TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Margus Hanni 124473IABMM

KOMPONENDIPÕHINE TARKVARAARENDUSRAAMISTIK KUI AVALIKU SEKTORI TARKVARAPLATVORM

Magistritöö

Juhendaja: Ants Torim
PhD

Tallinn 2017

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: Margus Hanni

06.05.17

Lõputöö ülesandeleht

Käesoleva magistritöö eesmärgiks on tuua välja põhjuseid, miks on mõistlik avalikus sektoris keskenduda tarkvaraarenduses tagaliidese ja nn standardsetele päring-vastus tehnoloogiatele ning komponendipõhisele lähenemisviisile.

Avalikus sektoris on leidnud rakendust SOA arhitektuuril ning REST edastusandmekihil baseeruvad tehnoloogiad ning lahendused. Arvestades menetlussüsteemi vajadusi ning ärilist keerukust on raske näha otsest vajadust miks tuleks siin keskenduda teenusepõhistele eesliidese tehnoloogiatele. Antud valiku tulemusena tekib juurde arendatavale tarkvaralahendusele täiendavaid tarkvarakihte, tõuseb keerukus funktsionaalsuses, valideerimisreeglite realiseerimises, jne.

Magistritöös kirjeldab ja analüüsib autor erinevad põhjuseid, miks kaaluda just avaliku sektori menetlussüsteemide arendamisel tagaliidese tehnoloogiaid. Tuues välja valitud tehnoloogiliste suundade plussid ja miinused, leides samas vastused järgmistele küsimustele:

1. Mis probleemid esinevad analüüsifaasis erinevate tehnoloogiate kogumite vaates ning missuguste täiendavate nüanssidega tuleb arvestada tarkvaraarenduses?

2. Mis on need tehnoloogilised valikud, mida tuleks avaliku sektori menetlussüsteemide loomisel kaaluda ning miks mõjutab tehnoloogiate valik tarkvaraarenduse efektiivsust ning loodava tarkvara jätkusuutlikust?

Magistritöö autor on ühe projekti raames välja töötanud tugiraamistiku komponendipõhise kontseptsiooni ning muutnud selle avalikult kätte saadavaks. Autori seisukohalt vastab arendatud tugiraamistik paremini avaliku sektori menetlussüsteemi arendus-vajadustele ning raamistik on töö üheks teesi osaks. Loodud raamistiku ning ees- ja tagaliidese tehnoloogiatega kaasneva erisuse paremaks mõistmiseks kirjeldab töö autor lahti kihilise ja mikroteenuste arhitektuuri mustrid ning analüüsib erinevaid tehnoloogilisi trende.

Annotatsioon

Käesoleva töö eesmärgiks on analüüsida tehnoloogilist suunda, mis on võetud tarkvaraarendus ettevõttes Nortal avaliku sektori tarkvaraliste lahenduste loomisel, mõistmaks kas valitud tehnoloogiline platvorm on õige menetlussüsteemide arendamiseks.

Autor võrdleb ees- ja tagaliidese tehnoloogiad läbi tarkvaraliste lahenduste ning mikroteenuste ja kihilise arhitektuuriliste mustrite. Analüüsis on kasutatud ThoughtWorks ja Nortali tarkvaraarenduse arenduskeelte ja raamistike radareid, töökuulutuste lehe Indeed statistikat ja Nortali sisemise uuringu "SOA, Angular ja modulaarse arhitektuuri" tulemusi.

Autori arvates ei ole Nortali poolt valitud tehnoloogiline suund sobiv avaliku sektori menetlussüsteemi arendamiseks. Eesliidese tehnoloogiate suuna asemel tuleb autori arvates kaaluda tagaliidese tehnoloogiad.

Analüüsi tulemusena pakub autor välja komponendipõhise tarkvaralise lahenduse ja sellega kaasneva komponendipõhise lähenemisviisi ning tarkvaralise üldise arhitektuuri läbi tarkvaraarhitektuuri mustrite. Autori seisukohalt on väljapakutud kontseptsioon avalikule sektorile parem, eraldades arhitektuurilisel menetlussüsteemist kliendi portaali ja vähendades nii menetlussüsteemi arendamise keerukust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 45 leheküljel, 7 peatükki, 19 joonist ja 1 tabelit.

Abstract

Component-Based Software Development Framework as a Public Sector Software Platform

The aim of the master's thesis was to analyse the trends of the technology and software development company Nortal in creating software solutions to Estonian public sector and to understand whether the chosen technology platform is appropriate to develop procedural systems.

The author compared back-end and front-end technologies by software platform solutions and by layered and microservices architectural patterns. ThoughtWorks and Nortal's language and frameworks technology radars, statistics from the jobs site Indeed and Nortal's internal survey "SOA, Angular and modular architecture" were used for analysis.

During the course of the analysis it was discovered that the main problems in developing on front-end technologies are related to the issue that from an architectural point of view there are more software layers, many different technologies, functionality doubling and restrictions connected to the service oriented architecture.

As a result the author reached a conclusion that the way it is currently used, the chosen technology platform is not the best option for Nortal to create procedural systems for Estonian public sector. After analysing the problems that are occurring in developing on front-end technologies, the author proposed a suitable solution, introducing a back-end technology based software development framework with a component-based approach.

In the final part of the thesis the author's presented his views on software solution and architectural patterns that could be more suitable for Estonian public sector for creating software solutions. Among these was the idea of reducing the complexity of procedural systems by separating public sector customer portals.

The thesis is written in Estonian and contains 45 pages of text, 7 chapters, 19 figures and 1 table.

Lühendite ja mõistete sõnastik

Nortal AS	Nortal*,* algselt oli nimeks Webmedia, rahvusvaheline strateegia muutuse, tehnoloogia ja tarkvaraarenduse ettevõte.
SOA	*Service-Oriented Architecture*, teenustele orienteeritud arhitektuur, kus iga teenus on eraldiseisev ning täidab kindlat eesmärki. Teenused suhtlevad omavahel läbi komponentide või erinevate võrgu protokollide vahendusel. [1]
REST	*Representational state transfer* või *RESTful* on suhtlusstandard, mille vahendusel suhtlevad erinevad süsteemid tekstiliselt üle interneti. [2]
MVC	*Model-View-Controller*, tarkvaraline arhitektuuriline muster, millega kirjeldatakse ära tarkvara kolme komponendi koostöö, ehk mudel (*Model*), mis hoiab endas andmeid, vaade (*View*), mida näeb lõpptarbija ning kus kuvatakse välja mudeli andmestik ning kontroller (*Controller*), mis teenindab päringuid ning on mudel objekti ja vaate kokku viija rollis. [3]
JavaScript	On objektorienteeritud programmeerimiskeel, mida kasutatakse peamiselt veebilehtede skriptimiseks. [4]
JSON	*JavaScript Object Notation* on kergekaaluline andmeedastusformaat. Tegemist on tekstifailiga, mis hoiab endas andmeid organiseeritud kujul [5]
JSP	*JavaServlet Pages,* tagaliidese tehnoloogia, mille abil genereeritakse dünaamiliselt veebilehekülgi. [6]
Eesliides	*Front-end*, millega viidatakse kasutajaliides poolsetele tehnoloogiatele, mis töötavad kasutaja veebilehes.
Tagaliides	*Back-end*, millega viidatakse serveri poolsetele tehnoloogiatele, mis töötavad näiteks rakendusserveris.
Veebilehitseja	Veebilehitseja ehk brauser on selline tarkvararakendus, mille abil saab lokaalses arvutis vaadata/kuulata Internetis (või ka kohtvõrgus) paiknevatel veebilehtedel olevat teksti, pilte, videot, heli- ja teisi faile ning muud informatsiooni. [7]
Tarkvara radar	Populaarsete tehnoloogiate, tööriistade, platvormide, keelde ja raamistike kaardistamise platvorm.
Päring-vastus tehnoloogia	Tehnoloogiad või nende kogum, kus iga päringule tuleb vastussõnum koos õnnestumise või mitteõnnestumise infoga ning vastavalt päringu iseloomule koos vastuse kehaga, milleks

	võib olla uus leht HTML vormingus.
Topoloogia	Kujundite üldiste omaduste uurimine. [8]
API	*Application Programming Interface,* kogum alamprogrammi mõisteid, protokolle ja tööriistu tarkvara ehitamiseks. Määrab kindlad reeglid erinevate tarkvarakomponentide vaheliseks suhtlemiseks. [9]
Regressioonitestimine	Tarkvara testimine eesmärgiga otsida vigu pärast koodi olulist muutmist. Täpsemalt otsitakse tarkvara regressioone ehk vanu vigu, mis võivad uuesti ilmuda. [10]
Java	Java on platvormist sõltumatu objektorienteeritud programmeerimiskeel. [11]
PHP	*Hypertext Preprocessor,* skriptimiskeel, mida kasutatakse peamiselt serveripoolsetes lahendustes dünaamiliste veebilehtede loomisel. [12]
CSharp	C#, platvormist sõltuv multi-paradigmaline programmeerimiskeel. [13]
ESB	*Enterprise service bus,* tehnoloogia, mille abil teostatakse liidestus erinevad süsteemide vahel. Üldjuhul leiab rakendust teenustele orienteeritud arhitektuuris. [14]
Ajax	*Asynchronous JavaScript and XML,* tehnika, mille abil luuakse asünkroonsete päringutega veebilahendusi. [15]
HTML	*HyperText Markup Language* ehk hüperteksti märgistuskeel on keel, milles märgendatakse veebilehti. HTML paneb paika üksnes dokumendi struktuuri, kuid võimaldab veebiehele kaasata skripte, eeskätt JavaScripti ja CSS-i, millega kirjeldatakse lehe kujundus. [16]
CSS	*Cascading Style Sheets* ehk kaskaadlaadistik on keel, milles märgitakse veebilehe kujundust. [17]
Teek	*Library,* kollektsioon funktsioone, makrosid, klasse vms komponente, mis on mõeldud korduvkasutuseks programmides. [18]
PDF	*Portable Document Format*, PostScriptil põhinev arvuti riist– ja tarkvaraplatvormist sõltumatu elektrooniliste dokumentide vorming. [19]
PostScript	Programmeerimiskeel graafiliste objektide (tekst, vektorina kirjeldatud objektid nagu ring, ruut, joon jms ning rastergraafika) kirjeldamiseks sõltumata tulemust realiseerivast seadmest (printer, kuvar vms). [20]

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Tänapäeva eesliidese tehnoloogiad arenevad kiires tempos, tulevad välja uued ning kaovad vanad. Keeruline ei ole luua uusi tehnoloogiaid lahendades omal viisil mingit tarkvaralist probleemi. Tähelepanuväärne on asjaolu, et uusi tehnoloogiad paiskavad maailma paljuski suured ettevõtted nagu Google või Facebook. Sellised ettevõtted loovad endi jaoks sisemiseks kasutamiseks mõeldud lahendusi, milles mingil hetkel nähakse võimalust pakkuda neid ka laiemale maailmale. Selliselt avalikustatud toode võib tunduda kui tasuta toode, kuid samas toote looja saab kasutajaskonnalt midagi olulist rohkem tagasisidena, kasutajatelt ideid mida teha toote juures paremaks, tarkvaras esinevad probleemid leitakse kiiremini, kuni sinnani välja, et väljastatakse raamatuid, käiakse koolitamas jne. Ehk toote looja saab siin üldiselt siiski kaudselt tulu.

Facebook [21] on tulnud välja tehnoloogiaga React [22]. Tegemist on avatud lähekoodiga JavaScripti teegiga, mida kasutatakse veebilehitsejas töötava kasutajaliidese loomiseks. Reacti kasutusele võtmisel käib vahend tihedalt käsikäes tehnoloogiaga Redux [23]. Tegemist on samuti avatud lähtekoodiga teegiga, mis leiab kasutust rakenduste erinevate olekute ja nendega kaasas käivate andmete hoidmiseks.

Google [24] on omakorda tulnud välja tootega Angular, mis lahendab sarnast probleemi nagu Facebooki poolt loodud React, kuid tegemist on tehniliselt kompaktsema veebilehtede arendamise tugiraamistikuga. Kõige esimene versioon koodnimega AngularJS [25] tuli välja 2010.aasta lõpus. Aastal 2016 tuli välja Angular 2+ [26], mis on oma ülesehituselt täiesti erinev võrreldes eelkäiaga ning üleminek uuele versioonile on praktiliselt võimatu.

Mitte ainult Google ja Facebook ei tule turule oma toodetega. Tehnoloogiate turul on sarnaseid probleeme lahendavaid tooteid veelgi nagu näiteks Vue.js [27], Aurelia, jne. Kusjuures Aurelia [28] on tõstmas kõvasti pead teiste seas.

Nagu eespool lähtub on eesliideses leidnud oma koha erinevad tehnoloogiad, millede vahel tuleb teha oma valik. Kuid kas me oskame valida, kas me alati teame ka

missugused on valikuvariandid? Kas aga üldse on vaja keskenduda eesliidese tehnoloogiale või mõni teine suund, kui on vaja keskenduda siis missugustes lahendustes?

1.1 Eesmärgid

Käesolevas töös üritab autor leida vastuse küsimusele, mis hetkedel tuleks avaliku sektori menetlussüsteemi lahendustes keskenduda uutele ja kiiresti arenevatele eesliidese ja mis juhtudel tagaliidese tehnoloogiatele, tutvustades ja võrreldes seeläbi kahte tarkvara arhitektuurilist mustrit.

Töö keskmes on tarkvarasüsteeme arendav ettevõte Nortal, kes on loonud erinevaid menetlussüsteeme avalikule sektorile. Ettevõte on liikunu arendatavate tarkvaraliste lahendustega eesliidese tehnoloogiate suunas. Autor analüüsib, kas võetud suund on alati õigustatud vaadates peale eesliidese tehnoloogiate valikuga tekkinud probleemidele. Probleemide analüüsimisel on eesmärgiks pakkuda välja autori omapoolne lahenduste kontseptsiooni, mis autori seisukohalt vähendab keerukust avaliku sektori menetlussüsteemide arendamisel.

Magistritöö eesmärgi toetamiseks tutvustab autor tema poolt loodud komponendipõhist tarkvara arendusraamistikku. Antud raamistik sai tööalaselt loodud ühe ministeeriumi kliendiportaali arendamise käigus. Autoril oli raamistiku loomisel põhiroll, ta mõtles välja raamistiku üldise tööpõhimõtete kontseptsiooni ning realiseeris raamistiku põhikomponendid, mis võimaldasid hiljem teistel meeskonna arendajatel luua reaalseid kuvasid ning sellega uut funktsionaalsust. Autor näeb, et loodud raamistiku kontseptsioon on sobiv menetlussüsteemide arendamiseks.

1.2 Metoodika

Käesolevas töös keskendub autor kõigepealt erinevate trendide analüüsimisele vaadates seeläbi peale ThoughtWorks ja ettevõtte Nortali tarkvaraarenduse keelde ja raamistike radarile. Tehnoloogilise suuna valiku tegemisel on oluline arvestada tarkvaratrendidega ning tööturul toimuvaga. Radari peatüki lõpus toob autor välja omapoolsed seisukoha.

Seejärel uurib autor kihilist- ja mikroteenuste arhitektuurilist mustrit, mis on vastavalt leidnud laialdast kasutamist taga- ja eesliideste lahendustes, tuues välja mustrite plussid

ja miinused. Mustrite mõistmine on oluline, kuna need on teesi ühe olulisemad pidepunktid ees- ja tagaliidese võrdluses. Autor võrdleb mustrite kasutust tarkvaraarenduses läbi enda ja ettevõtte Nortal töötajate kogemuste. Lisaks on uurimisalas rahvusvahelisel tööturul toimuvat peegeldav keskkond Indeed.

Magistritöö teises osas tutvustab autor tema poolt väljatöötatud tarkvaralist lahendust, vaadates sellele peale tehnilisema pilguga. Tegemist on tarkvaralise kontseptsiooniga, mis autori arvates sobib paremini avaliku sektori menetlussüsteemide arendamiseks. Lisaks kirjeldab loodud raamistik komponendipõhist lähenemisviisi, mis autori arvates toob endaga kaasa lisaväärtust tarkvaraarendamisel.

Töö lõpuosas analüüsib autor avalikus sektoris tehtud tehnoloogilisi otsuseid valikus ees- või tagaliidese tehnoloogiates. Autor toob välja omapoolsed seisukohad ning tarkvaralise arhitektuurilise kontseptsiooni, mis tema arvates on avaliku sektori jaoks sobivam menetlussüsteemide tarkvaraaredusplatvorm.

2 Tarkvaraarenduse keelde ja raamistike trendid

Tarkvara tehnoloogiate turul on ja lisandub üha rohkem erinevad tehnoloogiaid. Arendusprojektides saab rakendada kellegi poolt juba loodud loomet. Kolmandate osapoolte komponentide kasutamine peab olema aga alati põhjalikult läbi mõeldud.

Erinevate jätkusuutlikumate tehnoloogiate valimisel aitab kaasa suurt populaarsust leidnud tarkvaraarenduse radar. Esimesena tuli radariga välja ThoughtWorks [29]. Radar kujutab endas populaarsete tehnoloogiate, tööriistade, platvormide, keelde ja raamistike kaardistamise platvormi. Tegemist ei ole kindlasti vahendiga, mille toel tuleks teha sügavuti analüüsi või suuri otsuseid. Tegemist on pigem suuna näitajaga. Radaris on kasutust leidnud neli kategooriat:

- *Adopt* – tehnoloogiad, mille kasutusele võtmine on soovitav.
- *Trial* – tehnoloogiad, mille kasutusele võtmine on soovitav väiksemates projektides, kus on teatud risk talutav.
- *Assess* – tehnoloogiad, mida võiks uurida ja katsetada.
- *Hold* – tehnoloogiad, millest tuleks loobuda.

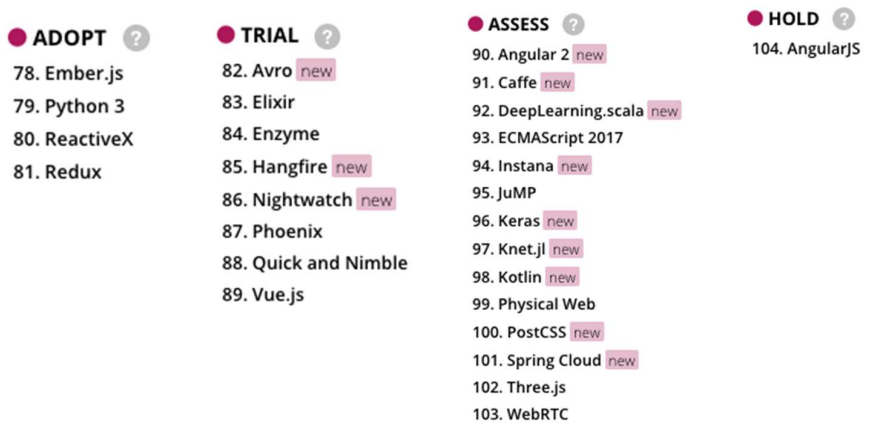
Antud töö raames keskendub autor ThoughtWorks ja Nortali koostatud keelde ja raamistike radari vaatele ning eesliidese ja Nortalis kasutust leidnud tehnoloogiatele.

2.1 ThoughtWorks

ThoughtWorks publitseerib uue radari seisu keskmiselt iga kuue kuu tagant. Radari koostajateks on ettevõtte mitmed vanemtehnikud, kes töötavad samas asutuses ning kasutavad oma projektides igapäevaselt radarisse märgitud tehnoloogiaid. Ettevõttes töötab orienteeruvalt 4000 inimest.

Ettevõttes Nortal suurt populaarsust omav AngularJS on aasta 2017 märtsi seisuga (Joonis 1) kaotanud väga tugevalt oma positsiooni liikudes viimasesse kategooriasse.

Seetõttu AngularJS kasutamine ei ole enam soovitatud. Antud olukord tuleneb antud töö autori arvates sellest, et Angular 2 on tulnud eelmisest aastast välja ning on hetkel positsioneeritud kolmandasse kategooriasse.



Joonis 1. Thoughtworks arenduskeelde ja raamistike radar

Radaris välja toodud enamus tehnoloogilistest raamistikest on loodud toimimaks eesliidestes. Peale AngularJS ei ole autorile teadaolevalt Nortali avaliku sektori projektides leidnud kasutust mitte ükski teine radaris väljatoodud eesliidese tehnoloogia. ThoughtWorks ajalugu näitab, et AngularJS ei ole jõudnud radaris aastast 2014 kunagi esimesse kategooriasse.

Lähtudes ThoughtWorks radaris väljatooduga ei tohiks juba aastast 2014 kasutada AngularJS'i suuremates tarkvaraprojektides. Angular 2 tuleks uutes projektides käsitleda ettevaatlikult ja ainult uurimise eesmärgil.

