LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

RESTful ARHITEKTŪRĀ VEIDOTU SERVISU OPTIMIZĀCIJA UN DROŠĪBA

BAKALAURA DARBS

Autors: **Rihards Fridrihsons**
Studenta apliecības Nr.: rf11012
Darba vadītājs: profesors Dr.sc.comp. Guntis Arnicāns

RĪGA 2015

ANOTĀCIJA

Lai tīmekļa servisi būtu augstā līmenī, tiem nepieciešams darboties ātri un droši. Ir dažādas metodes, kā uzlabot servisu ātrdarbību, drošību, lai varētu apkalpot vairāk lietotājus vienlaicīgi, un apkalpot tos ātrāk. Ir nepieciešams veikt dažādas optimizācijas metodes, kā arī nodrošināt servisu drošību pret dažādiem uzbrukumiem, kuru rezultātā lietotāju dati var tikt izmantoti ļaunprātīgi. Darbā tiks apskatītas metodes kā atrisināt vai vismaz daļēji atrisināt šīs problēmas.

ATSLĒGVĀRDI

REST, drošība, veiktspēja, tīmekļa servisi

ABSTRACT

Optimization and security of RESTful services

For web services to be in high level it has to work fast and safe. There are several methods to imporve performance, security of RESTful services to serve more users at the same time and to serve them faster. It is necessary to perform several optimisation methods and to provide the security of services against several attacks, which may lead to exposure of user's private data. The aim of this to review methods how to solve or at least partially solve these problems.

KEYWORDS

REST, security, performance, web services

SATURA RĀDĪTĀJS

Apzīmējumu saraksts 6

Ievads 7

1. REST Arhitektūras stils 8

1.1. HTTP 8

1.2. REST priekšrocības 8

1.3. REST metodes 9

1.4. Alternatīvas 9

2. REST servisu drošība 10

2.1. Autentifikāciju veidi 10

2.1.1. Pamata autentifikācija (Basic authentication) 10

2.1.2. OAuth 1.0 11

2.1.3. OAuth 2.0 11

2.1.4. Īssavilkuma autentifikācija (Digest Auth) 12

2.2. Drošības pamatprincipi 13

2.2.1. Autentifikācija un sesijas pārvaldība 14

2.2.2. HTTP metodes 15

2.2.3. Atļautās metodes 15

2.2.4. CSRF un XSS 15

2.2.5. Tiešas piekļuves objektu darbības 16

2.2.6. Ievaddatu validācija 16

2.2.7. Izvaddatu kodēšana 17

2.2.8. HTTPS 17

3. Veiktspēja 18

3.1. REST lietotnes projektējums izmantojot HTTP 18

3.2. Vertikālā mērogošana 20

3.3. Horizontālā mērogošana 20

3.4. Kešdarbe 21
3.4.1. Klienta puses kešdarbe 21
3.4.2. Servera puses kešdarbe 21
3.4.3. Individuālas platformas kešdarbes alternatīvas 22
3.5. Datu apjoms 22
4. Lietojuma projektēšana 23
4.1. Programmatūras pārskats 23
4.2. Mobilā lietotne 24
4.3. Administratora panelis 25
4.4. REST serveris 26
4.5. Resursu modelēšana - administratora funkcijas 27
4.6. Resursu modelēšana – lietotājs 28
4.7. Autentifikācija un autorizācija 29
4.8. Drošības principi 30
4.9. Veiktspējas testi 31
4.10. Stresa testi 33
4.11. Slodzes testi 38
4.12. Datu apjoms 39
Secinājumi 40
Izmantotā literatūra un avoti 41

APZĪMĒJUMU SARAKSTS

- HTTP - hiperteksta pārsūtīšanas protokols.
- HTTPS – hiperteksta pārsūtīšanas drošības protokols.
- XML – paplašināmās iezīmēšanas valoda.
- SOAP – vienkāršais objektpiekļuves protokols.
- API – lietojumprogrammas saskarne.
- HMAC – hash, kas aprēķināts no autentificējamajiem datiem un slepena datu bloka.
- OWASP – atvērtais tīmekļa lietotņu drošības projekts.
- CSRF - *Cross Site Request Forgery* uzbrukuma veids.
- XSS – *Cross Site Scripting* datoru drošības ievainojamību paveids, kas, visbiežāk, skar tīmekļa lietojumprogrammas.
- TLS – ir šifrēšanas protokols, kuru izmanto lai šifrētu datortīklos pārsūtīto informāciju.
- CPU - centrālais procesors.
- RAM - operatīvā atmiņa.
- JavaScript – skriptēšanas valoda.
- JSON - *JavaScript Object Notation* datu apmaiņas formāts.
- Android, iOS – mobilo ierīču operētājsistēmas.
- URL – *Uniform Resource Locator*.resursa piekļuves adrese.
- PostgresSQL - bezmaksas objektu relāciju datu bāzes pārvaldības sistēma.
- JCS – *Java Cashing System* servera puses kešdarbes bibliotēka.
- Heroku – Mākoņglabātuves lietotņu platforma
- AngularJS – JavaScript ietvars tīmekļa lietotņu veidošanai.
- BASE64 – kodējuma shēma.
- Sīkdatne – dati, kuri tiek glabāti klienta tīmekļa pārlūkā.

IEVADS

REST arhitektūras stils ir tīmekļa pamatā. To izstrādāja W3C Tehniskā Arhitektūras Grupa (TAG) paralēli HTTP 1.1, kas tika bāzēts uz HTTP 1.0 eksistējošā dizaina. [9]

Ar mūsdienās pieaugošo interneta lietotāju skaitu palielinās pieprasījums pēc dažādiem servisiem un API. REST arhitektūras stils ir populārākais tieši šādiem mērķiem. Pie liela klientu skaita palielinās slodze uz servera, līdz ar to ir jāmeklē iespējas, kā maksimāli ietaupīt servera resursus, lai spētu apkalpot vairāk klientu vienlaicīgi un apkalpot tos ātrāk, tādā veidā palielinot klientu apmierinātību.

Katrai lietotnei ir nepieciešama individuāla pieeja un citādāks risinājums, ņemot vērā izmantotās platformas, ietvarus un rīkus, lai uzlabotu veiktspēju un drošību. Darbā tiks apskatīts ar kādām metodēm iespējams uzlabot REST servera veiktspēju un drošību, ka arī ar praktiskā darba palīdzību tiks parādīts, kā pielāgot REST arhitektūrā veidotus servisus Java programmēšanas valodā priekš mobilajām lietotnēm, tiks veikta servera puses kešdarbe, datu apjoma samazināšana veiktspējas uzlabošanai, un tiks ievēroti drošības pricipi un pielietota Facebook autentifikācijas, izmantojot OAuth 2.0 protokolu, kā arī tiks veikta drošības un veiktspējas analīze.

Darbs ir organizēts šādā veidā:

1. nodaļā ir aprakstīts REST arhitektūras stils, tā priekšrocības, izmantotās HTTP metodes, kā arī aprakstīts HTTP protokols, kas ir būtiska REST stila sastāvdaļa. Nobeigumā apskatītas REST alternatīvas.

2. nodaļā ir apraksīta REST arhitektūras stilā veidotu tīmekļa servisu drošības pamatprincipi un biežāk lietotie autentifikāciju veidi.

3. nodaļā ir aprakstīta veiktspēja RESTful servisiem, optimāls lietotnes projektējums, izmantojot HTTP priekšrocības, horizontālā un vertikālā mērogošana, kā arī ir aprakstīta kešdarbe klienta pusē, servera pusē un individuālas platformas kešdarbe. Nodaļas nobeigumā aprakstīts – kā ierobežot datu objektu informāciju, lai samazinātu datu plūsmu.

4. nodaļā ir aprakstīts sistēmas projektējums, kura aizmugurpuses serveris balstās uz REST arhitektūras stilu, sistēmā iekļautās lietotnes, REST resursu modelējumi, izvēlētā autentifikācijas metode, drošības un veiktspējas principi, kā arī veikto testu apraksts.

1. REST ARHITEKTŪRAS STILS

REST ir programmatūras arhitektūras stils, REST ir abrievatūra no termina „Representational State Transfer", ko varētu iztulkot kā „Attēlojošā Stāvokļa Maiņa". REST satur vadlīnijas un labāko praksi priekš mērogojamiem „web servisiem". REST tiek plaši izmantots kā vienkāršāka alternatīva SOAP un WSDL bāzētiem servisiem.

1.1. HTTP

REST balstās uz HTTP protokolu, tāpēc ir vērts apskatīt, kas tad tas īsti ir. Tieši HTTP stabilitāte un vienkāršība ir galvenais iemesls, kādēļ REST stils ir tik plaši izplatīts, un ir vadošā metode sistēmu savienošanai.

„Tutplus" blogā autors *Ludovico Fischer* [4] HTTP definē kā protokolu, kurš ļauj sūtīt dokumentus tīmeklī. Protokols ir noteikumu kopa, kura nosaka kuras ziņas var tikt apmainītas un kuras ziņas ir atbilstošas atbildes uz citām ziņām.

HTTP satur divas lomas: serveris un klients. Klients vienmēr uzsāk sarunu; serveris atbild. HTTP ir teksta pamatā; t.i. ziņas būtībā ir biti ar tekstu, lai gan ziņas pamatteksts var saturēt arī cita veida datus. Teksta formāta lietojums ļauj ērtāk monitorēt HTTP resursu apmaiņu.

HTTP ziņas tiek veidotas no galvenes un pamatteksta. Pamatteksts bieži mēdz būt tukšs; tas satur datus, kuri tiek pārraidīti caur tīklu, balstoties uz galvenes instrukcijām. Galvene satur metadatus, piemēram, kodējuma informāciju, taču pieprasījuma gadījumā tas satur arī HTTP metodes nosaukumu, piemēram GET, POST, PUT vai DELETE. REST stila lietojumā tieši HTTP metode bieži vien ir svarīgāka informācija nekā ziņojuma pamatteksts.

1.2. REST priekšrocības

REST ir vienkāršs veids kā organizēt mijiedarbību starp patstāvīgām sistēmām. REST ir strauji mantojis popularitāti kopš 2005. gada. Mūsdienās REST stils tiek plaši izmantots, piemēram sociālo tīklu API – Twitter API, Facebook API. REST ir ļoti „viegls" datu ziņā, t.i. tas nodrošina minimālo metadatu plūsmu. REST balstās uz HTTP, līdz ar to, to var izmantot arī visur tur, kur ir pieejams HTTP.

REST arhitektūras stila priekšrocības:

- Ātrdarbība

- Mērogojamība
- Vienkārši interfeisi
- Viegli veikt izmaiņas komponentēs, ja ir nepieciešamība
- Komponenšu pārnesamība var ērti pārnest kodu ar datiem.
- Uzticamība – atsevišķu komponenšu kļūdas neietekmē sistēmas kopējo darbību.

1.3. REST metodes

REST arhitektūras stils balstās uz HTTP, tāpēc REST stils nosaka šadu HTTP metožu izmantošanu:

- GET – Resursu lasīšanai
- POST – Resursu izveidošanai
- PUT – Resursu rediģēšanai, atjaunošanai
- DELETE – Resursu dzēšanai

1.4. Alternatīvas

REST alternatīvas iekļauj veidot relatīvi sarežģītas konvencijas uz HTTP. Bieži tās satur pavisam jaunu XML bāzētu valodu. Visspilgtākais piemērs ir SOAP. Ir nepieciešams apgūt pavisam jaunas konvencijas, taču HTTP netiek izmantots ar pilnu potenciālu. [4] REST, turpretīm, izmanto visas HTTP priekšrocības.

2. REST SERVISU DROŠĪBA

Katram HTTP pieprasījumam, kura sastāvā ir drošības konteksts ir jāpārbauda vai lietotājs ir tas, par ko viņš uzdodās – autentifikācija, un vai lietotājam ir atļauts darīt to, ko viņš ir pieprasījis darīt – autorizācija. Šajā sadaļā arī tiks apskatīti lietotnes drošības pamatprincipi, kurus vajadzētu ievērot, izstrādājot REST servisus.

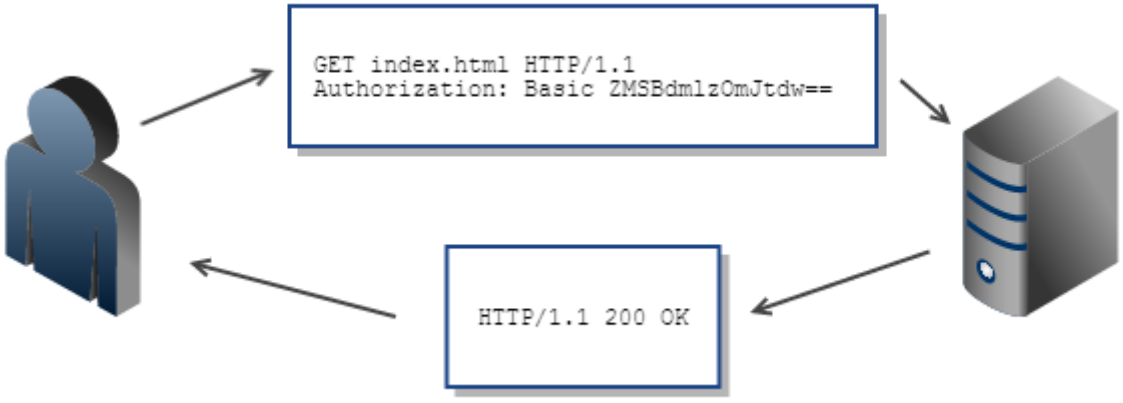
2.1. Autentifikāciju veidi

Šajā apakšnodaļā tiks apskatīti galvenie autentifikāciju veidi, kuri tiek pielietoti REST stila lietotnēs. Katrai no šīm metodēm ir savas priekšrocības un trūkumi, un kamēr vienai lietotnei vispiemērotākā būtu viena autentifikācijas metode, citai lietotnei būtu pavisam cita metode, tādēļ ir nepieciešams izvērtēt, kura metode katrai lietotnei ir piemērotākā.

2.1.1. Pamata autentifikācija (Basic authentication)

Avotā [1] Pamata autentifikācija tiek skaidrota ka HTTP transakcijas kontekstā, tā ir metode kā HTTP lietotāja aģents, piemēram tīmekļa pārlūks, nodod lietotāja vārdu un paroli pieprasījumam. Pamata autentifikācija visvienkāršākā no apskatītajām autentifikāciju metodēm, jo nav nepieciešami sīkdatnes, sesijas identifikatori nedz pieteikšanās lapa. Pamata autentifikācija izmanto statisko standarta HTTP galveni, kas nozīmē, ka nav nepieciešami papildus handshakes - izaicinājumrokasspiedieni. Tiek lietots Base64 kodējums, taču netiek lietota nekāda kriptācija vai hešošana, līdz ar to Pamata autentifikācija parasti notiek caur HTTPS protokolu.

Attēlā 2.1.1.1. redzama pamata autentifikācijas shēma.



2.1.1.1. att. **Pamata autentifikācijas shēma**

Lietotājs (klients) sūta Base64 kodētu lietotāja vārdu un paroli atdalītu ar „:” simbolu. Piemēram, ja lietotāja vārds ir „dāvis” un parole ir „bmw”, tad:

Kodējamais teksts: „dāvis:bmw”

Kodētais teksts: ZMSBdmlzOmJtdw==

HTTP galvene: Authorization: Basic ZMSBdmlzOmJtdw==

2.1.2. OAuth 1.0

Avotā [2] OAuth 1.0 tiek skaidrots sekojoši: OAuth ir atvērtais standarts priekš pilnvarošanas. OAuth nodrošina klientu lietotnēm „drošu deleģētu piekļuvi” servera resursiem resursu īpašnieka vārdā.

