Универзитет у Београду

Математички факултет

Имплементација веб сервиса коришћењем технологија NodeJS и MongoDB

Мастер рад

Ментор:
Проф. др Владимир Филиповић

Кандидат:
Милан Ђорђевић

Садржај

1. Увод

Убрзани развој Интернета крајем деведесетих година прошлог века показао је извесне слабости дотадашњих архитектура и решења и постала је евидентна потреба за развојем нових, како технологија, тако и методологија у циљу испуњавања нарастајућих захтева. Веб сервиси су се у овоме показали као доминантно и квалитетно решење. Циљ овог мастер рада је да пружи увид у део тих проблема и представи, још увек актуелне, покушаје да се на исте одговори. Он ће се бавити ближим прегледом једне од архитектура која се појавила као потенцијално решење, а то је *REST* архитектура.

Паралелно са популаризацијом Веб сервиса као начина повезивања система догодила се велика експанзија када су *JavaScript* серверске технологије и *NoSQL* базе података у питању. У овој великој популарности које су стекле ове технологије и покрети који су се оформили око њих, као и у тенденцији да се њихов раст и развој настави у будућности, огледа се значај теме којом се бави овај рад. Предмет изучавања рада биће две најпопуларније технологије које су настале као резултат ових трендова, а то су *Node.js* и *MongoDB*.

Node.js је мултиплатформско извршно окружење настало 2009. године које се временом развило у веома моћан асинхрони програмски оквир за развој скалабилних Веб апликација. Основне особине овог окружења су једноставност и веома висок степен перформанси. *MongoDB* је систем за управљање базама података базиран на документима који је последњих година стекао завидну популарност и донекле се издвојио од сличних решења. Његове главне карактеристике су брзина (како читања тако и уписа) као и флексибилност која краси сличне *NoSQL* системе за управљање базама података. Поред ових карактеристика, он садржи и веома богат скуп функционалности које га чине комплетним и моћним системом за управљање базама података.

Кроз ове две технологије ће бити демонстрирани основни концепти и принцип *REST* архитектуре. Настојање овог рада јесте да се покаже смисао постојања, како архитектуре, тако и конкретно ове две технологије, да се пружи преглед њихових функционалности и одговори на којим пољима је погодно користити ове технологије, где их треба избегавати и, уопште, који су њихове предности и недостаци.

Рад се састоји из две целине. У првој ће бити представљен кратак историјат развоја Веб сервиса, опис и основне карактеристике *REST* архитектуре, *JavaScript* језик, као и две серверске технологије које су настале из њега, *Node.js* и *MongoDB*.

Практични део рада састоји се из имплементације Веб сервиса коришћењем техника и методологија описаних у првом делу рада. Сервис који ће бити презентован је део система за евиденцију и праћење налога за интервенције (енг. *Issue Tracking System*). У склопу овог дела рада биће представљен модел базе података која лежи у основи овог система. Такође, биће приказани најзначајнији делови кода сервиса и дат приказ изгледа апликације која је развијена за сврху презентације сервиса али не представља саставни део овог рада. Но, чини се да она лепо заокружује приказ сервиса и може послужити у сврху додатног појашњења неких од наведених функционалности.

2. Настанак и развој Веб сервиса

Са почетком убрзаног развоја Интернета крајем прошлог века појавила се потреба за повећањем степена повезивости система. То је било нарочито значајно за компаније за које се одједном отворило потпуно ново тржиште које до тада нису могле ни да наслуте. Софтверска индустија је препознала те тенденције и почела озбиљно да истражује и ради на интероперабилности између система и на начинима на које би она могла бити постигнута. Решења до којих се дошло постала су позната као веб сервиси.

Према дефиницији *W3*[1] конзорцијума, веб сервис је софтверски систем дизајниран да подржи интероперабилне интеракције између две машине преко мреже. Он садржи интерфејс описан у формату погодном за машинску обраду. Други системи врше интеракцију веб сервисом користећи поруке (на начин описан у дефиницији сервиса) које су обично серијализоване коришћењем *XML* (или неког сличног) формата и које се шаљу путем *HTTP* протокола [1].

Прва решења за интероперабилност која су се појавила били су системи за дистибуирање објеката *DCOM* (*Distributed Component Object Model*) компаније *Microsoft* и *CORBA* (*Common Object Request Broker Architecture*) развијена од стране *Object Management* групе (*OMG*). Они су имали један велики недостатак који се огледао у чињеници да нису били међусобно компатибилни. То је у пракси представљао огроман проблем јер, у суштини, то је била потпуна супротност од онога због чега су веб сервиси замишљени. Након њих било је других решења (као што је *RMI*[2]) али код свих је био заступљен проблем компатибилности због језичких зависности и никада нису стекли велику популарност.

Покушавајући да реши проблем компатибилности *Microsoft* је до 1997. одлучио да истражи решења која ће користити *XML* као главни транспортни језик и која ће омогућити системима да комуницирају користећи *RPC*[3] преко *HTTP* протокола. То истраживање је 1998. изродило *XML-RPC* [2]. Са њим је постигнуто прво, у већој мери, технолошки независно решење. Из тих разлога *XML-RPC* је стекао велику популарност да би коначно, крајем 1999. године, израстао у стандард који данас познат под називом *SOAP (Simple Object Access Protocol)* [3]. *SOAP* протокол није заправо био потпуно технолошки независан јер захтева да се функционалност сервиса дефинише коришћењем специјалног језика, базираног на *XML* формату, који се назива *Web Service Description Language* (*WSDL*).

