Toni Takala

Spring-palvelinsovellus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

12.4.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Toni Takala Spring-palvelinsovellus 31 sivua 12.4.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja	Lehtori Peter Hjort

Insinöörityössä tutkittiin käytännön toteutuksien avulla mahdollisuuksia rakentaa moderneja teknologioita hyödyntävä geneerinen web-palvelinalusta. Tässä työssä käsiteltiin useita palvelinohjelmiston kehityksessä hyödynnettyjä teknologisia ratkaisuja.

Java-kielellä insinöörityössä ohjelmoidun palvelinalustan teknologiakatsauksen lisäksi työssä raportoitiin myös ohjelmointityön teknologioiden toteutuksista palvelinalustalla. Valmistuneen työn tarkoitus oli tuottaa jatkokehitetykseen soveltuva palvelinalusta, jonka päämäärä oli ensisijaisesti tuottaa ja hyödyntää rajapintapalveluita nykyaikaisia integraatioteknologioita hyödyntämällä.

Työn motivaationa toimi jo pitkään mielenkiintoa herättänyt Java-palvelinsovellusohjelmointi ja järjestelmien välinen kommunikaatio. Järjestelmien välisen kommunikaation rooli tulee kasvamaan vielä valtavasti tulevaisuudessa palveluiden siirtyessä entistä enemmän pois fyysisistä samalla alustalla tai toisiinsa kytköksissä olevista palveluista kohti pilvipalvelua.

Työssä tarkasteltiin teknologioita, joita hyödynnettiin valtaosin toisistaan eriytettynä työssä valmistuneessa palvelinalustakonseptissa. Rajapintateknologiat toteutettiin JSON-, REST- ja SOAP-tekniikoiden avulla, ja niitä käytettiin palvelinalustassa HTTP-protokollan välityksellä.

Palvelinalustasovelluksen keskeisenä komponenttina oli Spring Framework 4, joka yksinkertaistaa useita Java Enterprise Editionin standardinomaisia toteutuksia kehitystyössä. Valmistuneessa sovelluksessa hyödynnettiin yleisimpiä olio-ohjelmointia tukevia työkaluja, kuten Hibernaten tarjoamaa olioiden tallennusta ja tilanhallintaa sekä useita muita apukirjastoja, jotka tukivat eri osa-alueilla liiketoiminnallisia tarpeita.

Sovellukseen kehitettiin myös oma nykyaikaisia rajapintateknologioita mukaileva sovelluskomponentti, joka hyödyntää REST-, JSON- ja HTTP-teknologioita. Sovelluksessa komponentille annettiin työnimike ApiToken, joka kuvasi sen roolia rajapinnassa välitettävissä viesteissä palvelimen ja asiakkaan välillä. Palvelinalustalle tuotettiin myös kokeellinen PayPal-maksunvälityspalvelun asiakasrajapinta, jonka kautta tuotettiin asiakkaalle maksutapahtumia portaalin välityksellä.

Avainsanat	Java, JSON, Spring 4, SOAP, REST, Hibernate

Author Title Number of Pages Date	Toni Takala Spring-server application 31 pages 12 Apr 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Peter Hjort, Senior Lecturer

The purpose and goal of this final year project was to examine possibilities to build a software server platform with modern technologies. The main parts of the server implementation technologies are covered in this thesis.

In practice, the server software was built with the Java programming language. The technology and components of the software and its production phases were covered in the final year project. The goals of this study were to produce a basic server implementation and a concept of server software that could be integrated into multiple third party systems with modern integration technologies.

The motivation for this study was a deep interest in Java server software programming and communication between multiple distinct systems. The role of the communication between distinct systems will see a significant growth in the future since physical on-site servers are transforming into cloud server infrastructures. Transforming physical servers into cloud infrastructure services decreases serve maintenance costs and increases infrastructure and software services reliability.

Technologies implemented in the conceptual server application were examined separately from another. Integration technologies were produced by using the JSON, REST and SOAP technologies over the HTTP protocol.

The fundamental part of the server software was the Spring Framework 4 which brings a simplified way to solve some of the standard problems of the Java Enterprise Edition in software development. The most common tools of object oriented Java programming development were also utilized in this project such as Hibernate persistence management and various other libraries. Software development related frameworks were used as tools to support the development from a business aspect of the developed server concept.

The study project included a custom implementation of a software component that utilized the REST, JSON and HTTP technologies. This component was referred to as an ApiToken which describe its role between the server and integrated clients. The server was integrated with a global payment system provided by PayPal.

Keywords	Java, JSON, Spring 4, SOAP, REST, Hibernate

Sisällys

1 Johdanto

Tämän insinöörityön tarkoituksena on tutkia Javalla toteutetun web-palvelinsovelluksen rakennetta hyödyntäen moderneja teknologioita. Työssä sovelletaan useita integraatioteknologioita ja niistä luodaan yleiskäyttöinen palvelinsovellus. Tuotettavan sovelluksen tarkoitus on soveltua yleisellä tasolla moderneja integraatioita tukevaksi web-palvelinalustaksi myöhempää jatkokehitystä ajatellen.

Käytettävät teknologiat valitaan siten, että ohjelmiston olisi mahdollista skaalautua jatkokehityksen myötä hyvin sekä aloittavien pienten yritysten että keskikokoisten yritysten tarpeisiin. Palvelinsovelluksen ohjelmointikieleksi valittiin Java, sillä se on yksi maailman suosituimmista ohjelmointikielistä ja se koetaan edelleen moderniksi kieleksi ohjelmistokehittäjien keskuudessa [1].

Javan ympärille rakennettu kehityspino mahdollistaa nopean tavan tuottaa palvelinohjelmisto, joka mahdollistaa nopean käyttöönoton. Keskeisenä asiana työssä on toteuttaa palvelinalustakonsepti, jota on mahdollista laajentaa työssä esiteltävien teknologioiden toteutuksien myötä useisiin eri käyttötarkoituksiin jatkokehityksen myötä.

Jatkokehityksen mahdollisuuksia tarkastellaan työssä teoreettisesta näkökulmasta, mutta myös osittain esimerkkien kautta. Työssä esitellään valittuja teknologioita ja esitellään palvelinohjelmiston osaksi kehitettyjä teknologioita. Palvelinsovellukseen liittyviä keskeisiä konfiguraatioita esitellään esimerkein siten, kuin se liittyy käsiteltyihin aiheisiin.

Bisneslogiikkaa toteuttavien luokkien toimintaa kuvataan soveltuvin osin. Työssä syvennytään erityisesti Spring 4 -ohjelmointikehyksen hyödyntämiseen palvelinohjelmiston konfiguroinnissa, ohjelmointityöhön ja tuloksiin sekä järjestelmien välisiin REST- ja SOAP-integraatioteknologioihin.

Mielenkiintoni on ohjautunut tietotekniikan alalla vahvasti kaupallisen näkökulman tarkasteluun ohjelmointityössä ja ohjelmistomarkkinoilla. Pyrin ylläpitämään innovatiivista näkökulmaa tekemissäni ohjelmistoprojekteissa ja löytämään siten uusia tapoja tuottaa moderneja palveluita. Innovaatioiden rakentaminen ohjelmistoalalla vaatii mielestäni ohjelmoijan omistautumista innovoinnin kohteeseen ja parhaiden käytäntöjen soveltamista

oman näkökulman mukaisesti. Insinöörityö käsittelee teknisen toteutuksen näkökulmasta mahdollisuutta rakentaa palvelinalusta liiketoiminnan tueksi.

2 Tavoitteet

2.1 Motivaatio

Intohimoni on ollut oman yritystoiminnan aloittaminen. Työn kautta opintojen ohessa hankittu tietotaito järjestelmien toimittamisessa on osaltaan auttanut minua ymmärtämään sovelluksille yleisesti asetettuja odotuksia ja tunnistamaan nykyaikaiselle palvelinalustalle asetettuja odotuksia. Insinöörityön motivaationa oli tunnistaa työssä tuotetun ohjelmiston avulla mahdollisia web-palvelutuotannon markkina-alueita.

Teknologioihin tutustumisen ja käytännön toteutuksien avulla insinöörityössä kuvatusta ja kehitetystä palvelinalustasta voidaan saada merkittävää tietoa liiketoimintamahdollisuuksista tulevaisuudessa. Työssä toteutettu järjestelmä suunniteltiin tukemaan yritystoiminnassa usein toistuvien ja tarkkuutta vaativien liiketoimintaprosessien automatisointia ja tukemaan liiketoiminnallisia kehityskohteita.

Olen käynyt keskusteluita pienyrittäjien ja ammatinharjoittajien kanssa, jotka jakavat saman ongelman. Pieniltä yrityksiltä puuttuu usein kustannustehokas IT-infrastruktuuri. Tällä tarkoitetaan esimerkiksi asiakkaille tärkeän palvelun saatavuutta ja toisesta näkökulmasta palvelujen näkyvyyttä. IT-infrastruktuurin rakenteeseen lukeutuu myös yritysten sisäisten toimintojen hallinta ja liiketoiminnalle kriittisten toimintojen automatisoinnin tarve.

Edellä mainitun infrastruktuurin rakentaminen liiketoiminnan tarpeita tyydyttäväksi palvelukokonaisuudeksi mielletään liiketoiminnan kannalta riskialttiiksi sijoitukseksi, sillä infrastruktuurin toimintoihin joudutaan budjetoimaan merkittäviä resursseja. Toisekseen markkinoita ja myyntiä on vaikeaa ennustaa, jolloin kehitystyön tilaaminen voi hankaloitua. Usein vakavaraisilla pienillä ja markkina-alueensa vakiinnuttaneilla keskisuurilla yrityksillä on mahdollisuuksia budjetoida tarvittavien ohjelmistojen ja muun IT-infrastruktuurin kehityskustannuksia yrityksen tuottavuuden parantamiseksi. Kustannukset koostuvat ohjelmiston kehitystyöstä mutta myös laitteiston ja ohjelmiston ylläpitokustannuksista.

IT-infrastruktuurin ja automaatioihin budjetoineet yritykset ovat kasvattaneet tuottavuuttaan muita enemmän. [11.]

Pienyrittäjän käyttötarkoitusta ja budjetointia vastaavaa IT-infrastruktuuria ei ole juurikaan tarjolla nykymarkkinoilla. Tämä herättää myös kysymyksen, miksi pienille yrityksille ei ole muotoiltu IT-infrastruktuurin palvelukokonaisuutta, jossa tyypilliset yritystoiminnan vaatimat tavallisimmat automaatiotarpeet olisi mahdollistettu mielekkään budjetin rajoissa. Perusominaisuuksista asiakas voisi maksaa kiinteän hinnan, jolloin kehityskustannuksia saataisiin jaettua useamman samoja perustoiminnallisuuksia hyödyntävän asiakkaan myötä matalemmiksi. Tässä palvelumallissa palvelinalustan hankinta olisi mahdollista budjetoida nykyisestä poiketen myös pienten ja vasta aloittavien yritysten kohdalla kohtuullisella riskillä. Palvelumallilla tarkoitetaan IT-infrastruktuurin perustoiminnallisuuksien tarjoamista palvelinalustana, jonka avulla yrityksen toimintaa olisi mahdollista tehostaa. Edellä esitetyn palvelumallin mukaisen palvelinalustan hankinnan myötä yrityksen näkyvyys, toimintojen ja asiakkuuksien hallinta ja näiden edistäminen olisi mahdollista kohtuullisella budjetilla. Pienten yritysten IT-infrastruktuurin hankinta palveluna parantaisi budjetointiin liittyvien riskien hallintaa. Vastaavanlaisia huomioita hinnoittelusta, kysynnästä ja yritysten prosessien tehostamisesta on myös Matthew Burrows tehnyt artikkelissaan ”Operational efficiency – it’s not just about cost cutting” [2].