Pieteikties kādai lietotnei ar Google, Facebook, Twitter vai kādu citu sociālo tīklu kontu ir ne tikai ērti, tas ir arī drošāk nekā izveidot lietotnē jaunu kontu ar tādu pašu sociālā tīkla paroli.

Teiksim, ka vajag atļaut kādai lietotnei publicēt smieklīgus kaķu video kāda lietotāja Facebook profilā, taču viņš negribētu uzticēt šai lietotnei savu lietotāja vārdu un paroli un cerēt, ka tie netiks ļaunprātīgi izmantoti. Tieši šādu problēmu atrisina OAuth. Oauth atļauj noteiktai lietotnei veikt tikai tās darbības, kuras Tu esi atļāvis veikt. To varētu pielīdzināt atslēgām, Tu nedotu savu mājas atslēgu svešiniekiem ar cerību, ka nekas netiks nozagts.

2.1.3. OAuth 2.0

Avotā [5] OAuth 2.0 tiek aprakstīts šādi: Ir grūti lietot OAuth 1.0 priekš lietotnes, kura nav tīmekļa lietotne. Vēljovairāk, procedūra lai izveidotu Oauth implementācijas bibliotēku ir pārāk sarežģīta, un šī sarežģītā procedūra uzliek papildus slodzi pakalpojumu sniedzējam (Facebook, Google, Twitter, u.c.).

OAuth 2.0 uzlabo šos vājos punktus. OAuth 2.0 nav saderīgs ar OAuth 1.0 un tam vēljoprojāma nav gala versijas, taču daudzi interneta servisu uzņēmumi jau lieto OAuth 2.0.

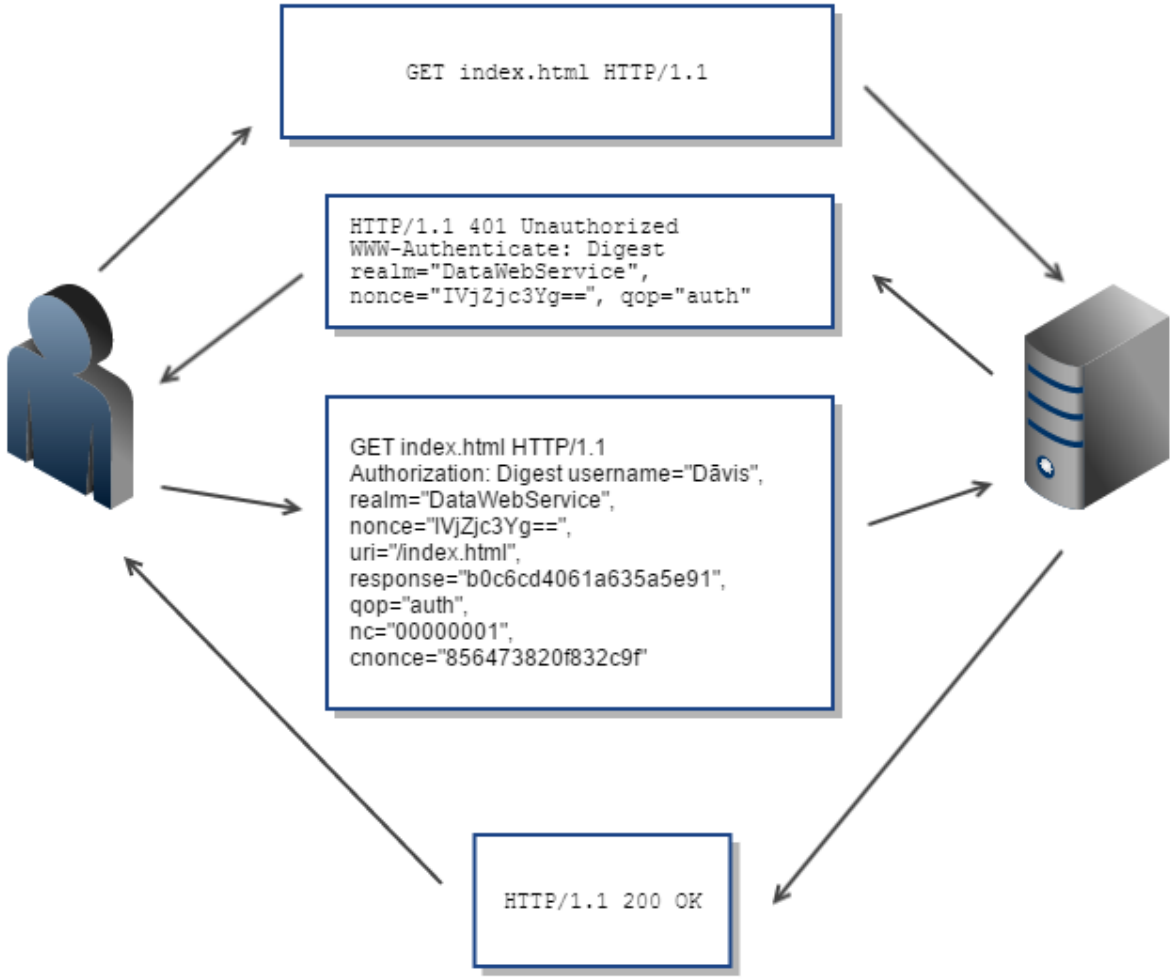
OAuth 2.0 galvenās iezīmes:

- Uzlabots lietotņu, kuras nav tīmekļa lietotnes, atbalsts
- Nav nepieciešama šifrēšana – tiek izmantots HTTPS nevis HMAC
- Vienkāršota signatūra – nav nepieciešama kārtošana un URL kodēšana
- Piekļuves pilnvaras uzlabojumi – OAuth 1.0 versijā piekļuves pilnvara bija derīga nepārtraukti. Twitter piekļuves pilnvara bija derīga mūžīgi. Drošības nolūkos OAuth 2.0 ļauj norādīt piekļuves pilnvaras derīguma laiku.

Kopumā OAuth 2.0 terminoloģija ir pavisam atšķirīga no OAuth 1.0 terminoloģijas. Abiem protokoliem ir vienāds mērķis, taču tie ir pavisam atšķirīgi.

2.1.4. Īssavilkuma autentifikācija (Digest Auth)

Rakstā „Digest Authentication on a WCF REST Service”[8] autors Patriks Kalkams skaidro īssavilkuma autentifikāciju šādi - Īssavilkuma autentifikācijas protokolā klients veic pieprasījumu serverim, atbildē serveris norāda HTTP galvenē ar kādu mehānismu resurss ir aizsargāts. Attēlā 2.1.4.1. redzama īssavilkuma autentifikācijas shēma.

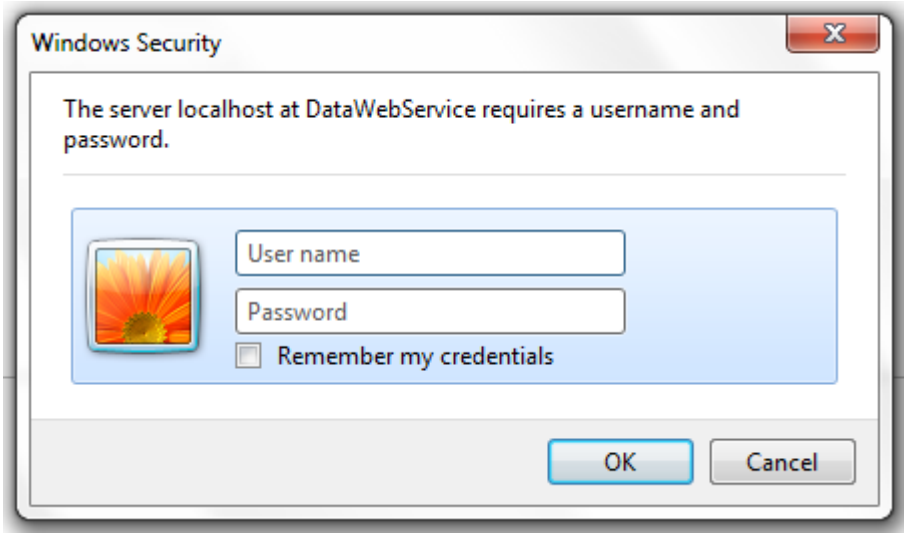


2.1.4.1. att. **Īssavilkuma autentifikācijas shēma**

HTTP galvene satur "WWW-Authenticate: Digest realm="DataWebService", nonce="IVjZjc3Yg==", qop="auth". Šeit, pirmkārt, serveris norāda, ka resurss tiek aizsargāts, izmantojot Digest (īssavilkuma) autentifikācijas metodi. Otrkārt, serveris norāda, ka jāizmanto *qop* „quality of protection” īssavilkuma autentifikācijas algoritms, realm=”DataWebService”

norāda kādam resursam serveris prasa paroli. Pēdējais elements ir teksta lauks *nonce,* kuru uzģenerē serveris un nokodē BASE64 kodējumā, un tas lietotājam būs jāiekļauj atbildes hešā.

Pēc tam lietotājs pārlūka saņems dialogu, kurā lietotājam jāievada lietotāja vārds un parole (Attēlā 2.1.4.2.).



2.1.4.2. att. **Īssavilkuma autentifikācijas dialogs**

Kad lietotājs ievada pieprasīto informāciju, pārlūks pieprasa resursu no servera vēlreiz, tikai šoreiz klients norāda papildus īssavilkuma autentifikācijas informāciju HTTP galvenē, skatīt attēlā 2.1.4.1. Vispirms galvenē parādās „Authorization: Digest", kas norāda autorizācijas mehānismu, tālāk seko „username="Dāvis"", kas norāda lietotājvārdu, *nonce* paliek nemainīgs no saņemtā, „url=/index.html" pieprasījuma URL, *qop* paliek nemainīgs no saņemtā, „nc="0000001"" ir *nonce* skaitītājs, kurš palielinās pēc katra pieprasījuma, *cnonce* ir klienta pusē ģenerēts nonce, kurš tiek iekļauts hešā, un svarīgākais lauks *response* ir heša vērtība, kura sastāv no:

- Hash1 = MD5(leatotājvārds:realm:parole)
- Hash2 = MD5(metode:DigestURI)
- Response = MD5(Hash1:nonce:nonceCount:clientNonce:qop:Hash2)

Pēc tam serveris veic tādus pašus MD5 hešus ar pieprasījuma lietotājvārdu un tā īsto paroli, un salīdzina vērtības, ja tās sakrīt, tad atgriež pieprasīto resursu klientam.

2.2. Drošības pamatprincipi

Šajā nodaļā tiks apskatīti pamatprincipi, kuri būtu jāievēro izstrādājot tīmekļa servisus REST stila arhitektūrā. Autori Erlends Oftedals un Endrjū van der Stoks rakstā OWASP

„Security cheat sheet” [7] ir apkopojuši svarīgākos pamatprincipus RESTful serivsu drošībai. Tie iekļauj:

- Autentifikāciju un sesijas pārvaldību
- HTTP metodes
- Atļautās metodes
- CSRF un XSS
- Tiešas piekļuves objektu darbības
- Ievaddatu validāciju
- Izvaddatu kodēšanu
- HTTPS

Nākamajās apakšnodaļās šie principi tiks aprakstīti detalizētāk.

2.2.1. Autentifikācija un sesijas pārvaldība

RESTful tīmekļa servisiem vajadzētu lietot sesijas bāzētu autentifikāciju ar sesijas pilnvaru caur POST vai, lietojot API atslēgu kā POST pamatteksta argumentu, vai kā sīkdatni.

Lietotājvārdiem, parolēm, sesijas pilnvarām, un API atslēgām nevajadzētu parādīties iekš URL adreses, jo tas var tikt pārtverts tīmekļa servera žurnālā.

Pieņemami:

- „https://example.com/resourceCollection/{id}/action”
- „https://twitter.com/vanderaj/lists”

Slikti:

- „https://example.com/controller/<id>/action?apiKey=a53f435643de32” (API Atslēga iekš URL adreses)
- „http://example.com/controller/<id>/action?apiKey=a53f435643de32” (transakcija nav pasargāta ar TLS – nav HTTPS; API atslēga iekš URL adreses)

REST stils paredz veidot servisus bez stāvokļa attēlošanas, taču dažkārt to ir nepieciešams darīt. Bieži tas tiek atrisināts sūtot stāvokli kā milzīgs teksta blāķi kā daļu no transakcijas.

Labāka pieeja ir sūtīt sesijas pilnvaru vai API atslēgu, lai saglabātu klienta stāvokli servera puses atmiņā, tādā veidā paaugstinot drošību, jo stāvoklis nevar tikt pārtverts, un izvairoties no mērogošanas problēmām.

„Atkārtojuma uzbrukuma” gadījumā „uzbrucējs” varētu iekopēt teksta daļu un izlikties par kādu citu, taču, lietojot piekļuves atslēgu, no tā var izvairīties, jo tas ir derīgs tikai uz noteiktu laika periodu.

Vēl nevajadzētu veikt konfogurāciju ar URL parametriem, piemēram: „https://example.com/users/2313/edit?isAdmin=false&debug=false&allowCSRPanel=false” kā rezultātā serverī var parādīties daudz jaunu „administratoru”.

2.2.2. HTTP metodes

RESTful API pārsvarā lieto GET (lasīšana), POST (izveide), PUT (rediģēšana) un DELETE (dzēšana) metodes. Ne visas metodes ir pieļaujamas katrai resursu kolekcijai, lietotājam, vai darbībai. Serveriem vienmēr vajadzētu pārbaudīt vai ienākošā HTTP metode ir derīga piekļuves pilnvarai vai API atslēgai un vai tā ir atļauta attiecīgajam resursam, darbībai vai ierakstam. Piemēram, RESTful API priekš bibliotēkas – nevajadzētu ļaut anonīmam lietotājam atļaut dzēst grāmatu kataloga ierakstus ar DELETE metodi, taču tie pavisam noteikti varētu apskatīt tos ar GET metodi. No otras puses bibliotekāram abas šīs metodes varētu būt pieļaujamas.

2.2.3. Atļautās metodes

RESTful servisiem var būt vairākas metodes vienai URL adresei dažādām darbībām uz atiecīgo elementu. Piemēram, GET pieprasījums varētu nolasīt elementu, PUT metode varētu rediģēt eksistējošu elemetnu, POST metode varētu izveidot jaunu elementu, un DELETE metode varētu dzēst eksistējošu elementu. Ir svarīgi servisam pareizi atšķirt atļautās metodes no neatļautajām, kā arī veikt autorizācijas pārbaudi. Uz neatļautām darbībām atgriezt HTTP statusu „403 Forbidden” vai „404 Not Found”.

2.2.4. CSRF un XSS

RESTful servisu resursu PUT, POST un DELETE metodēm vajadzētu būt pasargātām no „Cross Site Request Forgery” uzbrukumiem – uzbrukumi, kuros lietotājs nevērības rezultātā izsauc kādu pieprasījumu uz kādu servisu, kurā pārlūks ir saglabājis autentifikācijas informāciju, piemēram lietotājs nospiež uz pogas, kura veic izsaukumu uz „http://bank.example.com/transfer?account=Alice&amount=9999&for=Bob” un Alise neapzinoties ir pārskaitījusi naudu Bobam, jo tīmeklis bija saglabājis Alises konta paroli pārlūka sīkdatnē. Lai izvairītos no šī parasti pietiek ar pilnvaras bāzētu pieeju, jo pilnvarai beidzas derīguma termiņš.