И поред својих ограничења, *SOAP* и *XML-RPC* омогућили су Сервисно-оријентисаној архитектури (*Service-oriented architecture*, *SOA*) да постане доминантна архитектура за имплементацију аутоматских интеракција између дистрибуираних и хетерогених апликација а, самим тим, и за уједињавање пословних процеса унутар једне или између више компанија. За разлику од дистрибуираних објеката, основна примитива Сервисно-оријентисане архитектуре је услуга (*service*). Иако је, по својој дефиницији, *SOA* технолошки независна, *SOAP* се

[1] *World Wide Web Consortium – www.w3.org*

[2] *Remote Method Invocation* – Објектно-оријентисани *API* писан за језик *Java* који представља фукнционални еквивалент *RPC*-а

[3] *Remote Procedure Call* – Технологија која омогућује да се позивају програми на удаљеним рачунарима

успоставио као синоним за подразумевани транспортни протокол а *WSDL* као главни дефиниционi језик за опис функционалности сервиса.

Међутим, како су имплементације постајале све масовније, појавила се потреба за једноставнијим приступом приликом развоја сервиса, нарочито када су веб апликације у питању. У овом аспекту је *REST* (*Representational State Transfer*) архитектура тренутно непревазиђена [4]. Док је код *SOA* основна аутономна јединица кроз који се дефинише рад система услуга (сервис), код *REST* архитектуре је то ресурс. Развој *REST* архитектуре променио је устаљено схватање о томе шта веб сервиси представљају и довео до поделе на следеће две велике класе веб сервиса:

1) *REST*–компатибилни веб сервиси, код којих је основна сврха сервиса манипулација над репрезентацијама ресурса коришћењем униформног скупа операција које не чувају стања
2) Веб сервиси који могу имати имплементиран произвољни скуп операција специфичан за саму имплементацију [5].

У овом поглављу дат је кратак приказ историјата развоја веб сервиса. Али, како тема овог рада није изучавање или потрећење аспеката различитих класа веб сервиса већ конкретна имплементација, у будућим поглављима дискурс ће бити ограничен на *REST* архитектуру која је и основ за практични део рада. Наредна поглавља би требало да дају одговор зашто је ова архитектура нарочито погодна за коришћење са технологијама које су теме истраживања овог рада.

2.1 *REST* сервиси

Историјат развоја *REST* архитектуре започео је и текао паралелно са развојем верзије 1.1 протокола *HTTP* (1996.–1999.). Архитектура је добила свој коначни облик 2000. године када је описана у докторској дисертацији америчког информатичара Роја Филдинга[4] која носи назив Архитектурални стилови и дизајн софтверске архитектуре у мрежном окружењу (*Architectural Styles and the Design of Network-based Software Architecture*). У њој је Филдинг дефинисао *REST* као архитектурални стил за дистрибуиране хипермедијалне системе [6]. Првенствено, *REST* је настао је као одговор на робусне архитектуре за дистрибуирано слање објеката (*DCOM* и *CORBA*) које су тада биле заступљене. Циљ је био да се направи једноставна архитектура која ће радити на истим принципима на којима је и заснован *HTTP* протокол. Речима Филдинга:

"Мотивација за развој REST-а је прављење модела архитектуре који описује како би Веб требало да ради, таквог да може да послужи као оријентир при дефинисању стандарда протокола за Веб. "

Дакле, архитектуру карактерише замена сложених протокола једноставним (протоколе *CORBA, RPC, SOAP* мења генерички *HTTP* протокол). Такође, сервиси се не ослањају на протоколе базиране на *XML*-у као подршка својим интерфејсима [7], односно елиминисана је

[4] *Roy Fielding* – један од водећих аутора *HTTP* протокола и ко-оснивач *Apache* пројекта

потреба за умотавањем порука. Сходно томе, предности *REST*-а су лакша имплементација, брзина развоја и непостојање додатних елемената архитектуре.

2.1.1 Основе *REST* архитектуре

„REST би требало да подсећа на понашање добро пројектоване Веб апликације: мрежа страница (виртуални коначни аутомат), код које корисник пролази кроз апликацију бирањем веза (преласци стања), што резултује преношењем нове странице (наредно стање апликације) кориснику и њеним припремањем за употребу.“

Рој Филдинг

REST је стекао, са добрим разлогом, завидну популарност због своје једноставности, али као што је то случај са многим новим технологијама, ушао је у масовну употребу веома брзо што је изродило велики број заблуда и различитих тумачења. Због тога што *REST* представља стил, а не стандард или препоруку, критеријуми дизајна сервиса су веома уопштени и остављено је отворено за интерпретацију шта заправо представља један добро заснован *RESTful*[5] веб сервис. Но, и поред тога, *REST* никада неће, нити може, бити стандардизован јер не представља конкретну технологију.

Дистрибуирани систем организован у складу са *REST* стилом требало би да унапреди следеће аспекте система:

- *Перформансе* – комуникација би требало да буде једноставна и ефикасна
- *Скалабилност* – интеракције између компонената
- *Једноставност интерфејса*
- *Изменљивост компоненти* – подела одговорности (*separation of concerns*) коју предлаже *REST* омогућава да се компоненте могу мењати независно једна од друге
- *Преносивост* (*portability*) - *REST* је технолошки и језички независан, што значи да може бити имплементиран и коришћен од стране било које технологије
- *Поузданост* – особина не чувања стања (*statelessness*) омогућује релативно једноставан опоравак сервиса у случају системских кварова
- *Видљивост* – такође због не чувања стања постоји бенефит повећања видљивости. Сваки сервисни агент који, потенцијално, прати рад сервиса не мора ништа више да зна осим поруке у одговору захтева да би установио стање у којем се захтев који је послат налази [2], [6].

Из ове листе може се непосредно извући још неколико предности:

- Систем који је изграђен из више компоненти је доста толерантан на грешке и кварове јер једна компонента не утиче битно на рад друге

[5] *RESTful* је назив који је ушао у општу употребу за сервисе који су имплементирани коришћењем *REST* архитектуралног стила

- Повезивање компоненти је лако што смањује ризик приликом додавања нових функционалности или скалирања у било ком смеру
- Технолошка независност омогућује систему да буде приступачан ширем аудторијуму програмера [2].

