumista esimerkiksi yleiskäyttöisyyden, ylläpidettävyyden ja suorituskyvyn suhteen. Työssä sivutaan nopeasti myös tietojen siirtoa uutta järjestelmää edeltäneestä sovelluksesta ja näiden tietojen tarkistamista uuden järjestelmän liiketoimintasäännöillä, jolloin voidaan käyttää samaa tarkistuslogiikkaa sekä uuteen että vanhaan tietoon.

Työssä esiteltyä ratkaisua voidaan pitää onnistuneena kohdejärjestelmän tarpeisiin, kunhan suorituskykyyn liittyviin ongelmiin kiinnitetään tulevaisuudessa huomiota. Suorituskykyongelmat eivät kuitenkaan suoraan johdu suoraan syötteentarkistusjärjestelmän arkkitehtuurista, eivätkä ne tavanomaisissa tilanteissa ole kovin vakavia. Samanlaista teknistä ratkaisua voidaankin hyvin käyttää hyödyksi myös muunlaisissa web-järjestelmissä.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
LAITINEN, ANTTI: Problems in dynamic input validation
Master of Science Thesis, 51 pages
January 2016
Major: Software Engineering
Examiner: Prof. Tommi Mikkonen
Keywords: Dynamic, Validation, Web, ERP, Spring, JavaScript

Data integrity and accuracy are vital parts of almost all software systems. They are especially important in business critical applications, such as ERP systems (Enterprise Resource Planning systems), in which the data is used in e.g. customer data management, process tracking or billing. Data integrity can be ensured by validating the data before allowing it into the database. Problems may be encountered if the data validation needs to be performed often, if the validated data or the validation rules are complicated or if the validation errors needs to be shown to the user prior to actually saving the data. The last case easily leads to duplication of the validation code base, which is certainly not desired.

In this thesis an implementation of a generalized input validation architecture that addresses these problems is showcased. The context is to use the architecture as a part of a larger web system. First, the set of boundary conditions to the implementation are introduced. Afterwards the text proceeds to present an implementation that meets the set requirements. Last, the succesfulness of the implementation is reviewed in the light of some importants metrics like level of abstraction, ease of maintenance and validation performance. The text also briefly addresses the implementation of data migration from an older application. The validation architecture needs to support validating the old data with the new business logic rules with a common validation code base.

The implemented solution is well suited for the problem and for the total system architecture, but there are some concerns related to the performance of the data validation that need to be addressed in the future. The arised problems are not directly related to the validation architecture though, and are not very serious for an usual use case. This implies that the implemented technical solution may be useful and well suited to another types of web systems as well.

ALKUSANAT

Aloitin opiskelut Tampereen teknillisellä yliopistolla syksyllä 2008 ja aika tuntuu rientäneen. Kahden ensimmäisen opiskeluvuoden jälkeen aloitin nykyisessä työpaikassani osa-aikaisena vuonna 2010. Monta vuotta kului työpaikan ja yliopiston väliä juostessa, mutta lopulta sain kaikki tutkintoon kuuluvat opinnot suoritettua alkuvuoteen 2014 mennessä. Diplomityötä lukuunottamatta tietenkin. Diplomityö tuntui pitkään kaukaiselta, ehkä pelottavaltakin, mistä syystä sen aloittaminenkin venyi. Jälkikäteen ajateltuna prosessi olisi pitänyt aloittaa jo aiemmin, sillä viimeisten kurssien jälkeen henkinen etäisyys yliopistoon ja akateemiseen maailmaan vain kasvaa. Kirjoittamista on helppo vain lykätä entistä pidemmälle. Lopulta kollegoideni kasvavan painostuksen, opinto-oikeuden rajallisuuden ja diplomityölle sopivan työprojektin myötä sain kirjoitusprosessin alkuun.

Diplomityö on kirjoitettu syksyn 2015 aikana. Työn on tarkistanut professori Tommi Mikkonen, jonka erinomaisilla neuvoilla työn rakenne, loogisuus ja ulkoasu saatiin nopeasti oikealle uralle. Haluan myös kiittää ohjaajaani Antti Pulakkaa sekä kollegaani Sami Sandqvistia kattavasta ja muutenkin loistavasta ohjauksesta diplomityön kirjoittamisessa ja oikolukemisessa. Suuri kiitos kuuluu myös työnantajalleni Vincit Oy:lle, joka myöntää kaikille diplomityötään tekeville kaksi viikkoa palkallista vapaata kirjoittamista varten. Tämä kahden viikon rauhoitettu kirjoittamisjakso auttoikin omalta osaltaan saattamaan diplomityöprosessin loppuun kivuttomasti ja kohtuullisen lyhyessä ajassa. Työpaikalta löytyi myös hyvin kattava kirjahylly lähteiden etsimiseen.

Tampereella 22. joulukuuta 2015

Antti Laitinen

SISÄLTÖ

1. Johdanto **1**

2. Järjestelmän kuvaus ja asiakasvaatimukset **3**
2.1 Keskeiset toimialan käsitteet . . . 3
2.2 Motivaatio uudelle järjestelmälle . . . 4
2.3 Syötteentarkistus . . . 4
2.4 Tunnistetut haasteet . . . 5
2.5 Työn onnistumisen mittarit . . . 6

3. Taustateknologia **7**
3.1 Palvelinteknologia: Spring Framework . . . 7
3.1.1 Dependency Injection . . . 7
3.1.2 Spring validation . . . 7
3.1.3 Object Relational Mapping . . . 8
3.1.4 Järjestelmän kerroksellisuus . . . 9
3.2 Asiakas-palvelinrajapinta . . . 11
3.2.1 URL-koodattu data . . . 11
3.2.2 JSON-muotoinen data . . . 11
3.2.3 Jackson . . . 12
3.2.4 REST-rajapinta . . . 12
3.3 Web-sovellus . . . 13
3.3.1 Tärkeimmät toteutusteknologiat . . . 13
3.3.2 Rakenne . . . 14

4. Syötteentarkistuksen toteutus **16**
4.1 Syötteentarkistin . . . 16
4.1.1 Toiminta . . . 16
4.1.2 Käyttäminen web-sovelluksessa . . . 17
4.1.3 Peruslomake-elementit . . . 18
4.1.4 Moniosaiset kentät . . . 19
4.1.5 Tiedostonlisäyskentät . . . 19
4.2 Syötteiden lähetys . . . 20
4.2.1 Datan sarjallistaminen . . . 20
4.2.2 Palvelimen rajapinta . . . 22
4.3 Lomakedatan käsittely palvelimella . . . 23
4.3.1 Validaattorit . . . 23
4.3.2 Virheyhteenvedon laadinta . . . 25
4.4 Virheviestien käsittely . . . 25

4.4.1 Validointivastaus 25
4.4.2 Virheiden käsittely 26
4.4.3 Virheiden esittäminen käyttöliittymässä 27
4.5 Varoitusviestit ja muut käyttötapaukset 30
4.5.1 Varoitusvalidointi 30
4.5.2 Vain lukutila -validointi 34
4.5.3 Ehdotetut toiminnot 35
4.6 Datamigraation huomiointi 37
4.7 Alustava suorituskyvyn huomiointi 41

5. Tulosten arviointi **43**
5.1 Suorituskyky 43
5.1.1 Syötteentarkistuksen suoritusaikoja 43
5.1.2 Suorituskyvyn riittävyyden arviointi 44
5.2 Datamigraation onnistuminen 45
5.3 Ylläpidettävyys 45
5.3.1 Dokumentaation määrä 46
5.3.2 Koodin määrä 46
5.3.3 Valmiin kirjastokoodin tuki 47
5.3.4 Automaattiset testit 48
5.3.5 Ylläpidettävyyden arviointi 49
5.4 Jatkotoimenpiteet 49

6. Yhteenveto **51**

Lähteet **51**

1. JOHDANTO

Tuotannonohjausjärjestelmää kehitettäessä on oleellista kiinnittää huomiota tiedon eheyteen ja oikeellisuuteen. Tästä syystä kaikki järjestelmään syötettävä tieto tulisi tarkistaa palvelimella ennen sen tallentamista ja havaituista virheistä ilmoittaa selkeästi käyttäjälle. Usein on myös käyttäjäystävällistä ilmoittaa tehdyistä virheistä heti kun ne tehty, eikä vasta kun käyttäjä yrittää viimein tallentaa täyttämäänsä lomaketta. Erityisesti tämä korostuu, jos lomakkeen tiedot riippuvat toisistaan siten, että alussa tehty virhe vaikuttaa myös muiden tietojen oikeellisuuteen. Jos tallennettavat kokonaisuudet ovat lisäksi niin suuria, että lomakkeen täyttämiseen kuluu paljon työaikaa on virheistä ilmoittaminen myös hyvä työtehon lisäämiskeino. Tässä työssä käytetty termi *syötteentarkistuksen dynaamisuus* tarkoittaa juurikin sitä, että lomaketta tarkistetaan jo samalla kun sitä muokataan. Dynaamisuus tuo tiedon tarkistukseen haasteita, sillä tarkistuskertoja ja -kerroksia on useita. Tiheä tarkistusväli yhdistettynä raskaaseen tarkistuslogiikkaan aiheuttaa helposti suorituskykyongelmia. Lisäksi web-järjestelmissä palvelimen ja käyttäjäsovelluksen toteutuskielet ja -teknologiat eroavat toisistaan, jolloin tarvitsee miettiä jokin keino välttää tarkistuskoodin kaksinkertaistaminen.

Tässä työssä esitellään dynaamisen web-pohjaisen syötteentarkistimen toteutus, joka ottaa edellä mainitut haasteet huomioon. Syötteentarkistin on osa laajempaa järjestelmää, jonka on tarkoitus korvata edeltävä, yli kymmenen vuotta vanha tuotannonohjausjärjestelmä eräälle suomalaiselle sähköalan yritykselle. Yrityksellä on yhteensä useita kymmeniä tuhansia asiakkuuksia, joiden sähkönhallinta ja -hankinta on yrityksen vastuulla. Toteutetun järjestelmän piiriin kuuluu käytännössä yrityksen koko liiketoiminta sisältäen mm. kaikki asiakkuushallinnan, laskutuksen, sähkönhankinnan sekä asiakastiedotuksen toiminnot. Ohjelmisto on toteutettu räätälöitynä tilaustyönä ja toteutus on ollut käynnistä syksystä 2013. Arvioitu käyttöönotto järjestelmälle on suunniteltu toteutuvan vuonna 2016. Vaikka syötteentarkistuskomponentti onkin toteutettu näin tiiviisti osana asiakasprojektia, niin esiteltävä ratkaisu ei ole sinänsä toimiala- eikä asiakasriippuvainen. Lisäksi samanlainen ratkaisu sopii varmasti myös muunlaisiin kuin tuotannonohjausjärjestelmiin. Sähköalan käsitteistö ja siihen liittyvät esimerkit ja onnistumismittarit on kuitenkin pidetty osana työtä, sillä näitä käsitteitä käytetään erityisesti onnistumisen arvioinnissa hyödyksi.

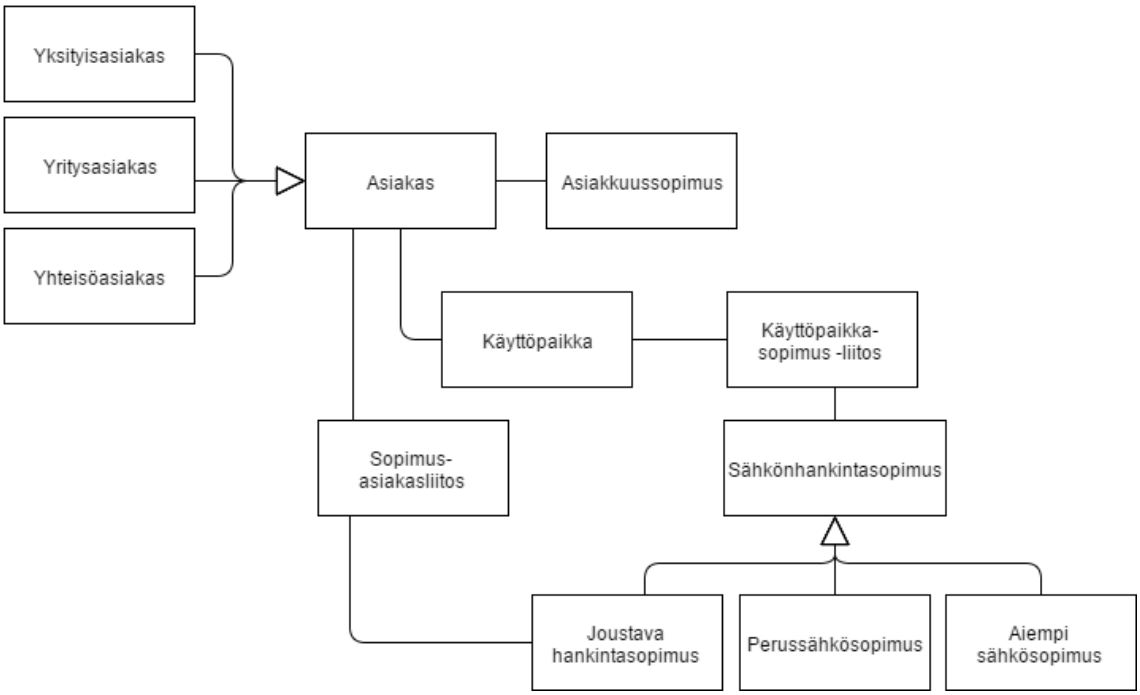
Ensimmäiseksi luvussa 2 esitellään lyhyesti toimialan käsitteitä, lähinnä siksi, että

tekstiä ja erityisesti siinä käytettyjä esimerkkejä on helpompi seurata. Samalla käydään läpi keskeisimmät motivaatiotekijät edellisen järjestelmän korvaamiselle, uudelle järjestelmälle esitetyt asiakasvaatimukset sekä muut toteutuksen reunaehdot. Toimialan ja vaatimusten esittelystä siirrytään taustateknologian, eli käytettyjen ohjelmointikehysten, kirjastojen sekä järjestelmän yleisarkkitehtuurin kuvaukseen luvussa 3. Nämä tekijät vaikuttavat merkittävästi syötteentarkistuksen toteutukseen, joka esitellään luvussa 4. Toteutusratkaisun käsittelyn jälkeen siirrytään arvioimaan toteutuksen onnistumista luvussa 2 esiteltävien onnistumismittarien suhteen. Jatkokehitykseen liittyviä seikkoja ja mahdollisia ratkaisuja havaittuihin ongelmiin käsitellään myös lyhyesti.

2. JÄRJESTELMÄN KUVAUS JA ASIAKASVAATIMUKSET

2.1 Keskeiset toimialan käsitteet

Järjestelmän tärkeimmät käsitteet ja niiden väliset suhteet on tiivistettynä esitetty kuvassa 2.1. Järjestelmän keskeisin käsite on asiakas, jonka kanssa yritys solmii sopimuksen sähkönhallinnan ulkoistamisesta (asiakkuussopimus). Käyttöpaikat ovat ne asiakkaat asunnot, kiinteistöt tai tilat, joissa sähkösopimuksen alaista sähköä käytetään. Asiakas voi olla yksityinen henkilö, yritys tai yhteisö, jonka käyttöpaikka tai -paikat siirtyvät yrityksen sähkönhankinnan piiriin. Sähkönhankinta toteutetetaan kilpailuttamalla sähkönmyyjiä, joiden kanssa yritys solmii joustavia hankintasopimuksia suurelle joukolle käyttöpaikkoja. Sopimuksen joustavuus tarkoittaa, että sopimuksen piiriin voidaan liittää asiakkaita ja näiden käyttöpaikkoja dynaamisesti. Perussähkösopimus sen sijaan laaditaan yleensä yksityishenkilöille. Hallinnoitujen käyttöpaikkojen sähkö hankitaan suurina erinä sähköpörssistä sähkönkiinnityksinä.



Kuva 2.1: Käsitekaavio tärkeimmistä käsitteistä

2.2 Motivaatio uudelle järjestelmälle

Ennen uuden järjestelmän käyttöönottoa yrityksessä on käytetty vastaavaa vanhaa web-pohjaista tietojärjestelmää. Järjestelmä on toteutettu alunperin pienellä budjetilla ja on sittemmin kasvanut yrityksen mukana samalla kun siihen on vähitellen lisätty uusia toimintoja. Toteutustiimi on myös vaihtunut ajan kuluessa ja nykyisin järjestelmää ylläpitää ainoastaan yksi ohjelmisto-osaaja. Kehitystapa on johtanut siihen, että tiedon- ja syötteentarkistus on huolimatonta, järjestelmässä on erityislogiikkaa asiakaskohtaisesti sekä suorituskyky on paikoin erittäin heikko. Viimein on saavuttu pisteeseen, jossa uusien toimintojen lisääminen on erittäin vaikeaa, tietokannan tieto on korruptoitunutta ja järjestelmä toimii ainoastaan tarpeeksi vanhan Internet Explorer -selaimen yhteensopivuustilassa.

Keskeisimpiä motivaatiotekijöitä ja samalla tärkeimpiä asiakasvaatimuksia tietojärjestelmäuudistukselle on siis tärkeysjärjestyksessä:

1. Tarve järjestelmälle, jossa tietokantaan ei päädy virheellistä tietoa.
2. Tarve reaaliaikaiselle ja kattavalle syötetyn tiedon tarkistamiselle. Järjestelmän monimutkaisten käsitesuhteiden vuoksi myös ammattilaiselle sattuu helposti ajatusvirheitä lomakkeiden täytössä.
3. Tarve järjestelmälle, joka kattavasta tarkistuksesta huolimatta on suorituskykyinen ja joka nostaa työn tuottavuutta.
4. Tarve uusille ominaisuuksille, joita ei ole enää mielekästä tai välttämättä edes mahdollista toteuttaa vanhaan järjestelmään.

2.3 Syötteentarkistus

Uuden järjestelmän tärkeimpiä motivaattoreita on tiedonhuolto ja se, että tiedon tarkistus on riittävän suorituskykyistä. Kaikkein oleellisinta on se, että virheellisen ja järjestelmän ristiriitaiseen tilaan saattavan tiedon syöttäminen ei olisi mahdollista. Lähes yhtä tärkeää on myös, että monivaiheisten ja hankalia käsitteitä sisältävien lomakkeiden virheellisestä täytöstä saa palautteen heti, eikä vasta yritettäessä tallentaa lomaketta.

Lisävaatimuksena syötteentarkistukselle oli, että tietyissä tilanteissa järjestelmän on osattava ehdottaa järkeviä oletusarvoja ja lisätoimenpiteitä käytettävyyden parantamiseksi. Esimerkiksi lähes aina käyttöpaikkojen liitokset hankintasopimuksiin alkavat ja päättyvät samoina päivinä kuin liitettävä sopimus, joten näitä päivämääriä on ehdotettava automaattisesti. Toisaalta myös uusi sopimus alkaa lähes aina edeltävää sopimusta seuraavana päivänä, joten uuden sopimuksen alkamispäiväksi on osattava ehdottaa edeltävän sopimuksen loppupäivän jälkeistä päivää.

Kaikki ongelmalliset syötteet eivät myöskään ole niin vakavia että ne estäisivät tallennuksen. Osasta virheitä on annettava vain välitön varoitus, mutta käyttäjä voi harkintansa mukaan tallentaa lomakkeen silti. Lisäksi tarvitaan keino estää joidenkin kenttien muokkaus, kun lomake on tietyssä tilassa.