CSRF var tikt panākts pat arī tad, ja tiek izmantojot nejauši ģenerētas pilnvaras, gadījumā, ja lietotnē eksistē kāds XSS – skripts, kurš ir injicēts no kāda lietotāja uz servera var veikt dažādas nevēlamas darbības, kā piemēram nozagt lietotāja sīkdatnes. XSS izmanto jau iepriekš iegūtu pieejas pilnvaru, lai veiktu nevēlamas darbības. Lai izvairītos no XSS serverim ir jāfiltrē ievaddati un izvaddati.

2.2.5. Tiešas piekļuves objektu darbības

Tas var likties pašsaprotami, taču, piemēram, bankas kontu RESTful tīmekļa servisam vienmēr būtu adekvāti jāveic autorizācijas un autentifikācijas pārbaudes: „https://example.com/account/325365436/transfer?amount=$100.00&toAccount=47384676". Šādā gadījumā būtu iespējams pārskaitīt naudu no jebkura konta uz jebkuru citu kontu. „https://example.com/invoice/2362365". Šādā gadījumā būtu iespējams iegūt visas pavadzīmes.

Lai izvairītos no šādiem gadījumiem vienmēr nepieciešams veikt severa puses pārbaudes un nekad nepaļauties uz klienta puses validāciju.

2.2.6. Ievaddatu validācija

Klienta puses validācija ir tikai palīgs klientam, taču to ir viegli apiet, tādēļ uz to nevar paļauties. Vienmēr vajag veikt arī servera puses validāciju katram laukam, lai izvairītos no SQL injekcijas uzbrukuma, kā arī nepieciešams nodrošināt korektu izvades kodējumu, lai izvairītos no XSS uzbrukuma.

Laba ideja ir žurnalēt ievades validāciju kļūdas, īpaši tad, ja klienta puses kods veic izsaukumu uz tīmekļa servisiem. Servisa izsaukumu var veikt ne tikai no formas, kurā tas ir paredzēts, līdz ar to, ja kādam klientam simtiem reižu sekundē ir validācijas kļūdu, tas varētu liecināt par ievainojamību meklēšanu. No tā varētu izvairīties limitējot servisu saskarni līdz noteiktam pieprasījumu daudzumam stundā vai dienā, lai izvairītos no uzbrukumiem.

Noteikti vajadzētu izmantot drošu parsētāju ienākošajiem tekstiem parsējot JSON, XML un citu formātu tekstus.

Ir grūti veikt vairums uzbrukumu, ja atļautās vērtības ir patiess un aplams vai skaitlis. Tādēļ vajag limitēt ienākošo datu formātu, piemēram, ja nepieciešams atlasīt recepti pēc id vērtības, kas ir „Long" vērtība, t.i. skaitlis ar vērtību no -2^{63} līdz 2^{63}, tad nevajadzētu atļaut ievadīt tekstuālu vērtību tā vietā.

Pie POST un PUT pieprasījumiem klients norādīs satura tipu (Content-Type), piemēram „application/xml” vai „application/json” sūtāmajiem datiem. Serverim nekad nevajadzētu izdomāt satura tipa vērtību, vienmēr vajag salīdzināt satura tipa vērtību ar pieļaujamo un vai saturs tiešām atbilst apsolītajam, citādi serverim vajadzētu atgriezt „406 Not Acceptable” statusa atbildi.

2.2.7. Izvaddatu kodēšana

Lai pārliecinātos, ka pārlūks vai lietotne ir interpretējusi pareizi dotā resursa saturu, serverim vienmēr vajadzētu sūtīt satura tipa (Content-Type) galveni ar pareizo satura tipu, un, vēlams, satura tipa galvenei vajadzētu iekļaut „charset” parametru, lai norādītu pareizo kodējumu. Serverim papildus vajadzētu sūtīt „X-Content-Type-options: nosniff” galvenē, lai pārlūks nemēģinātu uztvert citādāku satura tipu, kas varētu novest pie XSS ievainojamības. Papildus klientam vajadzētu sūtīt „X-Frame-Options: deny” galvenē, lai aizsargātos pret „drag'n drop clickjacking” uzbrukumiem vecākos pārlūkos.

2.2.8. HTTPS

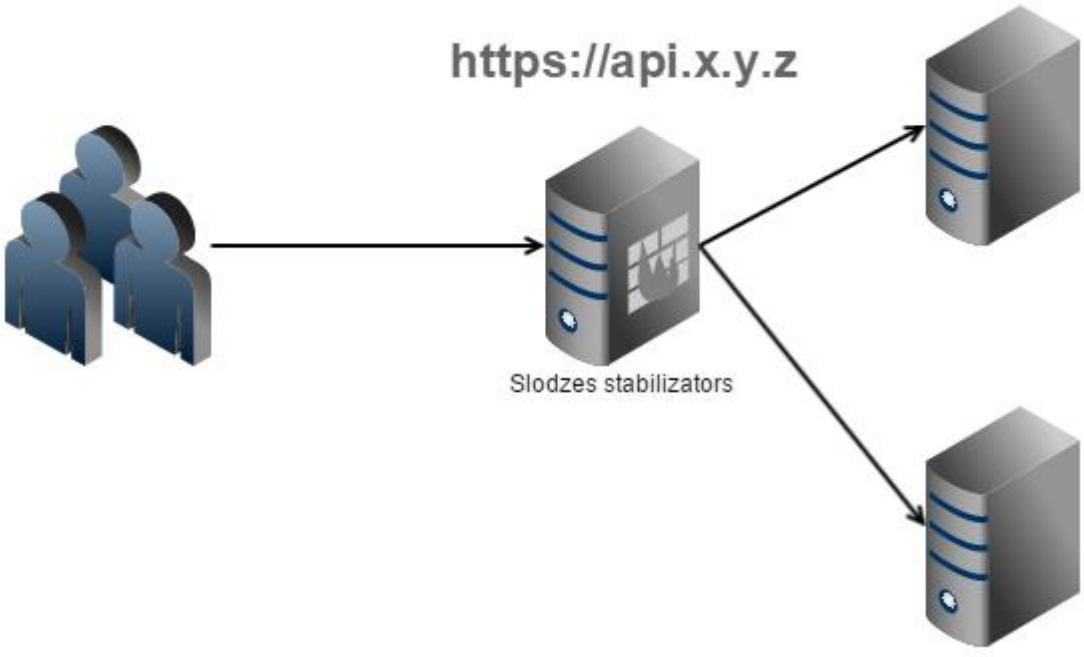
Ja publiskā informācija nav tikai lasāma, vajadzētu izmantot TLS, īpaši gadījumos, kad tiek sūtīta sensitīva informācija, piemēram, lietotāja vārds, parole, vai jebkāda datu rediģēšana, dzēšana. Papildu resursu resursu patēriņš priekš TLS ir niecīgs priekš mūsdienu tehnoloģijām, un tādā veidā klientu datu drošība tiek būtiski pastiprināta.

3. VEIKTSPĒJA

Lai lietotāji būtu apmierināti ar servisu, tam ir nepieciešams strādāt ātri, t.i. šajā gadījumā sniegt ātru atbildi uz lietojumprogrammas saskarnes izsaukumiem. Līdz ar to ir nepieciešams izvēlēties vispiemērotāko arhitektūras projektējumu un piemērotākās metodes servisu darbībai, lai tie spētu atbildēt uz klientu pieprasījumiem pēc iespējas ātrāk, kā arī jāņem vērā serveru pieejamība, t.i. serveriem ir jābūt pieejamiem 24 stundas diennaktī.

3.1. REST lietotnes projektējums izmantojot HTTP

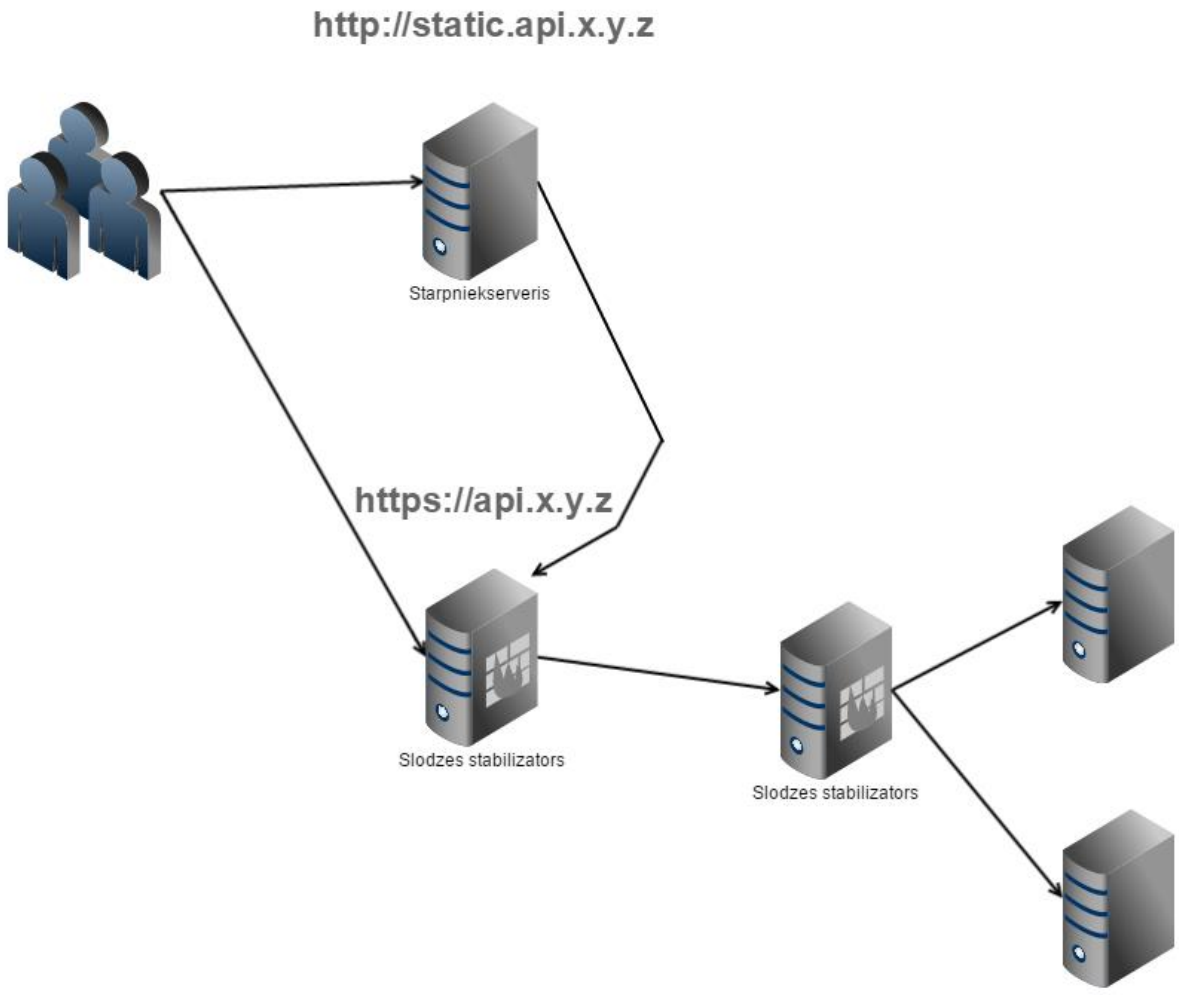
Vienkāršs REST lietotnes saskarnes projektējums, kas izmanto HTTP, bet neizmanto to efektīvi, varētu izskatīties šādi – viens ieejas punkts, šajā gadījumā „https://api.x.y.z", kas izmanto HTTPS priekš visiem pieprasījumiem un balstās uz slodzes stabilizatora, kurš uzņem klienta pieprasījumu un tālāk izvēlās kādu vienu no serveriem, kurš šo pieprasījumu apstrādās. Slodzes stabilizators nodrošina mērogošanu – varam viegli pievienot jaunus serverus, lai spētu apkalpot vairāk klientu vienlaicīgi, un kļūmjpārleci (*failover*) – lai viena servera „nobrukšanas" gadījumā klients tiktu apkalpots uz kāda cita servera. Shēma redzama attēlā 3.1.1.



3.1.1. att. **Vienkāršs REST lietotnes projektējums, izmantojot HTTP**

Šīs shēmas vājie punkti ir pie augšējās slodzes robežas, t.i. ļoti daudz klientu pieprasījumiem vienlaicīgi klientiem var nākt gaidīt rindā, jo viens slodzes stabilizators nespēj pietiekami ātri visus klientus apkalpot.

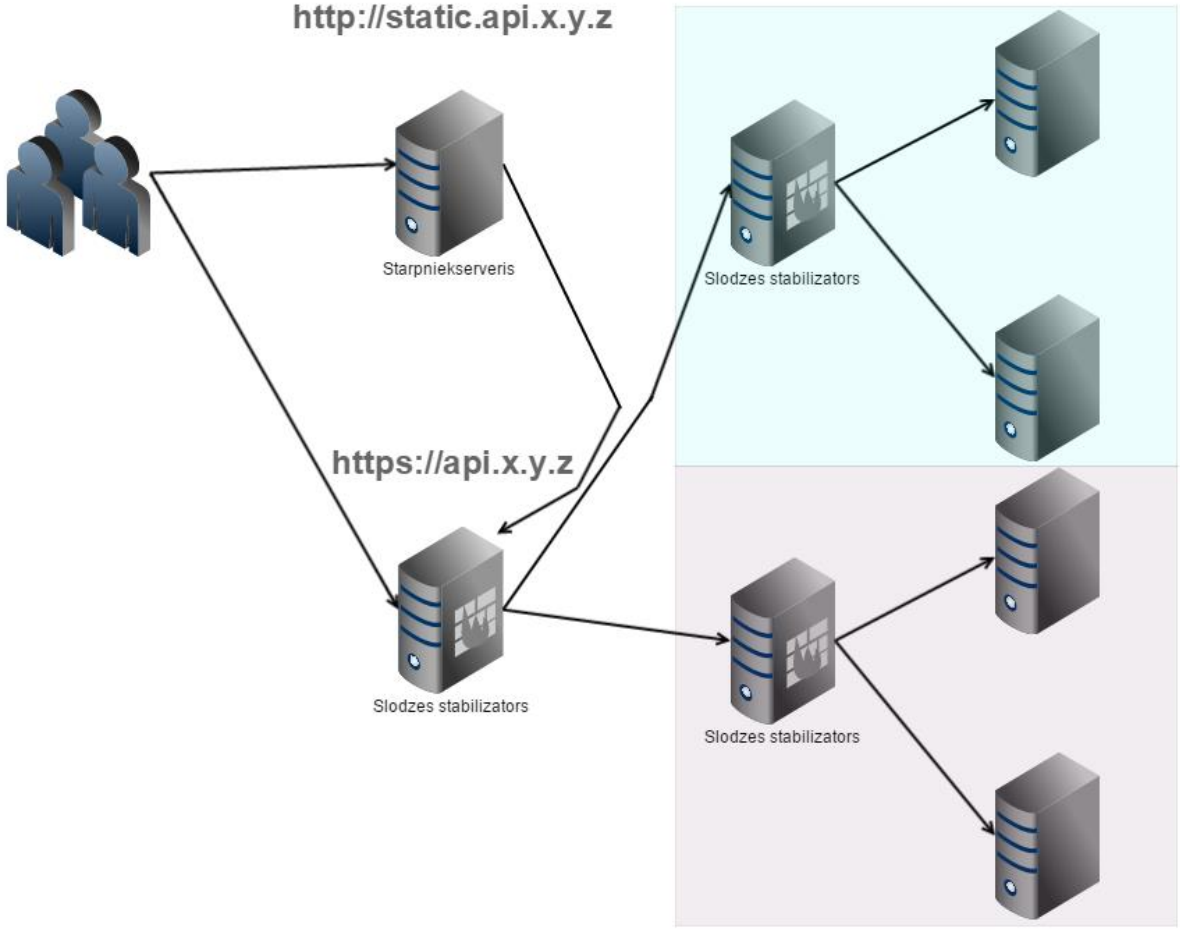
Nedaudz šo shēmu uzlabojot var panākt efektīvāku sistēmu, kura izmanto vairākus piekļuves punktus, kur statiskus datus, tādus, kuri ir vienādi visiem lietotājiem (piemēram daudzi GET pieprasījumi), var apkalpot starpniekserveris, kurš saglabā datus atmiņā no API servera, un atjauno tos, piemēram, reizi minūtē. Tādā gadījumā, ja būtu 1000 klientu pieprasījumi katru sekundi, tad minūtē būtu 60000 pieprasījumi, un uz API servera būtu tikai viens pieprasījums no starpniekservera uz 60000 klientu pieprasījumiem. Šāda shēma var būtiski uzlabot veiktspēju, jo API serverim būtu daudz mazāka slodze. Transakciju izsaukumi (POST, PUT, DELETE) lieto API serveri, kurš izmanto HTTPS un izmanto otru slodzes stabilizatora līmeni. Shēma redzama attēlā 3.1.2.