Четири главне особине *REST* архитектуре су: адресибилност, не чување стања, повезаност и униформни интерфејс [8].

2.1.2 Униформни интерфејс

Основна идеја иза *REST* архитектуре је да се атомични делови функционалности апликације и/или базе података изложе као ресурси кроз униформни интерфејс (*uniformed interface*). Ресурси су главни градивни блокови *REST* архитектуре. Они представљају је апстракцију свега што може бити концептуализовано (веб страна, слика, особа и сл.) и представљају једну од примитива веб архитектуре [9]. Ресурси су представљени следећом структуром:

- **Репрезентација** - Начин на који се подаци представљају (бинарна, *JSON*, *XML* и сл.).
- **Идентификатор** - *URL* који учитава тачно један ресурс у сваком тренутку
- **Мета подаци** - Тип садржаја, датум измене и слично
- **Контролни подаци** - Време од када је ресурс могуће мењати ресурс, контрола кеширања и слично [2].

У суштини, репрезентација је скуп бајтова и мета података који описују те бајтове. Један ресурс може имати више од једне репрезентације. Типично је да клијент од сервиса захтева тип репрезентације кроз механизам који се назива преговарање о садржају (*content negotiation*) који је део *HTTP* стандарда [10]. Оно подразумева да се у заглављу *HTTP* захтева шаље ознака тип садржаја који клијент очекује у одговору:

```
Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6,
image/jpeg; q=0.6, image/*;q=0.5, */*; q=0.1
```

Идентификатор ресурса треба да пружи јединствени начин да се идентификује ресурс и треба да пружи комплетну путању до ресурса преко *URI*-ја (*unique resource identifier*). Најважнија особина идентификатора ресурса је да у сваком тренутку може недвосмислено да приступи да ресурсу кога идентификује.

Формат транспортног језика, који је трационално био резервисан за *XML*, у последње време је, када су у питању *REST* сервиси, припао *JSON*-у. Постоји неколико разлога за то. Пре свега, једноставан је – садржи врло мало података које нису директно везане за информацију која се преноси, читљив је од стране човека и, коначно, има подршку за различите типове података.

2.1.3 Операције над ресурсима

С обзиром да се *REST* ослања на *HTTP* протокол за слање порука потребно је укратко описати његове основне карактеристике и начин функционисања. *HTTP* је генерички мрежни протокол који припада слоју апликације (стандардног *OSI* референтног модела) за дистрибуиране, колаборативне, хипермедијалне информационе системе. Он може бити коришћен за разне намене ван употребе хипер-текста кроз проширивање његових метода, кодова за грешке и заглавља [9]. У самом протоколу дефинисане операције *GET*, *POST*, *PUT* и *DELETE* заправо су операције над ресурсима. Из тог разлога природно је да предложене операције над ресурсима у оквиру *REST* архитектуре буду изложене и уско везане за методе које нуди *HTTP*. Ово се првенствено односи на операције писања, читања, ажурирања и брисања ресурса (енгл. *CRUD*[6] операције). Начин на који се користи *HTTP* протокол представља једну од основних разлика у односу на *SOA* где стоје у телу сваког захтева морају да стоје описи метода који се позивају.

Иако постоји још врста операција које *REST* сервис може пружити и које могу бити дефинисане кроз *API*, препорука се односи на генеричке операције дефинисане над једним ресурсом које би требало сервис да подржи.

- ***GET*** – Приступ ресурсу за читање
- ***POST*** – Обично унос новог ресурса
- ***PUT*** – Обично ажурирање постојећег ресурса
- ***DELETE*** – Брисање ресурса
- ***HEAD*** – Упит да ли ресурс постоји без враћања репрезентације
- ***OPTIONS*** – Упит који враћа листу метода над датим ресурсом који су на располагању

Наравно, пошто су то препоруке оне не морају бити испоштоване по сваку цену, а постоје и случајеви када их је немогуће испоштовати (нпр. када на серверима нису омогућени сви ови методи).

Операције писања, читања, ажурирања и брисања су основне и нужне, али су део већег подскупа операција које су потребне да се врше над ресурсима. Честе операције које нису до сада побројане укључују претрагу, филтрирање, рад са подресурсима и слично. Први начин на који могу да се реализују позиви за ове операције је кроз *URI*, на пример:

```
GET /api/v1/tickets/search
GET /api/v1/tickets/filtering
```

У овом случају се губи првобитна намена *URI*-ја који сада поред идентификовања самог ресурса служи и за реферисање акција над ресурсом или групом ресурса. Ово може изгледати као добра идеја на почетку, али на овај начин пораст комплексности система довешће до заначајног повећања *URI*-ја што може закомпликовати имплементацију клијентских апликација.

[6] Акроним за *Create, Read, Update* и *Delete* операције

Решење тог проблема лежи у употреби симбола „?“ који има улогу маскирања операција над ресурсом. За претходне примере:

```
GET /api/v1/tickets?q=[терм за претрагу]
GET /api/v1/tickets?filters=[делимитирана листа филтера]
```

2.1.4 Одржавање стања

Стање апликације односи се на стање које сервер мора да одржава између два захтева за сваког клијента. Најчешћа препорука приликом дизајнирања сервиса је да се стање апликације одржава на клијенту, односно да се сесије не чувају у меморији сервера. Начин на који се то постиже у *RESTful* сервисима је кодирање стања апликације у сам *URI* захтева [11]. Чување стања на клијентима не значи серијализацију стања сесије у *URI* или *HTML* форме на начин на који то ради, рецимо, *ASP.NET*. Ако је стање превелико да се транспортује или постоје сигурносна или питања приватности, препорука је да се стања чувају у трајнијем складишту као што су базе података или фајл систем, а да се шаље референца ка тим стањима. Циљ је да се постигне баланс између поузданости, брзине и скалабилности.