2.4 Tunnistetut haasteet

Vasteaika. Järjestelmän arvioitu kuormitus on noin 100-200 yhtäaikaista käyttäjää. Tietokantatauluja järjestelmässä on noin 200 ja näillä keskenään erittäin paljon vierasavainviittauksia. Usein tarvittava tieto näkymälle joudutaan tekemään useiden liitosten kautta, mikä yhdistettynä tietokannan olioabstraktioon (ORM, object-relational mapping) asettaa haasteita suorituskyvylle. Tämä haaste koskee tietenkin koko järjestelmää, mutta erityisen paljon siitä on painetta nimenomaan dynaamiselle syötteentarkistukselle. Kokenut käyttäjä tekee lomakkeeseen paljon muutoksia lyhyessä ajassa, ja kaikki virheet näissä tiedoissa täytyisi saada korkeintaan muutaman sekunnin viiveellä näkyviin käyttöliittymään. Monimutkaisesta liiketoimintalogiikasta johtuen tietueita ei voida tarkistaa paikallisesti, sillä muutettavan tiedon kaikki suhteet muihin tietoihin täytyy myös tarkistaa. Usein sallittujen arvojen joukko tietylle kentälle riippuu joko saman lomakkeen muista arvoista tai muiden tietokantataulujen rivien arvoista. Käytännössä siis koko lomake on aina lähetettävä tarkastettavaksi, eikä paikallisesti voida tarkistaa kuin kaikkein yksinkertaisimpia asioita.

Käsitteiden väliset suhteet. Eääksi keskeisimmistä haasteista järjestelmän kehityksessä on ollut, että sovellusalueen osittainenkin ymmärtäminen vaatii vuosien kokemuksen alalta. Edes järjestelmän tilaajalta ei löyty yhtäkään sellaista henkilöä, joka ymmärtäisi koko liiketoimintaprosessin kaikkine vivahteineen. Syötteentarkistuksen suhteen tämä luonnollisesti tarkoittaa vaikeasti määriteltävää, monimutkaista syötteentarkistuskoneistoa. Kaiken lisäksi on helppoa päätyä ns. ”dead lock” -umpisolmutilanteeseen, jossa eri tietokantataulujen rivit estävät toistensa muokkaamisen. Esimerkiksi lisättäessä sopimus-käyttöpaikkaliitosta (kuva 2.1), on tarkistettava asiakkuuden voimassaolo suhteessa sopimuksen voimassaoloon, valtakirjan olemassaolo, liitettävän käyttöpaikan tiedot ja voimassaolo, edeltävien sopimusten aktiivisuudet ja näiden voimassaolojaksot.

Edeltävän järjestelmän migraatio. Oleellisena huolenaiheena minkä tahansa toiminnanohjausjärjestelmän uudistuksessa on luonnollisesti vanhan tiedon siirto uuteen järjestelmään. Tavoitteena on saada vanhan järjestelmän data tarkistettua uuden järjestelmän tarkistuskoneistolla, saada löydetyt virheet korjattua sekä lopulta korjattu tieto tallennettua uuteen järjestelmään. Erityisen haastavaksi tämän tavoitteen tekee, että edeltävän järjestelmän tietokanta on rönsyinen, normalisoimaton ja vaikeasti ylläpidettävä. Tietoja ei ole eriytetty kunnolla omiksi relaatioikseen

ja relaatioiden nimet ovat epäkuvaavia (esim. temp_1, temp_2). Suorituskyvyn optimointi on myös huonoissa kantimissa. Vanhaa järjestelmää ylläpitää ja sen toimintaa ymmärtää kirjoitushetkellä vain yksi henkilö, mikä aiheuttaa luonnollisesti vaaran ylläpidon jatkuvuudelle.

2.5 Työn onnistumisen mittarit

Kun uutta järjestelmää ollaan ottamassa käyttöön, on hyvä miettiä mittareita työn onnistumiselle. Onnistumisen mittareiksi on valittu asiakasvaatimusten pääkohdat, kuten järjestelmän nopea vaste (suorituskyky), vanhan järjestelmän migraation huomioiminen sekä uuden järjestelmän ylläpidettävyys. Tässä työssä keskitytään näihin mittareihin nimenomaan syötteentarkistustyökalujen osalta, vaikka samat mittarit on laajennettavissa myös koko järjestelmään.

Järjestelmän suorituskykyä tarkastellessa tärkeitä mittareita ovat:

1. Syötteentarkistuksen vaste on oltava mahdollisimman lyhyt, korkeintaan muutama sekunti.

2. Vanhan järjestelmän data on tarkistettava mahdollisimman tehokkaasti. Tietokannan rivejä on paljon, joten yhden taulun migraatioajo ei saisi kestää päiväkausia.

3. Syötteentarkistus ei saisi kuormittaa palvelinta liikaa, jottei vaste kasva ja käytettävyys kärsi muilla järjestelmän osa-alueilla.

Datamigraation onnistuminen edellyttää, että mahdollisimman paljon vanhasta tiedosta saadaan siirrettyä automaattisesti uuteen järjestelmään. Tarvittaessa loput puutteellisista tai virheellisistä tiedoista siirretään tilaajan toimesta käsin. Tämän työn kannalta oleellisin onnistumisen mittari kuitenkin on, että datamigraatio on jotenkin huomioitu osana tarkistusjärjestelmää.

Ylläpidettävyys on ollut edellisessä järjestelmässä suuri haaste. Onkin siis järkevää kiinnittää uuden järjestelmän toteutuksessa erityistä huomiota ylläpidettävyyteen. Järkeviä mittareita tälle ovat esimerkiksi tietokannan normalisointi, dokumentaatio, automaattiset ja manuaaliset testitapaukset, koodin määrä sekä valitut toteutuskirjastot. Työn kannalta oleelliset mittarit ovat dokumentaatio, koodin kompleksisuus ja määrä, testitapaukset sekä tunnettujen toteutuskirjastojen käyttö.

3. TAUSTATEKNOLOGIA

3.1 Palvelinteknologia: Spring Framework

Spring Framework on Java-ohjelmointikielellä toteutettu web-sovelluskehys [1]. Ensimmäinen versio on julkaistu vuonna 2002. Projektin käyttämä versio 3 on vuodelta 2013, ja työn kirjoitushetkellä uusin versio on kesällä 2015 julkaistu 4.0. Spring Framework noudattaa olio- sekä aspektiohjelmoinnin paradigmoja.

3.1.1 Dependency Injection

Yleisellä tasolla Dependency Injection eli *riippuvuusinjektio* on yksinkertainen suunnittelumalli, jossa luokan riippuvuudet on ulkoistettu jollekin muulle, riippuvuudet tarjoavalle, taholle. Toisin sanoen, tietyistä rajapinnoista riippuvaisen luokan ei tarvitse tietää mikä toteutus sille annetaan. Kyseisestä asiasta käytetään myös termiä *loose coupling*, eli löyhä liitos. [2]

Spring Frameworkissa riippuvuusinjektion avulla löyhästi liitettyjen luokkien tarjoaminen toisilleen on tehty helpoksi. Injektoitava luokka merkitään annotaatiolla `@Component` ja se voidaan liittää siitä riippuvaan toiseen komponenttiin annotaatiolla `@Resource`. Listauksen 3.1 esimerkissä on käsitelty kahta luokkaa ja niiden välistä injektiota. Löyhä riippuvuussuhde voisi olla myös kaksisuuntainen. Kaikki tällä tavalla rakennetut oliot ovat ns. singleton -instansseja, eli luokista on kerrallaan vain yksi olio järjestelmässä. [1]

3.1.2 Spring validation

Spring Validation (org.springframework.validation) on tämän työn kannalta erittäin tärkeä osa Spring-ohjelmistokehystä, ja siihen kuuluu kaksi oleellista osaa: olion validoinnin yhtenäistävä Validator-rajapinta sekä käyttäjäsyötteen Java-olioksi muuntava DataBinder-luokka [1] [3].

Validator-rajapintaa käytetään tavanomaisten Java-olioiden (POJO) automaattiseen validointiin. Kyseessä on yksinkertainen, mutta yhtenäinen rajapinta, jonka toteuttavalle validaattorille annetaan validoitava olio ja joka palauttaa Errors -tyyppisen virheviestiolion. Virheolio sisältää käytännössä tietue-virhekoodipareja sekä yleisesti koko oliota koskevia, globaaleja virhekoodeja.

Listaus 3.1: Esimerkki dependency injection -suunnittelumallista

```
1  /* InjectedComponent.java */
2  @Component
3  public class InjectedComponent {
4    public void serviceMethod() {
5       /* Toteutus */
6    }
7  }
8
9  /* AnotherComponent.java */
10 @Component
11 public class AnotherComponent {
12   @Resource
13   private InjectedComponent injectedComponent;
14
15   public void foo() {
16     injectedComponent.serviceMethod();
17   }
18 }
```

DataBinder-luokan tehtävä on käyttäjäsyötteen URL-koodatun tai JSON-muotoisen merkkijonon muuntaminen vastaavaksi Java-olioksi, jolloin sen käsittely palvelimella suoraviivaistuu merkittävästi. Käyttäjäsyötteen käsittelyä tarkastellaan lähemmin jäljempänä, osana asiakas-palvelinrajapinnan toteutusta.

3.1.3 Object Relational Mapping

Perinteinen ongelma web-palvelinsovelluksen ja sen takana olevan säilytysteknologian kuten relaatiotietokannan välillä on datamuotojen ja paradigmojen erilaisuus. Relaatiotietokannan tieto on järjestetty tauluiksi ja riveiksi ja näiden väliset suhteet vieras-pääavainviittauksiksi. Palvelinohjelmiston tietorakenne taas on yleensä erilainen, esimerkiksi Javan tapauksessa käytössä on olioparadigma. Olioparadigmassa käsitteitä mallinnetaan luokilla ja luokan ominaisuuksia jäsenmuuttujilla. Suhteet toisiin käsitteisiin yleensä myös mallinnetaan täysin päinvastaisesti: relaatiotietokannassa lapsirelaatio viittaa vanhempaansa kun taas oliomallissa olio omistaa lapsensa. Myös lievempiä yhteensopivuusongelmia, kuten tietotyyppien eroja esiintyy relaatio- ja oliomallien välillä. Aatteellisiakin eroja on, sillä relaatiotietokannan paradigma on *deklaratiivinen* (logiikkaa ohjataan tietorakentein ja rajoittein) ja oliomalli lähinnä *imperatiivinen* (logiikkaa ohjataan eksplisiittisin komennoin). [4]

Automaattista tiedonmuuntoa näiden paradigmojen välillä hoitavaa ohjelmointitekniikkaa kutsutaan nimellä Object Relational Mapping (ORM). Javalle erilaisia ORM-työkaluja on lukuisia, ja näistä suosituin on Hibernate [5]. Se on erityisen kätevä Spring Frameworkin rinnalla käytettäväksi, sillä se toteuttaa Java Persistence -rajapinnan (JPA) ja pystyy siten ottamaan vastuulleen suuren osan tietokannan

kyselyistä, tietojen talletuksesta sekä tietokantaskeeman luonnista. Hibernate osaa automaattisesti luoda tietokannan taulut, pää- ja vierasavaimet sekä oleelliset rajoitteet (database constraints) tavallisista Java-luokista (POJO). Apuna tietokannan luomisessa käytetään luokkaan ja jäsenmuuttujiin liitettäviä Javan annotaatiomerkintöjä kuten `@Table` ja `@Column`. Taululle voi luoda myös surrogaattipääavaimen annotaatiolla `@Generated`. Listauksessa 3.2 on esitetty yksinkertaisen esimerkkitaulun luova luokka. [4]

Listaus 3.2: Esimerkki Hibernaten avulla luotavasta taulusta

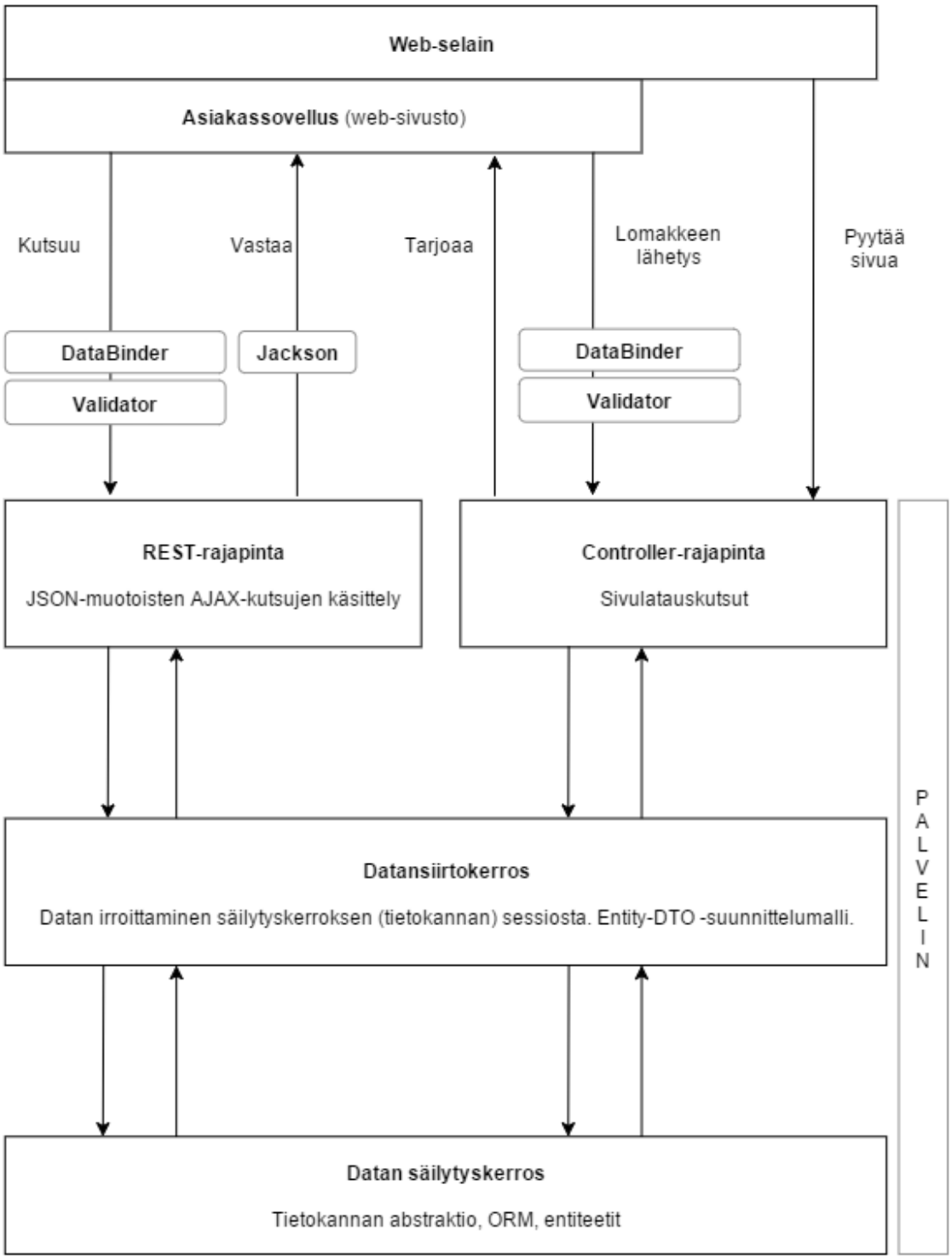
```
1 @Table(name="example_table")
2 public class Example {
3     @Generated
4     @Column(name="id")
5     private Long id;
6
7     @Column(name="name")
8     private String name;
9 }
```

3.1.4 Järjestelmän kerroksellisuus

Kohdejärjestelmän kerrosrakenne on yksinkertaistettuna esitetty kuvassa 3.1. Ylimpänä on web-selain ja siinä suoritettava asiakassovellus (MVC-suunnittelumallin view, *näkymä*). Asiakassovelluksen koodi tuotetaan ohjauskerroksessa (MVC:n controller, *ohjain*), ja selain voi pyytää muita näkymiä ohjaimelta esim. hyperlinkkien tai automaattisten sivusiirtymien seurauksena. Dynaamisia toimintoja varten on REST-rajapinta, jonka kanssa asiakasohjelma keskustelee JSON-tietoformaatin välityksellä. Ohjainkerrokselle lähetettävä syöte muutetaan Java-olioksi DataBinder-luokassa ja tarkistetaan Spring Validationin Validator-rajapinnan avulla. Jos lähetettävä syöte on väärän muotoista tai virheellistä, palautetaan HTTP-koodi 400 ”Bad Request” eli vääränmuotoinen pyyntö.

Datansiirtokerroksessa siirretään tietoa ohjaimen ja säilytyskerroksen välillä. Se siis abstrahoi säilytyskerroksen toteutustavan siten, ettei ohjain- ja näkymäkerrosten tarvitse tietää miten tietoa tallennetaan. Hibernaten avulla Javan olioksi muutettu relaatioentiteetti on vielä kiinnitetty tiettyyn tietokantaistuntoon, jolloin siihen liittyviä suhteita voidaan noutaa vielä saman istunnon aikana. Kun tarvittavat tiedot on haettu, luodaan siitä DTO, *data transfer object* eli datansiirto-olio. DTO on POJO, ja sillä ei ole mitään erityistä toiminnallisuutta, vaan ainoastaan rakenne. Tässä vaiheessa riippuvuus tietokantaan on katkennut, ja datansiirto-oliota voidaan vapaasti välittää eri sovelluskerroksille.

Datan säilytyskerros on abstraktio tiedon tallettamiselle. Lähes kaikissa näin toteutetuissa web-järjestelmissä tämä kerros on vastuussa kommunikoinnista jonkin



Kuva 3.1: Yleiskuva järjestelmän palvelinpuolen kerroksista.

relaatiotietokannan kanssa, mutta talletusteknologia voi hyvin olla myös esim. muistitietokanta tai jokin NoSQL-toteutus. Datan säilytyskerros toteuttaa Java Persistence -rajapinnan, jonka tarkoituksena on abstrahoida eri toteutusteknologiat yhtenäisen sovellusohjelmointirajapinnan (API, application programming interface) taakse. Java Persistence -rajapinta sisältää työkalut mm. relationaalisen datan hallintaan olioympäristössä sekä yhtenäisen kyselykielen relationaalisen datan hakemiseen (Criteria API). [1] [4]

3.2 Asiakas-palvelinrajapinta

Kuvassa 3.1 esitettyyn näkymä- ja ohjainkerrosten väliseen kommunikaatioon käytetään kahta eri dataformaattia. HTTP-protokollan mukaisen pyynnön ja vastaavan vastauksen runkoon liitetään URL-koodattu tai JSON-muotoinen tietorakenne merkkijonona. JSON on erityisesti JavaScriptin yhteydessä helppokäyttöinen dataformaatti, mutta perinteisten HTML-lomakkeiden yhteydessä kannattaa käyttää selaimen natiivisti tukemaa URL-koodattua eli ns. *x-www-url-encoded* -muotoa.