Inspiraatio tämän opinnäytetyön ja konkreettisen palvelinohjelmiston toteuttamiseen juontaa edellä käsitellyistä yritysmaailman ja ohjelmiston tuotteistamisen ongelmista sekä erityiskiinnostuksesta palvelualustan tuottamiseen ja kehittämiseen. Mallinsin työssä kehitetyssä palvelinohjelmiston rungolla kokonaisuutta, joka tuki pohjatoteutuksena pienen yrityksen vaatimia IT-infrastruktuurin perustoiminnallisuuksia erityisesti teknologiavalintoja esittelevinä esimerkkitoteutuksina jatkokehitystä ajatellen.

Kehityssuuntana alustan kaupallistamisessa saattaisi olla infrastruktuurin lisensointi ja palvelukohtainen ohjelmiston räätälöinti liiketoimintaa automatisoivien modulaaristen komponenttien avulla.

Palvelinohjelmistojen komponenttien yleiskäyttöisyyttä ja ohjelmiston rakenteen modulaarisuutta tukevaa teknologiaa on jo olemassa. Näin voidaan yksinkertaistaa ja tehostaa useita monimutkaisia prosesseja palvelinohjelmiston toteutuksessa. Ohjelmistokehityksen näkökulmasta tällä osa-alueella merkittävässä roolissa on myös työssä käsitelty Spring 4 -ohjelmistokehys, joka itsessään tarjoaa vaihtoehtoisen tavan tuottaa JavaEE-

palvelinsovelluksia. Huolellisesti suunniteltuna Spring antaa työkalut rakentaa kustannustehokas ratkaisu modulaarisena kokonaisuutena, jonka myötä ohjelmistoratkaisuja ja siten automaatioinfrastruktuuria voidaan tuottaa myös pienempien yritysten saataville.

Työssä kehitetyn palvelinalustasovelluksen rungon käyttötarkoitus oli tuottaa useita edellä mainittuja ylätasolla määritettyjä automaatioiden teknisiä toteutuksia myöhemmin kehitettävälle palvelinalustakokonaisuudelle. Tärkeimmiksi palvelinsovelluksen palvelukonsepteiksi ja tavoitteiksi määriteltiin asiakkuuksien tunnistaminen, maksunvälitys, palveluväylät muihin järjestelmiin sekä järjestelmän tukemien informaatiokanavien luonti liiketoiminnan ja asiakkuuksien välille.

Asiakkuuksien tunnistamisella tarkoitetaan yleistä asiakasrekisterin ylläpitoa. Yrityksen tunnistettua asiakkuuksiensa laadun se voi tehdä päätelmiä markkina-alueesta ja tulevaisuuden liiketoimintastrategiasta. Informaatiokanavien luonnilla tarkoitan asiakkuuksien ja liiketoiminnan kommunikointia jonkin tai joidenkin IT-infrastruktuuriin liittyvien palveluiden välityksellä. Insinöörityössä kehitetyssä sovelluksessa nämä huomioitiin HTTP(S)-protokollan kautta tarjotuilla palvelulla sekä SMS-viestipalvelulla, jonka keskeinen tarkoitus oli tarjota väylä suoraan tiedottamiseen asiakasrajapinnassa.

2.2 Käyttöönottosuunnitelma

Palvelinalustasovellus oli tarkoitus ottaa käyttöön yhdessä tai useammassa tavoitteissa asetetussa käyttötarkoituksessa. Tässä vaiheessa alustan täytyi tukea teknologiatasolla aiemmin mainittuja palveluita ja liityntöjä muihin palveluihin. Työssä valmistui palvelinalusta, joka oli teknologiatasolla määritellyn pohjan hahmotelma. Hahmotelmaa voidaan esitellä mahdollisia asiakkuuksia varten.

Käyttöönottoa varten tulisi huomioida asiakkaan olemassa olevat ratkaisut ja laitteistot ja niiden sovittaminen uuteen ympäristöön. Esimerkki käyttöönoton suunnittelua vaativasta kohteesta olisi asiakas, jolla on ollut käytössä jokin aiempi tuote, jolla on tuotettu sisältöä verkkosivustolle. Tässä tapauksessa tulee harkita asiakaskohtaisesti sekä käyttökohteen että asiakkaan tarpeiden mukaisia ratkaisuja. Palvelinalustan käyttöönotto ei estä asiakkuuksia jatkamasta esimerkiksi sisällöntuotantopalveluita, sillä palvelinalusta voidaan määrittää palvelemaan toisaalla uudelleenohjauksin olemassa olevista sisällöntuotantojärjestelmistä. Yksinkertaisimmillaan samalla taustapalvelimelta tuotettaisiin myös verkkosisältöä. Verkkosivuston tarjoaminen taustapalvelimella on kuitenkin herkkä

altistumaan tietoturvahyökkäyksille, jolloin tietoturvauhat tulisi ottaa erityisesti huomioon palveltaessa verkkosivuja taustapalvelimelta. On suositeltavaa käyttää erillistä edustapalvelinta, joka keskustelee taustapalvelimen kanssa.

Tietoturva-analyysia tulee suorittaa suunnitellun modulaarisuuden myötä toimintokohtaisesti. Näin voidaan varmistaa tietoturvallisuus tutkimalla yksittäisten komponenttien mahdollisesti luomat riskitekijät sekä turvata taustapalvelimen toiminta. Edustapalvelimen käyttö on suositeltavaa siltä varalta, että edustapalvelin altistuisi tietoturvahyökkäyksen kohteeksi. Taustapalvelin ja kriittinen data ei siten suoraan altistuisi hyökkäykselle. Esimerkki tietoturvahyökkäyksestä edustapalvelimeen voisi olla DDoS eli ”Distributed Denial of Service” -palvelunestohyökkäys. Hyökkäys aiheuttaa palvelimella suuren väliaikaisen rasituksen tuhansien samanaikaisesti saapuvien HTTP- tai HTTPS-pyyntöjen välityksellä. Tämä aiheuttaa palvelimella ruuhkaa, joka johtaa usein resurssien ehtymisen. Tällöin palvelimen palvelutehokkuus ja muut toiminnot hidastuvat merkittävästi. Mikäli taustapalvelimeen kohdistetaan kyseinen hyökkäys, on mahdollista, että hyökkäyksen aikana palvelimelle lähetettyä dataa häviäisi. Työssä kehitettävien ratkaisujen myötä muun muassa tietoturvallisuutta

Työssä valmistuneelle palvelinkonseptille ilmeni varhaisessa vaiheessa käyttökohde, jonka myötä kehitystyötä jatkettiin. Palvelinalustasta ja sen jatkokehittämisestä lähetettiin erillinen tarjous asiakkaalle. Tarjouskilpailu oli yksityinen, joten lähetettyä tarjousta ei julkaista tässä työssä. Tarjouskilpailussa alusta ei kyennyt vastaamaan kyseistä tarvetta, jolloin alustan tarjoamaa teknologiaa tuli suunnitella ja osittain myös jatkokehittää olemassaolevien komponenttien osalta suuntaan, mikä vastaa paremmin ja yleisemmällä tasolla liiketoiminnallisia tarpeita.

2.3 Aikataulutus

Projektityötä ja siihen liittyviä teknologiakokonaisuuksia alettiin suunnitella lokakuussa 2016. Projektille asetettiin lähtökohtaisesti tavoitteet siten, että työssä alustavasti hahmotellun palvelinsovelluksen valmistuttua se olisi modulaarinen, testattu ja dokumentoitu ohjelmistokokonaisuus, jonka päälle olisi mahdollista rakentaa tulevaisuudessa useita erilaisia sovelluksia, jotka jakavat saman toiminnallisen alustan. Aikatauluun liittyviä riskejä otettiin huomioon muun muassa opintojen ja työn osalta. Insinöörityössä valmistu-

neen palvelinohjelmiston tarkoitus ei ole olla tämän opinnäytetyön julkaisun aikana valmis kokonaisuus. Toteutetun palvelinsovelluksen avulla tarkasteltiin palvelukonseptiin liittyviä riskejä sekä teknologioiden että niihin liittyvän ohjelmointityön myötä.

Aikataulullisia riskejä huomioitiin aloituksessa myös siten, että teknologiavalintoihin kohdistuvia ratkaisuja ei haluttu tehdä liian nopeasti. Tällä tavalla pystyttiin teknologiaa korostavien toteutuksien myötä arvioida ajan ja haasteellisuuden näkökulmasta palvelinsovelluksen mahdollisia liiketoiminnallisia kehityskustannuksia. Tätä arviointimenetelmää käyttämällä voitiin todeta, että työssä valmistuneet komponentit edustavat konseptia, ei niinkään valmista kokonaisuutta. Palvelinalustan teknologioiden jatkokehitykselle voitiin näin ollen myös kokemusperäisesti arvioida tulevia kehityskustannuksia. Usein vastaavasta toteutuksesta käytetään termiä PoC, eli Proof of Concept, jolla pyritään tunnistamaan markkina-alueita sekä konseptiin liittyviä riskejä eri näkökulmista. Ohjelmistokehityksen näkökulmasta tämä antaa mahdollisuuden muotoilla palvelua markkinoiden kysyntää parhaalla mahdollisella tavalla vastaavaksi kokonaisuudeksi.

Teknologisiksi tavoitteiksi työssä valmistuneelle palvelinsovellukselle asetettiin, että toteutuksen tulisi, niin pitkään kuin mahdollista, olla mahdollisimman yleisellä tasolla. Tällä tarkoitetaan, ettei palvelinalustalle toteutettu varsinaista liiketoiminnallista sovelluslogiikkaa. Sovelluksen tuli sisältää valmiudet maksutapahtumien käsittelyyn vähintään yhden rajapinnan välityksellä, tarjota SMS-rajapinnan tuki sekä tuottaa näkymälle, tässä tapauksessa JavaScript -ohjelmointikielellä toteutetulle näkymäsovellukselle, vähintään yksi rajapinta palvelimen kanssa reaaliaikaisen kommunikoinnin saavuttamiseksi.

Työssä suunniteluvaiheessa olennaista oli myös, että tietoturva ja tietyt erityisesti rajapinnan haavoittuvuudet huomioitaisiin ja niitä vahvistettaisiin ratkaisulla, joka takaa palvelimen toimintavalmiuden korkean rasituksen tilanteessa. Tätä varten hahmotelmaksi muodostui eräänlainen dynaaminen muisti, jolla voitaisiin tarkistaa aktiivisia rajapinnan käyttäjiä tutkimalla rajapintakutsun mukana lähetettyjä 32 merkistä koostuvia merkistöyhdistelmiä, jotka sallivat rajapintakutsujen lähettämisen palvelimelle. Ratkaisua esitellään tässä työssä neljännessä kappaleessa.

Maksurajapinnaksi valittiin PayPal-palvelu, sillä se tarjoaa monikansallisen tuen ja on helposti lähestyttävä ja kansainvälisesti luotettu maksunvälityspalvelu.

2.4 Tulokset

Työn tuloksena valmistui palvelinalustakonsepti, joka voidaan integroida muihin järjestelmiin hyödyntämällä yleisimpiä järjestelmäintegraatioteknologioita. Palvelinalustalle toteutettiin vaiheittain integraatioita hyödyntämällä JSON-, REST- ja SOAP-teknologioita. Spring 4 -kehys tarjoaa työkalut Java-kehittäjälle luoda palvelinalusta kustannustehokkaasti, ja se tukee nykyaikaisia integraatioteknologioita hyvin. Työn aikana tutustuttiin myös PayPal-maksunvälityspalvelun rajapintaan, johon toteutettiin maksuväylä sandbox-ympäristössä.