3.1.2. att. **Efektīvāks REST lietotnes projektējums, izmantojot HTTP**

Vēl labāks variants būtu izmantot vairākus datu centrus (dažādas fiziskas vietas), un stabilizēt slodzi, tas dod daudz lielāku uzticamību, jo sistēma var turpināt darboties, pat ja datu

centrs nobrūk, piemēram, kāda dabas katastrofa vai tamlīdzīgi. Šādi var arī pielāgot veiktspēju, lai klientus apkalpo tam tuvākie serveri, nevis no otra pasaules gala. Shēma attēlota attēlā 3.1.2.



3.1.2. att. **Efektīvāks REST lietotnes projektējums ar vairākiem datu centriem**

3.2. Vertikālā mērogošana

Lai uzlabotu servera veiktspēju, pavisam vienkārši, var uzlabot servera komponentes, piemēram CPU un RAM, lai viens servera mezgls dalītā sistēmā spētu apkalpot vairāk klientus un apkalpot tos ātrāk.

Šī metode parasti ir dārga un ir kaut kāda augšējā robeža, kurus mūsdienu tehnoloģijas neļauj pārsniegt, t.i. nevaram uzlabot serveri neierobežoti.

3.3. Horizontālā mērogošana

Lai uzlabotu servera veiktspēju var izveidot vairākus servera mezglus, t.i. pie viena servera sistēmas pievienot vairākus serverus un piesaistīt slodzes balansētāju, lai sadalīt slodzi pa visiem serveriem, tādā veidā spētu apkalpot vairāk klientus.

3.4. Kešdarbe

Gandrīz visam internetam GET pieprasījumi ir būtiski vairāk kā cita veida pieprasījumi kopā, līdz ar to var ļoti ievērojami uzlabot veiktspēju ar kešdarbes palīdzību, saglabājot iepriekš izsauktā pieprasījuma atbildi kešatmiņā, zinot, ka resursa vērtība netiks mainīta var neveikt vēlreiz to pašu izsaukumu, bet atgriezt vērtību no iekšējās atmiņas.

3.4.1. Klienta puses kešdarbe

Klienta puses kešdarbe iedalās divās kategorijās:

- Derīguma termiņš – klients iegūst resursu no REST servera ar galvenē pievienotu derīguma termiņa lauku „Expires: {datums un laiks}", un ja tiek veikts tāds pats pieprasījums pirms termiņa beigām, tad pieprasījums neaiziet līdz serverim, bet tiek paņemta lokālā kopija. Pēc derīga termiņa beigām veikts pieprasījums atkal veic pieprasījumu REST serverim. Derīgums termiņu var norādīt arī ar „Cache-Control: max-age={laiks sekundēs}" galvenes vērtību, kur tiek norādīts cik ilgi saņemtais resurss ir derīgs kā lokālā kopija.
- *E-Tag* un *Last-Modified* – šī metode veic pieprasījumu serverim ar „If-none-match: „4jhn6b6"" lauku galvenē, kas kalpo kā versijas numurs attiecīgajam resursam. HTTP nav definēts kā ģenerēt *E-Tagu*, tādēļ katrs serveris to var implementēk citādāk, parasti tā ir kaut kāda heša funkcija. Tiek veikts pieprasījums serverim, serveris pārbauda vai *E-Tag* sakrīt, ja nesakrīt, tad nosūta „200 OK" atbildi ar pieprasīto resursu atbildes pamattekstā, ja *E-Tag* sakrīt, tad serveris atbild ar „304 Not modified" atbildi.

3.4.2. Servera puses kešdarbe

Katrai platformai, ietvaram un programmēšanas valodai ir savs izvēļu klāsts ar kešdarbes bibliotēkām, katrai ir savi plusi un mīnusi, bet pārsvarā visām galvenais mērķis ir saglabāt atmiņā objektus, parasti kādā asociatīvā datu struktūrā, piemēram, heštabulā atslēgas un vērtību pārus, un vēlāk tās atlasīt neveicot datubāzes izsaukumus. Daži piemēri būtu memcached, ehcache, JCS.

3.4.3. Individuālas platformas kešdarbes alternatīvas

Katrai platformai var papildus lietot individuālās patformas rīkus priekš klienta puses kešdarbes, piemēram Android platformai ir Volley bibliotēka, iOS platformai Core Data.

3.5. Datu apjoms

Kā pieprasījuma atbilde bieži tiek sūtīts statusa kods „200 OK" un atbildes pamattekstā ir XML vai JSON formāta objekts vai objektu kolekcija. Dažkārt klientam nav nepieciešams viss objekts ar apakšobjektiem, taču ir nepieciešama tikai pamatinformācija par objektu. Piemēram, ja klients vēlas sarakstu ar cilvēkiem no sociālā portāla, tad tam ir nepieciešams nosaukums un apraksts, bet nav vajadzīgi tādi lauki kā draugu saraksts, mīļāko filmu saraksts, u.c. Protams klients var vienkārši šos laukus ignorēt, taču tie tāpat tiek sūtīti pie katra pieprasījuma, tādā veidā tiek sūtīti nevajadzīgi dati un patērēta datu plūsma, kas ir īpaši svarīga mobilajām iekārtām.

No tā var izvairīties veidojot DTO (*Data Transfer Object*) – datu apmaiņas objektus, kuri satur tikai pašu nepieciešamaķo informāciju, tādā veidā ietaupot datu plūsmu, ielādes ātrumu, operatīvo atmiņu, kā arī dati kļūst pārskatāmāki, un izstrādātājiem ir vieglāk orientēties objektu saturā.

4. SISTĒMAS PROJEKTĒŠANA

Šīs nodaļas mērķis ir ar praktiskā piemēra palīdzību parādīt, kādā veidā drošības un veiktspējas principus var piemērot REST stila arhitektūrā veidotiem servisiem. Kā piemērs ir izvēlēta „Cookster" mobilās lietotnes aizmugursistēma. Projekts sastāv no 3 daļām - REST serveris, administratora panelis (kas atrodas uz tā paša servera) un iOS mobilā lietotne, kā arī ir plānots izstrādāt Android lietotni nākotnē. Projektā piedalās 3 cilvēki. Komanda sastāv no idejas galvenā autora kā projekta vadītāja, iOS izstrādātāja, REST servera un administratora paneļa izstrādātāja. Šī darba autors ir atbildīgs par REST servera un administratora paneļa izstrādi. Nākotnē ir plānota papildu funkcionalitāte, tādēļ ne administratora panelis un REST serveris vēl nav pilnībā pabeigts, un nākotnē tiks pilnveidoti. Šajā nodaļā detalizētāk tiks aprakstīta REST servera izstrāde.

Visi testi tika veikti, izmantojot Loader.io [6] rīku.

4.1. Programmatūras pārskats

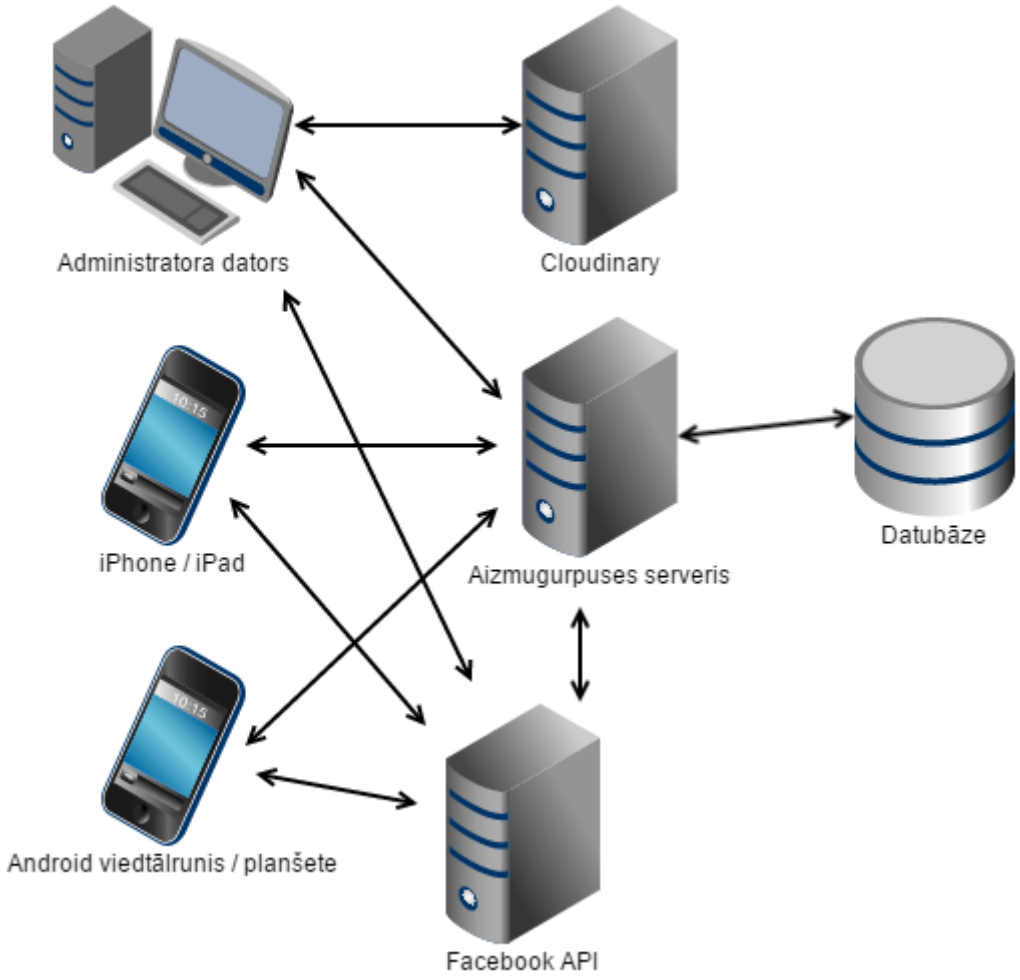
„Cookster" mobilās lietotnes sistēma sastāv no 3 galvenajām daļām:

- Aizmugurpuses serveris un datubāze– serveris uz Heroku mākoņa, uz kura stāv REST tīmekļa servisi, administratora panelis un PostgresSQL datubāze.
- Administratora panelis – lietotne, kur administratori var rediģēt datubāzes saturu un visu to, kas tiek attēlots mobilajā lietotnē.
- Mobilā lietotne – lietotājiem publiski pieejama mobilā lietotne iOS platformā un nākotnē plānots izstrādāt arī Android lietotni.

Sistēma izmanto ārējos servisus:

- Cloudinary – bilžu augšupielādēšanas un uzturēšanas serviss (datubāzē glabājas tikai bilžu URL adreses un publiskie bilžu ID).
- Facebook API – serverim nav iekšējā reģistrēšanās sistēma, tiek izmantota Facebook autentifikācija, un datubāzē glabājas tikai lietotāja Facebook ID un publiskie dati (bez jebkāda lietotājvarda un paroles).

Attēlā 4.1.1. ir redzama sistēmas virspusējās struktūras shēma.



4.1.1. att. **Sistēmas virspusējā struktūra**

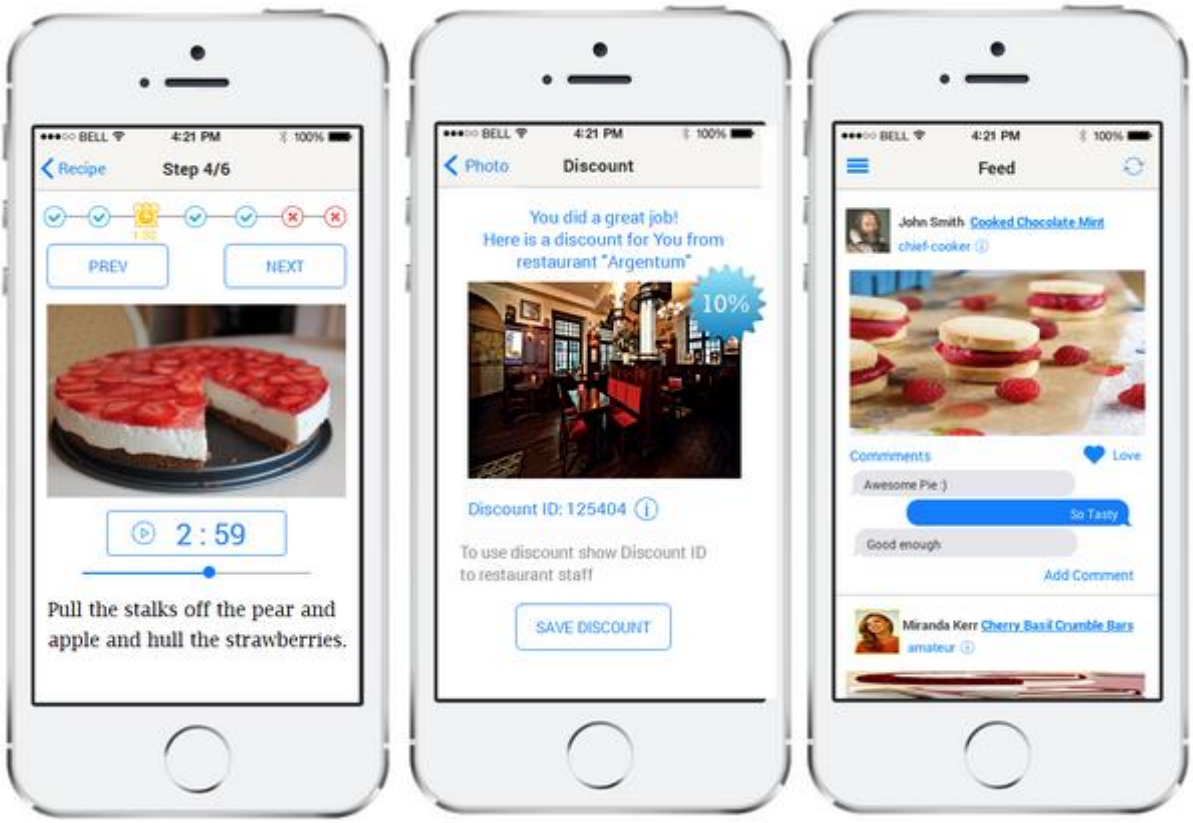
Tālāk tiks apskatītas sistēmas galvenās daļas.

