TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Carl-Robert Linnupuu 141877IASB

MODULAARSE MONOLIIDI JA MIKROTEENUSTE ARHITEKTUURI VÕRDLUS

Bakalaureusetöö

Juhendaja: Vladimir Viies
Ph.D
Dotsent

Tallinn 2019

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: Carl-Robert Linnupuu

_____

(kuupäev) *(allkiri)*

Annotatsioon

Modulaarse monoliidi ja mikroteenuste arhitektuuri võrdlus

Käesoleva bakalaureusetöö eesmärk on võrrelda monoliit- ja mikroteenusarhitektuuri ning anda ülevaade kuidas mikroteenuste paremaid külgi juurutada monoliitrakendustesse.

Töö esimene osa kirjeldab mõlemat arhitektuuri, antakse ülevaade süsteemide skaleeruvusest ning mudeli- ja lähtekoodi vahelisest seostest. Teine osa tutvustab erinevaid modulaarsuse printisiipe, mooduli sisutüüpe ja koodi struktureerimise viise. Veel võrreldakse ka koodi struktureerimise ja kapseldamise vahelisi erinevusi. Töö viimases osas käsitletakse modulaarse näidisrakenduse ehitamist. Selleks disainitakse lihtsakoeline arhitektuuri mudel, mis seostatakse lähtekoodiga. Rakendus ehitatakse programmeerimiskeeles Java, Spring Boot raamistikus. Mooduliteks tükeldamine ning sõltuvuste haldamiseks kasutatakse tarkvara automatiseerivat tööriista Gradle.

Lõputöö on kirjutatud eesti keeles keeles ning sisaldab teksti 38 leheküljel, 3 peatükki, 10 joonist, 2 tabelit.

Abstract

Comparison of modular monolith and microservices architecture

The purpose of this thesis is to compare modular monolith with microservices architecture and give an overview how to implement the benefits of microservices into a modular monolith.

The first chapter describes the theoretical side of both architectures, its benefits and downsides. Furthermore we define the concept of scalability and the difference between horizontal and vertical scaling. Moreover we give an overview of model-code cap and how to address this problem.

The second chapter introduces various modularity principles and module content types. Furthermore we give a brief comparison of different kinds of code structuring patterns and what suits best when it comes down to modularity. Finally we compare differences between code structuring and encapsulation.

Using these guiding principles, we visualize simple architecture model which we bind with source code. Also we use “package by component” code structuring approach to ensure a better encapsulation between classes. Monolithic application is written in Java with Spring Boot framework and module linking is done using the Gradle build automation tool.

The thesis is in Estonian and contains 38 pages of text, 3 chapters, 10 figures, 2 tables.

Lühendite ja mõistete sõnastik

PaaS	*(Platform as a Service)* platvormi-põhine pilveteenus
REST	*(Representational State Transfer)* veebi arhitektuuri stiil, mille kaudu süsteemid suhtlevad omavahel
HTTP(S)	*(Hypertext Transfer Protocol Secure)* turvaline hüperteksti edastusprotokoll arvutivõrkudes
Health-check	Kontroll veendumaks, et rakendus on funktsionaalselt toimiv antud ajahetkel
Caching	Protsess hoiustamaks andmeid vahemällu
Gradle	Avatud lähtekoodiga tarkvara ehitamist automatiseeriv tööriist
Spring	Tarkvara rakenduse raamistik
Java	Platvormist sõltumatu objektorienteeritud programmeerimiskeel
JVM	Java virtuaalmasin, mis interpreteerib kompileeritud Java baitkoodi arvutile arusaadavateks instruktsioonideks

Sisukord

Jooniste loetelu

Tabelite loetelu

Sissejuhatus

Sisenedes tänapäeva infotehnoloogia maailma, on pea võimatu ignoreerida teemat „Mikroteenused“. See on stiil, millele toetuvad suured tehnoloogia gigandid ehitamaks kvaliteetset tarkvara. Tunduks justkui, et see on ainuke viis millele lähtuda suurepärase tarkvara ehitamiseks.

Hea disaini välja mõtlemine loomaks head mikroteenus-arhitektuuri on sisuliselt sama vajalik ka hästi-struktureeritud monoliidi loomiseks. Seega see tõstab esile küsimuse: “kui ei suudeta ehitada hästi-struktureeritud monoliiti, siis miks arvatakse, et mikroteenused on selle jaoks vastus?”. [1]

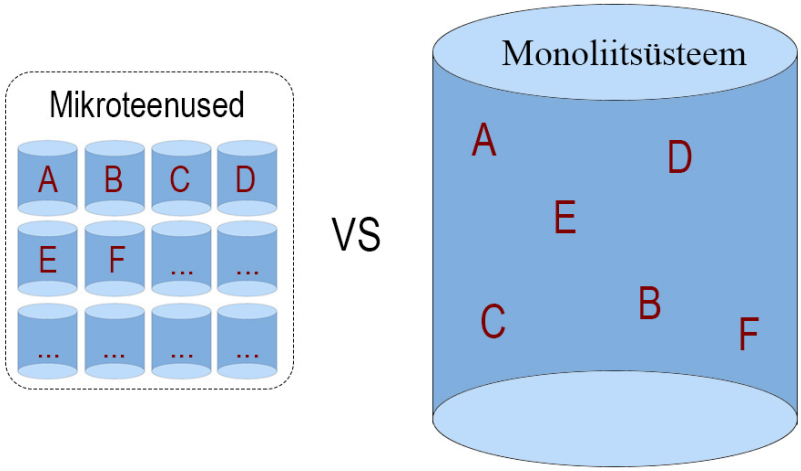
Antud bakalaureusetöö eesmärgiks on võrrelda monoliitarhitektuuri mikroteenus-arhitektuuriga ning selgitada kuidas juurutada mikroteenuste parimaid külgi monoliitsesse süsteemi. Eesmärgi saavutamiseks mõtestatakse lahti mõlema arhitektuuri teoreetiline pool, tuuakse seosed arhitektuuri mudeli ja lähtekoodi vahel ning lõpetuseks dokumenteeritakse modulaarse näidisrakenduse loomist ja selle sisaldavate komponentide kapseldamist.

Töö esimene osa keskendub mõlema arhitektuuri teoreetilisele poolele. Võrdleme nende kahe eesmärke ning eeliseid ja puuduseid. Defineerime süsteemide skaleeritavuse mõiste, võrdleme vertikaalset- ja horisontaalset skaleeruvust ning milliste lisakeerukustega tuleb arvestada mikroteenuste skaleeruvusel. Toome välja olulisemad seosed arhitektuuri mudeli ja lähtekoodi vahel. Selgitame mida tähendab mudel-koodi mittevastavus ning kuidas seda probleemi addresseerida. Töö teises osas defineerime modulaarsuse mõiste tarkvaraarenduses ning kirjeldame ideaalse mooduli sisutüüpe. Samuti analüüsime erinevaid koodistruktureerimisestiile ning selgitame, mis vahe on koodi organiseerimise ja kapseldamise vahel. Lõpetuseks nendele modulaarsusprintsiipidele toetudes, koostame lihtsakoelise e-kaubanduse protsesse kirjeldava arhitektuuri mudeli, mille seostame Java [2] lähtekoodi ja Gradle [3] funktsionaalsusega.

1 Tarkvara arhitektuurid

1.1 Mikroteenuste arhitektuur

Mikroteenuste arhitektuur, või lihtsalt *mikroteenused*, on isepärane meetod arendamaks tarkvara süsteeme, mis püüavad keskenduda eraldi seisvate funktsionaalsete moodulitena, kus on täpselt määratletud liidesed ning tema toimingud. Need aitavad ehitada rakendust väikeste teenuste komplektidena, millest igal teenusel on kindel eesmärk ning on iseseisvalt paigaldatav. Need teenused võivad olla kirjutatud erinevates keeltes ning saavad kasutada erinevaid andmesalvestustehnikaid. [4]



Joonis 1 Mikroteenused vs monoliitsüsteem

Jooniselt 1 on näha, et terviksüsteem on jagatud alamosadeks ehk mikroteenusteks, kus igal teenusel on konkreetne eraldiseisev ülesanne. Vastupidiselt mikroteenustele võime monoliitsüsteemis vaadelda kõiki ülesandeid kui terviksüsteemina.