3.2.1 URL-koodattu data

URL-koodattussa datamuodossa [6] data lähetetään merkkijonona, jossa tieto on koodattu avain-arvo -pareina. Nimensä mukaisesti URL-koodattu muoto sopii `GET`-metodin HTTP-pyyntöihin osana URL-osoitetta. Yleensä lomakkeen lähetyksessä, eli `POST`-metodin HTTP-pyynnössä data kuitenkin koodataan pyynnön runko-osaan. Esimerkki yksittäisten arvojen muotoilusta avain-arvopareiksi on esitetty listauksessa 3.3. Listallinen arvoja voidaan myös esittää yhden avaimen avulla, josta on annettu esimerkki listauksessa 3.4.

Listaus 3.3: Esimerkki yksittäisistä avain-arvopareista

```
1 Request URL: https://example.com/resource1
2 Request Method:POST
3 Content-Type: application/x-www-urlencoded
4 Request Body: avain1=arvo1&avain2=arvo2
```

Listaus 3.4: Esimerkki listallisesta arvoja yhdellä avaimella

```
1 Request URL: https://example.com/resource1
2 Request Method:POST
3 Content-Type: application/x-www-urlencoded
4 Request Body: lista=1&lista=2&lista=3
```

3.2.2 JSON-muotoinen data

JSON, eli *JavaScript Object Notation*, on ECMA-404 -standardissa määritelty tiedonsiirtoformaatti, joka perustuu JavaSciptin standardiin Standard ECMA-262 3rd

Edition vuodelta 1999. [7]

JSON-formaatissa datan rakenne noudattaa läheisesti JavaScriptin objektisyntaksia, jossa tieto esitetään rakenteisesti avain-arvopareina siten, että arvo voi olla jokin perustietotyyppi, lista tai sisempi avain-arvojoukko. Esimerkki rakenteesta on listauksessa 3.5.

Listaus 3.5: Esimerkki JSON-tiedonsiirtoformaatista

```
1 {
2     "id": 1,
3     "lista": [1,2,3,4],
4     "tiedot": {
5       "nimi": "Esimerkki",
6         "ika": 27
7     }
8 }
```

JSON-formaatti on suunniteltu mahdollisimman yksinkertaiseksi ja kevyeksi. Lisäksi sen ei nimenomaan yksinkertaisuutensa vuoksi uskota muuttuvan [7], eli se on myös ylläpidollisesti järkevä ja stabiili ratkaisu.

3.2.3 Jackson

Jackson on Java-kirjasto [8], joka on suunniteltu kahdensuuntaiseen datamuunnokseen merkkijonomuotoisen JSON-datan sekä Javan POJO-olion välillä. Java-olion jäsenmuuttujien nimet siirtyvät JSON-rakenteen avaimiksi ja jäsenmuuttujien arvot avainten arvoiksi.

3.2.4 REST-rajapinta

REST *(Representational State Transfer)* on WWW-sovelluksissa usein käytetty arkkitehtuurimalli [9], jossa palvelimelle tallennettuja tietoja pidetään resursseina. Resursseilla on aina oma osoite (URL), jonka avulla niihin pääsee HTTP-protokollan avulla käsiksi (esim. /tuotteet/<tuote-id>). Operaation luonne määritellään lisäksi HTTP-metodin eli esim. `GET, POST, PUT` ja `DELETE` avulla. Esim. `HTTP DELETE` -muotoinen pyyntö palvelimen osoitteeseen `/tuotteet/54211` poistaa tuotteen (resurssin), jonka tuote-id on 54211. Sen sijaan `GET`-metodi samaan resurssiosoitteeseen palauttaa tiedot jossain sopivassa tietomuodossa. Tässä projektissa käytössä on JSON-muotoinen data REST-kutsujen yhteydessä.

Puhtaassa REST-arkkitehtuurimallissa käyttäjän tilaa ei tallenneta palvelimelle (esim. selainevästeiden avulla), vaan suoritettava toiminto riippuu ainoastaan lähetettävän HTTP-pyynnön tiedoista. Käytännössä tätä ei kuitenkaan aina noudateta, sillä esim. käyttäjän kirjautumisen tila on helpompi, nopeampi ja usein myös turvallisempi toteuttaa perinteisesti evästeen mukana kulkevana istunto-id:nä varsinkin,

jos REST-rajapinnan lisäksi on käytössä myös perinteinen ohjainkerros (controller). Näin vältetään lähettämästä tarkkoja kirjautumistietoja jokaisen HTTP-pyynnön yhteydessä, kuten esim. ”HTTP Basic” ja ”HTTP Digest” -autentikointimalleissa. [9]

Tässä projektissa REST-rajapintaa käytetään dynaamiseen datanvaihtoon palvelimen kanssa. Tällä tavalla ei tarvita kokonaan uutta sivunlatausta, kun siirretään tietoja palvelimelle tai palvelimelta selaimelle. Myös kaistaa säästyy, sillä JSON-muodossa siirretään vain tarvittava määrä tietoja selaimelle kokonaisen sivun sijaan. Sivulataukset ja lomakkeen lähetykset on järjestelmässä toteutettu perinteisesti ohjain-rajapinnan välityksellä.

3.3 Web-sovellus

Web-sovellus on järjestelmän kerros, joka näytetään ja jonka koodi suoritetaan käyttäjän omassa selaimessa. Tarkastellaan seuraavaksi web-sovellukselle tärkeimpiä toteutusteknologioita sekä itse sovelluksen rakennetta.

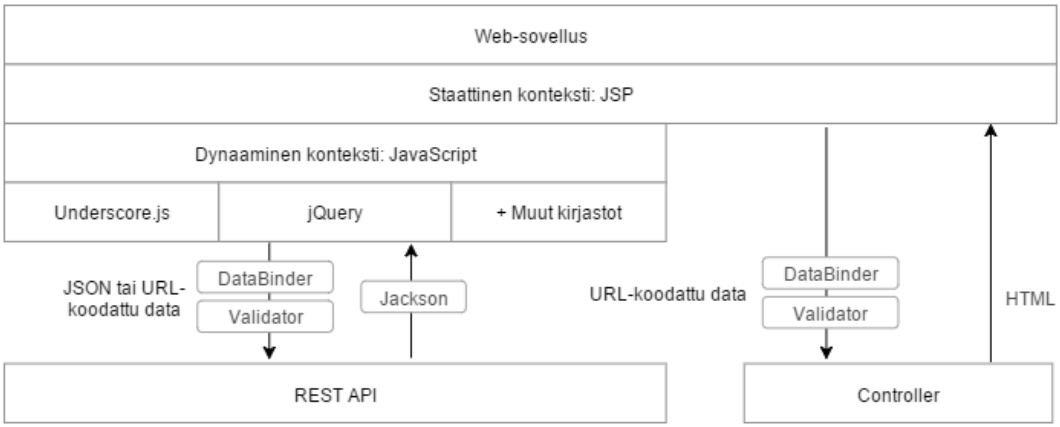
3.3.1 Tärkeimmät toteutusteknologiat

JSP ja JSTL. JSP eli *JavaServer Pages* [10] on alunperin 1999 julkaistu PHP-klooni Java-kielelle, jota käytetään yhdessä Java servlet -säiliön (container), kuten Apache Tomcatin kanssa. Projektissa JSP-sivuja käytetään kuitenkin vain lähinnä alustana muille teknologioille, sillä toteutuslogiikka on haluttu irrottaa kokonaan näkymäkerroksesta. JSTL on lyhenne sanoista *JSP standard tag library*. Kirjasto on laajennos JSP-kieleen, sisältäen yleensä web-projekteissa tarvittavia ominaisuuksia. Lopulliset JSP-sivut generoidaan HTML-koodiksi palvelimella ja palautetaan selaimelle sivupyynnön vastauksessa. [10] [11]

HTML5 on HTML-standardin uusin versio. Standardi valmistui lopullisesti lokakuussa 2014, mutta myös ennen sitä HTML5 on ollut jo laajassa käytössä. Se on yhdistelmä uusimpia HTML-, JavaScript ja CSS-tekniikoita ja siinä on aiempia versioita paremmin huomioitu dynaamiset web-sovellukset sekä mobiililaitteet. [12]

jQuery on kirjasto, joka sisältää paljon yleisesti tarvittuja funktioita JavaScript-kieleen. Tärkeimpinä on HTML-elementtien hakeminen DOM- eli dokumenttipuurakenteesta *(document object model)* sekä Ajax-pyyntöjen helppo tekeminen `$.ajax()` -funktion avulla. [13]

Ajax eli alunperin *Asynchronous JavaScript and XML* on yleisnimitys tekniikoille, joiden avulla palvelimen kanssa kommunikoidaan ja sovelluksen tilaa muokataan asynkronisesti. Ajax-toteutukseen käytetään tässä järjestelmässä jQueryn tarjoamaa `$.ajax()` -funktiota. Nimestään huolimatta datansiirtotekniikan ei tarvitse olla XML, vaan se voi olla myös URL-koodattua tai, kuten yleisimmin, JSON-



Kuva 3.2: Web-sovelluksen staattinen ja dynaaminen konteksti

muotoista tietoa. Myöskään toteutuskielen ei tarvitse olla JavaScript, jotta voidaan puhua Ajax-tekniikasta. Nykyisin akronyymi onkin menettänyt sinänsä merkitystään, mutta sana on jäänyt elämään. [9]

Underscore.js on kirjasto, joka toteuttaa JavaScriptille tärkeimmät funktionaalisen ohjelmointiparadigman mukaiset operaatiot, kuten esimerkiksi *map*, *reduce*, *filter*, *sort* ja *unique*. Nämä tekevät monenlaisten tietorakenteiden muuntamisesta, järjestelystä ja analysoinnista huomattavasti helpompaa. Toinen underscore.js:n ominaisuus on oma mallikieli (templating language), jota käytetään dynaamisesti JavaScriptillä generoitavien näkymien tuottamiseen (vrt. palvelimella generoitavat JSP-pohjat). [14]

Dropzone.js on ”drag-and-drop” -toiminnallisuuden toteuttava avoimen lähdekoodin tiedostonlatauskomponentti html-lomakkeiden yhteyteen [15]. Kirjasto toteuttaa monia ominaisuuksia kattavammin ja helppokäyttöisemmin kuin tavallinen HTML:n tiedostonlisäyselementti.

3.3.2 Rakenne

Web-sovelluksen rakenne on esitetty kuvassa 3.2. Web-sovelluskerroksen voidaan ajatella rakentuvan edelleen sekä staattisesta että dynaamisesta kontekstista. Staattinen konteksti valmistellaan palvelimella JSP-tekniikalla ja palautetaan selaimelle tekstimuotoisena HTML-sivuna. HTML-koodi jäsennetään selaimessa ja siitä luodaan puumainen tietorakenne, jota kutsutaan dokumenttioliomalliksi (*DOM, Document Object Model*) [16]. Staattinen konteksti sisältää myös dynaamisen kontekstin eli sovelluksen JavaScript-koodin. JavaScript-kerros kommunikoi palvelimen kanssa Ajax-kutsuilla REST-rajapinnan välityksellä. REST-rajapinnan vasteen avulla voidaan tuottaa sivulle dynaamisesti muuttuvaa sisältöä JavaScriptin ja erityisesti jQueryn DOM-manipulaatiotyökalujen avulla. Käyttäjäsyötteen tarkastuksessa käytetään juurikin tätä datansiirtomallia. Lisäksi JavaScriptin tukena on muita kirjas-

toja, kuten underscore.js. Underscore helpottaa palaavan datan käsittelyä ja jatkomuokkausta web-sovellukselle sopivaan muotoon. Toisaalta sen mallikielen avulla voidaan myös mallintaa web-sivulle dynaamisia näkymiä REST-rajapinnan vastedatasta.

4. SYÖTTEENTARKISTUKSEN TOTEUTUS

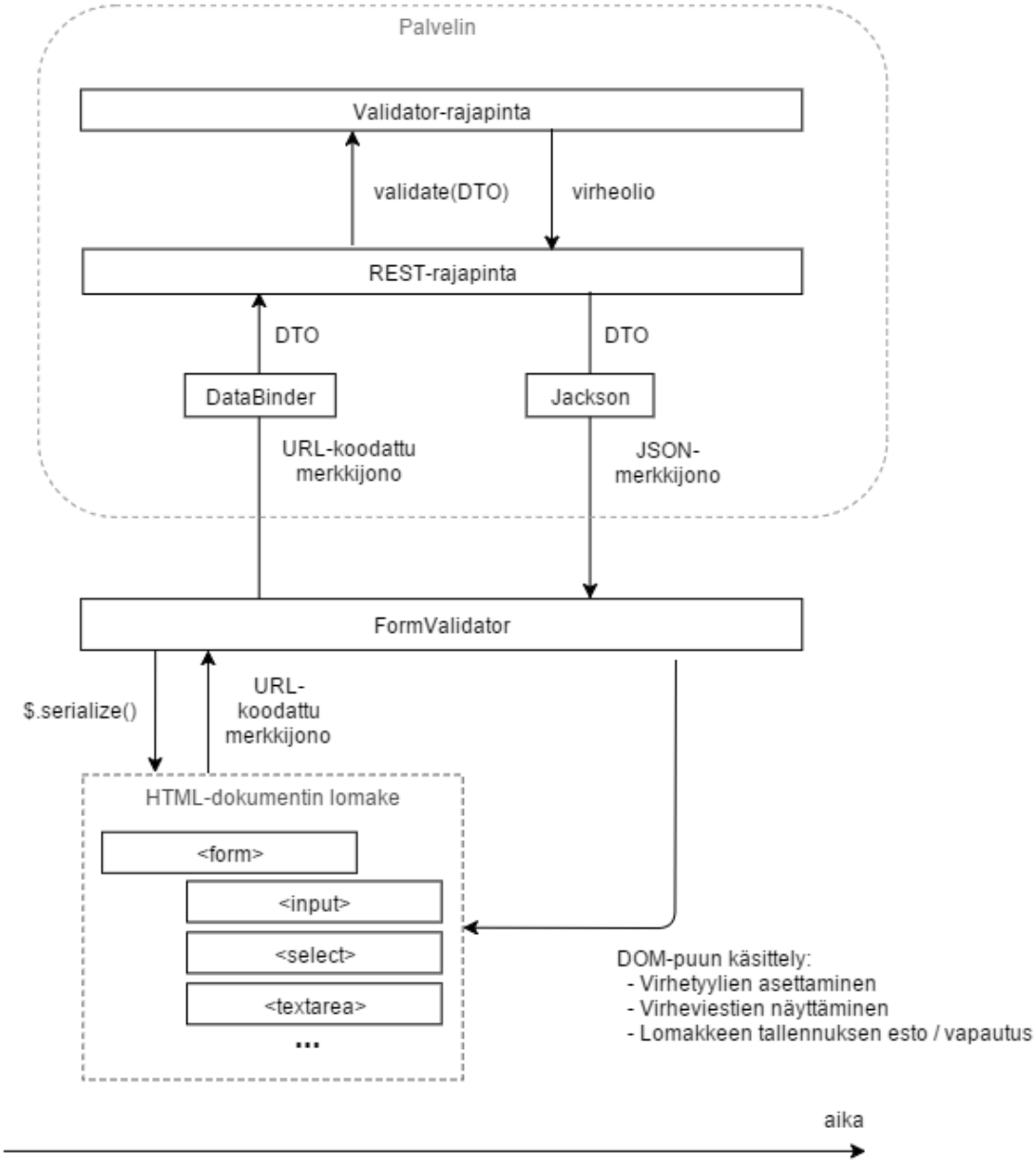
4.1 Syötteentarkistin

Syötteentarkistin on järjestelmän web-sovelluskerroksen osa, joka hallitsee validointiprosessia. Se on vastuussa esimerkiksi tarkasteltavan lomakkeen muutoksiin reagoinnista, sen seurauksena tapahtuvan muuttuneen lomakkeen validoinnista sekä virheellisten kenttien ja lomakkeen tyylien, virheviestien ja toiminnan mukauttamisesta tilanteeseen.

4.1.1 Toiminta

Syötteentarkistimen (toteuttava luokka FormValidator) perustoiminta on esitetty kuvassa 4.1. Lomakkeen muuttuminen aiheuttaa validoinnin käynnistymisen. FormValidator sarjallistaa tarkistettavana olevan lomakkeen jQueryn $.serialize() -funktion avulla URL-koodatuksi merkkijonoksi. Merkkijono lähetetään palvelimen REST-rajapinnalle, ja sen päästyä palvelimelle muutetaan se DataBinder-luokan (katso alakohta 3.1.2) avulla lomakkeen tiedot sisältäväksi Javan DTO-olioksi. REST-rajapinnan ohjainmetodi tarkistuttaa DTO:n sopivalla Validator-rajapinnan toteuttavalla validaattorilla, jossa tarkistetaan olion kentät ja niiden oikeellisuus verrattuna muuhun järjestelmän tietokantakerrokselle säilöttyyn dataan. Rajapinnalta palautuva virheolio on yksinkertainen avain-arvoparitietorakenne, joka kertoo, mitkä lomakkeen kentistä olivat virheellisiä ja miksi.

Virheoliosta koostetaan web-sovellukselle sopiva yhteenveto (esim. muunnetaan virhekoodit selväsanaisiksi ja käyttäjän kielen mukaisiksi virheviesteiksi). Yhteenveto-DTO muutetaan Jackson-kirjaston avulla JSON-merkkijonoksi, joka palautetaan selaimelle, ja edelleen FormValidator-oliolle, jatkokäsittelyyn. FormValidator merkitsee HTML-dokumentin lomakkeen virheelliset kentät CSS-luokalla `invalid`, jonka punainen väritys määritellään erikseen CSS-tyylitiedostossa. Vastaavasti FormValidator poistaa `invalid`-luokan niiltä lomakkeen elementeiltä, jotka läpäisivät tarkistuksen. Virheellisten elementtien vieressä näytetään niiden virheselitteen mukainen teksti. Lomakkeen lähetys estetään kokonaan, jos sen tiedot eivät läpäisseet validointia.



Kuva 4.1: Syötteentarkistuskoneiston toiminta ja tiedon liikkuminen eri kerroksissa

4.1.2 Käyttäminen web-sovelluksessa

Yksinkertaisin tapa alustaa syötevalidaattori lomakkeelle on esitetty esimerkissä 4.1. Rakentajaa kutsutaan JavaScript-kielelle tyypillisellä parametrioliolla, jolle on annettu pakolliset tiedot eli REST-rajapinnan validointiresurssin URL-osoite sekä lomake (form), jonka kenttiin validointi kohdistetaan. Muita mahdollisia rakennusparametreja ovat esimerkiksi toimintaan liittyvät takaisinkutsumetodit (callbacks), joiden avulla FormValidatorin toimintaa voidaan laajentaa paikallisesti.

Luokan alustaminen näin ei kuitenkaan yleensä ole järkevää, sillä rakentajasta palautuva instanssi jää helposti joko JavaScriptin globaaliin nimiavaruuteen elämään

Listaus 4.1: FormValidator-luokan kiinnittäminen lomakkeeseen

```
1 var formValidator = new FormValidator({
2     url: App.contextPath + '/api/v1/resource/validate',
3     form: 'form#lomakeId'
4 });
```

tai toisessa ääritapauksessa poistuu näkyvästä nimiavaruudesta funktiosta palattaessa (JavaScriptin nimiavaruus on funktion laajuinen). Ratkaisuna tähän on ollut toteuttaa luokalle oma jQuery-liitännäinen, jolloin FormValidator-instanssi voidaan tallettaa suoraan elementin data-attribuutiksi. Näin vältetään viitteen säilytysongelma. Lisäksi rakentajan form-parametri voidaan tässä ratkaisussa päätellä suoraan siitä elementistä, jolle liitännäistä kutsutaan. Viite FormValidator-instanssiin on myös helppo palauttaa samaisen liitännäisen kautta. Listauksen 4.2 esimerkin mukainen alustustapa onkin järjestelmässä yleisimmin käytössä oleva keino alustaa syötteentarkistus. Lähtökohtaisesti FormValidator on täysin autonominen, eli sen toimintaa ei tarvitse normaalitilanteessa perusalustuksen lisäksi ohjata.