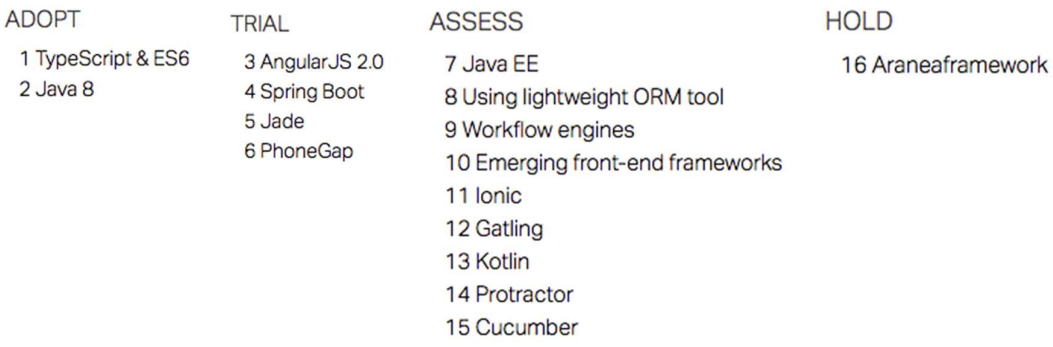
Lisana (Lisa 1) on täiendavalt välja toodud tervikpilt ThoughtWorks'i arenduskeelde ja raamistike radarist.

2.2 Nortal

Ettevõtte Nortali arenduskeelde ja raamistike radar [30] on ülesehituselt sarnaselt ThoughtWorks poolt looduga. Üks suuremaid vahesid seisneb siin selles, et radarit uuendatakse kord aastas, tehnikute tuumiku kokku saamisel. Võrreldes ThoughtWorks'i radariga on Nortali radari seis vanem. Nortali töötajate poolt kokku pandud radar peegeldab Nortali projektides toimuvat ning arvestada tuleb asjaoluga, et tegemist on

ettevõtte ülese radariga ning siia ei ole koondunud ainult avaliku sektori projektides arendajate mõtted.

Nortali radar (Joonis 2) näitab võrreldes ThoughtWorks'i radariga teistsugust seisu. Radaris on esindatud tagaliidese tehnoloogiad nagu Java, SpringBoot ning Nortali enda poolt arendatud Aranea raamistik. Aranea raamistik illustreerib ilmekalt kui kaua on võimalik arendada ja hoida üleval süsteeme, ilma et tehtaks suuri ümbermuudatusi või tehnoloogiate välja vahetamisi. Aranea raamistik on kasutuses jätkuvalt erinevates Nortali poolt arendavates tarkvaralistes lahendustes.



Joonis 2. Nortali arenduskeelde ja raamistike radar

Nortali radaris ei ole esindatud AngularJS. Samas Nortalis on erinevaid projekte, kus AngularJS leiab laialdast kasutust ning terved süsteemid on antud tehnoloogiaga abil üles ehitatud. Seetõttu peaks käesoleva töö autori arvates AngularJS olema vähemalt samal tasemel nagu on Aranea raamistik. Oluline on Nortali radari juures märkida, et soovitatud on kasutusele võtta ja liikuda edasi Java 8 ning TypeScript ja ES8'ga.

2.3 Mida saab trendidest oletada?

Lähtuvalt radaris toodule saab võtta seisukoha, et tehnoloogia Angular kasutuselevõtmine ei ole soovitatav uutes suuremates projektides.

Autor ei nõustu sellisel kujul ThoughtWorks'i radaris välja toodud trendidega menetlussüsteemide arendamise vaatevinklist. ThoughtWorks ei kajasta tagaliidese tehnoloogiaid, mis on olulisel kohal suurte süsteemide arendamisel. ThoughtWorks radaris välja toodud tehnoloogilist suunda tuleks kaaluda lahendustes, mis on teenuste

spetsiifilised ning kasutajaliidesele on seatud tugevamad nõuded. Sellisteks süsteemideks on näiteks kliendile suunatud portaalid.

Nortali radaris on tähelepanu all tagaliidese tehnoloogiad. Suurte äriliste lahenduste loomisel on oluline, et lahendus, mis kirjutati kaks aastat tagasi töötaks täna ja tulevikus, sõltumata sellest kas veebilehitseja on muutunud või turult kadunud ning selle asemele on tulnud uus veebilehitseja. Käesoleva töö autori arvates suudavad sellist stabiilsust pakkuda eesliidese tehnoloogiate asemel tagaliideste tehnoloogiad. Autor selgitab oma seisukohta antud töö järgnevates peatükkides.

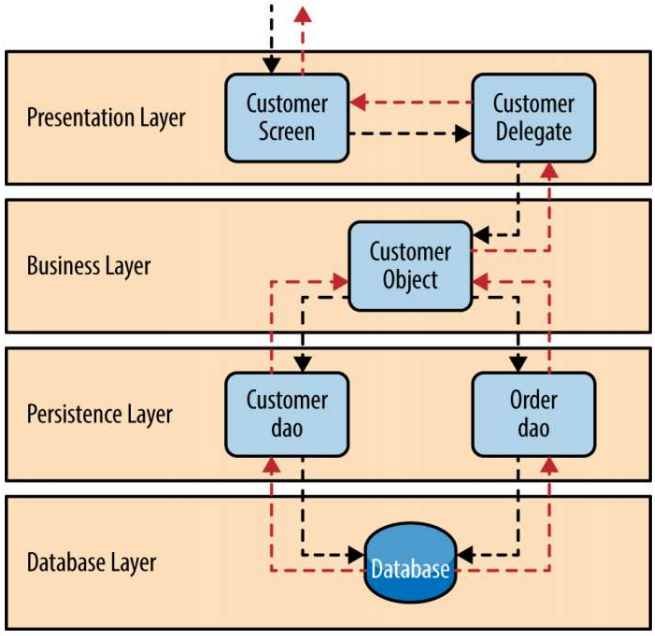
Autori üldine seisukoht on, et sõltumata radarites välja kuvatud trendidest tuleb igale tarkvaraprojektile vaadata peale individuaalselt. Eristada tuleb menetlussüsteeme ja kliendiportaale. Antud süsteemidele seatud nõuded on erinevad. Süsteemide kasutajaskond ja eesmärk määravad tehnoloogilised ja arhitektuurilised valikud.

3 Arhitektuuri mustrid

Sõltuvalt tehnoloogilistest valikutest on erinev loodava süsteemi tarkvaraline arhitektuur. Käesolevas peatükis keskendub autor kihilise ja mikroteenuste arhitektuuri mustrite kirjeldamisele. Antud mustrid võimaldavad eristada arhitektuuriliselt taga- ja eesliidese tehnoloogiaid ja nendest tulenevaid tarkvaralisi vajadusi.

3.1 Kihiline arhitektuuri muster

Kihilise arhitektuuri mustri abil kirjeldatakse tarkvara kihte horisontaalselt ja komponendi põhiselt. Igal arhitektuuri kihil on rakenduses kindel roll ja vastutus. Antud mustri puhul saab mõelda lähtudes päringu ja vastuse põhimõttest. Päringu teeb kasutaja läbi veebilehitseja ning antud päring ehk korraldus läbib kõik arhitektuurilised kihid, alustades esitluskihist liikudes kuni andmebaasi kihini välja. Vastavalt vajadusele võib päring jätta teatud kihid vahele näiteks liikuda ärikihist otse edasi andmebaasi kihti. M. Richards [31] on kirjeldanud väga hästi ära kihilise arhitektuuri üldise kontseptsiooni (Joonis 3).



Joonis 3. Kihilise arhitektuuris korralduse liikumine

Nagu joonisest lähtub läbib päring andmekihid ülevalt alla. Teatud juhtudel võib esineda, et päring ei jõua välja andmekihti, kuna varasemates kihtides oli võimalik päring teostada ilma täiendavate andmete pärimiseta.

Kihiline arhitektuur hoiab tarkvara üldist tööpõhimõtet lihtsana, lisamata juurde päringukeerukust. Üldjuhul teenindatakse igat päringut eraldiseisvana. Antud arhitektuuri muster on leidnud laialdast kasutust tagaliidese tehnoloogiates. Üldine praktika antud mustri juures on see, et tagaliides genereerib vastavalt päringu sisule uue kasutajaliidese kuva, mis tagastatakse päringu tulemusena ning kuvatakse välja veebilehitseja poolt.

Tegemist on laialdast kasutust leidnud arhitektuurilise mustriga. Laialdast kasutusele võtmist toetab asjaolu, et mustriga kaasneva üldise lähenemisviisi rakendamine ei ole keeruline. Enamus ettevõtteid arendavad tarkvara eristades tarkvarakihte ning seetõttu on antud muster ettevõtete jaoks loomulik valik.

Kihilisel arhitektuuri mustril on omad plussid ja miinused, mida tuleks arvestada mustri kasuks või kahjuks otsustamisel.

3.1.1 Plussid ja miinused

Sõltumata kihilisusest võib tarkvarasse muudatuste sisse viimine olla ajakulukas. Näiteks kui on vaja, et läbi esitluskihi liiguksid uued andmed edasi andmebaasi kihti, võib see tähendada seda, et muudatusi tuleb teha kõikides kihtides. Kui muutub andmekoosseis siis tuleb järele aidata andmekiht (*Database layer*). Vajadus võib olla vaadata üle salvestusreeglid, seega tuleb muuta püsivuskihti (*Persistence layer*). Teatud andmete korral võib olla vajadus teha täiendavaid kontrolle või lisategevusi andmete terviklikkuse tagamiseks, seega tuleb muuta ärikihti (*Bussiness layer*). Kindlasti tuleb muuta ka esitluskihis (*Presentation layer*) andmete sisestamise ja vajadusel ka saatmise osa.

Tarkvara uuenduste paigaldamine võib olla ajakulukas. Sõltuvalt muudatusest võib ühe muudatuse tulemuseks olla see, et kogu rakendusele tuleb teha uus paigaldus, mis sõltuvalt rakenduse ülesehitusest ja paigalduse käigust toimuvatest tegevustest võib olla ajakulukas.

Hea tarkvara testitavus. Komponendid kuuluvad kindlatesse kihtidesse ning igat kihti on võimalik eraldiseisvalt testida. Mitte testimist vajavad kihid saab katta nn testandmetega, sellist tegevust nimetatakse ka mokkimiseks. Funktsionaalsuse testimine ja veasituatsioonide tuvastamine kui ka logide ja muude tegevuste jälgimine on testijate jaoks lihtsam, seetõttu on tarkvara testimine üldjuhul efektiivsem ja kiirem.

Selline tarkvarakihilisus võib mõjutada jõudlust, kuna päringul võib olla vajadus läbida kõik kihid jõudmiseks kõige alumisse kihti. Igas kiht töötleb sisendit ja väljundit ning selline tegevus võib olla ajakulukas.

Kehv mastabeeritavus, inglise keeles *scalability.* Kuna tegemist on monoliitset tarkvara peegeldava arhitektuuri muster siis on keeruline süsteemi paigaldada erinevatesse üksustesse näiteks serveritesse, et nad moodustaksid ühtse terviku ning jagaksid samas omavahel ressursse ja koormust. Üheks võimaluseks mastabeeritavuse võimekuse tekitamiseks on lisada tarkvara arhitektuuris täiendavalt ette tarkvara, mille eesmärgiks on jagada koormust erinevate üksuste vahel. Selline lahendus toob siiski juurde täiendava keerukuse näiteks juba sessiooni haldamises.

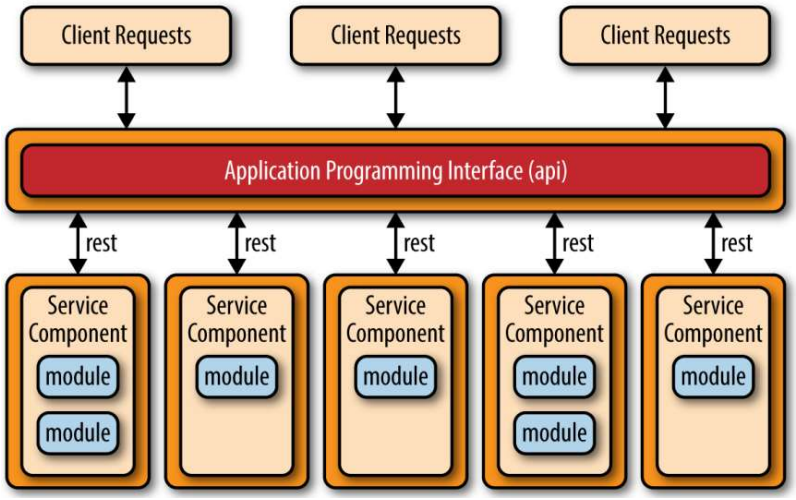
3.2 Mikroteenuste arhitektuuri muster

Mikroteenuste arhitektuuri muster on üha rohkem kanda kinnitamas. Tegemist on elujõulise alternatiiviga võrreldes monoliitsete ja teenus-orienteeritud arhitektuuriga (SOA) rakendustele. Antud muster on praeguse seisuga jätkuvalt arenemas ning seetõttu selle rakendamine tekitab segadust, kuna selgelt ei ole teada, mida antud muster endast kujutab ja kuidas seda tuleks kasutada.

Paremaks mustri mõistmiseks tuleb vaadata lähemalt tema kahte olulist kontseptsiooni. Üheks oluliseks määrajaks on see, et rakendatud on eraldiseisvate üksuste põhimõtet. Üksused sisaldavad teatud funktsionaalsust ning need on eraldisvalt paigaldatavad. Teine oluline kontseptsioon on see, et igale üksusele tuleks peale vaadata kui teenuste komponendile, mis hõlmab endas ühte või mitut moodulit. Ühe teenuse komponendi moodulid täidavad ühe eesmärgilist funktsiooni. Võrreldes SOA arhitektuuriga on mikroteenuste arhitektuur kergemini arusaadav, hallatav ning samas ka arendatav, kuna keerukust erinevate teenuste vahelises suhtlemises on vähem.

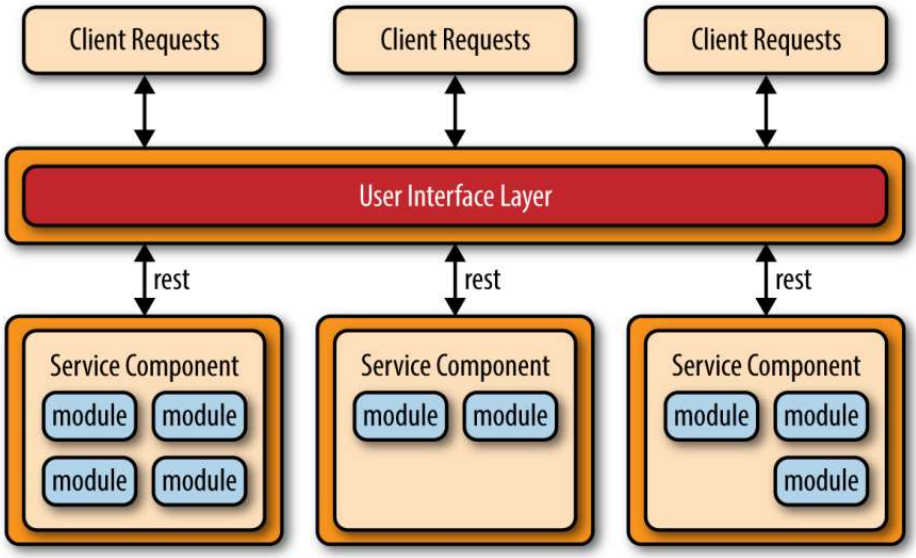
Levinud on kolm põhilist topoloogilist lähenemist: REST API põhine, rakenduse REST-põhine ja tsentraliseeritud sõnumipõhine.

REST API-põhine on kasulik veebilehtedele, kus on rakendatud väikesed individuaalsed teenused läbi mõne API. M. Richards [31] on ära visualiseerinud antud topoloogia üldise põhimõtted järgneval joonisel (Joonis 4).



Joonis 4. Mikroteenuste arhitektuuris API REST-põhine topoloogia

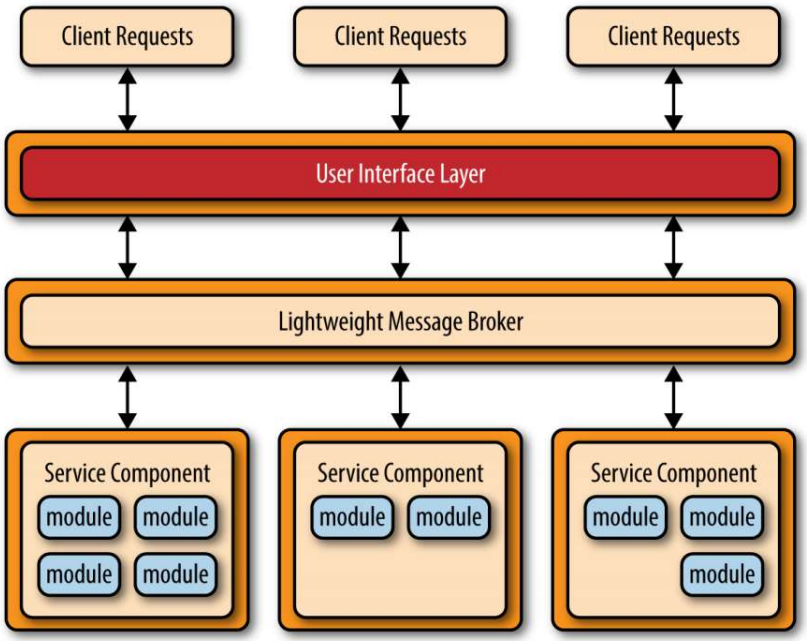
Rakenduse REST-põhine lähenemisviis erineb API REST-põhisest sellega, et kliendi päringud tulevad läbi traditsioonilise veebilehitseja või paksu-kliendi ärilise rakenduse. Paksu-kliendi all on mõeldud lahendust, kus rakendus on üles ehitatud nii, et ta suudab pakkuda rikkalikku funktsionaalsust ilma tagaliidese poole pöördumata. REST API-põhine lahendus töötab lihtsa API kihil. M. Richards [31] on ära visualiseerinud rakenduse REST-põhise topoloogia üldised põhimõtted järgneval joonisel (Joonis 5).



Joonis 5. Mikroteenuste arhitektuuris REST põhise rakenduse topoloogia

Rakenduse REST-põhises lähenemises on üldjuhul kasutajaliides rakendusena paigaldatud eraldi serverisse. Kasutajaliides läbi kasutaja interaktsiooni kutsub seejärel välja mõnes teises masinas paikneva REST teenuse, tarbides selle vahendusel ärilist funktsionaalsust.

Kolmas üldine lähenemisviis mikroteenuste arhitektuuri mustris on tsentraliseeritud sõnumite topoloogia. Tegemist on sarnase rakenduse REST põhise lähenemisviisiga, erinevusega, et REST teenuste asemel kasutatakse kergekaalulisi tsentraliseeritud sõnumivahendajaid nagu näiteks ActiveMQ, HornetQ, jne. Richards [31] on ära visualiseerinud üldised põhimõtted tsentraliseeritud sõnumivahetuse topoloogiast järgneval joonisel (Joonis 5).



Joonis 6. Mikroteenuste arhitektuuris tsentraliseeritud sõnumivahetuse topoloogia

Mikroteenuste arhitektuuriline muster pakub lahendusi esinevatele kitsaskohtadele monoliitsetes rakendustes ja teenusele orienteeritud arhitektuuri kasutavates lahendustes. Peamised rakenduse komponendid on tükeldatud väiksemateks loogilisteks tükkideks, toetades paremini üksusepõhist paigaldust, pakkudes seeläbi paremat mastabeeritavust.

3.2.1 Plussid ja miinused

Kihilise arhitektuuri mustriga tarkvaras muudatuste sisse viimine ei ole üldjuhul ajakulukas, kuna muudatused on isoleeritud teenuste komponendi põhiselt. See tähendab seda, et muudatus teostatakse tarkvaralise lahenduse ühes osas ning muudatuste sisseviimine ja paigaldamine ei mõjuta teiste teenuste komponentide tööd. Teenuse põhiseid komponente saab paigaldada eraldiseisvalt, kuna teiste teenuste komponentide vahelised seosed on nõrgad.

Hea tarkvara testitavus. Seoses eraldatuse ja ärilise funktsionaalsuse isoleerimisega sõltumatutesse rakendustesse on võimalik testimisel keskenduda konkreetsele funktsionaalsusele. Seetõttu on regressioontestimine lihtsam, kuna keskendutakse konkreetsele teenuste komponendile. Nõrga sõltuvuse tõttu on vähetõenäoline, et tehes arendusi ühes moodulis võib see vigu tekitada teise mooduli töös. Seetõttu ei ole üldjuhul vaja viia läbi terve rakenduse regressioonitestimist.

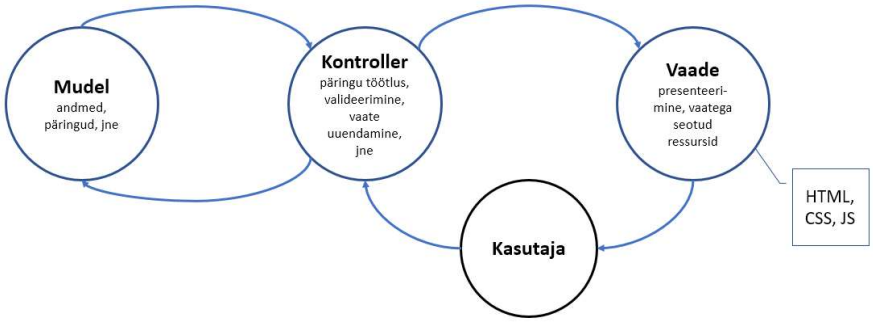
Mikroteenuste arhitektuuriline muster ei toeta üldjuhul hästi tarkvara jõudlust. Võimalik on küll luua rakendust, mis on hea jõudlusega, kuid arvestades tarkvaraliste komponentide paigutust, siis just see paigutus võib mõjutada üldist tarkvara töökiirust. Näiteks võivad erinevad teenuste väljakutsete lõppmoodulid asetseda füüsiliselt teises serveris, mistõttu hakkavad siin üldist kiirust mõjutama ka muud näitajad.