4.2. Mobilā lietotne

„Cookster" mobilā lietotne ir paredzēta, lai mācītos pagatavot dažādas receptes ar ērtu soļu bāzētu tehniku, lai lietotājiem būtu skaidri saprotams ko un kā darīt. Darba tapšanas laikā šī lietotne ir tikai prototipa stadijā iOS operētājsistēmai. Pēc darba tapšanas paredzēts to pabeigt līdz galam un izstrādāt arī Android lietotni.

Lietotājam atverot lietotni būs redzama pieteikšanās forma, taču lietotni var lietot arī nepiesakoties, pieteikšanās ļauj saglabāt mīļākās receptes un sniegt komentārus, kā arī dalīties ar receptēm Facebook portālā. Sākumlapā lietotājam redzama plūsma ar jaunākajām receptēm, tālāk lietotājs var izvēlēties kategoriju vai meklēt sev tīkamo recepti meklēšanas formā, kur var

meklēt recepti arī pēc pieejamajām sastāvdaļām. Kad recepte ir atvērta tiek parādīta informācija par autoru un tālāk seko apraksts par recepti un tam nepieciešamās sastāvdaļas, kad lietotājs ir gatavs, viņš var nospiest pogu „Start" lai sāktu gatavot, katrā solī ir aprakstīts un parādīts, kas ir jādara, solis var saturēt taimeri, un lietotājs var ērti staigāt pa soļiem, dažkārt kāds solis var tikt uzlikts uz taimera un lietotājs var turpināt nākamos soļus, kamēr iepriekšējā solī kaut kas, piemēram, vārās, tikmēr veicot citus darbus. Recepti pabeidzot lietotājs var nofotografēt rezultātu un padalīties ar draugiem Facebook sociālajā portālā. Lietotnes maketi redzami attēlā 4.2.1.

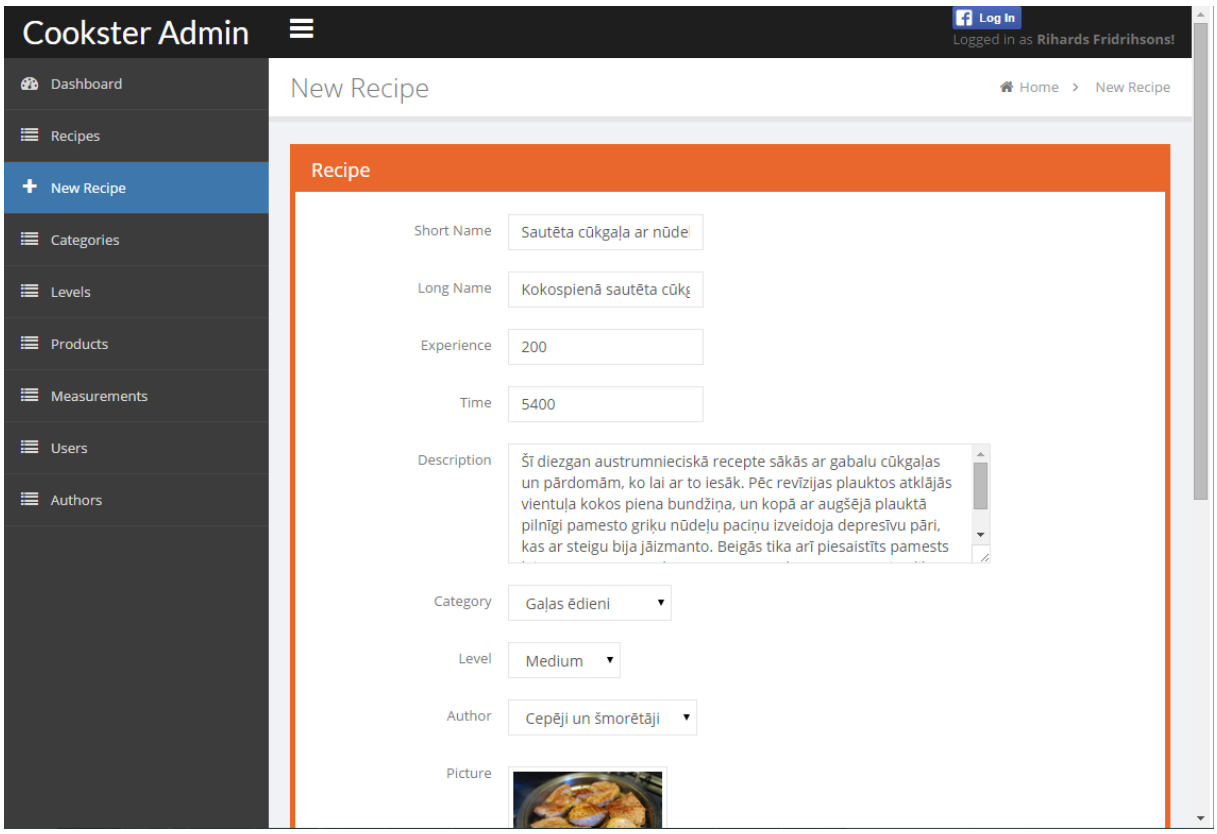


4.2.1. att. **„Cookster" iOS mobilās lietotnes maketi**

4.3. Administratora panelis

Administratora panelis satur nepieciešamos rīkus (formas) resursu izveidošanai, rediģēšanai un dzēšanai. Formas nodrošina ērtu objekta veidošanu – no formās ievadītajām vērtībām tiek izveidots JSON objekts, un tad tas tiek tālāk nosūtīts uz REST servera. Šāda metode tiek izmantota gan resursu izveidošanai gan rediģēšanai.

Administratora lietotne veidota uz AngularJS ietvara JavaScript skriptēšanas valodā. Attēlā 4.3.1. redzams administratora paneļa ekrānuzņēmums.



4.3.1. att. **„Cookster" administratora panelis**

Administrators, izmantojot jebkuru ierīci ar tīmekļa pārlūku, var piekļūt administratora panelim, kur var veikt tabulā 4.5.1. atrodamās funkcijas ar lietotājam ērtu saskarni.

4.4. REST serveris

REST serveris ir programmatūra, kura tiks apskatīta detalizētāk šajā darbā.
Lietotne sastāv no 3 lomām:

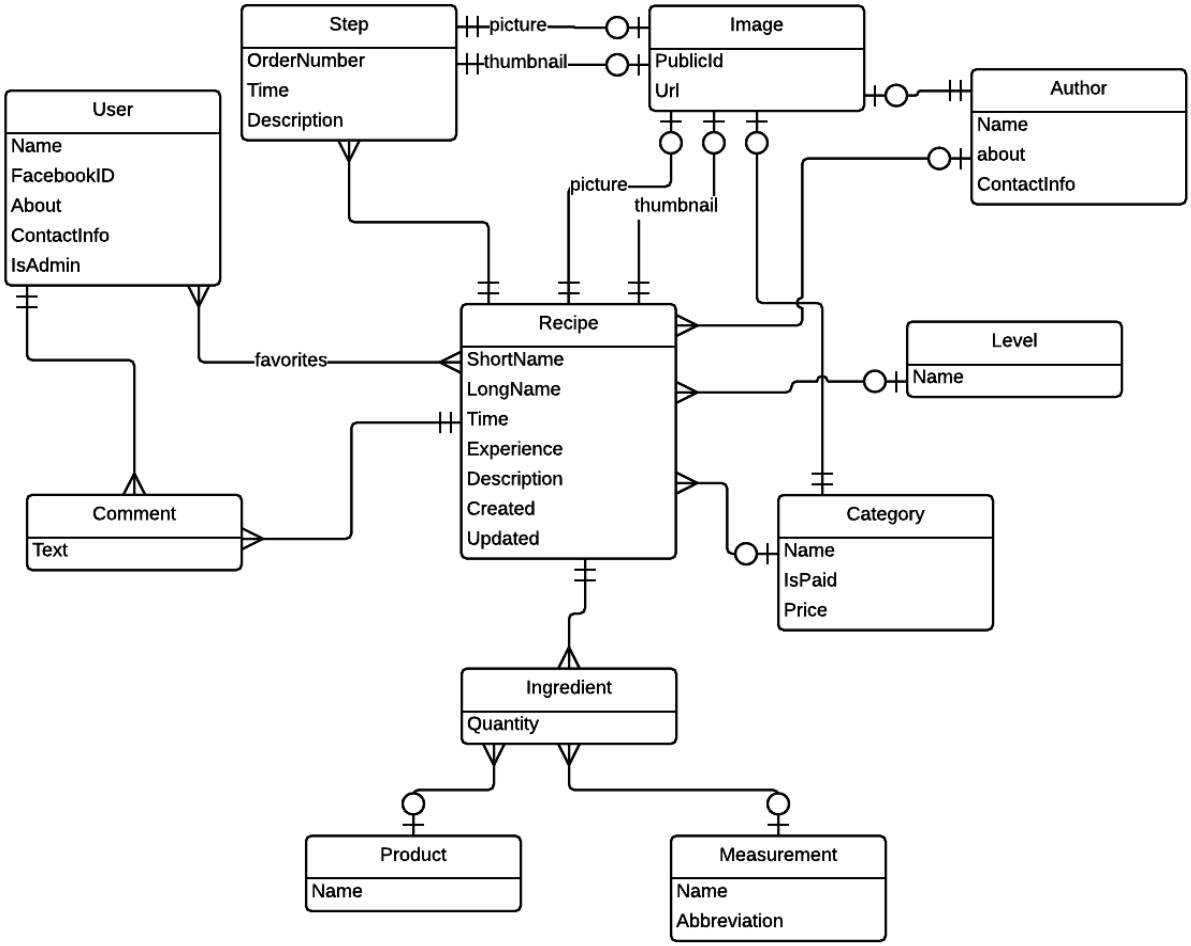
- Neautentificēts lietotājs
- Autentificēts lietotājs
- Administrators

Neautentificētie lietotāji var veikt visus GET pieprasījumus, t.i. skatīt receptes, lietotāju profilus meklēt receptes, autentificēti lietotāji papildus var saglabāt receptes pie favorītiem, veikt komentārus un dalīties ar nofotografēto, pabeigto recepti.

REST serveris ir veidots Java programmēšanas valodā uz Jersey ietvara un servera puses kešdarbei tiek izmantota ehcache bibliotēka. Tiek izmantots GitHub repozitorijs versiju kontrolei. Lietotne ir uzstādīta un laista no Heroku mākoņglabātuves, tā izmanto PostgreSQL datubāzi, kura arī atrodas uz Heroku mākoņa, tas tika izvēlēts, jo ir bezmaksas labs variants un

ērti savienojams ar GitHub repozitoriju un Jersey ietvaru. Bildes tiek glabātas uz Cloudinary mākoņa. Autentifikācija notiek izmantojot OAuth 2.0 un Facebook API.

Attēlā redzms datubāzes konceptuālais ER modelis.



4.1.3.1. att. **Konceptuālais ER modelis**

4.5. Resursu modelēšana - administratora funkcijas

Administratora funkcijas ir nodalītas ar „/admin" prefiksu URL adresē. Tabulā 4.5.1 redzamas administratoram pieejamās funkcijas.

4.5.1. tabula **Administratora funkcijas**

	URL	Apraksts
POST	/admin/categories	Jaunas kategorijas izveide
PUT	/admin/categories/{id}	Kategorijas rediģēšana
DELETE	/admin/categories/{id}	Kategorijas dzēšana
POST	/admin/authors	Jauna autora izveide

PUT	/admin/authors	Autora rediģēšana
DELETE	/admin/authors/{id}	Autora dzēšana
POST	/admin/ingrdients	Jaunas sastāvdaļas izveide
DELETE	/admin/ingrdients/{id}	Sastāvdaļas dzēšana
POST	/admin/measurements	Jaunas mērvienības izveide
DELETE	/admin/measurements/{id}	Mērvienības dzēšana
POST	/admin/products	Jauna produkta izveide
DELETE	/admin/products/{id}	Produkta dzēšana
POST	/admin/recipes	Jaunas receptes izveide
PUT	/admin/recipes/{id}	Receptes rediģēšana
DELETE	/admin/recipes/{id}	Receptes dzēšana
DELETE	/recipes/{id}/comments/{id1}	Receptes komentāra dzēšana

4.6. Resursu modelēšana – lietotājs

Lietotājiem pieejamās funkcijas apskatāmas tabulā 4.6.1. Pēc katras veiksmīgas Facebook autentifikācijas lietotājs tiek reģistrēts sistēmā, ja viņš iepriekš nav bijis reģistrēts.

4.6.1. tabula **Lietotāja funkcijas**

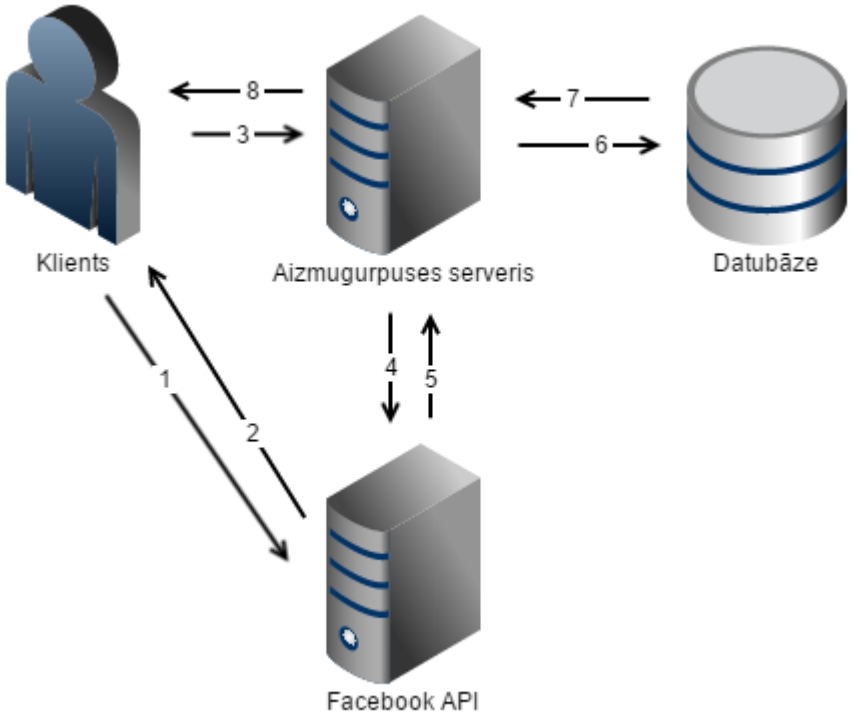
	URL	Apraksts
GET	/authors	Autoru saraksts
GET	/authors/{id}	Autora profils
GET	/authors/{id}/recipes	Autora recepšu saraksts
GET	/categories	Kategoriju saraksts
GET	/categories/{id}/recipes	Recepšu saraksts kategorijā
GET	/levels	Pagatavošanas sarežģītības līmeņu saraksts
GET	/measurements	Mērvienību saraksts
GET	/products	Produktu saraksts
GET	/recipes	Recepšu saraksts
GET	/recipes/{id}	Recepte
GET	/recipes/{id}/steps	Receptes soļu saraksts
GET	/recipes/{id}/comments	Receptes komentāri
POST	/recipes/{id}/comments	Komentēt recepti (Jābūt autentificētam)
GET	/users	Lietotāju saraksts

GET	/users/{id}	Lietotāja profils
GET	/users/{id}/favorites	Lietotāja favorīti
POST	/users/{id}/favorites	Saglabāt favorītu (Jābūt autentificētam)
DELETE	/users/{id}/favorites/{id1}	Dzēst favorītu (Jābūt autentificētam)
POST	/registerFacebookUser	Reģistrē Facebook lietotāju pēc Facebook API autentifikācijas, ja lietotājs nav iepriekš reģistrēts

Lai komentētu recepti, pievienotu un dzēstu favorītu autentifikācija notiek līdzīgi kā administratora autorizācija attēlā 4.3.2., vienīgi izsaukums tiek veikts no klienta ierīces un netiek pārbaudītas administratora privilēģijas, tiek pārbaudīts tikai vai autentificētais lietotājs atbilst pieprasījumā norādītajam, kā arī tiek pārbaudītas papildu funkcionālās prasības katrai funkcijai.