Palvelimen ja asiakkaan rajapintapalvelua varten kehitettiin ApiToken- ja ApiTokenService-luokissa teknologiaa, jolla voidaan estää muiden kuin tunnistettujen käyttäjien yhdistäminen insinöörityön palvelinalustalle tuotettuun rajapintapalveluun. Toteutuksessa sovellettiin PayPal API:n ajattelumallia palveluun hankittavan avaimen muodossa, jolloin avaimeksi kutsuttu todentamiseen käytetty merkkijono tallennetaan dynaamiseen muistiin. Tällöin palvelinalustalla voidaan dynaamisesti määritellä sallittuja yhteyksiä ja niiden määriä ja määrittää asiakastieto, sillä avain on kytketty asiakastietoon dynaamisessa muistissa.

3 Palvelinalusta

3.1 Yleiskuvaus teknologiavalinnoista

Kehitettävän palvelimen teknologiavalinnoissa päädyttiin käyttämään selkeärakenteista ja nopeaan käyttöönottoon soveltuvaa Java-ohjelmointikieltä. Kielen versioksi valittiin 8, joka edustaa myös viimeisintä versiostandardia Java-kehityksessä. HTTP(S)-palvelimen alustaksi valittiin Jetty-palvelinalustan versio 9. Koko palvelinsovellus rakennettiin Jetty-palvelimelle.

Syyt Jetty-alustan valintaan olivat sen yksinkertaistettu rakenne ja projektin tavoitteena ollut nopea tuotantovaiheen käyttöönotto. Jetty-palvelin voidaan kääriä JAR- eli Java Archive-pakettiin ja suorittaa yhdellä komennolla. Jetty-palvelinalusta tukee Servlet 3.1 API -standardia, mutta Spring 4 tukee ainoastaan versiota 3.0. Jettyn sisäänrakennettuja

ominaisuuksia hyödynnetään Spring Framework 4 -version kautta. Spring-kehyksen lähtökohtainen tarkoitus on yksinkertaistaa JavaEE-standardin mukaisia kehitysvaiheita ja tietyin osin paketoida valmiiksi ja nopeasti käyttöönotettavia web-palvelinsovelluksia. Tietokannan hallintaan käytetään PostgreSQL-tietokantakehystä. Se on helposti lähestyttävä ja tarjoaa paljon helposti käyttöönotettavia ja tuotantoteknisestä näkökulmasta hyödyllisiä ominaisuuksia. Eräänä ominaisuutena mainittakoon tietokantareplikaatio, joka tarkoittaa tietokantaan tallennetun datan kopiointia lähestulkoon reaaliajassa toiseen sijaintiin.

Ohjelmakoodin entiteettien, eli tallennettavien objektien, hallitsemiseen käytetään Hibernate 4 -sovelluskehystä, joka yksinkertaistaa merkittävästi mutkikkaita tietokantaoperaatioita ohjelmistokehityksen näkökulmasta. Kehittämisen aputyökaluna käytetään myös ohjelmistoprojektin Maven-konfiguraationhallintaa, jolla ohjelmiston rakennetta ja riippuvuuksia voidaan hallita keskitetysti. Sovelluskehityksessä apukirjastoina käytetään JSON-datan ja Java-objektien muunnoksien hallintaa helpottavaa Jackson 2 -kirjastoa ja Googlen Guava-kirjastoa, jonka avulla voidaan tehdä muuttujien ja objektien tarkistuksia. Siinä myös monet Java-sovelluskehitykselle tyypilliset ongelmat on ratkaistu standardinomaisesti.

Guavan tarkoitus on tuottaa yksinkertaistettua ja luettavampaa ohjelmakoodia. Ohjelmakoodin paisuessa satoihin luokkiin selkeät käytännöt auttavat ohjelmistokehittäjiä tunnistamaan ongelman ja siihen sovellettavaa ratkaisutapaa. Useat edellä mainitut ohjelmakirjastot hyödyntävät Apache Foundationin standardikirjastoja jotka kuuluvat myös tämän projektin konfiguraatioihin riippuvuuksina, mutta näihin kirjastoihin ei tässä työssä syvennytä tarkemmin. Muita kirjastoja esitellään siten, kuin ne liittyät esiteltävään ohjelmakoodiin. Kaikki projektin edellä mainitut kirjastot on myös määriteltiin Maven-konfiguraatiotiedostossa pom.xml:ssa.

Palvelinsovelluksen ensisijainen tarkoitus ohjelmistoteknisestä näkökulmasta on tuottaa dataa ja ohjata näkymän tuottavaa erillistä sovellusalustaa. Näkymäpalvelimen ja taustapalvelimen hajauttaminen vähentää tiettyjä tietoturvariskejä, joita web-palveluntuotantoon liittyy. Tässä työssä kehitettyjen palvelinohjelmiston ominaisuuksien demonstraatiot perustuvat palvelimen tuottamaan staattiseen HTML-sivuun ja JavaScript-kielellä siihen kehitettyihin kehyskirjastoihin, joihin tässä opinnäytetyössä ei perehdytä tarkemmin. Tärkeimmiksi näkymäteknologioiksi jatkokehittämisen kannalta voidaan mainita jQuery-, Knockout.js- ja moment.js-kirjastot. Näistä kehyskirjastoista erityisesti Knockout tarjoaa

JSON-datan mallintamiseen soveltuvan dynaamisesti päivittyvän tietomallin yhdessä käyttäjälle näkyvien komponenttien kanssa. Merkittävää on huomata, että Knockout.js:n avulla tuotetun muuttujan rakenne on itsessään sen tarkkailtavien kenttien koostama JSON-objekti. HTML-sivujen tyylittelyyn käytetään CSS3-kieltä, joka on yleinen HTM-komponenttien muotoiluun käytetty kieli.

Projektin hallinnoinnin osalta ohjelmiston versionhallintaan käytettiin Git-versiohallintaa. Git soveltuu erinomaisesti ohjelmakoodin eri kehityshaarojen kehittämiseen, joka myös on olennainen osa työtä silmälläpitäen tämän opinnäytetyön tarkoitusta. Git-repositorya hallinnoidaan ohjelmistokehityksessä IT-alalla vankan jalansijan saaneen Atlassian tuoteperheen ohjelmistojen avulla. Projektin hallintaan käytetään Atlassian Jira-, Confluence- ja Bitbucket-tuotteita.

Bitbucket, aiemmalta nimeltään Stash, on erityisesti Git-aputyökalu, joka helpottaa huomattavasti laajemman ja graafisen esitysmuodon ansiosta tarkastelemaan ohjelmakoodia ja helpottaa projektin kehityshaarojen hallintaa. Jiran rooli tässä projektissa yleisistä käytännöistä poiketen on seurata projektiin tehtäviä muutoksia yhdessä Bitbucketin kanssa. Bitbucketiin merkittävien Jira-tehtävien avulla versionhallinta helpottuu, sillä versioihin tehdyt muutokset ja muutostarpeet ovat kuvattuna Jira-tehtävinä.

Atlassian Confluencen rooli projektissa oli hallita ja säilyttää tietoa. Tiedon hallinta tarkoittaa, että tekniset ja alustaan liittyvät konfiguraatiot, jotka usein ovat mutkikkaita, olisivat kuvattuna tulevaisuuden varalta. Tällöin vältytään toistamasta työtä ja hukkaamasta aikaa asioihin, jotka eivät ole päivittäisiä ongelmia ja jotka ovat saattaneet kadota kehittäjän lähimuistista.

3.2 Riippuvuuksien hallintatyökalu Maven

Maven on Java-ohjelmakoodin käännöstyökalu, jonka avulla voidaan keskitetysti hallita ohjelmaan lisättäviä ulkopuolisia kirjastoja, jotka halutaan ohjelmakoodin saataville käännöksen aikana. Näitä ulkopuolisia kirjastoja kutsutaan yleisemmin riippuvuuksiksi. Maven on erityisesti suunniteltu modulaarisiin ohjelmakokonaisuuksiin, joita saattaa ohjelmistossa olla kymmeniä tai joissain jopa satoja.

Aiemmin ohjelmistokehityksessä riippuvuudet ladattiin versiohallinnasta, jolloin ei ollut tarkkaa tietoa, mitä riippuvuuksia moduuleissa saattoi olla. Kaikki sovelluksessa hyödynnettävät moduulit ja ulkopuoliset kirjastot ladataan Mavenin riippuvuuksienhallinnasta. Sama työkalu myös kääntää koodin ja luo projektista esimerkiksi komentotulkista käynnistettävän JAR-tiedoston. Projektin määritteleminen Mavenin projektitiedostoon pom.xml:aan on yksinkertaisimillaan suoraviivainen prosessi. Ensin määritellään projektin elementeissä käytettävä maven nimiavaruus, jonka jälkeen ryhmä-, artikkeli- ja versiotunnus sekä tiedostomuoto, johon käännetty ohjelma halutaan kuva 1. Tässä työssä muodoksi on valittu JAR-paketti, sillä projekti haluttiin paketoida täysin kyseisen rakenteen sisälle. Tämä yksinkertaisti asennusprosessia, joka oli myös määritelty palvelinalustakonseptin tavoiteissa. Projektin nimi- ja muotomääritysten jälkeen määrittelytiedostossa voitiin määritellä, mitä riippuvuuksia projektiin haluttiin sisällyttää.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>lycom.proxy.server</groupId>
    <artifactId>lycom-proxy-server</artifactId>
    <version>0.1</version>
    <packaging>jar</packaging>

    <name>Spring MVC Embedded Jetty</name>

    <properties>
        <hibernate.version>4.2.15.Final</hibernate.version>
        <spring.version>4.2.2.RELEASE</spring.version>
        <jetty.version>9.0.6.v20130930</jetty.version>
        <slf4j.version>1.7.5</slf4j.version>
        <jackson.version>2.6.3</jackson.version>
        <paypal.version>LATEST</paypal.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    </properties>

    <dependencies>

        <!-- Jetty embedded -->
        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-servlet</artifactId>
            <version>${jetty.version}</version>
        </dependency>