Hea mastabeeritavus, inglise keeles *scalability.* Kuna rakendus on tükeldatud eraldi üksusteks on iga üksus eraldiseisvalt mastabeeritav võimaldades nii igat üksust eraldiseisvalt optimeerida nii tarkvaralaselt kui riistvaraliselt.

3.3 Üldine mustrite võrdlus

Kihilisel ja mikroteenuste arhitektuuri mustritel on omad plussid ja miinused. Mustrite rakendamisel on tarkvaraarenduses oma kindel koht. Ettevaatlik tuleb olla kui on plaanis lahendada kõiki tarkvaranõudeid ühe arhitektuuri mustri abil. Näiteks kasutades teenuspõhist arhitektuuri siis seda tehes tuleb arvestada täiendava keerukusega.

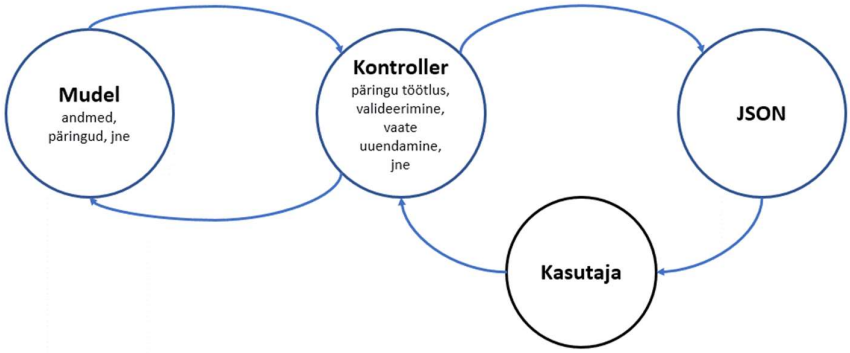
Kihiline arhitektuuri muster on üldlevinud, kuna selle rakendamisega kaasneb üldjuhul vähem keerukust. Üldine keerukus asetseb enamjaolt tagaliideses. Tagaliideses paiknevad ärilised nõuded, salvestamise reeglid, presentatsioonikihi vormingu loomine jne. Sisuliselt tähendab see seda, et arenduse vaates tuleb keskenduda tagaliideses toimuvale, mis muudab lihtsamaks nõuete analüüsi ehk sisendi loomise tarkvara arendamiseks. Tagaliidese tehnoloogiana on laialdast kastust leidnud MVC tarkvaraline arhitektuuriline muster. MVC üldiseks ülesandeks on tegeleda kasutaja saadetud päringu sisenditega, andmestiku ning vastuse genereerimisega. Joonis 7 kirjeldab üldise kontseptsiooni, kus kasutaja on sisendi andjaks ning MVC on saadud korralduste täitjaks. Kõige lõpus, peale korralduste täitmist, genereerib MVC kasutajale uue kuva, tagastades ressursid nagu HTMLi, CSS ja JavaScript, mille abil loob veebilehitseja kasutaja jaoks vajaliku kuva ehk vormi.



Joonis 7. MVC presentatsiooni kihiga tarkvaraline arhitektuuriline muster

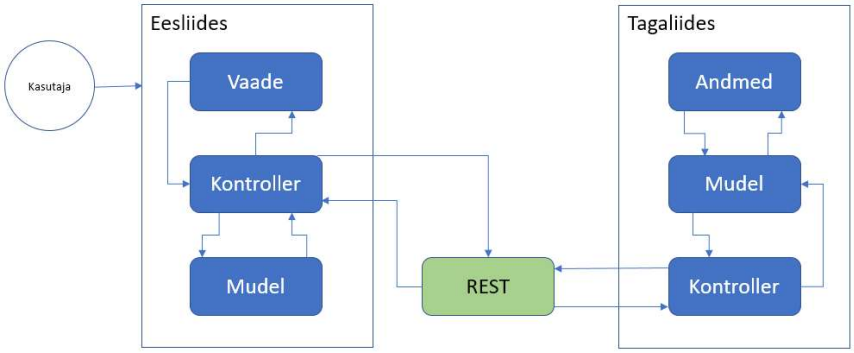
Mikroteenuse arhitektuurilise mustri juures liigub teatud keerukus tagaliidesest eesliidesesse. Liikuvaks keerukuseks on presentatsioonikihis olev kuvade loomine. Antud funktsiooni võtab üle eesliides. Tulenevalt sellest muutub tagaliidese funktsioon ning kuvade loomise asemel hakkab tagaliides tagastama andmeid näiteks kujul JSON. Eesliides hakkab tagaliidesega suhtlema teenuse väljakutsete vahendusel pöördudes serveris kindla aadressi poole, andes oma pöördumisega kaasa sisendparameetrid. Tagaliides töötleb eesliideselt saadud pöördumist ning tagastab vastuse. Tänu asjaolule, et kuva loomise funktsioon liikus eesliidesesse muutus sellega ka keerulisemaks eesliideses toimuv. Presentatsioonikihi funktsiooni ülevõtmisega tekivad eesliidesesse juurde täiendavad tarkvaralised kihid koos täiendavate funktsioonidega. Eesliides saab endale MVC ehk mudel-vaate-kontroller funktsioonid ning hakkab tegema seda, mida tegi varasemalt tagaliides.

Mikroteenuste arhitektuuri mustrilistes lahendustes tekib vajadus teostada eesliideses erinevaid valideerimistegevusi ning äriloogilisi kontrolle. Kõik kontrollid, mis teostatakse eesliideses tuleb teostada täiendavalt ka tagaliideses. Seda põhjusel, et eesliidest ei saa kunagi usaldada. Selline lähenemisviis toob endaga kaasa sarnaste kontrollide olemasolu ees- ja tagaliideses. Funktsionaalsuse dubleerimine muudab tarkvaraarenduse ajamahukamaks ning kergesti on tulema unustusvead, kus ees- või tagaliidesesse jäi mingi muudatus sisse viimata. Tagaliidesele jääb alles MVC funktsioon, selle erinevusega, et presentatsioonikihis tagastatakse JSON'it, mille tarbijaks on kasutajaliideses töötav eesliides. Joonis 8 kirjeldab üldist MVC kontseptsiooni, kus presentatsioonikihi ehk vaate osa on asendatud JSON'iga. Kõik muud tegevused, mis on presentatsioonikihist allpool on üldjuhul standardsed tegevused sõltumata kumma arhitektuurilise mustriga on tegemist. Teatud erisused kindlasti on nagu vigade töötlus ja tulemuste kuvamine, kuid ärilised tegevused on enamjaolt samad.



Joonis 8. MVC presentatsiooni kihita tarkvaraline arhitektuuriline muster

Joonis 9 visualiseerib üldist andmete voolu mikroteenuste arhitektuuri mustris, kus kasutaja on tegevuse algatajaks.



Joonis 9. Mikroteenuse arhitektuuri muster MVC vaates

Eesliidese kontroller täidab kasutaja poolt saadud korralduse ning saadab sisendparameetritega päringu tagaliidesesse. Tagaliideses on päringu vastuvõtjaks samuti kontroller, mille eesmärk on üldiselt sarnane eesliidese kontrollerige, valideerida sisendparameetrid, viia läbi vajalikud tegevused mudelobjekti täitmiseks ning vastusena eesliidesele saatmiseks. Eesliides teeb vastavalt tagaliideselt saadud vastuse põhjal edasi vajalikud toimingud näiteks kuvab uut infot või suunab kasutaja mõnele teisele kuvale, seda näiteks juhul kui on viidud läbi mõni muudatustegevus nagu salvestamine või kustutamine.

Mikroteenuste arhitektuuri muster võrreldes kihilise mustriga toetab hästi agiilse disaini printsiipe [32]. Muudatusi on lihtne viia sisse, esinevad nõrgad seosed teiste teenuste komponentide vahel ning paindlikum üksuste paigaldamine.

Lähtuvalt peatükis kirjeldatust on antud töö autori seisukohaks, et arhitektuurilise mustri valikul on oluline teha põhjalik eelanalüüs, selgitamaks välja täpsemad loodava süsteemi vajadused. Juhul kui analüüsi tulemusest lähtub, et planeeritaval lahendusel on monoliitse süsteemi tunnused, tuleks kaaluda kihilist arhitektuuri mustrit vastasel juhul tuleks vaadata mikroteenuste arhitektuuri suunas. Valiku tegemisel tuleb arvestada, et mikroteenuse arhitektuuri mustriga lahenduste arendamine on kallim ja ajanõudlikum kui kihilise arhitektuuri mustriga lahendused. Samas on kihilise arhitektuuri mustriga süsteem paindlikum ja paremini kohanev muutuvas maailmas.

4 Tarkvaraarendus

Sõltuvalt missugune arhitektuuriline muster valitakse muutub selle valikuga ka arendusprotsess. Üldised arendusprotsessi osadeks on nõuete analüüsimine ning tulemuse spetsifikatsioonina kirja panemine, prototüüpimine ehk lahenduse kasutajaliideses visualiseerimine, loodava lahenduse disainimine, lahenduse testimine, paigaldamine ehk kliendile üleandmine. Üheks osaks on ka haldustegevused nagu projekti juhtimine ja kliendisuhtlus. Iga loetletud arendusprotsessi osa on üldjoontes eraldiseivalt hinnatav ning planeeritav. Sõltuvalt valitud mustrist kaasneb või väheneb arendusprotsessi mingites osades keerukus näiteks spetsifikatsiooni, testimise või arendamise etapis. Järgnevalt vaadatakse peale arendusprotsessi vaates kihilisele ja mikroteenuste arhitektuuri mustritele.

4.1 Kihiline arhitektuuri muster tarkvaraarenduses

Peatükis „Kihiline arhitektuuri muster“ on kirjeldatud üldiselt arhitektuuri mustri põhimõtted. Käesolevas peatükis analüüsib antud töö autor, kuidas valitud arhitektuuri mustri kasutamine mõjutab tarkvaraarendusprotsesse.

Joonis 3 kihilise arhitektuuri mustri peatükis visualiseerib hästi mustri tööpõhimõtet. Erinevaid kihte on neli alustades esitlus- ja lõpetades andmebaasikihiga. Vahepeale jäävad veel äri- ja püsivuskiht. Selline arhitektuur hoiab tarkvara keerukust madalana, kuna kogu arendus toimub üldjuhul ühele platvormile suunatud tehnoloogiates. Nortali poolt arendatavates avaliku sektori menetlussüsteemides on tagaliidese arenduskeeleks Java.

Arenduskeel Java on üks populaarsemaid tagaliidese arenduskeeli võrreldes näiteks PHP ja CSharp’iga. Antud trendi toetab keskkonna Indeed statistika (Lisa 3), sõltumata asjaolust, et Java on hetkel langustrendis. Koos Javaga on langustrendis ka JavaScript. Võiks ju oletada, et seoses eesliidese tehnoloogiate arenemisega ja kasutusele võtmisega peaks JavaScript olema tõusutrendis. Indeed statistikast võib veel välja lugeda asjaolu, et Java positsioonidele tööpakkumus on küll langustrendis samas tööotsijate arv on tõusutrendis. Seetõttu tööjõuturul Java arendajatest puudust ei tohiks tulla. Mis omakorda räägib kihilise arhitektuuri mustri kasuks, kuna selle kasutamine on jätkusuutlikum näiteks kasvõi arvestades olemasolevat töökäte hulka.

Kihilise arhitektuuri mustri üheks suureks plussiks on selle keskendumine ühele kindlale tehnoloogiate kogumile, on siis see seotud Java või mõne muu arenduskeelega. Tarkvaraprojektis toimuvast on hea ülevaade, tegevused toimuvad järjestiku liikudes ülevalt alla ja alt üles. Üldjuhul tähendab see seda, et üks arendaja suudab toimetada kõikides tarkvarakihtides ning ei ole vaja spetsialiseeruda täiendavalt uute tehnoloogiate nagu näiteks eesliidese omadele. Tõsi, ilma eesliidese tehnoloogiateta ei saa. Samas kui läheneda komponendipõhise lähenemisviisiga siis väheneb veelgi eesliidese tehnoloogiate osakaal nende keerukuses. Tarkvaraprojektides üldised komponendid saavad valmis projekti alguses, seejärel hakatakse neid korduvalt kasutama kuni sinnani välja, et luuakse korduvkasutatavaid kuvasid. Kuna stabiilsemad eesliidese tehnoloogiad suuresti ei muutu siis ei ole vaja väga tihti antud komponente järele aidata.

Üheks tarkvaraarenduse lahutamatuks osaks on nõuete analüüs. Mida lihtsam on arhitektuuriline muster, seda lihtsam on teostada sellele ka analüüsi. Analüüsi väljundiks on uue funktsionaalsuse nõuded. Analüüs tähendab seda, et joonistatakse prototüüp, kasutades juba valmis prototüübitud komponent, pannakse kirja ärilised reeglid kuni vajadusel sinnamaani kust andmed pärinevad. Kogu tegevus on nn ühe süsteemi või kui võrrelda mikroteenuste arhitektuuri mustriga, siis mooduli või teenuste komponendipõhine. Sisuliselt ühe spetsifikatsiooni dokumendiga saab ära kirjeldada kogu uue funktsionaalsuse kõikide kihtide nõuded.

Kihilise arhitektuuri mustriga tarkvaralistes lahendustes on vähem modulaarsust ja tarkvara kihte. Tulenevalt sellest on lihtsam hinnata uue funktsionaalsuse loomisele kuluvat ajalist mahtu. Arendusprotsess on kiirem tulenevalt vähesest osapooltest arvust ning tarkvarakihtidest.

Käesoleva töö autori arvates on kihilise arhitektuuri mustri üheks oluliseks positiivseks näitajaks see, et antud mustri juures saab ja isegi tuleks kasutada sessiooni kasutaja andmete hoidmiseks. Sessioon tähendab seda, et on kindel koht, kuhu saab paigutada sisseloginud kasutajaga seotud andmed. Olgu siis nendeks andmeteks kasutajainfo või konkreetse protsessi või tegevuse andmestik. Selline sessiooni ehk oleku omamine lihtsustab mitmeid erinevaid tegevusi nagu andmekoosseisu kontrollimist või ajutiste andmete meeles pidamist. Sessiooni omamine võimaldab juba andmebaasist päritud andmed lisada sessiooni enda sisse, mis tähendab seda, et andmebaas on vähem koormatud ja teatud tegevused on kiiremad. Samas tuleb arvestada seda, et mida

rohkem andmeid paigutada sessiooni, seda suurem on serveri mälukasutus. Samas mälu on asi, mida saab üldjuhul alati lisada juurde.

Kihilise arhitektuuri mustri juures on ka miinuseid. Lähtudes päringu ja vastuse põhimõttest siis peale igat päringut genereeritakse uus kasutaja kuva. Presentatsioonikiht tihedalt seotud valitud tehnoloogiatega. Menetlussüsteemi kontekstis ei pruugi presentatsioonikihi selline jäikus olla otseseks probleemiks, kuna tegemist on süsteemiga, kus teostatakse menetlusprotsesse, millel on kindel vool ja tegevuste sammud. Üheks miinuseks antud mustri juures on aeganõudev tarkvara paigaldamine. Sõltuvalt süsteemist võib paigaldatav tarkvara olla suur ning paigaldustegevus võib olla aeganõudev. See ei tohiks olla suureks probleemiks, kuna paigaldust saab teha näiteks nädalavahetusel, mil ametnikud üldjuhul ei tööta. Teatud süsteemidel on muidugi omad nõuded näiteks meditsiini süsteemidel. Sellistele süsteemidele tuleb vaadata peale teise pilguga. Käesoleva töö fookuses on aga avaliku sektori menetlussüsteemid.

Kihiline arhitektuuri muster peegeldab sünkroonselt toimivat lahendust, mis tähendab seda, et mingite tegevuste teostamiseks tuleb saata kogu info serverisse ning kasutajale genereeritakse serveri poolt uus kuva. Nüüdsetes lahendustes ei pruugi see aga olla enam nii ning ka siin saab rääkida asünkroonsusest ja kullisside tagusest suhtlemisest, mida nimetatakse Ajax'iks. Ajax'i tehnika kasutamine võimaldab samuti saata serveri suunas päringuid nii, et kasutaja kuva ei muutu, kutsudes seeläbi välja erinevaid valideerimisreegleid või pärides uut infot. Võimalik on luua lahendust, kus eesliides on nn rumal ja ei tea ärireeglitest ja valideerimisreeglitest midagi. Tehes päringu serveri suunas teostab serveri ots vajalikud kontrollid ning tagastab eesliidesele vaid vajaliku tulemuse.

Erandiks, kus ei ole soovitav kasutada kihilist arhitektuuri mustrit on süsteemid, mis on väljaspool nähtavad nagu näiteks kliendiportaalid. Sellised süsteemid täidavad üldjuhul kitsast eesmärki, mis on mõeldud selleks, et klient saaks sisestada oma andmeid, vajadusel neid näha ning saada menetlusprotsesside tagasisidet. Selliste süsteemide puhul tuleks vaadata teiste arhitektuuriliste mustrite poole ning tuua see kihilise arhitektuuri mustri kõrvale. Seda tehes on tulemuseks see, et tekib nn hübriidlahendus, kus menetlussüsteem täidab oma rolli oma arhitektuuri mustriga ning kliendiportaal teist rolli näiteks mikroteenuste arhitektuuri mustriga. Kahe lahenduse kokku viimiseks

tuleks kasutada kas täiendavat andmevahetusmoodulit, mis on lüliks menetlussüsteemi ja kliendiportaali vahel (Joonis 19) või võtta kasutusse mõni ESB lahendus, mis on sõnumite vahendaja rollis.

4.2 Mikroteenuste arhitektuuri muster tarkvaraarenduses

Peatükis „Mikroteenuste arhitektuuri muster“ on kirjeldatud üldiselt arhitektuuri mustri põhimõtted. Käesolevas peatükis keskendub antud töö autor mikroteenuste arhitektuuris REST põhise rakenduse topoloogiale, kuna just selline muster on leidnud kasutust Nortalis arendatavates avaliku sektori projektides. Lisaks arvab käesoleva töö autor, et antud muster on ka sobivam menetlussüsteemi tarkvarale. Mikroteenuse arhitektuuri mustriga süsteemides on eesliides üldjuhul tunduvalt targem võrreldes kihilise arhitektuuri mustriga. See omadus on üks olulisemaid näitajad mikroteenuste arhitektuuri mustri rakendamise juures.

Joonis 5 mikroteenuste arhitektuuri mustri peatükis visualiseerib hästi mustri tööpõhimõtet. Olulised näitajad antud mustri juures on need, et juba arhitektuuri tasemel on eeldus, et kogu süsteem peab toetama asünkroonsust. Teine oluline näitaja on see, et erinevad moodulid on eraldatud loogilisteks tükkideks, moodustades nii teenuste komponentide komplekti.

Kui kihiline arhitektuuri muster keskendub tagaliideses toimuvale siis mikroteenuste arhitektuuri mustris on fookus liikunud eesliidesesse. Andmevahetuskihina on laialdast kasutust leidnud RESTful [5] lähenemine. Samas ei tohiks siin unustada teenustele orienteeritud arhitektuuri. Indeed statistikast (Lisa 4) lähtub, et SOA on aastast 2015 langustrendis ning RESTful on olnud vahelduva eduga tõusus ja siis languses. Samas tööotsijate vaatest on RESTful langustrendis ja SOA väikeses tõusutrendis. Kuna RESTful lahenduste vastane huvi paistab olevat langustrendis siis uute lahenduste planeerimisel tuleks seda asjaolu arvestada, kuna mingil põhjusel huvi töötaja vaatest selliste lahenduste vastu on praeguse seisuga raugemas. Lisaks heade RESTful lahenduste loomiseks tuleb alati loodavale peale vaadata SOA vaatest, seega seda rolli ei tohiks kindlasti vähe tähtsustada.

Mikroteenuste arhitektuuri mustriga lahendustes on arendamine võrreldes kihilise arhitektuuri mustriga aeganõudvam. Erinevaid loogilisi süsteeme on rohkem. Seetõttu

muudatuste sisse viimiseks võib olla vajalik arendada ka teisi mooduleid kui ka aidata järele muud nendega seonduvat. Arendusi võib vaja minna teha mõlemas nii ees- ja tagaliideses. Juhul kui on vaja arendada kuva, mis peab jälgima kasutaja sisestusi, valideerima sisestust, teostama äriloogilisi kontrolle, teostama erinevaid andmepäringuid testesse teenustesse, siis tähendab see seda, et kõike seda mida teeb eesliides peab hiljem tegema ka tagaliides. Seda põhjusel, et kasutaja sisendit ei saa ja ei tohi kunagi 100% usaldada. Seetõttu tuleb kõik sisestatu tagaliideses üksi pulki uuesti üle kontrollida ehk teha seda sama mida tegi eesliides. Sisuliselt tähendab see seda, et toimub tegevuste dubleerimine, mis omakorda võib olla veaohtlik, kuna kui eesliideses muutub reegel, siis peab see muutuma ka tagaliideses.

