Haaga-Helia
ammattikorkeakoulu Oy

Palvelinsovelluksen uudistaminen refaktoroinnilla ja rajapinnalla

Aida Luuppala

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2020

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

13.12.2020

Tekijä(t)
Aida Luuppala

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Palvelinsovelluksen uudistaminen refaktoroinnilla ja rajapinnalla

Sivu- ja liitesivumäärä
24

Opinnäytetyön otsikko englanniksi
Server application renewal with refactoring and API

Opinnäyteyö oli toiminnallinen opinnäytetyö, jonka tarkoituksena oli tehdä konseptitodistus työnohjaus-palvelinsovelluksen uudistuksesta. Geometrix Oy tilasi opinnäytetyön tarpeesta uudelle työnohjauksen käyttöliittymälle, joka vaati myös toimenpiteitä palvelinsovellukseen.

Opinnäytetyössä käsiteltiin palvelinsovelluksen uudistamista refaktoroinnin eli koodin sisäisen rakenteen parantamisen sekä rajapintasuunnittelun teoreettisessa viitekehityksessä.

Opinnäytetyössä halusin selvittää mikä oli paras lähestymistapa palvelinsovelluksen uudistamiseen ilman, että olemassa olevaa projektia olisi pitänyt tehdä uudelleen. Opinnäytetyöstä tuli raportti siitä, miten palvelinsovelluksen kehitystä kannattaisi jatkaa. Opinnäytetyössä havainnollistettiin mitä ensimmäisten uudistusten kohdalla oli tehty teknisesti.

Opinnäytetyö tarjosi hyvän esimerkin palvelinsovelluksen uudistusprojektista ja yhden lähestymistavan siihen, kuinka uudistus voitiin toteuttaa. Monissa organisaatioissa on vanhoja järjestelmiä, joiden tulee säilyä ehjinä, mutta vastaanottaa myös muutoksia ja uutta kehitystä. Opinnäyteyöni tarjosi lähestymistavan siihen, kuinka voidaan säilyttää vanha toiminnallisuus, mutta integroida uusia muutoksia ilman vanhan koodin rikkoutumista ja uudelleenkirjoittamista.

Opinnäytetyön tuloksena syntyivät työohjauksen uudistuksen ensimmäisen version rajapintaresurssit sekä konseptitodistus uudistustyön jatkamiselle.

Asiasanat
rajapinta, rest, refaktorointi, ohjelmistokehitys, java, backend, proof of concept, konseptitodistus

Sisällys

Käsitteet

POC	(Proof of Concept) Konseptitodistus
Refaktorointi	(Refactoring) Prosessi koodin sisäisen rakenteen parantamiseen
API	(Application Programming Interface) Rajapinta
REST	(Representational State Transfer) Rajapinnoissa käytetty arkkitehtuurityyli, joka hyödyntää HTTP:tä
HTTP	(Hyper Text Transfer Protocol) Protokolla verkossa tapahtuvaan tiedonsiirtoon
Client	Verkkoselain tai sovellus, joka lähettää HTTP-pyyntöjä (HTTP Request)
Server	Palvelin tai palvelinsovellus, joka vastaanottaa HTTP-pyyntöjä ja palauttaa HTTP-vastauksen (HTTP Response)
URI	(Uniform Resource Identifier) Rajapintaresurssin tunniste
Servlet	Pieni Java-ohjelma web-palvelimella, joka vastaanottaa pyyntöjä ja palauttaa vastauksen
Controller	Java luokka, joka hyödyntää servlet:iä ja kontrolloi sovelluksen tietovirtaa
JSP	(Java Server Pages) Java web-sovelluksen dynaaminen näkymä
MVC	(Model, View, Controller) Suunnittelumalli, joka erottaa liiketoimintalogiikan, esityslogiikan ja datan
Service	Java-luokka johon kootaan liiketoimintalogiikkaa
Liiketoimintalogiikka	Logiikka, joka muuntaa ohjelmointikielelle yrityksen toimintalogiikkaa ja määrää kuinka tietoa haetaan tai päivitetään
DTO	(Data Transfer Object) Tiedonsiirto-objekti
DAO	(Data Access Object) Tietokantaoperaatioita suorittava objekti
Legacy	Nimitys vanhentuneesta koodista tai ohjelmistosta
Repository	Arkisto tai säilytyspaikka projektille versionhallinnassa
Branch	Kehityshaara projektista versionhallinnassa
Master	Päähara projektista versionhallinnassa

1 Johdanto

Opinnäytetyöni on toiminnallinen opinnäytetyö, jossa aloitetaan Geometrix Oy:n työnohjaus-palvelinsovelluksen uudistusprojekti. Geometrix Oy tilasi tämän opinnäytetyön, tarpeesta työnohjauksen uudelle käyttöliittymälle, joka vaatii myös toimenpiteitä palvelinsovellukseen.

Opinnäytetyön tilaajana on suomalainen ohjelmistoyritys Geometrix Oy. Geometrix Oy on perustettu vuonna 2002 ja sen toimitusjohtajana sekä omistajana toimii Olli Alanko. Yritys sijaitsee Pitäjänmäessä Helsingissä ja henkilöstöä yrityksellä on 18. (Suomen asiakastieto Oy. Asiakastietorekisterin taloustiedot.) Geometrix Oy on erikoistunut paikkatietoon ja sen soveltamiseen liikkuvan työn ohjaamisessa. Geometrix Oy tekee ohjelmistoratkaisuja, jotka tehostavat yritysten toimintaa mobiilin karttapohjaisen lähestymistavan avulla. (Geometrix 2020. Meistä.)

Työnohjaus on sovellus, jota työnjohtajat käyttävät tehtävien ja resurssien hallinnoimiseen. Sovelluksella voidaan seurata töiden tilaa ja etenemistä reaaliaikaisesti. Työnjohtajat voivat osoittaa tehtäviä työntekijöille ja työntekijät löytävät kohteet kentällä karttapohjaisen mobiilikäyttöliittymän avulla. Työt voi myös merkata tehdyiksi paikan päällä ja tehtäviin voidaan lisätä huomioita ja valokuvia kohteesta. (Geometrix 2020, Työnohjaus.)

Opinnäytetyöni tavoitteena on uudistaa työnohjaus-palvelinsovelluksen arkkitehtuuria palvelemaan uutta käyttöliittymää. Tämä tapahtuu koodin rakenteen parantamisella ja uusien rajapintaresurssien toteuttamisella. Palvelinsovellusta käyttävät hyväkseen olemassa oleva vanha käyttöliittymä, mobiilisovellus sekä uusi käyttöliittymä. Tarkoituksena on vähentää duplikaattikoodia ja saada mobiili- sekä selainsovellukset käyttämään samoja rajapintaresursseja ja liiketoimintalogiikoita.

Kehitysprosessia tukevat työryhmän kanssa käydyt palaverit ja suunnitelmat. Opinnäytetyöni tuloksena ovat ensimmäiset rajapintaresurssit tehtävien ja resurssien hakemiseen sekä tehtävien osoittamiseen resurssille. Työnohjaus on iso sovellus, joka pitää sisällään paljon toiminnallisuuksia, joten kehitystyö jatkuu vielä opinnäytetyöni jälkeen. Opinnäytetyöni on startti ja konseptitodistus palvelinsovelluksen uudistukselle, jonka pohjalta kehitystyötä on hyvä jatkaa.

Opinnäytetyössä ei refaktoroida koko sovellusta. Opinnäytetyössä ei myöskään tehdä kaikille tarvittaville pyynnöille rajapintaresursseja, vaan keskitytään ensimmäisten rajapintaresurssien luomiseen ja niitä koskevan liiketoimintalogiikan eristämiseen omaan palvelu-

luokkaan. Opinnäytetyössä ei myöskään puututa olemassa oleviin DAO- ja manageriluokkiin tai koodin suoritustehoon.