Teenust võib mõelda kui eraldiseisevat üksust, mida saab paigalda kas teenust pakkuvasse platvormi *PaaS* või see võib olla lihtsalt tema enda operatsioonisüsteemi protsess. Need teenused peavad olema võimelised üksteisest sõltumatult muutuma ning olema paigaldatavad eraldiseisvate tükkidena nii, et teenusest sõltuvad tarbijad ei peaks lisategevusi selleks tegema. [5]

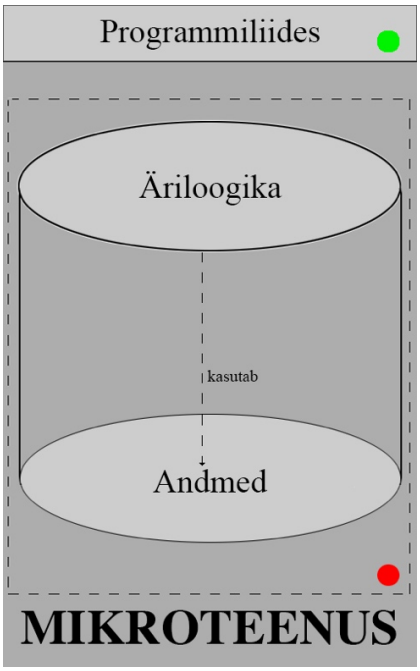
See tähendab seda, et kui näiteks tellimusi protsessiv osa vajab muudatusi, siis nende muudatuste tegemiseks ei pea süsteemi kui tervikut uuesti üles ehitama vaid piisab ainult ühe konkreetse teenuse muutmisest.

Üks populaarseim viis kuidas teenused omavahel suhtlevad on REST vahendusel HTTP(S) protokolli alusel. [6]

Sisuliselt teenus paljastab endas programmiliidese mille kaudu teenust tarbijad kliendid pääsevad ligi teenuse äriloogikale.

1.1.1 Eelised

Üheks suurimaks eeliseks mikroteenuste puhul on see, et teenusega seotud äriloogika on eraldatud programmiliidesest. Kõik teenusest sõltuvad tarbijad saavad ligi läbi tema kehtestatud programmiliidese. Mis tähendab seda, et teenuse sisemine loogika on tarbijate jaoks täielikult kapseldatud. [7]



Joonis 2 Abstraktne mikroteenus

Joonisel 2 on välja toodud illustreeriv mikroteenus, kus rohelise täpiga on näidatud mis on tarbijale kättesaadav ning punasega mis mitte.

Teised suurimad mikroteenuste eelised monoliitsüsteemi suhtes on:

- **Skaleeritavus** – õhukeselt lahtiseotud mikroteenused on parim praktika ehitamaks suuri skaleeruvaid süsteeme. See on eelduseks jõudluse optimiseerimisele, sest see võimaldab valida sobiva ning optimaalse tehnoloogia konkreetse teenuse jaoks. Nõutekohaselt lahtiseotuid teenuseid saab horisontaalselt ja üktseisest sõltumatult suurendada. [8]

- **Eraldi arendatav osa tervikust** – kuna igal mikroteenusel on erinev äritegevuse funktsionaalsus, siis arendusega seotud tegevused on piiritletud kontekstis, kus arendaja saab keskenduda ainult ühele spetsiifilisele äriprobleemile.

- **Koodi kvaliteet** – tarkvara jagamine väikesteks hästi määratletud mooduliteks on sarnane objekt-orienteeritud programmeerimistehnikale, mille mõte on parema koodi taaskasutavuses, koostavuses ning hooldavuses. [8] Tulemuseks on see, et väiksemaid tükke on lihtsam jälgida ning refaktoreerida.

- **Tehnoloogiavabadus** – mikroteenused võivad olla mistahes keeles või tehnoloogiaid kasutades implementeeritud. Tehnoloogiavabadus annab võimaluse elimineerida igasuguse pikaajalise pühendumise tehnoloogiast. Tehes suuri muudatusi olemasolevas süsteemis, on võimalus konkreetne teenus ümber kirjutada kasutades uuemat tehnoloogiat. [9]

- **Kättesaadavus** – mikroteenuste arhitektuur võimaldab kergemini implementeerida katki olevat osa terviksüsteemist. Protseduurid nagu rakenduse *health-check*, *caching* jpm võimaldavad parandada üldist kättesaadavust olevast süsteemist. [8] Näiteks kui ühes teenuses esineb mäluleke, siis ainult üks ja see sama konkreetne teenus on sellest veast mõjutatud ning teised teenused saavad tegevusi jätkata. Vastupidiselt monoliitsüsteemile, võib sarnane probleem katki teha terve süsteemi. [9]

- **Pidevvalmidus ja paigaldus** *(Continuous delivery and deployment)* – mikroteenused annavad võimaluse arendada süsteemi kui tervikut mitmete arendustiimide vahel. Iga tiim omab ja on vastutav ühe või mitme konkreetse teenuse eest. Tiim saab arendada, testida, paigaldada ning skaleerida teenuseid vastavalt vajadusele sõltumatult teistest tiimidest. [9]

1.1.2 Puudused

Nii nagu igal teiselgi arhitektuuristiilil esineb teatud puudusi, siis ei erine selle poolest ka mikroteenuste arhitektuur.

Mikroteenuste arhitektuur lisab keerukust juba projekti asjaolul, et teenused on hajutatud süsteemid [10]. Samuti on mikroteenustel veel üks väga tähtis negatiivne külg, mis võib põhjustada ühe suure arhitektuurilise vea. Selleks on tohutu keerukus teenuste paigaldatavuses ning nende opereerimises. Raskused tulevad esile kui ühe rakenduse asemel on vaja nüüdsest hallata X arv rakendust. [11] See toob omakorda kaasa suurema arendusressursi vajaduse.

Samuti kerkivad esile ka kompetentsed probleemid. Kui rakendused on kirjutatud erinevates programmeerimiskeeltes, kasutades isesuguseid mustreid, siis nii uutel kui ka olemasolevatel arendajatel on küllaltki raske ennast koodibaasi kiiresti sisse süüa.

Mõningad esile tõstetud probleemid, mida mikroteenuste arhitektuur veel pakub:

- **Jagatud süsteem** – selleks, et hästi toimiv suhtlus teenuste vahel toimiks, tuleb hoolikalt valida ning implementeerida protsessisisest kommunikatsioonimehhanismi. Samuti tuleb erinevates teenustes käsitleda ka osalisi tõrkeid ning muid hajutatud veaolukordi. [10]

- **Jagatud transaktsioonid** – mikroteenustel on jagatud andmebaasi arhitektuur. Äriprotsessides, kus tuleb uuendada mitmeid andmebaasi ärimudeleid korraga, peavad muudatused samuti kajastuma ka mitmes erinevas teenust omavas andmebaasis. Jagatud transaktsioonide kasutamine ei ole üldiselt valikuvõimalus ning lõpeb tavaliselt sündmuspõhise lähenemisega, mis on arendajatele palju väljakutsuvam tegevus. [10]

- **Testimine** – testimine mikroteenus-rakenduses muudab keeruliseks teenuste omavaheline sõltuvus. Teenuse testimiseks peab käivitama ka kõik teised teenused, mis on testitava teenusega sõltuvuses (või peab sõltuvuses olevad teenused kontekstist sõltuvalt konfigureerima). [10]

- **Muudatused, mis sõltuvad muudetavast** – monoliitses rakenduses piisab kui muudad sõltuvuses olevaid mooduleid, integreerid muudatused ning paigaldad

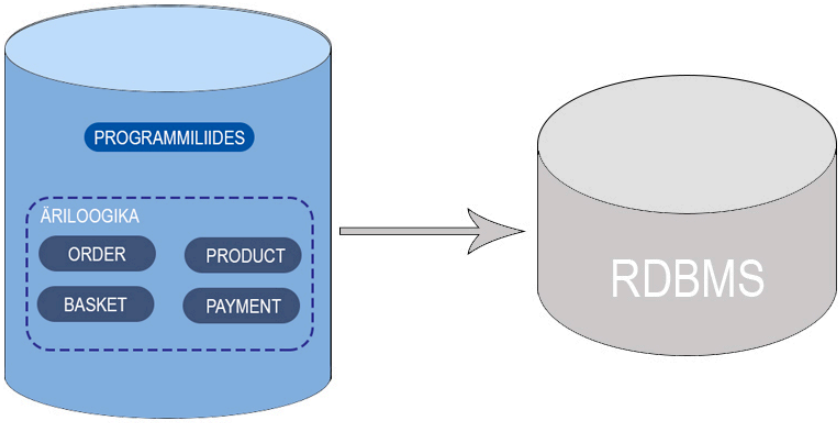
kõik korraga. Mikroteenustes tuleb sama tüüpi muudatust iga teenuse kohta hoolikalt planeerida ning koordineerida. [10] Tüüpiliselt missioonikriitilistes rakendustes pannakse muudatused spetsiaalse lipu taha, mida on võimalik väikse vaevaga sisse-välja lülitada ennetamaks kõige halvemat.