3. *JavaScript* програмски језик

JavaScript је изворно носио име *Mocha* када је развијен у компанији *Netscape* 1995. Септембра исте године појавила се бета верзија *Netscape Navigator*-а 2.0 који је имао уграђену верзију *Mocha*-е која је у тој верзији преименована у *LiveScript* [12]. После још једне промене имена Децембра 1995. језик је најзад добио име *JavaScript* које и данас носи. У том периоду је *Netscape* тесно сарађивао са компанијом *Sun*, творцем језика *Java* који је у том тренутку доживљавао велику експанзију. То је довело до спекулација да је *Netscape* желео да *JavaScript* преузме део славе коју је *Java* имала, иако два језика нису имала скоро ничег заједничког. То је произвело велику конфузију у програмерској заједници јер су многи поистовећивали један језик са другим. Упркос конфузији коју су прозивели *JavaScript* је постао веома популаран скрипт језик. Из тог разлога је *Microsoft* одлучио да развије своју имплементацију која је названа *JScript* и укључио га у верзију *Internet Explorer*-а 3.0, Августа 1996. Јуна наредне године *JavaScript* је стандардизован код интернационалне организације за стандарде (*ECMA*) под називом *ECMAScript* [13].

У наредним годинама *JavaScript* је одржавао примат и сматран је неком врстом стандарда за скриптовање на страни клијента. Потврда те тезе је да је једини клијентски језик који је подржан од стране свих великих прегледача. Та чињеница је нарочито била значајна великом експанзијом коју је Интернет доживео почетком 2000-их година када су се за језик заинтересовале компаније као што су *Google*, *Apple* и друге [12]. Персонални рачунари су најзад имали довољно добре перформансе да је постало могуће размишљати о експанзији у области апликација које би се већим делом извршавале на клијентској стани и пружиле сасвим нове могућности за имплементацију веб апликација.

3.1 *JavaScript* објектна нотација (*JSON*)

JavaScript објектна нотација или *JSON* представља формат записа објеката у текстуалном облику који је укључен у треће издање *JavaScript* (*ECMA-262*) стандарда [12]. Основна намена *JSON* формата је да се користи као механизам за серијализацију објеката у стрингове, практично пружајући исту функционалност као и *XML* само користећи мањи број слова за опис објеката. Атрибути објеката се смештају између витичастих заграда у облику парова (*кључ : вредност*) одвоједних зарезом.

```
{“key1”: “value1”, “key2”: “value2”…}
```

Типови које подржава *JSON* формат су бројеви, стрингови, буловске вредности, низови, објекти и *null.* Објекти који нису подржани се могу имплементирати користећи или постојеће методе за претварање објеката у стрингове и парсирање назад или имплементацијом сопствених метода који то раде у самој апликацији.

3.2 *JavaScript* серверске технологије

Идеја да *JavaScript* постане серверски језик датира још од периода његовог настанка. Међутим, две кључне ствари су га дуги низ година спутавале. Прва је репутација. Дуго година се на *JavaScript* гледало са подсмехом као на језик за почетнике и аматере. Друга ствар, још битнија, јесте да је језик имао изузетно лоше перформансе у порђењу са другим језицима због доминантно интерпретаторских извршних модела.

То се све променило поменутом експанзијом и интересовањем великих компанија. Оне су добринеле значајном убрзању извршавања *JavaScript*-а и одједном су се поново појавиле старе тенденције да се од њега направи пуноправни серверски језик који би могао да се умеша у борбу и буде конкуренција, пре свега, језицима попут *PHP*-а и *Java*-а [12].

4. Node.js

Све већим интересовањем за *JavaScript* серверске технологије дошле су и технолошке иновације првенствено у извршним моделима и окружењима. Једно од најраспрострањенијих извршних окружења је свакако *V8* развијено од стране компаније *Google*. Настало је као извршно огружење под којим ради прегледач *Google Chrome.* Али од свог објављивања *V8* је постао распрострањен међу пионирима *JavaScript* серверских технологија. Оно што овај оквир чини веома моћним је да се, за разлику од осталих извршних окружења у којима се покреће *JavaScript* а која су доминантно интерпретаторске природе, у *V8* окружењу *JavaScript* се компајлира у извршни код. Поред тога, оквир је одрговоран да пружи алокацију меморије као и сакупљање отпадака (*garbage collection*) [14]. На тај начин се постижу перформансе којима је и *Google Chrome* успео да освоји до тада, релативно, засићено тржиште веб прегледача. Окружење *V8* је трасирало пут за настанак многих програмских оквира од којих је вероватно

најзначајнији *Node.js* (или само *Node*). Од када је развијен 2009. године *Node* је нарастао у моћни и све популарнији асинхрони програмски оквир за развој скалабилних *JavaScript* апликација.

Намена *Node*-а првенствено је да служи за креирање сервера за веб апликације и његовим развојем је, коначно, *JavaScript* добио епитет пуноправног серверског језика. Апликације за које је најпогодније користити ову платформу су, такозване, *DIRT* (*data-intensive real-time*) апликације [15]. Разлог за то је чињеница да је *Node* развијен да пружи веома брз одзив, а то је нешто што му омогућава његов прилично неконвенционални извршни модел.

4.1 Извршни модел

Заједно са брзином *Node* је донео извршни модел који се најбоље може разумети ако се упореди са најраспрострањенијем веб сервером *Apache.* Док *Apache* обрађује *HTTP* захтеве, остављајући да се логика апликација имплементира у језицима попут *PHP*-а и *Java*-е, *Node* спаја апликациону и серверску логику на једном месту. Постоје критике везане за спајање ове традиционалне поделе одговорности, али је чињеница да је овај приступ донео до сада невиђену флексибилност.