Opinnäytetyön tietoperustaksi on valittu refaktorointi eli koodin sisäisen rakenteen parantaminen sekä rajapinnat. Refaktorointi on tärkeä kokonaisuus liiketoimintalogiikan siirron kannalta. Rajapinnat ovat tärkeä kokonaisuus uusien rajapintaresurssien suunnittelun ja luonnin kannalta. Opinnäytetyöni tarjoaa lähestymistavan siihen, kuinka mahdollisimman tehokkaasti saataisiin olemassa oleva ohjelmisto noudattamaan uutta joustavampaa ja tehokkaampaa sovellusarkkitehtuuria ilman koodin uudelleenkirjoittamista.

2 Koodin sisäisen rakenteen parantaminen refaktoroinnilla

Refaktorointi on prosessi, jonka tavoitteena on muuttaa ohjelmistojärjestelmää siten, ettei muuteta koodin ulkoista toimintaa, mutta parannetaan koodin sisäistä rakennetta. Refaktorointi ei siis näy loppukäyttäjälle, mutta parantaa ohjelmiston rakennetta ja sitä kautta kehitettävyyttä. Refaktorointi on kontrolloitu tapa siivota koodia ja pienentää bugien syntymismahdollisuutta tulevaisuudessa koodia muutettaessa, ylläpidettäessä tai uusia ominaisuuksia lisättäessä. Pohjimmiltaan refaktorointi on koodin suunnittelun parantamista sen jälkeen, kun koodi on alun perin kirjoitettu. (Fowler 2019, xiv.)

Ohjelmistokehityksen historiassa on suurimmaksi osaksi uskottu, että ensin suunnitellaan ja vasta sitten koodataan. Ajan myötä koodia muokataan ja järjestelmän eheys eli suunnitelman mukainen struktuuri vähitellen haalistuu, jonka seurauksena ohjelmistokehitys muuttuu enemmänkin kehittämisestä hakkeroinniksi. (Fowler 2019, xiv.) Usein ammattislangissa puhutaankin vanhojen tai olemassa olevien järjestelmien “puukottamisesta”, mikä hyvin kuvaa refaktoroimattoman legacy-koodin kehittämistä.

Refaktorointi tarjoaa päinvastaisen lähestymistavan vanhaan kaavaan. Refaktoroinnilla voidaan parantaa huonoa tai jopa kaoottista mallia ja työstää se uudelleen hyvin struktuuroiduksi koodiksi. Refaktoroinnin kaikki vaiheet ovat hyvin yksinkertaisia, mutta silti tehokkaita. Refaktorointi koostuu usein pienistä toimenpiteistä, kuten esimerkiksi jonkun kentän siirtämisestä toiseen luokkaan, koodiosion erottamisesta omaksi metodikseen tai koodin hierarkian muuttamisesta. Mutta näiden pienten muutosten kumulatiivinen vaikutus voi radikaalisti parantaa ohjelmiston struktuuria. (Fowler 2019, xiv.)

Refaktoroinnilla työn tasapaino muuttuu. Ohjelmistosuunnittelu tapahtuu tällöin jatkuvasti kehittäessä, toisin kuin että se tapahtuisi vain kehitystyötä aloittaessa. Kun järjestelmää rakennetaan jatkuvasti, opitaan myös parantamaan sen arkkitehtuuria. Tämän vuorovaikutuksen lopputulos on ohjelmisto, jonka arkkitehtuuri pysyy eheänä kehityksen jatkuessa. (Fowler 2019, xv.)

Jos satojen rivien koodiin pitäisi lähteä tekemään muutoksia, on muutosten tekeminen helpompaa, jos koodi on strukturoitu funktioiksi ja muiksi elementeiksi, jotka edesauttavat koodin ymmärtämistä. Jos ohjelmistosta puuttuu struktuuri, on helpompaa aloittaa kehitystyö lisäämällä struktuuria ja vasta sitten tehdä muutoksia. Jos on tarve lisätä uusia toiminnallisuuksia vanhaan koodiin, on hyvä aloittaa kehitysprosessi refaktoroimalla vanha koodi rakenteeltaan selkeäksi ja ymmärrettäväksi, ja vasta sitten lisätä uusi toiminnallisuus. (Fowler 2019, 4.)

Refaktorointi on tarpeellista kuitenkin vain silloin, kun koodi on jatkuvasti ylläpidettävää koodia. Jos koodi toimii ja siihen ei ole tarvetta lisätä mitään uusia toiminnallisuuksia, refaktotoinnista ei ole paljon hyötyä ja siksi sovellus on hyvä jättää toimimaan sellaisenaan. Toisin sanoen, jos kenenkään ei tarvitse kehittää sovellusta eli ymmärtää sovelluksen koodia, niin silloin legacy-koodista ei ole mitään haittaa. Mutta heti, kun jonkun on ymmärrettävä, miten koodi toimii ja kokee sen vaikeasti luettavaksi, on hyvä tehdä asialle jotain. (Fowler 2019, 5.)

2.1 Testaus refaktoroinnissa

Refaktorointiprosessia aloitettaessa on ensi varmistettava, että refaktoroitavalle osiolle on olemassa karttavat testit. Testit ovat olennaisia, sillä refatoroinnissa voi helposti syntyä virheitä (bugeja). Testit tarkistavat, ettei mikään mene rikki refaktorointiprosessissa. Varsinkin mitä suurempi sovellus, sitä todennäköisempää on, että joku saattaa mennä rikki refaktorointiprosessissa. (Fowler 2019, 5.)

2.2 Refaktorointimenetelmiä

Refaktorointimenetelmiä on lukuisia. Listasin tähän sellaisia, jotka ovat hyödyllisiä opinnäytetyöni kannalta.

2.2.1 Suuren funktion purkaminen pienempiin osiin

Siihen, milloin koodi kannattaa eristää omaksi funktiokseen, on paljon erilaisia koulukuntia ja lähestymistapoja. Yksi lähestymistapa on pituus. Esimerkiksi, jos funktio on niin suuri, ettei se mahdu yhdelle näytölle, voi olla hyvä eristää funktio pienempiin osiin. Toinen on koodin uudelleenkäytettävyys. Koodi, jota käytetään useammin kuin kerran pitäisi laittaa omaan funktioon, mutta vain kerran käytettävä koodi tulisi jättää olemaan. Fowlerin mukaan järkevintä on eristää koodi omaan funktioon silloin, kun koodinpätkän ymmärtämiseen joutuu käyttämään paljon aikaa. Eristettäessä on hyvä nimetä funktio sen perusteella “mitä” se tekee. Tällöin koodipätkän tarkoitus, tulee selväksi jo heti funktion nimestä, eikä funktion sisältämää koodia tarvitse välttämättä lukea saadakseen käsityksen funktion toiminnasta. (Fowler 2019, 106–107.)

Kun pitkää funktiota lähdetään hajottamaan osiin, on hyvä yrittää tunnistaa pisteitä, jotka erottavat koodin yleisen käyttäytymisen eri osat (Fowler 2019, 6). Fowler puhuu myös ymmärryksen siirtämisestä koodiin. Kun pitkää funktiota lukee ja erottaa sen eri osat mie-

lessään ja tekee muutokset, niin samalla siirretään ymmärrystä takaisin koodiin. (Fowler 2019, 7.)

Ensimmäiseksi on huomioitava kaikki muuttujat, jotka eivät ole enää samassa laajuudessa (scope). Muuttujat, joita käytetään uudessa pienemmässä funktiossa, mutta niitä ei muuteta, voidaan vain välittää uudelle funktiolle parametreina. Muutettavat muuttujat vaativat enemmän huolenpitoa. Jos tällaisia muuttujia on vain yksi, niin se voidaan palauttaa sellaisenaan, mutta useammat muuttujat voidaan asettaa erilaisiin tietorakenteisiin, kuten listaan tai objektiin. (Fowler 2019, 7.)