4.7. Autentifikācija un autorizācija

Lietotnei nav iekšējā reģistrēšanās sistēma, tā vietā autentifikācijai ir izvēlēts OAuth 2.0 protokols un Facebook autentifikācija, nākotnē plānots ieviest arī citu sociālo portālu autentifikāciju metodes. Facebook autentifikācija tika izvēlēta, jo OAuth 2.0 protokols ir ļoti ērts un viegli lietojams, kā arī Facebook sociālajam portālam ir ļoti liels lietotāju loks. Visas metodes, kuras izmanto autentifikāciju notiek caur HTTPS, lai pieprasījums tiktu kriptēts un nevarētu pārtvert piekļuves pilnvaru tīrā tekstā. Autentifikācijas shēma redzama attēlā 4.7.2.



4.7.2. att. **Autentifikācijas shēma**

1. Klients pieslēdzas sistēmai caur Facebook API.
2. Klients saņem no Facebook API piekļuves pilnvaru.
3. Klients veic izsaukumu uz REST servera ar „Authorization: {piekļuves pilnvara}" iekļautu galvenē.
4. Serveris validē piekļuves pilvaru caur Facebook API.
5. Serveris saņem pieprasītā lietotāja informāciju.
6. Serveris pārbauda vai lietotājs atbilst pieprasītajam, eksistē datubāzē un administratora funkciju gadījumā pārbauda vai lietotājs ir administrators.
7. Serveris saņem pozitīvu atbildi un izpilda pieprasīto izsaukumu.
8. Serveris nosūta klientam atbildi uz saņemto izsaukumu.

4.8. Drošības principi

- Autentifikācija un sesijas pārvaldība – tiek lietota Facebook autentifikācija, izmantojot OAuth 2.0 protokolu, kur piekļuves pilnvara tiek sūtīta caur HTTPS iekļauta pieprasījuma galvenē.
- HTTP metodes – administratora metodes ir aizargātas ar autorizāciju un lietotājafavorītu pievienošana un dzēšana ar autentifikāciju.

- Atļautās metodes – visas funkcijas ir definētas atsevišķi, definējot tām piekļuves metodi, t.i. katrai funkcijai ir norādīts – kāda metode ir atļauta.
- CSRF un XSS – OAuth 2.0 un piekļuves pilnvara pasargā pret CSRF un Hibernate ORM filtrē ievaddatu simbolus pret XSS.
- Tiešas piekļuves objektu darbības – visas validācijas notiek servera pusē, nekas netiek uzticēt klienta puses validācijai.
- Ievaddatu validāciju – šim nolūkam palīdz Jersey ietvars, kurš ļauj definēt ievada mainīgo un Hibernate ORM filtrē datubāzes izsaukumus.
- Izvaddatu kodēšanu – nepieciešams validēt klienta pusē (uz mobilās iekārtas), administratora panelī par to parūpējas AngularJS.
- HTTPS – Autentifikācija un autorizācija notiek, izmantojot HTTPS protokolu.

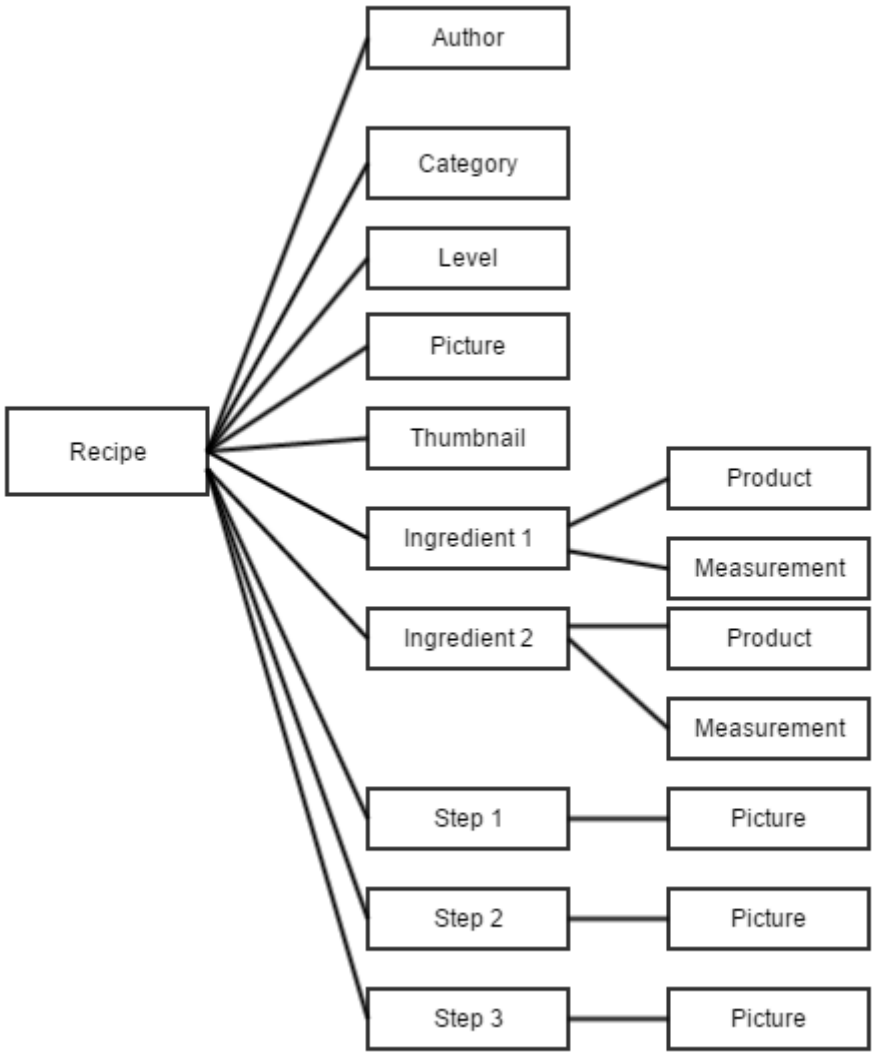
4.9. Veiktspējas testi

Mobilajai lietotnei vislielākais izsaukumu skaits sagaidāms uz visu recepšu izsaukumiem, kas ir sākuma skats un meklēšanas rezultāti, kā arī uz atsevišķām receptēm - /recipes un /recipes/{id} attiecīgi. Veikti testi, lai salīdzinātu veiktspējas ietekmi mainot dažādus faktorus.

Pirmais tests tika veikts uz vienas receptes ar 3 soļiem un 2 sastāvdaļām - http://cookster.herokuapp.com/rest/recipes/513.

Lai ielādētu vienu recepti no datubāzes ir nepieciešams salīdzinoši daudz „SELECT" izsaukumu, šajā gadījumā recepte satur 2 sastāvdaļas un 3 soļus, līdz ar to ir nepieciešami 18 „SELECT" izsaukumi no datubāzes – receptes izsaukums, visi receptes lauku objektu izsaukumi no attiecīgajām tabulām, un tālāk vēl sastāvdaļas objekts sastāvdaļas objektiem nepieciešami produkta un mērvienības izskaukumi, kā arī soļiem vajadzīgi to bilži izsaukumi. Tas viss var būt diezgan laikietiplīgs process.

Attēlā 4.9.1. redzama receptes ielādes hierarhija.



4.9.1. att. **Receptes ielādes hierarhija**

Veicot kešdarbi pie pirmās ielādes receptes tiek saglabātas operatīvajā atmiņā, kas imitē kešatmiņu, un pie nākamajiem izsaukumiem jau objekts tiek nolasīts no servera operatīvās atmiņas. Tādā veidā tiek ekonomēts laiks, kurš tiek patērēts savienojumam ar datubāzi.

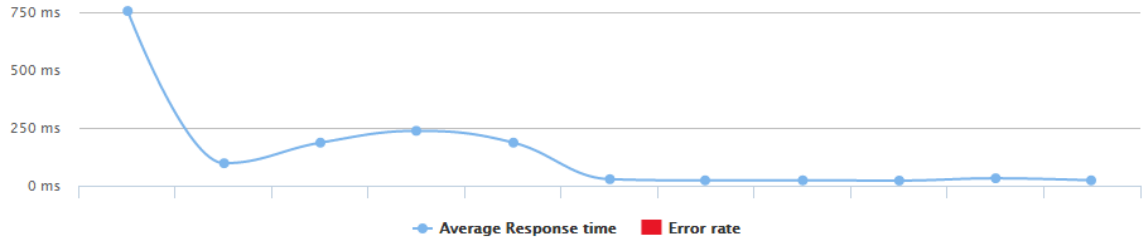
Tika veikti testi, lai pārbaudītu cik efektīvāk ir izmantot servera puses kešdarbi. Kopumā tika veikti 11 testi – 5 bez kešdarbes, 6 ar kešdarbi. Katrs tests ilga 30 sekundes un tika veikts 1 pieprasījums sekundē, kas kopumā sastāda 30 pieprasījumus vienā testā.

Pirmie tika veikti testi bez kešdarbes, tas nozīmē, ka katrā izsaukumā tika veikti 18 pieprasījumi datubāzei. Redzams, ka pirmais tests ir būtiski lēnāks nekā pārējie, tas varētu būt skaidrojams ar Heroku datubāzes noslodzi. Vidējais atbildes laiks testiem bez kešdarbes bija 292 milisekundes.

Pēc tam tika veikti 6 testi ar servera puses kešdarbi, kur ir manāmas būtiskas izmaiņas. Testu vidējais laiks ir ļoti līdzīgs – svārstās no 22 miliseknd̄m līdz 32 milisekundēm, vidējais

visu testu laiks ir 25 milisekundes. Varam secināt, ka kešdarbe būtiski uzlabo atbildes laiku, šajā gadījumā redzams gandrīz 12 reizes ātrāks atbildes laiks.

Attēlā 4.8.2. redzami loader.io testu rezultāti uz vienas receptes izsaukumu.



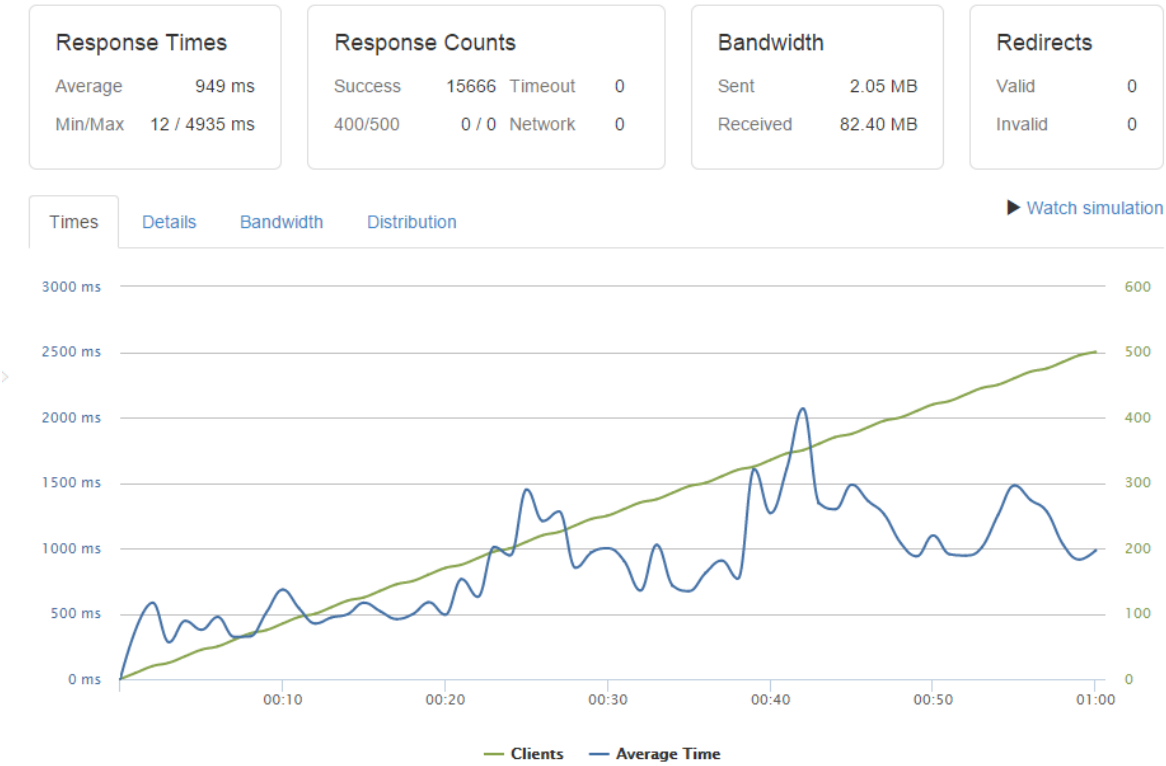
Date/Time	Avg Resp Time	Success Resp	Redirects (valid/invalid)	Timeout Errors	Network Errors	Errors (400/500)	Avg Error Rate
2015-05-26 08:06 PM	23 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 08:05 PM	32 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 08:04 PM	22 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 08:04 PM	23 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 08:03 PM	23 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 08:02 PM	28 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 07:58 PM	186 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 07:57 PM	237 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 07:56 PM	186 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 07:53 PM	97 ms	30	0 / 0	0	0	0 / 0	0.0 %
2015-05-26 07:50 PM	755 ms	30	0 / 0	0	0	0 / 0	0.0 %

4.9.2. att. **Vienas receptes atbilžu laika testu rezultāti**

4.10. Stresa testi

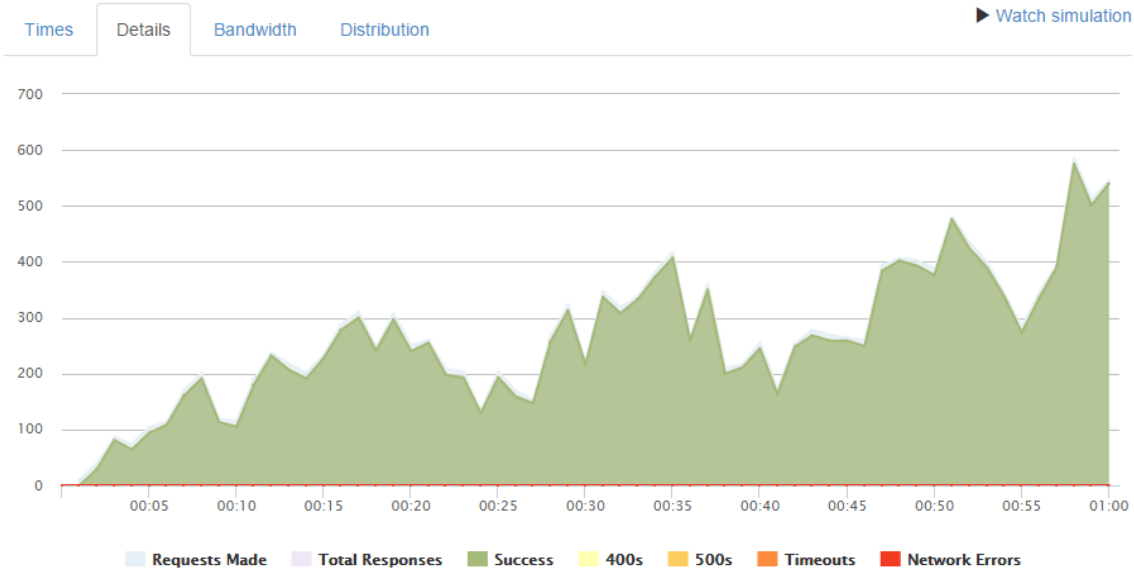
Stresa testi testē sistēmas darbību pie maksimālās slodzes. Tika veikti 4 testi pie sistēmas bez servera puses kešdarbes, katrs pieprasījums veic datubāzes izsaukumu, un 4 testi pie tās pašas sistēmas ar servera puses kešdarbi.