Mikroteenuste arhitektuuri mustris ei ole tavaks kasutada sessiooni mõistet ehk siin ei ole kohta kus hoida kasutajaga seotud informatsiooni sarnaselt kihilise arhitektuuri mustriga. See tähendab seda, et üldjuhul tagaliides teab vaid teatud autentimise informatsiooni ja sellega tema teadmus piirdub. Olekuvabadus ehk sessiooni puudumine tähenda aga seda, et kasutaja spetsiifilised andmed võivad asetseda ainult eesliideses. Tagaliides toimib nii, et iga pöördumise korral tehakse täiendavaid tegevusi näiteks andmete pärimisi. Puudub võimalus hoida kasutaja ajutisi andmeid kui just neid ei salvestata kuhugile maha. Selline lähenemine lisab tarkvaralahendusele teatud keerukuse lisategevuste näol.

Keerulisemaks muutub ka analüüsifaas (Lisa 5), kuna analüüsida tuleb kolme erinevat loogilist kihti nagu eesliides, andmeedastuskiht, tagaliides. Andmeedastuskiht on siinkohas üks olulisemaid ja suuremaid arendustempo mõjutajaid. Tegemist on kihiga, mille vahendusel suhtleb eesliides tagaliidesega, seetõttu peab selle spetsifitseerimine olema standardiseeritud sellisel kujul, et eesliides ja tagaliides saaksid andmetest ühtselt aru. Kui andmeedastuskihi eraldiseisev spetsifitseerimist ehk disainimist ei ole saavutatud, tähendab see seda, et taga- ja eesliides on keeruline eraldiseisvalt arendada. Taga- ja eesliidese eraldiseisev arendamine on antud töö autori arvates üheks oluliseks kasuteguriks mikroteenuste arhitektuuri mustris. Tegemist on ka üheks olulise põhjusega, miks tuleks seda mustrit teatud tarkvaralahenduste juures kasutada. Juhul kui aga sellist eraldiseisvuse taset ei ole saavutatud siis ei ole spetsialiseerumist ja arendaja peab valdama korraga ees- kui tagaliidese tehnoloogiaid. Kuna aga taga- ja eesliideses toimivad erinevatel põhimõttetel siis mõjutab see kindlasti arendaja õpikõverat ning ka tarkvara kvaliteeti, kuna kõikides tehnoloogiates ei saa olla spetsialist.

Testimise üldine automatiseerimine on mikroteenuste arhitektuuri mustri juures kindlasti parem, igat lahendust, moodulit kuni teenuseni välja saab eraldiseivalt testida. See on samuti üks olulisemaid eelistusi antud mustri kasuks. Samas kui rääkida testija poolsest testimisest siis antud töö autori arvates muster mõjutab seda kindlasti negatiivselt. Erinevaid kohti, kus viga võib tekkida on tunduvalt rohkem, kuna taga- ja eesliidesed toimivad ja logivad oma tegemisi erinevalt. Näiteks kliendi rakenduses tuleb jälgida veebilehitseja konsooli, monitoorida andmete liikumist pannes vajadusel kokku erinevate REST-teenuste väljakutsed, mis moodustavad tervikandmekoosseisu, jälgida tuleb tagaliideses toimuvat ja sealseid logisid. Testimise keerukuse tõusmise aspekti on välja toodud ka Nortalis läbi viidud uuringus (Lisa 5). Veakorral on vaja koostada ka raport, mis tähendab seda, et kõik need jälgitavad osad genereerivad mingi tulemi, mis tuleb ära kirjeldada, teatud juhtudel võib see olla väga aeganõudev. Lisaks võib arendajal olla keeruline viga korrata, seega parandamine võib sammuti rohkem aega võtta.

Arvestada tuleb asjaoluga, et palju äriloogikat asetseb mikroteenuste arhitektuuri mustri korral kasutaja arvuti veebilehitsejas, mis on kasutaja poolt vajadusel nähtav ning manipuleeritav. Kasutaja saab vaadata kuidas äriliselt rakendus töötab ning vajadusel ja oskuste olemasolu korral on tal võimalik teatud kontrollidest mööda hiilida. Antud asjaoluga tuleb alati arvestada ja see tähendab seda, et täiendavat aega tuleb panustada erinevate kontrollide teostamisse.

Mustri kasuks räägib kindlasti see, et tekivad eraldiseisvad moodulid, mida saab eraldiseivalt paigaldada, arendada, testida. Tekib eraldiseisev kasutajaliides, mida saab samuti eraldiseivalt arendada ja mis on samas ka paindlik. Lahendus on mastabeeritav. Andmevahetuskihti saab käsitleda eraldiseisvana.

Teatud tarkvaralistes lahendustes on mikroteenuste arhitektuuri mustril kindlasti väga tugev koht. Sellistes kasutajaliidestes, kus loodav lahendus peab olema paindlik, arvestama kiireid keskkondlikke muutusi kui ka muutusi võimalikes lähenemisviisides. Sellistes tarkvaralistes lahendustes tuleks just kaaluda antud mustri kasutusele võtmist. Tegemist oleks siin lõpptarbijatele suunatud lahendustega, mis täidavad kindlat eesmärki ja kus kasutajaskonda ei saa kuidagi kontrollida. Sellised süsteemid on liidestatud täiendava mooduli (Joonis 19) või lahendusega, mis on andmevahendaja rollis.

4.3 Arendusefektiivsus ettevõtte Nortali näitel

Ettevõttes Nortal on arendatud erinevaid süsteeme nii suuremaid kui väiksemaid ning aegade jooksul on ettevõte tulnud välja erinevate tarkvaraliste toodetega, mis on leidnud laialdasemat kasutust ka väljaspool ettevõtet. Üheks selliseks tooteks on tarkvaraarendusraamistik nimega Aranea raamistik [33]. Tegemist on sisuliselt mudel-vaade-kontroller (MVC) põhise veebiarendusraamistikuga. Antud raamistik oli loomishetkel oma ajastust ees, turul midagi sarnast tol hetkel ei olnud. Praeguseks on antud raamistik jätkuvalt kasutuses erinevates asutuste süsteemides nagu näiteks Töötukassas. Kuna raamistikku enam edasi ei arendata siis liigutakse ka Töötukassas edasi uute tehnoloogiate peale. Aranea raamistik on hea näide sellest, et kui kaua võib üks tarkvaraarenduse raamistik elada. Arvatavasti elaks raamistik siiani edasi, kui sellele oleks taha tulnud korralikum kommuun.

Seoses uute tehnoloogiate peale tulemisega ning antud juhul näiteks Aranea raamistiku eluea lõppemisega vaadati Nortalis edasi uute tehnoloogiate suunas. Tarkvara turul otsuste tegemise hetkel midagi head tagaliidese tehnoloogiate osas pakkuda ei olnud. Pead kergitas siis Vaadin raamistik [34], kuid see ei sobinud oma vähepaindliku esitluskihi tõttu. Seetõttu vaatas ettevõte eesliidese tehnoloogiate suunas, otsustades AngularJS kasuks. Samas ei mõeldud läbi, mida see valik endaga täpselt kaasa toob. Sisuliselt päevapealt muutus lähenemisviis ja vaja oli kompetentsi JavaScripti arendajate vallas. Tekkis vajadus eesliidese arendajate järele. Selliseid inimesi aga ettevõttes otseselt ei olnud ja siin tuli appi ettevõtte Visuali meeskond. Otsuste tegemisel hetkel ei lisaks peale arendusprotsessidele, mis muutusid tänu tehnoloogilise valiku tegemisele. Siiani ei ole ettevõttes juurutatud head viisi kuidas kirjutada spetsifikatsioone, milles on terviklikult ära kirjeldatud ees- ja tagaliites ning andmevahetuskihis toimuv.

Uue tehnoloogia valikuga oli ettevõttes näha langusi arendustempos ning enam ei oldud nii efektiivsed. Ettevõttes otsustati läbi viia uuring eesliidese tehnoloogiate peale üle läinud projektides seas, selgitamaks välja mis seisukohal ja mis probleeme nähakse projektides.

4.4 Nortalis mikroteenuse arhitektuuri mustriga lahenduste sisemine uuring ja analüüs

Nortalis viidi läbi uuring peale uute eesliideste tehnoloogiate peale üle minemist. Uuringus toodi välja erinevad murekohad võrreldes Aranea raamistikuga. Antud uuring on lisatud lisana (Lisa 5). Uuringus selgus, et on palju probleeme, millede võimaliku tekkimise peale algselt ei oldud mõeldud nii analüüsi, arenduse kui testimise valdkonnas.

Analüüsifaasis selgus, et ei ole enam võimalik luua teatud funktsionaalsust, mida sai teha varasemate tagaliidese tehnoloogiatega, näiteks muutus esiliideses andmetabelite käsitluses. Nimelt ei olnud võimalik enam teostada üle tabeli andmete sorteerimist ja filtreerimist juhul kui tabeli andmestik pannakse kokku erinevate teenuste vahendusel. Sellised andmed pärinevad erinevatest kohtadest ning tervikandmekoosseis andmete filtreerimiseks või sorteerimiseks puudub. Lisaks on välja toodud põhjused, miks analüüsimise peale on kulunud rohkem aega. Üheks põhjuseks oli see, et ees- ja tagaliides tuleb dokumenteerida eraldi ning probleeme on andmekihi kirjeldamisel.

Arenduse valdkonnas toodi välja erinevate tarkvara kihtide ja moodulite rohkus. Iga kiht nõuab eraldi aega ja arendusressurssi. Lisaks tuleb iga kiht katta täiendavalt testidega. Puudub tarkvaraline abivahend teenuste kirjeldamiseks, mis on sisuliselt väga suureks takistuseks selleks, et arendada eraldiseisvalt ees- ja tagaliidest.

Testimise vallas toodi murekohana välja teenuste rohkus, mis moodustavad kokku tervikandmestiku. Seetõttu on keeruline testimise käigus kokku saada tervikpilti andmekoosseisudest. Lisaks toodi testimise poole pealt välja, et teenuste poolt tagastatavad andmestikud on suuremahulised, mis tähendab seda, et nende väljapärimine ja visuaalne kontrollimine võtab palju aega. Antud töö autori seisukoht on, et teenuste poolt tagastatavad suuremahulised andmetikud viitavad kehvale teenuste disainile.

Uuringus toodi välja, et üldiseks probleemiks kõikides valdkondades oli suur õpikõver. Tehnoloogiad ja töövahendid olid uued ja nende tundma õppimine võtab aega.

Lisaks toodi välja veel mitmeid teisi erinevaid probleeme, millele uuringu käigus viidati. Uuringu tulemus on toodud täies mahus välja lisana (Lisa 5) ning antud töö

autor soovitab uuringu tulemusega tutvuda kõigil, kes plaanivad arendada mikroteenuste arhitektuuri mustriga tarkvara.

Nortal uuris täiendavalt, mida arvavad juhtivaarendajad sellest, millal jõutakse arenduse tempoga sama efektiivsustasemele nagu oldi Aranea raamistikuga.

4.4.1 Nortali juhtivarendajate arvamus valitud tehnoloogia arendustempole

Lisaks tehnoloogia valikust kaasnenud murekohtadele uuriti Nortalis, mida arvavad ettevõtte juhtivarendajad tarkvara arendamise tempost uutel tehnoloogiatel.

Sõltuvalt projekti suurususest ja realiseeritavast keerukusest vastasid kõik juhtivarendajad sellele küsimusele ühtselt, et antud tehnoloogiatel arendades ei jõuta kunagi arendamise tempos samale tasemele nagu oldi Aranea raamistikuga. Põhjuseid selleks on mitmeid, kuid üheks kindlaks näitajaks oli siin täiendavate kihtide olemasolu ja sellega kaasnev keerukus. Keskmine arendustempo langus jäi 18% juurde, see tähendab seda, et uute lahenduste realiseerimisel ollakse 18% vähem efektiivsed. Antud uuring ei kajasta efektiivsuse poole pealt analüüsi ning testimise vahet. Kuid arvestades probleeme, mis toodi ka uuringus välja, siis saab oletada, et ka neis rollides ollakse vähem efektiivsed.

4.4.2 Tänane seis ja järeldused

Vahepeal on Nortalis aeg edasi liikunud ning tarkvaraarenduse efektiivsus on paranenud. Seda juba põhjustel, et nüüdseks on olemas vastav kompetents ning oskusteave liigub projektide vahel. Samas ei ole siiani saavutatud sama efektiivsust, mis oli arendades Aranea raamistikul. Praeguseks saab öelda, et keskmine arenduse efektiivsus on Aranea raamistikust 10% aeglasem.

Käesoleva töö autori seisukoht on, et valitud suuna kohta ei saa otse öelda, et tehnoloogia valikul tehti vale otsus. Valiku teostamise hetkel oli tegemist tehnoloogiaga, mis oli kindlasti katsetamist väärt ning eesliidese tehnoloogiate osas oli valik õige. Samas sai katsetamisest midagi enamat ning eesliidese tehnoloogiad on liikunud paljudesse Nortali avaliku sektori projektidesse. Autori seisukoht on, et menetlussüsteemide arendamise osas oleks pidanud vaatama teiste tagaliidese tehnoloogiate suunas.

Tänase seisuga on välja tulnud Angular oma uue versiooniga ning on näha, et uued projektid liiguvad tehnoloogi kasutusele võtmise suunas. Sellise tehnoloogilise muudatuse tulemuseks on aga uus õpikõver. Angular on hea näide sellest kui kiiresti muutub eesliidese tehnoloogiate maailm.

Aeg näitab, mis saab AngularJS'ist kui Angular2+ on oma kanna ära kinnitanud. Suure tõenäosusega kaob mingil hetkel eelmiselt versioonilt Google tugi ning edasi arendamise võtab enda vastutada mõni arendajate kommuun.

Samas on vahepeal välja tulnud erinevaid uusi tehnoloogiaid, mille suunas tuleks pigem eesliidese arenduses vaadata, üheks selliseks on Aurelia raamistik.

Kokkuvõtvalt saab öelda, et mis puudutab arendusefektiivsust siis eesliidese tehnoloogiatega ongi arendus aeglasem, samas saab tänu tehnoloogia suunale luua rikkalikuma kasutajaliidese. Tehnoloogia valimisel tuleb lähtuda, et kes on kasutajad, kui pikk on süsteemi planeeritav eluiga ja mis nõudeid peab loodav tarkvara täitma täna ja ka tulevikus oma eluaja jooksul. Kindlasti ei peaks keskenduma ainult eesliidese tehnoloogiatele ning vaadata tuleb ka tagaliidese tehnoloogiate suunas, viimased ei ole väljasurnud ja arenevad jõudsasti edasi.

Peatükis „

Komponendipõhine tarkvararaamistik“ on väljatoodud tarkvaraarenduse raamistiku kontseptsioon ja lahendus, mis võiks antud töö autori arvates rahuldada avaliku sektori menetlussüsteemi arendamise vajadused efektiivsemalt kui puhtalt eesliidese tehnoloogiad.

5 Komponendipõhine tarkvararaamistik

Turul on erinevaid raamistikke, mis on leidnud tarkvara arendamises laia kasutust. Üldjuhul on tegemist erinevate tehnoloogiate kogumitega, mis täidavad igas tarkvara kihis oma kindlat rolli. Kõikides nendes raamistikes saab üldjuhul luua komponente, olgu selleks komponendiks vormil kuvatav väli koos oma reeglite ja nõuetega, kuvatava blokini koos erinevate väljadega ja oma äriloogikaga. Seda kas ja kui palju luuakse või saab luua komponente, mida saab erinevates kohtades korduvalt kasutada, on juba kinni loodava tarkvara spetsiifikas kui ka mingil määral valitud tehnoloogiates.

Tarkvaraarenduses on alati mõistlik lähtuda komponendipõhisest kontseptsioonist, kus juba loodud funktsionaalsust saab korduvalt kasutada. Selline lähenemine lihtsustab uute funktsionaalsete kuvade loomist ning sellega kiireneb ka arendusprotsess. Loodud komponendid võivad olla täielikult täisväärtuslikud ehk sisult eraldiseivad, mis tähendab seda, et need omavad ka äriloogilist teavet valideerimisreeglitest kuni erinevate andmekogudega suhtlemiseni välja. Komponent võib alla ka korduvakasutatav visuaalne komponent, mis ei tea midagi otseselt ärist, kuid kuvade lihtsustamise eesmärgil on need võetud kasutusele ning neile on antud kindlad omadused.

Peatükis „Arhitektuuri mustrid“ on käsitletud kihilist ja mikroteenuste arhitektuurilist mustrit. Järgnevas peatükis tutvustatakse tehnoloogial Spring MVC [35] loodud komponendipõhist lähenemist toetavat tagaliidese tehnoloogiad kasutavat raamistikku. Antud raamistik peegeldab kihilist arhitektuuri mustrit, mis antud töö autori seisukohast on alternatiiv avaliku sektori menetlussüsteemide arendamiseks.

5.1 Spring MVC komponendipõhine tarkvaraarendusraamistik

Aastal 2012 osales ettevõtte Nortal hankes, kus üks Eesti ministeeriumitest soovis saada kliendiportaali lahendust, mille vahendusel saaks klient esitada taotlusi ning dokumente. Süsteemil peab olema liidestus erinevate andmekogude, registritega ja dokumendihaldusega. Lisaks pidi lahendus toetama versioneerimist ja loodud dokumentide viimist PDF kujule. Nortal oli hankes edukas. Tegemist oli hankega, millel oli fikseeritud skoop ning eelarve. Siiani oli Nortal oma lahendusi loonud Aranea raamistikul, seda raamistikku aga enam antud hankes ei saanud ja ei olnud ka mõistlik kasutada. Seetõttu sai vaadatud alternatiivsete lahenduste suunas nagu Struts [36],

Velocity [37], Vaadin ja Spring MVC. Ükski ei täitnud neist koheselt või siis täielikult soovitud nõudeid nagu ühtne valideerimis- ja äriloogika kontrollide käsitlus ees- ja tagaliideses, komponendipõhist lähenemist, kus komponent on ka kindla funktsionaalsuse kogum jne. Lisaks oli projektis ajaline surve mistõttu ei olnud võimalik erinevaid lahendusi pikalt kaaluda. Seetõttu sai tehtud otsus luua ise tugiraamistik, mis täidab etteantud nõuded. Antud lahenduse kontseptsiooni välja pakkujaks oli antud magistritöö autor ning enamus alglahendusest on kirjutatud tema poolt. Tugiraamistiku loomise algushetkel sai tehtud otsus, et tarkvara alustehnoloogiaks saab olema Spring MVC [38], mis oli tol hetkel ja on siiani väga populaarne. Valitud tehnoloogia peale ehitas magistritöö autor komponendipõhist lähenemist toetava kihi, mille üldine põhimõtte kirjeldab lahti autor alljärgnevalt. Hiljem liikusid teised arendusmeeskonna liikmed ning koos arendati välja vormdokumentide loomise kontseptsioon ja lahendus.

Tänu tehtud otsusele õnnestus tarkvaraprojekt lõpetada õigeaegselt, mis on üldjuhul tarkvaraarenduses haruldane. Sellest lähtuvalt oli näha, et kontseptsioonina komponendipõhine lähenemine tagaliidese tehnoloogiate baasil on tarkvaraarenduses olemas oma koht. Kontseptsiooni toetava lahenduse abil on võimalik luua kiiresti ja avaliku sektori menetlussüsteemi protsesse toetav tarkvaraline lahendus.

Antud tugiraamistiku tuumlahendus sai magistritöö autori poolt liigutatud ministeeriumi kliendiportaali lahendusest välja ning see on tõstetud Nortali avalikku koodihoidlasse, mis asub aadressil https://github.com/nortal/spring-mvc-component-web.

Järgnevates peatükkides kirjeldatakse antud raamistiku üldist kontseptsiooni. Kuna raamistik on mahukas ning antud töö suunaks on alternatiivse kontseptsiooni välja pakkumine, siis ei minda raamistiku kirjeldamises väga detailseks. Samas teatud detailsus siiski jääb.

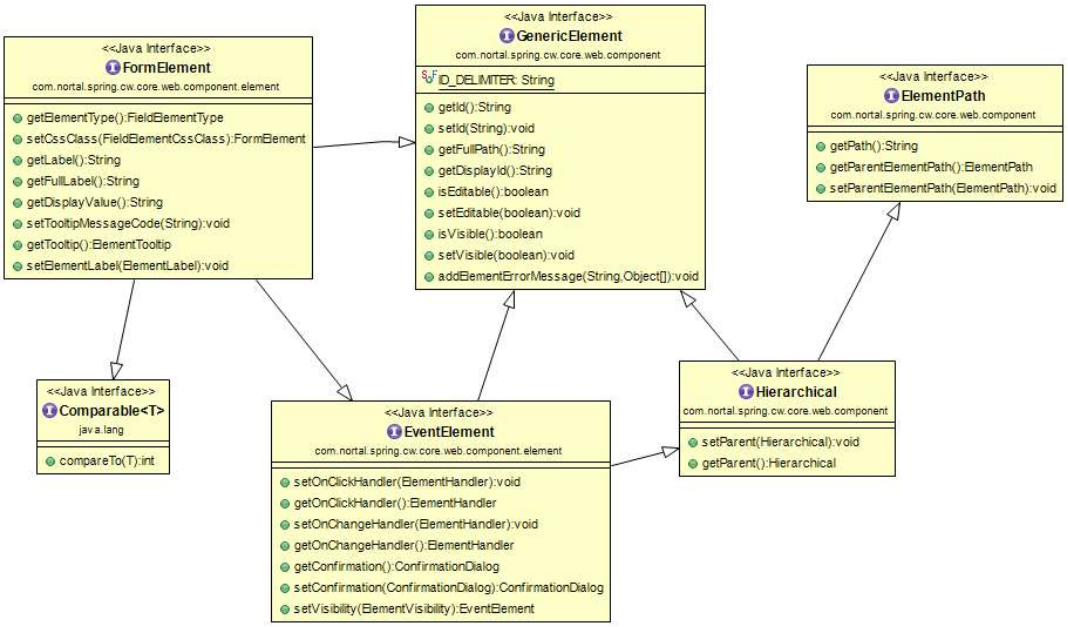
5.1.1 Raamistiku üldine kontseptsioon

Arendatud komponendipõhine tarkvararaamistik kujutab endas Spring MVC raamistiku peale arendatud täiendavat abstraktsiooni. Spring MVC üheks oluliseks eesmärgiks on reageerida kasutajapoolsetele päringutele, suunata need õigesse teenindavasse komponenti ning seejärel genereerida kasutaja kuva. Praeguseks on loodud raamistiku poolt toetatud JSP põhine kuvade loomine, täiendavalt on arendamisel Thymeleaf tugi.