Node се разликује од великог броја веб сервера по начину на који реализује конкурентне позиве. *Apache* за сваку конекцију са клијетном издаје једну нит (*thread*) из својих резерви нити (*thread pool*), док код *Node*-а се за све операције користи скоро искључиво само једна нит [12]. Ово, на први поглед, може да делује као лоша идеја, али постоје битне разлике у начину на који сервери врше обраду захтева. Традиционални сервери користе блокијрајуће *I/O* операције. Што значи да нит која је издата за обраду тог захтева остаје замрзнута док се он не изврши. Примера ради, ако постоји захтев за читање из базе података, сервер издаје захтев и онда чека на извршење упита и на крају врати одговор. Очито, *Node* не би могао да функционише на тај начин јер, како процес има само једну нит, сви клијенти били би на чекању ако постоји макар један неразрешен захтев. Насупрот том приступу, *Node* користи неблокирајући *I/O*, односно пошаље упит бази података и, уместо да чека одговор, он наставља да обрађује друге захтеве док се не заврши упит над базом и онда тек врати одговор тако што асинхроно покрене извршење повратне функције која је одгворна да процесуира резултат упита.

4.2 Асинхроно програмирање и обрада изузетака

Један од веома важних карактеристика програмског модела *Node*-а је да се скоро сви позиви функција извршавају асинхроно. Код парадигме *Node*-а која се понекад назива стил прослеђивања продужетка (*continuation-passing style* или *CPS*), асинхроне функције узимају као додатни аргумент функцију која се позива након извршења асинхроног кода. Функција која је

прослеђена као аргумент се у том случају назива продужетком (*continuation*) или чешће функцијом са повратним позивом (*callback function*).

```
function checkSession (id, callback) {
    var valid;
    if(/* тест */)
        valid = true;
    else
        valid = false;

    callback(valid);
}
```

Пример 1: Пример функције која као аргумент узима вредност повратне функције

Сада се може позиву ове функције доделити као аргумент функција која ће се извршити након извршења те функције, тј. у повратку.

```
checkSession(id, function(valid){
    if(valid) {
        /* Примарна акција */
    }
    else {
        /* Обрада грешке */
    }
});
```

Пример 2: Позив функције из претходног примера

Синтакса повратних функција може врло лако да доведе до ситуације која је позната као „пакао провратних позива“ (*callback hell*) [12]. То је ситуација када постоји више угњеждених повратних функција које је тешко читати и одржавати. У наредном примеру се може видети једна рудиментарна функција за пријаву корисника која врши један упит над базом података, врши валидацију лозинке на основу хеш вредности која је смештена у бази и позива функцију која креира нову сесију.

```
db.collection('user', function (err, collection) {
    collection.findOne({username: username}, function (err, item) {
        if (item) {
            if (bcrypt.compareSync(password, item.password_hash) == true) {
                session.insertSession(session_id, userId, ipAddress,
                    function (sessionId) { /*...*/ });
            }
            else { /*...*/ }
        }
        else { /*...*/ }
    });
});
```

Пример 3: Пример једноставне функције за пријаву корисника

Већ на овом једноставном примеру може се приметити како врло лако повратни позиви могу прилично закомпликовати код сервиса. Примера ради, функција за пријаву корисника која се користи у имплементационом делу рада је много сложенија. Но ово се може решити, или барем у знатној мери олакшати, коришћењем уланчаних функција што је једна од могућности *Node*-а и програмског оквира *Express* о којем ће бити речи у поглављу 4.4.

Асинхроно програмирање има једну битну последицу када су изузеци (*exceptions*) у питању. Када је сихнроно *JavaScript* програмирање у питању, уобичајно је да се изузеци обрађују коришћењем *try-catch-finally* исказа. Међутим асинхрони модел над којим је изграђен *Node* дозвољава повратним функцијама да се изврше ван *try-catch* блока, што овај цео исказ чини прилично бескорисним. Блок *try-catch* се и даље може упротребљавати над синхроним позивима функција, али имајући у виду да се већи део *Node*-а односи управо на асинхроне позиве, јасно је да овај блок има врло мали број примена и практично и није у употреби. Постоји два начина на који се решава проблем обраде изузетака први је да се приликом позива повратног метода као аргумент прослеђује *error* објекат који означава да се десила грешка приликом извршавања функција. Он би требало да се проверава приликом сваког „изласка“ из асинхроне функције и може се обрадити случај грешке на том месту или се тај објекат може проследити даље (ако постоји још угњеждених позива).

```
db.collection('sessions', function(error, collection) {
    if(error) { /* Обрада грешке */}
    else { /* Успешно извршена операција */ }
});
```

Пример 4: Пример провере колекције из *MongoDB* базе података

Други начин да за обраду изузетака јесте да се подеси глобални руковалац догађајем (*event handler*) који проверава да ли се десио необрађени изузетак (*uncaught exception*).

```
process.on("uncaughtException", function(error) {
    console.log("Uhvaćen je izuzetak!")
});
```

Глобалне процедуре за обраду изузетака су корисне када је спречавање пада апликације. Када нису правилно обрађени изузеци апликација може бити остављена у неодређеном стању што значи да наставак рада апликације може произвести додатне грешке. Тако да је препоручљиво избегавати овај случај и имати одговарајуће провере и обраде изузетака. Уопште, необрађени изузеци као последицу имају завршетак процеса *Node* апликације и ово је један од највећих недостатака овог извршног модела.