- **Suurenenud mälukasutus** – Mikroteenus-arhitektuur asendab N monoliitrakenduse instantsi N * M teenuste instantsidega. Kui iga teenus jookseb eraldi JVM-is (Java virtuaalmasinas), siis nõutav kulu on M-korda nii palju kui on Java virtuaalmasinaid. Pealegi, kui iga teenus jookseb konkreetses virtuaalmasinas (näiteks EC2 pilveteenuses), nagu on täna olukord ettevõttes Netflixis, siis nõutav kulu oleks veelgi suurem. [9]

1.2 Monoliitarhitektuur

Monoliitrakendused on loomulik viis, kuidas rakendused arenevad. Enamik rakendusi algab ühest konkreetsest eesmärgist või vähestest seotud eesmärkidest. Aja jooksul toimub nendesse rakendustessse erinevate featuuride implementeerimine, tarnimaks ärilisi vajadusi. [12]

Monoliit kirjeldab tarkvara kui *ühe-astmelist rakendust*, kus kõik erinevad komponendid on moodustatud üheks tervikuks. [13]



Joonis 3 Monoliitarhitektuur

Joonisel 3 on kujutatud e-kaubanduse struktuuri kirjeldav monoliitne arhitektuur. See koosneb ühest eraldiseisvast monoliitrakendusest, mis suhtleb temaga seotud relatsioonilise andmebaasiga.

Kui mikroteenused keskenduvad konkreetselt ühele ärikriitilisele ülesandele, siis monoliit seevastu tegeleb kõigega.

1.2.1 Eelised

- **Kommunikatsiooni kulud** – kulud komponentide vahel on pea nullilähedased, kuna kood asub sama rakenduse kõhus. [12]

- **Lihtne arendada** – projekti alguses on monoliitrakendusega palju lihtsam edasi minna. [13]

- **Lihtne testida** – lihtsasti implementeeritavad testid [13]. Kui mikroteenuste puhul peab võrgupäringud ümber konfigureerima, siis monoliitses rakenduses see nii ei pruugi olla.

- **Lihtne paigaldada** – selleks, et terve süsteem saaks toimima, piisab ainult ühe monoliidi kokkuehitamisest ning serverisse paigaldamisest.

1.2.2 Puudused

Kui rakendus on juba piisavalt suureks muutunud ning meeskond suuruselt kasvanud, siis tekib monoliitse arhitektuuri puhul esile väga palju puudusi, mis muutuvad ajaga üha olulisemaks: [14]

- **Hooldatavus** – kui rakendus on saavutanud juba piisavalt massiivse koodibaasi, siis on seda väga keeruline mõista ning kiirete ja korrektsete muudatuste tegemine on üsna väljakutsuv tegevus [13]. Tüüpiline tulemus on see, et arendusprotsess aeglustub. Seda soosib samuti ka see, et modulaarsed piirangud sootuks puuduvad. [14]

- **Ülekoormus** – mida suurem on rakendus, seda kauem võtab aega ka rakenduse töölepanek. See mõjutab samuti arendajate produktiivsust, sest palju aega kulub lihtsalt konteineri käivitamise ootamiseks. [14]

- **Pidevpaigaldus** – suur monoliitne rakendus takistab sagedast paigaldatavust serveritesse. Iga komponendi väiksema muudatuse korral, peab terve süsteemi uuesti üles paigaldama. [14] Samuti on raske samasuguste artifaktide paigalduste kaudu saavutada operatiivset agiilsust [15].

- **Töökindlus** – mistahes mooduli viga (näiteks mälu leke) võib kogu protsessi alandada. Pealegi, kuna kõik instantsid rakenduses on identsed, siis viga võib mõjutada ka tervet rakenduse kättesaadavust. [13]

- **Skaleeritavus** – monoliitse rakenduse skaleeruvuse puuduseks on see, et seda on võimalik skaleerida ainult ühes mõõtmes. Pealegi igal rakenduse komponendil on erinevad ressursinõuded – üks võib olla protsessori intensiivne, samas kui teine võib olla intensiivne just mälu osas. [14]

- **Uute tehnoloogiate kasutuselevõtt** – sõltumata sellest, kui lihtsad on arenduse algstaadiumid, on monoliitses arhitektuuris raske adopteerida uusi tehnoloogiaid, sest muudatused programmeerimiskeeltes või raamistikes mõjutavad tervet rakendust [13]. Monoliitne arhitektuur sunnib sõna otseses mõttes jääma kinni tehnoloogiaga, mis valiti arenduse alguses. [14]

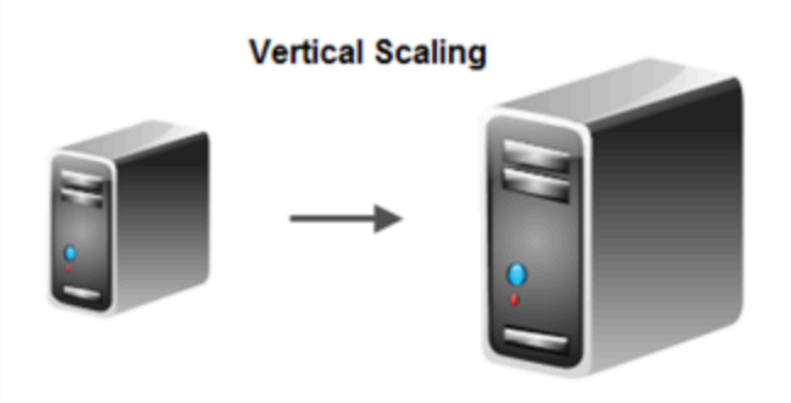
1.3 Skaleeritavus

Mingil ajahetkel tekib igal piisavalt suurel ettevõttel vajadus süsteemide skaleeruvuse järele [16]. Antud peatükis defineerime skaleeritavuse mõiste ning mis vahe on horisontaalsel- ja vertikaalsel skaleerumisel.

Tarkvaraarenduses kasutatav termin *skaleeritavus* tähendab soovitud süsteemi, võrgu või protsessi suurenemist, või omadust mis näitab tema **võimet käitlemaks** suurenevas koguses andmeid. Näiteks süsteemile ressursi juurde lisamisel võib see viidata tema läbilaskevõimele andmete koormuse suurenemise korral. [17]

1.3.1 Vertikaalne skaleeritavus

Vertikaalne skaleeritavus tähendab sisuliselt süsteemi võimsuse suurenemist mingi riistvara komponendi juurde lisamisel. Sellepärast nimetatakse seda ka vertikaalseks, sest töötlusseadmete, mälu või salvestusruumi järjestikusel juurde lisamisel suureneb süsteemi võimsus kuid muutumatuks jääb tema arhitektuur ning infrastruktuur. [18]



Joonis 4 Vertikaalne skaleerumine [18]

Joonisel 4 on kujutatud süsteemi skaleerumist vertikaalselt, mis muudab süsteemi olemuselt suuremaks ning võimsamaks, kuid arhitektuuriline disain jääb samaks.