Listaus 4.2: FormValidator-luokan kiinnittäminen lomakkeeseen

```
1 // Alustus
2 $('form#lomakeId').formValidator({
3     url: App.contextPath + '/api/v1/resource/validate'
4 });
5
6 // Instanssin viitteen saaminen
7 var formValidator = $('form#lomakeId').formValidator();
```

4.1.3 Peruslomake-elementit

HTML-standardin mukaiset lomake-elementit ovat `<select>`, `<textarea>` ja `<input>`. Select-elementti on alasvetovalikko ja textarea on vapaa, monirivinen tekstikenttä. Input-elementillä toiminta määräytyy ”type”-attribuutin avulla, jonka mahdollisia arvoja HTML5-standardissa hieman yli 20 erilaista. Tärkeimmät tyypit ovat vapaa teksti (*text*), monivalinta (*radio*), ruksivalinta (*checkbox*) sekä piilotettu (*hidden*). Lisäksi on monia automaattisesti tarkastettuja tyyppejä kuten numerosyöttö (*number*), url-osoite (*url*) tai puhelinnumero (*tel*). Tässä projektissa on käytetty näistä vain perustyyppejä, sillä selaimen suorittama automaattinen tarkistus aiheuttaisi vain päällekkäisyyttä ja visuaalisia tyyliongelmia järjestelmän oman validointikoneiston kanssa. [12] [17]

HTML-standardin mukaisesti lomake-elementeillä on ”name”-attribuutti, jonka arvo toimii avaimena palvelimelle lähetettävässä, URL-koodatussa, datamuodossa. Avainta vastaava arvo taas katsotaan lomake-elementin senhetkisestä arvosta [17]. Perustyypeillä sarjallistaminen URL-koodattuun muotoon on suoraviivaista, mutta

järjestelmässä tarvitaan validointia myös monimutkaisemmille lomake-elementeille. Esimerkki tällaisesta tarpeesta on tiedoston tallentaminen lomakkeen mukana. Toisaalta osa virheistä ei liity suoraan mihinkään tiettyyn lomake-elementtiin, sillä virhe voi johtua myös ei-hyväksyttävästä kombinaatiosta eri elementtien arvoja.

4.1.4 Moniosaiset kentät

Osa lomakkeiden validoitavista kentistä on moniosaisia, merkittävimpänä esimerkkinä päivämääräkenttä. Päivämäärä koostuu päivä-, kuukausi- ja vuosielementeistä, mutta syötteentarkistuksessa on kätevää hylätä päivämäärä kokonaisuutena. Tästä syystä validoinnin kulmakivi eli ”name”-attribuutti osoitetaan päivämääräkentät sisältävään ylempään säiliöelementtiin. Päivämääräkentän esitys järjestelmässä on havainnollistettu esimerkissä 4.3.

Listaus 4.3: Päivämääräkentän esittäminen HTML-lomakkeessa

```
1 <div class="date-fields-container" name="exampleDate">
2     <input type="text" name="exampleDate.date"/>
3     <input type="text" name="exampleDate.month"/>
4     <input type="text" name="exampleDate.year"/>
5 </div>
```

Tällä tavalla virhe voidaan kohdistaa elementin nimen perusteella joko koko päivämäärään tai mihin tahansa sen alakentistä. Esitetty pistenotaatio on välttämätön myös DataBinder-luokan toiminnan kannalta.

4.1.5 Tiedostonlisäyskentät

Tiedostojen lisääminen on oleellinen osa järjestelmän käyttöä, sillä toimialaan kuuluu paljon erilaisia sopimuksia, sopimuspohjia, valtakirjoja, laskuja ja muita dokumentteja. Järjestelmässä on toteutettu dropzone.js -kirjaston pohjalta oma, validointikoneiston kanssa yhteensopiva tiedostonlisäyskomponentti. Käyttöönotto voidaan tehdä mille tahansa sivulla olevalle säiliöelementille. Yksinkertainen tapaus on käsitelty esimerkissä 4.4.

Parametri `bindToForm` synkronoi tiedostojen lähetyksen annetun lomakkeen lähettämisen kanssa. Validoinnin kannalta oleellisin parametri on kuitenkin `includeMetaDataToForm`, jonka avulla komponentille kerrotaan mitä metatietoja tiedostoista lähetetään validoinnille. Tällaisia tietoja ovat esimerkiksi, että onko tiedostoja lisätty tiedostonlisäyskenttään ja minkänimisiä ja -kokoisia lisätyt tiedostot ovat. Itse tiedostoja ei lähetetä validoinnille suorituskykysyistä. Metatietojen perusteella tiedostokentät voidaan hylätä. Metatietoja vastaavat kentät on lisättävä myös vastaanottavaan DTO-rakenteeseen palvelimella.

Listaus 4.4: Dropzone-komponentin liittäminen HTML-elementtiin

```
<form id="exampleForm" ...>
  <div id="exampleDropzone" name="exampleFile"></div>
</form>

<script>
  $(document).ready(function() {
    $('#exampleDropzone').dropzone({
      bindToForm: '#exampleForm',
      includeMetaDataToForm: {
        includedFields: ['isEmpty', 'fileHeaders']
      }
    });
  });
</script>
```

4.2 Syötteiden lähetys

Kohdassa 4.1 käsiteltiin syötteentarkistimen käyttö, sen yleisarkkitehtuuri, tiedon kulku eri kerroksissa sekä lähetettävän tiedon mallintaminen HTML-standardin mukaisilla käyttöliittymäelementeillä. Seuraavaksi HTML-elementein mallinnettu tieto pitää sarjallistaa sekä muuttaa palvelimen käyttämän Java-kielen olioksi, jotta tiedon oikeellisuus voidaan tarkistaa palvelimella.

4.2.1 Datan sarjallistaminen

Lomakkeella lähetettävä data sarjallistetaan merkkijonoksi ja lähetetään palvelimelle käyttäen jQueryn funktioita. Funktio `$.serialize()` muuttaa lomakkeiden sisältämän tiedon avain-arvopareiksi. Merkkijono lähetetään `$.ajax()` -funktiolla REST-rajapinnalle. Koodilistauksessa 4.5 on esitetty FormValidator -luokan validoinnin lähetyskoodi. Oleellisimpina kohtina on rivit **10-16** (datan keruu) ja **21-37** (datan lähetys palvelimelle). Muu toiminnallisuus funktiossa liittyy suorituskykyparannuksiin, eli ei päästetä useaa validointipyyntöä kerralla palvelimelle. Virhekäsittely on jätetty tässä tarkastelun ulkopuolelle, sillä siihen ei ole työn tekohetkellä keksitty vielä lopullista ratkaisua.

Listaus 4.5: Tietojen lähetys palvelimelle

```
validate: function() {
  if (this._validationInProgress) {
    this._validationRequested = true;
    return;
  }

  var form = this._options.form;
  var formData = null;

  // Kerätään lähetettävä data
  if (form.is('form')) {
    formData = form.serialize();
  }
  else {
    formData = form.find('input, select, textarea').serialize();
  }

  var self = this;
  this._validationInProgress = true;

  $.ajax({
      type: 'POST',
      url: this._options.url,
      data: formData,
      // Kutsutaan pyynnön palattua palvelimelta.
      complete: function() {
        self._validationInProgress = false;
        if (self._validationRequested) {
          // Uusi validointi jonossa odottamassa
          self._validationRequested = false;
          self._doValidate();
        }
      },
      // Kutsutaan normaalissa tilanteessa
      success: function(data) {
        // Käsittelyn jatkaminen normaalisti
        self._onValidateSuccess(data);
      },
      // Kutsutaan palvelimen virhetilanteessa
      error: function() {/* virheisiin reagointi */}
  });
}
```

4.2.2 Palvelimen rajapinta

URL-koodattu merkkijono lähetetään palvelimen rajapinnalle, jossa se muutetaan automaattisesti DataBinder-luokan avulla vastaavaksi DTO-olioksi. Myös pyynnön ohjaus oikeaan ohjaimen metodiin tapahtuu Spring Frameworkin puolesta lähes automaattisesti. Ainoat tarvittavat asiat on määritellä Javan annotaatioiden avulla käsittelijälle, mihin osoitteeseen ja HTTP-metodiin se vastaa, minkä muotoista dataa se odottaa ja mihin formaattiin vastausdata koodataan. Esimerkki yksinkertaisesta validointikäsittelijästä on listauksessa 4.6. Ainut varsinainen koodirivi peruskäsittelijässä on validointivirheiden järjestely vastausolioksi rivillä **14**. Esimerkin mukaisia käsittelijämetodeja on suurin osa järjestelmän validointirajapinnoista.

Listaus 4.6: REST-ohjainmetodi

```
1  @Controller
2  @RequestMapping("/api/v1/exampleResource")
3  public class CustomerApiResource extends AbstractApiResource {
4    @ResponseBody
5    @RequestMapping(
6      value = "/validate",
7      method = RequestMethod.POST,
8      consumes = "application/x-www-form-urlencoded",
9      produces = "application/json")
10   public FormError validate(
11     @ModelAttribute @Validated CustomerDTO exampleDTO,
12     BindingResult result) {
13
14     return createValidationResult(result);
15   }
16 }
```

Esimerkkikoodissa ExampleDTO:n edessä oleva annotaatio `@Validated` etsii validation-paketista (konfiguroitava tieto Spring Frameworkissa) automaattisesti sopivaa validointiluokkaa, validoi ExampleDTO:n ja antaa tulokset BindingResult-oliossa [18]. BindingResult toteuttaa Spring Validationin Errors-rajapinnan, ja on järjestelmän kannalta käytännössä avain-arvoparikokoelma virheavaimia ja niihin liittyviä virhekoodeja. Virheavaimina käytetään lomakkeen kenttien nimiä tai, jos virhettä ei voida kohdistaa yhteen tiettyyn kenttään, niin muuta kuvaavaa merkkijonoa. Validaattoriolion lisäksi `@Validated` -annotaatio pystyy tarkistamaan yksittäisiä kenttiä yleiskäyttöisillä validaattoreilla, jotta spesifeissä validaattoreissa voidaan keskittyä vain monimutkaisempiin asioihin. Mahdollisuuksia tällaisille validaattoreille on esitetty esimerkissä 4.7. Annotaatio `@NotEmpty` tarkistaa, että kenttää vastaavan jäsenmuuttujan arvo on jokin muu kuin null. Merkkijononen ja listojen tapauksessa lisävaatimuksena on, että niiden pituus on suurempi kuin nolla. Tämän

lisäksi järjestelmässä on määritelty muutamia yleiskäyttöisiä kenttävalidaattoreita, joista esimerkissä käytetään y-tunnuksen tarkistamiseen `@BusinessIdentifier` -annotaatiota. Vastaavanlaisia omia validaattoreita on esimerkiksi päivämäärän järkevyyden tarkistukseen, puhelinnumeron muodon tarkistukseen ja lukuisiin muihin yleisesti tarvittaviin muotoseikkojen tarkistuksiin.

Listaus 4.7: Mahdollinen DTO-rakenne

```
1  public class CustomerDTO {
2
3    private Long id;
4
5    @NotEmpty
6    private String name;
7
8    @NotEmpty
9    @BusinessIdentifier
10   private String businessIdentifier;
11
12   @NotEmpty
13   private CompanyStatus status;
14
15   /* Getter- ja setter-metodit jätetty pois */
16 }
```

4.3 Lomakedatan käsittely palvelimella

Edellä käsiteltiin URL-koodatun lomakedatan muuntaminen Java-olioksi, validaattorin kutsuminen `@Validated`-annotaation avulla ja virheiden saaminen BindingResult-oliossa. Monimutkaisin osuus, eli virheiden tunnistaminen tapahtuu kuitenkin validaattoreissa, joita käsitellään seuraavana.

4.3.1 Validaattorit

Validaattorit valitaan ohjaimelle saapuville DTO-olioille pääosin automaattisesti. Listauksessa 4.6 esitetyn esimerkin tapauksessa riittää, että validaattori noudattaa samanlaista rajapintaa, joka on esitetty listauksessa 4.8.

Listaus 4.8: ExampleDTO:n validaattori

```
1 @Component
2 public class CustomerDTOValidator implements Validator {
3   @Override
4   protected void validateItem(CustomerDTO dto, Errors errors) {
5     /* Toteutus */
6   }
7 }
```

Validaattorin tarkoituksena on lisätä annetussa DTO:ssa havaitut virheet Errors-rajapinnan toteuttavalle oliolle. Tämä olio on nyt siis sama instanssi kuin aiemmin puhuttu BindingResult. Errors-olio sisältää jo tässä vaiheessa virheet, jotka on havaittu alustavan annotaatiopohjaisen DTO-validoinnin yhteydessä (esimerkki 4.7).

Jatketaan toteutuksen käsittelyä samalla asiakas-DTO -esimerkillä. Y-tunnus yksilöi yrityksen, joten samaa Y-tunnusta ei saisi olla eri asiakkailla järjestelmässä. Tarvitaan siis keino validaattorissa tarkistaa tietokantakerrokselta, onko tallennettavana oleva Y-tunnus vapaana. Listauksen 4.9 esimerkki laajentaa aiempaa listausta 4.8 kattamaan pyynnön tietokantakerrokseen.

Listaus 4.9: ExampleDTO:n validaattori

```
1  @Component
2  public class CustomerDTOValidator implements Validator {
3    @Resource
4    private CustomerRepository customerRepository;
5
6    @Override
7    protected void validateItem(CustomerDTO dto, Errors errors) {
8      boolean isBusinessIdAvailable = customerRepository.
           businessIdentifierCount(dto.getBusinessIdentifier()) == 0;
9
10     if (!isBusinessIdAvailable) {
11       errors.reject("businessIdentifier", "businessId.taken");
12     }
13   }
14 }
15
16 /* CustomerRepository.java */
17 public interface CustomerRepository extends JPARepository {
18   @Query("SELECT COUNT(*) FROM Customer c WHERE c.businessId = ?1")
19   public int businessIdentifierCount(@Param String businessId);
20 }
```

CustomerRepository on Java Persistence API:n (JPA) mukainen rajapinta, jonka avulla päästään käsiksi datan säilytyskerrokseen. Spring Framework luo rajapinnalle automaattisesti toteutuksen annettujen `@Query`-annotaatioiden perusteella. Annotaatiossa oleva kysely on HQL-kieltä (*Hibernate Query Language*), joka muistuttaa SQL:ää mutta varsinaisten tietokantarelaatioiden sijaan käyttää Hibernaten omia luokkia kyselyiden muotoiluun (katso esimerkki 3.2). Dependency Injection -suunnittelumallilla (katso kohta 3.1.1) saadaan liitettyä JPARepository-rajapinnan mukainen automaattinen toteutus CustomerDTOValidator-komponentille. Näin ollen validaattorin ei tarvitse tietää tarkkaa toteuttavaa luokkaa, vaan kyseessä on löyhä riippuvuussuhde. Rajapinnan palveluiden avulla pystytään hakemaan tietokannasta tarvittavia tietoja validointia varten. Esimerkiksi tapauksessa, jossa Y-tunnus

oli jo käytössä, lisätään tieto tästä Errors-oliolle sen reject() -metodilla.

4.3.2 Virheyhteenvedon laadinta

Validaattoreissa muodostettavan Errors-olion rajapinta ei sellaisenaan ole Jacksonilla sarjallistuva, sillä se on vain kokoelma rajapintametodeja, joilla kerättyjä virheitä voi lisätä tietorakenteeseen, ja toisaalta myöhemmin kysyä niitä sieltä. Tästä syystä ennen vastauksen palautusta selaimelle kerätään tarpeelliset tiedot omaan, helposti sarjallistuvaan rakenteeseen. Tietorakenteen pääpiirteet on koodilistauksessa 4.10.

Listaus 4.10: FormErrors-tietorakenne

```
1  @JsonSerialize(include = JsonSerialize.Inclusion.NON_EMPTY)
2  public class FormError {
3    private final Map<String, String> globalErrors;
4    private final List<FormValidationError> validationErrors;
5  }
6
7  @JsonSerialize(include = JsonSerialize.Inclusion.ALWAYS)
8  public static class FormValidationError {
9    private final String field;
10   private final String errorCode;
11   private final String errorMessage;
12   /* Rakentaja sekä getter- ja setter-metodit jätetty pois */
13 }
```

Virheet on jaoteltu kahteen eri tyyppiin. Kenttäkohtaiset validointivirheet on `validationErrors`-jäsenmuuttujassa ja koko lomaketta koskevat globaalit virheet `globalErrors`-jäsenmuuttujassa. Kenttäkohtaisista virheistä annetaan kentän nimi, virhekoodi sekä käyttäjän kielelle lokalisoitu virheviesti. Virheviestien lokalisointi onnistuu myös kätevästi riippuvuusinjektion avulla; liitetään lokalisointipalvelun toteuttava luokka tietorakenteen koostavalle ohjainkerrokselle.

4.4 Virheviestien käsittely

Kohdassa 4.3 käsiteltyjen toimenpiteiden seurauksena palvelimen vaste sille lähetettyyn dataan on valmisteltu ja virheyhteenveto on valmiina palautettavaksi selainsovellukselle. Tässä kohdassa käydään läpi virheyhteenvedon tietojen käsittely selainsovelluksessa sekä virheiden esittäminen käyttäjälle sovelluksen käyttöliittymässä.

4.4.1 Validointivastaus

FormErrors-tietorakenne sarjallistetaan Jackson-kirjaston avulla JSON-muotoon. Jos esimerkin 4.9 validaattorissa hylätään CustomerDTO varatun Y-tunnuksen vuoksi, näyttää Jacksonin tuottama vastaus esimerkin 4.11 mukaiselta.

Listaus 4.11: Virheiden JSON-rakenne

```
{
  validationErrors: [{
     field: "businessIdentifier",
     errorCode: "businessId.taken",
     errorMessage: "Y-tunnus on jo käytössä toisella asiakkaalla"
  }]
}
```

4.4.2 Virheiden käsittely

Listauksen 4.11 mukainen tietorakenne päätyy `$.ajax()`-kutsun `success()`-kuuntelijan kautta jatkokäsittelyyn `FormValidator._onValidateSuccess(data)` -metodille, jonka toteutus on esitetty koodilistauksessa 4.12. Data-parametri on virhe-JSON suoraan muutettuna vastaavaksi JavaScript-olioksi. Metodin pääpiirteet ovat siis vanhojen virhetyylien poisto (eli normaalitilan palautus), validointivastauksen mukaisten virheellisten kenttien merkitseminen, lomakkeen lähetyksen vapautus tai estäminen ja validointikuuntelijoille validoinnin tuloksesta ilmoittaminen.