4.3 Модули

Модули су библиотеке специфичне намене које проширују основну (*core*) функционалност оквира. Иако је језгро *Node*-а само по себи доста функционално, права снага овог програмског оквира јесте велика заједница отвореног кода која је имплементирала велики број независних (*third party*) модула. Модули су доступни за преузимање уз помоћ конзолног програма који се зове *npm* (*Node package manager*)[7]. И само језгро *Node*-а написано је користећи модуле који се аутоматски укључују у сваки програм написан за ово окружење.

[7] Библиотека пакета је доступна на адреси: https://npmjs.org

Модули се укључују у програм коришћењем функције *require*. Она као аргумент прима име модула или локацију фајла где је он смештен и, уколико путања постоји, враћа назад објекат који може да се користи као интерфејс за коришћење тог модула.

```
var express = require('express');
```

Уколико се модули укључују користећи путање ка фајловима, то се може учинити апсолутном и релативном путањом или из *node_moduls* директоријума. Ако функција не може да нађе тражени фајл, аутоматски ће покушати да дода екстензије *.js*, *.json* и *.node* на путању која је задата [12].

```
var user = require('./routes/user');
```

Уколико дође до колизије назива између модула који треба да буде укључен и неког од модула из језгра, природно, приоритет има подразумевани модул из основне библиотеке.

У практичном делу рада биће више речи о самим модулима које ће бити потребно укључити у сврхе имплементације као што су драјвери за приступ бази података, модули за сигурност, уписивање у дневник догађаја и сл.

4.4 Програмски оквир *Express*

Користећи *Node*-ове уграђене библиотеке, могу се директно имплементирати *HTTP* методе коришћењем уграђеног модула *http.* Међутим, за потребе развоја сервиса у реалним условима, испоставило се, да је непожељно увек користи функционалности ниског нивоа које пружа уграђени *http* модул. Разлог за то је што би био изнова писан исти код приликом сваке операције коју је потребно извршити на *HTTP* протоколом. Из тога је проистекла идеја да се развије један једноставни програмски оквир који ће омогућити да се олакша имплементација уобичајних операција као што су обрада *HTTP* захтева.

Филозофија *Express* програмског оквира јесте се не прилагођава специфичним захтевима које могу имати веб апликације, јер је скуп тих захтева велик и разноврстан, већ да кроз једноставност пружи подршку и могућност да сви они буду испуњени. *Express* програмски оквир је базиран на програмском оквиру *Sinatra* развијеном за језик *Ruby* и заправо пружа још један ниво апстракције над *http* модулом [12]. У наставку ће бити речи о основама и начином на који се дефинишу крајње тачке сервиса.

4.4.1 Основе и рутирање

За коришћење *Express* програмског оквира најпре је потребно учитати модул коришћењем *require* функције и иницијализовати *app* објекат. Основа апликација писаних у овом програмском оквиру јесте да се дефинишу путање (руте) униформног интерфејса веб сервиса и операције које се могу извршити над тим ресурсима. Након тога се позива

посредничка функција која ослушкује захтеве који стижу ка серверу. У наредном примеру се види једноставна апликација у којој се дефинише *GET* операција на основној (*root*) путањи сервера.

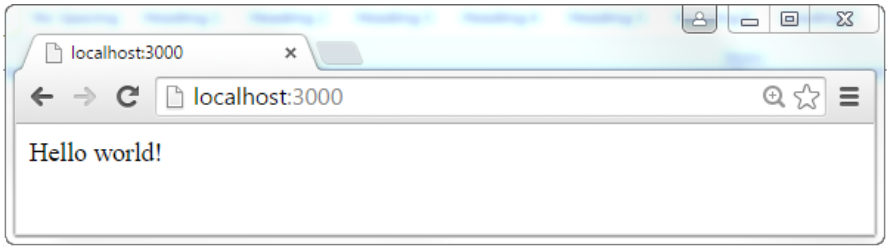
```
var express = require('express');
var app = express();

app.get('/', function(request, response){
    response.send('Hello world!');
});

app.listen(3000);
```

Пример 5: Једноставна апликација написана у *Express* програмском оквиру

Када се апликација покрене, она ослушкује захтеве на порту 3000 и могуће јој приступити из прегледача преко адресе *http://localhost:3000*.



Слика 1: Једноставна „Hello world“ апликација написана коришћењем *Express* оквира

Прегледом заглавља *HTTP* захтева, који је прегледач послао, може се видети да је позив функције *get* из *Node*-а практично пресликана верзија онога што се заиста позива из прегледача. Поред имена метода стоји релативна путања (односно рута) у апликацији, верзија протокола који се користи и на ком серверу се извршава.

```
GET / HTTP/1.1 Host: localhost:3000
```

Функција *get* узима као аргументе релативну путању ресурса и посредничку функцију у која је одговорна да разреши позив и да врати одговор клијенту. Из самог имена функције се може наслутити да она представља имплементацију метода који ослушкује позив *HTTP GET* метода на одређеним путањама и има дефинисане објекте помоћу којих може обрађивати захтеве и враћати одговоре. На сличан начин су имплементирани и остали *HTTP* везници.

Када су дефинисане све потребне путање, потребно је позвати функцију *listen*, која као аргумент узима број порта на којем сервер ослушкује захтеве.

Дакле, основну структуру *Express* апликације чине три целине: укључивање модула, дефинисање путања и посредничких фунција и позив функције за ослушкивање.

Поред везника *HTTP* протокола *app* објекат пружа још неколико врло корисних метода:

- ***app.use(name, value)*** – служи за подешавање променљивих окружења (*environment variables*) које даље *Express* може користити
- ***app.use([path], callback)*** – користи се да би се дефинисао посредник који ће руковати *HTTP* захтевима који се шаљу серверу

- ***app.VERB(path, [callback...], callback)*** – користи се да би се дефинисала функционалност за *HTTP* везнике на појединим путањама
- ***app.route(path).VERB([callback...], callback)*** – други начин да се дефинишу позиви *HTTP* метода на нивоу једне руте. *Express* подржава уланчавање ових метода, тако да је ово згодан начин да се на једном месту дефинише рута до једног ресурса и све методе које су подржане над њим
- ***app.param([name], callback)*** – служи да се веже одређена функционалност за сваки захтев који садржи одређени параметар рутирања - пример: *app.param('userId', callback)* [16].