2.2.2 Funktion nimen muuttaminen

Hyvä nimi auttaa ymmärtämään mitä funktio tekee. Jos funktio on hyvin nimetty sen perusteella ”mitä” se tekee, voidaan jo nimestä päätellä funktion toiminta ilman, että tarvitsee välttämättä katsoa itse funktion koodia. (Fowler 2019, 124.)

2.2.3 Muuttujan uudelleennimeäminen

Kuten muissakin ohjelmiston elementeissä, nimen tärkeys on suhteessa siihen, kuinka laajasti muuttujaa on käytetty. Jos muuttujaa käytetään esimerkiksi vain yhdessä lambda-funktiossa, sen uudelleennimeäminen on tuskin tarpeen. Muuttujien nimissä on hyvä käyttää kokonaisia sanoja, jotka kuvaavat mitä tietoa muuttuja sisältää. Esimerkiksi laajalti käytetyt muuttujat, joiden nimi on vaikka vain yksi kirjain, on hyvä nimetä tarkemmin. Muuttujia uudelleennimetessä on etsittävä kaikki viittaukset muuttujaan ja muuttaa ne kaikki. Koodieditorit tarjoavat tähän hyviä työkaluja. (Fowler 2019, 137.)

2.2.4 Parametriobjektin luominen

Usein yhdessä funktiosta toiseen liikkuva joukko muuttujia on hyvä korvata yksinkertaisella tietorakenteella. Tämä vähentää parametrilistan kokoa funktioissa, joissa käytetään uutta tietorakennetta. (Fowler 2019, 140.)

2.2.5 Funktioiden yhdistäminen luokkaan

Luokat ovat perustekijä useimmissa nykyaikaisissa ohjelmointikielissä. Luokat sitovat muuttujia ja funktioita yhteiseen ympäristöön. Luokat ovat ensisijainen rakenne objektiorientoituneessa ohjelmoinnissa, mutta hyödyllisiä muissakin lähestymistavoissa. Hyvin lähekkäin toimivat funktiot, jotka käyttävät samoja muuttujia, voi olla hyvä siirtää omaan

luokkaan. Luokan sisällä on helppo tehdä funktiokutsuja ilman paljon argumentteja ja luokan voi monistaa käytettäväksi ohjelman muissa osissa. (Fowler 2019, 144.)

2.2.6 Primitiivisen tietotyypin korvaaminen objektilla

Tietotyypit, joiden käsittelyyn liittyy paljon logiikkaa, on hyvä korvata omalla objektilla. Tällöin, kun näitä tietotyyppejä käsitellään ympäri ohjelmaa, niin ei synny duplikaattikoodia, vaan kaikki tietyn tietokentän muokkaamiseen liittyvät operaatiot ovat yhden objektin sisällä. Tällaisia tietotyyppejä voivat olla esimerkiksi päivämäärät, puhelinnumerot ja rahasummat yms. (Fowler 2019, 174.)

2.2.7 Koodin siirtäminen funktioon

Duplikaattikoodin poistaminen on yksi parhaista toimenpiteistä hyvän koodipohjan ylläpitämiseen. Jos jotain koodinpätkää kutsutaan joka kerta, kun jotain tiettyä funktiota kutsutaan, kannattaa koodinpätkä sisältää funktioon mukaan. Näin kaikki tulevaisuuden muutokset toistuvaan koodiin, voidaan tehdä vain yhteen paikkaan. (Fowler 2019, 213.)

2.2.8 Kuolleen koodin poistaminen

Hyvät kääntäjät poistavat käyttämättömän koodin ja käyttämätön koodi ei vie paljon tilaa tuotannossa, mutta se on kuitenkin taakka silloin, kun on tarve ymmärtää, kuinka ohjelmisto toimii. Kun koodia ei enää käytetä, se kannattaa poistaa. Versionhallinnasta koodi voidaan hakea tarvittaessa käyttöön uudestaan. (Fowler 2019, 237.)

3 Rajapinnat sovellusten integroinnissa

3.1 Mikä on rajapinta?

Tyypillisesti ohjelmia käyttävät ihmiset käyttöliittymän (user interface) kautta. Mutta enenemässä määrin ohjelmistoja käyttävät ihmisten lisäksi myös muut ohjelmistot. Tämä vaatii rajapinnan eli API:n (Application Programming Interface). Rajapinnat tarjoavat mahdollisuuden liittää, integroida ja laajentaa ohjelmistoja. Tässä kappaleessa keskitytään web-rajapintoihin, jotka tarjoavat tietoresursseja (data resources) web-teknologioiden avulla. Tyypillisimpiä applikaatioita, jotka käyttävät rajapintoja ovat verkkosovellukset, mobiilisovellukset, älylaitteet ja pilvipalvelut. (Biehl 2016, 19.)

Rajapinnat tarjoavat uudelleenkäytettävän liittymän, johon erilaisia sovelluksia voidaan yhdistää helposti. Rajapinnat eivät kuitenkaan tarjoa visuaalista käyttöliittymää, eivätkä ole suoraan näkyvissä loppukäyttäjälle ja tyypillisesti loppukäyttäjä ei ole tekemisissä rajapintojen kanssa. Rajapinnat toimivat ns. konepellin alla ja niitä kutsuvat suoraan vain muut ohjelmistot. Rajapintoja käytetään koneiden välisessä kommunikaatiossa ja kahden tai useamman systeemin integraatiossa. (Biehl 2016, 19.)

Ohjelmistojen kehittäjät ovat ainoat henkilöt, jotka ovat vuorovaikuteuksessa rajapintojen kanssa ja hyödyntävät rajapintoja muissa sovelluksissa tai ratkaisuissa. Tämän vuoksi kehittäjät ovat rajapintojen kohderyhmä ja asiakkaat, ja rajapinnat kehitetään kehittäjäien tarpeisiin. Rajapinnat tarjoavat ominaisuuksia, jotka ovat välttämättömiä ohjelmistojen yhdistämiselle, laajentamiselle ja integroinnille. Yhdistämällä eri yrityksien sovelluksia, rajapinnat yhdistävät yrityksiä myös muihin yrityksiin. (Biehl 2016, 19–20.)

3.2 Kuinka rajapintoja käytetään?

Rajapinnat ovat yksi komponentti osana tyypillistä verkkoratkaisua. Tyypillinen verkkoratkaisu koostuu sovelluksesta, rajapinnoista ja palvelinsovelluksista (backend system). Sovellus tarjotaan loppukäyttäjille, mutta loppukäyttäjä ei kutsu rajapintaa suoraan, vaan rajapintaa kutustaan sovelluksen kautta. (Biehl 2016, 22.)

Kuva 1. Rajapinta-ratkaisut (mukaillen Biehl 2016, 22)

Kuten kuvassa 1 on esitetty sovellus (client) kutsuu rajapintaa ja prosessoi rajapinnan tarjoaman datan. Sovellus on vastuussa käyttäjäkokemuksesta. Rajapinnat ovat vastuussa datasta, jota sovellukselle tarjotaan. Rajapinta-alusta hallinnoi sovellusliittymiä ja yhdistää rajapinnan taustajärjestelmiin. Palvelinsovellukset (backend systems) toteuttavat liiketoimintalogiikan, tallentavat tietoja tai suorittavat algoritmeja. (Biehl 2016, 22–23.)

3.3 Mikä on HTTP?

Jotta voidaan esitellä REST-arkkitehtuurityyli, jota tässä opinnäytetyössä käytetään, on hyvä tietää HTTP:stä, sillä REST-arkkitehtuurityyli hyödyntää HTTP:tä.