Lisaks on laiendatud andmete oleku mõistet, mis tähendab seda, et kasutaja sessiooni sees hoitakse kuvatud lehe olekut koos kasutaja andmetega. Selline lahendus vähendab andmete valideerimise ja ajutise hoidmise keerukust, lisaks koormab see vähem andmekihti, kuna varem päritud andmed on talletatud kasutaja sessiooni korduv kasutamiseks. Nii kiireneb uue funktsionaalsuse loomine, kuna näiteks väheneb keerukus andmete terviklikkuse kontrollimiseks.

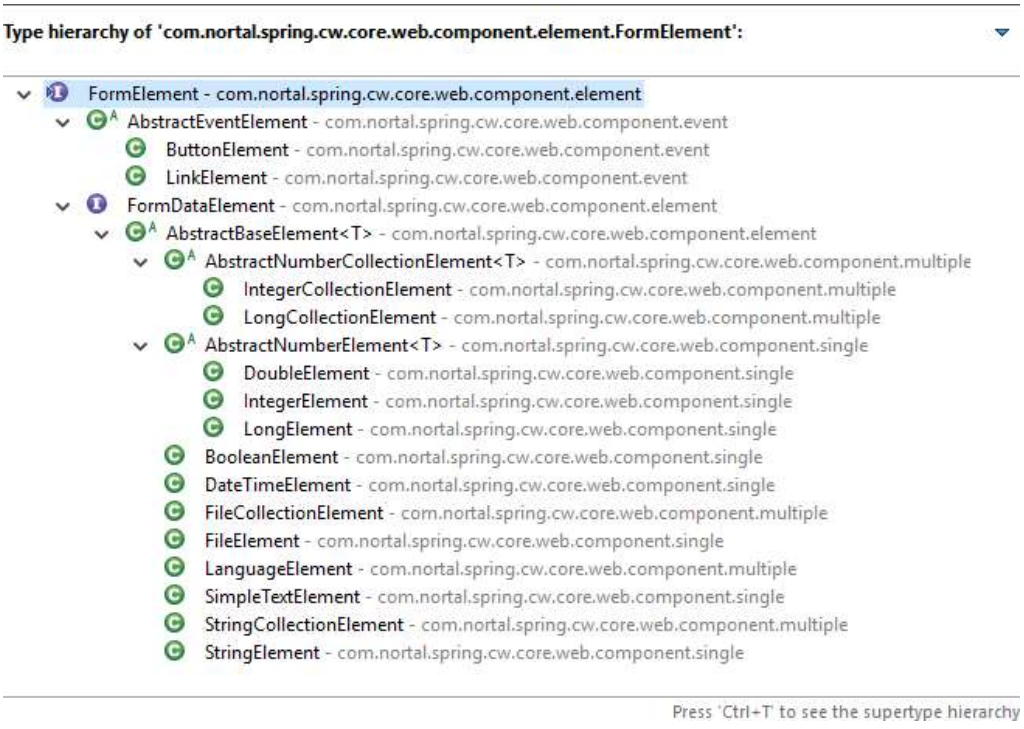
Fundamentaalne lähenemine antud raamistikus on see, et kõik esitluskihiga seotud objektid on sisuliselt elemendid. Kasutuses on liides *GenericElement*, mis on üldine veebi komponentide liides ning antud liidest implementeerivad kõik vormi elemendid ja komponendid. Sisuliselt on antud juhul tegu veebis kuvatavate objektidega ning neil on sarnased põhiomadused. Igal veebiobjektil on *path* ehk rada, tema leidmiseks, *label* ehk tema nimi, mis vajadusel kuvatakse välja, jne. Igale objektile on võimalik määrata tema omanik ehk *parent*, mis peab olema komponendi tüüpi. Element ei saa omada teist elementi, vaid elemente saavad omada vaid komponendid ning komplektsed komponendid saavad omada teisi komponente. See on oluline kitsendus ja sellisel viisil on kogu arhitektuur paremini mõistetav.

Vormelementide liideseks on *FormElement* (Joonis 10). Liides laiendab üldist elementide liidest *GenericElement*, kus kirjeldatakse ära iga elemendi ja komponendi põhiomadused, milleks on teda idendifitseeriv tunnus, rada antud elemendi või komponendi leidmiseks elementide heirarhias, kas element on muudetav ja kas see on nähtav. Lisaks laiendab vormielementide liidest sündmuste liides *EventElement*, mille abil saab vormielemendile anda erinevate sündmustega seotud teavet. Näiteks mis juhtub siis kui sisendväli muutub või mis juhtub siis kui väljale või lingile klikitakse.



Joonis 10. Vormelemendi liidese FormElement klassdiagramm

Joonis 11 on välja toodud vormelemendi liidese kasutuskohad hierarhilises vaates. Antud vaate toob välja hetkeks realiseeritud erinevate vormielementide liigid.



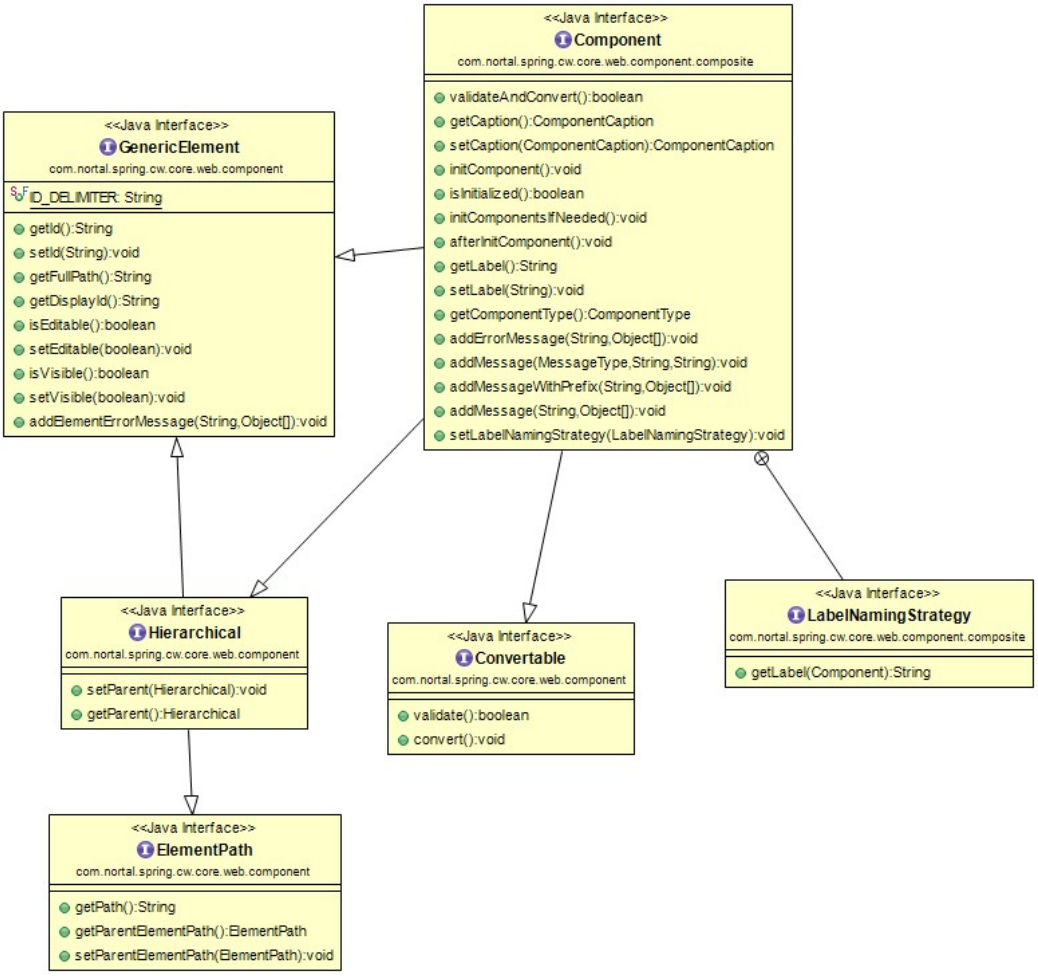
Joonis 11. Vormelementide liidese *FormElement* kasutus-hierarhia

Vormielementide tabelis (Tabel 1) on ära kirjeldatud üldised vormielemendi koos nendega kaasas käivate üldiste omadustega.

Tabel 1. Üldised vormielemendid

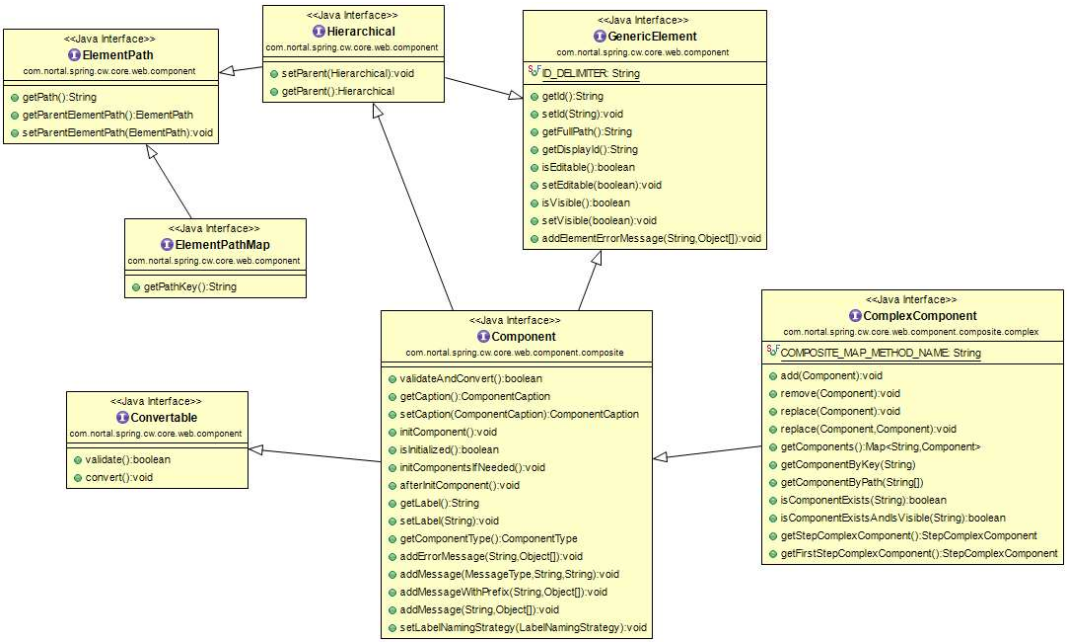
Lihtkomponendid	Mitmikvalikud	Tervikkomponendid	Täiendavalt
Teksti Number • Täisarv • Ujukomaarv Kuupäev • Kuupäev • Kuupäev/Kell • Kell Ruutkastike Radio Pealkiri Sündmused: • Link • Nupp	Tekst Number • Täisarv • Ujukomaarv	Protsessi samm Nimekiri • Sorteerimine • Fltreeri • Muuda/kustuta + ise tehtud • Uue rea lisamine • Paginatsioon Vorm Modaalaknad: • Dialoog • Hüpikaken	Valmiskomponentide kasutus Saab kasutada abiinfot nii pealkirja kui elemendi juures Listi kõiki osi saab üle kirjutada: • Üksik veerg • Read: päise, sisu, jalus Teadete tasemed: • INFO • OK • WARNING • ERROR

Komponentide liides *Component* (Joonis 12) laiendab samuti liidest *GenericElement*, kuid komponentidel puudub teadmus sündmuste osas, seetõttu ei laiendata sündmuste liidest. Komponendi põhiomaduseks on eristada vormielemente teistest komponentidest. Igal komponendil on oma nimetus, mis kuvatakse vajadusel välja kasutajaliideses. Igasugune tekstide leidmine on võtmepõhine, mis tähendab seda, et tekstid on elementidest eraldatud ja neid hoitakse mujal võtme ja väärtuse kujul. Õige teksti tuvastamine toimub komponendi raja või unikaalse tunnuse alusel. Vajadusel saab komponentide juures anda kaasa täiendava strateegia õige teksti leidmiseks. Selline lähenemine võimaldab kasutusele võtta mitme keelsuse.



Joonis 12. Komponentide liidese *Component* klassdiagramm

Komponentide üldine liides Component ja vormielementide liides FormElement ei tea otseselt midagi teistest komponentidest. Ainuke info teise komponendi kohta on asumine hierarhias. Kasutusele on võetud komplektse komponendi mõiste ehk liides ComplexComponent (Joonis 13). Komplektse komponendi liidese üldiseks eesmärgiks on omada infot teiste komponentide kohta.



Joonis 13. Komplektse komponentide liidese *ComplexComponent* klassdiagramm

Komponendid on sisuliselt ärilised objektid, mis omavad endas ka ärilist infot. Ärilise info all on mõeldud kust andmed tulevad, mida andmetega teha saab ja missugusel kujul andmed hiljem maha salvestatakse või vajadusel isegi kuhu salvestatakse. Selline kompleksne teadmine ärist kui ka oma vaate omamine võimaldab neid lihtsal viisil kasutada korduvalt erinevates kuvakomponentides.

Komponentide hierarhia joonisel (Joonis 14) on kirjeldatud hierarhilises vaates liidese kasutuskohad. Antud vaatest lähtub hetkeks realiseeritud erinevate komponentide liigid.

Joonis 14. Komponentide liidese *Component* kasutus-hierarhia

5.1.2 Komponent Java koodis

Komponendi kirjeldamine Java koodis on üpris paindlik. Iga komponendi juures tuleb kirjeldada ära komponendi tüüp, anda talle tunnus ning määrata ära komponendiga seotud mudelobjekt ehk objekt, mis on aluseks vormi elementide kuvamisel. Komponendi eesmärk põhineb selles, et see viib kokku mudelobjekti ja vormiväljad, mida kuvatakse välja loodavas kuvas. Uue elemendi lisamiseks komponenti, mis on aluseks ka elemendi vormil välja kuvamiseks, tuleb kasutada meetodit *add*, mille sisendargumendiks on konkreetse välja nimi mudelobjektist. Kui see saab tehtud rakendub automaatne nn välja juhtimise mehhanism, kus tuvastatakse, mis väljaga on

tegemist ja mis kujul andmeid sinna sisestada saab. Võimalik on juhtida ka välja valideerimisreegleid läbi liidese *EventElement* pakutavate sündmuste toe.

```
CwFormComponent<ObjectTest> formComp = new CwFormComponent<>(VORM,
ObjectTest.class);
formComp.setEditable(true);

DateTimeElement dateTimeField = formComp.add("dateTimeField");
dateTimeField.setFormat(DateTimeElementFormat.DATETIME);

DateTimeElement dateField = formComp.add("dateField");
dateField.setFormat(DateTimeElementFormat.DATE);

DateTimeElement timeField = formComp.add("timeField");
timeField.setFormat(DateTimeElementFormat.TIME);

StringCollectionElement list = formComp.add("list");
list.setMultiValue(new MultiValueHolder(SelectElementType.MULTISELECT,
getTestMultiselectData()));

formComp.add("decimalField");
formComp.add("integerField");
formComp.add("longField");

formComp.add("textField");
formComp.add("requiredTextField");

StringElement longTextField = formComp.add("longTextField");
longTextField.initRichText();

formComp.setData(getData());
```

Joonis 15. Komponendi loomise näide Java koodis

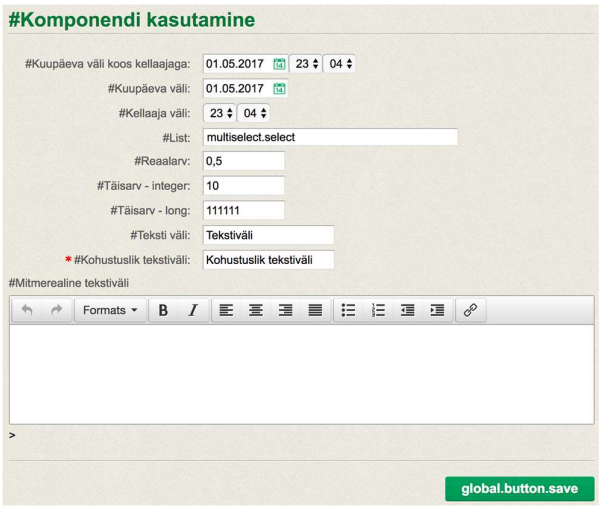
Joonis 15 kirjeldab ühe näidiskomponendi loomist, kus esimene rida ütleb, mis on komponendi tunnuseks ning kasutatav andmeobjekt ehk objekti signatuur. Järgmine rida ütleb, kas komponendis kuvatavad väljad on muudetavad. Keskmine osa defineerib ära kuvatavad väljad. Kõige viimane rida initsialiseerib vormiandmed. See on üks üldisemaid näiteid, kuidas toimub uue komponendi loomine. Lisana (Lisa 7) on välja toodud näidis komponendi terve Java kood.

5.1.3 Esituskihis kasutatav JSP

Ajalooliselt sai valitud JSP esitluskihi genereerimistehnoloogiana see tõttu, et valiku tegemise hetkel ei olnud näha alternatiive. Kuna ajaline surve oli peal ning arendajatel oli olemas head teadmised JSP vallas siis sai otsustatud antud tehnoloogia kasuks.

Nüüdseks on ajad muutunud ning on väga hea alternatiiv Thymeleaf'i kujul, mida tuleks tulevikku vaadates tõsiselt kaaluda. Thymeleaf'ist tuleb juttu järgmises peatükis.

JSP sisuliseks eesmärgiks on viia kokku vorming kuvatavate andmetega. Andmed antakse ette Java klassina defineeritud komponendi poolt. JSP abil kirjeldatakse ära visuaalne pool see tähendab paigutus, kuidas kasutajale kuvatakse konkreetse vormi andmeid. Esitluskihi genereerimise protsess on viidud lihtsale kujule ning üldine keerukus on peidetud koduvkasutavatesse JSP komponentidesse. Eelmises punktis toodud Java näitega seotud JSP faili definitsioon on välja toodud peatüki allpool oleval joonisel (Joonis 17). Joonisel esimesel real antakse ette vormi tunnus, mis sai ette antud komponendi defineerimisel. Seejärel algab puhas HTML vorming, millega pannakse paika üldine lehe ülesehitus, kus ja mis kujul kuvatavad elemendid asetsevad. Seejärel HTMLi vormingu sees kutsutakse välja spetsiifilised JSP elemendid andes ette Java komponendis kirjeldatud vormi elemendi välja nime. Sisuliselt sellisel viisil toimub esitluskihi kirjeldamine. Kõige lõpus viimane rida lisab lehe lõppu tegevuse nupud nagu salvestamine. Lehe lõpus olevad tegevuse nupud on komponendi enda juurest juhitavad. Antud testkomponendi tulemus on välja toodud allpool oleval joonisel (Joonis 16).



Joonis 16. Esituskihis kasutatava JSP baasil genereeritud vorm

```
<tag:componentItem item="vorm">
	<table class="form">
		<tbody>
			<tr>
				<tag:formInput element="dateTimeField" />
			</tr>
			<tr>
				<tag:formInput element="dateField" />
			</tr>
			<tr>
				<tag:formInput element="timeField" />
			</tr>
			<tr>
				<tag:formInput element="list" />
			</tr>
			<tr>
				<tag:formInput element="decimalField" />
			</tr>
			<tr>
				<tag:formInput element="integerField" />
			</tr>
			<tr>
				<tag:formInput element="longField" />
			</tr>
			<tr>
				<tag:formInput element="textField" />
			</tr>
			<tr>
				<tag:formInput element="requiredTextField"
/>
			</tr>
			<tr>
				<tag:formTextareaLarge
element="longTextField" />
			</tr>
		</tbody>
	</table>
</tag:componentItem>

<tag:pageActions />
```

Joonis 17. Komponendi loomise näide JSP näitel

5.1.4 Esituskihis kasutatav Thymeleaf

Thymeleaf'i tööpõhimõte on sarnane JSP omaga, kuid vundamendi poolepealt on see siiski täielikult erinev. Tegemist on Java XML/XHTML/HTML5 mallide põhjal kuvade genereerimistööriistaga. Võrreldes JSP'ga on siin tegemist puhtama HTML'iga ning nn seoste tekitamine Java komponendiga on atribuutide põhine. Selline lähenemine muudab HTMLi puhtamaks ning juba prototüübis loodud tulemi saaks kopeerida

enamvähem üks-ühena tagaliidesesse. See muudab arendusprotsessi veelgi tõhusamaks, kuna prototüübis saadud sisendit ei pea enam viima teisele kujule vaid selle saab otse kasutusele võtta.