4.4.2 Рутери

Други начин да се дефинишу крајње тачке сервиса јесте коришћењем објеката који се називају рутери. Рутери, на неки начин, врше исту фунцију као и позив *use* функције *app* објекта која дефинише путању, ослушкивач *HTTP* метода и повратну функцију у примеру на основне апликације из претходног поглавља.

Међутим, они пружају још један ниво модуларности тако што пружају могућност разбијања путања на мање целине над којима се дефинишу функције које њима руководе. Суштина је да се у корену апликације дефинишу основне путање ресурса и да се оне, даље, преусмере на рутере који су дефинисани за сваки од њих.

```
app.use('/user', router);
```

У апликацији се дефинише основна путања до ресурса *user* и прослеђује јој се рутер објекат који је одговоран за разрешавање позива ка овом ресурсу. Рутер објекат је дефинисан у засебном модулу на следећи начин:

```
var router = express.Router();

router.route('/')
    .all(/*...*/)
    .get(/*...*/)
    .post(/*...*/)
    .put(/*...*/)
    .delete(/*...*/);
```

Дефинише се путања и везници који се ослушкују. Може се приметити да је у дефиницији рутера путања дефинисана основном путањом „/“. Разлог томе је што путања која се дефинише унутар рутера је локална путања унутар путање ресурса. То значи да се она додаје основној путањи ресурса дефинисаној у *app.use()* позиву, односно да овим крајњим тачкама може се приступити коришћењем путање */user.* Може се дефинисати произвољно много локалних путања којима ће бити омогућен приступ на исти начин.

```
router.route('/:id');
```

4.4.3 Посредничке функције (*middleware*)

Основу оквира чине поновно искористиве посредничке фунције (*middleware*). Посредничка функција има три аргумента, објекте захтева (*request*) и одговора (*response*) и повратну функцију (*next*). Објекти захтева и одговора представљају омотаче (*wrapper*) истоимених објеката које су део функционалности *http* модула. Повратна функција (*next*) се позива када се комплетира извршење посредничке функције и тај позив даје сигнал оквиру да је треба да се пређе на извршење наредних функција у извршном стеку [17].

Посреднике функције се првенствено користе као функције за обраду догађаја приликом ослушкивања позива ка крајњим тачкама сервиса (*endpoint*). Поред тога, оне пружају могућност разбијања компоненти апликације на модуларне целине и у значајној мери могу решити проблеме пакла повратних позива који су инхерентни за *JavaScript*.

Начин на који се дефинишу те модуларне целине је уланчавање позива у неку врсту извршног стека. Постоји два начина да се уради то: позивом функције *all* или уланчавањем позива унутар самог позива функције.

Функција *all* – је специјална функција која ће се извршити без обзира који тип захтева стигне до рутера. Помоћу ње се може имплементирати заједничка функционалност за све типове позива, као што је провера сесије.

```
router.route('/:id')
    .all(authController.isValidSession)
    .post(userController.changePassword);
```

Ови позиви се решавају секвенцијално „наниже“, односно, у овом примеру, прво ће бити извршен позив *all(authController.isValidSession)* па онда све остале функције у стеку.

Уланчавање унутар позива функције – Поред овог „вертикалног“ уланчавања, могуће је урадити и „хоризонтално“ улачавање, односно секвенцијално позивати функције посреднике унутар самог позива функција рутера.

```
router.route('/')
    .post(authController.isUserAdministrator,
            userController.addUser);
```

Посредничке функције унутар једног позива такође се секвенцијално извршавају.

У оба ова случаја могуће је контролисати ток даљег извршења повратном функцијом *next.* Позивом *next* функције прелази се на следећу функцију у извршном стеку. Уколико треба прекинути извршење, било због појаве грешке или из неког другог разлога, могуће је у било ком тренутку вратити одговор коришћењем *response* објеката. Уколико се не пошаље одговор и не позове следећа функција, захтев ће се сматрати неуспешним након истека тајм-аут периода.

4.4.4 Прослеђивање параметара

Постоји неколико начина на који се могу проследити параметри посредничким функцијама. Први начин је коришћење именованих параметара:

```
app.get('/user/:id', ...});
```

Параметар *:id* се парсира аутоматски и смешта у објекат *request.params* који садржи све именоване параметре. Вредност овог параметра се смешта у променљиву *request.params.id.*

Поред именованих параметара, може се проследити упитни низ (*query string*) тако што се дода карактер „?“ на име путање и након тода парови *[кључ]=[вредност].*

```
/user?key1=value1&…&keyN=valueN
```

Елементима упитног низа се може приступити из посредничких функција преко *request.query* објекта на сличан начин као код именованих параметара.

4.4.5 *Request* и *response* објекти

Request објекат представља објекат који садржи све информације везане за тренутни *HTTP* захтев који долази од стране клијента. Он има следеће објекте:

- ***req.query*** – објекат садржи упитни низ (*query string*) који је прослеђен са захтевом; параметрима низа се може приступити директно: *req.query.param1, … , req.query.paramN*
- ***req.params*** – садржи све именоване параметре рутирања; присуп појединим параметрима је сличан као и код упитног низа
- ***req.body*** – садржи парсирано тело *HTTP* захтева
- ***req.path***, ***req.host***, и ***req.ip*** – су додатне информације које се могу добити о клијенту
- ***req.cookies*** – парсирање колачића које шаље клијент

Response објекат служи за слање одговора клијенту од стране сервера. Неки од његових најважнијих метода су:

- ***res.status(code)*** – који служи да се постави *HTTP* статусни код одговора
- ***res.set(field, [value])*** – служи да се подеси *HTTP* заглавље одговора
- ***res.cookie(name, value, [options])*** – постављање колачића одговора
- ***res.redirect([status], url)*** – редирекција на другу путању
- ***res.send([body|status], [body])*** – метод који се користи за не-стримујуће одговоре; метод аутоматски поставља у заглавље одговора који тип садржаја се враћа (*Content-Type*), дужину тела одговора (*Content-Length*) и сл.