Listaus 4.12: Virheiden käsittely FormValidator-luokassa

```
_onValidateSuccess: function(data) {
  var form = this._options.form;

  // Vanhojen virheellisten kenttien CSS-tyylien poistaminen
  // ja vastaavien virheviestien poistaminen käyttöliittymästä
  this._destroyOldErrorStyles();
  this._destroyErrorPopovers();

  // Uusien virheellisten kenttien CSS-tyylit ja virheviestit
  this._markErrors(data);
  this._createErrorPopovers();

  // Muuta lomakkeen tilaa (esim. estä lähettäminen jos virheitä)
  this._setupFormSubmitButtons(data);

  // Ilmoita globaalien virheiden ja yksittäisten kenttien
  // virheistä mahdollisille kuuntelijoille
  this._fireGlobalErrorHandlers(data.globalErrors);
  this._fireFieldErrorHandlers(data);

  // Ilmoita validoinnin kokonaistuloksesta mahdolliselle
  // kuuntelijalle
  if (this._options.callbacks.onValidated) {
    var hasGlobalErrors = data.globalErrors.length > 0;
    var hasValidationErrors = data.validationErrors.length > 0;
    var isValid = !hasGlobalErrors && !hasValidationErrors;
    this._options.callbacks.onValidated(isValid, data);
  }
}
```

4.4.3 Virheiden esittäminen käyttöliittymässä

Koodilistauksen 4.12 funktiosta saa yleiskuvan virheviestien käsittelystä. Poistetaan vanhentuneet virhetyylit ja -viestit, luodaan uudet, hallitaan tarkistettavan lomakkeen tilaa ja kutsutaan validointituloksista kiinnostuneita kuuntelijoita. Miten FormValidator osaa yhdistää oikeat virheet oikeisiin kenttiin? Entä miten ja missä globaalit virheviestit näytetään? Palataan lähetettävään lomakkeeseen, jonka yksinkertainen HTML-toteutus on esitetty esimerkissä 4.13.

Listaus 4.13: Asiakastietojen HTML-lomake

```
<form id="customerForm" action="/customers/1234/edit">
  <input type="hidden" name="id" value="1234"/>
  <input type="text" name="name" value=""/>
  <input type="text" name="businessIdentifier" value=""/>
  <select name="status">
    <option value=""></option>
    <option value="ACTIVE">Aktiivinen</option>
    <option value="BANKRUPT">Konkurssissa</option>
    <option value="DEBT_REORGANIZATION">Saneerauksessa</option>
    <option value="OUT_OF_BUSINESS">Lopettanut toimintansa</option>
  </select>
</form>
```

Koska lomakkeen data muutettiin lähetysvaiheessa URL-koodattuun muotoon lomake-elementtien name-attribuuttien perusteella, on myös luonnollista että FormValidator merkitsee kentät virheellisiksi saman attribuutin perusteella. FormValidatorin `_markErrors()` -metodi etsii siis halutut elementit ja merkitsee ne virheellisiksi lisäämällä niille CSS-luokan `invalid`. Huomioitavaa on myös se, että merkattava kenttä ei *välttämättä* ole itsessään lomakekomponentti, vaan voi olla myös muuntyyppinen HTML-elementti. Tästä on hyötyä esimerkiksi päivämääräkenttien validoinnissa. Lomaketta vastaavassa DTO-oliossa päivämääräkenttä on yksittäinen jäsenmuuttuja, mutta käyttöliittymässä se on eriteltävä kolmeksi syöttökentäksi (päivä, kuukausi ja vuosi). Näin ollen virhetyyli voidaan merkitä suoraan koskemaan nämä kentät sisältävää HTML-elementtiä. Virheviestien merkkauskoodi on annettu koodilistauksessa 4.14.

Listaus 4.14: Virheellisten elementtien etsiminen ja merkkaus

```
_markErrors: function(errors) {
  var self = this;

  // Underscore.js foreach-funktio.
  _.each(errors, function(error) {
    // CSS-tyylit
    var fields = self._findInputsWithFieldName(error.field);
    self._setErrorStylesToElements(fields);

    // Virheviestit talteen elementeille (tarvitaan myöhemmin)
    var errors = fields.data("invalid-field-message");
    errors = errors || [];
    errors.push(error.errorMessage);
    fields.data("invalid-field-message", errors);
  });
}
```

Pelkän virhetyylin lisäksi myös virheviestit on saatava näkymään käyttöliittymässä. Järjestelmän lomakkeet vievät vaakasuunnassa usein koko näytön verran tilaa, joten virheiden näyttäminen suoraan elementtien vieressä ei ole mahdollista. Tästä syystä kenttäkohtaiset virheviestit on toteutettu niin, että käyttäjän viedessä hiiren virheellisen elementin päälle näytetään viesti elementin viereen ilmestyvässä ponnahduselementissä (popover). Globaalit virheviestit näytetään erikseen FormValidator-luokalle kerrottavissa säiliöissä. Aloitetaan jälkimmäisellä käyttötapauksella, jossa validaattorin toimintaa tarvitsee ohjata, jotta virhe näkyisi halutussa paikassa. Listauksen 4.15 esimerkissä on esitetty FormValidator-luokan käyttö globaalin virheviestin näyttämisessä. Näin osoitettuun virhesäiliöön lisätään virheen sattuessa tietyt CSS-luokat oletustyyleineen sekä tietysti itse virheviesti.

Listaus 4.15: Globaalien virheviestien kiinnitys tiettyyn HTML-elementtiin

```
// FormValidator on alustettu jo aiemmin
var formValidator = $('#customerForm').formValidator();

// Virheviestielementin osoittaminen validaattorille
formValidator.bindContainerToError(
  "#containerElement",
  "global.error");
```

Funktion toteutus on esitetty koodilistauksessa 4.16. Helpoin keino reagoida globaaleihin virheisiin on käyttää FormValidator-luokan omia kuuntelumekanismeja. Toteutus on näin tehtynä hyvin yksinkertainen, mutta JavaScript-kielen nimiavaruuksien, jotka ovat siis laajuudeltaan aina tietyn funktion mittaisia, vuoksi ajettava koodi on palautettava sisemmän funktion (JavaScript-klosuurin) mukana. Funktio tukee yleisen tapauksen (else-haara) lisäksi myös HTML-taulukon sisällä esitettävää

globaalia virhesäiliötä (if-haara). Jäljempänä mainittua käyttötapausta käytetään, jos globaali virhe halutaan liittää juuri tiettyyn käyttöliittymän taulukkoon.

Listaus 4.16: Globaalien virheviestien kiinnitys tiettyyn HTML-elementtiin

```
1  bindContainerToError: function(container, code) {
2    // Käytetään FormValidatorin omaa ilmoituskoneistoa hyödyksi.
3    this.onGlobalError(code, (function(c) {
4      // Nimiavaruuden eristykseen tarvitaan JavaScript-klosuuri
5      // (eli sisempi funktio)
6      return function(error) {
7        c.addClass('invalid');
8
9        if (c.hasClass('table-error-wrapper')) {
10         var messageEl = $('<div class="invalid-caption"></div>');
11         messageEl.text(error);
12         c.prepend(messageEl);
13       }
14       else {
15         c.addClass('global-error-container');
16         c.text(error);
17       }
18     };
19   })($(container)); // Haetaan jQuery-säiliö klosuurille
20 }
```

Yksittäisten ponnahdusvirheviestien näyttämiseen käytetään erillistä Popover-luokkaa. Luokka on myös toteutettu osana samaa projektia, mutta sen toteutusta ei käsitellä tarkemmin tässä työssä. Samaa luokkaa käytetään laajalti myös muualla järjestelmässä. Luokka kiinnitetään tiettyyn elementtiin rakentajaoptioiden avulla. Popover-elementtien luonti virheellisille kentille on esitetty koodilistauksessa 4.17. Virheviestit talletettiin elementteihin jo merkkausvaiheessa (koodilistaus 4.14) avaimella `''invalid-field-message''`. Popover-olio osaa näyttää ja piilottaa itsensä jQueryn hiiritapahtumien, kuten ”hiiri elementin päällä” (*mouseover*) tai ”hiiri pois elementin päältä” (*mouseout*), perusteella. Virheviestejä voi myös olla monta yhtä elementtiä kohden, jolloin ne annetaan Popover-luokalle listana. Luokka hoitaa sisäisesti viestien muotoilun järkevän näköiseksi HTML-listaelementiksi.

Listaus 4.17: Globaalien virheviestien kiinnitys tiettyyn HTML-elementtiin

```
1  _createErrorPopovers: function() {
2    var form = this._options.form;
3    var targets = form.find('.invalid');
4
5    targets.each(function(i, el) {
6      $(el).popover('destroy');
7      var messages = $(el).data("invalid-field-message");
8
9      $(el).popover({
10       message: messages.length > 1 ? messages : messages[0],
11       allowHtmlInMessage: true,
12       class: 'invalid'
13     });
14   });
15 }
```

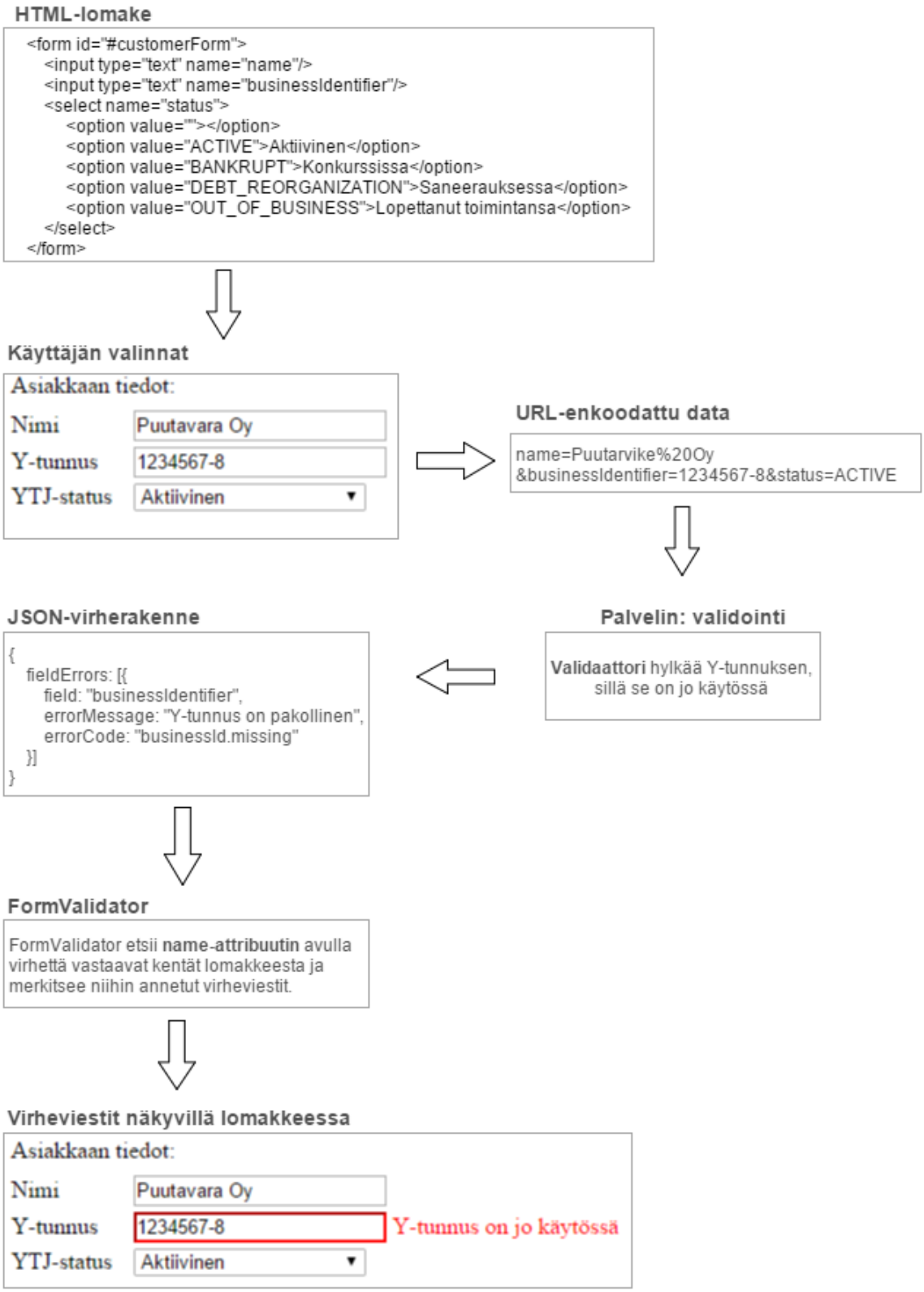
Tässä vaiheessa lomake on lähetetty, validoitu palvelimella ja virheet esitetty käyttäjälle käyttöliittymässä ja lomakkeen lähetys estetty virhetilanteissa. Koostekuvassa 4.2 on vielä kertaus koko tähänastisesta prosessista. Peruskäyttötapaus tietojen tarkistuksesta on tähän astisella toteutuksella siis katettu.

4.5 Varoitusviestit ja muut käyttötapaukset

Järjestelmän asiakasvaatimuksissa (katso kohta 2.3) oli toiminnallisena vaatimuksena aiemmin käsitellyn virhevalidoinnin lisäksi, että käyttöliittymässä on pystyttävä näyttämään myös varoituksia. Erona virheillä ja varoituksilla oli se, että varoitukset eivät estä lomakkeen tallennusta ja ne eroavat visuaalisesti varsinaisista virheistä. Virheviestit esitetään punaisella ja varoitusviestit keltaisella värillä. Toisaalta oli myös vaatimus, että lomakkeen tietyillä tiloilla osaa kentistä ei saisi muokata lainkaan. Nämä käyttötapaukset voidaan onneksi kohtuullisen helposti kattaa laajentamalla tässä luvussa esiteltyä validointiprosessia kattamaan myös muuntyyppisiä viestejä.

4.5.1 Varoitusvalidointi

Varoitusvalidointiviestien erona varsinaisiin virheisiin on oikeastaan vain keltainen väri. Toisaalta niistä on kuitenkin näytettävä lomakkeen yhteydessä kooste ja varmistettava käyttäjältä varoitusten ohitus. Ne tarvitsee siis jotenkin erottaa varsinaisista virheistä eikä niitä voida tarkistaa samoilla validointiluokilla. Tarkistus REST-rajapintaan (validointiresurssiin) asti toimii kuten virhetarkistuksessakin, mutta sen jälkeen eroja on. Laajennetaan aiempaa esimerkkiä 4.6 validointirajapintametodin



Kuva 4.2: Tiivistelmäkaavio datan keruusta, tarkistuksesta ja virheiden käsittelystä

toteutuksesta varoitusvalidoinnin tarpeisiin, jolloin saadaan esimerkin 4.18 mukainen rajapintametodi.

Listaus 4.18: REST-ohjainmetodi

```
1  @Controller
2  @RequestMapping("/api/v1/exampleResource")
3  public class CustomerApiResource extends AbstractApiResource {
4
5    @Resource
6    private CustomerDTOWarningValidator customerDTOWarningValidator;
7
8    @ResponseBody
9    @RequestMapping(
10     value = "/validate",
11     method = RequestMethod.POST,
12     consumes = "application/x-www-form-urlencoded",
13     produces = "application/json")
14   public FormError validate(
15     @ModelAttribute @Validated CustomerDTO exampleDTO,
16     BindingResult result) {
17
18     BindingResult warnings = new MapBindingResult(
19       Maps.newHashMap(),
20       "customerDTO");
21     customerContactDTOWarningValidator.validate(
22       customerContactDTO,
23       warnings);
24
25     return createValidationResult(result, warnings);
26   }
27 }
```

Muutokset validointiresurssissa. Ratkaisuna on validoida varoitukset myös Validator-rajapinnan kautta, mutta vaihtaa toteuttava luokka pelkästään varoituksiin keskittyväksi (esim. CustomerDTOWarningValidator). Aiemmin esitelty `@Validated`-annotaatio ja sen käyttö automaattiseen validaattorin kutsumiseen on jo varattu virhevalidoinnin käyttöön, joten validaattori tarvitsee lisätä REST-rajapinnan käyttöön riippuvuusinjektiolla. Itse ohjainmetodiin lisätään esimerkin 4.18 tapauksessa rivit **18-23**. Näillä riveillä oikeastaan tehdään manuaalisesti sama asia kuin mitä Spring Validationin `@Validated`-annotaatio tekee virhevalidoinnin tapauksessa automaattisesti.

Muutokset validointivastauksen tietorakenteessa. Virheviestit lisätään FormErrors-tietorakenteeseen sellaisenaan, jolloin Jacksonin läpi palautuva JSON-rakenne muuttuu listauksen 4.19 esimerkin mukaiseksi. Myös varoitukset voivat olla joko globaaleja tai kenttäkohtaisia.

Listaus 4.19: Varoitusten lisääminen validoinnin paluuviestiin

```
{
  globalErrors: [...],
  globalWarnings: [...],

  validationErrors: [...],
  validationWarnings: [...]
}
```

Muutokset validointivastauksen käsittelyssä. Käsittelyssä on otettava erilaisen visuaalisen tyylin lisäksi huomioon se konfliktitilanne, että samaan kenttään kohdistuu sekä virheitä että varoituksia. Listauksessa 4.20 on esitetty FormValidatorin `_onValidateSuccess()` -ja `_markErrors()` -metodien muokkaus tukemaan uutta virhetyyppiä.

Listaus 4.20: Virheiden käsittely FormValidator-luokassa

```
_onValidateSuccess: function(data) {
  var form = this._options.form;

  // Muutetaan tietorakennetta siten, että kaikki virheet ja
  // varoitukset ovat samassa listassa.
  // Jokaiseen listan virheobjektiin lisätään myös "type"-
  // ominaisuus (error, warning), jotta viestin tyyppiä ei
  // kadoteta.
  var errors = this._flattenValidationMessages(data);
  this._markErrors(errors);

  // Omat kuuntelijat kummallekin viestityypille, joten
  // annetaan pelkkien virheiden lisäksi myös varoitukset
  this._fireGlobalErrorHandlers({
    errors: data.globalErrors,
    warnings: data.globalWarnings
  });

  /* ... loput funktiosta muuttumatonta */
}

_markErrors: function(groups, errors) {
  var that = this;

  _.each(errors, function(error) {
    var fields = that._findInputsWithFieldName(error.field);

    // Lisätään virheviestin tyyppi (virhe tai varoitus) kutsuun
    // mukaan, jolloin saadaan CSS-tyylit eroteltua.
    that._setupElementsForErrorType(fields, error.type);

    /* ... loput funktiosta muuttumatonta */
}
```

Suurimmaksi osaksi FormValidatorin laajennus uudenlaisen virheen tukemiseksi on siis suoraviivaista. Tilanne, jossa samaan kenttään kohdistuu sekä virheitä että

varoituksia ratkaistaan siten, että varoitukset näytetään samassa ponnahdusviestissä missä virheetkin. Tämä vaatii vain parin rivin muutoksen `_createErrorPopovers()` -funktioon, mikä on esitetty listauksessa 4.21.