Vertikaalset skaleerimist kasutatakse kõige sagedamini keskastme rakendustes kui ka väiksete või keskmise suurusega ettevõtetes. [19]

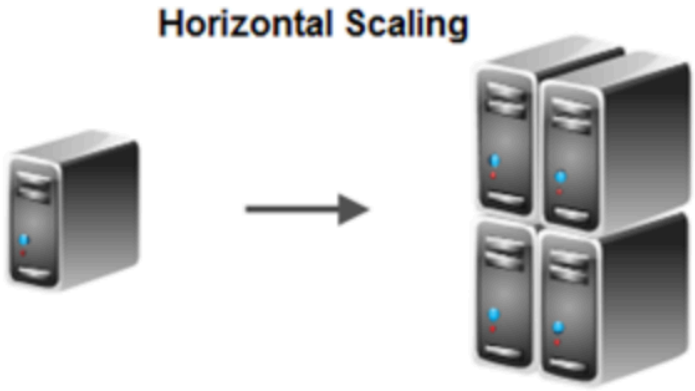
1.3.2 Horisontaalne skaleeritavus

Selle asemel, et riistvara kaudu jõudlust saavutada, toetub horisontaalne skaleeritavus seevastu spetsiaalselt arhitektuuri disainile. Süsteem disainitakse välja selliselt, et rohkemate instantside lisamisel pilve või virtuaalmasinasse skaleerub süsteem ise. [18]

See protsess hõlmab klastris olevate sõlmede arvu suurendamist, vähendades iga sõlme vastutust, pakkudes täiendavaid lõpp-punkte kliendi ühendustele. See on võime suurendada süsteemi võimsust, ühendades mitmeid riist- või tarkvaraüksuseid nii, et nad töötaksid kui ühe loogilise üksusena. [19]

Siin tasub meelde jätte, et iga instants mida kloonitakse, on ka eraldi iseseisev osa, millel on enda käituskeskkond, koodibaas, protsessilõimed jpm. [18]

Joonisel 5 on kujutatud süsteemi skaleerumist horisontaalselt, kus süsteem kui füüsilise objektina ei suurene vaid kasvab tema instantside arv, mis tagab suurema stabiilsuse ning väiksemad kulud.



Joonis 5 Horisontaalne skaleerumine [18]

Horisontaalne skaleerumine on peaaegu alati soovituslikum lähenemine vertikaalsele skaleerumisele, sest siis ei teki muret, et ressurss saaks otsa. Selle asemel, et rakendus lahti ühendada ning ühendus kaotada, võimaldab horisontaalne skaleerumine hoida juba töötavat ressursside kollektsiooni elus, ajal kui lisatakse rohkem ressurssi olemasolevale juurde. [20]

Horisontaalne skaleerimine on äärmiselt vajalik, kui tegemist on teenuste kõrge kättesaadavusega. [19]

Mikroteenuste skaleeruvusel tekivad mõningad lisakeerukused. Selle asemel, et mõelda ainult ühest jooksvast rakendusest ühes serveris. Peab mikroteenuste puhul mõtlema ka tehnoloogiavabaduse peale (rakendused võivad olla kirjutatud kasutades erinevaid tehnoloogiaid). Samuti tuleb mõelda ka erinevatel riistvaradel- või eriasukohas paiknevatele pilveteenustele paigaldatud rakendustest. Sest kui rakenduse nõudlus suureneb, siis kõik alus-komponendid peavad koordineeritult skaleeruma või vähemalt suutma tuvastada millised individuaalsed osad peavad nõudluse kasvule suurenema. [21]

1.4 Arhitektuuri mudeli ja lähtekoodi vaheline seos

On oluline mõista seost arhitektuuri mudeli ning lähtekoodi vahel.

Lähtekood on nii lõpptulem, kui ka vahend kuidas erinevaid ärilisi vajadusi tarnida. Arhitektuuri mudel see-vastu ei ole lõpptulem ega ka tarnitav tükk vaid on kasulik ainult juhul, kui seda saab siduda lähtekoodiga. [22]

Selleks, et teha vahet arhitektuuri mudeli ja koodi vahel, on kasulik koostada nimekiri, mida üks neist täpsemalt sisaldab. Tabelis 1 on välja toodud enim kasutatavaid elemente, mida võib kohata arhitektuuri mudelites ning lähtekoodides. [22]

Tabel 1 Arhitektuuri mudelites ning lähtekoodis kasutatavad terminid [22]

Asukoht	**Elemendid**
Arhitektuuri mudel	Moodulid, komponendid, ühendused, pordid, stiilid, vastutuse jaotused, disaini otsused, protokollid
Lähtekood	Paketid, klassid, meetodid, muutujad, funktsioonid, protseduurid, deklaratsioonid

Sõnavara. Nende terminite võrdlus arhitektuuri mudelis ja lähtekoodis viitab sellele, et mõlemad kasutavad erinevat sõnavara kirjeldamaks tüüpiliselt samu asju. Näiteks arhitektuuri mudel koosneb *moodulitest* kuid seevastu lähtekood koosneb *pakettidest (packages)*, mis on nime poolest erineva tähendusega kuid sisuliselt on need samad asjad. [22]

Abstraktsioon. Arhitektuuri mudelid kipuvad olema rohkem abstraktsemad kui lähtekood kahel viisil. Esiteks, üksikelement arhitektuuri mudelis sageli seob kokku mitut elementi lähtekoodis. Näiteks komponent arhitektuuri mudelis võib kokku võtta mitut objekti lähtekoodis. Teiseks, kui mõlemad kirjeldavad sama elementi, siis arhitektuuri mudelid üldiselt kirjeldavad seda elementi vähem detailsemalt kui teeb seda lähtekood. Arhitektuuri mudel võib peatuda selle elemendi kirjelduse osas nii pea, kui ta jõuab moodulite ja komponentideni. Lähtekood seevastu kirjeldab antud elementi detailsemat läbi klasside, meetodite ning muutujate kaudu. [22]

1.4.1 Mudel-koodi mittevastavus

Arhitektuuri mudelid ning lähtekood ei kirjelda samu asju üheselt. Erinevus nende kahe vahel on mudel-koodi mittevastavus. [22]

Arhitektuuri mudelid sisaldavad abstraktseid kontseptsioone, näiteks komponente, aga programmeerimiskeeled selliseid kontseptsiooni ei tunne. Veel peale selle sisaldavad

mudelid ka intensiivseid elemente, näiteks disaini otsuseid ja piiranguid, mida pole võimalik lähtekoodis kuidagi kajastada. [22]

Järelikult arhitektuuri mudeli ja lähtekoodi vaheline suhe on keeruline. Mudelid aitavad välja selgitada süsteemi keerukust ning skaleeruvust, sest need on abstraktsed ja intensiivsed. Lähtekood seevastu käivitub masinatel, sest see on konkreetne ning ekstensiivne. [22]. Sellest tulenevalt tekib probleem – ***arhitektuuri diagrammid ei sobi kokku kirjutatava lähtekoodiga***. [7]

George Fairbanks väidab enda raamatus, et üks viis kuidas saab mudel-koodi lõhet addresseerida, on läbi arhitektuuriliselt-ilmse koodistiili [22]. See tähendab, et koodi struktuur peab peegeldama arhitektuurset kavatsust [7].

2 Modulaarsus tarkvaraarenduses

2.1 Modulaarsuse põhimõtted

Mikroteenuste arhitektuur on täna jõudsalt hoogu kogumas, kuid kõige tähtsam pikas perspektiivis on modulaarsus. [23]

Tarkvaraarenduses viitab **modulaarsus** rakenduse tükeldamist väikesteks mooduliteks, mis käsitlevad erinevat äriloogikat. [24]

Modulaarsust saab jagada kolmeks juhtpõhimõtteks: [25]

- **Kapseldamine** – peidab implementatsiooni detailid komponentide sisse, mis tagab nõrga sidestumise *(low coupling)* erinevate osade vahel. [25] Nõrk sidestumine tähendab sisuliselt seda, et muudatus ühes moodulis ei tohi kajastuda teistes moodulites.

- **Hästi-määratletud liidesed** – kõike pole võimalik peita (muidu süsteem ei tee midagi olulist), seega hästi-määratletud ning stabiilsed liidestused komponentide vahel on möödapääsmatu. [25]

- **Seosed** [25] – komponentide vahelised seosed peavad olema hästi piiritletud ning ei tohi sisaldada ringseoseid.