Tegemist on väga tugeva alternatiiviga JSP'le ning kindlasti tuleks tuleviku tagaliidese lahendustes kaaluda antud tehnoloogia kasutusele võtmist.

Spring MVC komponendipõhisel tarkvaraarendusraamistikul on Thymeleaf tugi olemas, kuid see ei ole elementide osas jõudnud väga kaugele ning täisväärtusliku kasutuselevõtmiseks on veel palju teha.

Peegeldades Java komponendina loodud test komponenti näeb Thymeleaf'i mall välja sarnane JSPga, selle vahega et JSP definitsioonid on asendunud Thymeleaf'i atribuutide definitsioonidega (Joonis 18).

Eelmise aasta seisuga on Thymeleaf'ist väljas juba kolmas väljalase ning antud tehnoloogia on leidnud juba laialdast kasutamist.

```
<table class="form">
	<tbody>
		<tr>
			<td th:dateTime="${dateTimeField}">Kuupäeva ja
kellaaja väli</td>
	</tr>
		<tr>
			<td th:time="${timeField}">Kellaaja väli</td>
		</tr>
		<tr>
			<td th:list="${list}">Nimekirja väli</td>
		</tr>
		<tr>
			<td th:decimal="${decimalField}">Ujukomanumbri
väli</td>
		</tr>
		<tr>
		<td th:decimal="${numberlField}">Numbri väli</td>
		</tr>
		<tr>
			<td th:decimal="${longField}">Täisarvu väli</td>
		</tr>
		<tr>
			<td th:text="${equiredTextField}">Teksti väli</td>
		</tr>
		<tr>
			<td th:number="${longTextFiel}d">Teksti väli</td>
		</tr>
	</tbody>
</table>
```

Joonis 18. Komponendi loomise näide Thymeleaf'i näitel

5.2 Kokkuvõtvalt

Tegemist oli loodud raamistiku lühikese tutvustusega, mis andis siiski ülevaate üldistest raamistiku tööpõhimõtetest ning eesmärkidest. Selleks, et anda pikem ja detailsem ülevaade tuleks arvatavasti kirjutada eraldiseisev magistritöö.

Käesoleva peatüki üldine eesmärk oli tutvustada kontseptsiooni, mis kindlasti parandaks tarkvaraaraarenduse efektiivust keskendudes tagaliidese tehnoloogiatele. Raamistiku põhjaks ei pea olema Spring MVC, selleks võib olla ka miski muu, kuid oluline on selle tehnoloogia kasutusele võtmisel keskenduda komponendipõhisele lähenemisele ja loodu korduva kasutamise põhimõtetele. Lähtudes sellest on võimalik arendada efektiivsemalt ning arendusprotsess on paremini hoomatav. Erinevad ärireeglite kogumid on kirjeldatud neid kasutavate komponentide juures ning need reeglid on

korduvkasutatavad nii ees- ja tagaliideses. Komponentide loomine peab sellist ühtset reeglite kasutamisviisi võimaldama, vastasel juhul tuleks ikkagi dubleerida teatud kontrolle ees- ja tagaliideses. Arendatud tugiraamistiku puhul teostatakse kontrollid kullisside taguselt vastu serveris kirjeldatud reeglitele. Vea esinemise korral kuvatakse viga välja konkreetse välja juures või vajadusel kuvatakse välja üldine veateade.

Arendatud tugiraamistik areneb edasi ning antud töö autorile on teada, et üks asutus plaanib võtta raamistiku kasutusele ning seda koos Thymeleaf'i pakutava võimekusega. Selle tegevuse käigus on koostööna plaanis aidata ja luua täiendav raamistiku dokumentatsioon.

Antud töö autori seisukoht on, et loodud tarkvaraline lahendus ei ole küll veel täisuslik, kuid see toob välja mõned põhitõed ja suunad, millega tuleks arvestada kui kasutada tagaliidese tehnoloogiaid. Üheks põhisuunaks peab olema see, et see mis on juba ükskord loodud ei tohiks saada uuesti loodud mõnes teises kohas vaid juba loodut tuleb tervikuna uuesti kasutada.

6 Avaliku sektori tarkvaraplatvorm

Käesoleva töö autori arvates tundub olevat avalikus sektoris kui identiteedikriis. Mingil määral võib olla selles süüdi see, et kaua on oldud kinni vanas tehnoloogias, olgu selleks kasvõi näiteks Aranea raamistik. Liikudes nüüd uute tehnoloogiate suunas tehti mõnedes ministeeriumites ja riigiasutustes täielik kannapööre ja fookus liikus eesliidese tehnoloogiatele. Oluline on märkida, et tellija kirjutab juba hanketingimustesse sisse, mis tehnoloogiaid ta soovib, et arenduspartner tarkvaraarenduses kasutaks. Selline otsus tehnoloogia suuna vahetamiseks on autori arvates olnud liiga rutakas, kaalumata on jäänud alternatiivsed lahendused ning see, kas valitud tehnoloogiad sobivad püstitatud tarkvaralise probleemi lahendamiseks.

Tänaseks päevaks on seis selline, et kõnealustes asutustes minnakse edasi AngularJS'i raamistikuga. Tõsiasi on ka see, et väljas on juba Angular2+. Oluline tähelepanek on ka see, et eesliidese tehnoloogiad arenevad kiiresti, vanad kaovad, uued tulevad peale. Käesoleva töö autor näeb siin tarkvaraprojektidele täiendavat riski. Näiteks kui raamistiku arendaja otsustab AngularJS toe lõpetada siis tähendab see seda, et seni kuni kehtib garantii peab tuge edasi pakkuma arenduspartner. Tegemist on arenduspartnerile suunatud riskiga, millega peaks arenduspartner hankesse sisenedes arvestama.

Võrdluseks saab tuua näitena ühe ministeeriumi kliendiportaali, mis tarkvararaamistik on tagaliidese tehnoloogiatel baseeruv ning ühe riigiasutuse, kus mindi üle uutele eesliidese tehnoloogiatele. Antud lahendustes on rakendatud vastavalt kihilist ja mikroteenuste arhitektuuri mustreid. Ministeeriumi kliendiportaalis arendati ühe arendaja poolt valmis üks taotluse sisestamise vorm keskmiselt nädalaga. Riigiasutuses, kus kasutatakse uusi eesliidese tehnoloogiaid võtab see arendus aega keskmiselt kolm korda rohkem. Tõsi, üks-ühele neid projekte ei saa võtta, kuna eesmärk mida saavutatakse on erinev. Samas oli antud töö autor mõlemas projektis juhtivtarkvaraarendaja rollis. Seetõttu saab autor kindlasti öelda, et valides menetlussüsteemiks mõne tagaliidese tehnoloogiate kogumi ja ka vastavalt kihilise arhitektuuri mustri oleks arendusprotsess kindlasti efektiivsem ja klient saaks oma lahendused kätte kiiremini ja odavamalt.

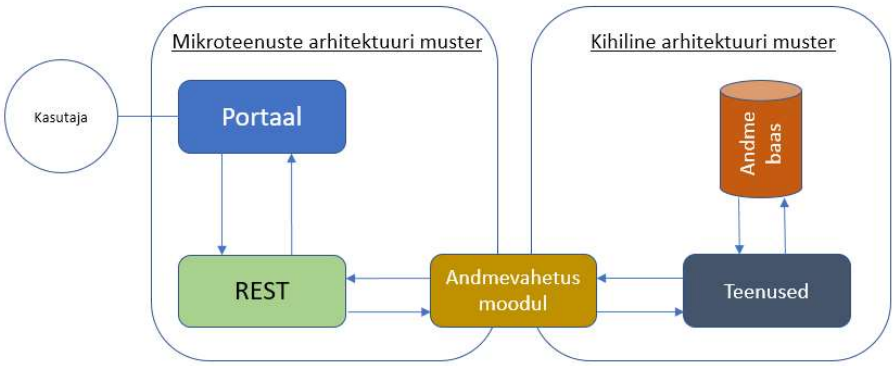
Mikroteenuste arhitektuuri muster, mis mõnes riigiasutuses on leidnud rakendust, lisab projektile juurde täiendava keerukuse, mis antud projektide puhul ei ole antud töö autori

arvates vajalik. Kõnealuses riigiasutuses on võetud teemaks lahendus, kus soovitakse erinevate moodulite andmebaasid omavahel siduda, et saada mööda SOA keerukustest ja sellega kaasnevatest murekohtadest. Kui see plaan jõustub siis sisuliselt loobutakse mikroteenuste arhitektuuri mustrist ning erinevad moodulid hakkavad üksteisega suhtlema andmebaasi kihi vahendusel.

6.1 Sobiva arhitektuuri mustri valimine

Sobiva arhitektuuri mustri valimisel tuleb läheneda sellest, et mis probleemi ja kelle jaoks tarkvara arendades lahendatakse. Kui tegemist on menetlussüsteemi ja sellega kaasnevate protsessidega siis tuleks vaadata kihilise arhitektuuri mustri suunas. Menetlustoimingud on üldjoontest sammu põhised, mis tähendab seda, et liigutakse sammust sammu. Sammud sisaldavad üldjuhul standardseid tegevusi ning seetõttu erinevatel menetluse protsessidel on palju ühisosa.

Kui meil on aga tegemist lahendusega, mille kasutajaskond on väljaspool asutust näiteks kliendid ning kasutajaliides peab olema toetatud erinevatel seadmetel, nõutud on modulaarsus, siis tuleks vaadata mikroteenuste arhitektuuri mustri suunas. Sellised süsteemid on kiiresti muutuvad ning peavad tihti kaasas käima kliendi soovidega. Süsteemid peavad olema visuaalselt ilusad, interaktiivsed ning paindlikud. Kihilise arhitektuuri mustri korral ja keskendudes tagaliidese tehnoloogiatele võib kõikide selliste nõuete täitmine osutuda keeruliseks. See aga ei tähenda seda, et kõik süsteemid peaksid olema üles ehitatud mikroteenuste arhitektuuri mustril. Pigem tähendab see seda, et tuleb tekitada vahelüli, mis võimaldab näiteks kahel erineva mustriga lahendusel omavahel suhelda. Selleks sobib täiendava mooduli tekitamine, mis on avaliku kasutajaliidese vahendusel nähtav. Vahemoodul kas saadab päringuid edasi menetlussüsteemi või on võimeline kohe otse ära kasutama menetlussüsteemi ärikihi tarbeks juba loodud teenuseid. Joonis 19 kujutab üldist lahendust kahe erineva tarkvara arhitektuuri mustriga lahenduse ühendamiseks. Selline lahendus on ka turvalisem, kuna siin saab rakendada täiendavaid teenuste väljakutsete kontrolle.



Joonis 19. Mikroteenuste ja kihilise arhitektuuri mustri tarkvaraliste lahenduste ühendamine

Alternatiiviks on ka kasutada ESB lahendust, kuid ainult kahe süsteemi integreerimiseks võib see olla ülepingutatud. Sisuliselt on soovitus see, et rakendatud peaks olema nn hübriid-mustriline lähenemine. Selline lähenemine lahendaks ära põhiprobleemid nagu menetlussüsteemi arendusefektiivsus ning mõned mikroteenuste arhitektuurilise mustriga kaasnevad tehnilised eripärad. Antud eristuseks positiivseks küljeks on ka see, et arendusmeeskond võib olla jaotunud ja spetsialiseerunud. Ühed arendavad eesliides tehnoloogiatel üles ehitatud lahendust ja teised tagaliidese tehnoloogiatel.

7 Kokkuvõte

Käesolevas töös võrdles autor erinevaid ees- ja tagaliidese tehnoloogilisi suundasid, analüüsides tehnoloogilise radari trende ning rahvusvahelise turu töökuulutusi läbi tehnoloogiliste märksõnade. Trendide analüüsimise käigus selgus, et üldine trend on liikumine eesliidese tehnoloogiate suunas samas töökuulutustest lähtuvalt paistab olevat trend raugemas.

Kihilise ja mikroteenuste arhitektuuri mustrite võrdlemisel selgus, et mikroteenuste arhitektuuriline muster on tunduvalt keerulisem oma rohkemate kihtide tõttu võrreldes kihilise arhitektuuriga. Keerukuse tõid välja ka ettevõtte Nortal arendusmeeskonnad, kes olid oma projektidega liikunud eesliidese tehnoloogiate suunas. Nortali sisemises uuringus toodi välja ka olulisena asjaolu, et teatud lahendusi ei olegi praegusel kujul mikroteenuste arhitektuuri mustriga lahendustes võimalik luua nagu näiteks tabelite sorteerimist ja filtreerimist. Lisaks selgus, et juhtivarendajad ei näe võimalust kunagi saavutada tarkvaraarenduses sama head efektiivsus nagu oli tagaliidese tehnoloogia Aranea raamistiku kasutamisel. Nortali uuringu läbiviimise ajahetkest kuni praeguseni on efektiivsus küll paranenud, kuid see on siiski 10% kehvem kui varasemate tehnoloogiatega.

Töö üheks osaks oli magistritöö autori poolt loodud komponendipõhise tarkvaraarenduse raamistiku kontseptsioon ja lahendus. Raamistiku arendamisel on rakendatud tarkvara kihilise arhitektuuri mustrit ning tegemist on tagaliidese tehnoloogiatel baseeruva lahendusega. Autor tutvustab töös üldiseid raamisiku põhimõtteid ning toob välja selle plussid ja miinused võrreldes mikroteenuste arhitektuuri mustriga kasutavate lahendustega.

Analüüsides erinevaid trende, arhitektuurilisi mustreid ning Nortali uuringu tulemusi ja lähtudes pikaajalisest arendaja kogemusest ei ole antud töö autor eesliidese trendi suundadega täielikult nõus. Autori seisukoht on, et kõiki tarkvaralisi lahendusi ei tohi lahendada eesliidese tehnoloogiate abil. Selle asemel tuleb teatud lahenduste puhul kaaluda ka tagaliidese tehnoloogiaid. Autori poolt loodud komponendipõhine

tarkvaraarendusraamistik näitab ilmekalt kuidas vähem keerukama arhitektuuri mustriga ja tagaliidese tehnoloogiatel baseeruva raamistikuga on võimalik luua samuti täiesti konkurentsivõimeline tarkvaraline lahendus. Tagaliidese tehnoloogiatel baseeruvad lahendused on mõeldud üldjuhul töötama nii, et peale igat serveri poole pöördumist genereeritakse uus kasutajaliidese kuva. Nagu antud töö autor tõi välja ei pea see nii olema ning ka tagaliidese lahendustes saab kasutada andmete liigutamiseks kulisside tagust pärimise võimekust.

Antud töö autor pakub avaliku sektori tarkvaraarhitektuurina välja hübriid-lahenduse, mille abil eraldada üksteisest eesliidese tehnoloogiatel ja tagaliidese tehnoloogiatel toimiv lahendus. Võimaldades sellisel viisil rakendada ees- ja tagaliidese tehnoloogiaid sobivamal viisil. Tagaliidese tehnoloogiana pakub autor välja enda poolt loodud komponendipõhise tarkvaralise lahenduse kontseptsiooni, mis on autori arvates lihtsam ja menetlussüsteemi arendamiseks sobivam ning millest võiks saada avaliku sektori komponendipõhine tarkvaraarendusplatvorm. Antud seisukohta toetab teostatud arhitektuurimustrite võrdlus, Nortali uuringus välja toodud murekohad ning antud raamistikul arendatud ühe ministeeriumi kliendiportaali õnnestumine.

Kasutatud kirjandus

[1] T. Erl, Service-Oriented Arhitecture, New Jersey: Prentice Hall PTR, 2004.

[2] M. D. Hansen, SOA Using JAVA Web Services, Indiana: Pearson Education, Inc, 2009.

[3] G. Warin, Mastering Spring MVC 4, Birmingham: Packt Publishing Ltd., 2015.

[4] „JavaScript,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/JavaScript. [Kasutatud 29 04 2017].

[5] L. Richardson ja M. Amundsen, RESTful Web APIs, Sebastopol: O'Reilly Media, Inc.,, 2013.

[6] „JavaServer Pages,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/JavaServer_Pages. [Kasutatud 10 04 2017].

[7] MTÜ Arvutikaitse, „Veebilehitseja,“ [Võrgumaterjal]. Available: http://www.arvutikaitse.ee/arvutikaitse-algtoed/veebilehitseja/. [Kasutatud 19 04 2017].

[8] E. K. Instituut, „Topoloogia,“ [Võrgumaterjal]. Available: http://www.eki.ee/dict/qs/index.cgi?Q=topoloogia&F=M. [Kasutatud 30 04 2017].

[9] „Application programming interface,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Application_programming_interface. [Kasutatud 29 04 2017].

[10] „Tarkvara testimine,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/Tarkvara_testimine. [Kasutatud 29 04 2017].

[11] „Java,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/Java. [Kasutatud 29 04 2017].

[12] „PHP,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/PHP. [Kasutatud 29 04 2017].

[13] „C Sharp (programming language),“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/C_Sharp_(programming_language) . [Kasutatud 29 04 2017].

[14] „Enterprise service bus,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Enterprise_service_bus. [Kasutatud 29 04 2017].

[15] A. Osman, JavaScript Design Patterns, Sebastopol: O'Reilly Media, Inc, 2012.

[16] „HTML,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/HTML. [Kasutatud 29 04 2017].

[17] „CSS,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/CSS. [Kasutatud 29 04 2017].

[18] „Teek,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/Teek. [Kasutatud 01 05 2017].

[19] „Portable Document Format,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/Portable_Document_Format. [Kasutatud 02 05 2017].

[20] „PostScript,“ [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/PostScript. [Kasutatud 02 05 2017].

[21] „Facebook,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Facebook . [Kasutatud 31 03 2017].

[22] „React (JavaScript library),“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/React_(JavaScript_library). [Kasutatud 31 03 2017].

[23] „Redux (JavaScript library),“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Redux_(JavaScript_library) . [Kasutatud 31 03 2017].

[24] „Google,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Google. [Kasutatud 31 03 2017].

[25] „AngularJS,“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/AngularJS. [Kasutatud 31 03 2017].

[26] „Angular (application platform),“ [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Angular_(application_platform). [Kasutatud 31 03 2017].

[27] „Vue.js,“ Evan You, [Võrgumaterjal]. Available: https://vuejs.org/. [Kasutatud 31 03 2017].

[28] „Aurelia,“ Blue Spire, [Võrgumaterjal]. Available: http://aurelia.io/ . [Kasutatud 31 03 2017].

[29] „ThoughtWorks,“ ThoughtWorks, Inc., [Võrgumaterjal]. Available: https://www.thoughtworks.com/. [Kasutatud 31 03 2017].

[30] „TechRadar,“ Nortal AS, [Võrgumaterjal]. Available: http://blog.nortal.com/radar/index.html. [Kasutatud 31 03 2017].

[31] M. Richards, Software Architecture. Understanding Common Architecture, Sebastopol: O’Reilly Media, Inc., 2015.

[32] R. C. Martin, Agile Software Development. Principles, Patterns, and Practices, Upper Saddle River: Pearson, Education, Inc. Alan R Apt, 2003.

[33] „Araneaframework,“ Nortal AS, [Võrgumaterjal]. Available: https://nortal.github.io/araneaframework/. [Kasutatud 30 04 2017].

[34] „Vaadin,“ Vaadin Ltd, [Võrgumaterjal]. Available: https://vaadin.com/framework . [Kasutatud 30 04 2017].

[35] „Web MVC framework,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html. [Kasutatud 29 04 2017].

[36] „Apache Struts,“ The Apache Software Foundation, [Võrgumaterjal]. Available: https://struts.apache.org/ . [Kasutatud 29 04 2017].

[37] „Apache Velocity Project,“ The Apache Software Foundation, [Võrgumaterjal]. Available: https://velocity.apache.org/. [Kasutatud 29 04 2017].

[38] C. Walls, Spring in Action, New York: Manning Publications Co, 2015.

[39] „Thoughtworks technology radar - Language and frameworks,“ [Võrgumaterjal]. Available: https://www.thoughtworks.com/radar/languages-and-frameworks. [Kasutatud 31 03 2017].

[40] „java, php, c#, and javascript Job Trends,“ [Võrgumaterjal]. Available: https://www.indeed.com/jobtrends/q-java-q-php-q-c%23-q-javascript.html. [Kasutatud 24 03 2017].

[41] „soa and restful Job Trends,“ [Võrgumaterjal]. Available: https://www.indeed.com/jobtrends/q-soa-q-restful.html.

[42] Lõputöö koostamise ja vormistamise juhend, Tallinn: Tallinna Tehnikaülikool, 2016.