HTTP (Hyper Text Transfer Protocol) on protokolla, joka mahdollistaa resurssien (esimerkiksi HTML-sivun) haun. Tiedonvaihto verkossa perustuu client-server-protokollaan, mikä tarkoittaa sitä, että pyynnöt käynnistää vastaanottaja, joka on yleensä selain. (MDN web docs 2020, An overview of HTTP.) Nämä client:in lähettämät pyynnöt ovat HTTP-pyyntöjä (HTTP Requests) ja niihin palvelimen tarjoamat ja client:in vastaanottamat vastaukset ovat HTTP-vastauksia (HTTP Responses) (W3Schools 2020, What is HTTP?).

3.4 REST arkkitehtuurityyli

REST (Representational State Transfer) on arkkitehtuurityyli rajapinnoille. REST määrittelee joukon arkkitehtuurisia rajoituksia (constraint) ja sopimuksia (agreement). REST on

yksi käytetyimmistä arkkitehtuurityyleistä. REST on suunniteltu hyödyntämään optimaalisesti HTTP-pohjaista (Hypertext Transfer Protocol) infrastruktuuria ja protokollaa. Keskeinen käsite REST:issä on resurssi, jolla on yhtenäinen käyttöliittymä (uniform interface). (Biehl 2016, 81.)

REST-arkkitehtuurityyli asettaa seuraavia rajoituksia:

- Käyttää mahdollisimman paljon HTTP-ominaisuuksia.
- Resurssien suunnittelu menetelmien tai operaatioiden sijaan.
- Yhtenäisen käyttöliittymän (uniform interface) käyttö, joka on määritelty HTTP-menetelmillä, joilla on hyvin määritelty semantiikka.
- Tilaton kommunikaatio asiakkaan (client) ja palvelimen (server) välillä.
- Löysän kytkennän (loose coupling) käyttö ja pyyntöjen riippumattomuus.
- HTTP-palautuskoodien (return codes) käyttö.
- Mediatyyppien (media-types) käyttö.

(Biehl 2016, 81.)

Rajapintaa, joka noudattaa yllä mainittuja rajoitteita, kutsutaan termillä RESTful. Koska REST on suunniteltu hyödyntämään HTTP:tä, niin se tarjoaa paljon hyötyjä, sillä HTTP-pohjainen infrastruktuuri, kuten palvelimet, välimuistit (cache) ja välityspalvelimet (proxy), ovat hyvin saatavilla. HTTP-pohjaisen verkon suosio tarjoaa todisteen HTTP-pohjaisten arkkitehtuurien skaalautuvuudesta ja pitkäikäisyydestä. REST hyödyntää HTTP-pohjaisia rajoituksia ja sopimuksia sekä soveltaa niitä sovellusliittymiin. (Biehl 2016, 91.)

3.5 Resurssit rajapinnoissa

Resurssi on tietorakenne, joka voidaan serialisoida eri esitystapoihin, kuten esimerkiksi, JSON representaatioksi tai XML representaatioksi (Biehl 2016, 93). Representaatio on siis resurssin serilisaatio. Koska resurssi on pelkästään raakaa dataa, on se serialisoitava, jotta sitä voidaan lähettää rajapinnan ja client:in välillä. Jotta resurssin dataa voidaan käsitellä taas palvelinsovelluksen puolella, on se myös deserialisoitava takaisin liiketoiminta-objektiksi. HTTP:ssä serialisoinnin sääntöinä käytetään MIME-tyyppejä (Multipurpose Internet Mail Extensions). Kun resurssin representaatio lähetetään HTTP-pyynnöllä, niin resurssien representaatiot löytyvät pyynnön tai vastauksen HTTP body:sta. (Biehl 2016, 123.)

3.5.1 URI-suunnittelu

Rajapinta realisoituu resurssina, johon voi viitata sen URI:lla. URI (Uniform Resource Identifier) ja sen formaatti ovat standardisoituja. IETF RFC 3986 standardi määrittää kaksi eri URI tyyppiä. URI:a voi käyttää nimenä (URN Uniform Resource Names) tai lokaattorina (URL Uniform Resource Locators). (Biehl 2016, 117.)

Hyviä käytäntöjä URI:en luomiseen:

- URI:ssa käytetään vain pieniä kirjaimia
- Viimeisenä merkkinä ei käytetä kauttaviivaa ”/”
- Välimerkkiä “-” käytetään erottamaan sanoja ja luomaa luettavuutta
- URI:ssa ei käytetä tiedostotyyppejä. (Biehl 2016, 119.)

3.5.2 JSON

JSON (JavaScript Object Notation) on standardisoitu objektien merkintäkieli, jota käytetään objektien serialisointiin JavaScript:issa. JSON tukee hierarkkisia tietotyyppejä ja listoja. JSON:in serialisointia ja deserialisointia tukevat nykyään muutkin suosituimmat ohjelmointikielet kuin vain JavaScript. XML:llään (Extensible Markup Language), joka on yksi muoto esittää resursseja, verrattuna JSON vie vähemmän tilaa ja siksi JSON:in koko bitteinä on pienempi. JSON:in juurielementit voivat olla listoja tai objekteja. JSON objekti koostuu avain-arvo (key-value) pareista ja arvot voivat olla primitiivisiä tietotyyppejä, listoja tai muita objekteja. Tietokenttien nimeäminen on tyypillisesti kirjoitettu camelCase-tyylisesti. (Biehl 2016, 131–132.)

3.6 Pyyntöjen tekeminen ja parametrien sijainti pyynnöissä

REST:issä client tekee pyyntöjä palvelimelle saadakseen ja muokatakseen palvelimella olevaa tietoa, jotka esitetään resursseina. Pyynnön tekemiseen tarvitaan:

- HTTP-metodi (suosituimmat esitelty seuraavassa luvussa).
- Header, johon voidaan lisätä tietoa pyynnöstä sekä parametreja.
- URI eli polku resurssiin, URI:n loppuun voidaan lisätä parametreja.
- Vaihtoehtoinen body, johon voidaan lisätä parametreja. (Codecademy 2020, What is REST?.)

Parametreja voidaan lähettää siis HTTP-pyynnön mukana header:issa, URI:ssa tai body:ssa.

3.7 Yleisimmät HTTP metodit

HTTP-metodeilla tehdään operaatioita resursseihin. Näitä operaatioita kutsutaan CRUD-operaatioiksi (Create, Read, Update, Delete), jolla tarkoitetaan resursseille tehtäviä perusoperaatioita: luominen, lukeminen, päivittäminen ja poisto. (Biehl 2016, 145.)

3.7.1 GET

GET-metodia käytetään tietyn resurssin tietojen hakemiseen. Jos resurssi, johon GET-pyyntö lähetetään, on olemassa, niin palautetaan statuskoodi 200. Jos taas resurssia ei ole olemassa palautetaan statuskoodi 404 (Not Found). (Biehl 2016, 150.)

3.7.2 POST

POST-metodia käytetään tyypillisesti uuden resurssin luomiseen. POST-metodia käytetään myös suurissa kyselyissä, joille pitää lähettää paljon parametreja. URL:iln maksimikoko rajoittaa polkuparametreja (path parameters) ja kyselyparametreja (query parameters), jotka lisätään URL:iin, mutta POST-metodin parametrit siirretään lomakeparametreina (form parameters) HTTP body:ssa ja siksi niiden määrää tai pituutta ei ole rajoitettu. (Biehl 2016, 150.)