Moodul on tarkvarakomponent või rakenduse osa, mis sisaldab ühte või mitut rutiini. Rakendus võib sisaldada mitut erinevat moodulit, kus igal moodulil on unikaalne ning eraldiseisev äriloogika. [26]

Moodulite sisu võib jagada järgnevalt: [27]

- **Privaatne** – privaatset sisu saab muuta ilma teiste moodulite mõjutamiseta. Privaatse sisu muudatused ei mõjuta teisi mooduleid. [27]

- **Eksporditud** – eksporditud sisu jagatakse teiste moodulitega. Muutused nendes moodulis mõjutavad teisi mooduleid. Seetõttu on tal koormus areneda kontrollitult, sest eksporditud moodul ei tea temast sõltuvate moodulite konteksti. [27]

- **Imporditud** – imporditud sisu pärineb teisest moodulist. Imporditud moodul on selgesti sõltuv imporditud sisust. [27]

Moodulid teevad programmeerija töö lihtsamaks ning arusaadavamaks, sest see võimaldab neil keskenduda ainult ühele konkreetsele funktsionaalsele äriloogikale. [26]

2.2 Koodi struktureerimise stiilid

Selleks, et tarkvara säiliks ning toimiks, tuleb rakendada teatud reeglid. Üks neist on koodi struktureerimine. See tähendab sisuliselt lähtekoodi organiseerimist kaustade vahel (Java mõistes pakettide vahel), mis annab arusaadavama pildi olemasolevast loogikast ning seab teatud modulaarsuse piirangud.

2.2.1 Paketid kihtide tasemel

Esimene ning võib-olla kõige lihtsam disainimuster on traditsiooniline horisontaalne kihiline arhitektuur, kus lähtekood eraldatakse tehnilisest perspektiivist. Sellist lähenemist kutsutakse sageli "paketid kihtide tasemel" ingl. keeles "package by layer". [33]

Tüüpiliselt selline kihiline arhitektuur koosneb kolmest kihist - liidestus, äriloogika ning andmebaasi loogika. Sellisele mustrile viitavad paljud raamatud, õpetused, koolituskursused, näidiskoodid jne. See on kiire viis midagi implementeerida ning jooksutada ilma suurema keerukuseta. [33]

2.2.2 Paketid featuuride tasemel

Teine viis kuidas koodi organiseerida on rakendada disainimustrit "paketid featuuride tasemel" ingl. keeles "package by feature". Erinevalt horisontaalsele tükeldamisele, tükeldatakse nüüd vertikaalselt. Tüüpilises implementatsioonis kõik lähtekood pannakse ühte Java paketti (või kausta), mille nimi peegeldab featuuri kontseptsiooni. [33]

2.2.3 Paketid komponentide tasemel

Kolmas viis kuidas koodi organiseerida on stiilis "paketid komponentide tasemel" ingl. keeles "package by component".

See on hübriidne lähenemine eelnevatele stiilidel, kus kood pannakse kokku üheks jämedakujuliseks komponendist, kus kihiline eraldatus on säilitatud kuid eelnevalt

teistele lähenemistele on komponendi implementatsioon täielikult tarbija jaoks peidetud. Tegemist on tarkvarasüsteemi teenusekeskse vaatega, mis on umbes sama mida pakub ka mikroteenusarhitektuur. [33]

2.3 Vahe koodi organiseerimisel ja kapseldamisel

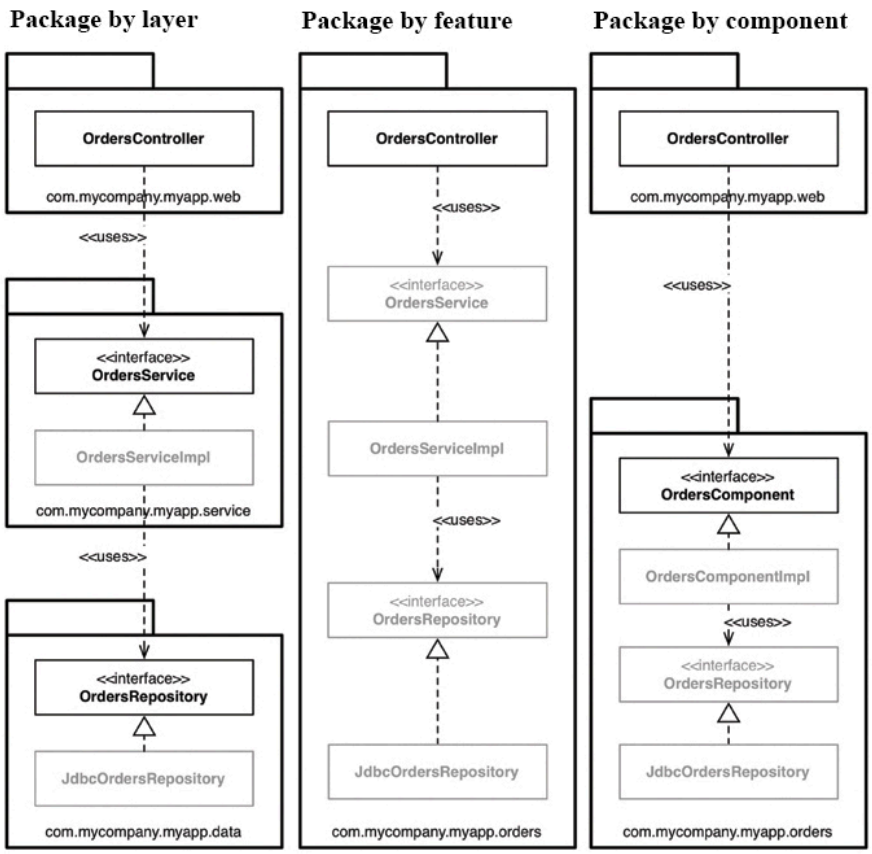
On oluline teha vahet koodi organiseerimisel ja kapseldamisel.

Kui teha kõik Java rakenduse klassitüübid avalikuks, siis klassid muutuvad sisuliselt organiseerimise mehhanismiks. Kuna *public* tüüpe saab kasutada kus iganes koodibaasis, siis võib neid pakette sisuliselt ignoreerida, sest need ei too mingit reaalset väärtust. [33]

Teistpidi, kui kasutada *public* tüüpe ainult kohtades kus neid tõesti vaja läheb, siis on võimalik saavutada teatud sorti modulaarsed piirangud.

Java juurdepääsu modifikaatorid ei ole perfektsed, kuid neid ignoreerides me lihtsalt küsime probleeme juurde. [33]

Joonis 6 kirjeldab eelnevalt nimetatud disainimustreid, kus halliga kaetud osasid saab kitsendada Java modifikaatorite abil.



Joonis 6 Koodi struktureerimise stiilid kitsendatud ligipääsudega [33]

Jooniselt on näha, et antud stiilid on struktuuriliselt küll erinevad ent süntaktiliselt peaaegu identsed. [33]

Tüüpiliselt on koodi struktureerimise stiilid mõeldud eraldamaks teatud sorti ärilist tegevust. Näiteks veebiloogika peab olema eraldatud äriloogikast, äriloogika oma korda eraldatud andmebaasi loogikast jne. See tähendab samuti ka seda, et veebiloogika ei tohi otseselt suhelda andmebaasiga.

Probleem seisneb selles, et kui uuele arendajale selliseid reegleid ei tutvustata, siis võib tekkida olukord, kus kood koosneb paljudest absurdsetest seostest ning mis võib halvimal juhul lõppeda suure palli mudaga (ingl. keeles big ball of mud).

Liikudes jooniselt 7 vasakult-paremale, siis “package by layer” lähenemisel peavad nii OrdersService kui ka OrdersRepository liidesed olema avalikud, sest nendevaheline sõltuvus on eraldi pakettides. Probleem seisneb selles, et OrdersControllerist saab välja kutsuda loogikat, mis paikneb OrdersRepository klassis. [33]

“Package by feature” lähenemisel on OrdersController ainus sisenemispunkt selles paketis ning kõikide teiste ligipääsud saame kitsendada. Ent see ei lahenda samuti esialgset probleemi, sest kuna kõik klassid asuvad nüüd ühes paketis, siis on võimalik OrdersControllerist välja kutsuda OrdersRepository loogikat. [33]

Viimases mustris “package by component” peavad nii OrdersController kui ka OrdersComponent olema avalikud - ülejäänud klassid saame kitsendada. See tagab nüüd olukorra, kus OrdersController saab välja kutsuda ainult OrdersComponent loogikat.