Pirmā testa rezultāti redzami attēlā 4.10.1.



4.10.1. att. **Stresa tests nr. 1 – vidējie atbilžu laiki**

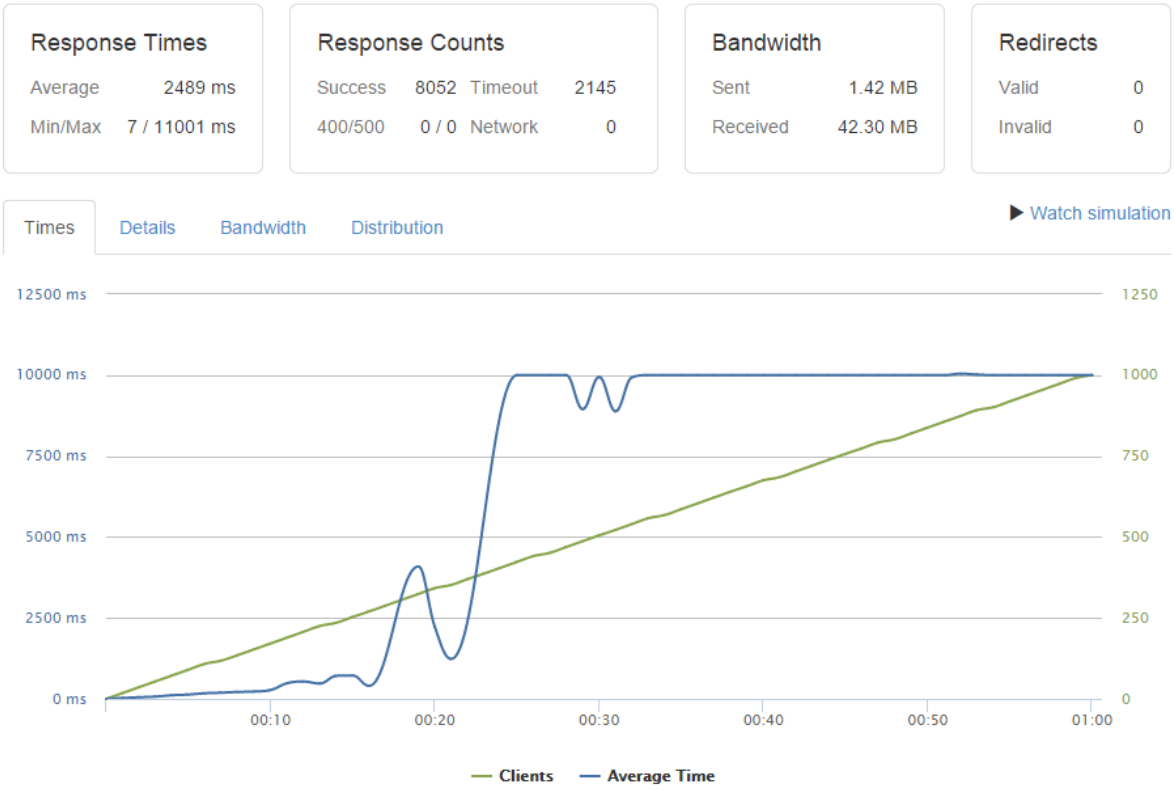
Pirmajā testā lietotāju skaits aug lineāri no 0 līdz 500 minūtes laikā, uz visiem 15666 pieprasījumiem saņemta gaidītā atbilde, vidējais atbildes laiks ir 949 milisekundes un ilgākais atbildes laiks – 4935 milisekundes. Attēlā 4.10.2. redzami pirmā testa atbilžu statusi un to skaits.



4.10.2. att. **Stresa tests nr. 1 – atbilžu statusi**

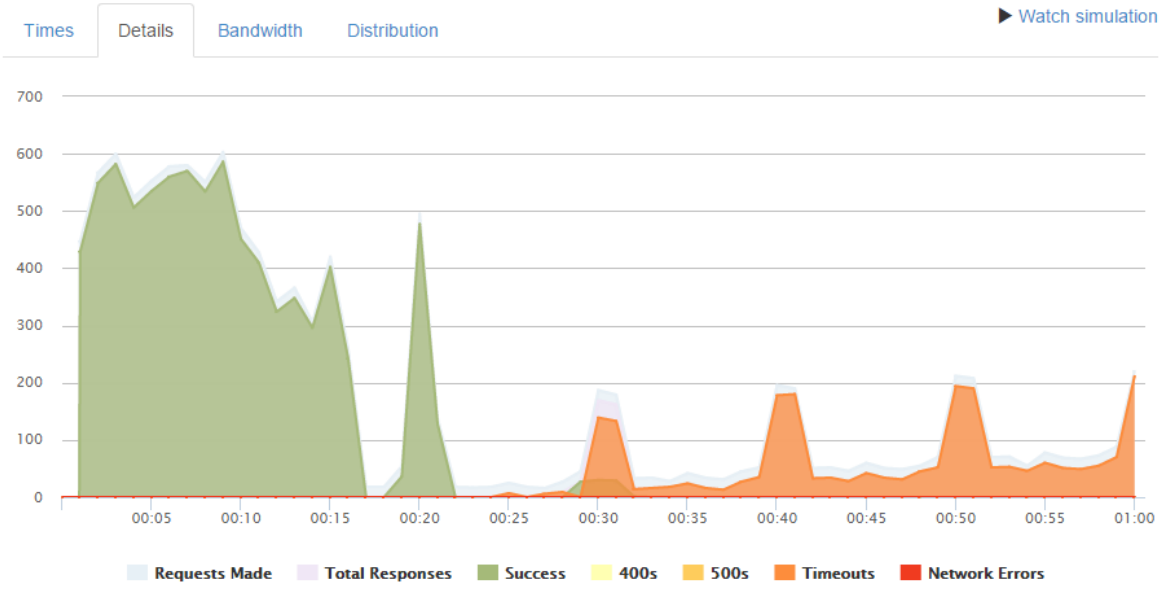
Visi statusi veiksmīgi, un pieaugot klientu pieprasījumu skaitam pieaug atbilžu skaits.

Otrā testa rezultāti redzami attēlā 4.10.3.



4.10.3. att. **Stresa tests nr. 2 – vidējie atbilžu laiki**

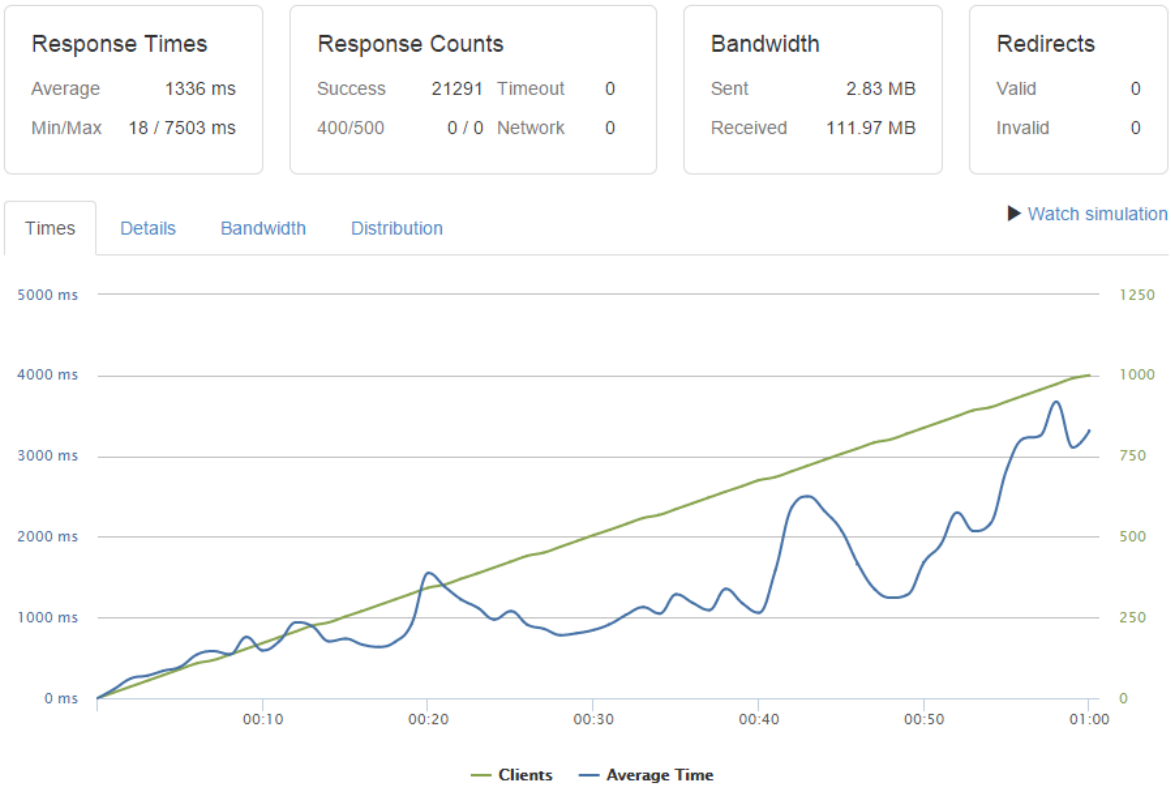
Otrajā testa ir palielināts lietotāju skaits, kurš aug lineāri no 0 līdz 1000. Redzams, ka pie aptuveni 450 lietotājiem/sekundē serverim ir pārāk liela slodze, un tas vairs nespēj apkalpot turpmākos pieprasījumus. Attēlā 4.10.4. redzami otrā testa atbilžu statusi un to skaits.



4.10.4. att. **Stresa tests nr. 2 – atbilžu statusi**

Redzams, ka 25. sekundē sāk parādīties taimauti (serveris nesniedz atbildi 10 sekunžu laikā), un tie turpinās līdz pat testa beigām.

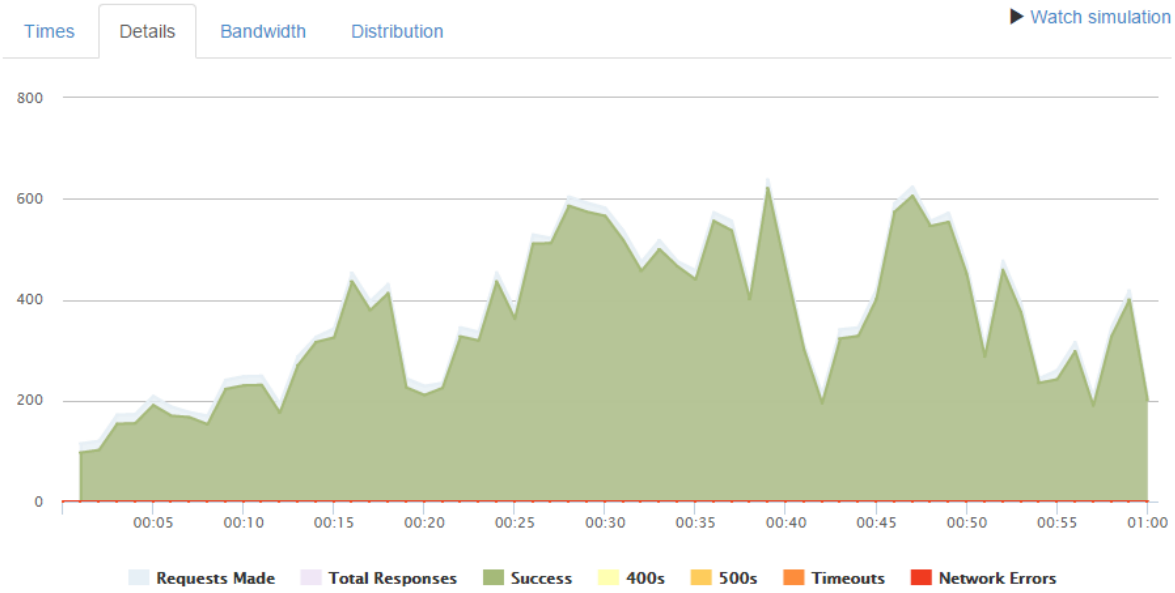
Trešā testa rezultāti redzami attēlā 4.10.5.