Lisa 1 – ThoughtWorks arenduskeelde ja raamistike radar

Radar seisuga 31.03.2017 [39]

ADOPT
78. Ember.js
79. Python 3
80. ReactiveX
81. Redux

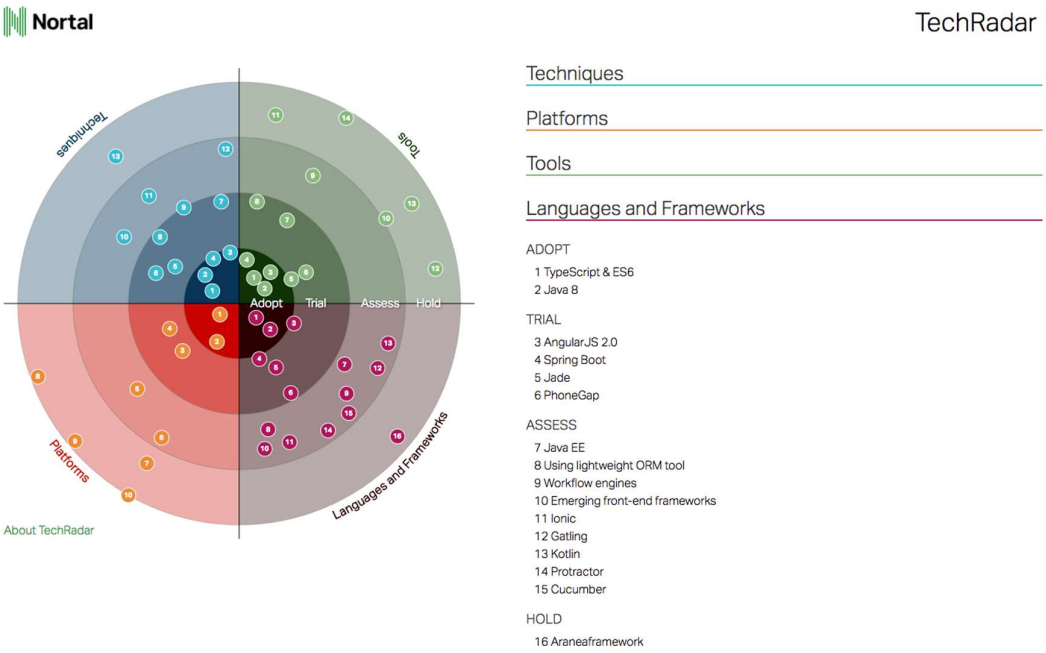
TRIAL
82. Avro new
83. Elixir
84. Enzyme
85. Hangfire new
86. Nightwatch new
87. Phoenix
88. Quick and Nimble
89. Vue.js

ASSESS
90. Angular 2 new
91. Caffe new
92. DeepLearning.scala new
93. ECMAScript 2017
94. Instana new
95. JuMP
96. Keras new
97. Knet.jl new
98. Kotlin new
99. Physical Web
100. PostCSS new
101. Spring Cloud new
102. Three.js
103. WebRTC

HOLD
104. AngularJS

Lisa 2 – Nortali arenduskeelde ja raamistike radar

Radar seisuga 31.03.2017 [30]

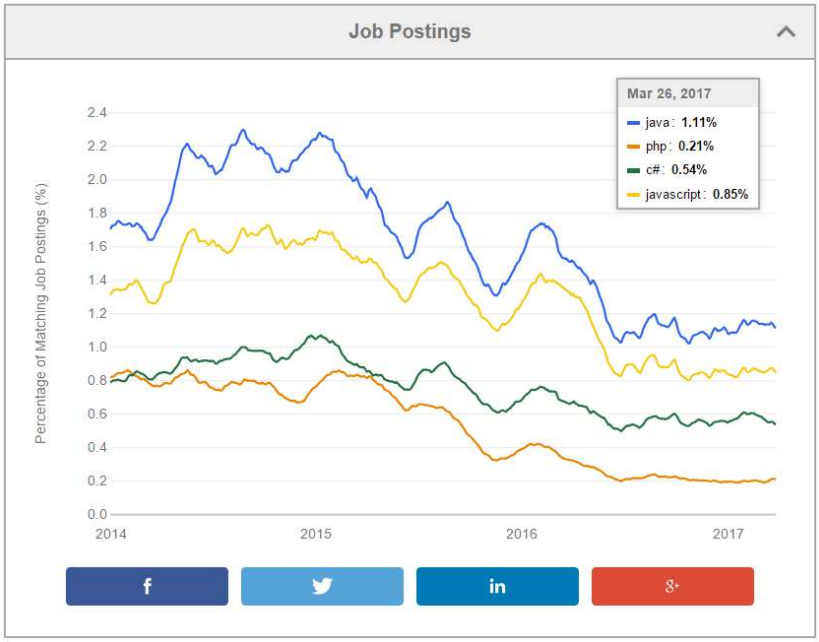


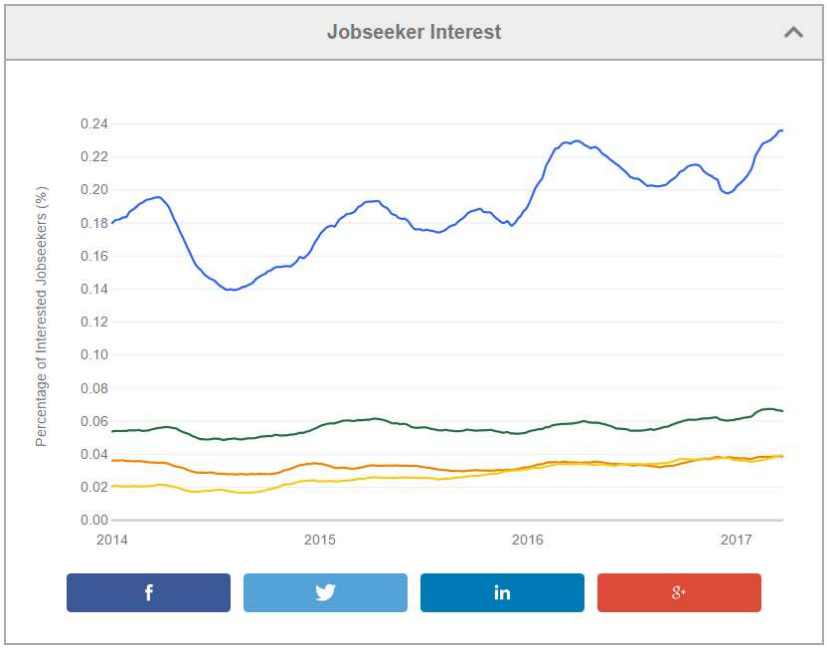
Lisa 3 – Keskkonnast Indeed töökohtade üldine trend: Java, PHP, C# ja JavaScript

Seisuga 24.03.2017 [40]

java, php, c#, and javascript Job Trends

java × php × c# × javascript × + Add Term Find Trends





Lisa 4 – Keskkonnast Indeed töökohtade üldine trend: SOA ja RESTful

Seisuga 24.03.2017 [41]

Lisa 5 – Ettevõtte Nortal sisemine uuring: SOA, Angular ja modulaarne arhitektuur

Käesoleva lisa on Nortalis aastal 2015 läbi viidud sisemine uuring, mille käigus üritati välja selgitada, miks uue tehnoloogiaga arendamine on võrreldes varasemate tehnoloogiatega aeganõudvam. Uuringus on välja toodud probleemid ning nende võimalikud lahendused.

Eesmärk

Eesmärk on

1. kirjeldada kitsaskohad, mida SOA+Angular arhitektuuriga alustanud tiimid peaks teadma, et vigu ja liigset ajakulu vältida.
2. üritada aru saada, miks uus arhitektuuriline lähenemine erinevatel rollidel rohkem aega nõuab kui nt ARANEA või Spring MVC

Analüüs

Lahenduse valikute piirangud analüütikutele (ja arhitektidele)

Andmete piiratud kättesaadavus nimekirjade sorteerimiseks

Kui andmed tuleb nimekirja leida erinevatest moodulitest REST teenustega ja nimekirjas võib olla palju kirjeid. Kui pole enda mooduli andmed, ei saa väliste nimetuste järgi sorteerida suurtes nimekirjades, sest see tekitab jõudluse probleeme. Saab selle järgi, mis sisaldub enda moodulis salvestatud andmestikus, aga see ei pruugi olla kasutajale loogiline ja arusaadav.

- Näide: kliendi andmed on eraldi moodulis, kust neid ülejäänud moodulid REST teenustega pärivad. Kui enda moodulis andmeobjektidele mahasalvestada vaid kliendi id, siis kasutajale kuvatav nimi tuleb leida REST teenusega teisest moodulist. Tuhandele nimekirja kirjele nime külge laadimine, võib teha nimekirja laadimise aeglaseks. Kui neid aga külge ei laadi, siis saab sorteerida vaid kliendi id järgi, mis kasutaja vajadust nime järgi sorteerida ei rahulda.
- Lahendus:
  - Analüüsis kindlasti selgeks teha, kui palju kirjeid nimekirjas saab olla, kui palju erinevate väärtustega neid, mida tuleb REST teenusega pärida (nt 1000 kirjet, aga maksimaalselt 100 klienti ei tekita probleemi; aga 1000 klienti juba tekitab). Rääkida disainifaasis juba arendajaga läbi, mis andmed vaja REST teenustega leida ja kui palju võib olla korraga leidmist vajavaid väärtuseid. Arendaja roll öelda, kas tekib jõudlus probleeme.
    - Kui ei teki, saab teha tavapärase sorteerimise
  - Kui on jõudlusprobleemi risk, siis järgmised variandid, millest ükski pole väga hea lahendus:
    - kui on olemas tsentraalne moodul/teenused, mida tarbivad kõik ülejäänud kohad, siis sarnane loogika peaks kehtima ka DB tasemel - näiteks ID põhjal joinitavad globaalsed view'd (isiku põhiandmed, KL nimed, taotluse põhiandmed jne.). Neid globaalseid view'sid võivad defineerida erinevad moodulid. Vajadus peaks olema põhjendatud - lihtsalt lambist mingeid globaalseid view'sid tegema pole mõtet hakata ja need globaalsed view'd peaks siis ka andmebaasis asuma omaette baasis/skeemis. On siiski libe tee, sest on modulaarse eraldatuse põhimõttega vastuolus ning võib tekkida ka dubleerimist: sama asja ühes kohas leitakse REST teenusega ja teises globaalse view'ga. Seni pole rakendanud.
    - salvestada enda baasi rohkem kui id ja sünkida andmeid teise mooduliga regulaarselt (PMA-s nt nii tehtud, et teise mooduli kliendi andmetest salvestavad endale lisaks kliendi id-le ka nime ja isikukoodi). Samas see ei ole tegelikult korrektne lahendus, sest nii kaob modulaarsusel mõte, kui kõik moodulid üksteise andmeid duubeldavad.
    - sundida kasutajat nimekirja leitavaid objekte piirama, et tal poleks võimalik tekitada olukorda, kus korraga vaja üle 200 väärtuse leida erinevate REST teenustega. Ehk et kõigepealt mingite parameetrite järgi pärima (eraldi otsinguvormil) ja kui kirjeid vähem kui 200, siis alles kuvatakse nimekiri
    - uurida meie kliendilt, kui oluline on antud veerus sorteerimise võimalus. Kui on vajalik, ei tohiks sundida klienti sellest loobuma.

Andmete piiratud kättesaadavus nimekirjade filtreerimiseks

Sarnaselt sorteerimisega probleemiks, kui andmed tuleb nimekirja leida erinevatest moodulitest REST teenustega ja nimekirjas võib olla palju kirjeid. Ei tekita probleeme, kui filtreeritavas veerus kuvatakse välise teenusega nt meie moodulisse salvestatud id või koodi järgi päritud väärtus ja filtreerimiseks kuvatakse valikmenüü, kus iga väärtuse puhul teada vastavus kasutajale kuvatava ja meie andmeobjektile salvestatud väärtuse vahel (nt valikmenüüs klassifikaatori elementide inimkeelsed väärtused, baasi salvestatud andmeobjektile klassifikaatori elemendi kood ning teada, mis kood igale inimkeelsele väärtusele vastab >> võimalik filtreerimine teha kasutajale inimkeelse väärtuse järgi, aga süsteemis koodi järgi). Probleem tekib, kui vastavus pole endale cache'itud andmetest teada, nt kliendi id ja nimi.

- Lahendus: samad kui sorteerimisel, aga lisaks veel:
  - otsinguvormi kasutamine filtreerimise rea asemel
  - sellistes veergudes eraldi otsingu lahendus. Et ei saa filtriribale kliendi nime sisestades filtreerida, vaid peab filtriribal avama lingist kliendi otsingu, otsima kliendi nime/koodi järgi ja siis tema järgi filtreeritakse)

Navigeerimine

Asukoha meelde jätmine ei pruugi alati olla lihtsasti realiseeritav (hübriid kasutaja liikumisest ja sisu arhitektuurist)

- Näide: Kasutaja vajutab 'Lisa uus' nuppu, avaneb lisamise vaate url 'blabla/lisa'. Kasutaja vajutab lisamise kuval Salvesta nuppu, avaneb salvestatud objekti url 'blabla/id' (objekti id, mis salvestati) ehk url vahetub. Kasutaja vajutab Katkesta või back ja ootab, et ta viiakse sinna, kus ta Lisa uus nuppu vajutas. Ei või viia lihtsalt url'i võrra tagasi, sest siis satub lisamise vaatesse. Ei saa defineerida üheselt ka Lisa uus nupu url-i asukohta, kui see on mitmes kohas, nt mingis nimekirjas ja ka detailkuval. Kokkuvõttes ei saa kasutajat tagasi lihtsalt viia sinna, kust ta tuli
- Lahendus:
  - Luua custom ajaloo loogika lahendus, mis kasutaja navigeerimist trackib ja mida võimalik eri kohtades kasutada
  - Kuni custom lahendus puudub, defineerida analüüsis üheselt, mis kohta tuleb kasutaja tagasi viia

FE kuva suurus ja keerukus

Mida rohkem on elemente, millega tegeletakse vastavalt kasutaja valikutele, seda aeglasemaks kuva laadimine ja seal tehtavad tegevused muutuvad. ARANEAs tekitavad suured dünaamilised kuvad serveri poolel probleeme, Angularis kasutaja rakenduses

- Lahendus:
  - analüüsis pöörata tähelepanu sellele, et kuvad oleksid piisavalt väikesed: mõistlik piir oleks 1000 watcherit. Arendajaga valideerida kindlasti ka see ületatud.
  - kui kuval pole midagi muud vaja teha kasutajal peale vaatamise, siis aitab nt lahtiklõpsatavates akordionites andmete kuvamine või kerimisel vaid kasutatava osa laadimine (a la Skype vestluse ajalugu või FB). Kui kuval on vaja kõike lõpuks salvestada või kinnitada, siis selline jupikaupa laadimine ei aita, sest iga jupiga watcherite arv kasvab. Lõpuks salvestamiseks on ikka liiga suur ja kasutaja peab liiga kaua ootama, kuni süsteem salvestab
  - jagada kuvad loogilisteks sammudeks/tab'ideks
  - kasutada lahendust, kus kuval on nimekiri, millest saab avada ükshaaval osi ja nendega toimetada. Kui vajalik lõpuks koguandmestiku valideerimine (nt kinnitamisel), saab teha nii, et veateates on kasutajale link, mis viib sinna kohta, kus viga oli ehk kasutaja ei pea ise hakkama nimekirjast läbi klõpsima osi ja otsima, kus viga on.

Sõnumite kuvamine (vead, hoiatused, teated)

Tegeletakse vaid ekraani osaga, mitte kogu kuvatava infoga

- Lahendus: Teateid kuvada selliselt et kasutaja näeks neid, kuid tema kuva ei tohiks muutuda. Kui ma näiteks olen sisestamise vormis ja vajutan nupule salvesta ning saan vea, siis ei ole alati vajalik, et mind viidaks teate juurde, vaid selle asemel tuleks teade minu juurde. Üks näide, mis ei ole küll kõige parem, kuid annab vast mõtte edasi: htt p://www.wduffy.co.uk/blog/wp-content/demos/jquery-scrolling-element/

Turvalisuse tagamine

Analüütik peab arendajale mingi tegevuse tingimuste kirjeldamisel arvestama sellega, et REST teenus on eraldi välja kutsutav

- Näide: Nimekirjas lingist Muuda avaneb muutmiskuva, kus on alati võimalik salvestada ning lisaks kinnitada saab, kui on kinnitamise privileeg olemas
  - ARANEAs on vaja kirjeldada Muuda lingi kuvamistingimused (kasutajal see õigus, taotlus sellises staatuses jne), Salvestamisele eraldi tingimusi ei ole vaja kirjeldada, sest ilma kuvale jõudmata ei saa kasutaja salvestada. Kinnitamise tegevusele tuleb panna vaid kinnitamise privileegi piirang
  - SOAga saab kasutaja lisaks rakenduse nuppudele aga kasutada url-e ja REST teenuseid välja kutsuda (kohati häkkides). Nt on muutmiskuva võimalik avada kirjutades urli reale vastava objekti id-d sisaldav aadress ning häkkides on võimalik ka salvestamise ja kinnitamise teenust välja kutsuda ilma rakenduses nupule vajutamata. Seepärast peavad arendajale kirjeldatavad salvestamise tingimused sisaldama ka muudetavuse ('Muuda' lingist kuva avamise) tingimusi, sest muidu on võimalik salvestada ebasobivat objekti.
- Lahendus ehk millal tingimuse kontrollile lisada kasutajale tagastatav veateade ja millal mitte:
  - tegevuse (nt andmete laadimise, salvestamise, kinnitamise) kontroll kirjeldada koos veateatega, kui rakenduses võimalik nuppu vajutada, aga tegevus tegelikult keelatud. Nt kasutajal pole kinnitamiseks vajalikku privileegi.

- kontroll kirjeldada ilma teateta, kui rakenduses pole võimalik nuppu vajutada ehk kontroll puhtalt selleks, et põlveotsas ei saaks teenust häkkida. Mingit ilusat inimkeelset veateadet sellistele häkkijatele tagastada pole vaja

Ressursside laadimine

Angular rakenduse laadimisel laeb Angular kasutajaliidessesse ressursid nagu CSS ja JS. Mõningatel juhtudel ka HTMLid, see sõltub sellest, kuidas lahendus on üles ehitatud. Alati ei ole vaja ja mõistlik kõiki ressursse alla laadida ning teatud juhtudel võib tekitada see probleeme.

- Näide:
  - Sisselogimise leht, mille kujundus on üldjuhul üldisest rakendusest erinev. Lisaks ei ole üldjuhul login lehe kuvamisel vaja kasutaja arvutisse laadida muud infot, mida logimise aknas ei ole vaja. Võivad tekkida konfliktid, näiteks logimise stiili elemendid kirjutatakse üle või vastupidi. Kogu rakenduse allalaadimine muudab logimise lehe esmakordsel avamise aeglaseks.
- Lahendus:
  - Eraldada täielikult logimine ja sisupool. Sisuliselt tekib kaks Angular lahendust. Või mõelda ressursside injecti mise peale ehk laeme ressursi siis, kui meil seda reaalselt vaja läheb.

Põhjused, miks analüüsile on kulunud rohkem aega

Õpikõver

SOA ja modulaarse arhitektuuriga alustades on õpikõver ja kulub aega, et

- nihutada harjumuspärast mõtlemismalli
- aru saada, mis moodi uus arhitektuur erineb harjumuspärasest (ARANEAst) ja kuidas see mõjutab arendajatele info kirjeldamist
  - Näide: Kui ARANEAs on muutmiskuva, kus kasutaja näeb alati Salvesta nuppu, siis analüütik kirjeldab muutmiskuvale jõudmise tingimused (kasutajal see õigus, taotlus sellises staatuses jne) ja Salvestamisele eraldi tingimusi ei kirjelda, sest ilma kuvale jõudmata ei saa kasutaja salvestada. SOAga saab kasutaja aga eraldi REST teenust (häkkides) välja kutsuda ehk vaja Salvestamisele panna ka kõik need piirangud, mis on muutmiskuvale jõudmisel.
- teada ja aru saada eespool kirjeldatud valikute piirangutest

- aru saada, kuidas dokumenteerida süsteemi selliste klotsidena, et need oleksid lihtsalt ja mugavalt taaskasutatavad järgmiste kasutuslugude kirjeldamisel ning samas luua lugejale klotsides tervikvaade
- Lahendus:
  - täiendada ja jagada siin lehel kirjeldatud valikute piiranguid ning infot levitada
  - SOAga alustav analüütik viia kokku SOA kogemusega analüütikuga. Juhendamisest ja kogemuse jagamisest on väga palju kasu

Angulari protomootor

Angulari protomootoriga prototüüpimine võtab rohkem aega, sest tuleb kasutada angulariseeritud HTMLi ega saa kasutada nii lihtsalt harjumuspäraseid visuaalseid abivahendeid nagu Dreamweaver. Angulari protomootori kasutuselevõtmise idee oli, et arendaja saab võtta sealt copy-paste ja front-endi kood ongi põhimõtteliselt koos. Tegelikult arendajal kaks valikut: võtta analüütiku kood ja veenduda, et see ei sisalda vigu (kuva välimus võib olla ilus, aga Angulari elementide kasutus vigane) või teha ise nullist. Ilmselt isiklik eelistus, kumb lihtsam ja kiirem on

- Lahendus:
  - loobuda Angulari protomootori kasutamisest. Protot teeks analüütik lihtsas HTMLis ning see oleks ennekõike analüütiku ja kliendi töövahend
    või
  - arendaja juhendab analüütikuid, et nende kirjutatav prototüüp oleks koodialgena kasutatav. Analüüsis võtab algul õpiaega, aga kokkuvõttes arenduses tekib võit.

FE ja BE eraldi dokumenteerimine

SOA lähenemine eraldab esitluskihi ja teenuskihi. Kuna nad on sedavõrd eraldatud, siis PRIA2014-s proovisime eraldada ka nende spetsifikatsioonid. Kuid see muutis arusaamise terviklahendusest keerulisemaks (kliendil, arendajal ja testijal oli raske lõpuks aru saada, kuidas kõik kokku peaks kõlama). Samuti nõudis see analüütikult rohkem aega, kuna füüsiliselt pidi lihtsalt rohkem kirjutama ja kui midagi arenduses muutus, siis mitmes kohas uuendama.

