TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

IDK40LT

Raido Roben 123900IABB

PARIMA JAVA RAAMISTIKU LEIDMINE MIKROTEENUSTE ARENDAMISEKS TELIA EESTI AS NÄITEL

Bakalaureusetöö

Juhendaja: Inna Švartsman
MSc
Lektor

Haimar Kulbas
MSc
Infosüsteemide arhitekt

Tallinn 2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Raido Roben

23.05.2016

Annotatsioon

Antud töö esimene eesmärk on näidata mikroteenustest saadav kasu monoliitse arhitektuuriga võrreldes. Teine eesmärk on leida Java-põhine raamistik, mis sobib Telia Eestile mikroteenuste arendamiseks kõige enam.

Lõputöö esimese eesmärgi saavutamiseks viidi läbi arhitektuuride vaheline võrdlus. Tulemusteni jõuti teoreetilisi allikaid kasutades ning arhitektuure kaheksast erinevast aspektist võrreldes. Teise eesmärgi saavutamiseks tegi Telia uuringu, mille abil valiti välja kaks kõige sobilikumat raamistikku ettevõtte infosüsteemide arendamiseks. Raamistike paremaks mõistmiseks ehitas töö autor mõlema raamistikuga sarnase veebirakenduse. Tehnoloogiate võrdluses toetub autor nii enda kogemustele kui ka välistele allikatele.

Töö käigus jõuti kahe olulise tulemiseni. Esiteks tõestati, et mikroteenustel põhinev arhitektuur on monoliitsest arhitektuurist jätkusuutlikum. Seda enamasti seetõttu, et mikroteenuseid käsitletakse iseseisvate rakendustena, mida on võimalik lihtsa vaevaga hallata, uuendada või välja vahetada. Teiseks leiti, et parim Java raamistik mikroteenuste arendamiseks Telia Eesti jaoks on Spring Boot. Antud raamistik pakub seda, mida soovib iga mikroteenuseid arendav ettevõte: kiiret võimalust alusprojekti genereerimiseks ning lihtsat sõltuvuste haldamist. Lisaks on Spring Bootil suur kasutajate hulk. Võrdluses Dropwizardiga hinnati StackOveflow'st, GitHub'ist ja Google Trends'ist saadud andmete põhjal Spring Booti populaarsust 8-10 korda suuremaks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 43 leheküljel, 5 peatükki, 14 joonist, 2 tabelit.

Abstract

Finding the Best Java Framework for Developing Microservices Based on the Example of Telia Eesti AS

In the last 10 years a lot of companies have used monolithic architecture for back-end development. Nonetheless, there are some considerable disadvantages while using it: monolithic application are complex to maintain and update. Therefore, companies are looking for a more suitable alternative, that increases the level of control. Since 2014, a new approach for developing back-end systems, has been brought up- the microservices. It is believed that microservices will be the key for resolving all the problems caused by monolithic architecture.

The aim of this thesis is to demonstrate how companies will benefit from using microservices. Since it is important to have the best tools for software development, another aim, is to find the best framework for writing microservices based on the example of Telia Eesti AS.

The first goal was achieved by comparing the selected technologies. The architectures were compared from eight different angles backed by theoretical materials. To accomplish the second goal, a research was conducted by Telia to sort out the two best frameworks for microservices development. In order to gain additional insight the author developed a web application using the two selected frameworks. The results are based on both external sources and the authors own findings.

The thesis reached to two important conclusions. Firstly, it was shown microservises are far more sustainable approach compared to monolithic architecture. Microservices are developed as independent applications: they are easier to maintain, upgrade and replace. Secondly, the findings demonstrate, that the best framework for developing mircroservises on the example of Telia Eesti AS, is Spring Boot. Spring Boot provides all the things one need for developing application based on microservices: a fast way to generate a template and simple dependency management. In addition, Spring Boot has a

large community support. According to the data provided by StackOverflow, GitHub and Google Trends, Spring Boot is 8 to 10 times more popular than Dropwizard.

The thesis is in Estonian and contains 43 pages of text, 5 chapters, 14 figures, 2 tables.

Lühendite ja mõistete sõnastik

BEA WL	*BEA WebLogic*, Java rakendusserver
POJO	*Plain Old Java Object*, Java objekt
EMT	Eesti Mobiiltelefon
CORBA	*Common Objekt Request Broker Architecture*, komponenttehnoloogia
SQL	*Structured Query Language*, struktuurpäringukeel
PL/SQL	*Procedural Language/Structured Query Language*, protseduuriline ja struktureeritud päringukeel
J2EE	*Java 2 Platform Enterprise Edition*, Java platvorm
EJB	*Enterprise JavaBeans*, ettevõtte serveri tarkvara
JSP	*JavaServer Pages*, servlet tehnoloogia
HTML	*Hyper Text Markup Language*, veebidokumentide märgistuskeel
API	*Application Programming Interface*, rakendusliides
REST	*Representational State Transfer*, veebi arhitektuuri stiil
MVC	*Model View Controller*, tarkvara arhitektuuri muster
JPA	*Java Persistence API*, Java rakendusliidese standard
DAO	*Data Access Object*, andmeobjekt
STS	*Spring Tool Suite*, Springile spetsialiseeruv arenduskeskkond
JDBC	*Java Database Connectivity*, rakendusliides andmetega tegelemiseks
JSON	*JavaScript Object Notation*, andmevahetusvorming

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Telia Eesti AS on Eesti suurim telekommunikatsiooni ettevõte, mis on moodustatud EMT ja Elioni ühendamisel. Telia liitmine toimus 2016. aasta alguses, mistõttu pole ettevõtte töö jõudnud täielikult ühtlustuda ning kasutatavad infosüsteemid jagunevad üldjoontes kahte leeri: EMT ja mobiilseid teenuseid pakkuvad süsteemid ning Elion ja televisiooniteenuseid pakkuvad süsteemid. Antud töös keskendutakse EMT-poolsete süsteemide arendusele.

Bakalaureusetöö esimene eesmärk on näidata mikroteenustest saadavat kasu monoliitse arhitektuuriga võrreldes. Teine eesmärk on leida Java-põhine raamistik, mis sobib Telia Eestile mikroteenuste arendamiseks kõige enam.

Esimene eesmärk saavutatakse arhitektuuride omavahelisel võrdlusel teoreetilisi allikaid kasutades. Teise eesmärgi saavutamiseks viidi Telia poolt läbi uuring, mille tulemusena jäi valikusse kaks kõige sobilikumat raamistikku mikroteenuste arendamiseks. Töö autor õppis raamistikke esmalt tundma ning seejärel arendas mõlema raamistikuga lihtsa veebirakenduse. Kuna mõlema tehnoloogiaga ehitati sarnane rakendus, siis lähtutakse parima raamistiku välja selgitamisel autori enda tähelepanekutest ja kogemustest. Lisaks kasutatakse ka välistest allikatest leitud informatsiooni.

Bakalaureusetöö jaguneb kaheks suuremaks peatükiks. Esimene peatükk on teoreetiline. Kõigepealt tutvustatakse seda, kuidas on Telias varasemalt süsteeme arendatud. Seejärel selgitatakse monoliitse ja mikroteenustel põhineva arhitektuuri olemust, millele järgneb nende kahe põhjalik võrdlus. Lõpetuseks esitatakse võrdluse tulemustest tehtud järeldused. Teine peatükk on praktiline. Esmalt esitatakse detailne kirjeldus sellest, kuidas arendada veebirakendust valikus olevate raamistikega. Kui mõlema raamistikuga loodud rakendus on valmis, hakatakse raamistikke omavahel võrdlema. Võrdlusel lähtutakse teguritest, mis on Telia jaoks kõige tähtsamad. Viimasena tehakse järeldused, mille tulemusena pakutakse Teliale välja parim Java-põhine raamistik mikroteenuste arendamiseks.

Käesolevast tööst on eelkõige kasu Telia Eestile, sest töö on tehtud firma poolt ette antud parameetreid järgides. Samuti on käesolev töö kasulik kõikidele ettevõtetele, kes soovivad monoliitse arhitektuuri mõne lihtsama ja kergemini hallatava arhitektuuri vastu välja vahetada. Teema on aktuaalne, sest reaalsuses vaevleb tarkvara monoliitsuse küüsis enamik suuri ettevõtteid, kelle infosüsteemid on loodud kümme või rohkem aastat tagasi.

Lõputöö on kirjutatud 2016. aasta kevadel Tallinna Tehnikaülikooli lektori Inna Švarstmani ning Telia Eesti infosüsteemide arhitekti Haimar Kulbase juhendamisel.

2 Teoreetiline osa

2.1 Telia ajalugu

Telia Eesti AS on Eesti suurim telekommunikatsiooni ettevõte. Kuna Telia on moodustatud varasemast EMT'st ja Elionist, tegeleb ta nii mobiili- kui televisiooniteenuste pakkumisega. Käesolevas töös Telia Eesti ajaloost rääkides keskendutakse sellele, mis on seotud EMT tegevusega.

1990ndate aastate alguses, kui EMT oma tegevusega alustas, arendati põhisüsteeme klient-server tehnoloogiat kasutades. Täpsemalt kasutati Oracle Forms nimelist lahendust. Olemuselt oli see igas tööjaamas asuv klientrakendus, mis kasutas andmete hoiustamiseks Oracle andmebaasi ja serveriga suhtlemiseks TCP/IP protokolli. Seoses interneti arenguga, samuti 1990ndate aastate alguses, ilmusid turule esimesed veebirakendused. Veebirakenduste arendamisega hakkas tegelema ka EMT. Algselt toimus arendus Oracle platvormil PL/SQL'ga, mille abil genereeriti HTML veebivaated. Antud tehnoloogiast loobuti kiiresti, sest rakendusi oli raske hallata. Mindi üle Java-põhiste rakenduste arendamisele ning kasutusele võeti JSP ja servlet tehnoloogiad. [1]

2000ndate aastate alguses hakati katsetama CORBA komponenttehnoloogiat. Täpsemalt kasutati Borland Visibrokerit, mis töökindlusele vaatamata oli raskesti programmeeritav. Arenduskiirus antud vahevaraga oli liiga aeglane. Samal ajal muutus üha populaarsemaks Javal põhinev J2EE ja EJB komponenttehnoloogia. Kuna EMT kasutas paljude rakenduste kirjutamiseks Javat, asendati CORBA J2EE'ga. EJB vajas töötamiseks rakendusserverit ning kõikidest turul olnud valikutest otsustati BEA WL'i kasuks. Antud tarkavara oli töökindel, lihtne ja hästi skaleeruv. BEA WL'i ja EJB'sid kasutati enamiku rakenduste juures aastaid. Kuigi EJB'sid oli võrreldes CORBA'ga kerge programmeerida, raskendas arendust asjaolu, et iga väiksema muudatuse korral tuli teenus rakendusserverisse uuesti paigaldada. Selle tagajärjel muutus arendustsükkel suhteliselt pikaks. Samuti süvenes aja jooksul rakenduste monoliitsus. Nimelt liideti EJB'd kokku ning paigaldati serverisse ühe suure monoliitse tükina. Suurte tükkide arendamine, hooldamine ja testimine oli aga aeglane ja vaevaline. [2], [3]

2000ndate aastate alguses muutus populaarseks Java Spring raamistik, mis lubas komponente arendada puhta Java põhimõttel. Nimelt, kui EJB sõltus oluliselt rakendusserverist, siis Springi Java klasse POJO'sid sai arendada ilma rakendusserverita. See kiirendas oluliselt üksikute Java klasside arendamist ning lihtsustas testimist. Lisaks jättis Spring vabaduse kasutada muid konteinereid, mille sees Java Beans'e jooksutada. Kuna Oracle ostis BEA WL'i ära, muutus selle kasutusõigus endisest kallimaks, mille tulemusena võeti suund EJB ja BEA WL'st loobumise suunas. Olgugi et arenduskiirus ja testitavus paranesid, ei lahendanud see järjest enam süvenevat monoliitsuse probleemi. Väga paljud Java klassid olid üksteisest suuresti sõltuvad ning paigaldamine toimus endiselt suurte tükkide kaupa. Suure sõltuvuse tõttu oli raskendatud ka muudatuste läbiviimine. Asjaolu, et ettevõtte erinevad arenduspartnerid arendavad rakenduse tükke erineva kiirusega, teeb antud olukorra veelgi keerulisemaks. Selleks, et muudatusi eri aegadel läbi viia, tuleks monoliitne tükk väiksemateks teenusteks lahti võtta. [4]
Telia plaan on üleminek mikroteenustele. Mikroteenuste termin on tulnud kasutusse alles paari viimase aasta jooksul. Tegu on arhitektuurilise muudatusega, mis peaks Telia jaoks lahendama monoliitsusest tingitud probleemid. Järgnevas peatükis räägitakse monoliitse ja mikroteenustel põhineva arhitektuuri olemusest täpsemalt.