Tulemuseks on see, et kui ignoreerida pakette ning kasutada *public* terminit kohtades kus seda tegelikult pole vaja, siis ei ole väga oluline millist arhitektuuri stiili või mustrit me üritame luua. [33]

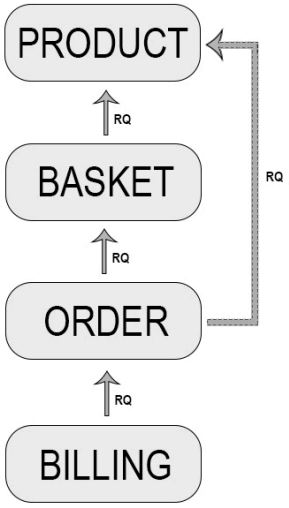
3 Näidisrakendus

3.1 Arhitektuuri mudeli koostamine

Antud töös lähtume lihtsustatud toote ostmise peastsenaariumist ning selle üldisemad kasutuslood on järgnevad:

- Toodete kuvamine – Süsteem kuvab kliendile tooteid;
- Toote lisamine ostukorvi – Klient valib toote ning lisab ostukorvi;
- Ostukorvi kinnitamine – Klient kinnitab ostukorvi;
- Tellimuse esitamine – Süsteem esitab kliendile tellimuse;
- Tellimuse kinnitamine – Klient kinnitab tellimuse (sooritab ostu);
- Tehingu valideerimine – Süsteem valideerib ostutehingu.

Nendest punktidest tulenevalt võime joonistada neli eraldiseisvat moodulit ning märkida nendevahelised sõltuvused joonisel 7.



Joonis 7 Näidisrakenduse moodulite seosed

Joonist 7 võime käsitleda kui neljaks alamosaks jagatud tervikrakendust, kus iga alamosa sisaldab eraldiseisvat äriloogikat. Samuti on veel näha, et nendevahelised ringseosed puuduvad.

Alamprojektide lühikirjeldus:

- **Billing** – sisaldab arveldusega seotud loogikat (arvete koostamine tellimuste põhjal, maksete protsessimine jms);
- **Order** – sisaldab tellimuste protsessimise loogikat (ostukorvi sisu põhjal tellimuste koostamine, tellimuste uuendamine jne);
- **Basket** – ostukorviga seotud loogika (toodete lisamine ostukorvi, ostukorvi kinnitamine, ostukorvi uuendamine);
- **Product** – sisaldab toodetega seonduvat loogikat.

Tabelis 2 on välja toodud projektide vahelised sõltuvused ning milliseid transitiivseid sõltuvusi nad koodimuudatuste kaudu mõjutavad.

Tabel 2 Moodulite sõltuvused

Mooduli nimi	Sõltuv moodulitest	Muudatused mõjutavad mooduleid
Billing	Order	-
Order	Basket, Product	Billing
Basket	Product	Order
Product	-	Basket, Order

3.2 Implementatsioon

Käesolev rakendus ehitatakse Spring Boot raamistiku peale ning alamosade sõltuvuste juhtimiseks kasutatakse tööriista Gradle.

Spring Boot on osa Spring raamistikust, [28] mis võimaldab luua iseseisvaid rakendusi mida saab "lihtsalt jooksutada" kõige väiksema vaevaga. Enamus Spring Boot rakendusi vajavad koheseks käivitamiseks minimaalset konfigureerimist. [29]

Gradle on avatud lähtekoodiga tarkvara ehitamist automatiseeriv tööriist *(build automation tool)*, mis teeb lihtsaks projekti ühiste osade kokkuehitamist, kasutades selleks eeltöödeldud funktsionaalsust defineeritud pluginite kaudu. See tähendab, et tarkvara kokkuehitamine sisuliselt konfigureerib defineeritud ülesannete kogumi ning seejärel võtab need kokku. [30] Tööülesanne võib olla mistahes rakendusega seotud automatiseeritud tegevus.

Gradle valideerib ning teostab skripte kolmes etapis: [30]

- Initsialiseerimine – seadistab sobiva keskkonna ning määrab millised projektid sellest osa võtavad. [30]
- Konfiguratsioon – konfigureerib ülesannete graafi ning määrab missugused ülesanded jooksevad millises järjekorras, lähtudes sellest mida kasutaja soovib käivitada. [30]
- Käivitamine – käivitab konfiguratsioonifaasi lõpus kasutaja soovitud ülesande [30]

3.2.1 Rakenduse modulaarsuse saavutamine

Selleks, et rakendust saaks mooduliteks jagada, tuleb defineerida **juur- ning alamkaustades** failid nimega *build.gradle*. See fail ütleb millised sõltuvused, pluginad ning muud tööülesanded antud projekti kuuluvad.

```
import org.springframework.boot.gradle.plugin.SpringBootPlugin

plugins {
  id 'org.springframework.boot' version '2.1.4.RELEASE' apply false
}

allprojects {
  apply plugin: 'java'
  apply plugin: 'io.spring.dependency-management'

  repositories {
    jcenter()
  }

  dependencyManagement {
    imports {
      mavenBom (SpringBootPlugin.BOM_COORDINATES)
    }
  }
}
```

Koodist võib lugeda, et on defineeritud ka Spring Boot plugin lausega *apply false*. Selle mõte on defineerida globaalne Spring versioon, mis võimaldab uusi Springi sõltuvusi kasutsele võtta ilma versiooni märkimata.

Selleks, et määrata millised moodulid projekti kuuluvad, tuleb defineerida juurkausta fail settings.gradle:

```
rootProject.name = 'modular-monolith'
rootDir.eachDirMatch ~/^mod-.*/, {
  include it.name
}
```

Moodulid on antud rakenduse mõistes tavalised kaustad eesliitega "**mod-**", mis sisaldavad lähtekoodi.

Veel peale juurkausta konfiguratsioonifailide on defineeritud ka fail gradle/common.gradle. Selle faili mõte on jagada ühist projektide vahelist konfiguratsiooni ning kolmanda osapoole sõltuvusi.

```
dependencies {
  compileOnly 'org.projectlombok:lombok:1.18.6'
  annotationProcessor 'org.projectlombok:lombok:1.18.6'
  implementation 'org.springframework:spring-context'
}
```

Nende sõltuvuste defineerimist oleks saanud teha ka juurkausta build.gradle failis, kuid kuna projektis on ka alamprojekte, mis neid sõltuvusi ei vaja, siis oleks sinna määramine olnud liigne tegevus.

- **compileOnly** – märgib sõltuvusi, mis on vajalikud ainult programmikoodi kompileerimiseks kuid mitte kunagi programmi käitusajaks [31]
- **annotationProcessor** – sisuliselt sama *compileOnly* skoobiga, kuid lähtutakse koodis olevatest annotatsioonidest
- **implementation** – märgib kompileerimiseks vajalikku sõltuvusi, kuid ei lekita transitiivseid sõltuvusi tarbija enda klasside kompileerimise asukohta [32]

Järgmise sammuna kirjeldame lahti alamprojektide konfiguratsioonifailid:

- **mod-boot/build.gradle**

Antud projekti võib mõelda kui rakenduse mootorit, millele teised moodulid toetuvad. Sisaldab endas Spring Boot pluginat, mis võimaldab rakendust käsurealt käivitada: *./gradlew bootRun*.

```
apply plugin: 'org.springframework.boot'

dependencies {
  compile 'org.springframework.boot:spring-boot-starter'
  runtime rootProject.childProjects.values().findAll {
    it.plugins.hasPlugin('java')
  }
}
```

- **mod-web/build.gradle**

Selleks, et rakenduse tarbijad antud äriloogikale ligi pääseksid on vaja head programmiliidestust. Antud mooduli mõte on eraldada liidestus äriloogikast.