Listaus 4.21: Ponnahdusviestien näyttäminen käyttöliittymässä

```
1  _createErrorPopovers: function() {
2    var form = this._options.form;
3    // Kohteiksi haetaan myös varoituksen saaneet kentät
4    var targets = form.find('.invalid, .warning');
5
6    targets.each(function(i, el) {
7      /* ... muuttumaton alku ... */
8      $(el).popover({
9        message: messages.length > 1 ? messages : messages[0],
10       allowHtmlInMessage: true,
11       // Jos elementillä on 'invalid'-tyyli, niin merkitään koko
12       // ponnahdusviesti virhe-tyyppiseksi
13       class: $(el).hasClass("invalid") ? 'invalid' : 'warning'
14     });
15   });
16 }
```

Varoitusten näyttäminen ja kuittaaminen lomakkeen lähetyksen yhteydessä on melkoisen suoraviivaista FormValidatorin käytössä olevan tietorakenteen pohjalta, mutta on vaatinut kohtuullisen määrän koodia toteuttaa autonomisesti ja kaikissa lomakkeissa toimivaksi. Siitä syystä se on hyvä mainita osana validoinnin haasteita, mutta ei ole kuitenkaan tarkoituksenmukaista esitellä tässä työssä kokonaisuudessaan.

4.5.2 Vain lukutila -validointi

Kolmas käyttötapaus validaattorikoneiston hyödyntämiselle on tiettyjen kenttien asettaminen ”vain luku” -tilaan. Merkittävin ero virhe- ja varoitustarkistukseen on, että mitään ilmoitusviestejä ei tarvita lainkaan, vaan riittää, että kentät menevät lukittuun tilaan eikä niitä voi muokata. Toisaalta lomakkeen elementin asettaminen lukutilaan ei ole ainoastaan visuaalinen, vaan myös toiminnallinen muutos. Tämä aiheuttaa yllättävän paljon muutoksia validointivastauksen käsittelyyn. Muuten muutokset aikaisempaan ovat pieniä. Tietojen lähetys palvelimelle pysyy tässäkin käyttötapauksessa samana. Kuten varoitusvalidoinnissakin, tehdään uusi validaattorityyppi, esim. CustomerReadOnlyValidator merkitsemään halutut kentät. Lisäksi FormErrors -tietorakenteeseen ja siten validointivastaukseen lisätään vastaavasti kentät `globalReadOnlyFields` ja `readOnlyFields` erottamaan lukukentät varsinaista virheviesteistä.

Muutokset validointivastauksen käsittelyyn. CSS-tyyliluokilla ei voida

muuttaa lomake-elementtien toiminnallista tilaa, vaan tähän tarvitaan muita keinoja. HTML-kielessä lomake-elementin attribuutti ”disabled” muuttaa sen täysin toimintakyvyttömään tilaan. Ongelmallisena sivuvaikutuksena on, että selain ei lähetä disabled-tilaisten elementtien arvoja palvelimelle lomakkeen lähetyksessä (kuten ei validointipyynnössäkään). Toisaalta HTML5-standardi määrittelee juurikin tästä syystä myös attribuutin ”readonly”, joka estää käyttäjää muokkaamasta lomaketta vaikuttamatta kuitenkaan lomakkeen lähetykseen. Toteutuksessa on kuitenkin päädytty käyttämään vain luku -tilan toteutukseen disabled-attribuuttia, koska HTML5:n mukainen readonly-tila ei estä kenttien *ohjelmallista muokkaamista*. Tämä on ongelmallista, sillä käyttöliittymässä on paljon logiikkaa lomakkeiden automaattitäyttöä ja arvojen ehdottamista varten, eikä ole hyvä ajatus valuttaa vastuuta readonly-tilasta ulos FormValidator-luokasta. Näin ollen disabled-attribuutin ongelmat on kierretty FormValidator-luokan sisäisellä toteutuksella. Lomakkeen lähetystä varten ”disabled”-tilassa olevat tiedot on korvattava ”hidden” -tyypin `<input>` -elementeillä. Toisaalta FormValidator ei saisi vapauttaa muusta kuin validointisyistä ”disabled”-tilaan saatettuja elementtejä. Kolmantena harmina on, ettei HTML-kielessä voi asettaa disabled-tilaa kootusti, vaan jokainen lomake-elementti on merkittävä erikseen (vrt. virhevalidoinnin tilanteeseen, jossa virhetyylit pystyy asettamaan esim. päivämääräkentän kaikille elementeille merkkaamalla kenttien isäelementille CSS-tyyli `invalid`). Muokattu `_setErrorStylesToElements()` -funktio lisättyine kommentteineen on esitetty listauksessa 4.22.

Lomakkeeseen lisättyjen substituutti-elementtien poistaminen ja normaalitilan palautus tapahtuu aina validointipyynnön palatessa palvelimelta. Toimintalogiikka on sinällään täysin sama kuin virhe- ja varoitusvalidoinnissakin, joissa virhetyylit poistetaan aina ennen uuden validointitilan asetusta.

4.5.3 Ehdotetut toiminnot

Viimeisenä liittännäisvaatimuksena validaattorikomponenttiin on tiettyjen toimintojen automaattinen ehdottaminen virhetilanteiden sattuessa. Hyvänä esimerkkinä on käyttöpaikan edeltävän sopimusliitoksen päättäminen uutta liitosta kirjattaessa. Jos aiempi liitos on ollut voimassa toistaiseksi, niin annetaan tietty virhekoodi validointivastauksen yhteydessä. Sopimussivulla oleva validointikuuntelija havaitsee halutun lomakkeen tilan ja lisää käyttöliittymään napin, jolla aiempi liitos päätetään uutta liitosta edeltävään päivään. Käyttäjän suorittaessa toiminnon validointi ajetaan uudestaan ja lomake todetaan tältä osin virheettömäksi. Tämän käyttötapauksen kattaminen ei siis vaadi uutta validointikoodia edeltävään nähden, mutta on esimerkki virhespesifisten kuuntelijoiden käytöstä. Virhekuuntelijametodit ovat `onGlobalError(code, callback) {...}` sekä `onFieldError(field, code, callback) {...}`. Toteutukset ovat triviaaleja; talletetaan kuuntelijafunk-

Listaus 4.22: Read-only kenttien merkkaaminen

```
_setErrorStylesToElements: function(elements, errorType) {
  // Tarve: myös lapsielementit pitää merkitä "disabled"-
  // attribuutilla
  var withChildInputs = $(elements).add($(elements).find('input,
     select, textarea'));

  withChildInputs.each(function() {
    // Hallitaan validoinnin kautta vain sellaisia elementtejä,
    // joita ei muuta kautta ole saatettu "disabled"-tilaan.
    if (!$(this).prop("disabled") //this == muokattava elementti
    || $(this).attr("data-disabled-by-validation")) {
      var isDisabled = errorType === 'readonly';
      $(this).prop("disabled", isDisabled);
      if (isDisabled) {
        $(this).attr("data-disabled-by-validation", true);
      }
    }
  });

  // Vanha tapaus: asetetaan vain CSS-tyylit
  var markedInputs = errorType == 'readonly' ?
    withChildInputs : $(elements);
    markedInputs.addClass(this._CSS_ERROR_CLASSES[errorType]);

  // Tietojen korvaaminen, vain lukutilatapauksessa
  if (errorType == 'readonly') {
    var that = this;
    $(withChildInputs).each(function() {
    // Ohitetaan elementit, joille ei voi disabled-attribuuttia
    // asettaa
    if ($(this).is('input,select,textarea')
    && ($(this).is(':not(:checkbox):not(:radio)')
    || $(this).is(':checked'))) {
      var substitute = $('<input>');
      substitute.attr('type', 'hidden');
      substitute.attr('name', $(this).attr('name'));

      // Tunnistetaan korvattu elementti UUID:llä
      var uuid = that._UUIDv4();
      substitute.attr('id', uuid);
      substitute.val($(this).val()); // Arvon kopiointi

      // Lisääminen lomakkeeseen, uuid talteen
      $(that._options.form).append(substitute);
      $(this).data("substitute-input-id", uuid);
    }

    // Dropzone.js -tapaus: oma readonly-tilan hallinta
    if ($(this).is('.dz-boost-container')) {
      $(this).trigger('readonly', true);
    }
  });
  }
}
```

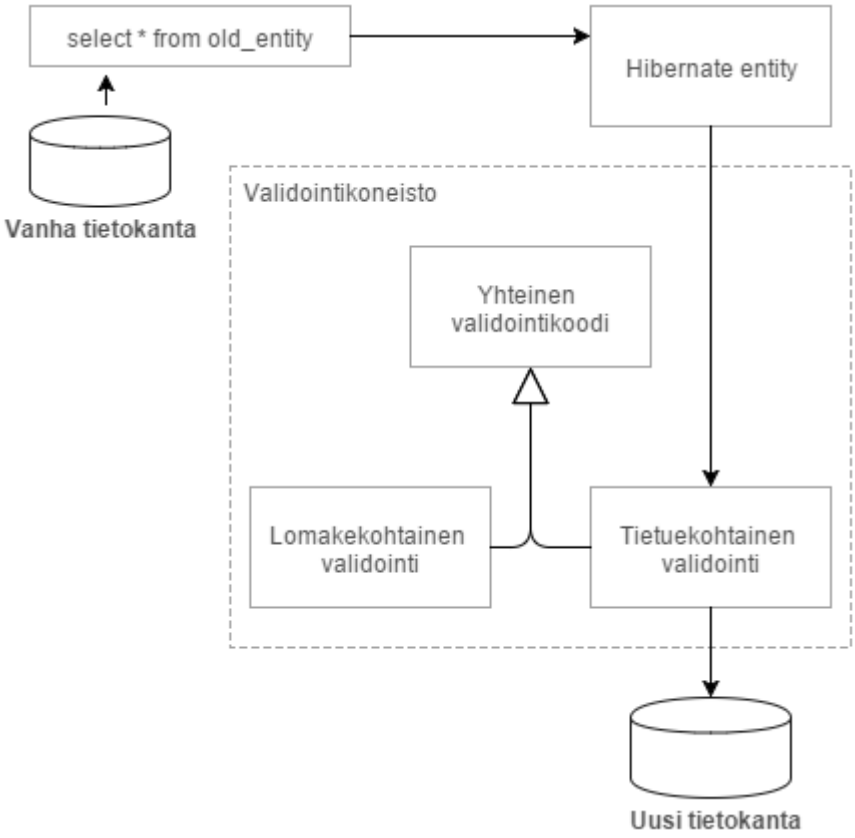
tiot vain omiin listoihinsa ja joita sitten validoinnin vastauksen yhteydessä kutsutaan `_onValidateSuccess()` -metodissa.

Ehdotettujen toimintojen toteutuksen myötä kaikki alakohdassa 2.3 esitetyt toiminnalliset asiakasvaatimukset syötteentarkistukselle on katettu.

4.6 Datamigraation huomiointi

Tähän mennessä työssä esitelty validointikoneisto huomioi nyt järjestelmään *uutena syötetyn tiedon* tarkistamisen tarpeet. Kuten asiakasvaatimusten yhteydessä kohdassa 2.4 todettiin, merkittävä osa järjestelmän tarvitsemasta tiedosta on kuitenkin peräisin vanhasta järjestelmästä. Esimerkiksi sopimuksia ja sähkönkiinnitystoimeksiantoja on vanhassa järjestelmässä tehty jo vuosiksi eteenpäin, joten vanhan datan saaminen ajetuksi järjestelmään on ehdottoman oleellista. Vanhan datan olisi myös läpäistävä uuden järjestelmän säännöstö, sillä liiketoimintalogiikka odottaa tiedon olevan sääntöjen mukaista. Johdonmukainen etenemistapa on käyttää hyväksi samoja validointiluokkia tuotavan tiedon tarkistukseen kuin syötteentarkistuskin käyttää. Tämä ei kuitenkaan suoraan ole mahdollista, sillä luvun 4 toteutuksessa tarkistettiin vain lomakkeita vastaavia DTO-olioita. Tietokantaan päätyvät Hibernaten tietueluokat (entity) ovat kuitenkin eri luokkia, joten rajapinta ei täsmää. Lisäksi monissa yksittäisissä tapauksissa tietueluokat ovat myös hieman eri muotoisia kuin niiden luomiseen käytettävä lomake-DTO, joka voi sisältää esim. metatietoja tai vain osan sitä vastaavasta tietueesta. Käyttöliittymäriippuvuuden sekä työmäärän vuoksi vanhasta järjestelmästä tuotavaa tietoa ei ole järkevää ensin luoda DTO-olioksi, sitten muuttaa tietueolioksi, ja lopulta vasta tallentaa tietokantaan. Ratkaisuna on päätetty jakaa palvelimen validointiluokat lomakekohtaisiin, tietuekohtaisiin sekä näille yhteisiin osiin. Vanhan tiedon tarkistus tapahtuu lukemalla vanhan järjestelmän tietokannasta rivi, luomalla siitä Hibernaten tietueluokka, validoimalla kyseinen tietueluokka ja tallentamalla se tietokantaan. Kuva lomakekohtaisen validoinnin ohituksesta ja datamigraation kulusta on esitetty kuvassa 4.3.

Validointikoodin jakamisessa osiin tarvitsee huomioida vain virhevalidoinnin vaatimukset, sillä varoitus- ja lukutilavalidointeja ei tarvitse suorassa tietuetallennuksessa huomioida. Virhevalidointi taas jaetaan osiin siten, että DTO- ja tietuevalidaattorit saavat yhteisen kantaluokan. Näin saatujen uusien validointiluokkien nimeämiskäytäntönä on vastaavasti *DTOValidator*, *EntityValidator* sekä näiden abstrakti kantaluokka *AbstractValidator*. Kantaluokan rajapintaa varten taas tarvitaan validoitaville DTO- ja tietueluokille yhteinen tietotyyppi. Tässä vaiheessa kohdataan ongelma, sillä sekä DTO:t että tietueluokat on jo peritty omista kantaluokistaan eikä Javassa ei ole moniperintää. Yhteinen rajapinta olisi mahdollinen toteutustapa, mutta siinäkin on ongelmansa. Validointiarkkitehtuuria sotkettaisiin näin POJO-tyylisten DTO- ja tietueluokkien sekaan hankaloittaen liiketoimintalogiikan

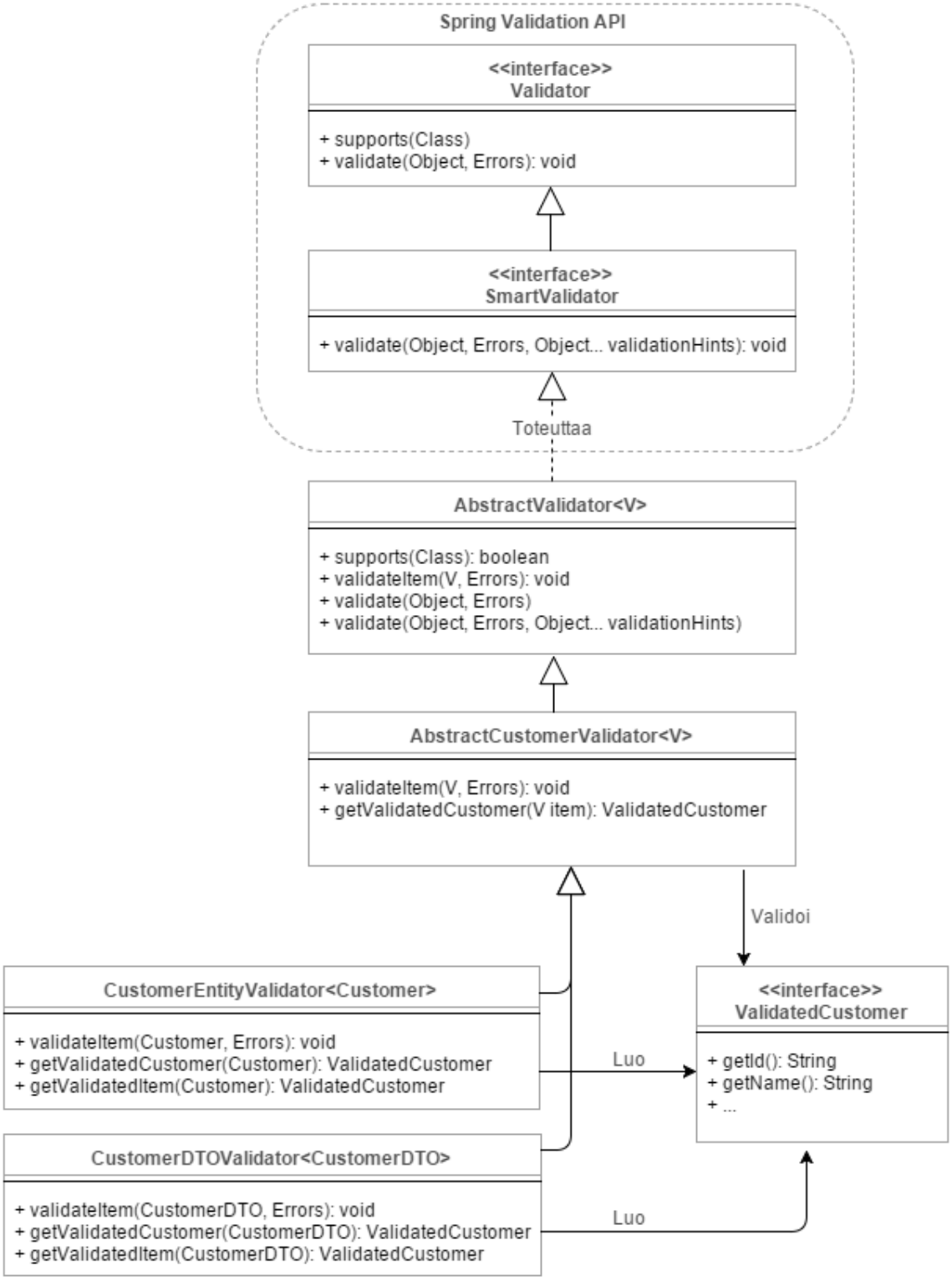


Kuva 4.3: Datamigraation toteutuslogiikka

toteutusta. Rajapinnan toteutus molempiin luokkiin olisi myös ongelmallista, sillä samannimisten metodien paluuarvona saattaa olla eri tietotyyppejä. Liiketoimintalogiikassa ja tietueluokissa käytetään esimerkiksi päivämäärien esittämiseen toiminnallisuudeltaan kattavaa *org.joda.time.LocalDate* -tietotyyppiä ja DTO-luokissa taas projektin omaa päivämäärä-DTO:ta (*DateFields* -tietotyyppi). Kolmas ja toteutettu ratkaisu on luoda sekä DTO- että tietueluokille yhteinen rajapinta, mutta eriyttää sen toteutus validaatioluokille. Tässäkin ratkaisussa on haasteensa, mutta ainakin ongelmat pysyvät validointikomponentin sisällä, eivätkä leviä koko järjestelmän laajuisiksi, kuten muissa mahdollisuuksissa.

Kuvassa 4.4 on UML-kaavio osiin jaetun validointirajapinnan arkkitehtuurista. Kuvan esimerkissä kyseessä on asiakkaan tietojen validointi, mutta kaikki järjestelmän käsitteet käyttävät samaa arkkitehtuuria.