2.2 Monoliitne ja mikroteenustel põhinev arhitektuur

Monoliitse arhitektuuriga disainitud rakendus on suurem kogum, mis koosneb omavahel tihedalt sõltuvuses olevatest väiksematest osadest. Antud väidet illustreerib ka joonisel 1 olev vasakpoolne pilt, kus suur kast iseloomustab tervikrakendust ning värvilised kujundid erinevaid rakenduse tükke. [5]

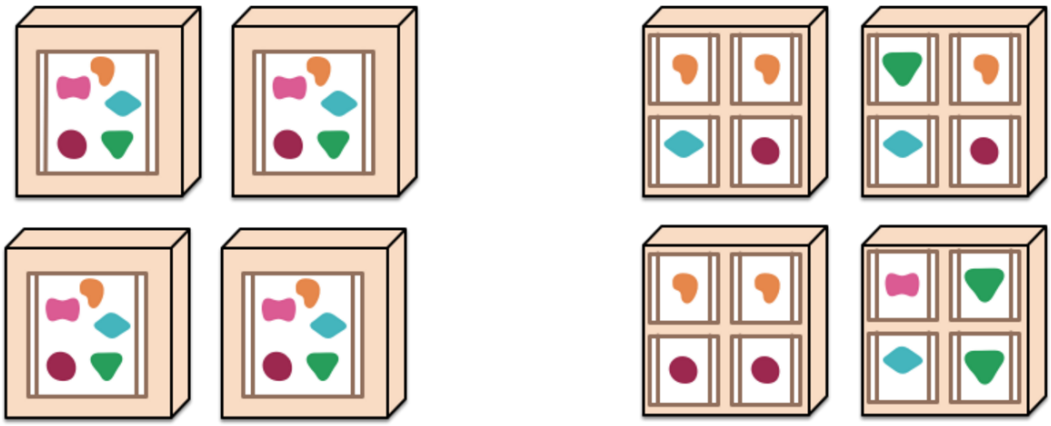
Monoliitsed rakendused võivad olla edukad, kuid järjest enam tunnevad inimesed nende osas rahulolematust. Suure monoliitse rakenduse muutmine on tülikas, sest rakenduse osad on üksteisega tihedalt seotud. Läbi aegade on olnud raske säilitada head modulaarsust. Veelgi raskem on olnud muudatuste tegemine, mis mõjutaksid vaid seda moodulit, milles muudatusi tehti. [5]

Täpsemalt öeldes koosneb monoliitne rakendus mitmest moodulist ning baseerub ühel kindlal koodikogumil. Moodulid on jaotatud nii ärilisteks kui tehnilisteks osadeks. Monoliitse rakenduse puhul on raske säilitada head modulaarsust, sest rakenduses

olevad moodulid on omavahel tihedalt seotud. Sellest tulenevalt on ühe mooduli muutmiseks vaja terve rakendus uuesti ehitada ning serverisse paigaldada. [5]

IT-tööstus on ettevõtte rakenduste arendamiseks kasutanud pikka aega monoliitset lähenemist. Paljud firmad on monoliitsete rakenduste ehitamisele ja täiustamisele investeerinud aastaid. Monoliitset arhitektuuri kutsutakse ka mitmekihiliseks, sest antud arhitektuuri kasutades on rakendused jaotatud nii esitlus-, äri-, andme- kui rakenduskihiks. [5]

Viimase viie aasta jooksul on inimesed hakanud veebis käimiseks kasutama teistsuguseid seadmeid. Kui varasemalt kasutati internetis olemiseks lauaarvuteid, siis viimaste aastatega on järjest enam hakatud kasutama mobiile ja teisi nutiseadmeid. Kuna nutiseadmeid on endaga lihtne igale poole kaasa võtta, siis vastav trend järjest süveneb. Selle tulemusena on tekkinud vajadus lisaks arvutirakendustele arendada ka mobiilseid rakendusi. Mobiilseid rakendusi saab arendada läbi API'de, mis monoliitset arhitektuuri ei toeta. Kokkuvõttes on nii monoliitse rakenduse halb hallatavus kui ka nutiseadmete suur populaarsus tekitanud vajaduse muuta kasutatavat süsteemiarhitektuuri. Hea asendus monoliitsele arhitektuurile on üleminek mikroteenustele. [5]



Joonis 1. Monoliitse ja mikroteenustel põhineva süsteemi arhitektuur. [6]

Mikroteenustel põhinev arhitektuur on lähenemine, kus üks suur rakendus on jagatud paljudeks väiksemateks rakendusteks. Antud väidet illustreerib parempoolne pilt joonisel 1, kus suur kast iseloomustab teenuste kogumikku ning väike kast ühte mikroteenust. Igat rakendust käsitletakse kui pisiteenust, mida saab iseseisvalt arendada, paigaldada ja testida. Antud teenused kasutavad suhtlemiseks REST API'it ning

pannakse käima eraldi protsessina. Lisaks on nad minimaalselt tsentraliseeritud ning võivad vabal valikul kasutada erinevaid programmeerimiskeeli ja andmehaldustehnoloogiaid. [6]

Selleks, et paremini mõista mikroteenuste poolt pakutavaid võimalusi, tuleb järgnevalt juttu mikroteenuste omadustest.

2.3 Monoliitse ja mikroteenustel põhineva arhitektuuri erinevused

1. Tarkvara komponentideks jagamine

Komponent on tarkvaraühik, mida saab eraldi asendada ja uuendada. Monoliitne arhitektuur kasutab tarkvara komponentideks lammutamisel nii teeke kui teenuseid. Monoliitse rakenduse komponent koosneb mitmest omavahel seotud teenusest. Teek on protsessisisene komponent, mis on ühendatud programmi ja mida kutsutakse välja sisemälu funktsiooni kutsungitega. Teenus on protsessiväline komponent, mis loob ühenduse veebipäringu või kaugprotseduuri kutsungiga. Mikroteenustel põhineva arhitektuuri puhul vastab ühele komponendile üks eraldiseisev teenus. [6]

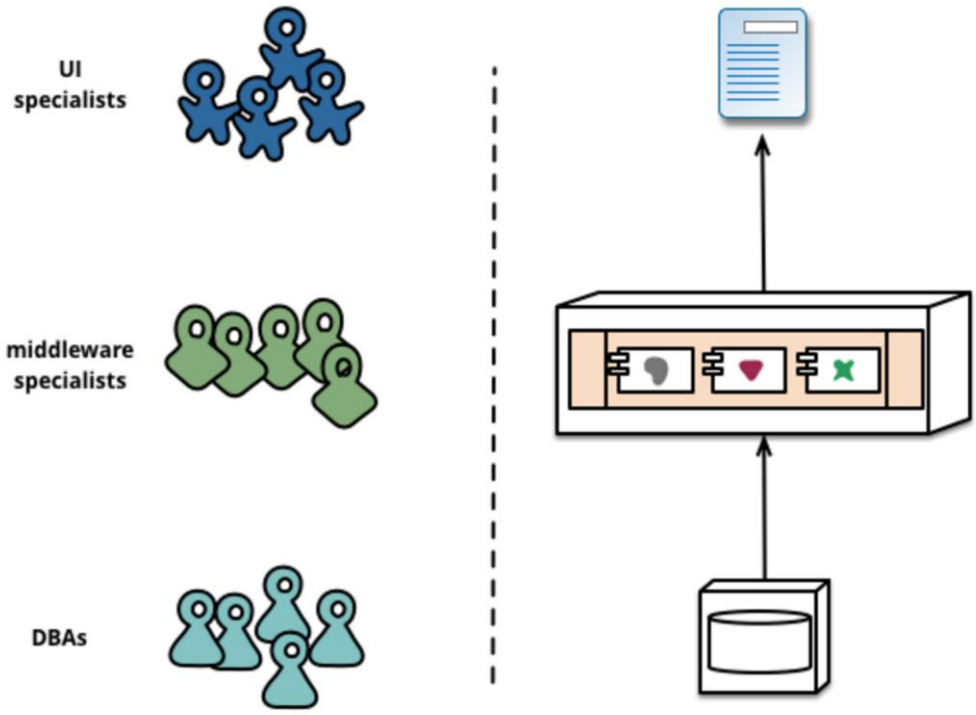
Peamine põhjus, miks kasutada teenuseid komponentidena, on see, et neid saab iseseisvalt paigaldada. Kui võtta rakendus, mis koosneb ühe protsessi ulatuses mitmest teegist, siis ühe komponendi muutmiseks tuleb kogu rakendus uuesti paigaldada. Samas, kui see rakendus on laiali võetud mitmeks teenuseks, siis muudatused, mis toimuvad ühe teenuse piires, nõuavad ainult selle konkreetse teenuse uuesti paigaldamist. Eesmärk on minimeerida koodi sõltuvust. [6]

2. Meeskondade moodustamine

Öeldakse, et suhtlusahel, mida organisatsioon rakenduse ehitamiseks kasutab, on peegelpilt sellest, kuidas rakenduse osad üksteisega suhtlevad. Sellest tulenevalt toimub meeskondade komplekteerimine vastavalt liikmete erialale. Täpsemalt jaotatakse kasutajaliidesega tegelevad inimesed ühte, põhisüsteemiga tegelevad inimesed teise ning andmebaasiga tegelevad inimesed kolmandasse gruppi. [6]

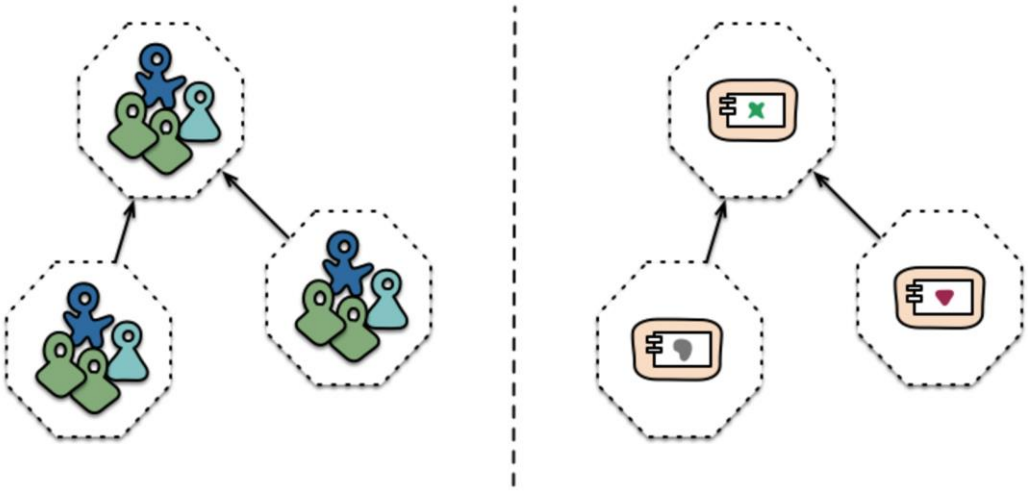
Vaadates joonist 2, siis alt ülesse liikudes näeme, et andmebaasi meeskond ja kasutajaliidese meeskond otse ei suhtle. Suhtlus toimub läbi põhisüsteeme haldava meeskonna. Analoogselt toimub põhisüsteemide kaudu ka rakenduse kasutajaliidese ja

andmebaasi vaheline suhtlus. Antud struktuurile vastavalt tegeleb iga meeskond vaid ühe kindla valdkonnaga, mistõttu peab igasuguse muudatuse korral meeskondi kokku tooma ja koordineerima. Taoline tegevus tekitab aga lisakulusid nii rahas kui ajas. Tark lähenemine oleks moodustada meeskond, mis sisaldaks spetsialisti igast eespool mainitud valdkonnast. [6]



Joonis 2. Ettevõtte ja süsteemi struktuur. [6]

Mikroteenused just taolisi meeskondi kasutavadki. Meeskondade jaotus toimub teenuste põhiselt. Teisisõnu tegeleb üks meeskond ühe teenusega, kattes kõik teenusega seonduvad valdkonnad. Antud teenus sisaldab nii kasutajaliidest, püsivat mäluruumi kui serveripoolset loogikat. Seetõttu on tähtis, et teenust arendav meeskond koosneks erinevate erialade inimestest. Joonise 3 parempoolselt kujutiselt on näha, et monoliitsete rakenduste puhul komplekteeritakse meeskonnad inimeste tegevusvaldkonna järgi. Mikroteenuste puhul valitakse liikmed vastavalt teenusele, nii et iga meeskond sisaldaks erinevate erialade spetsialiste. [6]



Joonis 3. Meeskondade moodustamine. [6]

3. Tugevam side arendatud tarkvaraga

Monoliitse rakenduse puhul kasutatakse mudelit, mille eesmärk on terviktoote ühe osa arendamine. Tarkvara käsitletakse kui projekti. Tüki valmimisel lisatakse see tervikule ning tüki loonud meeskond sellega edasi enam ei tegele. Halduse ja uuendamisega tegeleb mõni muu osakond. [6]