        <!-- Spring -->

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>commons-logging</artifactId>
                    <groupId>commons-logging</groupId>
                </exclusion>
            </exclusions>
        </dependency>
```

Kuva 1: Maven-projektin konfiguraatio.

3.3 Spring-sovelluskehys

Spring-sovelluskehys koostuu noin 20 eri moduulista, jotka ryhmitellään yleisesti ottaen kahdeksaan kategoriaan. Tavoitteena valtavassa sovelluskehyksessä on tarjota kehittäjälle valmiiksi paketoituja parhaiden käytäntöjen mukaisia ohjelmistokehityksen ratkaisuja sovelluskomponenttien myötä. Toteutetussa palvelinalustassa hyödynnettiin Web-, Core Container- ja Data Access/Integration -ryhmiä. Jatkokehityksen näkökulmasta myös testaukseen tarkoitettu moduuliryhmä oli tarkoitus ottaa käyttöön myöhemmässä vaiheessa todentamaan liiketoiminnallisten komponenttien virheetön toiminta.

Kukin Springin seitsemästä kategoriasta käsittää vielä useita yksittäisiä moduuleita. Spring kehyksen valinnasta mielenkiintoisen tekee se, että jopa yksittäistä moduulia tai kategoriaa voidaan käyttää lähes minkä tahansa sovelluksen kehityksessä, eikä kaikkia Spring kehyksen ominaisuuksia ole välttämätöntä käyttää, sillä moduulit ovat toisistaan riippumattomia. Kehittäjä voi valita esimerkiksi moduulin jostain kategoriasta ja hyödyntää näitä yksittäisen moduulin tarjoamia ominaisuuksi sovelluksen kehittämisessä.

Tässä työssä tarkastellaan sovelluskehittämisen näkökulmasta Core Container -ryhmän spring-core-, spring-webmvc-moduuleja ja Spring Security -ryhmän spring-security-crypto-moduulia. Springin tuoma merkittävin rooli työssä kehitettävälle sovellukselle on tarjota valmiiksi hyödynnettäviä JavaEE-kehittämisen työkaluja kehittäjän käyttöön ja tuoda sekä sovellus- ja bisneslogiikan eriyttäviä työkaluja ja annotaatioita, jotka helpottavat sovelluksen luettavuutta myöhempää kehitystyötä ajatellen. [10.]

Työssä kehitettävä palvelinalustasovellus perustuu Spring 4 -versioon, joka tukee Java 8:n uusimpia ominaisuuksia ja mullistaa aikaisemman palvelinsovelluksen konfiguraatiomallin, jossa annotaatiokonfiguraation myötä palvelinsovelluksen vaatimat konfiguraatiot voidaan toteuttaa ohjelmakoodissa Javan annotaatio-ominaisuuksien avulla. Javan annotaatioteknologiaa ei käsitellä tässä työssä tarkemmin.

3.4 HTTP- ja HTTPS-pyyntöjen käsittely

Sovelluspalvelimen alustateknologiaksi opinnäytetyössä valittiin Jetty 9 -palvelin. Jetty tukee web-kehityksessä erityisesti HTTP(S)-taustapalvelimelle olennaisia ominaisuuksia. Jettyn valinta projektiin oli tuloksena tavoitteisiin asetettujen määritelmien mukaisesti olla mahdollisimman helposti jaeltava kokonaisuus.

Jettyllä on erittäin pitkä historia palvelinalustana, ja se on kehittynyt pitkän kehitystyön tuloksena tehokkaaksi alustaksi, joka kykenee kilpailemaan suorituskyvyssä sovelluspalvelinten huipulla. Palvelin voitiin konfiguroida käynnistettäväksi yhdellä komennolla, kun muut tavanomaiset sovelluksen toiminnalle tarpeelliset taustapalvelut ovat käynnistettynä.

Web-lähtöisessä ohjelmistokehityksessä on perinteisesti käytetty Tomcat-, JBoss-, tai GlassFish-sovelluspalvelinta, joka käsittelee JAR- sekä WAR-paketteja ja pitää huolta sovellusten käynnistyskonfiguraatioista. Näihin sovelluspalvelimiin on mahdollista asentaa useampi tuotettu palvelu tietyin konfiguraatioin. Tämä on kuitenkin hieman mutkikasta eikä myöskään usein kovin käytännöllistä.

3.5 Hibernate-sovelluskehys ja annotaatiot

Javan standardinmukainen oliotallennus on määritelty toteutettavan erään standardin mukaisesti JPA- eli ”Java Persistence Api” -käyttöliittymän toteuttavien luokkien avulla. Kuten muilla tiedon tai olioiden tallennuskehyksillä Javassa myös Hibernatella on JPA:n mukainen implementaatio, jota työssä hyödynnetään. Annotaatiot ovat itsessään JPA:n standardikirjaston annotaatioita, mutta Hibernate on määrittänyt tämän spesifikaation mukaiset toiminnallisuudet kehyksessään.

JPA-spesifikaation mukaisilla annotaatioilla ohjeistetaan Hibernate-kehykselle, kuinka olio ja mahdolliset oloiden riippuvuudet muihin olioihin tulisi tallentaa tietokantaan. Nämä tietokantaan tallennettavat POJO-objektit eli ”Plain Old Java Object”, tutummin entiteetit, ovat rakenteellisesti yksinkertaisia Java-luokkia, joissa attribuutit on merkitty JPA-spesifikaation mukaisin annotaatioin. Annotaatioilla voidaan ohjeistaa Hibernatea käsittelemään objektien välisiä suhteita relaatiomallin mukaisesti. Merkittävimpiä työssä käytettyjä JPA-annotaatioita ovat @Entity, @Table, @Id, @OneToOne, @OneToMany ja @ManyToOne. Erikoistapauksena entiteetille voidan myös määrittää @ManyToMany -relaatio.

Entiteettejä eli tallennettavia POJO-olioita tulee JPA-spesifikaation mukaisesti määrittää @Entity-annotaatioilla. Tämä määritys tulee olla määritettynä luokan otsikossa, jotta Hibernate hyväksyy olion tallennettavaksi tietokantaan. Tyypillisesti Hibernaten konfiguraatioissa on määritelty pakettirakenne eli polku, josta tallennettavaksi tarkoitetut entiteetit Hibernate tunnistaa annotaation avulla. Työssä kehitettävässä palvelinalustassa Hibernate konfiguroidaan osittain sovelluksessa, minkä vuoksi entiteetit täytyy määritellä myös ohjelmallisesti Hibernaten tunnistettavaksi. Tämä on mahdollista Javan reflektion, eli ohjelman suorituksen aikaisen luokkien tarkastelun avulla, jota työssä kehitettävässä sovelluksessa hyödynnettiin.

Entiteettien ohjelmallista lukua varten on luotu luokka, joka tarjoaa parametrina saadun pakettirakenteen tuloksena kaikki Java-luokat Hibernaten kyseisten luokkien annotaatioiden tutkimiseen (kuva 2). Olio voidaan myös osoittaa muuhun kuin nimeämiskäytäntöjen mukaiseen tietokannan tauluun. Tätä voidaan ohjata asettamalla entiteetille @Table-annotaatio, jolle parametrina annetaan haluttu tietokannan taulun nimi.