Muutosten myötä normaalin Validator-rajapinnan sijaan AbstractValidator toteuttaa SmartValidator-rajapinnan, jonka erona normaaliin Validator-rajapintaan on, että sille voi `validationHints`-parametrissa välittää validointiin liittyvää lisätietoa. Tämä on ensijaisesti migraation suorituskykyyn liittyvä muutos, eikä sillä ole toiminnallista merkitystä useimpiin validaattoreihin. Esimerkiksi satojen tuhansien sähkönkiinnitystietojen tarkistamista nopeuttaa huomattavasti se, ettei jokaiselle



Kuva 4.4: Validaattoritoteutuksen eriyttäminen

tarvitse hakea riippuvuuksia tietokannasta yksitellen, vaan niitä voidaan nopeilla natiivi-SQL -kyselyillä hakea aina suuremmalle joukolle kerrallaan. Eräs suurimmista haasteista datamigraatiovalidoinnin suorituskyvyssä onkin se, että tietueet tarvitsee tällä arkkitehtuurilla tarkistaa yksi kerrallaan.

Tietue- ja DTO-kohtaisten validaattoreiden yhteiset validointisäännöt on abstrahoitu geneeriseen AbstractValidator<V> -kantaluokkaan. Yhteisten sääntöjen validointi tapahtuu erityisen Validated-rajapinnan kautta, josta luodaan erilliset anonyymit toteutukset EntityValidator ja DTOValidator -luokissa. Listauksessa 4.23 näytetään, miten konkreettisesta validaattoriluokasta palautetaan validoitava ValidatedCustomer -rajapinnan toteutus. Esimerkissä on näytetty vain tietuekohtaisen validaattorin toteutus, mutta DTO-kohtainen toteutus toimii täysin samalla logiikalla. Validoinnin jatkuminen geneerisestä `validateItem(V, Errors)` -metodista normaaliin Validator-rajapinnin mukaiseen `validate(Object, Errors)` -metodiin on esitetty esimerkissä 4.24.

Listaus 4.23: Esimerkki ValidatedCustomer-toteutuksen luomisesta

```
1  @Component
2  public class CustomerValidator extends  AbstractCustomerValidator<
       CustomerDTO> {
3    @Override
4    public static ValidatedCustomer getValidatedCustomer(final
         Customer validatedItem) {
5      // Palautetaan rajapinnan toteuttava anonyymi luokka.
6      return new ValidatedCustomer() {
7        @Override public Long getId() {
8            validatedItem.getId();
9        }
10       @Override public String getName() {
11           validatedItem.getName();
12       }
13       /* Loput kentät samalla tavalla */
14   }
15 }
```

Listaus 4.24: Yhteinen ValidatedItem-rajapinta esimerkkiluokalle toteutettuna

```
1  /* AbstractCustomerValidator */
2  public abstract class AbstractCustomerValidator<V> extends
       AbstractValidator<V> {
3
4      @Override
5      protected void validateItem(V item, Errors errors) {
6          validate(getValidatedCustomer(item), errors);
7      }
8  }
```

4.7 Alustava suorituskyvyn huomiointi

Toiminnallisten vaatimusten lisäksi monimutkainen syötteenvalidointi luo suorituskykyhaasteita. Pullonkaulojen löytäminen vaatii perusteellisia ajoaikamittauksia realistisella datajoukolla. Toteutuksen yhteydessä on kuitenkin tunnistettu muutamia selviä ja helppohoitoisia kipupisteitä, jotka on jo tässä vaiheessa huomioitu. Tarkastellaan alustavia toimenpiteitä suorituskyvyn parantamiseksi web-sovelluksen, palvelimen ja tietokannan näkökulmista.

Web-sovellus. Käyttäjän selaimessa toimivan sovelluksen suorituskykynäkökulma on kaksijakoinen: miten käyttöliittymä kuormittaa selainta ja toisaalta kuinka paljon käyttöliittymän aloittama validointiprosessi kuormittaa palvelinta. Kerrallaan validoitavien lomake-elementtien realistisen maksimimäärän on arvioitu olevan noin muutamia satoja kappaleita per validointipyyntö. Tällaisen määrän muokkaaminen selaimessa jQueryn avulla on melkoisen nopeaa, joten alustavassa suorituskyvyn huomioinnissa keskitytään vain palvelimen kuormaan. Helpoin tapa keventää sitä on rajata validointipyyntöjen määrää aikayksikössä. Toisaalta pyyntöjen rajaaminen venyttää validoinnin vasteaikaa. Kompromissina toteutuksessa on päädytty 1,00s viivästysaikaan, eli validointipyyntö lähetetään vasta kun viimeisimmästä lomakkeen muutostapahtumasta on kulunut 1 sekunti. Aikaraja on luonnollisesti muokattavissa, joten sitä voidaan säätää asiakaspalautteen mukaan. Eräs kuormankevennystapa sekä web-sovelluksen että palvelimen kannalta on myös tiedostonlisäyskomponentin (alakohta 4.1.5) toteutus siten, että lähetetään vain tiedostojen metatiedot validoitaviksi. Vielä ei ole näköpiirissä tarpeita tarkistaa syötettävien tiedostojen sisältöä, mutta sekin on tällä arkkitehtuurilla mahdollista toteuttaa tapauskohtaisesti myöhemmin.

Palvelin. Validointiluokissa tapahtuvan syötteentarkistuksen nopeus ei ole suoranaisesti aiemmin esitellyn validointikoneiston käsissä, mutta tietyillä käytännöillä ajoaikaa voidaan merkittävästi leikata. Helpoin keino on tunnistaa mahdollisten virheiden välisiä riippuvuuksia, jolloin tietyn virheen kohtaamisen jälkeen ei ole mielekästä jatkaa validointia *saman ominaisuuden* osalta. Esimerkiksi käyttöpaikkojen sopimusliitosten päällekkäisyyksiä on turha tarkistaa, jos liitoksen päivämäärät ovat syöttämättä tai virheellisiä. On kuitenkin huomioitava, että kaikki kullakin hetkellä *korjattavissa olevat* virheet näytettäisiin jo yhden validointipyynnön myötä. Validaattorissa ei siis saa katkaista validointia mielivaltaisesti heti ensimmäisen virheen jälkeen. Toinen merkittävä nopeutuskeino liittyy tietueiden hakemiseen Hibernaten avulla. Ajoaika nopeutuu huomattavasti, jos validoinnissa tarvittava tieto haetaan tietokannasta kerralla. Hyvin yleinen virhe ORM-kirjastoja, kuten Hibernatea, käytettäessä on hakea validaattorin riippuvuuksia useassa osassa (pahimmillaan for-silmukan sisällä), jolloin jokaisen uuden tietokantaistunnon avaaminen lisää

validoinnin ajoaikaa.

Tietokanta. Tietokannan optimointi indeksein tai denormalisoinnein perustuu kysely- eli käyttötapauskohtaiseen arviointiin, eikä siihen kannata alustavasti kiinnittää huomiota. Ainut poikkeus on taulujen pääavainindeksit, joita tarvitaan varmasti. Lähes kaikki järjestelmän taulut käyttävät pääavaimena surrogaatti-id:tä (katso esimerkki 3.2). Validaattorit, kuten muutkin palvelimen komponentit, käyttävät ensisijaisesti Long-tyyppistä pääavainta tietueiden hakemiseen Hibernaten avulla. Tästä syystä kaikille tauluille on luotu ryvästetty pääavainindeksi (clustered index). Ryvästetyssä indeksissä tietokannan rivit ovat fyysisesti levyllä indeksoidun sarakkeen mukaisessa järjestyksessä, mikä nopeuttaa kyseisen sarakkeen avulla tehtyjä hakuja [19].

5. TULOSTEN ARVIOINTI

5.1 Suorituskyky

Syötteentarkistuksen suoritusaikojen testaamiseen on helpointa käyttää suoraan projektin testipalvelinta (virtuaalikone), joka ominaisuuksiltaan on tuotantokonetta vastaava. Suoritusjat mitataan Chrome-selaimen kehitystyökaluja apuna käyttäen, jolloin saadaan jokaisesta validointikutsusta tietoon pyyntöjen ja vastausten siirtoajat sekä palvelimen suoritusaika (eli selaimen odotusaika). Näistä tiedoista lähinnä palvelinajasta ollaan kiinnostuneita, sillä järkevällä yhteysnopeudella muut ajat ovat suhteellisen pieniä. Niihin ei lisäksi voida juurikaan vaikuttaa muuten kuin palvelimen sijainnilla, joka on sekin jo valmiiksi Suomessa. Referenssiyhteytenä käytetään Soneran 100M/100M kotilaajakaistaa, jolla voitiin myös havaita siirtoaikojen merkityksettömyys muuhun suoritusaikaan verrattuna.

5.1.1 Syötteentarkistuksen suoritusaikoja

Syötteentarkistuksen ajoaikojen mittaukseen käytetään Chrome-selaimen työkaluja. Koska ajoaikaan vaikuttaa suuresti lomakkeen sisältö (esim. onko sopimuksella yksi vai viisisataa käyttöpaikkaliitosta), pyritään testauksessa löytämään sekä huonoin tapaus että keskimääräinen tapaus. Huonoimmassa tapauksessa tarkastelun alla oleva lomake suunnitellaan suorituskyvyn kannalta mahdollisimman heikoksi, mutta kuitenkin siten, että lomakkeen tila vastaa jotakin realistista käyttötilannetta. Näin saatu maksimiajoaika on siis näytejoukon huonoin aika. Keskimääräisessä tapauksessa pyritään käymään läpi suurehko joukko käsiteltävän lomakkeen tietueita ja muokkaamalla niitä monipuolisesti. Koska suoritusajat vaihtelevat paljon myös käsitteen sisällä, on taulukkoon laskettu mukaan myös keskihajonta D. Testin tulokset on esitetty taulukossa 5.1. Koska testauskriteerit ovat olleet löyhiä, on taulukon tietojen myös tarkoitus toimia lähinnä suuntaa-antavina ja suorituskyvyn kertaluokkaa edustavina lukuina.

Suoritusajat vaihtelevat paljon käsitteestä toiseen ja myös käsitteiden sisällä. Keskimääräiset suoritusajat useimmille tutkituille käsitteille ovat haarukassa 100-200ms. Käsitteiden monimutkaistuminen odotetusti lisää sekä keskimääräistä suoritusaikaa että suoritusajan keskihajontaa. Ääritapauksena on joustavat hankintasopimukset, joiden suoritusaika on erittäin riippuvainen sopimukseen liitettyjen asiak-

Lomake	Max (ms)	Keskiarvo (ms)	D (ms)	N (kpl)
Asiakkaat	248	126	35,08	40
Asiakkuussopimukset	260	101	65,32	68
Käyttöpaikat	306	139	92,26	74
Aiemmat sähkösopimukset	124	118	117,6	35
Perussähkösopimukset	1210	545	162,7	70
Joustavat sähkösopimukset	11940	2021	3232	100
Sähkönkiinnitykset	207	138	24,22	70

Taulukko 5.1: Syötteentarkistuksen suoritusaikoja tärkeimmille lomakkeille

kaiden ja käyttöpaikkojen määrästä. Yksinkertaisten joustavien sopimusten ajoajat olivat parhaimmillaan n. 600ms, mutta kymmenien asiakasliitosten sekä satojen käyttöpaikkaliitosten myötä suoritusaika venyi hurjaan 11,9 sekuntiin. Vastaavaa hajontaa, mutta pienemmässä mittakaavassa, on myös perussähkösopimuksella. Perussähkösopimuksilla liitettyjen käyttöpaikkojen määrä vaihtelee myös paljon, mutta silti hallitummin kuin joustavalla sähkösopimuksella.

5.1.2 Suorituskyvyn riittävyyden arviointi

Taulukossa 5.1 esitettyjen validoinnin suoritusaikojen perusteella osa käsitteistä on riittävän nopeita. Toisaalta esimerkiksi perussähkösopimuksen tarkistus on pahimmassa tapauksessa melkoisen hidas ja joustavan sähkösopimuksen vaikeimmat tapaukset menevät reilusti hyvksyttävän rajan yli. Melkein 12 sekunnin viive lomakkeen muokkauksessa ja validointivastauksen saamisessa on erittäin ongelmallinen, sillä kyseisessä ajassa palvelin ehtii tukkeutua lukuisista uusista validointipyynnöistä. Lomakkeentarkistuksen suorituskykyä ei siis voida kaikissa tapauksissa pitää riittävänä, vaan sen parantamiseen tarvitsee vielä keskittyä ennen käyttöönottoa.

Pitkät tarkistusajat tietyillä käsitteillä aiheuttavat luonnollisesti ongelmia myös datamigraatioon. Lähtödata on myös joidenkin käsitteiden kohdalla niin ongelmallista, että vain hyvin pieni murto-osa, joissain tapauksissa jopa alle 1%, saadaan virheettömästi tuotua työssä esitellyllä validointiarkkitehtuurilla. Migraatiodatan tuonnissa onkin jouduttu kehitettämään erilaisia käsitekohtaisia tiedonkorjaus-, ja nopeusoptimointiratkaisuja. Näistä syistä tarkkoja suoritusaikoja ei ole järkevää analysoida tarkemmin tämän työn puitteissa, sillä niiden vertailukelpoisuus suhteessa validointiarkkitehtuuriin on kyseenalainen. Eräs suorituskykyyn liittyvä optimointikeino esiteltiin kuitenkin jo kohdassa 4.6. Spring Validationin SmartValidator -rajapinta sallii osan validointitiedosta tuotavan validaattorin ja tietueen ulkopuolelta `validationHints` -parametrissa. Tällä tavalla käsitteen yhteydessä tarkistettavia riippuvuuksia voidaan hakea kerralla suuria määriä ja käyttää näin saatua tietorakennetta tietokantahakujen asemesta. Merkittäviä suorituskykyparannuksia on myös saatu varaamalla yksi tehokas palvelinkone ainoastaan migraatiota var-

ten, sekä lisäämällä kohdetietokannan välimuistin kokoa. Suorituskykyparannusten myötä yksinkertaisimmat ja muista käsitteistä riippumattomat tietueet saadaan tuotua muutamassa sekunnissa, mutta esimerkiksi käyttöpaikkojen migraatioajo kestää edelleen yli useita tunteja. Tämä johtuu siitä, että tuotavia käyttöpaikkoja on runsaasti sekä siitä, että tarkistettavia esivaatimuksia erilaisten päivämäärien, asiakkaiden, verkkoyhtiöiden ja sähkötuotteiden suhteen on runsaasti. Tätä monimutkaisempia käsitteitä ei kokonaisuudessaan saada vielä tuotua lainkaan, mutta kokonaissuoritusajat ovat oletettavasti vielä käyttöpaikka-ajoakin suurempia. Tähän viittaisi myös alakohdassa 5.1.1 kerätyt tiedot lomakevalidaattorien suoritusajoista.

5.2 Datamigraation onnistuminen

Datamigraation huomioiminen osana syötteentarkistimen arkkitehtuuria käsiteltiin kohdassa 4.6. Datamigraatio aiheutti ongelmia esimerkiksi tietue- ja DTO-tietotyyppien yhteensopivuuden suhteen, sillä validaattorin rajapinta oli suunniteltu tarkistamaan vain DTO-tyyppisiä olioita. Ongelma jouduttiin ratkaisemaan AbstractValidator-rajapinnalla sekä tietue- ja DTO-kohtaisten validaattorien eriyttämisellä. Lisäksi havaittuihin suorituskykyhaasteisiin vastaaminen aiheutti muutoksia arkkitehtuuriin. Pahimmillaan satojen tuhansien tietokantarivien (tietueiden) tuonti ja tarkistus yksi kerrallaan aiheutti vakavia suorituskykyongelmia. Ongelma onnistuttiin kuitenkin kiertämään käyttämällä SmartValidator-rajapintaa ja sen `validationHints`-parameria, jonka mukana tietuevalidaattorille voidaan välittää kaikille tietueille yhteistä esikäsiteltyä tietoa. Näin vältetään samojen tietojen hakeminen kaikille tietueille erikseen.

Ratkaisu täyttää datamigraation tarpeet ja on siten työn kannalta onnistunut. Sen avulla on pystytty tuomaan ja tarkistamaan tietoa vanhasta järjestelmästä ilman tarkistuslogiikan kahdentamista.

5.3 Ylläpidettävyys

Järjestelmän ylläpidettävyyttä voidaan karkeasti yrittää arvioida esimerkiksi seuraavin keinoin:

1. Dokumentaation määrä. Onko toteutuksesta olevassa kattava dokumentaatio? Pystyykö joku muu kuin alkuperäisen toteutustiimin jäsen tekemään muutoksia järjestelmään?

2. Oman toteutuskoodin ja kirjastokoodin määrä. Kirjastokoodin käyttö helpottaa ylläpitoa, sillä kirjastossa tai sovelluskehyksessä ilmenneet virheet yleensä korjataan jonkun muun kuin projektiryhmän toimesta. Riippuvuus kirjastokoodista aiheuttaa kuitenkin ongelman, jos kirjaston kehitys hiipuu tai loppuu

kokonaan. Toisaalta suuri oman toteutuskoodin määrä lisää riskiä piileville ohjelmointivirheille.

3. Järjestelmän testaus. Onko järjestelmässä kattavat automaattiset testit, jolloin koodimuutosten aiheuttamat regressiovirheet huomataan nopeasti?

5.3.1 Dokumentaation määrä

Järjestelmän kehitys on vielä työn kirjoitushetkellä kesken, joten valmista dokumentaatiota on vain vähän. Järjestelmän vaatimukset ovat muuttuneet ja selkeytyneet vasta projektin edetessä, joten kattavan toteutusdokumentaation ylläpitäminen ei ole ollut vielä mahdollista. Syötteentarkistuskoneiston kannalta dokumentaatio on erityisen kriittistä itse validointiluokissa, joissa tietorakenteiden tai niiden riippuvuussuhteiden muuttuminen aiheuttaa herkästi ns. dominovaikutuksen. Sen sijaan syötteentarkistuskoneiston arkkitehtuuri on lopulta melkoisen yksinkertainen ja on seikkaperäisesti selostettu mm. tässä työssä.

5.3.2 Koodin määrä

Ylläpidettävän toteutuskoodin määrää voidaan mitata yksinkertaisesti koodirivien määränä (lines of code, LOC). Tämän lisäksi hyödyllistä tietoa on esimerkiksi validointiluokkien kokonaismäärä sekä validointiluokkien tai -koodin määrä käsitettä kohden. Näillä mittareilla pystytään arvioimaan ylläpitokuormaa ja projisoimaan koodikannan kasvua lisättäessä uusia käsitteitä järjestelmään tulevaisuudessa. Koodin kokonaismäärä on jaoteltuna osa-alueittain taulukossa 5.2. Kommenttirivien osalta on sulkeisiin laskettu myös koodirivien suhde toteutuskoodiin. Ilmoitetut toiminnalliset koodirivit eivät sisällä tyhjiä tai kommenttirivejä.

Komponentti	Tiedostoja	Tyhjää	Kommentteja (%)	Koodia
Validaattorit	270	4419	397 (2,48%)	15979
FormValidator	2	105	82 (19,7%)	417
Dropzone	2	104	199 (43,6%)	456
Muu tukikoodi	5	106	141 (33,5%)	421
Testikoodi	16	683	138 (5,10%)	2707
Yhteensä	295	5417	964 (4,82%)	19987

Taulukko 5.2: Koodirivien määrät osa-alueittain

Ylivoimaisesti suurin osa toteutuskoodista on siis syötettä tarkistavissa validaattoreissa. Palvelimen ja asiakassovelluksen välinen rajapinta on hyvin ohut, vaikeasti laskettava ja siksi jätetty pois taulukosta 5.2. Asiakassovelluksen puolella koko validointijärjestelmä mahtuu hieman yli tuhanteen JavaScript -koodiriviin.