Mikroteenuste puhul rakendatakse ideed, kus iga meeskond tegeleb arendatud tarkvaraga terve selle eluea vältel. Tarkvara käsitletakse kui toodet. Antud lähenemise puhul muutuvad tarkvara autorid enda poolt loodud rakendusega lähedasemaks. Nad näevad, kuidas rakendus sündis ning kuidas see tuleb toime tootmises. Lisaks suureneb suhtlus arendajate ja tarkvara kasutajate vahel, sest arendajad pakuvad lisaks tarkvarale ka kasutajatuge. Arendusmeeskonna ja kasutajate vaheline otsene suhtlus on tähtis, sest arendajad saavad sel juhul täpseima ülevaate tarkvara funktsioneerimisest. [6]

4. Targad otspunktid või tark kanal

Paljude teenuste puhul kasutatakse arhitektuurimudelit, kus kogu loogika on paigutatud rakenduste vahelisse kanalisse. Enterprise Service Bus on näide taolisest tarkvara arhitektuurist. Kõik keerulisemad otsused- andmete visualiseerimine, teisendused või marsruutimine- tehakse rakenduste vahelises osas. [6]

Mikroteenustel põhineva arhitektuuri korral lähenetakse asjale vastupidi: kogu tarkus asub rakenduste otspunktides. Mikroteenustel põhinevate rakenduste puhul

tähtsustatakse madalat sõltuvust ning kõrget ühtekuuluvust. Lihtsa ja kiire seadistusega on võimalik rakendus koos ülejäänud tarkvaraga käima panna. Samuti saab seda kergesti eemaldada, ilma et mõne muu teenuse funktsioneerimist mõjutaks. Iga teenus omab kindlat äriloogikat ning käitub kui filter: võtab vastu päringuid, rakendab äriloogikat ning tagastab vastuseid. Monoliitsed rakendused kasutavad nii esimest kui teist lähenemist. [6]

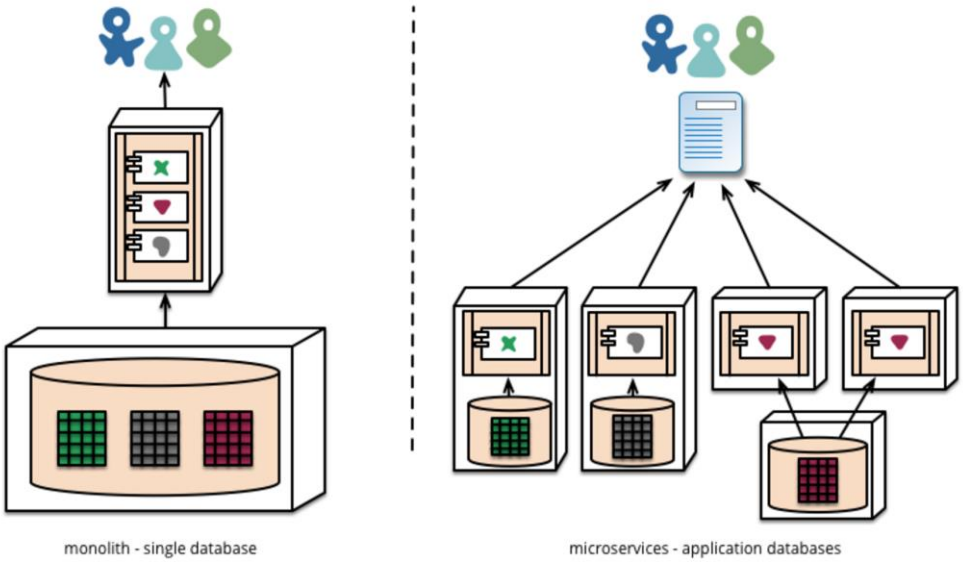
5. Arendusmeetmete tsentraliseeritus

Tsentraalne arendamine on tegevus, kus ollakse kinni ühe tehnoloogia poolt pakutavates tööriistades ja raamistikes. Väikese rakenduse puhul see erilist rolli ei oma, kuid mahukama koodiga teenuste puhul hakkab see arendamist piirama. Pole olemas ühte tööriista, mis lahendaks kõik arenduse käigus tekkivad probleemid. Igale probleemile tuleks läheneda erineva nurga alt. Seetõttu on mõistlik valida uue teenuse arendamisel just selline tehnoloogia, millega saab töö kõige efektiivsemalt tehtud. Monoliitse rakenduse puhul järgitakse ametlikke standardeid ning erinevate tööriistade asemel kasutatakse üldiselt vaid ühte tehnoloogiat. Mikroteenuste arendajad eelistavad aga tööriistu, millega keegi teine on juba varasemalt sarnase probleemi lahendanud. Sellest tulenevalt kasutatakse mikroteenuste arendamiseks mitut tehnoloogiat. [6]

6. Andmehalduse tsentraliseeritus

Kui monoliitsed rakendused salvestavad andmed ühte andmebaasi, siis mikroteenused võivad andmeid jaotada mitme andmebaasi vahel. Nagu näidatud joonisel 4, kasutatakse monoliitse rakenduse puhul ühte andmehaldustehnoloogiat. Rakenduse osadele on loodud vajalikud andmetabelid, kuid kõik tabelid asuvad samas andmebaasis. [6]

Mikroteenuste puhul valitakse andmebaas vastavalt sellele, mis antud teenusega kõige paremini kokku sobib. See tähendab, et iga teenus kasutab andmete hoiustamise tehnoloogiat, mida parasjagu parimaks peab. Samas pole ka välistatud, et erinevad teenused kasutavad ühist andmebaasi. Arendusmeetmetele sarnaselt hinnatakse ka andmehalduses kõrgelt seda, kui on võimalik kasutada erinevaid tehnoloogiaid. Iga käibel olev tehnoloogia on mingis kindlas valdkonnas teistest parem, seega andmebaasi valik sõltub rakenduse ülesandest. [6]

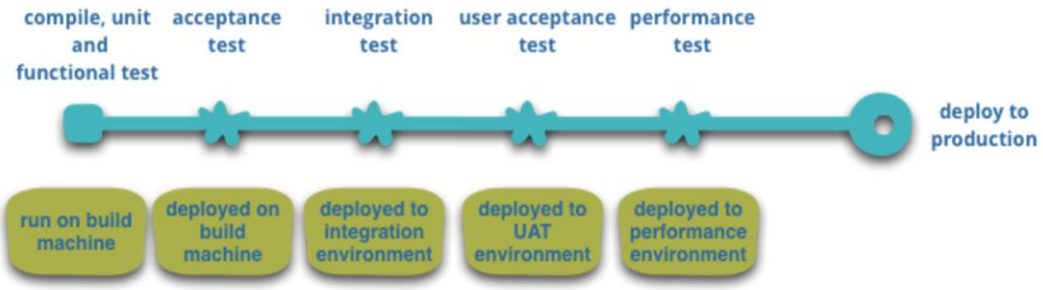


Joonis 4. Andmebaaside kasutus. [6]

7. Infrastruktuuri automatiseerimine

Üldiselt tahavad arendajad kindlad olla, et enne tootmisesse laskmist funktsioneerib tarkvara veatult. Seetõttu luuakse rakendustele võimalikult palju teste. Kuna erinevate arendusetappide vältel vahetuvad keskkonnad, milles rakendus töötab, oleks vaja rakendusi automatiseerida. Rakendus on automatiseeritud juhul, kui teda saab igas arenguetapis serverisse paigaldada. [6]

Nagu jooniselt 5 ilmneb, siis teenuse arendamine jaotub viide faasi. Esimeses faasis pannakse rakendus käima ning sooritatakse funktsionaalsed ja ühiktestid. Teises etapis paigaldatakse rakendus masinasse, kus seda parasjagu arendatakse. Lisaks viiakse läbi vastuvõtutest kontrollimaks, et tarkvara poolt püstitatud nõuded oleks täidetud. Kolmandas etapis paigaldatakse rakendus arendatavast arvutist mingisse teise keskkonda ning kontrollitakse selle töötavust integratsioonitestiga. Integratsioonitestis liidetakse eraldiseisvad moodulid kokku ning kontrollitakse nende ühist töötamist. Neljandas faasis paigaldatakse rakendus tellija poolt kasutatavasse keskkonda ning viiakse läbi test, milles tellija kontrollib antud rakenduse kasutatavust. Viiendas ja viimases faasis paigaldatakse toode keskkonda, kus see päriselt tööle hakkab. Samal ajal kontrollitakse veelkord toote töökindlust. Pärast nimetatud faaside läbimist ongi toode täielikult valmis. Infrastruktuuri automatiseerimist kasutatakse põhiliselt mikroteenuste puhul. Monoliitse teenuse arendamisel kasutatakse seda reeglina nii kaua, kuni koodihulk muutub liiga suureks ning puudub täielik kontroll arendatava teenuse üle. [6]



Joonis 5. Rakenduse arendusetapid. [6]

8. Vea tuvastamine

Mikroteenuste arendamisel on tõrgetega toimetulek tähtsal kohal. Kuna teenused võivad igasuguse probleemi korral suvalisel ajahetkel katki minna, on oluline tuvastada tõrked koheselt ning võimaluse korral teenus automaatselt taastada. Samas peab valmistuma ka selleks, et tõrke likvideerimine ei toimu koheselt. Nii monoliitsete kui mikroteenustel põhinevate rakenduste puhul pühendatakse palju aega reaalajas monitooringule, kontrollides nii arhitektuurilisi elemente (kui palju päringuid sekundis andmebaasile saadetakse) kui ka ärilises mõttes olulisi näitajaid (kui palju tellimusi minutis esitati). Taoline monitooring aitab kaasa vea tuvastamisele varajases faasis. Mida kiiremini viga avastatakse, seda kiiremini viga parandatakse. [6]

9. Järeldus

Tabel 1. Monoliitse ja mikroteenustel põhineva arhitektuuri võrdlus.

Võrreldav näitaja	**Mikroteenustel põhinev arhitektuur**	**Monoliitne arhitektuur**
Komponentideks jagamine	Ühele komponendile vastab reeglina üks teenus	Ühele komponendile vastab mitu omavahel seotud teenust
Meeskondade moodustamine	Meeskondade moodustamine toimub teenusepõhiselt	Meeskonnad on komplekteeritud vastavalt arendusvaldkonnale
Tugevam side arendatud tarkvaraga	Tarkavara käsitletakse kui toodet	Tarkavara käsitletakse kui projekti
Targad otspunktid või tark kanal	Targad otspunktid	Targad otspunktid või tark kanal
Arendusmeetmete tsentraliseeritus	Kasutusel võivad olla erinevad tehnoloogiad	Kasutusel üks tehnoloogia tarkvaraarenduseks

Võrreldav näitaja	Mikroteenustel põhinev arhitektuur	Monoliitne arhitektuur
Andmehalduse tsentraliseeritus	Kasutusel võivad olla erinevad tehnoloogiad	Kasutab ühte andmebaasi tehnoloogia
Infrastruktuuri automatiseerimine	Täielikult automatiseeritud	Osaliselt automatiseeritud
Vea tuvastamine	Reaalajas monitooring	Reaalajas monitooring

Tabel 1 võrdleb monoliitset ja mikroteenustel põhinevat arhitektuuri kaheksast eri aspektist. Võrdlusest selgub, et monoliitsed ja mikroteenustel põhinevad rakendused on olemuselt küllatki erinevad. Mikroteenuste puhul vastab ühele komponendile üks teenus, mida arendab terve tema eluea jooksul üks kindel meeskond. Meeskond koosneb erinevate erialade spetsialistidest, kes kasutavad teenuse ehitamiseks erinevaid arendusmeetmeid ning andmehaldustehnoloogiaid. Arendatav teenus on täielikult automatiseeritud ning teda saab igas arenguetapis serverisse paigaldada. Mikroteenustel põhineva arhitektuuri korral toimub veatuvastus läbi pideva reaalajalise monitooringu ning kasutatav loogika asub rakenduse otspunktides. Monoliitse rakenduse puhul vastab ühele komponendile mitu teenust, mistõttu ei saa teenuseid iseseisvana käsitleda. Uue teenuse lisamine või vana eemaldamine võib mõjutada mõne teise teenuse funktsioneerimist. Seega võib muudatuste tegemine monoliitse rakenduse korral osutuda küllaltki tülikaks. Meeskondade moodustamine toimub antud arhitektuuri puhul sama valdkonna inimeste ühte gruppi paigutamisel. Sellise lähenemise korral on erineva valdkonna inimeste omavaheline suhtlus rakenduse raames piiratud. Samuti puudub arendajatel lähedus loodud tarkvaraga, sest tarkvara liigub erinevate osakondade vahel. Lisaks on monoliitsete rakenduste arendamine suhteliselt tsentraalne. Nii arendusmeetmete kui andmebaaside kasutamises puudub valikuvabadus, mistõttu ollakse kinni ühes tehnoloogias.