- Lahendus:
  - Analüüsi dokumentatsioonis keskenduda tervikvaate, tervikliku kasutusloo kirjeldamisele ning mitte üritada jagada seda esitluskihi ja teenuskihi vahel ega mõelda välja konkreetseid REST teenuseid, sest seda jaotust ja detailsust peaks mõtlema välja arendaja ja see selgub arenduse käigus.
  - Küll peab dokumentatsiooni struktuuri välja mõeldes pöörama tähelepanu taaskasutatavatele klotsidele (et ühes kohas saaks kirja ja saaks excerptina hõlmata, kuhu vaja) ehk Conflu lehti võib olla nt 5, aga lugejale on neist kokku orkestreeritud üks tervikut kirjeldav kasutuslugu
  - NB! Sel kujul ei anna analüüsi dokumentatsioon arendajale infot, mis JSON objekte ja mis REST teenuseid, kus kasutatakse. See info on vajalik, et teaks taaskasutada ega leiutada uusi asju. Selleks vajalik lahendus on kirjas arenduse osas.

Arendus

Põhjused, miks arendusele on kulunud rohkem aega

Õpikõver

- Õpikõver on SOA + AngularJS peale minemisel üpris suur:
  - tundma tuleb õppida uusi tehnoloogiaid, vt ka kihtide rohkus
  - muutuma peab ka nn mõttemaailm
- Lahendus:
  - täiendada ja jagada siin lehel kirjeldatud infot
  - SOA ja Angulariga alustav arendaja viia kokku kellegagi, kel vastav kogemus olemas. Juhendamisest ja kogemuse jagamisest on väga palju kasu

Kihtide rohkus

Võrreldes varasema lähenemisviisiga, kus enamus töö tehti ära BEs on Angulari puhul kaaluskaus liikunud rohkem FE suunas. Sellega seoses on juurde tekkinud täiendavad kihid. PRIA2014 näitel:

- Esitluskiht (JS) – tegeleb andmete pärimise ja representeerimisega, kutsub välja REST teenuseid. Peab teadma ka Bootstrap poolt pakutavaid võimalusi
- Kontrollerkiht – Võtab vastu sissetulevad päringud ja tagastab tulemuse. Dokumenteerimiseks kasutatakse Swaggeri võimalusi. JSON muudetakse DTO (Data transfer object) mudelobjektiks
- Interaktorkiht – Tegeleb esitluskihist DTO objekti sisu valideerimisega, DAO kihi jaks mudelobjektiks loomisega, teenuste välja kutsumisega
- Teenuskiht – Komplekssete andmetoimingute teostamine kutsudes välja erinevaid DAO (Data access object) toiminguid
- Andmekiht – DAO toimingud andmete manipuleerimiseks, salvestamiseks, kustutamiseks
- Andmebaasikiht – Andmed, funktsioonid, protseduurid, jne

Miks selline kihilisus rohkem aega nõuab:

- Iga kiht nõuab eraldi aega
- Põhitöö tuleb teha ära FE kihis, kus me ei ole harjunud toimetama: õpikõver ja vigadest õppimine
- Iga kiht tuleks sisuliselt katta testidega, mis tõttu kihtide rohkus nõuab rohkem aega testide kirjutamiseks

Moodulite rohkus

Üldjuhul tuleb suhelda erinevate moodulitega andmete pärimise või salvestamise eesmärgil. Mida rohkem on sellised nn väliseid teenuse välja kutsumisi seda ajamahukamaks kipub komponendi arendamine kujunema:

- Praktika on näidanud, et üldjuhul esineb teise mooduli teenuse kasutamisel probleeme ning nende probleemide lahendamine võtab aega.
- Tehniline lahendus on keerulisem ja võtab rohkem aega erinevate andmekoosseisude loomine ning nende põhjal toimingute teostamine. Näiteks tabelite loomine koos filtreerimise ja sorteerimise võimekusega

Olekuvabad (stateless teenused)

Araneas, kui kasutaja logib sisse, siis luuakse kasutajasessioon, kus hoitakse tema ja kuvade andmeid. Seda me enam ei tee, BE kihis on meil kasutaja turvakontekst. Olekuvaba tähendab, siis seda, et iga teenus on eraldiseisev ja ta ei tea teistest teenustest midagi. Seepärast on vaja pöörata ekstra tähelepanu teenusarhitektuurile, et nt eri objektide andmete salvestamine käiks ühe konkreetse teenuse vahendusel. Muidu tekivad andmeterviklikkuse probleemid.

- Näide: Kasutaja failide üles laadimine. Kui kasutaja lisab taotlusele faile, siis kuidas need siduda taotlusega. BE kihis puudub sessioon, mis seoks ajutiselt kasutaja ja tema andmed. Võimalik on muidugi REST teenuse vahendusel nüüd üles laadida nn ajutiselt. Kuid see ei ole võib-olla kõige mõistlikum, kuna kasutaja võib oma valikuid muuta, või üldse loobuda. Sellisel juhul oleks vaja täiendavat tegevust nn ajutiste failide kustutamiseks.
- Lahendus: Andmeterviklikkuse tagamiseks luua lahendus nii, et andmete salvestamine toimub ühe konkreetse teenuse vahendusel. Selle asemel et luua REST teenused ajutiste failide üleslaadimiseks, muutmiseks, kustutamiseks ja taustatöö üleliigsete eemaldamiseks, laadida kasutajafailid kasutajaliidessesse ja alles siis kui kasutaja soovib salvestada saata failid koos teiste andmetega.

Info killustatud mitme spetsifikatsiooni vahel

SOA lähenemine eraldab esitluskihi ja teenuskihi. Kuna nad on sedavõrd eraldatud, siis proovisime eraldada ka nende spetsifikatsioonid. Kuid see muutis arusaamise üldisest lahendusest keerulisemaks (nt kust ma leian kontrollid, mida hetkel vaja kontrollida? Ühe kuva laadimisel võib olla vaja kasutada erinevaid teenuseid, teenuste kirjeldused on eraldi spetsifikatsioonides, mis vaja üles leida ja läbi lugeda)

- Lahendus:
  - Analüüsi dokumentatsioonis keskenduda tervikvaate, tervikliku kasutusloo kirjeldamisele ning mitte üritada jagada seda esitluskihi ja teenuskihi vahel ega mõelda välja konkreetseid REST teenuseid, sest seda jaotust

ja detailsust peaks mõtlema välja arendaja ja see selgub arenduse käigus. Küll tasub analüütikul dokumentatsiooni struktuuri välja mõeldes pöörata tähelepanu taaskasutatavatele klotsidele (et ühes kohas saaks kirja ja saaks includida, kuhu vaja) ehk Conflu lehti võib olla nt 5, aga lugejale on neist kokku orkestreeritud üks tervikut kirjeldav leht
- See tähendab, et analüüsi dokumentatsioonist ei leia arendaja ega testija infot olemasolevate JSON objektide ja REST teenuste kohta (kus, mida ja kuidas kasutatakse), mis on vajalik selleks, et võimalusel taaskasutada olemasolevat, mitte luua hunnikut unikaalseid teenuseid, milledest kaob kiiresti ülevaade. Kui taaskasutust pole, siis pole ka SOAst oodatud võitu. Seega on tolle info talletamine väga oluline, kuid see peab toimuma arenduse käigus ja soovitavalt mingi automaatse abivahendiga, mis meil hetkel puudub (vt järgmist probleemi kirjeldust).

Puudub tarkvaraline abivahend teenuste kirjeldamiseks

REST teenus, mida kasutajaliidese vahendusel tarbitakse on lihtsustatult öeldes sisuliselt liides tarkvara kasutusliidese ja andmebaasi vahel. Ehk kuidas andmed jõuavad punktist A punkti B ja mis on ja mis kujul saadetakse punktist B vajalikud sisendid andmete pärimiseks punkti A. Kuna tegemist on liidesega, siis sellised liidesed peaks sisuliselt dokumenteerima, kus kasutatakse, kuidas kasutatakse. Kui need on teada, siis on tulevikus lihtsam otsustada, et kus me saame sama teenust veel kasutada. Kui me nii ei mõtle, siis on meil tulevikus hunnik unikaalseid teenuseid milledest kaob kiiresti ülevaade ja SOA lähenemisest oodatud taaskasutuse võitu ei teki.

- Lahendus: Leida vahend, mis lihtsustab REST põhimõtete mõistmist ja nende ellu viimist. Tarkvaraline abivahend ennekõike arendajale (info saamiseks ka analüütikule ja testijale), mis võimaldab:
  - disainida REST teenuseid
  - REST teenuseid struktureerida
  - dokumenteerida REST teenuseid
  - luua Swagger väljund
  - JSONit kirjeldada
  - uute JSONite kirjeldamisel olemasolevate kasutamist jättes alles seose. JSONeid saab korduvkasutada ja tekib kaart, kus on näha nende kasutust
  - tulemust linkida Confluence'iga

Testimine

Testimise jaoks eelised

SOA

Rakendus koosneb paljudest eri teenustest, mis moodustavad terviklahenduse.

- Testimise alamülesanneteks jaotamine on lihtsam ja sellest tulenevalt on terviku testimine lihtsam kuna saab keskenduda ka väikestele teenustele suure komponendi kõrvalt (väiksemat juppi on tervikuna kergem haarata ning talle testjuhte välja mõelda).
- Testija saab paremini rakendust debuggida, kuna ta pääseb sisemistele teenustele kergelt ligi ning komponendi eri osasid eraldi vaadata.

Põhjused, miks testimisele on kulunud rohkem aega

Õpikõver

- Uued vahendid ja viisid tegevuste jälgimiseks
  - Tuleb selgeks teha REST teenused ja kuidas neid testida.
  - JSONite loomine, lugemine ning muutmine.
  - Swagger.
  - Developer tools
- Lahendus:
  - täiendada ja jagada siin lehel kirjeldatud infot
  - SOA ja Angulariga alustav testija viia kokku kellegagi, kel vastav kogemus olemas. Juhendamisest ja kogemuse jagamisest on väga palju kasu

Ekstra tähelepanu jõudlusele

Jõudlusele tuleb palju tähelepanu pöörata, kuna see on teadaolev probleem nii SOA-ga (kuna palju eri teenuseid suhtleb üle võrgu üksteisega) kui ka Angulariga (suurte vaadete puhul, kus palju elemente, millel watcherid)

SOA

- Palju erinevaid teenuseid, mis moodustavad terviklahenduse. Sellest tulenevalt spetsifikatsioonid võivad olla rohkem laiali ning keerulisem lugeda ning otsida õiget infot. Sellest tulenevalt komponendi tervikpildi saamine keerulisem. T estimine on ajamahukam, kuna vaadatakse nii teenuseid kui ka tervet komponenti.
  - Lahendus:
    - Analüüsi dokumentatsioonis keskenduda tervikvaate, tervikliku kasutusloo kirjeldamisele ning mitte üritada jagada seda esitluskihi ja teenuskihi vahel ega mõelda välja konkreetseid REST teenuseid,

sest seda jaotust ja detailsust peaks mõtlema välja arendaja ja see selgub arenduse käigus. Küll tasub analüütikul dokumentatsiooni struktuuri välja mõeldes pöörata tähelepanu taaskasutatavatele klotsidele (et ühes kohas saaks kirja ja saaks includida, kuhu vaja) ehk Conflu lehti võib olla nt 5, aga lugejale on neist kokku orkestreeritud üks tervikut kirjeldav leht
    - See tähendab, et analüüsi dokumentatsioonist ei leia arendaja ega testija infot olemasolevate JSON objektide ja REST teenuste kohta (kus, mida ja kuidas kasutatakse), mis on vajalik selleks, et võimalusel taaskasutada olemasolevat, mitte luua hunnikut unikaalseid teenuseid, milledest kaob kiiresti ülevaade. Kui taaskasutust pole, siis pole ka SOAst oodatud võitu. Seega on tolle info talletamine väga oluline, kuid see peab toimuma arenduse käigus ja soovitavalt mingi automaatse abivahendiga, mis meil hetkel puudub (vt järgmist probleemi kirjeldust).
- Autentimine
    - Kõik sõnumid, mis teenuste vahel liiguvad on turvatud ning neile tuleb autentimise tunnused pidevalt kaasa anda.

Angular JS

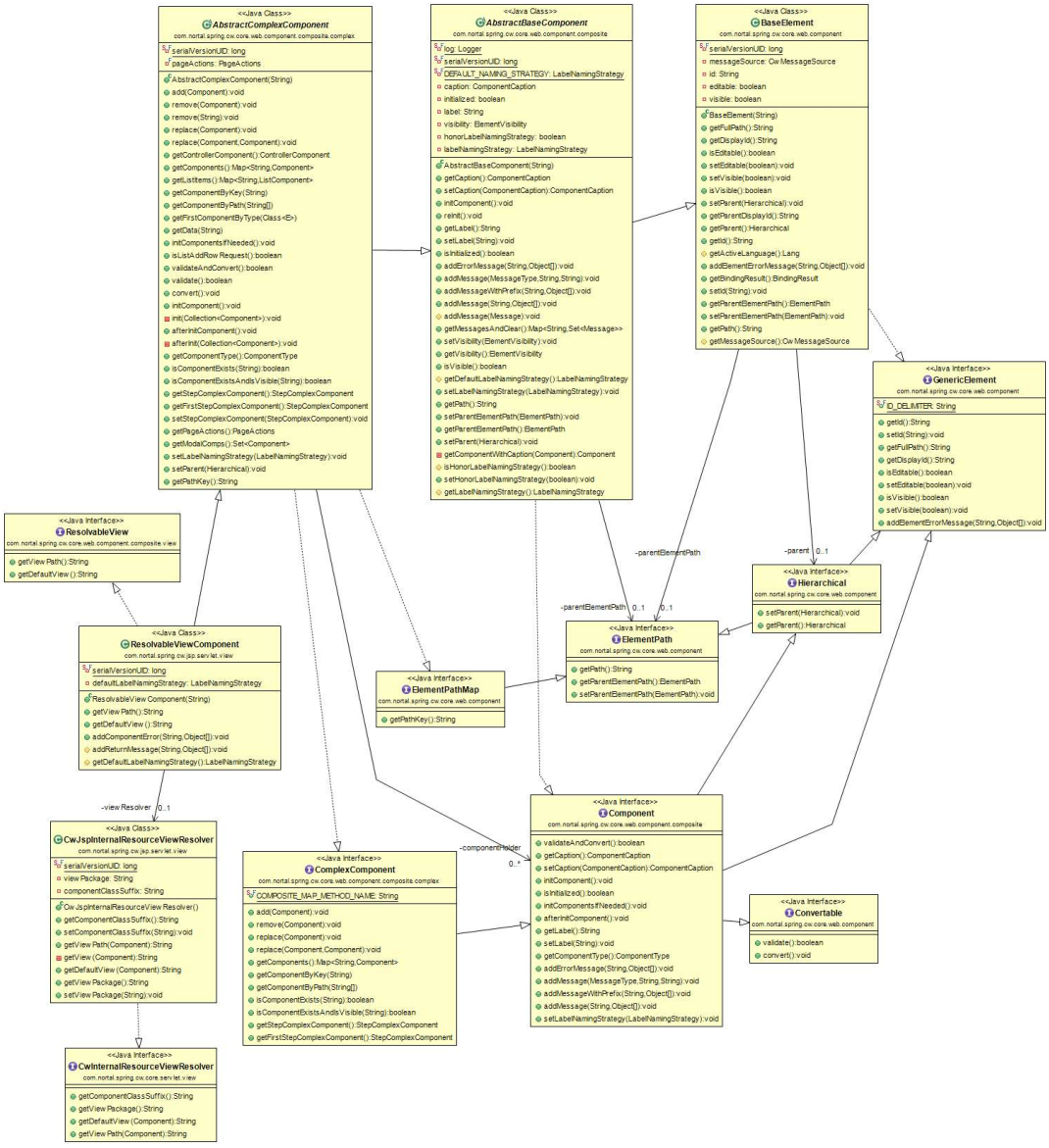
Angular JS

- Testandmete moodustamine võib keeruline ja aeganõudev olla kui REST teenuste JSONid väga suureks aetakse.
- Tuleb tunda JavaScripti spetsiifilisi veasituatsioone ning jälgida JavaScrpiti konsooli pidevalt
    - Ehk tuleb lisaks rakenduse logile pidevalt jälgima hakata ka developer toolsi, kuna front-endi pannakse Angulari puhul palju rohkem äriloogikat.
- Front-end ja back-end on väga lahku löödud ning nende vead tulevad eri kohtadest välja. Sellest tulenevalt tuleb vigu erinevalt käsitleda (nt kui on eraldi front-end ja back-end arendajad)
- Juhul kui üks arendaja teeb back-endi ning teine front-endi samal komponendil siis vigade tõenäosus suur ning suhtlus keerulisem kuna pidevalt vaja välja selgitada kummalt midagi küsida.

- Uued HTTP staatused, nende tähendus ja eesmärk.
- Turvaprobleemid on suured ja tuleks rõhku panna turvatestimisele, kuna kasutaja võib näiteks saada navigeerida ennast kuskile, kuhu ei tohiks saada ja samuti kuna front-end ja back-end on nii eraldi siis tuleb vaadata, et piirangud oleks mõlemas kihis.

- Kuna kuvadel ei laeta kogu lehte tihtipeale vaid osa sellest, siis tuleb jälgida, et kõik kuva elemendid on oodatult laetud.
    - Tuleks arvestada seda punkti ka näiteks Seleniumi kasutusele võtmisel (ning kaaluda ka Protractorit).
    - Samuti võib see tekitada mälulekke probleeme ning tuleks ka seda kontrollida vahel.

Lisa 6 –Spring MVC komponendipõhise tarkvaraarendusraamistiku komplektse komponendi abstraktne klassdiagramm



Komplektsed komponendid omavad endas teisi komponente, mille sees on konkreetsed välja kuvatavad elemendid. Komplektsel komponendil on enda vaade, kus on kirjeldatud tema sees elavate komponentide paigutus.

Lisa 7 –Spring MVC komponendipõhise tarkvaraarendusraamistiku komplektse komponendi Java koodi näide

```java
package com.nortal.spring.cw.jsp.web.portal.component.test;

import java.math.BigDecimal;

/**
 * @author Margus Hanni
 *
 */
public class TestComponent extends ResolvableViewComponent {

   private static final long serialVersionUID = 1L;
   public static final String TEST_KOMPONENT = "testKomponent";
   private static final String VORM = "vorm";

   public TestComponent() {
      super(TEST_KOMPONENT);
   }

   @Override
   public void initComponent() {
      setCaption(new ComponentCaption("#Test komponent"));
      buildForm();
      super.initComponent();
   }

   private void buildForm() {
      add(getVorm());

      // Näitame komponendi salvestuse nuppu ainult siis kui sammu komponent ei
      // ole aktiivne
      getPageActions().addMainButton(EventElementFactory.createButton(Type.SAVE).setVisibility(new ElementVisibility() {

         private static final long serialVersionUID = 1L;

         @Override
         public boolean isVisible(Hierarchical parent) {
            return getFirstStepComplexComponent() == null;
         }
      }));
   }

```

```
60    private CwFormComponent<ObjectTest> getVorm() {
61
62       /*
63        * Väljade omadused leitakse mudelobjekti annotatsioonidelt
64        */
65
66       CwFormComponent<ObjectTest> formComp = new CwFormComponent<>(VORM, ObjectTest.class);
67       formComp.setEditable(true);
68
69       DateTimeElement dateTimeField = formComp.add("dateTimeField");
70       dateTimeField.setFormat(DateTimeElementFormat.DATETIME);
71
72       DateTimeElement dateField = formComp.add("dateField");
73       dateField.setFormat(DateTimeElementFormat.DATE);
74
75       DateTimeElement timeField = formComp.add("timeField");
76       timeField.setFormat(DateTimeElementFormat.TIME);
77
78       StringCollectionElement list = formComp.add("list");
79       list.setMultiValue(new MultiValueHolder(SelectElementType.MULTISELECT, getTestMultiselectData()));
80
81       formComp.add("decimalField");
82       formComp.add("integerField");
83       formComp.add("longField");
84
85       formComp.add("textField");
86       formComp.add("requiredTextField");
87
88       StringElement longTextField = formComp.add("longTextField");
89       longTextField.initRichText();
90
91       formComp.setData(getData());
92       return formComp;
93    }
94
95    private Map<Object, Object> getTestMultiselectData() {
96       Map<Object, Object> dataMap = new HashMap<>();
97
98       for (int i = 1; i < 12; i++) {
99          dataMap.put("valik_" + i, "Valik " + i);
100      }
101
102      return dataMap;
103   }
104
105   private ObjectTest getData() {
106      ObjectTest objectTest = new ObjectTest();
107      objectTest.setDateField(new Date());
108      objectTest.setDateTimeField(new Date());
109      objectTest.setTimeField(new Date());
110      objectTest.setDecimalField(BigDecimal.valueOf(0.50));
111      objectTest.setIntegerField(10);
112      objectTest.setLongField(111111L);
113      objectTest.setTextField("Tekstiväli");
114      objectTest.setRequiredTextField("Kohustuslik tekstiväli");
115      objectTest.setLongTextField("");
116
117      return objectTest;
118   }
119}
```