3.7.3 PUT

PUT-metodia käytetään tyypillisesti, kun halutaan muokata resurssia, mutta sitä voi myös käyttää, jos halutaan luoda uusi resurssi johonkin tiettyyn URL:iin. PUT-metodin pyynnössä on yleensä mukana koko representaatio luotavasta tai päivitettävästä resurssista HTTP-pyynnön body:ssa. Jos resurssi annetulla URL:illa on jo olemassa, PUT-metodilla muokataan olemassa olevaa resurssia. Jos taas resurssia ei ole olemassa, PUT-metodilla luodaan uusi sellainen. Vastaus PUT-metodipyyntöön voi olla tyhjä tai uusi representaatio resurssista. (Biehl 2016, 151.)

3.7.4 DELETE

DELETE-metodia käytetään resurssin poistamiseen tietystä URL:ista. DELETE-pyynnössä ei ole mukana ollenkaan HTTP body:a. Vastaus DELETE-pyyntöön voi pitää sisällään esimerkiksi representaation poistetusta resurssista. Jos vastaus pitää sisällään body:n, niin statuskoodi 200 OK palautetaan, mutta jos poistetusta resurssista ei tarjota vastauksena representaatiota, palautetaan statuskoodi 204 (No Content is returned). (Biehl 2016, 151.)

3.8 HTTP statuskoodit

Kun esimerkiksi joku edellä mainituista HTTP-metodeista suoritetaan jollekin resurssille, palautetaan vastauksena vähintään statuskoodi, joka ilmaisee operaation onnistumisen tai epäonnistumisen. Statuskoodeihin liittyy kuitenkin vielä paljon yksityiskohtaisempaa tietoa. Tämä mahdollistaa esimerkiksi rajapintaa kutsuvalle sovellukselle mahdollisuuksia toimia tietyllä tavalla saadessaan tietyn statuskoodin. Esimerkiksi

- jos tilapäinen ongelma ilmenee palvelimella, pyyntö voidaan suorittaa myöhemmin uudelleen,
- jos pyyntö oli epämuodostunut (malformed), voidaan pyyntö kirjoittaa uudelleen oikeaan muotoon ja lähettää pyyntö uudelleen
- jos pyydetty resurssi on siirretty, voidaan pyyntö tehdä uudelleen resurssin uuteen osoitteeseen. (Biehl 2016, 155.)

Kaikissa tavallisimmissa pyyntöjen onnistumis- tai epäonnistumistapauksissa HTTP statuskoodi määritellään numeerisena koodina ja lyhyenä ihmisen luettavana kuvauksena (Biehl 2016, 156).

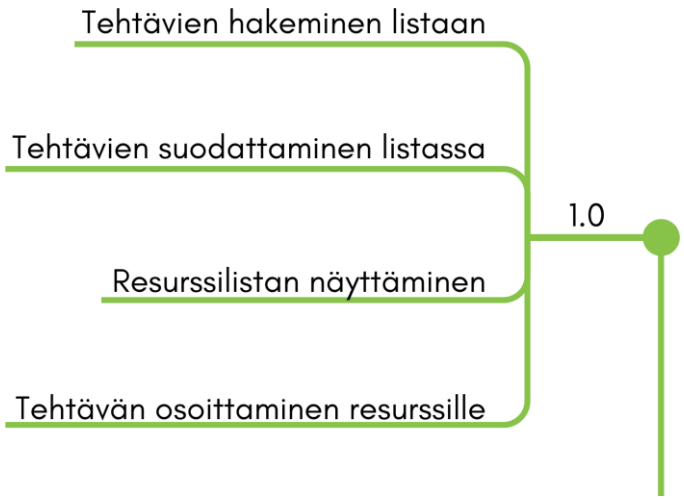
Numeeriset HTTP-statuskoodit koostuvat kolmesta numerosta ja ensimmäinen numero jakaa koodit viiteen eri ryhmään seuraavasti: (Biehl 2016, 156).

- Numerolla yksi (1xx) alkavat statuskoodit ovat tiedottavia (Informational) ja tarjoavat ei-kriittistä tietoa (Non-critical information) (Biehl 2016, 156).
- Numerolla kaksi (2xx) alkavat statuskoodit viestivät operaation onnistumisesta (Biehl 2016, 157).
- Numerolla kolme (3xx) alkavat statuskoodit viestivät uudelleenohjauksesta. Client:in on tällöin tehtävä uusi pyyntö vastauksen mukana tulleen header:in parametreihin perustuen. (Biehl 2016, 157.)
- Numerolla neljä (4xx) alkavat statuskoodit viestivät client:in virheistä (Client Error), jolloin client on vastuussa ongelmasta (Biehl 2016, 157).
- Numerolla viisi (5xx) alkavat statuskoodit viestivät palvelin virheistä (Server Error), jolloin joko palvelin tai rajapinta on vastuussa virheestä (Biehl 2016, 157).

4 Palvelinsovelluksen uudistaminen refaktoroinnilla ja rajapinnalla

4.1 Työnohjauksen roadmap

Työnohjaus on kattava sovellus ja käyttötapauksia on lukuisia. Työnohjauksen kehitysprosessia ohjaa roadmap, jonka mukaan kehitystyötä tehdään versio kerrallaan. Opinnäytetyöni keskittyy roadmap:in ensimmäiseen version palvelinsovelluksen muutoksiin. Kehitysprosessia ohjaavat roadmap:in lisäksi lähes viikoittaiset palaverit sekä lisäksi tekniset palaverit.



Kuva 2. Työnohjauksen roadmap versio 1.0 (mukaillen Alanko 12.11.2020)

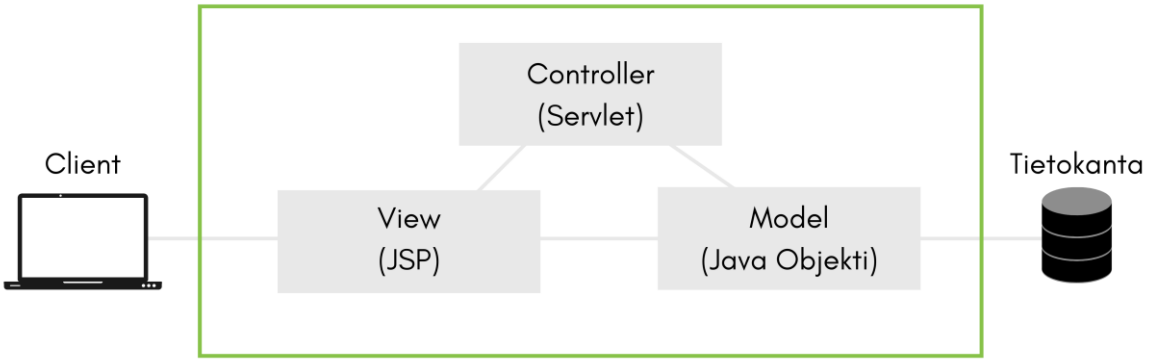
Roadmap:in ensimmäisen version palvelinsovelluksen muutostöihin kuuluu:

- Tehtävälistan datan tuottava rajapintaresurssi.
- Resurssilistan datan tuottava rajapintaresurssi.
- Tehtävän osoittamisen rajapintaresurssi.

Työnohjaus voidaan jaotella karkeasti kahteen osaan: resurssit ja tehtävät. Työnohjauksessa resurssi voi olla työntekijä, ryhmä tai organisaatio. Ryhmä on joukko käyttäjiä.

4.2 Työnohjauksen lähtötilanne

Työohjauksen palvelinsovellus noudattaa MVC-suunnittelumallia (Model, View, Controller), joka erottaa liiketoimintalogiikan, esityslogiikan ja datan. Controller toimii ohjaimena View:n eli käyttöliittymän ja Model:in eli dataobjektin välillä. Controller vastaanottaa kaikki sovellukseen tulevat pyynnöt. (JavaTpoint. MVC in JSP.)