Võttes arvesse kahe arhitektuuri sarnasusi ja erinevusi, tuleb selgelt välja mikroteenustel põhineva arhitektuuri paremus. Mikroteenuseid kasutades on arendajal reaalne kontroll ja vastutus oma töö üle- kaks väärtust, mida efektiivseks arenduseks vaja on.

10. Evolutsioon

Mikroteenuseid kasutavad üldjuhul need, kes on kaasas käinud süsteemiarhitektuuri evolutsiooniga ning näevad teenuste komponentideks võtmises täiendavat vahendit

rakenduse kontrollimiseks. Kontrolli all peetakse silmas seda, et rakendust saab regulaarselt, lihtsalt ja kiiresti muuta. Iga kord, kui süsteeme komponeerida, tuleks mõelda sellele, kuidas jaotada komponente nii, et teised tarkvaraosad sellest kahjustada ei saaks. Mikroteenuste puhul toimub komponentide uuendamine ja välja vahetamine lihtsalt ning teistest osadest sõltumatult. [6]

Rääkides vanadest rakendustest, mida soovitakse üle viia mikroteenustele, oleks mõistlik lähenemine see, kui suure monoliitse tüki ümber ehitada uusi sõltumatuid mikroteenuseid. The Guardiani veebileht on hea näide rakendusest, mis algselt disainiti monoliitsuse põhimõtetel ning mida tänapäeval täiustatakse mikroteenustega. Monoliitne tükk on küll veebilehe tuum, kuid igasugused lisarakendused põhinevad mikroteenustel. Antud lähenemine on eriti efektiivne ajutiste rakenduste puhul. Näiteks veebilehte, mis haldab mingisugust spordiüritust ning on aktuaalne vaid võistluse toimumise hetkel, on võimalik sobilike tööriistadega väga kiiresti ehitada ja üles seada. Samuti saab selle pärast spordiürituse lõppu kergesti ka maha võtta. [6]

Monoliitse ja mikroteenustel põhineva arhitektuuri kombineerimise poole püüdleb ka Telia. Mikroteenustele üleminekuks otsib Telia raamistikku, mis toetaks antud arhitektuuri kasutamist. Kuna ettevõtte poolt kasutatavad süsteemid on enamjaolt arendatud Javas, on esimene nõue raamistiku valikul see, et ta põhineks Javal. Teine nõue on see, et raamistik annaks edu arenduskiiruses ja kvaliteedis. Kuna mikroteenus täidab üldiselt vaid ühte kindlat ärilist funktsiooni, on ta olemuselt lihtne ning erineb teistest mikroteenustest enamasti ainult äriloogika poolest. Seega põhi, millele äriloogikat kirjutatakse, on mikroteenuste puhul üldjoontes sama. Säästmaks arendusele kuluvat aega, otsitakse raamistikku, mis suudab vajaliku põhja ise genereerida. Lisaks aja kokkuhoiule ei pea arendaja iga rakenduse puhul sama aluskoodi uuesti välja kirjutama. Kood on vaikimisi olemas ning konfigureerimine toimub automaatselt. See, et arendaja kirjutab vähem koodi, kahandab ka vigade tekkimise tõenäosust. Mida väiksem on tõenäosus vigade tekkeks, seda vähem kulub energiat tarkvara haldamiseks. Seega antud omadustega raamistik annaks tulemusi mitmes valdkonnas. Telia Eesti AS viis läbi uuringu leidmaks enda jaoks sobivamaid raamistikke. Raamistikud, mis vastasid eelpool mainitud kriteeriumitele ning jäid uuringujärgselt valikusse, on Dropwizard ja Spring Boot. Selleks, et saada Dropwizardist ja Spring Bootist parem ülevaade, arendatakse käesolevas töös mõlema raamistikuga lihtne veebirakendus.

Rakendus peab võimaldama andmete sisestamist ja kuvamist. Järgnevalt kirjeldatakse samme, mida rakenduste arendamiseks tuleb teha.

3 Praktiline osa

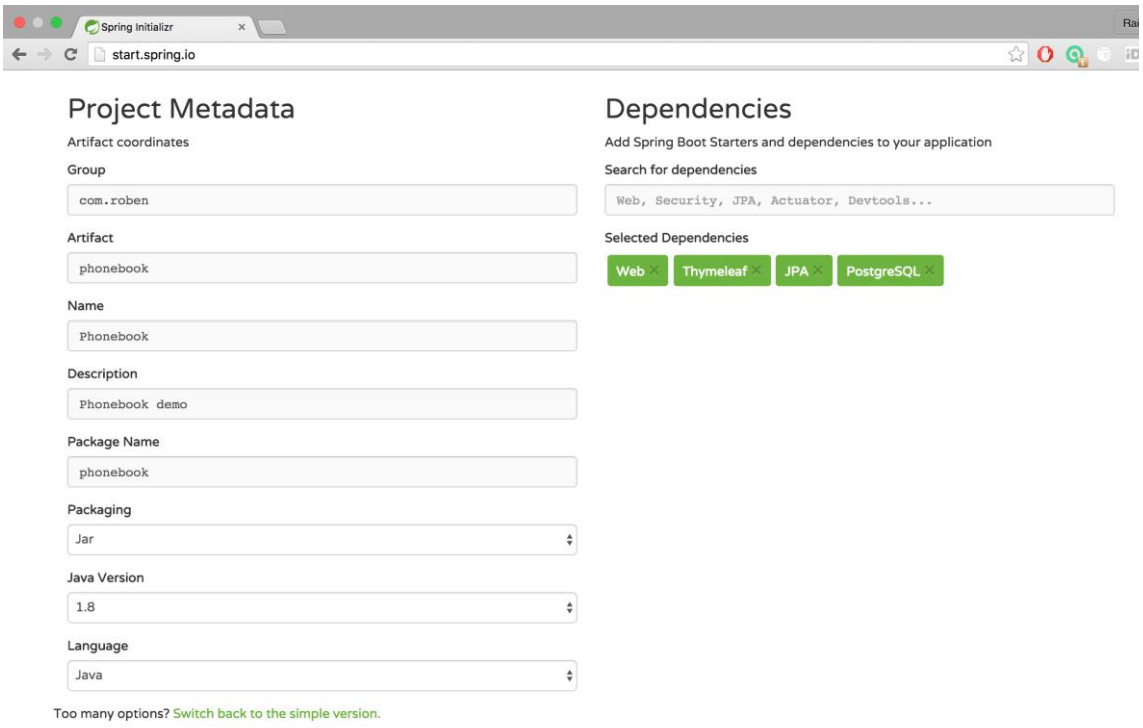
3.1 Rakenduse arendus Spring Bootiga

Raamistiku õppimiseks ning rakenduse arendamiseks kasutas töö autor raamatut "Spring Boot in action". [7]

Järgnevalt kirjeldatakse seda, kuivõrd vähese vaevaga on võimalik Spring Booti kasutades ehitada lihtne veebirakendus. Täpsemalt ehitatakse telefoniraamatu teenus, mille abil saab kontakte sisestada ja lugeda.

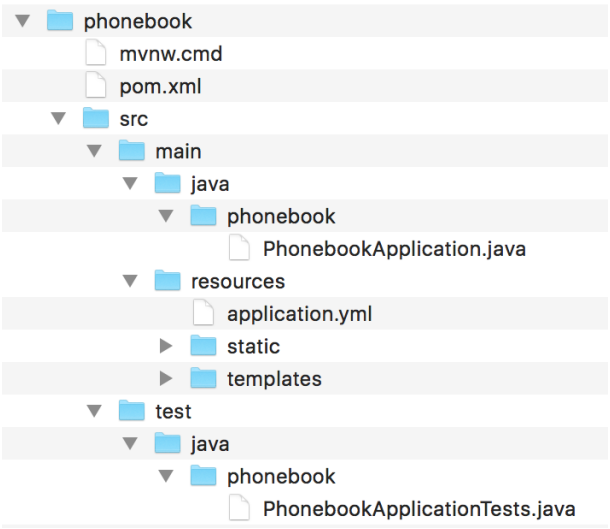
Teenuse arenduskeskkonnaks on valitud STS, mis on kohandatud Springil põhinevate rakenduste arendamiseks. Tehnilise poole pealt vaadates kasutatakse veebipäringute juhtimiseks Spring MVC'd ning rakendusserverina Apache Tomcati. Vaadete kuvamiseks kasutatakse Thymeleafi ning andmebaasiga suhtluse loomiseks Java Persistence API'it ja Hibernate'i. Andmebaasiks on valitud PostgreSQL. Rakenduse ehitamiseks kasutatakse Apache Mavenit.

Selleks, et eelpool välja toodud pluginaid projekti lisada, ei pea Spring Booti puhul palju vaeva nägema. Spring Initializeriga, mis asub aadressil http://start.spring.io/, on võimalik genereerida täpselt selline alusprojekt nagu ise soovid. Töö autori eelistusi arvesse võttes moodustati projekt joonisel 6 kuvatud andmetega.



Joonis 6. Spring Initializer'i veebiliides.

Spring Initializeri poolt genereeritud projekti struktuur on kuvatud joonisel 7.



Joonis 7. Spring Boot'i alusprojekti struktuur

Nagu näha, siis rakenduse põhikood on paigutatud src/main/java harusse, ressursid on paigutatud src/main/resources harusse ning testkood asub src/test/java harus. Lisaks on jooniselt näha erinevaid faile, mis projekti genereerimisega kaasa tulid. Pom.xml'is kirjeldatakse seda, kuidas projekti Maveniga kokku panna. PhonebookApplication.java paneb rakenduse käima ning application.yaml on koht, kus saab kirja panna Spring

Booti seadistused. PhonebookApplicationTests.java on tavaline testklass, mida antud projekti raames ei täiendata.

Tutvumaks Spring Booti eripäradega, näidatakse järgnevalt nende klasside või failide sisu, milles vastavat raamistikku on kasutatud.

Esmalt võetakse vaatluse alla PhonebookApplication klass (Lisa 1). Tähelepanelikult vaadates on näha, et enne klassi deklareerimist, kirjutatakse annotatsioon @SpringBootApplication. Antud annotatsioon võimaldab automaatse seadistuse kasutamist Spring Bootiga.

Järgnevalt uuritakse rakenduse kompileerimise faili pom.xml'i (Lisa 2). Antud fail on näha ka joonisel 7 kuvatud rakenduse struktuuris. Spring Boot pakub rakenduse ehitamise lihtsustamiseks erinevaid pluginaid. All olevas koodis on välja toodud pom.xml'is kajastuvad pluginad, mille pani kaasa projekti loomisel Spring Initializer.

```
<parent>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-parent</artifactId>
	<version>1.3.3.RELEASE</version>
	<relativePath/> <!-- lookup parent from repository -->
</parent>
```

Antud koodi vaadates on näha, kui lihtsasti toimub Spring Booti deklareerimine. Rakenduse looja ei pea sobiliku Spring Booti versiooni numbrit ise teadma, sest see saadakse kaasa spring-boot-starteriga. Varasemalt, Springi kasutades, oleks pidanud verisooni numbrit ise teadma.

Spring Booti versiooni defineerimisele lisaks toimub pom.xml'is sõltuvuste haldamine. Paljud arendaja jaoks vajalikud sõltuvused on kaasa antud spring-boot-starteriga. Seega tõenäosus, et rakenduse arendamiseks vajaminev sõltuvus on spring-boot-starteriga juba päritud, on suur. Piisab vaid lihtsast deklareerimisest pom.xml'is. Springi Booti poolt lisatud sõltuvusi saab enamasti välja kutsuda kas plugina tootjanime või osutatava funktsionaalsuse järgi.

Järgnevas koodis on näidatud, kuidas pom.xml'is kajastatakse sõltuvusi.