Selle tulemusena eraldame veebiloogika äriloogikast ning hoiame juba varakult ära ebavajalikud kihtide vahelised seosed.

```
apply from: "$rootDir/gradle/common.gradle"

dependencies {
    implementation(
            'org.springframework.boot:spring-boot-starter-web',
            'org.springframework.boot:spring-boot-starter-
hateoas',
            project(':mod-basket'),
            project(':mod-product'),
            project(':mod-order'),
            project(':mod-billing')
    )
}
```

Viimase sammuna peegeldame joonisel 7 koostatud sõltuvuste graafi lähtekoodiga:

- **mod-basket/build.gradle**

```
apply from: "$rootDir/gradle/common.gradle"

dependencies {
    implementation(project(':mod-product'))
}
```

- **mod-billing/build.gradle**

```
apply from: "$rootDir/gradle/common.gradle"

dependencies {
    implementation(
```

```
            project(':mod-order')
    )
}
```

- **mod-product/build.gradle**

```
apply from: "$rootDir/gradle/common.gradle"
```

- **mod-order/build.gradle**

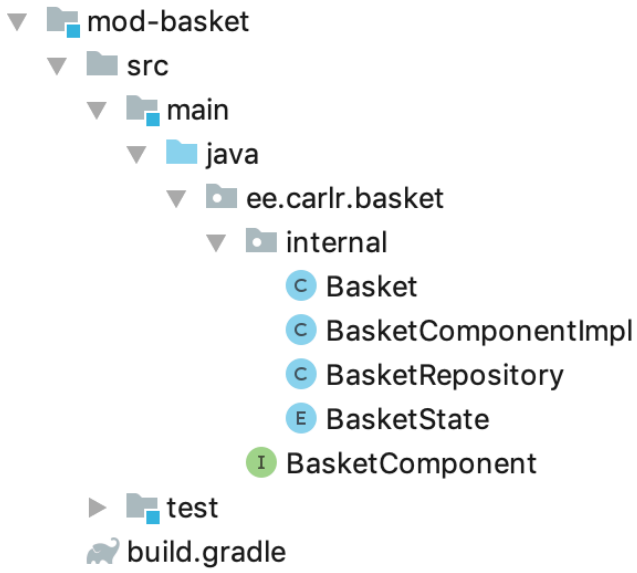
```
apply from: "$rootDir/gradle/common.gradle"

dependencies {
    implementation(
            project(':mod-basket'),
            project(':mod-product')
    )
}
```

3.2.2 Äriloogika implementeerimine

Arendatavas rakenduses kasutame "paketid komponentide tasemel" koodi struktureerimist, et saavutada modulaarsus ka moodulite enda seisukohast.

Tulles tagasi projekti struktuuri juurde, siis igale moodulile on defineeritud isiklik lähtepuu *(source tree)* ning talle kuuluv konfiguratsioonifail. Lähtepuu on sisuliselt kaustade kollektsioon, mis organiseerib kaustade ning failide asukohad moodulis.

Joonisel 8 on välja toodud ostukorvi mooduli lähtepuu, mis koosneb talle kuuluvast konfiguratsioonifailist *build.gradle* ning lähtekoodist paketis *ee.carlr.basket*.



Joonis 8 Ostukorvi lähtepuu

Antud jooniselt on näha, et paketis *ee.carlr.basket* on defineeritud ainult BasketComponent liides ning kõik temaga seotud implementatsioon on paketis *ee.carlr.basket.internal*. Oleks olnud võimalus ka kõik failid ühte paketti defineerida ning modulaarsus oleks endiselt säilinud, kuid loetavuse huvides hoiab autor implementatsiooni eraldi paketis.

Antud moodulis on avalikuks tehtud liides BasketComponent (tarbijatel on vaja äriloogikale kuidagi ligi pääseda) ning klass Basket (tellimusi vormistatakse ostukorvi põhjal order moodulis). Ülejäänud klassid saame kapseldada ning tarbijad ei pea nendest midagi teadma. Sama loogika kehtib ka teiste moodulite kohta.

Moodulite detailne kirjeldus ning koodinäited:

1) **mod-basket** - ostukorviga seotud äriloogika: ostukorvi loomine, pärimine, kinnitamine ning toote lisamine

   Koosneb ühest komponendist:

```
public interface BasketComponent {
  Basket createBasket(Object customerDetails);

  Basket getBasket(Long basketId);

  void addProduct(Long basketId, Long productId);

  void confirmBasket(Basket basket);
}
```

2) **mod-billing** – arveldamisega seotud äriloogika: maksete protsessimine ning arve koostamine ja pärimine arve- või tellimuse numbri alusel

   Koosneb kahest komponendist:

```
public interface InvoiceComponent {
  Invoice getInvoice(Long invoiceId);

  Invoice getInvoiceByOrderId(Long orderId);

  void updateInvoice(Invoice invoice);
}

public interface PaymentComponent {
  void processPayment(PaymentResponse paymentResponse);
}
```

3) **mod-order** – tellimustega seotud äriloogika: tellimuste koostamine, pärimine ning uuendamine

Koosneb ühest komponendist:

```
public interface OrderComponent {
  Order createOrder(Basket basket);

  Order getOrder(Long orderId);

  void updateOrder(Order order);
}
```

4) **mod-product** – toodetega seotud äriloogika: kõikide toodete pärimine ning toote pärimine tootenumbri alusel

   Koosneb ühest komponendist:

```
public interface ProductComponent {
  List<Product> getAllProducts();

  Product getProduct(Long productId);
}
```

5) **mod-boot** – peamoodul, mis koosneb Spring Boot'i Application klass main meetodist. Kõik teised olulised moodulid, mis rakenduses esinevad on selle mooduli runtime või compile skoobiga sõltuvuses.

```
@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

6) **mod-web** – rakenduse liidestus, mille kaudu tarbijad saavad äriloogikale ligi:

   - */basket*

```
@GetMapping(value = "/{basketId}")
ResponseEntity<Basket> getBasket(@PathVariable Long basketId)

@PostMapping
ResponseEntity<Void> createBasket(@RequestBody Object customerDetails)

@PostMapping(value = "/confirm")
ResponseEntity<Void> confirmBasket(@RequestParam Long basketId)

@PutMapping(value = "/add")
ResponseEntity<Void> addProduct(@RequestParam Long basketId,
                                @RequestParam Long productId)
```

- */product*

  ```
  @GetMapping(value = "/getAll")
  ResponseEntity<List<Product>> getAllProducts()
  ```

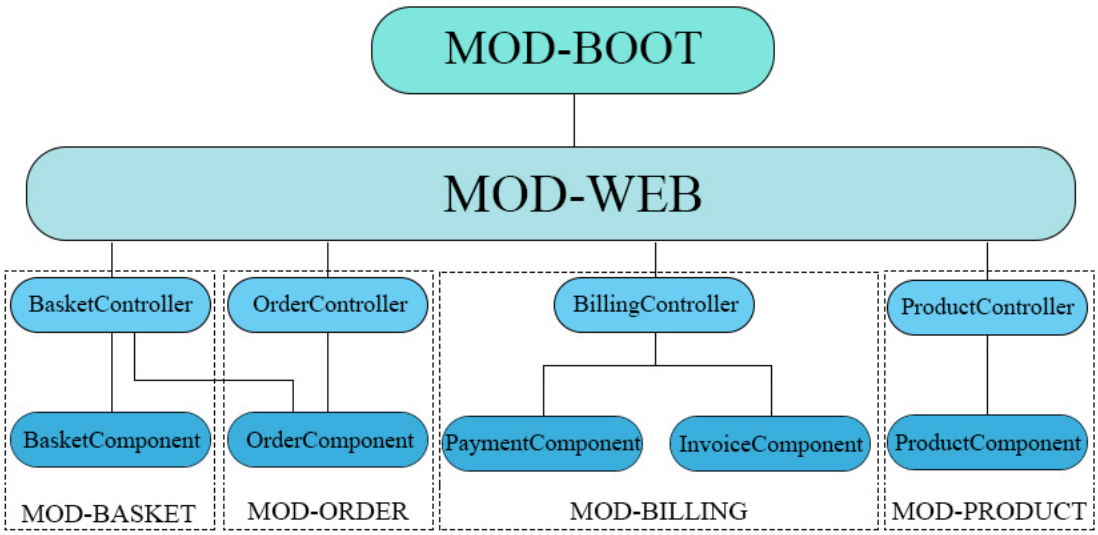
- */order*

  ```
  @GetMapping(value = "/{orderId}")
  ResponseEntity<Order> getOrder(@PathVariable Long orderId)
  ```

- */billing*

  ```
  @PostMapping(value = "/processPayment")
  ResponseEntity<Void> processPayment(@RequestBody PaymentResponse
  paymentResponse)