Kuva 3. Työnohjauksen sovellusarkkitehtuurin lähtötilanne (mukaillen JavaTpoint. MVC in JSP)

Servlet on pieni java-ohjelma, joka on käynnissä web-palvelimella. Servlet vastaanottaa ja palauttaa vastauksen useimmiten HTTP-pyyntöihin. (Javadoc 2011, Interface Servlet.)

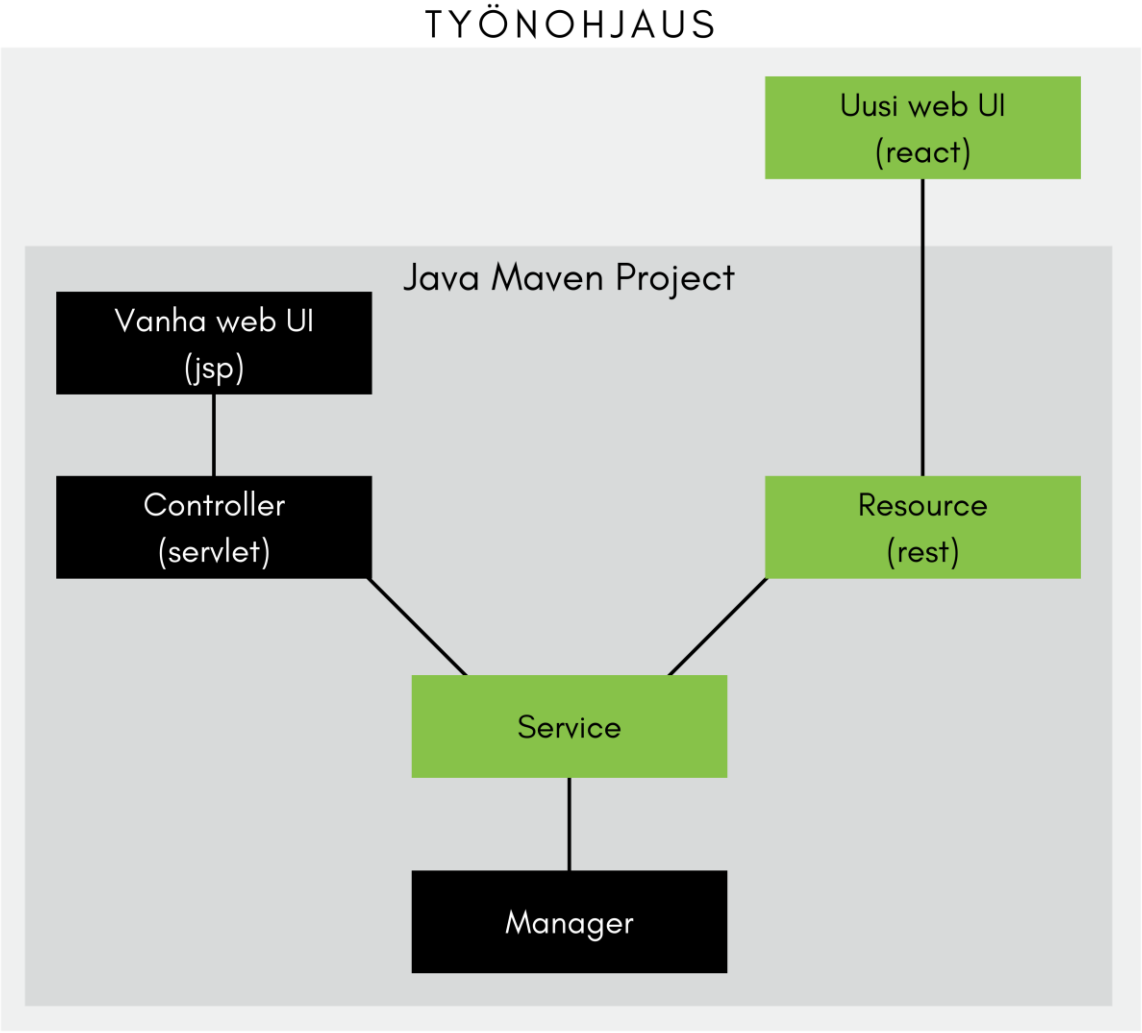
Controller hyödyntää servlet-teknologiaa ja toimii sekä View:nä ja Model:ina. Controller kontrolloi tietovirtaa Model-objektiin sekä päivittää View:n kun tieto muuttuu. Controller pitää View:n ja Model:in erillään. (Tutorials Point 2020, Design Patterns - MVC Pattern.)

Model edustaa JAVA POJO (Plain Old Java Object) objektia, joka kantaa dataa sovelluksessa (Tutorials Point 2020, Design Patterns - MVC Pattern.).

View koostuu JSP-sivuista (Java Server Pages), joka pohjautuvat Java Servlet API:iin ja mahdollistavat dynaamisen näkymän (Tutorials Point, JSP Tutorial).

Lähtötilanteessa työnohjauksen controller-luokat pitävät sisällään paljon työnohjauksen liiketoimintalogiikkaa, joka on tarkoitus siirtää kontrollereista pois omiin service-luokkiin, jotta olemassa olevaa liiketoimintalogiikkaa voidaan käyttää myös uusiin rajapintaresurs-seihin tulevissa pyynnöissä.

4.3 Työnohjauksen tekninen toteutus



Kuva 4. Työnohjauksen arkkitehtuuriuudistus (mukaillen Virtanen 16.7.2020)

Kuvassa 4 on havainnollistettu työnohjauksen uutta arkkitehtuuria uudistukseen liittyen. Vihreällä taustalla olevat komponentit ovat uutta toteutusta, ja opinnäytetyöni keskittyy erityisesti Service ja Resource komponenttien kehittämiseen.

Vanhassa toteutuksessa pyynnöt tulevat jsp:ltä controller:ille. Controller vastaanottaa jsp:ltä tulevat HttpServletRequest:it, suorittaa liiketoimintalogiikan ja kutsuu manageriluokkia. Tämän jälkeen controller asettaa palautusarvot HttpServletResponse:en ja palauttaa vastauksen jsp:lle, josta jsp lukee arvot käyttöliittymään.

Vanhaa toteutusta refaktoroidaan siten, että siirretään kaikki liiketoimintalogiikka controller:ilta service:lle ja service kutsuu manageriluokkia. Eli controller ainoastaan vastaanottaa vanhasta käyttöliittymästä tulevia pyyntöjä, jotka ohjataan service:n sisältämän liiketoimintalogiikan läpi.

Service kerros luodaan sen vuoksi, että vähennetään duplikaattikoodia, kun uudesta käyttöliittymästä tulevat pyynnöt voidaan ohjata saman service:ssä olevan liiketoimintalogiikan läpi. Samalla kaikki liiketoimintalogiikka on eristetty omaksi kerrokseksi erilliseksi pyynnöistä, jolloin kaikki pyynnöt, kuten vanhasta ja uudesta käyttöliittymästä sekä mobiilista tulevat pyynnöt, käyttävät samaa liiketoimintalogiikkaa.

Uudessa toteutuksessa pyynnöt tulevat erillisestä react-sovelluksesta rajapintaresursseille. Rajapintaresurssit ja service:t tehdään ja nimetään toiminnallisuuksien mukaan, esimerkiksi TaskResource, TaskService, jotka hoitavat tehtäviin liittyvät pyynnöt ja toiminnallisuudet.

4.4 Työnohjauksen backend teknologiat (Tech Stack)

- Java – Työnohjauksen palvelinsovelluksen on ohjelmointikieli.
- JAX-RS framework – hoitaa DTO:iden parsinnan JSON:iksi ja takaisin liiketoimintaobjekteiksi sekä URI-polkujen ja REST-metodien määrittelyn.
- Hibernate (ORM Object-Relational Mapping) – työkalu, jota käytetään tietokantataulujen ja objektien kartoittamiseen.
- Maven – Projektin kääntäminen.
- Git – Versionhallinta.
- PostgreSQL – Tietokanta.