Validaattorikoodin suuri määrä selittyy osittain sillä, että yhtä validoitavaa käsitettä kohden tarvitaan melkoinen määrä luokkia. Pahimmillaan käsite vaatii kolme erillistä virhevalidaattoria, varoitusvalidaattorin, vain luku -validaattorin sekä yhteisen ValidatedItem -rajapinnan määrittelyn. Yhteensä käsitteen lisääminen siis saattaa vaatia jopa kuuden uuden validointitiedoston lisäämistä projektiin. Java on kielenä lisäksi hyvin verboosi, eli koodirivejä kertyy esimerkiksi lukuisten asetus- ja noutofunktioiden (setters, getters) vuoksi runsaasti. Suurimman osan näistä riveistä kuitenkin kirjoittaa ohjelmointiympäristön automaattiset työkalut.

Kommentit ovat osa järjestelmän dokumentaatiota ja niitä on kirjoitettu vaihtelevasti. Validaattorikoodissa kommentteja on vain 2,48% toiminnallisen koodin määrästä. Web-sovelluksen puolella kommenttirivien osuus on paljon merkittävämpi, enimmäkseen johtuen kattavista käyttöohjeista syötteentarkistuskomponentille ja sen asetusparametreille. Sisäisen toteutuksen dokumentaatiota on tästä huolimatta heikosti, kuten validaattoreissakin.

5.3.3 Valmiin kirjastokoodin tuki

Tärkeä mittari toteutuksen ylläpidettävyydelle on käytettyjen koodikirjastojen ja -kehysten tuen jatkuminen mahdollisimman pitkälle tulevaisuuteen. Toinen oleellinen asia on, että kehittäjien vaihtuessa myös uudet tekijät oppisivat nopeasti valittujen kirjastojen käytön. Tätä edesauttaa merkittävästi, että kirjastot ovat hyvin dokumentoituja ja yleisesti käytössä verkkopalveluissa. Kunkin kirjaston kohdalla arvioidaan niiden tunnettavuutta, yleisyyttä, dokumentaatiota ja kasvutrendiä. Kooste arvioinnin tuloksista on esitetty taulukossa 5.3.

Java-kieltä käytetään palvelinteknologiana W3Techs -sivuston keräämien tietojen mukaan noin 3,0% kaikista web-palveluista. Tästä huolimatta se on kolmanneksi yleisin kieli palvelinsovelluksien toteutukseen PHP:n ollessa ylivoimainen (81,4%) ja ASP.NET -kielen ollessa toisella sijalla (16,4%). Javan käyttö on lisäksi erityisen suosittua korkeiden dataliikennemäärien sivustoilla, kun taas PHP:ta käytetään laajemmin pienemmissä verkkopalveluissa. Tämä osittain selittää erittäin suuret erot prosentuaalisissa käyttömäärissä. [20] Spring MVC Framework on Javan päälle kehitetyistä sovelluskehyksistä suosituin, ja jota RebelLabs-yrityksen tuottaman raportin mukaan käytetään jopa 40% Javalla tehdyistä web-palvelinsovelluksista [21]. Käyttöasteen, Oraclen tuen ja jatkuvien versiopäivitysten myötä Spring -kehyksen ylläpidettävyyttä voidaan pitää erittäin hyvänä.

Asiakassovelluksen teknologioista oleellisin on jQuery, joka jatkaa edelleen ylivoimaisesti suosituimpana JavaScript-kirjastona. W3Techs -sivuston ylläpitämän tilaston mukaan jQueryä käyttää peräti 95,2% tutkituista web-palveluista [22]. JavaScriptin merkitys yleisesti on voimakkaasti kasvussa dynaamisten web-sovellusten yleistyessä, joten jQueryn suosio tuskin on hiipumassa kovin pian. Kirjaston elin-

voimaisuutta voidaan siis kiistatta pitää erittäin hyvänä. JavaScriptin ja jQueryn tukena on käytetty myös funktionaalista paradigmaa JavaScript-kieleen tuovaa Underscore.js -kirjastoa. Built With -sivuston mukaan Underscoren kokonaiskäyttöaste kaikista web-sivustoista on alle 0.1%, mutta 10 000 suosituimman sivuston joukossa jopa 3,3% [23]. Käyttöaste on maltillisessa kasvussa [23], jota luultavasti hieman hidastaa myös kilpailevana kirjastona pidettävä Lodash.js. Kirjaston suosio erityisesti käytetyimpien sivustojen keskuudessa, nouseva kasvutrendi ja aktiivinen kehittäjäyhteisö GitHub -versionhallintasivustolla takaavat kirjastolle hyvän ennusteen tulevaisuutta ajatellen. Viimeisenä merkittävänä kirjastona tarkastellaan tiedostojen lisäämiseen sekä lähetyksen hallinnointiin käytettävää Dropzone.js -kirjastoa. Kirjasto on melko nuori, sillä ensimmäinen versio on julkaistu GitHub-sivustolle vuonna 2012. Kirjasto ei myöskään ole kovin yleisessä käytössä jo siitäkään syystä, että se toteuttaa hyvin kapean ominaisuusspektrin. Luotettavaa tilastotietoa on siis vaikea saada, mutta kehittäjäyhteisön aktiivisuudesta GitHub-sivustolla voidaan tehdä päätelmiä. Kirjoitushetkellä yksi ainut henkilö on tuottanut 71,3% kaikista päivityksistä versionhallintaan, joten kehitys on selvästi riippuvaista kyseisestä henkilöstä. Toistaiseksi kehitys on kuitenkin jatkunut aktiivisena, joten kirjaston elinvoimaa voidaan pitää kohtalaisena. Ylläpidettävyyttä helpottaa lisäksi se, että kirjaston toiminnot on abstrahoitu järjestelmän oman rajapinnan taakse. Näin ollen kirjasto on tulevaisuudessa mahdollista korvata myös omalla toteutuksella.

Kirjasto	Arvio elinvoimaisuudesta
Spring MVC	Erittäin hyvä
jQuery	Erittäin hyvä
Underscore.js	Hyvä
Dropzone.js	Kohtalainen

Taulukko 5.3: Käytetyt toteutuskirjastot ja -kehykset ja niiden elinvoimaisuus

5.3.4 Automaattiset testit

Järjestelmän automaattinen yksikkö- ja integraatiotestaus suoritetaan JUnit-kirjaston [24] avulla. Taulukossa 5.2 esitetty testikoodin määrä kattaa ainoastaan palvelinpuolen, ts. validointiluokkien testitapaukset, sillä JavaScripti-koodin automaattiseen testaukseen ei projektissa vielä ole otettu käyttöön työkaluja. Testien koodikattavuusmittaukset on jätetty työn ulkopuolelle, sillä testattavat liiketoimintasäännöt eivät ole kaikilta osin vielä määritelty ja testien kirjoittaminen on tästä syystä lykkääntynyt. Testikoodia on kaikkiaan vasta n. kuudesosa validointikoodin määrästä, joten testikattavuus on väkisinkin heikko.

5.3.5 Ylläpidettävyyden arviointi

Järjestelmädokumentaatio on puutteellista ja vaikeuttaa siksi merkittävästi esim. henkilöstövaihdoksia projektissa. Järjestelmän kehitys on lisäksi vielä kesken ja vaatimukset ovat jossain määrin liikkuvia, joten tarkan määrittelyn kirjoittaminen on vaikeaa. Käsitteisiin liittyvät säännöt olisi kuitenkin dokumentoitava tarkasti, jotta liiketoimintalogiikka olisi ymmärrettävissä vielä projektin ylläpitovaiheessa ja mahdollisesti toisen projektitiimin toimesta.

Validointikoodin suuri määrä ja sen kompleksisuus aiheuttaa haasteen ylläpidolle. Uusien validoitavien käsitteiden lisääminen vaatii DTO- ja tietueluokkien lisäksi pahimmillaan kuuden uuden validointiluokan lisäämisen. Näiden luokkien lisäksi uusien validointiluokkien toteutus tulisi testata automaattisilla integraatiotesteillä, jolloin käsitteeseen liittyvän, ylläpitoa vaativan, koodin määrä kasvaa entisestään. Onneksi esim. taulukossa 5.2 esitetty validointikoodin määrä on liioiteltu, sillä suuri osa validointitoteutuksen koodiriveistä on ei-spesifioivaa, kuten getter- ja setter-metodeja sekä rajapintametodien esittelyitä.

Käytettyjen toteutuskirjastojen suhteen ei voida havaita merkittäviä ylläpidettävyysongelmia. Pienempien JavaScript-kirjastojen elinkaari on useimmiten lyhyt ja yhden kehittäjän varassa, mutta näihin kirjastoihin nojautumista on onnistuneesti vältetty. Ainut poikkeus on Dropzone.js -kirjaston hieman epävarma tulevaisuus, mutta riippuvuus kyseisestä kirjastosta on tarvittaessa korvattavissa toisella toteutuksella.

Syötteentarkistuksen sääntöjen automaattinen testaus on vielä heikkoa. Hyvä asia kuitenkin on, että automaattista testausta on mietitty, testejä on jo kirjotettu ja vaatimusten stabiloituessa testikattavuutta on mahdollista lisätä jo olemassa olevalla testausarkkitehtuurilla.

5.4 Jatkotoimenpiteet

Tässä luvussa löydetyt syötteentarkistuksen ylläpidettävyyteen liittyvät huolet voidaan tiivistää seuraaviin kohtiin:

1. Tiettyjen käsitteiden tarkistuksen suorituskyky on liian huono.
2. Dokumentaatiota on liian vähän. Käsitteiden säännöt ja riippuvuudet ovat muille kuin tekijöille itselleen hankalia ymmärtää ja selvittää.
3. Järjestelmän automaattinen testaus on vasta alkutekijöissään.
4. Koodikannan kasvu on hyvin voimakasta lisättäessä uusia käsitteitä järjestelmään.

Suorituskyvyn pullonkaulana käsitteiden tarkistuksessa on tällä hetkellä validaattoriluokkien ja sekä tietokannan hitaus, eikä niinkään mikään itse syötteentarkistimeen liittyvä seikka. Ratkaisuna on yrittää nopeuttaa tarkistuslogiikkaa, esimerkiksi ohittamalla tiettyjä tarkistuksia jos näihin tarkistuksiin liittyvät esiehdot ovat virheellisiä. Tässä huonona puolena kuitenkin on, että kaikkia virheitä ei välttämättä saada kerralla näkyviin. Toinen, jo projektin aikana hyväksi havaittu keino, on järjestää haut niin, että tietokannan hallintajärjestelmän puolella esikarsitaan tietoa mahdollisimman paljon, joka käytännössä tapahtuu siirtämällä tarkistuslogiikkaa monimutkaisiin kyselyihin. Tämä taas usein vaatii natiivin, käytetylle tietokannalle riippuvaisen SQL-kyselyn laadintaa. Natiivien kyselyiden käyttö edelleen nakertaa Java Persistence API:n periaatteita, eli oliokielen segregaatiota alla olevasta säilytyskerroksesta.

Toiseen kohtaan ainut ratkaisu on kirjoittaa kattavampaa määrittelydokumentaatiota. Paine käyttöönottoaikataulusta sekä vaatimusten muuttuminen on siirtänyt dokumentaation kirjoittamisaikataulua eteenpäin. Viimeistään käyttöönottovaiheen ja tilanteen rauhoittumisen myötä pitäisi dokumentoida käsitteet sekä niiden validointisäännöt tarkasti.

Vaikeiden validointisääntöjen ja järjestelmän laajentumisen myötä automaattisen testauksen merkitys vain korostuu tulevaisuudessa. Viimeistään järjestelmän ylläpitovaiheessa koodikattavuus olisi saatava järkevälle tasolle, jotta regressiovirheet pysyisivät paremmin hallinnassa. Automaattisten testien kirjoittaminen ja ajaminen olisi hyvä palvelinkoodin lisäksi laajentaa myös web-sovelluskoodin puolelle. Syötteentarkistuksen dynaamisen luonteen vuoksi JavaScript-koodilla on kuitenkin merkittävä osuus tarkistuksen kokonaisarkkitehtuurista.

Ylläpidettävän koodikannan voimakas kasvu uusien käsitteiden lisäämisen yhteydessä on tunnistettu merkittäväksi haasteeksi, mutta se on lähinnä seurausta monimutkaisesta liiketoimintalogiikasta. Monimutkaisen logiikan hallintaa taas auttaa dokumentaatio ja testaus. Tästä syystä mitään erityisiä jatkotoimia ei tarvitse koodin arkkitehtuurin osalta tehdä.

6. YHTEENVETO

Tässä työssä tutkittiin ei-triviaalin syötteentarkistuskoneiston toteutusta osana laajaa web-pohjaista järjestelmää. Tekstissä edettiin vaatimusten määrittelystä toteutusympäristön esittelyyn, syötteentarkistuksen toteutukseen ja vaatimusten täyttymisen arviointiin.

Toteutusympäristön kuvauksessa esiteltiin syötteentarkistimen kannalta keskeiset toteutuskehykset ja -kirjastot, järjestelmän kerrosrakenne, eri osien vastuualueet sekä tiedon liikkuminen eri kerrosten välillä. Palvelimen osalta keskityttiin Spring-sovelluskehyksen tarjoamiin valmiisiin validointitekniikoihin ja niiden hyödyntämiseen syötteentarkistusjärjestelmässä. Syötteentarkistimen web-sovelluspuolen käyttämät JavaScript-kirjastot myös esiteltiin.

Syötteentarkistuksen toteutuksessa esiteltiin ensin yksinkertainen perustapaus validoinnista, jota sen jälkeen laajentannettiin monimutkaisempiin käyttötapauksiin, kuten varoitusvalidointiin ja vain lukutila -validointiin. Validointiin epäsuorasti liittyvän, syötteen korjaus- ja ehdotuslogiikan lomakekohtainen integrointi syötteentarkistimeen esiteltiin pikaisesti. Lopuksi laajennettiin myös palvelinpuolen validointiarkkitehtuuria kattamaan datamigraation vaatimukset. Tämän myötä asiakasvaatimuksena esitellyn vanhan järjestelmän datamigraation toteuttaminen on mahdollista. Datamigraatio on vaatinut paljon ylimääräistä käsitekohtaista koodia pelkän tarkistuskoneiston lisäksi, joten suurin osa datamigraation toteutuksesta on jätetty työn ulkopuolelle. Ratkaisun onnistumisen arvioinnissa otettiin kantaa validoinnin suorituskykyyn sekä ratkaisun ylläpitoon liittyviin haasteisiin. Datamigraation haasteet otettiin tarkasteluun mukaan erityisesti suorituskyvyn osalta, sillä se käyttää samaa validointikoodia kuin mitä uudenkin tiedon lisääminen.

Tässä vaiheessa on vielä mahdotonta tarkalleen sanoa, kuinka hyvin työssä esitelty ratkaisu toteuttaa kaikki suorituskyky- ja ylläpidettävyysvaatimukset ja selviää niiden asettamista haasteista. Ratkaisun parantamiseksi on kuitenkin tulosten arvioinnissa onnistuttu esittämään lukuisia ehdotuksia, joten ongelmat vaikuttavat korjattavissa olevilta. Lisäksi vakavimmat ongelmat, kuten datamigraatiossa tuotavien tietueiden huono sopivuus toteutettuihin validointisääntöihin, johtuvat ensi sijassa muista kuin syötteentarkistimen toteutukseen liittyvistä seikoista. Ratkaisua voidaan näin ollen pitää kokonaisuudessaan onnistuneena, vaikka kaikkia ongelmia ei diplomityön kirjoituksen aikana saatukaan vielä ratkaistua.

LÄHTEET

[1] Walls, G. 2011. Spring in Action, 3rd edition. Wiley India Pvt. Limited. 424pp.

[2] Dhananjay, N. A beginners guide to Dependency Injection. WWW. Luettu 17.12.2015. Saatavilla: http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection

[3] Validation, Data Binding, and Type Conversion. WWW. Luettu 17.12.2015. Saatavilla: http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/validation.html

[4] Bauer, C., King, G. Java Persistence With Hibernate, Revised Edition. 2007. Manning Publications New York. 823pp.

[5] Open Source Persistence Frameworks in Java. WWW. Luettu 10.12.2015. Saatavilla: http://java-source.net/open-source/persistence

[6] Hoehrmann, B. The application/www-form-urlencoded format. WWW. Luettu: 10.12.2015. Saatavilla: https://tools.ietf.org/html/draft-hoehrmann-urlencoded-01

[7] The JSON Data Interchange Format. 2013. WWW. Luettu 17.12.2015. Saatavilla: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[8] Jackson JSON Processor Wiki. WWW. Luettu 17.12.2015. Saatavilla: http://wiki.fasterxml.com/JacksonHome

[9] Richardson, L., Ruby, S. 2007. RESTful Web Services. O'Reilly Media Sebastopol. 407pp.

[10] Bergsten, H. JavaServer Pages, 3rd Edition. 2004. O'Reilly & Associates Sebastobol. 722pp.

[11] JSP Standard Tag Library. WWW. Luettu 10.12.2015. Saatavilla: https://jstl.java.net/

[12] Lawson, B., Sharp, R. Introducing HTML5. 2011. New Riders Berkeley. 216pp.

[13] jQuery. WWW. Luettu 10.12.2015. Saatavilla: https://jquery.com/

[14] Underscore.js. WWW. Luettu 10.12.2015. Saatavilla: http://underscorejs.org/

[15] Dropzone.js. WWW. Luettu 10.12.2015. Saatavilla: http://www.dropzonejs.com/

[16] What is the Document Object Model. WWW. Luettu 22.12.2015. Saatavilla: http://www.w3.org/TR/DOM-Level-2-Core/introduction.html

[17] HTML5 input element documentation. WWW. Luettu 17.12.2015. Saatavilla: https://developer.mozilla.org/en/docs/Web/HTML/Element/Input

[18] BindingResult documentation. WWW. Luettu 10.12.2015 Saatavilla: http://docs.spring.io/spring-framework/docs/2.5.6/api/org/springframework/validation/BindingResult.html

[19] Clustered and Nonclustered Indexes Described. WWW. Luettu 17.12.2015. Saatavilla: https://msdn.microsoft.com/en-us/library/ms190457.aspx

[20] Usage of server-side programming languages for websites. WWW. Luettu 14.10.2015. Saatavilla: http://w3techs.com/technologies/overview/programming_language/all

[21] Top 4 Java Frameworks Revealed. Real Life Usage Data of Spring MVC, Vaadin, GWT and JSF. WWW. Luettu 17.12.2015. Saatavilla: http://zeroturnaround.com/rebellabs/top-4-java-web-frameworks-revealed-real-life-usage-data-of-spring-mvc-vaadin-gwt-and-jsf/

[22] Usage statistics and market share of JQuery for websites. WWW. Luettu 14.10.2015. Saatavilla: http://w3techs.com/technologies/details/js-jquery/all/all

[23] Underscore.js Usage Statistics. WWW. Luettu 14.10.2015. Saatavilla: https://trends.builtwith.com/javascript/Underscore.js/

[24] What is Junit Test Framework? WWW. Luettu 17.12.2015. Saatavilla: http://www.tutorialspoint.com/junit/junit_test_framework.htm