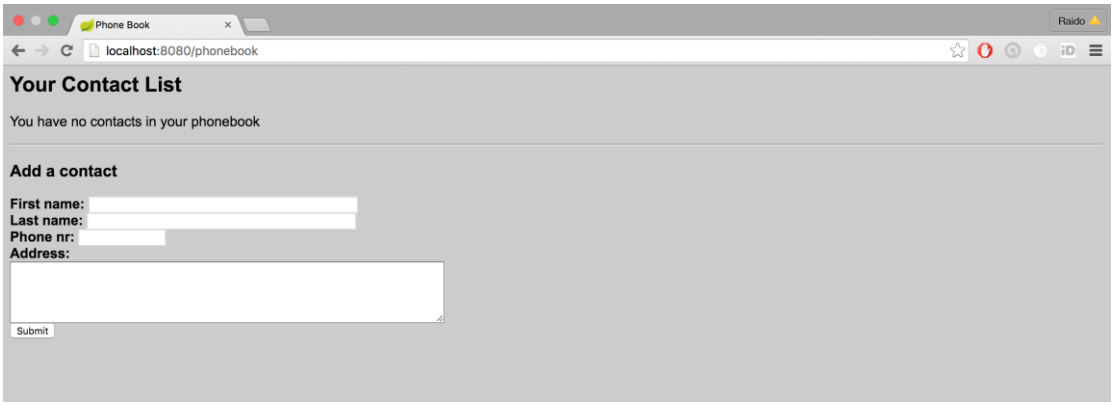
```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
	<groupId>org.postgresql</groupId>
	<artifactId>postgresql</artifactId>
	<scope>runtime</scope>
</dependency>
```

Nagu jooniselt 6 võis näha, siis telefoniraamatu teenuse arendamiseks kasutatakse JPA'd, Thymeleafi ning PostgreSQL'i. Antud lisad on tootjanime järgi kirjeldatud ka sõltuvuste lahtris. Erilist tähelepanu tuleks pöörata sellele, kuivõrd lihtne on pluginaid deklareerida. Mitte ühegi plugina puhul pole vaja kasutatavat versiooni, artifact id´d ega group id´d eraldi täpsustada. Kõik see päritakse spring-boot-starteriga automaatselt. Teisisõnu öeldes peab ilma Spring Bootita läbi viima põhjaliku uuringu selle kohta, millist plugina versiooni kasutada, millised group ja artifact id´d mingile sõltuvusele sobivad ning kas kõik lisatud sõltuvused vastavate versioonidega suudavad üldse koos töötada. Kõigele muule lisaks otsib Spring Boot sõltuvuste poolt kasutatavad teegid koos õigete versioonidega, lisades need ka projekti faili.

Selleks, et rakendus tööle hakkaks, tuleb projektile juurde lisada uusi klasse. Arvestades seda, et antud rakenduse keskseks objektiks hakkab olema telefoniraamatusse lisatav kontakt, siis järgnevalt luuaksegi klass Contact. (Lisa 3)

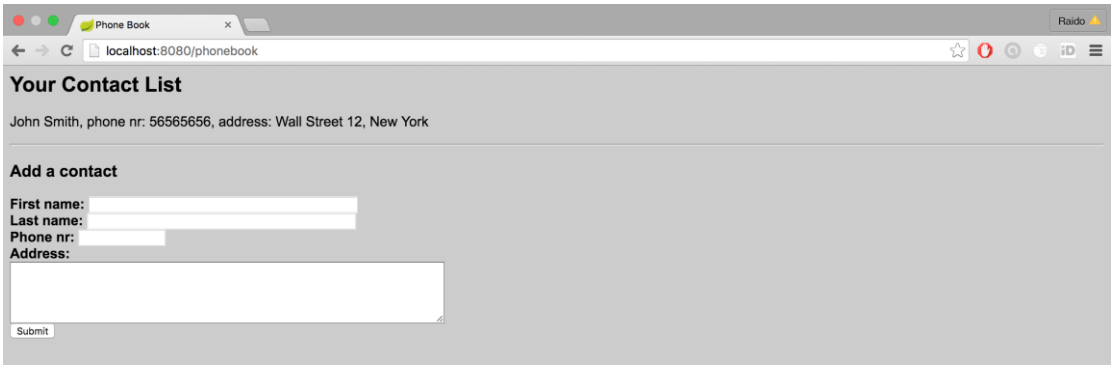
Selleks, et telefoniraamatu kontaktid oleksid jäädavalt olemas, defineerime liidese PhonebookRepository (Lisa 4), millega toimub kontaktide salvestamine andmebaasi. Antud liides kasutab Spring Data JPA'd. Selleks, et salvestada kontakte andmebaasi, tuleb luua andmebaasiga ühendus. Andmebaasi ühenduse loomine on seadistatud application.yaml failis (Lisa 5). Lisaks on vaja kirjutada SQL laused, millega luuakse andmebaasi kontaktide tabel (Lisa 6).

Kuna telefoniraamatut soovitakse HTML'is kuvada, tuleb luua rakenduse veebipoolne vaade. Esmalt kirjeldatakse kontrolleri klass PhonebookController (Lisa 7). Kontroller on vahelüli andmebaasi ja kasutajaliidese vahel. Kasutajaliides annab kontrollerile teada, milliseid andmeid ta kuvada soovib ning kontroller teeb andmete saamiseks päringu andmebaasile. Andmebaasist saadud andmed saadab kontroller kasutajaliidesele, mis kuvab telefoniraamatu koos kontaktidega HTML vaates. Järgmisena luuakse rakenduse vaade phonebook.html (Lisa 8), mis asub rakenduse struktuuris src/main/resources/templates kataloogis. Nüüd on ka kõik vajalikud tükid rakenduse töötamiseks olemas. Rakenduse käivitamiseks kasutatakse STS'i, mis lokaalserveril pordiga 8080 paneb rakenduse tööle. Minnes aadressile http://localhost:8080/ on kuvatud loodud telefoniraamat nii nagu näidatakse joonisel 8.



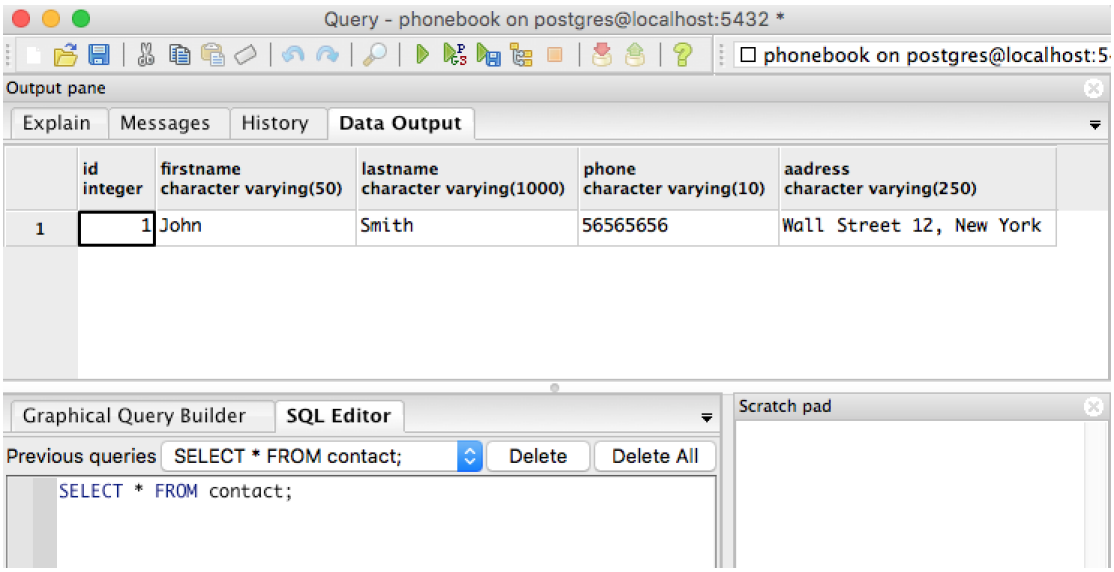
Joonis 8. Telefoniraamatu vaade ilma kontaktideta.

Algselt on telefoniraamat tühi. Seetõttu lisatakse telefoniraamatusse kontakt, kelle eesnimi on "John", perekonnanimi "Smith", telefoninumber "56565656" ning aadress "Wall Street 12, New York". Pärast kontakti lisamist näeb telefoniraamat välja selline nagu on näidatud joonisel 9.



Joonis 9. Telefoniraamat koos esimese kontaktiga.

Lisatud kontakti saab näha ka andmebaasist. Selleks kasutatakse PostgreSQL andmebaasi tööriista pgAdmin III ning kirjutatakse sobilik päring andmete kuvamiseks. Jooniselt 10 on näha, et kontakt on andmebaasi edukalt salvestatud.



Joonis 10. Andmebaasi päring kontaktide kuvamiseks.

3.2 Rakenduse arendus Dropwizardiga

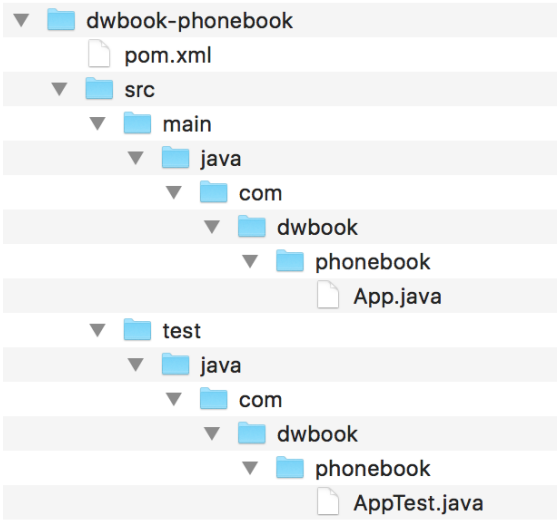
Raamistiku õppimiseks ning rakenduse arendamiseks kasutati raamatut "RESTful Web Services with Dropwizard". [8]

Järgnevalt kirjeldatakse seda, kuivõrd kergesti saab Dropwizardiga ehitada lihtsat veebirakendust. Nii nagu Spring Booti puhul, ehitatakse ka Dropwizardiga telefoniraamatu teenus, mille abil saab kontakte sisestada ja lugeda.

Arenduskeskkonnaks valiti Eclipse, mis on kohandatud Javas arendatavate programmide kirjutamiseks. Tehnilise poole pealt vaadates kasutatakse veebipäringute juhtimiseks MVC mudelit ning rakendusserverina Jetty't. Vaadete kuvamiseks kasutatakse Moustache'i ning andmebaasiga suhtluse loomiseks DAO'd. Andmebaasiks on valitud MySQL. Rakenduse ehitamiseks kasutatakse Apache Mavenit.

Esimese sammuna genereeritakse rakenduse alusprojekt, millega määratakse rakenduse struktuur. Projekti genereerimiseks avatakse arvutis olev käsurea programm, liigutakse sobivasse kausta ning sisestatakse järgnev käsk:

```
mvn archetype:generate
-DgroupId=com.dwbook.phonebook
-DartifactId=dwbook-phonebook
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```



Joonis 11. Dropwizardi alusprojekti struktuur.

Nagu jooniselt 11 on näha, siis rakenduse põhiprogramm on paigutatud src/main/java harusse ning testprogramm asub src/test/java harus. Lisaks on näha erinevaid projekti töötamiseks vajaminevaid faile. Pom.xml'is (Lisa 9) on kirjutatud informatsioon projekti seadistuse ja sõltuvuste osas. Selleks, et Dropwizardit kasutada, on vaja faili lisada uusi sõltuvusi. App klassis (Lisa 10) toimub rakenduse käivitamine. AppTest on tavaline testklass, mida antud teenuse puhul ei kasutata.

Pärast rakenduse alusprojekti valmimist hakatakse esmalt täiendama pom.xml'i. Kuna Dropwizard põhineb Mavenil, on kõik vajalikud moodulid kättesaadavad Maven Central Repository's. Seega vajalike moodulite alla laadimiseks ja lisamiseks on tarvis määrata õige mooduli id. Antud moodulid võetakse peagi põhjalikuma vaatluse alla. Enne aga räägitakse sellest, kuidas rakenduse ehitus täpsemalt välja näeb ning milliseid klasse ja faile selle jaoks täiendavalt vaja on.

Kuna käesolevas töös ehitatakse telefoniraamatu rakendust, on tarvis implementeerida vajalik funktsionaalsus kontaktide varundamiseks ja haldamiseks. Seega luuakse ContactResource klass (Lisa 11), mis tegeleb HTTP päringute vastu võtmisega ning JSON'is esitatud vastuste genereerimisega. Vastav klass käitub MVC mudelis kontrollerina.

Järgnevalt kirjutatakse klass, mis annab parameetrid REST'i poolt loodud ressursile. Kuna telefoniraamatu keskseks objektiks on kontakt, luuakse klass Contact (Lisa 12), milles kirjeldatakse kontaktile vastavad parameetrid: id, eesnimi, perenimi ja telefoninumber.

Selleks, et telefoniraamatu abil andmeid hoiustada ja välja kuvada, tuleb luua ühendus rakenduse ja MySQL'i andmebaasi vahel. Kui MySQL on arvutisse installitud, saab andmebaasi siseneda läbi käsurea, kirjutades sinna:

```
mysql -u root -p
```

Olles MySQL'i kasutajaga sisse loginud, antakse käsk telefoniraamatu andmebaasi ning kontaktide tabeli (Lisa 13) loomiseks. Lisaks on vaja luua ContactDAO klass (Lisa 14), millega tehakse andmebaasi päringud.

Lihtsustamaks andmebaasi andmerea kaardistamist andmeobjektiks, luuakse klass ContactMapper. (Lisa 15)

Hetkeseisuga on Dropwizardi teenusel olemas vajalik funktsionaalsus, kuid puudub telefoniraamatu kuvamiseks vajaminev liides. HTTP päringute tegemiseks läbi veebilehitseja on vaja luua HTTP klientprogramm. Dropwizardisse on sisse ehitatud Jersey klient, mida antud juhul ka kasutatakse. Järgnevalt luuakse ClientResource (Lisa 16) klass, mis kuulab ja kinnitab HTTP get-päringud. Samuti kutsub ta välja sobiliku meetodi ContactResource klassist ning teisendab saadud vastuse inimsõbralikku formaati.

HTML vaadete loomiseks kasutatakse Mustache'i. Vaadete hoiustamiseks luuakse uus kataloog nimega views, mis asub src/main/resources/views harus. Vastavasse kataloogi tekitatakse contact.mustache fail (Lisa 17), mille ülesanne on kuvada JSON'na leitud kontakti andmed HTML tabelis.

Selleks, et vaadetega suhelda, luuakse ContactView (Lisa 18) klass. ContactView asub src/main/java/views harus. Antud klassist tagastatud kontakti objekt saadetakse Mustache vaatele.

Viimasena võetakse vaatluse alla pom.xml. Vastavas failis kirjeldatakse kõik projektis kasutatavad pluginad.

Selleks, et lisada telefoniraamatu teenusele klientprogramm, on vaja sõltuvustesse lisada dropwizard-client moodul nii nagu on näidatud alljärgnevas koodis:

```
<dependency>
	<groupId>io.dropwizard</groupId>
	<artifactId>dropwizard-client</artifactId>
	<version>0.7.0</version>
</dependency>
```

Mustache on vaadete kuvamise platvorm, mida Dropwizard toetab. Selle kasutamiseks lisatakse sõltuvustesse dropwizard-views-mustache moodul. Pom.xml'i täiendatakse järgneva koodiga:

```
<dependency>
	<groupId>io.dropwizard</groupId>
	<artifactId>dropwizard-views-mustache</artifactId>
	<version>0.7.0</version>
</dependency>
```

Andmebaasiga ühenduse loomiseks peab lisama mooduli MySQL kasutamiseks. Pom.xml'i kirjutatakse alljärgnev kood:

```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<version>5.1.22</version>
</dependency>
```

Samuti läheb vaja dropwizard-jdbi moodulit, mis annab loa andmebaasiga ühendumiseks. Lisatakse dropwizard-jdbi mooduli, mida kirjeldab alljärgnev kood:

```
<dependency>
	<groupId>io.dropwizard</groupId>
	<artifactId>dropwizard-jdbi</artifactId>
	<version>0.7.0</version>
</dependency>
```

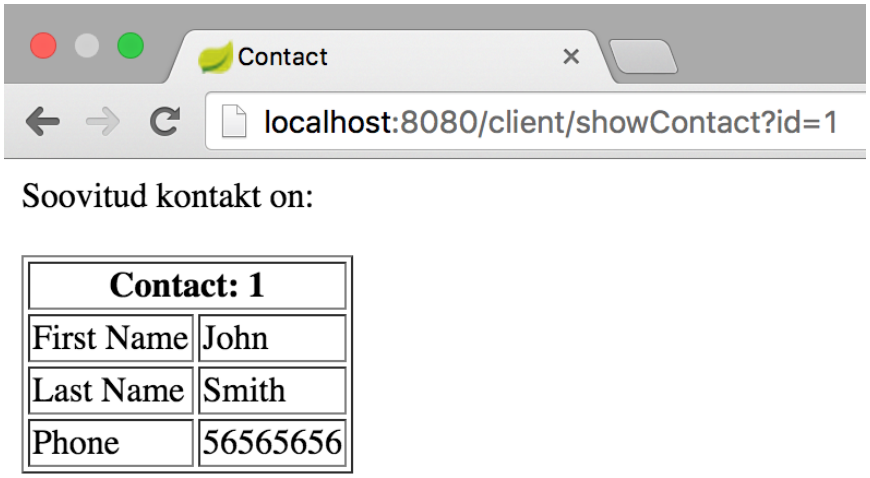
Kuna Dropwizard põhineb Mavenil, mida kasutatakse ka projekti ehitamiseks, siis pom.xml'i lisatakse automaatselt Dropwizardi baassõltuvused ehk dropwizard-core moodul. Sõltuvus on kirja pandud järgmiselt:

```
<dependency>
	<groupId>io.dropwizard</groupId>
	<artifactId>dropwizard-core</artifactId>
	<version>0.7.0</version>
</dependency>
```

Sellega ongi telefoniraamatu teenus valmis. Rakendus pannakse tööle Eclipse'is. Esimese kontakti kuvamiseks tuleb avada veebilehitseja ja minna aadressile: localhost:8080/client/showContact?id=1. Kuna alguses pole andmebaasi ühtegi kontakti lisatud, kuvatakse tühi tabel. Kontakti lisamiseks kasutatakse käsurea tööriista cURL ning kirjutatakse käsk:

```
curl --verbose --header "Content-Type: application/json" -X POST -d
'{"firstName": "John", "lastName":"Smith", "phone":"56565656"}'
http://localhost:8080/contact
```

Käsurida kuvab vastuse, et uus kontakt on andmebaasi lisatud. Minnes uuesti aadressile localhost:8080/client/showContact?id=1, nähakse tabelit esimese kontakti andmetega nii, nagu on näidatud joonisel 12.



Joonis 12. Kontakti kuvamine veebivaatest.

Lisatud kontakti saab näha ka andmebaasist. Selleks sisenetakse käsurealt MySQL andmebaasi ning kirjutatakse sobilik päring andmete kuvamiseks. Jooniselt 13 on näha, et kontakt on andmebaasi edukalt salvestatud.

```
mysql> select * from contact where id=1;
+----+-----------+----------+----------+
| id | firstName | lastName | phone    |
+----+-----------+----------+----------+
|  1 | John      | Smith    | 56565656 |
+----+-----------+----------+----------+
```

Joonis 13. Kontakti kuvamine andmebaasist.

3.3 Spring Booti ja Dropwizardi võrdlus

Spring Boot ja Dropwizard on konkureerivad Java-põhised raamistikud. Selleks, et nende poolt pakutavaid võimalusi paremini mõista, tuleb neid erinevate näitajate alusel võrrelda. Raamistike võrdluseks kasutatud näitajad on valitud vastavalt sellele, mis on Telia Eesti jaoks kõige olulisemad. Kuna mõlema raamistikuga ehitati sarnane rakendus, siis võrdluses lähtutakse lõputöö autori poolsetest tähelepanekutest ja kogemustest. Lisaks tuuakse välja välistest allikatest leitud informatsioon.

1. Kogukonna tugi

On üsna tõenäoline, et koodi kirjutamisel jäädakse varem või hiljem kuskile toppama ning proovitakse internetist abi leida. Seetõttu on oluline, et vastavat raamistikku kasutaks võimalikult suur kogukond. Mida rohkem on raamistiku kasutajaid, seda rohkem on esitatud lahendusi erinevatele probleemidele. StackOverflow'st otsides ilmneb, et Spring Booti kohta on esitatud pea kümme korda rohkem küsimusi kui Dropwizardi kohta. Täpsed numbrid on 9186 [9] ja 939 [10]. Sarnase tulemuseni jõuab ka GitHub'is repositooriumite arvu pärides. Spring Bootiga on seotud 9086 [11] projekti ning Dropwizardiga 1875 projekti [12]. Lisaks kajastab Spring Booti populaarsust ka Google Trends, mille andmed on täpsemalt näha joonisel 13. Mõistet "spring boot" on Google'i otsingumootorisse sisestatud enam kui 8 korda rohkem kui sõna "dropwizard" [13].

Joonis 14. Raamistike populaarsus Google Trends'i järgi. [13]

Telefoniraamatu teenust arendades märkas ka töö autor, et Spring Booti kohta eksisteerib rohkem informatsiooni. Seda nii otsingumootorist probleemidele vastuseid otsides kui ka andmebaasides oleva seonduva kirjandusega tutvudes.

2. Sõltuvuste haldamine

Läbi sõltuvuste saab rakendusele lisada uusi pluginaid, mis kasvatavad rakenduse arendamise valikuvõimalusi. Seetõttu on tähtis, et sõltuvuste sisestamine toimuks kiirelt ja mugavalt. Dropwizardil on mitu põhimoodulit, mida saab Mavenit kasutades projekti importida. Sellegipoolest peab olema valmis, et mõningad moodulid ei sobi üksteisega kokku. Antud vastuolu täheldati ka telefoniraamatu teenust arendades. Nimelt pidi iseseisvalt otsima igale sõltuvusele töötavaid ning teiste sõltuvustega kokku sobivaid versiooninumbreid. Spring Booti puhul oli sõltuvuste sisestamine äärmiselt lihtne. Genereerides Spring Initializeriga alusprojekti, saab valida, milliseid pluginaid soovitakse projektis kasutada. Samas saab pluginaid lisada ka pärast projekti genereerimist. Spring Booti on ehitatud palju sõltuvusi, mille lisamiseks pole vaja versiooninumbreid teada. Tuleb märkida vaid vajamineva plugina nimi ning Spring Boot automaatselt otsib sobiliku töötava versiooni.

3. Alusprojekti loomise lihtsus

Mikroteenuste puhul on tähtsal kohal arenduskiirus. Seetõttu soovitakse, et koodiosad, mis igal teenusel on vaikimisi samad, oleks alusprojektiga kaasa antud. Antud juhul saab arendaja keskenduda sellele, mis on kõige tähtsam- äriloogikale. Dropwizardi puhul toimus projekti genereerimine arhetüübi alusel läbi käsurea. Spring Booti puhul genereeriti alusprojekt Spring Initializeriga, mis teeb projektide moodustamise ülimalt mugavaks. Spring Initializeriga sai kaasa anda ka rakenduses vajaminevad pluginad, mida Dropwizardi käsurea lahendus ei võimaldanud.

4. Meetrikud

Olgugi, et telefoniraamatu arendamisel antud valdkonda ei puudutatud, on meetrikud mikroteenustest rääkides tähtsal kohal. Meetrikud aitavad vältida tõrkeid. Tõrgetest hoidumiseks tuleb rakendust pidevalt monitoorida. Kuna Dropwizard on algselt arendatud Coda Hale'i poolt, siis pärandati talle ka ettevõtte poolt välja töötatud lahendused: meetrikud. Meetrikute moodulit on Dropwizardi rakendusele lihtne juurde liita. Lisaks pakub moodul erinevaid võimalusi projekti jälgimiseks. Üldiselt on Dropwizardi meetrikutes olemas kõik populaarsemad mõõdikud, kuid võimalus on neid ka omalt poolt juurde ehitada. Spring Boot pakub meetrikute valdkonnas vaid kõige algelisemaid mõõdikuid. Mõõdikute hulga poolest edestab Dropwizard Spring Booti ülekaalukalt. Sellepärast on Spring Booti ametlikus dokumentatsioonis kirjas, et rakenduse parema kontrollitavuse saavutamiseks on soovituslik kasutada Dropwizardi poolt pakutavaid meetrikuid. [14]

5. Pidev integreerimine

Pidev integreerimine on tarkvaraarenduses kasutatav tava, mille puhul väikesed muudatused on testitud ja raporteeritud enne, kui nad lisatakse töötavasse koodi. Antud lähenemise eesmärk on pakkuda kohest tagasisidet. Kui töötavasse rakendusse lisatud kood on vigane, suudetakse see kiiresti tuvastada ning ära parandada. Üldjoontes sõltub pidev integreerimine meeskondade töökorraldusest, kuid mingil määral on abiks ka kasutatava raamistiku valik. Tähtis aspekt pideva integreerimise juures on jälgida, et rakendust oleks peale igat muudatust lihtne ehitada. Võimaluse korral võiks ehituseks

kuluda minimaalne arv käske. Kuna mõlemad rakendused kasutavad Mavenit, saab neid ehitada käsuga:

```
mvn package
```

Lisaks, kuna mõlemad raamistikud on Java-põhised, siis on neile võimalik lisada tööriistu, mis aitavad jälgimist paremini rakendada. Pluginaid, mis antud lähenemist hõlbustaks, pole kummagi raamistiku puhul eraldi sisse ehitatud. Kasutada saab kolmanda osapoole lahendusi. Seega antud näitaja puhul on raamistikud võrdsed. [15]

3.4 Järeldused

Tabel 2. Spring Booti ja Dropwizardi võrdlus.

Võrreldavad näitajad	**Spring Boot**	**Dropwizard**
Kogukonna tugi	Kõrgem	Madalam
Sõltuvuste haldamine	Lihtsam	Keerulisem
Alusprojekti loomise lihtsus	Lihtsam	Keerulisem
Meetrikud	Vähesel määral	Väga palju erinevaid
Pidev integreerimine	Raamistik eraldi ei toeta, on võimalik kasutada 3. osapoole pluginaid	Raamistik eraldi ei toeta, on võimalik kasutada 3. osapoole pluginaid

Kuna tegu on tehnoloogiatega, mis pärast Telia-poolset uuringut sõelale jäid, siis olgu lisatud, et mõlemad raamistikud sobivad väga hästi mikroteenuste arendamiseks. Sellegipoolest on töö eesmärk leida parim raamistik. Sobivaima raamistiku leidmiseks võetakse arvesse näitajad, mis on Telia jaoks kõige tähtsamad ning võrreldakse tulemusi, mis raamistike puhul ilmnesid. Võrreldavad näitajad on välja toodud tabelis 2.

Spring Boot on Springi edasiarendus, mis väljastati 2014. aastal. Tegu on järjest populaarsemaks muutuva raamistikuga, mille puhul on lihtne hallata sõltuvusi ning luua uus alusprojekt. Dropwizard on eraldiseisev Java raamistik, mis on loodud 2012. aastal. Kui 2014. aastal oli ta Spring Bootist populaarsem, siis 2016. aastaks on Spring Boot Dropwizardist umbes 8-10 korda populaarsem. Seda nii StackOverflow, GitHubi kui Google Trendsist saadud andmete järgi. Spring Booti suuremat kasutajate arvu märkas ka töö autor. Kahe raamistiku võrdluses sai internetist sarnaste küsimuste esitamisel Spring Booti puhul rohkem vasteid ning jõuti kiiremini lahendusteni.

Kui Dropwizard toetab kolmandatelt osapooltelt saadud pluginate kasutamist, siis Spring Booti on paljud sõltuvused varasemalt sisse ehitatud. Seetõttu on sõltuvuste haldamine Spring Bootiga tunduvalt lihtsam. Peamiselt väljendub lihtsus versiooninumbrite märkimises. Kui Spring Bootiga on sõltuvuste deklareerimiseks vaja kasutatava tehnoloogia nime, siis Dropwizardiga on kindlasti vaja teada ka antud tehnoloogia töötavat versiooninumbrit. Kui rakenduses kasutatavaid tehnoloogiaid on vähe, ei pruugi versioonide märkimine erilist tüli tekitada. Kui aga erinevaid pluginaid on palju, võib versioonide kokku sobitamine tekitada suuri probleemi. Versiooninumbri otsimisega puutus kokku ka töö autor. MySQL'i jaoks sobiliku versiooni leidmisele kulus arvestatav hulk aega.

Alusprojekti loomist toetavad mõlemad raamistikud. Sellegipoolest toimub Spring Bootiga alusprojekti loomine mõnevõrra lihtsamalt. Spring Booti poolt kasutatav Spring Initializer võimaldas projekti genereerimise vaid paari valiku sisestamisega. Dropwizardi puhul toimub alusprojekti genereerimine läbi käsurea, mis kasutajasõbralikkuse poolest on Spring Initializerist tunduvalt ebamugavam.

Siiamaani on jäänud mulje, et Spring Boot on Dropwizardist igapidi üle. Meetrikutest rääkides kerkib aga esile Dropwizard. Kui Dropwizardile on sisse ehitatud laialdases valikus erinevaid mõõdikuid, siis Spring Bootiga tulevad kaasa vaid mõned üksikud mõõdikud. Mõlema raamistiku puhul saab mõõdikuid ka juurde ehitada. Spring Booti kaitseks võib öelda nii palju, et Dropwizardi poolt loodud meetrikuid saab Spring Bootil põhinevale rakendusele juurde liita.

Viimasena uuriti pideva integreerimise põhimõtte rakendatavust erinevatele raamistikele. Olgugi, et antud idee kasutamine kajastub peamiselt arendusmeeskonna töökorralduses, on seda võimalik kinnistada ka mitteinimlikke vahendeid kasutades. Nagu välja tuli, siis Spring Bootil ja Dropwizardil eraldi ühtegi tööriista, mis antud lähenemist järgida aitaks, sisse ehitatud ei ole. Sellegipoolest on nad mõlemad Java-põhised ning saavad samaväärselt kasutada mõne kolmanda osapoole loodud tarkvara. Antud valdkonnas jäid raamistikud viiki.

Võrdluse käigus ilmneb, et viiest vaatluse all olnud kategooriast tuli kolmel korral võitjaks Spring Boot. Valdkondasid, mille järgi raamistikke võrrelda, on veel ning seetõttu pole õige väita, et Dropwizard oleks Spring Bootist halvem. Tõenäoliselt sõltub

raamistiku headus sellest, milliseid omadusi kõige rohkem hinnatakse. Sellegipoolest võib öelda, et Telia jaoks sobib neist kahest paremini just Spring Boot. Nagu Telia ajaloost on näha, siis varasemalt arendati tarkvara Springiga. Sellega seoses oleks Spring Bootile üleminek igati loogiline jätk. Kõike eelpool mainitud arvesse võttes leiab töö autor, et parim Java-põhine raamistik mikroteenuste arendamiseks Telia Eesti AS näitel on Spring Boot.

4 Kokkuvõte

Telia Eesti puhul on tegu ettevõttega, kes on varasemalt oma süsteeme arendanud monoliitset arhitektuuri kasutades. Kuna antud arhitektuur seab rakenduste uuendamise ja haldamise osas piirangud, siis on ettevõtte soov minna üle arhitektuurile, mis nimetatud probleemid likvideeriks. Lisaks sobivale arhitektuurile on vaja leida ka sobiv arendusraamistik, mis aitaks antud arhitektuuri maksimaalse efektiivsusega teostada.

Bakalaureusetöö esimene eesmärk oli välja tuua mikroteenustest saadav kasu monoliitse arhitektuuriga võrreldes. Töö tulemusena selgus, et mikroteenused pakuvad täpselt seda, mida Telial on vaja: iga rakendus on eraldiseisev üksus, mida saab individuaalselt lisada, kustutada või uuendada. Kui monoliitse arhitektuuri puhul toimub tarkvaraarendus etapphaaval, kus üks meeskond tegeleb arendusega ning teine meeskond haldusega, siis mikroteenuste puhul tegeleb sama meeskond teenusega terve selle eluea vältel. Antud lähenemise puhul seostatakse tarkvara kindla autoriga, mille abil tugevneb side arendajate ja tarkvara vahel. Kuna arendajad pakuvad ka kasutajatuge, siis on neil selgem ülevaade antud tarkvara tugevustest ja nõrkustest. Lisaks ilmnes, et mikroteenused ei ole sõltuvad ühest arendusmeetmete ja andmehalduse tehnoloogiast. Mikroteenuste puhul on võimalus valida iga probleemi lahendamiseks sobivaim tehnoloogia. Töö tulemusena selgus, et erinevalt monoliitsetest rakendustest on mikroteenused täielikult automatiseeritud: neid saab iga arendustsükli lõpus serverisse paigaldada ning jooksutada.

Töö teine eesmärk oli leida Telia Eestile parim Java-põhine raamistik mikroteenuste arendamiseks. Raamistike leidmiseks viis Telia läbi uuringu, mille tulemusel jäi sobilike valikutena alles Spring Boot ja Dropwizard. Vastavate raamistike võrdluses ilmnes, et Spring Boot on Telia jaoks parim variant mikroteenuste arendamiseks. Antud arhitektuuri paremus tuli välja Telia poolt määratud näitajate võrdluses. Spring Booti kasutajate hulk on suurem, mille tulemusena on kasutajate poolne tugi laiaulatuslikum. Lisaks on Spring Booti puhul lihtsustatud sõltuvuste haldamine, sest soovitud tehnoloogiate versiooninumbreid pole üldjuhul teada vaja. Tänu Spring Initializerile on

alusprojekti loomist oluliselt lihtsustatud. Dropwizardi puhul toimub uue projekti genereerimine läbi käsurea, mis kahe raamistiku võrdluses on pigem tülikas variant.

Lõputöö eesmärgid saavutati: kahele esitatud probleemile leiti kindlad vastused, toetudes nii autori enda tähelepanekutele kui ka välistest allikatest leitud faktidele.

Mikroteenused lahendavad potentsiaalselt paljusid tarkvarasüsteemidega seotud probleeme. Tegemist on alates 2014. aastast kasutuses oleva terminiga ning seetõttu on ennatlik öelda, kas praegune positiivne tagasiside on püsiv. Edaspidistes töödes saab keskenduda mikroteenuste efektiivsele implementeerimisele. Samuti on võimalik uurida mikroteenuste sidumist monoliitsete rakendustega.

Kasutatud kirjandus

1. Kulbas, H. Kolmekihiline arhitektuur. Tallinn: Telia Eesti AS, 2004
2. Kulbas, H. Telia arhitektuur 2008. Tallinn: Telia Eesti AS, 2008
3. Pärnsalu, M. Mitmekihiline arhitektuur. Tallinn: Telia Eesti AS, 2012
4. Pärnsalu, M. Arhitektuur ja raamistik. Tallinn: Telia Eesti AS, 2012
5. Monolithic vs MicroService Architecture.
   [WWW] https://www.linkedin.com/pulse/20141128054428-13516803-monolithic-vs-microservice-architecture (06.04.2016)
6. Microservices.
   [WWW] http://martinfowler.com/articles/microservices.html (07.04.2016)
7. Walls, G. Spring Boot in Action. New York : Manning, 2015.
8. Dallas, A. RESTful Web Services with Dropwizard. Birmingham : Packt, 2014.
9. Spring Boot Tagged Questions.
   [WWW] https://stackoverflow.com/questions/tagged/spring-boot (17.05.2016)
10. Dropwizard Tagged Questions
    [WWW] https://stackoverflow.com/questions/tagged/dropwizard (17.05.2016)
11. Spring Boot Repository Results
    [WWW] https://github.com/search?utf8=%E2%9C%93&q=spring+boot (17.05.2016)
12. Dropwizard Repository Results
    [WWW] https://github.com/search?utf8=%E2%9C%93&q=dropwizard(17.05.2016)
13. Google Trends
    [WWW] https://www.google.com/trends/explore#q=dropwizard%2C%20spring%20boot&cmpt=q&tz=Etc%2FGMT-3 (17.05.2016)
14. Spring Boot and Dropwizard in microserces development. [WWW] http://www.schibsted.pl/blog/spring-boot-and-dropwizard-in-microservices-development/ (06.05.2016)
15. Continuous Integration.
    [WWW] http://martinfowler.com/articles/continuousIntegration.html (12.05.2016)

Lisa 1- PhonebookApplication klass

```
package phonebook;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.web.servlet.config.annotation.ViewControll
erRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfig
urerAdapter;

@SpringBootApplication
public class PhonebookApplication extends
WebMvcConfigurerAdapter {

    public static void main(String[] args) {
        SpringApplication.run(PhonebookApplication.class,
args);
    }

    @Override
    public void addViewControllers(ViewControllerRegistry
registry) {
      registry.addRedirectViewController("/", "/phonebook");
    }

}
```

Lisa 2- Spring Booti pom.xml fail

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>

      <groupId>com.example</groupId>
      <artifactId>phonebook</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <packaging>jar</packaging>

      <name>phonebook</name>
      <description>phonebook demo</description>

      <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>1.3.3.RELEASE</version>
            <relativePath/> <!-- lookup parent from repository
-->
      </parent>

      <properties>
            <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
            <java.version>1.8</java.version>
      </properties>

      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-data-
jpa</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-
thymeleaf</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-starter-
web</artifactId>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-
web</artifactId>
                <exclusions>
                    <exclusion>

      <groupId>com.fasterxml.jackson.core</groupId> <!--
jacksoni exclusion-->
                    </exclusion>
                </exclusions>
            </dependency>
            <dependency>
                <groupId>com.h2database</groupId>
                <artifactId>h2</artifactId>
                <scope>runtime</scope>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-
test</artifactId>
                <scope>test</scope>
            </dependency>
      </dependencies>

      <build>
            <plugins>
                <plugin>

      <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-
plugin</artifactId>
                </plugin>
            </plugins>
      </build>

</project>
```

Lisa 3- Contact klass Spring Bootiga

```
package phonebook;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Contact {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private Long id;
	private String user;
	private String firstName;
	private String lastName;
	private String phone;
	private String address;

	public String getAddress() {
		return address;
	}

	public void setAddress(String address) {
		this.address = address;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getUser() {
		return user;
	}

	public void setUser(String user) {
		this.user = user;
```

```
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public String getPhone() {
		return phone;
	}

	public void setPhone(String phone) {
		this.phone = phone;
	}


}
```

Lisa 4- PhonebookRepository liides

```
package phonebook;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

public interface PhonebookRepository extends
JpaRepository<Contact, Long> {

      List<Contact> findByUser(String user);

}
```

Lisa 5- application.yaml fail

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/phonebook
    username: postgres
    password: postgres
  jpa:
    database-platform: org.hibernate.dialect.PostgreSQLDialect
```

Lisa 6- Contact tabeli loomise lause PostgreSQL'is

```
create table Contact (
  id serial primary key,
  firstName varchar(50) not null,
  lastName varchar(1000) not null,
  phone varchar(10) not null,
  aadress varchar(250) not null
);
```

Lisa 7- PhonebookController klass