Työnohjauksen palvelinsovelluksen uudistus halutaan tehdä siten, että vanha toteutus ei rikkoutuisi. Tämän vuoksi työnohjauksen palvelinsovelluksen uudistuksessa pystyään jo käytetyissä teknologioissa. Näiden teknologioiden riippuvuudet löytyvät jo projektista eikä uusia riippuvuuksia tarvitse lisätä. Lisäksi firmasta löytyy osaamista ja dokumentaatiota näiden teknologioiden osalta.

Rajapinta-arkkitehtuurityyliksi on valittu REST, sillä projektissa on jo riippuvuudet valmiina rajapintojen toteuttamiselle ja jotakin rajapintoja projektiin on toteutettu jo ennestään muita käyttötarkoituksia varten. Lisäksi REST on yksi suosituimmista ja siksi myös tuetuimmista rajapinta-arkkitehtuurityyleistä ja REST rajapintakehittäminen java:lla on hyvin tuettua.

4.5 Kehitysprosessi

Tässä kappaleessa käytetyt koodiesimerkit eivät vastaa työohjaukseen tehtyjä koodimuutoksia, vaan niiden tarkoitus on vain havainnollistaa muutoksia kooditasolla. Koodiesimer-

keistä nähdään, että refaktorointimenetelmistä on eniten käytetty koodin siirtämistä funktioon ja funktioiden yhdistämistä luokkaan. Palvelinsovelluksesta löytyy jo ennestään kattavat testit työstettäville osa-alueille, jotka tarkistavat, ettei mikään mene rikki refaktorointiprosessissa.

4.5.1 Controller

Refaktoroinnin jälkeen controller:ille jäävät vain seuraavat tehtävät:

- Parsii jsp:ltä tulleen pyynnön mukana tulleet arvot parametreiksi ja välittää ne service:lle metodikutsun mukana.
- Parsii service:n palauttaman liiketoimintaobjektin, asettaa arvot vastaukseen ja palauttaa vastauksen jsp:lle.

```
public class TasksGet extends Controller {

    @Override
    protected void processRequest(Session session, HttpServletRequest request, HttpServletResponse response) {

        String jspPath = request.getParameter("jspPath");

        // parsitaan parametrit pyynnöstä
        int param = ParameterUtil.parseIntFromRequest(request, "param");

        // luodaan service ja välitetään parametrit service:lle
        TaskService service = new TaskService(session);
        List<Task> tasks = service.queryTasks(param);

        // asetetaan haettu/päivitetty data vastaukseen
        request.setAttribute("tasks", tasks);
        request.getRequestDispatcher(jspPath).forward(request, response);


    }
}
```

Esimerkkikoodi 1. Esimerkki refaktoroidusta controller:ista

4.5.2 Rajapinta

Uutta käyttöliittymää varten toteutetaan rajapinnat REST-resursseina.

- Resurssit tehdään käyttötapauskohtaisesti, esimerkiksi tehtävien haku hakuehdoilla on kokonaan oma resurssinsa.
- Rajapintaan lisätään versio. Tämä mahdollistaa rajapinnan muuttamisen ilman, että vanhat toteutukset hajoavat. Kun halutaan tehdä päivityksiä rajapintaan, niin voidaan tehdä uusi rajapinta uudella versiolla ja jättää vanha versio paikalleen. Esimerkiksi v1, v2 jne.
- Rajapinnat tehdään RESTful periaatteella:
    - GET .../v1/task/<id> - hakee yksittäisen tehtävän

- POST .../v1/task/<id> - päivittää tehtävän rivitietoja, parametreina annetaan vain päivitettävät kentät.

(Virtanen 16.7.2020.)

Rajapintaresurssin tehtävät ovat:

- Parsia uudelta käyttöliittymältä tai mobiilista tulleiden pyyntöjen mukana tulleet DTO-objektit parametreiksi ja välittää parametrit service:lle metodikutsun mukana.
- Parsia service:ltä tulleen liiketoimintaobjektin DTO:ksi ja lähettää sen vastauksen mukana uudelle käyttöliittymälle tai mobiilille.

```
// resurssin URI-tunnistepolku
@Path("/v1/tasks")
public class TaskResource {

    ...
    ...
    ...

    @GET // HTTP-metodi
    @Produces(APPLICATION_JSON_UTF8)
    @Path("/{param}") // vastaanottaa polkuparametrin
    public Response queryTasks(@PathParam("param")int param) {

        // luodaan sessio pyynnöstä
        Session session = Session.initSessionRest(request);

        // luodaan service ja välitetään parametrit service:lle
        TaskService service = new TaskService(session);
        List<Task> tasks = service.queryTasks(param);

        // luodaan palautettavat DTO-objektit service:n palauttamista liiketoimintaobjekteista
        List<TaskDTO> taskDTOs = TaskDTO.fillTaskDTOListFromTasks(tasks);

        return Response.ok(taskDTOs, MediaType.APPLICATION_JSON).build();

    }
}
```

Esimerkkikoodi 2. Esimerkki uudesta rajapintaresurssista

Ensimmäisten rajapintaresurssien mukana implementoidaan uusia tarpeita. Osa työnohjauksen tehtävistä tulee työnohjaussovelluksesta (jos tehdään uusi tehtävä työnohjaussovelluksen kautta) ja osa tehtävistä tulevat ulkopuolisista lähteistä, kuten asiakkaiden omista rajapinnoista. Työnohjauksen tehtävälistaan olisi hyvä saada tieto siitä, mikä tehtävä tulee ulkopuolelta ja mikä on luotu työnohjaussovelluksessa. Lisäksi olisi hyvä saada työnohjauksen tiedot sekä ulkopuolisen lähdejärjestelmän tiedot. (Alanko 9.6.2020.)

Tehtävälistan datan tuottaman rajapintaresurssin representaation mukana palautetaan myös tiedot:

```
Boolean isTask;
Boolean isNewTask;
```

Jokaisen tehtävälistan tehtävän mukana tulee uutena Boolean-tyyppiset muuttujat isTask ja isNewTask. Jos isTask on true, tehtävä on luotu työnohjaussovelluksessa, jos taas isNewTask on true, tehtävä tulee ulkopuolisesta lähteestä. Ulkopuolisen lähdejärjestelmän tiedot implementoidaan myöhemmin.

4.5.3 Service

Uutta ja vanhaa client toteutusta varten tehdään uusi service kerros, jonne kopioidaan liiketoimintalogiikka vanhoista controller:eista. Tarkoitus on kutsua tässä kerroksessa olemassa olevia manageriluokkia, kuten on aikaisemmin tehty vanhoissa controller:eissa. Service-luokan pitäisi muodostaa yhtenäinen rajapinta, jota voidaan kutsua sekä vanhoista controller:eista että uusista rajapintaresursseista. (Virtanen 16.7.2020.)