```
1  package lycom.proxy.server.entity;
2
3  import javax.persistence.Entity;
15
16 @Entity
17 @Table(name = "sms_message")
18 public class SmsMessageEntity implements SmsMessage{
19
20     @Enumerated(EnumType.STRING)
21     private Source source;
22
23     @Enumerated(EnumType.STRING)
24     private State state;
25
26     private String message;
27
28     private DateTime sent;
29
30     private Integer customerRelation;
31
32     @Id
33     @GeneratedValue(strategy = GenerationType.IDENTITY)
34     private Integer id;
35
36     private DateTime created;
37
38     private String numberFrom;
39
40     private String numberTo;
```

Kuva 2: Hibernaten avulla tallennettava entiteetti.

Jälkimmäiset listatut annotaatiot määrittävät erityisesti tietokannan relaatiomallin oliorelaatioita. Id määrittää tietokannan ja olion yksilöllisen tunnisteen, eli taulun pääavaimen. Muut määrittävät relaatiomallin mukaiset 1-1-, 1-n-, n-1- ja m-n-relaatiosuhteet. Annotaatioiden nimeämiskäytäntö kuvaa hyvin tietokantarelaatioiden määritelmiä. Esimerkiksi @ManyToOne viittaa n-1:n mukaiseen tietokantarelaatioon. Tässä työssä ei käsitellä tarkemmin olioiden välisten relaatioiden mallintamista.

3.5.1 Relaatiomalli

Työssä toteutettavassa sovelluksessa hyödynnetään relaatiotietokantamallinnusta. Mallinnuksen avulla voidaan vähentää ohjelmasuunnittelussa, ohjelmoinnissa ja käytön aikana tapahtuvia sovelluksen virhetilojen esiintymisiä. Kun pää- ja mahdolliset vierasavaimet on määritelty oliolle oikein relaatiomallin mukaisesti PostgreSQL-tietokantamoottorille, ohjelmakoodin loogiset virheet kannan kyselyissä aiheuttavat Hibernatessa virheenkäsittelyn ja raportoinnin JVM-, eli Java Virtual Machine-virheenkäsittelijälle. Työssä poiketaan parhaan käytännön mukaisesti hieman relaatiomallista siten, että useimmissa tapauksissa kullakin entiteetillä on oma erityinen id-kenttä, joka on tietokannan juokseva numero ja erottaa objektin muista taulun objekteista. Tämä tapa on tarkoituksenmukainen, kun objektilla halutaan olevan jokin yhteinen yleinen avaimen muodon määritelmä. Yleisesti numerointi helpottaa myös tietokannan oliorakenteiden hahmottamista, kun kyseessä ovat monimutkaiset relaatiot olioiden välillä.

3.5.2 Konfigurointi

Hibernaten riippuvuudet on kuvattu Maven-rakenteessa. Tärkeimmät kirjastot ovat hibernate-core ja hibernate-entitymanager. Hibernate konfiguroidaan erityisellä konfiguraatiotiedostolla, joka nimeämiskäytännön mukaisesti on ”hibernate.cfg.xml”. Tämän tiedoston kautta määritellään sovelluksessa useita tietokantayhteyksiin liittyviä asetuksia. Huomioitavaa on, että työssä käytettyjen tavanomaisten konfiguraatioiden lisäksi mukana on myös C3PO-yhteysaltaan määrittelyt, sekä adapterikirjasto huomattavasti monipuolisemman joda-kirjaston tarjoamien aikamuuttujien tallennukseen.

Tyypillinen ORM-standardien mukainen konfiguraatiotiedosto ”queries.hbm.xml” sisältää valmiiksi nimettyjä hakuja tietokantaan. Valmiiksi nimetyt SQL-lauseet parantavat ohjelmakoodin luettavuutta ja mahdollistavat myös keskitetyn muutoin ohjelmakoodista irrallaan olevan SQL-lauseiden hallinnan.

4 Integraatiot

4.1 Toteutuksen yleiskuvaus

Palvelinintegraatioiden tiedonsiirto toteutettiin insinöörityössä pääasiassa JSON-tiedonsiirtona. Tätä varten on olemassa hyviä JavaScript-laajennuksia, joilla JSON-objektien mallinnus datasta on mielekästä. Esimerkkikirjastona voidaan mainita Knockout.js-kirjasto, joka muuttaa tyypillisen JavaScript-ohjelmakoodin rakenteen hallitummaksi. Knockoutin rooli on kuvata data helposti lähestyttävän mallin myötä, jossa Knockout-kehys pitää huolen näkymän ja mallin välisistä muutoksista. Jotkin rajapinta-palvelut vaativat perinteisemmän tavan, jossa palvelimen tulee lähettää HTTP-protokollan GET tai POST -pyyntö ja tarjota jokin tietty kanava takaisin rajapintaan saapuville vastauksille. Eräänä esimerkkinä perinteistä mallia noudattava tiedonsiirrosta olisi esimerkiksi Luottokunnan verkkomaksurajapinta, jossa rajapintakutsuja voidaan lähettää http-protokollan POST-pyynnön parametreissa.

Tässä opinnäytetyössä toteutettiin PayPal API Sandbox -integraatio, joka on toteutukseltaan mutkikkaampi kuin Luottokunnan verkkomaksuintegraatio [4]. Sandbox-integraatio tarkoittaa testausympäristöä, joka vastaa tuotantoympäristöä mutta jossa tehdyt transaktiot ainoastaan mukailevat teknisesti aitoa maksuliikennettä palvelussa. PayPal-integraation valitsemiseen vaikutti sen kansainvälisyys ja edullinen hinnoittelu, jossa transaktiosta peritään tariffi eli maksunvälityksestä peritty verrattain pienehkö korvaus. PayPal on muutoin perusominaisuksiltaan ilmainen sekä tavalliselle kuluttajalle että yritystoimintaa varten. [13.]

Sovelluksessa keskeisessä roolissa olevaa JSON-muotoista tiedonsiirtoa voidaan hyödyntää esimerkiksi Quriirin tai Zoner.fi:n tarjoaman tekstiviestinvälityksen API-kuvauksen mukaisen rajapinnan toteuttavalla JSON-muotoisella mallinnuksella Knockout.js-data-objektin avulla. Näin sekä palveluun saapuvat että lähtevät viestit voitaisiin mallintaa reaaliaikaisena. Knockout hoitaa taustalla sille linkitetyn HTML-elementin päivityksen data-objektin tilanmuutosten mukaisesti. Näkymässä siis voidaan tällöin näyttää käyttäjälle lähetetyt viestit lähestulkoon reaaliajasssa. Työtä varten jätettiin tarjouspyyntö palvelun-tarjoajille, mutta testiviestit alittivat merkittävästi testaukseen tarvittavat viestimäärät, jotka oli arvioitu noin 50 kappaleen suuruisiksi. Quriiri tarjoaa sekä HTTP- että JSON-rajapinnat, mutta Zoner.fi API kuvaus on julkaistu vain HTTP-pyynnöille. Palvelinalustalle

toteutettiin kuitenkin entiteetti, jota käytetään viestien tallentamiseen tietokantaan kehitettäessä SMS-rajapinnan implementaatio.

4.2 Rajapinnat

Rajapinnoilla ohjelmistotekniikassa tarkoitetaan sitä ohjelman tai palvelun osaa, joka on luotu avoimeksi sovitun määritelmän mukaisesti. Rajapintoja voidaan tarjota teknologiakohtaisesti, jolloin esimerkiksi ohjelmointikieleen on sisäänrakennettu tuki siirtää ja vastaanottaa dataa toisen erillään olevan palvelimen kanssa. Ohjelmointikieleen sidottu rajapintateknologia on jo pitkään seurannut sivusta sovelluksen tai yleisesti ottaen objektin tilaa tekstimuodossa esittävien teknologioiden avulla.

Tyypillisiä objektin tilaa esittäviä teknologioita ovat JSON ja SOAP. Vaikka nämä kaksi ovat toteutukseltaan hyvin erilaisia, ne esittävät kumpikin yksinkertaisella tavalla selkokielisesti sovelluksen tai objektin tilan. Nämä kaksi teknologiaa ovat yleistyneet ylitse muiden yksinkertaisesta syystä: molemmille on ominaista käyttää HTTP-protokollaa tiedonsiirtoväylänä.

Toteutukseltaan JSON:n ja SOAP:n välillä on kuitenkin hyvin paljon eroavaisuuksia. SOAP:lla tehty toteutus on monimutkaisempi toteuttaa, mutta tällä tavalla lähetetyn datan rakenne hallitumpaa kuin REST-teknologian integraatioiden. JSON:ssa objektin rakenteelle ei aseteta varsinaista tarkistusta eikä rajapintakutsussa ole erikseen määritetty, minkälaisen kutsun yhteyttä ottava osapuoli voi suorittaa

JSON-rajapintaan voidaan tuottaa dataa, jota ei osata käsitellä oikealla tavalla. Tästä syystä JSON-rajapintaa voidaan pitää heikomman tietoturvan piirteet omaavana, kun taas SOAP:ssa nämä yhteyspyynnöt on tarkasti määriteltynä. Lisää varmuutta ja turvallisuutta takaa myös SOAP:ssa määriteltävän WSDL:n mukaisen määritelmän toteutuminen asiakaspuolella. Asiakkaan lähettämä SOAP-pyyntö tarkistetaan vastaanottavalla palvelimella, ja mikäli se ei vastaa WSDL:n mukaisia rakenteellisia vaatimuksia, pyyntö hylätään. Näin ollen SOAP-teknologialla voidaan varmistaa palveluun saapuvat ja siitä lähtevät rajapintakutsut. [6.]

4.2.1 REST-teknologia

REST eli ”Representational State Transfer” tarkoittaa vapaasti suomennettuna ohjelman tai objektin tilan kuvausta. REST ei ole ohjelmointikieli eikä standarditoteutus jollain tietyllä ohjelmointikielellä. Sen kehitti muun muassa WWW-standardin kehittäjätiimissä työskennellyt Roy Fielding [7]. Huomattavaa on, että REST on konsepti, jonka ohjeistusta tulisi seurata niin hyvin kuin mahdollista. Tasoja on teoriatasolla kolme, ja ne esittävät konseptin toteutumista seuraavin lyhyin määritelmin:

Taso 0:
Ei REST-pohjautuva ollenkaan

Taso 1:
Web-kutsut URL:n resurssikutsuina

Taso 2.
Resurssikutsujen määrittämiä resursseja kutsutaan HTTP-standardin mukaisin verbein. Muita HTTP-protokollan verbejä ovat muun muassa PUT ja DELETE. Tällöin myös oletetaan, että edellä mainitut HTTP-pyynnöt palauttavat pyynnön tilakoodin vastauksessaan.

Taso 3:
Englanninkielinen termi HATEOAS eli ”Hypermedia Text As the Engine Of Application State” kuvaa tätä REST-konseptin kolmatta tasoa. Sillä tarkoitetaan, että kahden aiemman tason toteuttava REST-rajapinta tulisi kuvata myös kaikki kyseiseen resurssiin liittyvät muut resurssikutsut. Tämä käsittää itsessään REST-konseptin kolmatta tasoa noudattavan palvelun itsensä dokumentaation suoraan resurssipyynnön vastauksen yhteydessä. Kolmannen tason palvelu informoi resurssikutsun pyynnön perusteella vastausrungossaan, miten pyyntöön linkittyviä resursseja on mahdollista käyttää. [6.]

Sovelluksessa REST-konseptia pyrittiin noudattamaan mahdollisimman hyvin. Sovellus asettuu REST-tasoissa ensimmäisen ja toisen tason välille. Varsinaisesti muita kuin GET- ja POST-pyyntöjä ei palvelimelta toistaiseksi työssä toteutettavassa palvelimesta lähetetä. Spring-kehys tarjoaa mielekkään tuen REST-konseptin määritelmän toiselle tasolle. Vastaukset voidaan paketoida tietyiksi ”ResponseEntity”-luokkien olioiksi, jotka palauttavat tyyppimääritelmän mukaisen tyypin HTTP-vastauksessa, joka sisältää kyseisen HTTP-pyynnön tilan ja tästä riippuen myös vastauksen rungon. Työssä kehitettävässä

palvelinalustassa käytetään tätä Spring-kehyksen ominaisuutta. Sen avulla työn palvelinsovellus tavoittaa REST-konseptin toisen tason. Työn sovelluksessa hyödynnetään REST-konseptia jossa onnistuneessa resurssikutsussa sovellus tarjoaa näkymäsovellukselle tietoa JSON-muodossa. Kukin resurssikutsu on määritelty omaksi polukseen (kuva 5).