```
package phonebook;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/phonebook")
public class PhonebookController {

  private static final String user = "craig";

      private PhonebookRepository phonebookRepository;

      @Autowired
      public PhonebookController(PhonebookRepository
phonebookRepository) {
            this.phonebookRepository = phonebookRepository;
      }

      @RequestMapping(method=RequestMethod.GET)
      public String phonebookContacts(Model model) {

            List<Contact> phonebook =
phonebookRepository.findByUser(user);
            if (phonebook != null) {
                  model.addAttribute("contacts", phonebook);
            }
            return "phonebook";
      }

      @RequestMapping(method=RequestMethod.POST)
      public String addToContactList(Contact contact) {
            contact.setUser(user);
            phonebookRepository.save(contact);
            return "redirect:/phonebook";
      }

}
```

Lisa 8- Spring Bootiga tehtud HTML fail

```
<html><head>
    <title>Phone Book</title>
    <link rel="stylesheet" th:href="@{/style.css}"></link>
  </head><body>
    <h2>Your Contact List</h2>
    <div th:unless="${#lists.isEmpty(contacts)}">
      <dl th:each="contact : ${contacts}">
        <dt class="contactHeadline">
          <span th:text="${contact.firstName}">First
name</span>
          <span th:text="${contact.lastName}">Last
name</span>, phone nr:
          <span th:text="${contact.phone}">Phone</span>,
          address:
          <span th:if="${contact.address}"
                th:text="${contact.address}">Address</span>
          <span th:if="${contact.address eq null}">
                No address available</span>
        </dt>
      </dl>
    </div>
    <div th:if="${#lists.isEmpty(contacts)}">
      <p>You have no contacts in your phonebook</p>
    </div>
    <hr/>
    <h3>Add a contact</h3>
    <form method="POST">
      <label for="firstName">First name:</label>
        <input type="text" name="firstName"
size="50"></input><br/>
      <label for="lastName">Last name:</label>
        <input type="text" name="lastName"
size="50"></input><br/>
      <label for="phone">Phone nr:</label>
        <input type="text" name="phone"
size="15"></input><br/>
      <label for="address">Address:</label><br/>
        <textarea name="address" cols="80"
rows="5"></textarea><br/>
      <input type="submit"></input>
    </form></body>
</html>
```

Lisa 9- Dropwizardi pom.xml fail

```
<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>com.dwbook.phonebook</groupId>
    <artifactId>dwbook-phonebook</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>dwbook-phonebook</name>
    <url>http://maven.apache.org</url>

    <repositories>
        <repository>
            <id>sonatype-nexus-snapshots</id>
            <name>Sonatype Nexus Snapshots</name>

<url>http://oss.sonatype.org/content/repositories/snapshots</url>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>io.dropwizard</groupId>
            <artifactId>dropwizard-core</artifactId>
            <version>0.7.0</version>
        </dependency>
            <dependency>
     <groupId>io.dropwizard</groupId>
     <artifactId>dropwizard-views-mustache</artifactId>
     <version>0.7.0</version>
   </dependency>
   <dependency>
     <groupId>io.dropwizard</groupId>
     <artifactId>dropwizard-assets</artifactId>
     <version>0.7.0</version>
   </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
```

```
            <version>5.1.22</version>
        </dependency>
        <dependency>
            <groupId>io.dropwizard</groupId>
            <artifactId>dropwizard-jdbi</artifactId>
            <version>0.7.0</version>
        </dependency>
        <dependency>
            <groupId>io.dropwizard</groupId>
            <artifactId>dropwizard-client</artifactId>
            <version>0.7.0</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>1.6</version>
                <configuration>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*.SF</exclude>
                                <exclude>META-INF/*.DSA</exclude>
                                <exclude>META-INF/*.RSA</exclude>
                            </excludes>
                        </filter>
                    </filters>
```

```
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestReso
urceTransformer">

<mainClass>com.dwbook.phonebook.App</mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Lisa 10- App klass

```
package com.dwbook.phonebook;

import com.dwbook.phonebook.resources.ClientResource;
import com.dwbook.phonebook.resources.ContactResource;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import io.dropwizard.Application;
import io.dropwizard.setup.Bootstrap;
import io.dropwizard.setup.Environment;
import org.skife.jdbi.v2.DBI;
import io.dropwizard.jdbi.DBIFactory;
import com.sun.jersey.api.client.Client;
import io.dropwizard.client.JerseyClientBuilder;
import io.dropwizard.views.ViewBundle;

public class App extends Application<PhonebookConfiguration> {

    private static final Logger LOGGER =
LoggerFactory.getLogger(App.class);

    public static void main(String[] args) throws Exception {
        new App().run(args);
    }

      @Override
    public void initialize (Bootstrap<PhonebookConfiguration>
b) {
           b.addBundle(new ViewBundle());
}

    @Override
    public void run(PhonebookConfiguration c, Environment e)
throws Exception {
        LOGGER.info("Method App#run() called");
        for (int i = 0; i < c.getMessageRepetitions(); i++) {
            System.out.println(c.getMessage());
        }
        System.out.println(c.getAdditionalMessage());

        // Create a DBI factory and build a JDBI instance
        final DBIFactory factory = new DBIFactory();
        final DBI jdbi = factory
```

```
                .build(e, c.getDataSourceFactory(), "mysql");
        // Add the resource to the environment
        e.jersey().register(new ContactResource(jdbi,
e.getValidator()));

        // build the client and add the resource to the
environment
      final Client client = new
JerseyClientBuilder(e).build("REST Client");
      e.jersey().register(new ClientResource(client));
    }
}
```

Lisa 11- ContactResource klass

```
package com.dwbook.phonebook.resources;

import com.dwbook.phonebook.representations.Contact;
import com.dwbook.phonebook.representations.ContactList;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.skife.jdbi.v2.DBI;
import com.dwbook.phonebook.dao.ContactDAO;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validator;
import javax.ws.rs.core.Response.Status;

@Path("/contact")
@Produces(MediaType.APPLICATION_JSON)
public class ContactResource {

    private final ContactDAO contactDao;
    private final Validator validator;

    public ContactResource(DBI jdbi, Validator validator) {
        contactDao = jdbi.onDemand(ContactDAO.class);
        this.validator = validator;
    }

    @GET
    @Path("/{id}")
    public Response getContact(@PathParam("id") int id) {
        // retrieve information about the contact with the
provided id
        Contact contact = contactDao.getContactById(id);
        System.out.println("teretere " + contact);
        return Response
                .ok(contact)
                .build();
    }
```

```
    @POST
    public Response createContact(Contact contact) throws
URISyntaxException {
        // Validate the contact's data
        Set<ConstraintViolation<Contact>> violations =
validator.validate(contact);
        // Are there any constraint violations?
        if (violations.size() > 0) {
            // Validation errors occurred
            ArrayList<String> validationMessages = new
ArrayList<String>();
            for (ConstraintViolation<Contact> violation :
violations) {

validationMessages.add(violation.getPropertyPath().toString()
+ ": " + violation.getMessage());
            }
            return Response
                    .status(Status.BAD_REQUEST)
                    .entity(validationMessages)
                    .build();
        } else {
            // OK, no validation errors
            // Store the new contact
            int newContactId =
contactDao.createContact(contact.getFirstName(),
                    contact.getLastName(),
contact.getPhone());
            return Response.created(new
URI(String.valueOf(newContactId))).build();
        }
    }
}
```

Lisa 12- Contact klass Dropwizardiga

```
package com.dwbook.phonebook.representations;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.dropwizard.validation.ValidationMethod;

import org.hibernate.validator.constraints.*;

public class Contact {

    private final int id;

    @NotBlank
    @Length(min = 2, max = 255)
    private final String firstName;

    @NotBlank
    @Length(min = 2, max = 255)
    private final String lastName;

    @NotBlank
    @Length(min = 2, max = 30)
    private final String phone;

    public Contact() {
        this.id = 0;
        this.firstName = null;
        this.lastName = null;
        this.phone = null;
    }

    public Contact(int id, String firstName, String lastName,
String phone) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
    }

    public int getId() {
        return id;
    }
```

```
    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getPhone() {
        return phone;
    }
}
```

Lisa 13- Contact tabeli loomine MySQL'iga

```
CREATE TABLE IF NOT EXISTS `contact` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `firstName` varchar(255) NOT NULL,
  `lastName` varchar(255) NOT NULL,
  `phone` varchar(30) NOT NULL,
PRIMARY KEY (`id`)
)
ENGINE=InnoDB
DEFAULT CHARSET=utf8
AUTO_INCREMENT=1;
```

Lisa 14- ContactDAO klass

```
package com.dwbook.phonebook.dao;

import com.dwbook.phonebook.dao.mappers.ContactMapper;
import com.dwbook.phonebook.representations.Contact;
import org.skife.jdbi.v2.sqlobject.*;
import org.skife.jdbi.v2.sqlobject.customizers.Mapper;

import java.util.List;


public interface ContactDAO {

    @Mapper(ContactMapper.class)
    @SqlQuery("select * from contact where id = :id")
    Contact getContactById(@Bind("id") int id);

      @Mapper(ContactMapper.class)
    @SqlQuery("select * from contact")
    List <Contact> getContactAll();

    @GetGeneratedKeys
    @SqlUpdate("insert into contact (id, firstName, lastName, phone)
values (NULL, :firstName, :lastName, :phone)")
    int createContact(@Bind("firstName") String firstName,
@Bind("lastName") String lastName, @Bind("phone") String phone);

    @SqlUpdate("update contact set firstName = :firstName, lastName
= :lastName, phone = :phone where id = :id")
    void updateContact(@Bind("id") int id, @Bind("firstName") String
firstName, @Bind("lastName") String lastName, @Bind("phone") String
phone);

    @SqlUpdate("delete from contact where id = :id")
    void deleteContact(@Bind("id") int id);
}
```

Lisa 15- ContactMapper klass

```
package com.dwbook.phonebook.dao.mappers;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.skife.jdbi.v2.StatementContext;
import org.skife.jdbi.v2.tweak.ResultSetMapper;
import com.dwbook.phonebook.representations.Contact;

public class ContactMapper implements ResultSetMapper<Contact>
{
   public Contact map(int index, ResultSet r, StatementContext
ctx)
   throws SQLException {
      return new Contact(
      r.getInt("id"), r.getString("firstName"),
      r.getString("lastName"),r.getString("phone"));
   }
}
```

Lisa 16- ClientResource klass

```
package com.dwbook.phonebook.resources;
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import com.dwbook.phonebook.representations.Contact;
import com.dwbook.phonebook.representations.ContactList;
import com.sun.jersey.api.client.*;
import com.dwbook.phonebook.views.AllContactView;
import com.dwbook.phonebook.views.ContactView;

@Produces(MediaType.TEXT_HTML)
@Path("/client/")
public class ClientResource {

    private Client client;

    public ClientResource(Client client) {
        this.client = client;
    }

   @GET
    @Path("showContact")
    public ContactView showContact(@QueryParam("id") int id) {
        WebResource contactResource =
client.resource("http://localhost:8080/contact/" + id);
        Contact c = contactResource.get(Contact.class);

        System.out.println("tere Tali! " + c);

        return new ContactView(c);
    }

    @GET
    @Path("newContact")
    public Response newContact(@QueryParam("firstName") String
firstName, @QueryParam("lastName") String lastName,
@QueryParam("phone") String phone) {
        WebResource contactResource =
client.resource("http://localhost:8080/contact");
        ClientResponse response =
contactResource.type(MediaType.APPLICATION_JSON).post(ClientRe
sponse.class, new Contact(0, firstName, lastName, phone));
        if (response.getStatus() == 201) {
            // Created
```

```
            return Response.status(302).entity("The contact
was created successfully! The new contact can be found at " +
response.getHeaders().getFirst("Location")).build();
        } else {
            // Other Status code, indicates an error
            return
Response.status(422).entity(response.getEntity(String.class)).
build();
        }
    }
}
```

Lisa 17- ContactView klass

```
package com.dwbook.phonebook.views;

import com.dwbook.phonebook.representations.Contact;
import io.dropwizard.views.View;

public class ContactView extends View {
      Contact contact;
      public ContactView(Contact contact) {
            super("/views/contact.mustache");
            this.contact = contact;
      }
      public Contact getContact() {
            return contact;
      }
}
```

Lisa 18- Dropwizardiga tehtud HTML fail

```
<html>
      <head>
            <title>Contact</title>
      </head>
      <body>
      Soovitud kontakt on:
      <br><br>
            <table border="1">
                  <tr>
                        <th colspan="2">Contact:
{{contact.id}}</th>
                  </tr>
                  <tr>
                        <td>First Name</td>
                        <td>{{contact.firstName}}</td>
                  </tr>
                  <tr>
                        <td>Last Name</td>
                        <td>{{contact.lastName}}</td>
                  </tr>
                  <tr>
                        <td>Phone</td>
                        <td>{{contact.phone}}</td>
                  </tr>
            </table>
      </body>
</html>
```