Service:n tehtävinä ovat

- Vastaanottaa parametreja controller:eilta ja rajapintaresursseilta.
- Suorittaa liiketoimintalogiikka sekä DAO- ja manageriluokkien kutsuminen.
- Palauttaa parametrien mukaan haettu tai päivitetty liiketoimintaobjekti.

```
public class TaskService {

    private final Session session;

    public TaskService(Session session) {
        this.session = session;
    }

    public List<Task> queryTasks(int param) {
        // suoritetaan controller:ista siirretty liiketoimintalogiikka

        ...
        ...
        ...

        List<Task> tasks = // kutsutaan DAO-/manageriluokkia

        // palautetaan haetut/päivitetyt liiketoimintaobjektit
        return tasks;

    }
}
```

Esimerkkikoodi 3. Esimerkki uudesta service-luokasta

4.5.4 Versionhallinta ja työnkulku

Palvelinsovelluksen uudistusprojektille ei ole perustettu omaa erillistä repository:a versionhallintaan, vaan työ tehdään olemassa olevaan repository:yn. Työ etenee siten, että luodaan haara (branch) tiketti kerrallaan. Java projektin web.xml:lstä, johon controllerit:it ovat kartoitettu tiettyihin URL-endpoint:eihin, sekä käyttötapauksien mukaan katsotaan pyyntö kerrallaan, minkä controller:in liiketoimintalogiikka refaktoroidaan uuteen service-luokkaan ja samalle pyynnölle tehdään uusi rajapintaresurssi.

Kun muutokset ovat tehty, commit:oidaan muutokset haaraan, katselmoidaan, testataan ja merge:tään master:iin. Palvelinsovelluksen muutokset saadaan toimitukseen sitä mukaan, kun niitä tehdään, joten mobiilikehittäjät ja uuden käyttöliittymän kehittäjät voivat hyödyntää uusia rajapintaresursseja omassa kehitystyössään. Muutoksia ei voi säilyttää pitkään omassa haarassaan, sillä työnohjaukseen tehdään jatkuvasti muuta kehitystä, joten merge konfliktien välttämiseksi kehitystyö on saatava nopeasti master:iin.

4.5.5 Pakettihierarkian muutokset

Refaktorointiprosessissa pyritään samalla muuttamaan pakettihierarkian jaottelua toiminnallisuuksien mukaan, sillä työnohjaus on suuri sovellus ja perinteisessä MVC eli Model View Controller -jaottelussa, paketit paisuvat suuriksi ja niistä on vaikea löytää oikeaa luokkaa.

Kuva 5. Esimerkki vanhasta pakettirakenteesta

Pakettihierarkiaa muutetaan siten, että esimerkiksi kaikki tehtäviin liittyvät luokat löytyvät MVC-jaoteltuna task-paketin alta.

Kuva 6. Esimerkki uudesta pakettirakenteesta

Rajapintaresurssit löytyvät rest-paketin alta versioittain ja toiminnallisuuksittain jaoteltuna.

Kuva 7. Esimerkki rajapintaresurssien uudesta pakettirakenteesta

5 Pohdinta

Opinnäytetyön tuloksena syntyivät seuraavat rajapintaresurssit:

- GET → /v1/tasks
    - Palauttaa datan tehtävälistaan.
- GET → /v1/resources
    - Palauttaa datan resurssilistaukseen.
- POST → /v1/assignments/assign
    - Suorittaa tehtävän osoittamisen liiketoimintalogiikan ja palauttaa osoitetun tehtävän.

Sekä rajapintaresurssien liiketoimintalogiikan sisältävät service-luokat:

- TaskService
- ResourceService
- AssignmentService

Lisäksi tarvittavat DTO-luokat resurssien esittämiseen ja muutokset uusia pyyntöjä vastaaviin vanhoihin controller:eihin.

Opinnäytetyöni on hyvä esimerkki palvelinsovelluksen uudistusprojektista ja tarjoaa yhden lähestymistavan siihen, kuinka uudistus voidaan toteuttaa. Lisäksi opinnäytetyö tarjoaa näkökulmia koodiperustan jatkuvaan kehittämiseen, jotta koodirakenne pysyisi mahdollisimman joustavana muutoksille ja uudelle kehitykselle.

Monissa organisaatioissa on vanhoja legacy-järjestelmiä, joiden tulee säilyä ehjinä, mutta vastaanottaa myös muutoksia ja uutta kehitystä. Opinnäyteyöni tarjoaa lähestymistavan siihen, kuinka voidaan säilyttää vanha toiminnallisuus, mutta integroida uusia muutoksia ilman vanhan koodin rikkoutumista ja uudelleenkirjoittamista.

Opinnäytetyön tuloksena syntyneiden toimivien rajapintaresurssien sekä refaktoroidun liiketoimintalogiikan perusteella kehitystyötä voidaan jatkaa tässä opinnäytetyössä tehdyn konseptitodistuksen mukaan. Opinnäytetyö on luonut toimivan esimerkin ja pohjan kehitystyön jatkamiselle. Opinnäytetyön tuloksena syntyneet rajapintaresurssit tarjoavat helpon integraation sekä uudelle käyttöliittymälle sekä mobiilisovellukselle.

Opinnäytetyössä halusin selvittää mikä on paras lähestymistapa palvelinsovelluksen uudistamiseen ilman että olemassa olevaa projektia pitäisi korvata uudella. Tavoitteenani oli

tehdä raportointi siitä, miten palvelinsovelluksen kehitystä kannattaisi jatkaa. Opinnäytetyössäni olen pyrkinyt havainnollistamaan mitä ensimmäisten uudistusten kohdalla on tehty teknisesti.

Opinnäytetyöstä aloittaessani työnohjaus oli minulle uusi sovellus ja opinnäytetyöni ohessa työskentelin myös muissa kehitysprojekteissa, joten aikaa opinnäytetyöhön käytettäväksi oli rajallisesti. Onnistuin kuitenkin tuottamaan työnohjauksen uudistuksen ensimmäiseen versioon tarvittavat rajapintaresurssit, joten kuvailisin työtä onnistuneeksi. Opin opinnäytetyön aikana paljon refaktoroinnista ja rajapintaresurssien kehittämisestä osana olemassa olevaa järjestelmää.

Lähteet

Alanko, O. 9.6.2020. Toimitusjohtaja. Geometrix Oy. Työnohjauksen käyttöliittymän kehittäminen palaveri. Helsinki.

Alanko, O. 12.11.2020. Toimitusjohtaja. Geometrix Oy. Työnohjausprojektin roadmapin suunnittelu palaveri. Helsinki.

Biehl, M. 2016. RESTful API Design Book. API-University Press.

Codecademy 2020. What is REST?. Luettavissa: https://www.codecademy.com/articles/what-is-rest. Luettu: 12.12.20202.

Fowler, M. 2019. Refactoring: Improving the Design of Existing Code. Addison-Wesley. Boston, United States.

Geometrix 2020. Meistä. Luettavissa: https://www.geometrix.fi/?page_id=39. Luettu: 17.11.2020.

Geometrix 2020. Työnohjaus. Luettavissa: https://www.geometrix.fi/?page_id=91. Luettu: 17.11.2020.

Javadoc 2011. Interface Servlet. Luettavissa: https://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html#:~:text=A%20servlet%20is%20a%20small,servlet. Luettu: 12.12.2020.

JavaTpoint. MVC in JSP. Luettavissa: https://www.javatpoint.com/MVC-in-jsp. Luettu: 2.12.2020.

MDN web docs 2020. An overview of HTTP. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Luettu: 12.12.2020.

Suomen asiakastieto Oy. Asiakastietorekisterin taloustiedot. Verkkoaineisto. Luettavissa: https://www.asiakastieto.fi/yritykset/fi/geometrix-oy/17790520/taloustiedot. Luettu: 2.12.2020.

Tutorials Point 2020. Design Patterns - MVC Pattern. Luettavissa: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm Luettu 12.12.2020.

Tutorials Point 2020. JSP Tutorial. Luettavissa: https://www.tutorialspoint.com/jsp/index.htm. Luettu: 12.12.2020.

Virtanen, A. 16.7.2020. Ohjelmistosuunnittelija. Geometrix Oy. Työnohjauksen tekninen toteutus palaveri. Helsinki.

W3Schools 2020. What is HTTP?. Luettavissa: https://www.w3schools.com/whatis/whatis_http.asp. Luettu: 12.12.2020.