4.10.5. att. **Stresa tests nr. 3 – vidējie atbilžu laiki**

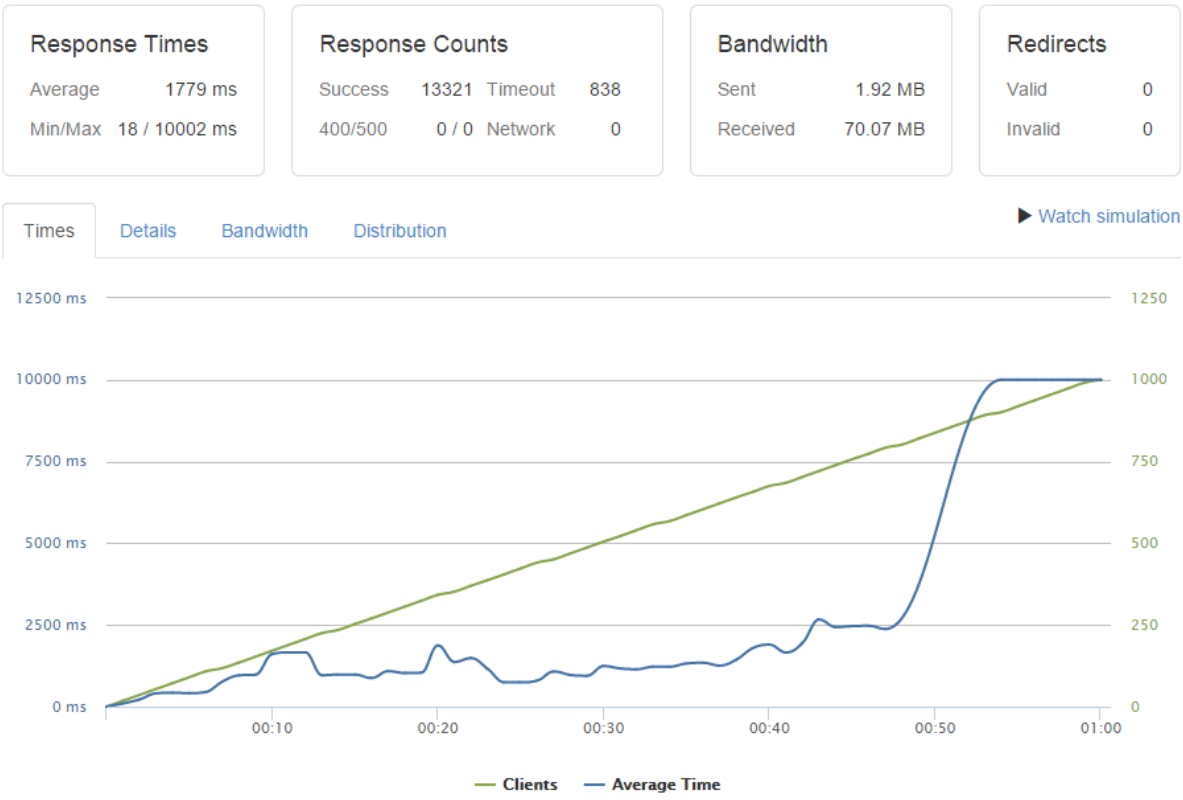
Lietotāju skaits atkal aug no 0 līdz 1000, un notiek sagaidāmais rezultāts, ka pie klientu skaita palielināšanās gandrīz vienmērīgi palielinās arī atbildes laiks. Vidējais atbildes laiks – 1336 milisekundes, maksimālais atbildes laiks – 7503 milisekundes. Attēlā 4.9.6. redzami trešā testa atbilžu statusi un to skaits.

Ceturtā testa rezultāti redzami attēlā 4.10.7.



4.10.6. att. **Stresa tests nr. 3 – atbilžu statusi**

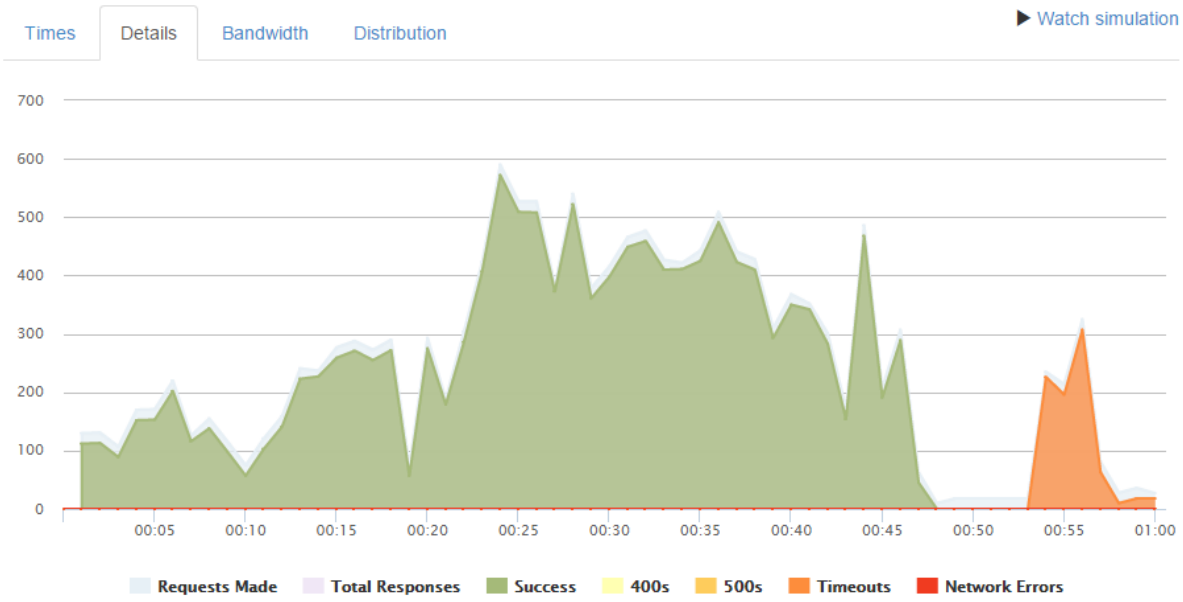
Līdzīgi kā pirmajā testā, nekas īpašs un nesagaidāms netika sagaidīts.



4.10.7. att. **Stresa tests nr. 4 – vidējie atbilžu laiki**

Lietotāju skaits atkal aug no 0 līdz 1000, un pie klientu skaita palielināšanās gandrīz vienmērīgi palielinās arī atbildes laiks. 50. sekundē serveris neiztur slodzi un pārstāj atbildēt uz

pieprasījumiem. Vidējais atbildes laiks – 1779 milisekundes, maksimālais atbildes laiks pārsniedz 10 sekundes. Attēlā 4.10.8. redzami ceturtā testa atbilžu statusi un to skaits.



4.10.8. att. **Stresa tests nr. 4 – atbilžu statusi**

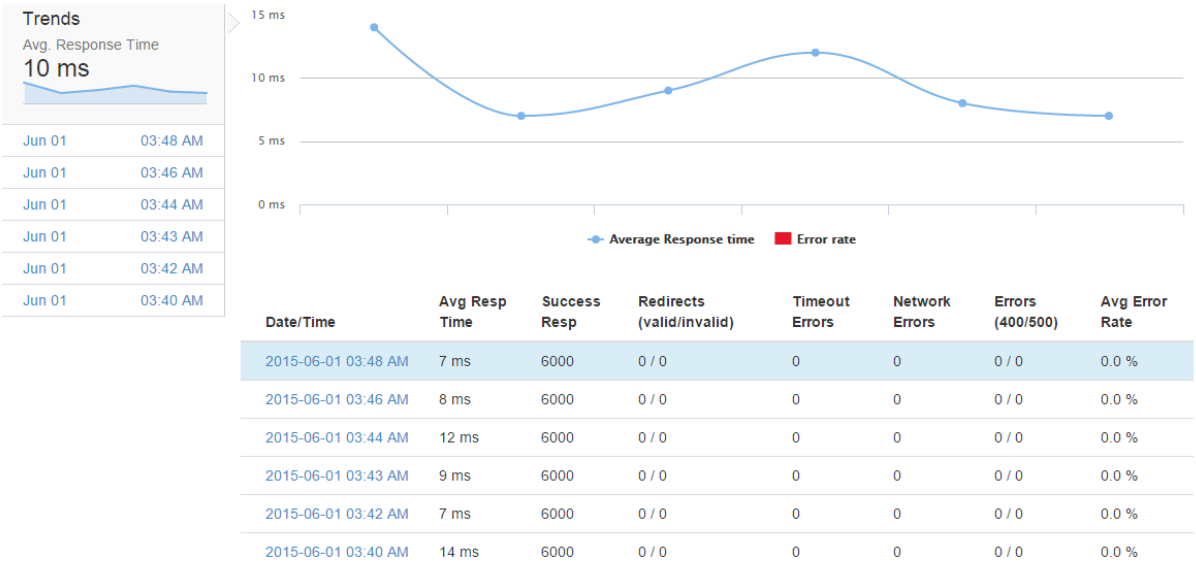
Redzams, ka 48. sekundē serveris pārstāj dot atbildes un iestājās taimauti.

Veicot stresa testus bez kešdarbes, Heroku PostgresSQL pirmajās sekundēs atgrieza kļūdas paziņojumu par pārāk daudz datubāzes pieprasījumiem, tādēļ testi nevarēja turpināties.

Veicot stresa testus var veikt secinājumus, ka serveris noteikti nevar izturēt pieaugošo klientu skaitu pieprasījumus un lietotnes klientu skaita pieauguma gadījumā nepieciešams veikt mērogošanu, taču servera sākuma mērķis ir apkalpot 100 konstatus lietotājus vienlaicīgi, tādēļ ir nepieciešami slodzes pārbaudes testi uz 100 lietotājiem.

4.11. Slodzes testi

Tika veikti 6 konstantas slodzes testi katrs ilga minūti un veica 100 pieprasījumus sekundē. Tāpat kā stresa testiem, veicot slodzes testus bez kešdarbes, Heroku PostgresSQL pirmajās sekundēs atgrieza kļūdas paziņojumu par pārāk daudz datubāzes pieprasījumiem, tādēļ testi nevarēja turpināties. Attēlā 4.11.1. ir redzami testu rezultāti.



Date/Time	Avg Resp Time	Success Resp	Redirects (valid/invalid)	Timeout Errors	Network Errors	Errors (400/500)	Avg Error Rate
2015-06-01 03:48 AM	7 ms	6000	0 / 0	0	0	0 / 0	0.0 %
2015-06-01 03:46 AM	8 ms	6000	0 / 0	0	0	0 / 0	0.0 %
2015-06-01 03:44 AM	12 ms	6000	0 / 0	0	0	0 / 0	0.0 %
2015-06-01 03:43 AM	9 ms	6000	0 / 0	0	0	0 / 0	0.0 %
2015-06-01 03:42 AM	7 ms	6000	0 / 0	0	0	0 / 0	0.0 %
2015-06-01 03:40 AM	14 ms	6000	0 / 0	0	0	0 / 0	0.0 %

4.11.1. att. **Slodzes testu rezultāti**

Slodzes testi uz servera ar konstatu 100 lietotāju pieprasījumiem sekundē serveris strādā stabili ar vidējo atbildes laiku 10 milisekundes, un atbild visām veiksmīgām atbildēm.

4.12. Datu apjoms

Ja viena objekta izsaukumam galvenokārt ir nepieciešami visi objekta dati, tad kolekcijas gadījumā bieži pietiek ar tikai galvenajiem objekta laukiem. Kā piemērs varētu būt recepšu serviss – kad mobilajā lietotnē tiek atvērta recepte, ir nepieciešami visi receptes pieejamie dati, taču lietotnes sākuma skatā, kur ir recepšu saraksts, nav nepieciešams, piemēram, receptes soļi, sastāvdaļas, garais nosaukums. Izveidojot datu apmaiņas objektu ar minimāli nepieciešamajiem datiem tiek ne tikai ietaupīts sūtāmais datu apjoms, kurš ir svarīgs mobilajām lietotnēm, jo bieži tiek izmantots mobilais internets, kuram ir ierobežots datu apmaiņas limits, bet arī no drošības aspekta vienmēr nepieciešams atļaut tikai nepieciešamās darbības un neko vairāk, šajā gadījumā – tikai nepieciešamos datus.

Iepriekš apskatītajai receptei ar 3 soļiem un 2 sastāvdaļām JSON datu izmērs būtu 2736 baiti, ņemot vērā, ka tā ir tikai testa recepte, tad īstai receptei ar vairākiem soļiem un sastāvdaļām datu izmērs būtu vismaz 5000 baiti. Šīs receptes datu apmaiņas objekts kolekcijai ir tikai 564 baiti, kas šajā gadījumā samazina izmēru gandrīz 5 reizes, „sarežģītākai" receptei datu apjoma starpība būtu vēl lielāka. Vienam objektam šī starpība – pāris kilobaiti var nelikties liela, taču, ja tiek pieprasīta kolekcija tiek ietaupīts simtiem kilobaitu, un, ja klients bieži pieprasa atjauninājumus no servera, tad jau tiek ietaupīti daudzi megabaiti, un tas ir tikai vienam lietotājam.

SECINĀJUMI

Darba mērķis ir izpētīt RESTful servisu drošības principus, autentifikācijas metodes – pret ko nepieciešams pasargāties, izstrādājot RESTful servisus, kā uzlabot servisu veiktspēju, kā arī ar praktiskā darba palīdzību parādīt, kā ievērot drošības un veiktspējas principus, izstrādājot RESTful servisus. Abi mērķi tika pilnībā sasniegti.

Darba tapšanas gaitā tika izveidoti RESTful servisi un administratora panelis mobilajai lietotnei, kā arī apkopota informācija par REST stila arhitektūru, tās drošības pricipiem un veiktspējas uzlabošanas metodēm.

Ievērojot pāris vienkāršus principus ir iespējams būtiski uzlabot REST servera veiktspēju, kas ir redzams veiktajā veiktspējas analīzē. Tādā veidā iespējams apkalpot vairāk klientu vienlaicīgi, un apkalpot tos ātrāk.

Nav viens unikāli labākais risinājums RESTful servisu izstrādē, tādēļ ir jāapskata visas risinājumu iespējas un pielāgot tos projekta vajadzībām, lai sasniegtu pēc iespējas labākus rezultātus un palielinātu klientu apmierinātību.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Wikipedia, Basic access authentication, [tiešsaitse] – [skatīts 19.05.2015]. Pieejams: http://en.wikipedia.org/wiki/Basic_access_authentication
2. Lifehacker, Understanding OAuth, [tiešsaitse] – [skatīts 19.05.2015]. Pieejams: http://lifehacker.com/5918086/understanding-oauth-what-happens-when-you-log-into-a-site-with-google-twitter-or-facebook
3. Sitepoint, Understanding http digest access authentication, [tiešsaitse] – [skatīts 19.05.2015]. Pieejams: http://www.sitepoint.com/understanding-http-digest-access-authentication/
4. Tutplus, A beginners guide to http and rest, [tiešsaitse] – [skatīts 19.05.2015]. Pieejams: http://code.tutsplus.com/tutorials/a-beginners-guide-to-http-and-rest--net-16340
5. Cubrid, Dancing with OAuth understanding how authorization works, [tiešsaitse] – [skatīts 19.05.2015]. Pieejams: http://www.cubrid.org/blog/dev-platform/dancing-with-oauth-understanding-how-authorization-works/
6. Loader.io, Simple cloud based load testing, [tiešsaitse] – [skatīts 26.05.2015]. Pieejams: http://loader.io
7. OWASP, REST security cheat sheet , [tiešsaitse] – [skatīts 19.05.2015]. Pieejams: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet
8. Codeproject.com, Digest Authentication on a WCF REST Service, [tiešsaiste] – [skatīts 19.05.2015]. Pieejams: http://www.codeproject.com/Articles/162726/Digest-Authentication-on-a-WCF-REST-Service
9. Tech.groups.yahoo.com, "Fielding discusses the development of the REST style", [tiešsaiste] – [skatīts 26.05.2015]. Pieejams: http://web.archive.org/web/20091111012314/http://tech.groups.yahoo.com/group/rest-discuss/message/6757

Bakalaura darbs „RESTful arhitekūrā veidotu servisu optimizācija un drošība” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____ Rihards Fridrihsons

Rekomendēju/nerekomendēju darbu aizstāvēšanai
Vadītājs: profesors Dr.sc.comp. Guntis Arnicāns _____ 01.06.2015.

Recenzents: Dr.sc.comp. Kārlis Čerāns _____

Darbs iesniegts Datorikas fakultātē 01.06.2015.
Dekāna pilnvarotā persona: Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē
__.06.2010. prot. Nr. _____
Komisijas sekretārs(-e): _____