```
159    @RequestMapping(value = "/smsmessage", produces = "application/json")
160    public ResponseEntity<String> smsMessage(@RequestParam(required = true) String data) {
161        // Get beans by config
162        ApplicationContext context = new AnnotationConfigApplicationContext(BeanConfig.class);
163
164        System.out.println("/smsmessage -endpoint call");
165
166            ObjectMapper objMapper = new ObjectMapper();
167            try {
168                // Return view with Views-filter
169                SmsMessage smsMessage = (SmsMessage) objMapper.readerFor(SmsMessage.class);
170
171                String smsJSON = objMapper.writeValueAsString(smsMessage);
172                return ResponseEntity.ok(smsJSON);
173            } catch (JsonProcessingException e1) {
174                // No Catch
175            }
176        return ResponseEntity.ok(null);
177    }
```

Kuva 3: REST-pyynnön käsittelevä metodi sovelluksessa.

4.2.2 JSON-teknologia

Määritelmänsä mukaan JavaScript Object Notation on JavaScript-ohjelmointikielessä objektin ominaisuuksien viittausmenetelmä. Lyhyeltä nimitykseltään JSON on standardoitu menetelmä esittää objektin sisältämä data kenttä-arvoparien avulla. JSON kuvaa selkokielisesti JavaScript-objektin tilaa. Se on yleisesti suosittu menetelmä web-kehityksessä siirtää dataa palvelimen ja asiakasohjelman välillä. Kuvantamismallia verrataan usein pitkään web-palveluiden rajapintateknologioita hallinneen ja teknologialtaan useaan eri palveluun hyvin skaalautuneen kilpailijateknologia XML (Extensive Markup Language) -pohjaiseen datan välitykseen, joka tunnetaan kenties paremmin rajapintapalveluissa SOAP-toteutuksina.

Jon Bock ja David Aktary vastasivat artikkelissa JSON:n ja XML:n eroavaisuuksista, tyypillisistä sovelluskohteista ja tulevaisuuden näkymistä kummankin teknologian osalta. Artikkeli käsitteli asiantuntijoiden näkemystä teknologioiden eroavaisuuksista ja sovelluskohteista. Bockin mukaan SOAP-integraatiot ovat yleisesti ottaen vaativampia ja si-

sältävät enemmän rakenteellisia haasteita kuin REST-integraatioiden toteutukset yleisesti ottaen. Eroavaisuudella tarkoitetaan SOAP- ja REST-teknologioiden hyödyntämisestä erilaisissa projekteissa, joista REST-integraation toteuttaminen on huomattavasti nopeampaa kuin vastaavan rakenteellisesti monimutkaisemman SOAP-integraation. Aktaryn mukaan XML mahdollistaa rakenteellisemman ja eheämmän kehyksen monimutkaiseksi datan esitysmuodoksi. Hän lisää, että projektimuodolla on merkitystä teknologiavalintaan, josta esimerkkinä mainitsee vesiputous- ja agile-projektinhallintamenetelmät. Kun toisessa olennainen osa on tuottaa ohjelmakoodia tuotantoon jatkuvasti, lähestulkoon päivittäin, on tällöin JSON luontainen ratkaisu datakehyksen mallinnukseen joustavuudessaan.Toisaalta kun halutaan varmistaa sovelluksen toiminta sekä rajapintaan saapuvien viestien tyypit ja datan muoto, ovat tällöin XML-teknologiaan perustuvat rajapintateknologiat hyvä ratkaisu.

JSON-objektin mallintaminen juontaa juurensa JavaScript-kielen tapaan esittää objektin sisältämää tietoa. JSON-data on kuin mitä tahansa muuta tekstipohjaista dataa, mutta sillä on rakenne, joka on standardoitu JavaScript-kielessä. Useat alustat tukevat JSON-tiedon esitysmuotoa, josta hyvänä esimerkkinä HTTP-standardi, joka määrittelee JSON:n erääksi tavaksi lähettää ja vastaanottaa dataa. HTTP-standardissa tämä tieto määritellän pyynnön ja vastauksen otsikkotietona.

JSON:n suosion syy on Tim Perryn mukaan [6] selkeä, sillä datan kuvantamismalli on helppo ymmärtää ja toteuttaa. Artikkelissa myös otetaan huomioon JavaScriptin asema ja suosio nykyisessä ohjelmistokehityksessä, jossa JavaScript on erityisesti web-kehityksessä näkymäteknologian peruskomponentti ja JSON taas läheinen osa kyseistä teknologiaa. [6.]

Tässä työssä tarkastellaan toteutusprojektia, joka hyödyntää JSON-mallinnusteknologiaa osana web-palveluita. Teknologiaa on sovellettu tutussa kontekstissa näkymäteknologiana, mutta myös integraatiorajapinnan datan välityksessä. Näkymäteknologian ja palvelimen välillä JSON yksinkertaistaa työvaiheita, joissa kohtuullisen yksinkertaista dataa käsitellään molemmin puolin sekä asiakasohjelmassa että palvelimella. Tässä vaiheessa myös voidaan huomata JSON:n heikkous. Saapuvan datan mallia ei varsinaisesti tarkisteta tai tiedetä etukäteen, sillä JSON ei ota kantaa missään vaiheessa datan muotoon tai datan tyypitykseen. Tätä varten dataan, sen tyyppiin ja sisältöön tulee kiinnittää palvelukehityksessä huomiota, erityisesti jos palvelu on tarkoitettu julkisesti saataville.

Työssä kehitetyssä sovelluksessa käytettiin JSON-mallinnusta osana RESTful-konseptin toteuttavaa kolmannen osapuolen web-rajapintapalvelua. Rajapinta toteutettiin PayPalin ja työssä kehitetyn palvelimen välille. Tämä integraatio toteutettiin siten, että työssä kehitettävä palvelinohjelmisto toimi REST-palvelutuotannon näkökulmasta asiakkaan roolissa ja PayPal palvelun tuottajana.

JSON-dataa siirretään ensimmäisen palveluun kirjautumisen jälkeen tulevissa kutsuissa palveluntuottajan määrittämiin palveluosoitteisiin. Näitä palveluosoitteita kutsutaan URL:ksi, eli ”Uniform Resource Locator”:ksi, joilla tyypillisesti tarkoitetaan web-palveluissa tapaa osoittaa jokin tietty resurssipolku, joka vastaa esimerkiksi asiakasohjelman HTTP- ja HTTPS-pyytöihin. PayPal-integraatiota käsitellään erikseen työssä luvussa 4.4.

4.2.3 RESTful-konsepti

RESTful-konsepti kuvaa REST-palveluteknologian takana olevia toteutusohjeistuksia. REST ei ole ohjelmointikieli, vaan se voidaan mieltää tietynlaisena tasojärjestelmänä, jolla kuvataan kuinka hyvin REST-rajapintapalvelu toteuttaa konseptin määrittelemät vaatimukset. RESTful-konseptia kehittämässä on ollut muun muassa WWW-standardin aikoinaan valmistellut Martin Fowler [7]. RESTful palvelun määritelmän kaikki tasot toteuttava palvelu noudattaa siten tietynlaista rajapintalupausta. Tasoja on yhteensä kolme, ja ne on lueteltuna REST:a käsittelevässä luvussa 4.2.1. Näihin aiemmin lueteltuihin tasoihin viitataan tässä luvussa. Erään määritelmän mukaan voidaan sanoa, että mikäli palvelu toteuttaa kaikki luetellut tasot, sitä voidaan kutsua RESTful-rajapintapalveluksi.

Jotta palvelu voisi olla konseptin mukaisesti RESTful-palvelu, sen tulee toteuttaa tasot 1-3. Ensimmäisessä tasossa merkittävää on rajapintapalvelun kutsujen mahdollistaminen pääteosoitteissa, jossa palvelun tuottaja tuottaa tietyn polun osoittamaan resurssin perusteella tietyn muuttumattoman vastauksen. Tämä tarkoittaa esimerkiksi, että palvelu noudattaa HTTP-standardin käsitystä tilattomuudesta ja että WWW-standardin tehokkuuden mahdollistava tiedon väliaikaistallennus olisi mahdollista.

Toisen tason REST-palvelun tulee toteuttaa HTTP-standardin mukaiset metodit GET, POST, HEAD, PUT, DELETE, OPTIONS ja CONNECT. Metodien tulee myös vastata

palvelimelta saapuvilla standardin mukaisilla paluuviestin kuittauksilla pyyntökohtaisesti. Lyhyesti näitä ovat 200-sarjan paluuviestit, jotka kuvaavat onnistunutta pyyntöä ja palvelimen onnistunutta suoritusta. Virheitä kuvaa 400-sarja, jolla palvelin ilmoittaa, että pyyntö oli virheellinen tai että palvelimen prosessoinnissa oli ongelma. Lisäksi olennainen osa vastauksen kuittauskoodauksesta on 500-sarja, jolla ilmoitetaan palvelimen ongelmista prosessoida pyyntö.

Kolmannen tason REST-palvelu on usein toteutuessaan implisiittisesti toteuttanut aiemmin mainitut tasot. Tällä tasolla käytetään usein termiä HATEOAS rajapintapalvelutoteutuksen yhteydessä. HATEOAS-ohjeistus tarkoittaa, että rajapinnan tulee tarjota navigointimahdollisuus rajapintapalvelussa. Navigointimahdollisuudella tarkoitetaan niitä rajapinnan resursseja, jotka liittyvät aiempaan rajapintakutsuun. Tämä toteutetaan esimerkiksi asettamalla parametri ”ref” ja arvo, johon liitetään URL, jolla itsessään viitataan palvelun kuluttajan saatavilla oleviin linkityksiin suoritetun pyynnön jälkeen. [5.]

4.2.4 SOAP-teknologia

SOAP on XML-viesteihin perustuva integraatioteknologia. SOAP-teknologialla voidaan välittää dataa alustariippumattomasti kahden palvelimen välillä. Aiemmin SOAP määriteltiin lyhenteenä ”Simple Object Access Protocol” sanoista. SOAP:n standardointiehdotuksen 1.2-versiosta lähtien SOAP-teknologian nimi on määritetty nykyisessä muodossaan. SOAP hyödyntää HTTP-protokollaa tiedonsiirrossa, mutta myös muita protokollia voidaan käyttää tiedonsiirtokerroksena. HTTP-protokollan hyödyntäminen on ollut merkittävä etu kahden fyysisesti erillään sijaitsevan palvelimen välisenä integraatioissa. Usein palomuurit sallivat oletuksena 80-portin, joka on HTTP:n standardi tiedonsiirtoväylä. SOAP-implementaatioiden kautta erillään olevat palvelimet voivat kutsua rajapintaan määritettyjä metodeita.

SOAP perustuu XML-viesteihin, joilla määritetään kutsuttava metodi ja sille mahdolliset parametrit. XML-viesteille luodaan tietty viestin muodon määritelmä WSDL eli Web Service Description Language, jonka määritelmät molemmat integraation osapuolet tulee toteuttaa omissa toteutuksissaan. Näiden viestien lähetykseen liittyvään tekniseen toteutukseen ei tässä työssä perehdytä tarkemmin muilta osin kuin työssä kehitettävän palvelinalustan integraation osalta.

Spring-kehys tukee SOAP-integraatioita standarditoteutuksin. Tietyt toistuvat integraation toteutusvaiheet voidaan jättää standarditoteutuksen tehtäväksi. Työssä on hyödynnetty Spring:n tukemaa SOAP:n standarditoteutuksen implementaatiota Contract first-implementaationa. Toinen tapa tuottaa SOAP-rajapinta on luoda WSDL kehitetystä ohjelmakoodista, mutta tämä on ongelmallista XML:n ja tiettyjen Javan kokoelmien välillä. Tästä syystä työssä on implementoitu ensin mainittu tapa toteuttaa SOAP-rajapintapalvelu. [5.]

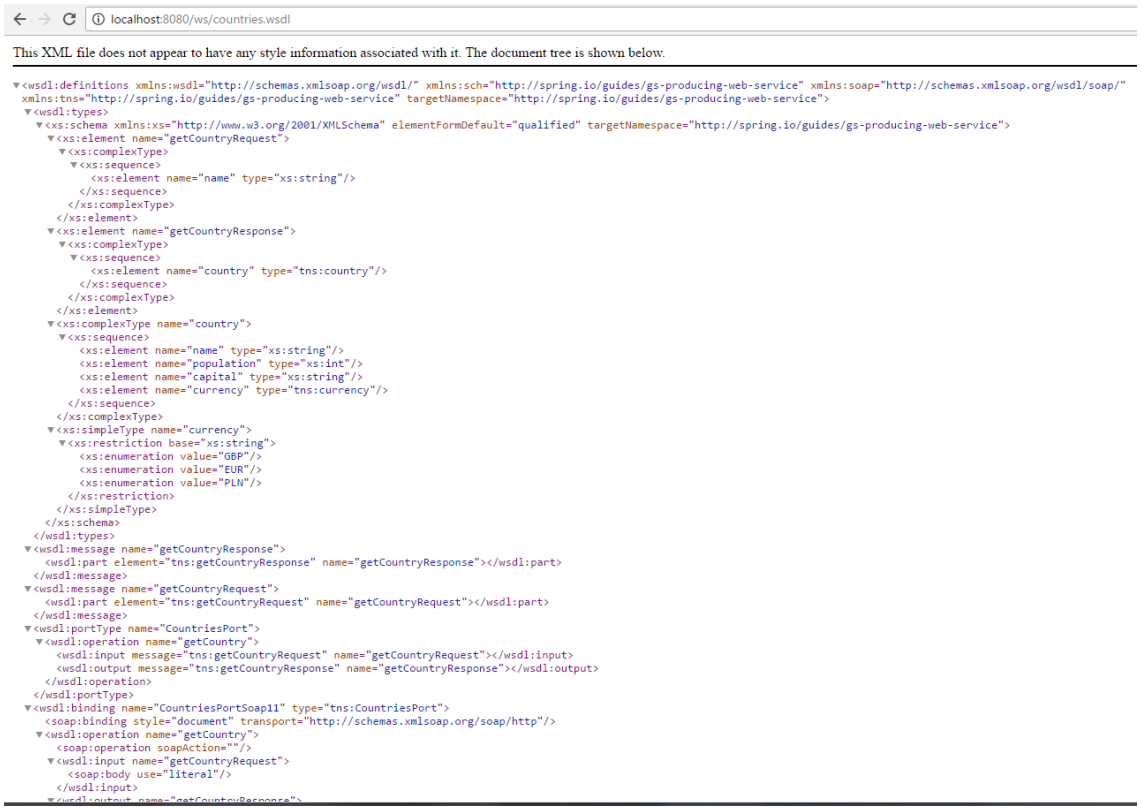