  @GetMapping(value = "/invoice")
  ResponseEntity<Invoice> getInvoice(@RequestParam Long orderId)
  ```

Lõplik rakenduse struktuur on esitatud joonisel 9.



Joonis 9 Rakenduse struktuur

Kokkuvõte

Antud töö eesmärk oli võrrelda monoliitarhitektuuri mikroteenus-arhitektuuriga ja selgitada kuidas mikroteenuste parimaid külgi juurutada monoliitsüsteemi.

Töö alguses keskenduti mõlema arhitektuuri teoreetilise poolele. Mikroteenuste üheks suurimaks eeliseks monoliitsüsteemi suhtes on modulaarsus. Vastupidiselt monoliidile, kus tarkvara on kirjeldatud kui ühe-astmelist rakedust, toetub mikroteenus-arhitektuur seevastu teenuste komplektidena, kus igal teenusel on kindel eesmärk. Modulaarsust tarkvaraarenduses võib jagada kolmeks juhtpõhimõtteks: tugev kapseldamine, hästi-määratletud liidesed ja piiritletud seosed. Selleks, et tarkvara säiliks ja toimiks tuleb rakendada koodi struktureerimise reegleid.

Neid juhtpõhimõtteid ära kasutades, koostati lihtsakoeline arhitektuuri mudel, mis pandi vastavusse Java lähtekoodiga. Antud rakenduses kasutati koodi struktureerimiseks "paketid komponentide tasemel" lähenemist, sest see tagas parema kapseldamise klasside vahel. Valminud rakendus on sisu-tühi ning vajalik ainult selleks, et kirjeldada ühe modulaarse monoliidi koostamist.

Autor on seisukohal, et projekti arhitektuuri valikul ei tohiks lähtuda selle populaarsusest. Mikroteenused on samm edasi modulaarse monoliidile ning kui ei osata ehitada iseseisvaid mooduleid, siis ei aita kaasa ka mikroteenused. Hästi-määratletud monoliit on eelduseks heale mikroteenus-arhitektuurile, sest moodulit on võimalik vajadusel väikse vaevaga teha mikroteenuseks.

Kasutatud kirjandus

[1] Codingthearchitecture.com. (2019). *Modular monoliths - Coding the Architecture*. [online] Available at: http://www.codingthearchitecture.com/presentations/sa2015-modular-monoliths [Accessed 21 Apr. 2019].

[2] Oracle.com. (2019). *Java Software | Oracle*. [online] Available at: https://www.oracle.com/java/ [Accessed 20 May 2019].

[3] Features, G., Plugins, C., Kotlin, G., Gradle, M., Resources, M. and Enterprise, G. (2019). *Gradle Build Tool*. [online] Gradle. Available at: https://gradle.org/ [Accessed 20 May 2019].

[4] Smartbear.com. (2019). *What are Microservices? | API Basics | SmartBear*. [online] Available at: https://smartbear.com/solutions/microservices/ [Accessed 21 Apr. 2019].

[5] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc., 2015.

[6] Docs.microsoft.com. (2019). *Communication in a microservice architecture*. [online] Available at: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture [Accessed 20 May 2019].

[7] Architecture, C. (2019). *An architecturally-evident coding style - Coding the Architecture*. [online] Codingthearchitecture.com. Available at: http://www.codingthearchitecture.com/2014/06/01/an_architecturally_evident_coding_style.html [Accessed 20 May 2019].

[8] Amazon (2019). *Benefits of Microservices - Microservices on AWS*. [online] Available at: https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/benefits-of-microservices.html [Accessed 21 Apr. 2019].

[9] microservices.io. (2019). *Microservices Pattern: Microservice Architecture pattern*. [online] Available at: https://microservices.io/patterns/microservices.html [Accessed 21 Apr. 2019].

[10] Microservices Practitioner Articles. (2019). *Monolithic vs. Microservices Architecture*. [online] Available at: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59 [Accessed 21 Apr. 2019].

[11] Bolboaca, A. (2019). *Modular Monolith Or Microservices? | Mozaic Works*. [online] Mozaic Works. Available at: https://mozaicworks.com/blog/modular-monolith-microservices/ [Accessed 21 Apr. 2019].

[12] Nortal. (2019). *Are monolithic software applications doomed for extinction? - Nortal*. [online] Available at: https://nortal.com/blog/are-monolithic-software-applications-doomed-for-extinction/ [Accessed 20 May 2019].

[13] Medium. (2019). *Introduction to Monolithic Architecture and MicroServices Architecture*. [online] Available at: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63 [Accessed 20 May 2019].

[14] microservices.io. (2019). *Microservices Pattern: Monolithic Architecture pattern*. [online] Available at: https://microservices.io/patterns/monolithic.html [Accessed 20 May 2019].

[15] MuleSoft. (2019). *Microservices vs Monolithic Architecture*. [online] Available at: https://www.mulesoft.com/resources/api/microservices-vs-monolithic [Accessed 20 May 2019].

[16] TechBeacon. (2019). *The challenges of scaling microservices*. [online] Available at: https://techbeacon.com/app-dev-testing/challenges-scaling-microservices [Accessed 21 Apr. 2019].

[17] Gerardnico.com. (2019). *Software Design - Scalability (Scale Up|Out) [Gerardnico]*. [online] Available at: https://gerardnico.com/code/design/scalability [Accessed 21 Apr. 2019].

[18] Codit. (2019). *Scalability within micro-services architecture | Codit*. [online] Available at: https://www.codit.eu/blog/micro-services-architecture/ [Accessed 21 Apr. 2019].

[19] Esds.co.in. (2019). *What is Vertical Scaling & Horizontal Scaling?*. [online] Available at: https://www.esds.co.in/blog/vertical-scaling-horizontal-scaling/#sthash.mqwjdOId.dpbs [Accessed 20 May 2019].

[20] Engineer, C. (2019). *Horizontal Vs. Vertical Scaling: Which Is Right For Your App?*. [online] Mission. Available at: https://www.missioncloud.com/blog/horizontal-vs-vertical-scaling-which-is-right-for-your-app/ [Accessed 21 Apr. 2019].

[21] TechBeacon. (2019). *The challenges of scaling microservices*. [online] Available at: https://techbeacon.com/app-dev-testing/challenges-scaling-microservices [Accessed 21 Apr. 2019].

[22] George H. Fairbanks, *"Just Enough Software Architecture: A Risk-Driven Approach"*, 2010.

[23] Architecture, C. (2019). *Software architecture vs code - Coding the Architecture*. [online] Codingthearchitecture.com. Available at: http://www.codingthearchitecture.com/2014/05/29/software_architecture_vs_code.html [Accessed 20 May 2019].

[24] Techopedia.com. (2019). *What is Modularity? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/24772/modularity [Accessed 20 May 2019].

[25] Mak, S. (2019). *Modules vs. microservices*. [online] O'Reilly Media. Available at: https://www.oreilly.com/ideas/modules-vs-microservices [Accessed 20 May 2019].

[26] Techopedia.com. (2019). *What is a Module? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/3843/module [Accessed 20 May 2019].

[27] TheServerSide.com. (2019). *Successful modularity depends on your dependency model*. [online] Available at: https://www.theserverside.com/feature/Successful-modularity-depends-on-your-dependency-model [Accessed 20 May 2019].

[28] Docs.spring.io. (2019). *Spring Framework Overview*. [online] Available at: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html [Accessed 20 May 2019].

[29] Spring.io. (2019). *Spring Projects*. [online] Available at: https://spring.io/projects/spring-boot [Accessed 20 May 2019].

[30] Docs.gradle.org. (2019). *What is Gradle?*. [online] Available at: https://docs.gradle.org/current/userguide/what_is_gradle.html [Accessed 20 May 2019].

[31] Home, G., Forums, C., Plugins, C., Enterprise, G., Posts, A. and Features, N. (2019). *Introducing Compile-Only Dependencies*. [online] Blog.gradle.org. Available at: https://blog.gradle.org/introducing-compile-only-dependencies [Accessed 20 May 2019].

[32] Docs.gradle.org. (2019). *The Java Library Plugin*. [online] Available at: https://docs.gradle.org/current/userguide/java_library_plugin.html#sec:java_library_separation [Accessed 20 May 2019].

[33] Martin, R. C., & Brown, S., *Clean Architecture: A Craftsman's Guide to Software*, 2017.