Nykyisellään SOAP-integraatiot ovat vähenemässä niiden toteutusten monimutkaisuuden vuoksi. Tämä ei tarkoita sitä, että teknologia olisi täysin vanhentunut. Tämän teknologian käyttökohteita löytyy vielä useilta eri aloilta, joissa halutaan tarkkaan määriteltyä rajapintakuvausta niin rajapinnan kuin saapuvan ja lähtevän datan osalta. SOAP lisää tietoturvallisuutta hyödyntämällä useita XML:n ominaisuuksia, joilla voidaan varmistaa että lähtevän ja saapuvan tiedon muoto on oletetun määritelmän mukaista.

4.3 SOAP-implementaatio

Spring on ollut osa SOAP-integraatioiden tuottamisessa jo ennen, kuin se paketoitiin omaksi ohjelmointikehyksekseen. Näin ollen Spring tukee hyvin integraation toteutusta. Haastavin osuus integraation toteutuksessa on konfiguroida MessageDispatcherServlet-standardiluokka, jonka avulla voidaan tulkita XML-muotoisia pyyntöjä. Tämä luokka on laajennos standardiin MVC-konfiguraation DispatcherServlet:iin, ja se sisältää nimenomaisesti SOAP- ja XML-kohtaisia lisäominaisuuksia. Työssä SOAP-rajapintapalvelun tuottajan poluksi konfiguroitiin */ws/**, johon SOAP-integraatiot tulee ohjata. Tähdellä merkitty loppuosa on tuttu wildcard-merkintä *, jolla tarkoitetaan kaikkia mahdollisia polkuja alkuosan jälkeen.

SOAP-palvelun tuottamiseksi Spring-kehyksessä vaaditaan lisäksi Spring servlet-konfiguraatio SOAP-viestien tukemiseksi palvelinalustalla ja wsdl4j-kirjasto. Rajapintasopimuksen toteuttavat luokat voidaan luoda automaattisesti kehitystyökalujen avulla. Tässä työssä on käytetty rajapintaluokkien luomiseksi rajapintasopimukseen perustuvaa Java-luokkien käännöstä Maven-projektiin määriteltynä lisäosana. Lisäosa suorittaa rajapintaluokkien luonnin automaattisesti ohjelmakoodin kääntämisen yhteydessä. Työssä on esimerkkitoteutus rajapinnan toteutuksesta, josta osoituksena palvelimen tarjoama WSDL-rajapinnan toteututuskuvaus.

SOAP-palveluntuottaja usein julkaisee rajapintakuvauksen WSDL:n eli ”Web Service Definition Language”-standardin mukaisen rajapintakuvauksen. Usein rajapintapalvelu tarjoaa sen liityntäpisteessään. Tyypiliisesti lisäämällä ”*?WSDL*” liityntäpisteen URL:hin saadaan vastaukseksi WSDL-rajapintakuvaus. Rajapintaan liittyvät asiakkaat voivat tämän kuvauksen avulla automaattisesti esimerkiksi palvelupisteen tavoin luoda rajapinnan toteuttavat asiakasluokat ohjelmakoodin käännöksen yhteydessä. Työssä on toteuttu rajapintatilaus kolmannen osapuolen palvelupisteeseen (kuva 6).



Kuva 6. SOAP-rajapinnan tuottama WSDL-kuvaus.

Työssä integraatio toteutettiin sopimuspohjaisella menetelmällä, jossa määritellään ensin XSD eli ”XML Schema Language” -määrittely, jonka avulla voidaan luoda sopimuksen mukaiset rajapinnan toteuttavat ohjelmaluokat. Toinen tapa luoda SOAP-rajapintatoteutus on koodin kirjoittaminen ensin ja rajapintatoteutuksen generointi toteutuksen pohjalta. Ohjelmakoodiin perustuvan rajapinnan generointi on toisinaan mutkikasta, sillä tietyt objektirakenteet on vaikea määritellä yksiselitteisesti XML:n ja Java-objektin välillä. [5.]

4.4 PayPal-maksuintegraatio

Tässä työssä toteutettiin maksuintegraatio kansainvälisesti tunnettuun maksunvälittäjään ja pitkään alalla toimineeseen PayPaliin. Palvelun maksurajapinta on toteutettu RESTful-konseptin kaikki tasot täyttäen, sisältäen kompleksisen HATEOAS-linkityksen. Rajapinnan kanssa keskustellaan HTTPS:n kautta. Palveluntuottajan rajapintaan voidaan kohdistaa kutsuja sisällyttämällä erikseen pyydettävä API tunniste. Tunniste haetaan POST-pyynnössä, jonka otsikkotietoihin on lisätty tunnisteen hakemiseen liittyviä kenttiä. API-tunniste tulee lisätä kaikkiin rajapinnan kutsuihin sen hakemisen jälkeen. Tunnistetta haettassa viestin runkoon tulee olla liitetty grant_type, jonka arvona client_credentials. Muissa rajapinnan kutsuissa viestin runko saattaa olla tyhjä tai sisältää erityisiä parametreja, joilla esimerkiksi voidaan vahvistaa maksutapahtuma.

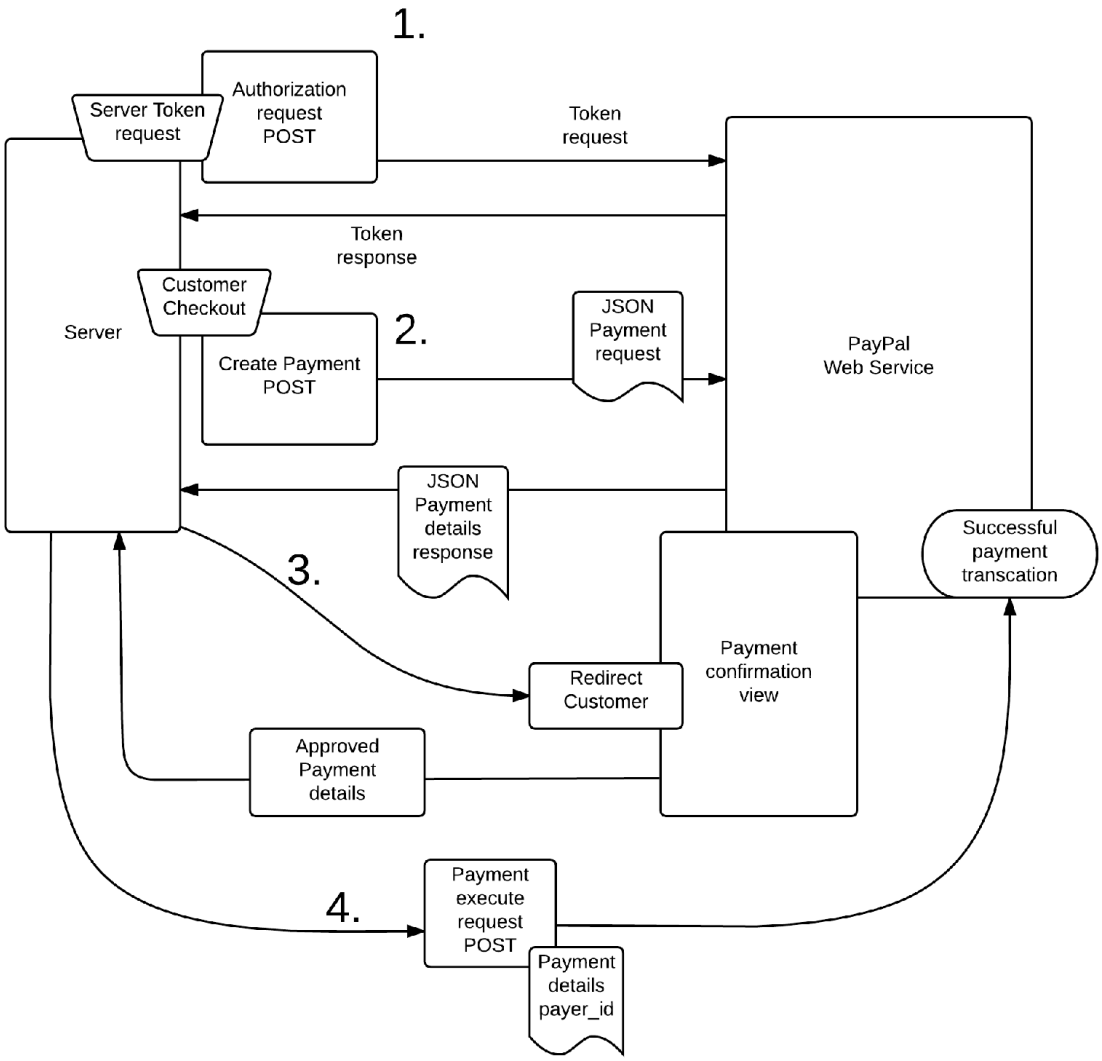
PayPal-maksutapahtuman flow aloitetaan vaihtamalla PayPal-rajapinnan kanssa tunniste, joka asetetaan myöhempien rajapintakutsujen otsikkotietueeseen HTTPS-pyynnössä. Tämän jälkeen luodaan Payment-objekti, joka määritellään JSON-muodossa. Payment objekti kuvaa laskua joka luodaan asiakkaan vahvistettavaksi PayPal-palvelussa. Objektin luonnin jälkeen se lähetetään PayPal-rajapintaan, joka tarkistaa lähetetyn JSON-muotoisen objektin sisällön eheyden ja asettaa maksun luoduksi.

Luotu lasku palautetaan palvelimelle PayPalissa asetetusta konfiguraatiosta riippuen HTTP- tai HTTPS-pyyntönä JSON-muodossa. Tämä objekti sisältää URL:n, johon asiakas voidaan ohjata suorittamaan maksu PayPal-palvelun kautta. PayPalin hyväksytyn maksun jälkeen asiakas ohjataan takaisin maksutapahtuman lähteenä olleeseen palveluun sisältäen URL-parametreissa maksutapahtuman sekä maksajan tunnisteen.

Maksutapahtuma hyväksytään, kun palvelin lähettää PayPalin palauttaman objektin viittaaman payer_id-parametrin osana pyynnön runkoa maksutapahtuman vahvistuksessa. Mikäli asiakas ei palaudu maksutapahtuman lähteenä olleeseen palveluun, ei asiakasta voida veloittaa sillä payer_id-parametria ei ole määritetty ennen kuin maksutapahtuma on suoritettu PayPalin puolesta.

Palatessa lähdepalveluun vastaanotetaan myös tarvittavat tiedot lopullista vahvistusta varten. Näin voidaan myös osoittaa asiakkaalle onnistunut maksutapahtuma, mutta myös tallentaa tärkeitä maksutapahtuman yksilöiviä tietoja turvallisesti maksutapahtuman luoneen lähdepalvelun tietokantaan. [5.]

Kuvassa 7 on esitetty PayPal-maksutapahtuman vuokaavio, jonka tarkoituksena on esittää yksinkertaistettu malli integraation toteutuksesta palvelinalustalla.



Kuva 7. PayPal-integraation toteutuksen vuokaavio palvelinalustaintegraatiossa [4].

4.5 ApiToken-luokka

ApiToken edustaa insinöörityössä kehitettyä JSON-, REST- ja HTTP-teknologioita soveltavaa luokkaa. Palvelinalustalle toteutettu rajapintapalvelun turvaprotokolla toimii tunnisteena kolmannen osapuolen rajapinta-asiakkuuden ja työssä kehitettävän palvelimen välillä. ApiToken on kehitetty erityisesti REST-rajapinnan tietoturvan vuoksi OWASP:n

REST-palvelun tietoturvamääritelmän mukaisesti. [12.] Rajapintapalvelua voidaan hallinnoida edellä mainitun luokan ominaisuuksien avulla, sillä ApiToken-tunniste kytketään rajapintapalvelun asiakkuustietoon rajapintapalveluun liityttäessä.

Tunnisteet lisätään dynaamiseen muistiin, ja ne toimivat HTTP-standardista tuttujen sessioiden tapaan. Rajapintapalvelun käyttölupa lunastetaan pyytämällä tunnistetta käyttäjälle annettua tunnisteen lunastuskoodia vastaan. Tämä on toteutettu kirjautumisessa, minkä jälkeen käyttäjä voi tarkastella lunastuskoodia ja sen tilaa. Merkittävää on huomata, että sekä lunastuskoodilla että myös tunnisteella on vanhenemisaika. Tunnisteen vanheneminen on tietoturvallisuuden kannalta tärkeää, jolloin tunniste uusitaan voimassaolevaa tunnistekoodia vastaan.

Tunnistekoodi toimii kuin palvelun tilauksena, jolloin tätä tietoa vastaan on mahdollista määrittää tunnisteavaimen luovutus rajapintapalveluiden käyttäjille. Näin asiakastieto on kytketty rajapintapalvelussa tunnisteen lunastuskoodin välityksellä ja on siten yksilöitävissä. Näin tietyt väärinkäytökset ja rajapinnan tapahtumat voidaan jäljittää niiden lähteisiin.

5 Tietokanta

5.1 Hibernate-objektitallennus

Hibernaten EntityManager-rajapintatoteutus pitää huolen olioiden tallennuksesta ja olioiden tilan eheydestä. ORM eli Object Relation Mapping-toteutus määrittää, miten objektin kentät ja tietokannan taulut linkittyvät toisiinsa. EntityManageria ohjeistetaan pääosin Entityjen kautta, jotka sisältävät vihjeitä, tässä työssä annotaatioita, joiden mukaan objektit linkittyvät esimerkiksi toisiinsa. Näillä ikään kuin vihjeisiin perustuvilla konfiguraatioilla voidaan muun muassa myös alustaa tietokantaan mallinnettavia ja sieltä noudettavia objekteja tarpeen mukaan.

Tässä työssä Hibernate valittiin aiemmista projekteista hankitun osaamisen, korkean tason olioiden tallennuksen ja liiketoimintaa kokonaisuudessaan kuvaavien objektien tallennustarpeen mukaisena. Tästä liiketoimintaa kuvaavasta mallista esimerkiksi voidaan ottaa asiakas-objekti eli Customer-luokasta luotu olio.

Asiakkaan tallennukseen on määritetty tiettyjä oletuksia asiakkaaseen liittyen, kuten että tavallisella asiakkaalla on tiettyjä ominaisuuksia, joita ei voi olla ilmoittamatta ennen olion tallentamista tietokantaan.

5.2 Yhteysallas C3P0

Palvelinalusta käyttää C3P0-kirjaston luokkia ottaessaan yhteyttä tietokantaan. C3P0 valittiin yhteysaltaan hallintaan sen nopean käyttöönoton ja yleisyyden vuoksi. Hibernate sisältää kehitykseen ja testaukseen soveltuvan yhteysaltaita hyödyntävän vakiokirjaston, mutta on yleisesti suositeltavampaa käyttää kolmannen osapuolen kirjastoja, eivätkä ne vaadi monimutkaisia konfiguraatioita. Työssä käytetyn testiympäristöön sovitetun yhteysaltaan konfiguraatio on esitetty kuvassa 8.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>

        <!-- Connection settings -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://localhost:5555/testikanta</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">postgres</property>

        <!-- Configure hibernate connection pool (C3PO) -->
        <property name="hibernate.connection.provider_class">org.hibernate.service.jdbc.connections.internal.C3P0ConnectionProvider</property>

        <!-- SQL dialect -->
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Print executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create all database on startup -->
        <property name="hibernate.hbm2ddl.auto">validate</property>

        <!-- Hibernate connection pooling, requires Maven dep. -->
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">50</property>
        <property name="hibernate.c3p0.idle_test_period">3000</property>


        <property name="jadira.usertype.autoRegisterUserTypes">true</property>
        <property name="jadira.usertype.javaZone">UTC</property>
        <property name="jadira.usertype.databaseZone">UTC</property>
        <!-- Annotated entity classes -->

      <!-- Wont work since not using JTA-conventions <mapping resource="queries.hbm.xml"/> -->

    </session-factory>
</hibernate-configuration>
```

Kuva 8: Hibernaten C3P0-yhteysaltaan konfiguraatio.

Yhteysaltaan käytön tuomat resurssien ja laskenta-ajan säästöt ovat huomattavat. Käytännössä ja lyhyesti esiteltynä yhteysaltaan toteutus avaa sopivan määrän yhteyksiä tietokantaan ohjelmasta, ryhmittää tietokannan hakulauseet nopeasti suoritettaviin kokonaisuuksiin, luo esimääritellyt SQL-lauseet ja ajaa ne usean yhteysinstanssin kautta.

Tietokantaan yhdistävien instanssien määrää säädellään tarpeen mukaan ja vapautuneita yhteyksiä kierrätetään uusien SQL-operaatioiden käyttöön. Tällöin aikaa säästyy uuden instanssin luomiselta sekä instanssin kytkemisestä tietokantaan jokaisen yksittäisen haun yhteydessä.

Kuvassa 9 on esitetty mittaustuloksia, jossa verrataan yhteysallasta käyttävää ja ilman yhteysallasta toteutettua ohjelmaa, jotka suorittavat useita tietokantahakuja. Kuvassa on taulukoitu erään sovelluksen suoritusaikoja kantaoperaatioiden lukumäärän perusteella. Ylempi luku kuvaa ohjelmaa, joka on kytketty käyttämään yhteysallasta. [8.]

	100 Iterations	100 Iterations	1000 Iterations	3000 Iterations
Pooling	547 ms	<10 ms	47 ms	31 ms
Non-Pooling	4859 ms	4453 ms	43625 ms	134375 ms

Kuva 9. Yhteysaltaan suorituskykymittaus.

Ajat ovat millisekunti-yksikkönä, joten taulukosta on selkeästi huomattavissa yhteysallasta käyttävän ohjelman suorituskykyero. Ensimmäisessä sarakkeessa on huomattavissa poikkeavan korkea suoritusajan mittaustulos yhteysallasta käyttävässä ohjelmassa. Tämä selittyy fyysisen yhteyden luomisella tietokantaan, sekä yhteysaltaan hallintaolioiden alustamisella. Yhteys sijoitettiin vielä yhteysaltaaseen, josta se nostettiin ohjelman käyttöön. Myöhemmät mittaukset yhteysallasta käyttävän sovelluksen tapauksessa hyödyntävät aiemmin tehtyjä alustuksia. Poikkeus taulukon yhteysallasta käyttävän sovelluksen mittaustuloksissa 1 000 ja 3 000 suorituskierroksen välillä selittyy Javan suorituksen optimoinnista, joka pyrkii minimoimaan itsenäisesti sovelluksen käyttämiä resursseja.

Yhteysallasta käyttävän ohjelman suoritusajan mittaustulokset perustuvat merkittävästi dynaamisesti luotujen yhteyksien määrään. Kun kaikki yhteydet ovat käytössä, yhteysaltaan hallinnoija lisää yhteyksien määrää. Yhteysaltaan hallinnoija on ohjelmoitu optimoimaan suorituskykyä, mutta sille voidaan antaa alustavat ja raja-arvot, joita hallinnoijan tulee noudattaa yhteysallasta optimoidessaan.

Useammalla yhteydellä myös samanaikaisesti suoritettavien operaatioiden määrä kasvaa. Ilman yhteysallasta toimivan ohjelman suoritukseen kuluttama aika kasvaa kanta-operaatioiden lukumäärän suhteen lineaarisesti. [9.]

6 Jatkokehitys

Työssä valmistunut palvelinalusta tekee mahdolliseksi integroida muiden osapuolten palveluita useiden järjestelmien välillä hyödyntäen suosituimpia integraatioteknologioita. Palvelimen pohjatoteutus tukee palvelimen hyödyntämistä useissa eri sovellutuskohteissa tarjoilemaan web-sisältöä tai toimimaan linkkinä järjestelmien välisissä integraatioissa. Kehitettäessä työssä valmistuneen pohjatoteutuksen päälle voidaan tuleva kehitystyö ohjata suoraan liiketoimintakriittisiin komponentteihin sovelluksessa. Integraatioteknologioiden avulla palvelinalusta mahdollistaa myös pilvipalvelutuotannon, jolloin fyysistä IT-infrastruktuuria ei ole välttämätöntä hankkia. Näin voidaan tuottaa vain liiketoiminnalle välttämätön laitteisto, jolloin laitteiston ylläpitokustannukset minimoituvat.

Työssä toteutettuja rajapintoja voidaan hyödyntää useissa eri käyttökohteissa. SOAP-teknologia pitää edelleen pintansa merkittävänä rajapintateknologiana. Esimerkiksi Suomessa Kela tukee SOAP-rajapintaa. Ruotsissa Kelan vastine on Försäkringskassan API, joka tukee myös SOAP-rajapintaa asiakkaiden korvaustietojen siirtämisessä asiakasjärjestelmien välillä. Myös useat pankit luottavat SOAP-teknologiaan ja esimerkiksi Nordea ja Osuuspankki tarjoavat rajapintakuvauksen SOAP-teknologialle.

Toisaalta modernit REST-rajapinnat yleistyvät myös nopeasti ja useista palveluista on saatavilla julkistaa dataa palvelun toiminnasta. Esimerkiksi työssä tarkasteltu PayPal API tukee JSON-muotoista viestinvälitystä, joten kyseistä teknologiaa käytetään myös moderneissa web-rajapintapalveluissa.

7 Yhteenveto

Insinöörityölle asetettiin teknologiatavoitteiksi löytää ne potentiaaliset palvelinalustan integraatioteknologiat, joita voitaisiin jatkokehityksen myötä hyödyntää järjestelmien välisien palveluiden tuottamisessa. TiVi-lehden julkaisussa [11] nostettiin esille useasti REST-teknologian tuomat edut ja työssä esiteltyjen REST- ja SOAP-integraatioteknologioiden tärkeys palvelininfrastruktuurien suuntautuessa yhä enemmän fyysisestä laitteistosta kohti pilvipalveluita. Artikkelissa mainittiin myös SOAP-palveluiden pitävän sijaa erityisesti julkisen sektorin palveluissa.

Työssä havaittiin, että poikkeuksellinen tapa konfiguroida palvelinsovellus hyödyntämällä Spring Framework 4:n ominaisuuksia oli huomattavasti oletettua hankalampaa ja vaati osittain myös perusteellista selvittämistä tiettyjen komponenttien osalta. Tulee myös huomioida, että palvelinalustan päälle tulevaisuudessa rakennettavat liiketoimintakeskeiset komponentit hyödyntävät kaikki yhteisesti palvelinalustalle työssä toteutettuja komponentteja. Tehty työ voi korvata itsensä myöhemmässä vaiheessa moninkertaisesti, sillä esimerkiksi integraatioteknologiatukea mahdollisissa tulevissa järjestelmätoimituksissa ei tarvitse ohjelmoida enää alusta lähtien.

Työn tärkein tavoite oli perehtyä vaihtoehtoisiin toteutustapoihin ja moderneihin teknologioihin tuottaa palvelinsovellus. Työssä onnistuin luomaan modernia sovelluskehystä käyttäen palvelinsovelluksen, jota on alustavasti mahdollista käyttää muiden sovelluskonseptien testauksessa sekä liiketoiminnallisessa tarkoituksessa jatkokehityksen myötä.

Lähteet

1 Thibaud, J. 2016. Top programming languages to learn in 2017. Verkkodokumentti https://www.codingame.com/blog/top-programming-languages-to-learn-in-2017/. Luettu 20.12.2016.

2 Burrows, Matthew. 2012. Operational efficiency – it's not just about cost cutting. Verkkodokumentti. http://www.bsmreview.com/oppseff.shtml. Luettu 15.1.2017.

3 Paytrail API. 2017. Verkkodokumentti Paytrail. https://www.paytrail.com/kehitta-jille-form-ja-rest-rajapinnat. Luettu 7.3.2017.

4 PayPal API. 2017. Verkkodokumentti. PayPal. https://developer.paypal.com/docs/api/payments/. Luettu 1.2.2017.

5 Mudunuri, Srinivas. 2015. Spring Framework. First print.

6 Patrizio, Andy. 2016. XML is toast, long live JSON. Verkkodokumentti. CIO. <http://www.cio.com/article/3082084/web-development/xml-is-toast-long-live-json.html>

7 Representational state transfer. 2017. Verkkodokumentti. Wikipedia. <https://en.wikipedia.org/wiki/Representational_state_transfer>. Luettu 15.1.2017.

8 JDBC Connection pooling. Progress. 2017. Verkkodokumentti. Progress. <https://www.progress.com/tutorials/jdbc/jdbc-jdbc-connection-pooling>. Luettu 2.4.2017.

9 Connection Pooling with hibernate 4. 2017. Verkkodokumentti. Java beginners tutorial. <http://javabeginnerstutorial.com/hibernate/connection-pooling-with-hibernate-4/>. Luettu 5.4.2017.

10 Spring Framework Reference. 2017. Verkkodokumentti. Spring <docs.spring.io/spring/docs/current/spring-framework-reference/html/new-in4.3.html>. Luettu 1.2.2017.

11 TiVi, 4 / 2017.

12 OWASP. 2017. Verkkodokumentti. REST Security Cheat Sheet. <https://www.owasp.org/index.php/REST_Security_Cheat_Sheet>. Luettu 12.4.2017.

13 PayPal. 2017. Verkkodokumentti. PayPal - Fees. <https://www.paypal.com/fi/cgi-bin/webscr?cmd=_display-fees-outside>. Luettu 10.2.2017.