- ***res.json([status|body], [body])*** – метод је идентичан *res.send()* методу када је у питању слање објеката; он форсира слање објеката у *JSON* формату, као што само име метода каже
- ***res.render(view, [locals], callback)*** – служи да се генерише поглед и врати као *HTML* одговор

5. NoSQL

У последњих десет година Интернет је поставио огроман изазов релационим базама података на начин на који нико није могао да предвиди. Да би се у потпуности могли разумети разлози тога, потребно је вратити се на саме почетке развоја релационог модела. Релациони модел је први формулисао и предложио Едгар Ф. Код[8] 1969. године [1]. У периоду када је модел предложен ресурси су били скупи, медијуми релативно непоуздани, моћ процесирања слаба. Сходно томе, велика пажња је посвећена очувању интегритета података. Он се постиже коришћењем, такозваних, *ACID* (*Atomicity, Consistency, Isolation, Durability*) трансакција:

- Атомичност (*atomicity*) – у једној трансакцији или су успешно извршене све операције или је трансакција поништена (*roll back*)
- Конзистентност (*consistency*) – резултат рада трансакције не може да остави базу података у неконзистентном стању
- Изолација (*isolation*) – догађаји који се дешавају унутар једне трансакције остају скривени од свих осталих трансакција
- Издржљивост (*durability*) – завршене трансакције морају бити толерантне на било који облик кварова и грешака у систему [18].

Релациони системи за управљање базама података (*Relational Database Management System, RDBMS*) који су имплементирани над овом парадигмом функционишу одлично када су у питању окружења која не захтевају велики број конкурентних упита над базом података. Међутим, показало да су захтеви који се често постављају у Интернет окружењу потпуно некомпатибилни са ова четири принципа и захтевају да се негде направи компромис. Тај компромис, испоставља се, своди се на следеће три особине које је Ерик Бруер[9] назвао *BASE:*

- Основна доступност (*Basic availability*) – за сваки захтев је гарантовано да ће добити одговор, било да је успешно извршена операција или порука о грешци
- „Меко стање“ (*Soft State*) – стање система се временом може променити, понекад без икакве интеракције, у циљу постизања коначне конзистентности
- Коначна конзистентност (*Eventual consistency*) – база података може тренутно бити неконзистентна, али на крају се постиже конзистентност [19], [20].

Ове три особине су послужиле као основ за развој, такозваних, *NoSQL* система за управљање базама података.

[8] *Edgar F. Codd* – Енглески информатичар
[9] *Eric Brewer* – Амерички информатичар, редовни професор на Беркли Универзитету

NoSQL покрет је настао почетком 21. века. Он је фокусиран на развијање скалабилних база података способних да опслуже милионе корисника са могућношћу пораста до случаја који је данас актуелан, а то су милијарде уређаја који су повезани на интернет. *NoSQL* је генерички термин који описује класу система за управљање базама података које карактерише не ослањање на традиционални релациони модел. Назив, изворно, потиче од чињенице да овакви системи податке не организују у табеле па, самим тим, се не користе *SQL* језик за писање упита. У међувремену назив попримио интерпретирацију „не само *SQL*“ (*not only SQL*). Ова класа система за управљање базама података настала је под утицајем развоја модерних веб технологија и растуће потребе за превазилажењем проблема који традициони релациони систем имају као што су хоризонтално скалирање, паралелизација и цена (имплементације и одржавања). Док су релационе базе података дизајниране да решавају проблеме атомичности и конзистентности података, *NoSQL* системи се труде да реше проблеме скалабилности (*scalability*) и доступности (*availability*) [19]. За поједине пословне намене где је очување интегритета података од великог значаја, релациони модел је и даље незамењив, али када је у питању медијум као што је Интернет постоји потреба за еволуцијом нових методологија, на свим пољима, па ни базе података нису изузетак.

Аспекати у којима се могу прецизније уочити разлике између релационих система и *NoSQL* система су флексибилност схема, комплексност упита, ажурирање података и скалабиност.

Схеме у релационим системима су доста круте и нефлексибилне, измене су ретке и захтевају да се предузме доста предострожности, а у неким случајевима их је и немогуће начинити. *NoSQL* системи немају то ограничење јер се податци смештају у делимично структуиране структуре којима се колоне могу додавати производно без бојазни да ће то да утиче на неки део система.

У релационим системима подаци су нормализовани у велики број табела и стога упити често морају да укључе бројна спајања што одузима и време за имплеметацију и може да буде скупоцено за извршење. У *NoSQL* системима не постоје комплексни упити зато што не постоји референцијална зависност међу различитим табелама. Ово је функционални недостатак ових система који се решава тако што, када је потребно извршити упит који укључује више табела, мора се извршити већи број мање комплексних упита. Често се овај проблем решава у самом дизајну базе података тако што се у табеле смештају денормализовани подаци који могу бити и редундатни. Остали проблеми који се могу појавити са перформансама могу се решити кеширањем података, поједностављивањем упита, и извршавањем комплексних операција на вишем нивоу (нивоу апликације). Велики број система пружа могућност уграђеног кеширања.

Ажурирање података често подразумева да се више табела мора ажурирати у једној трансакцији. То утиче доста на перформансе система јер сви уписи и измене у табеле које се користе морају да се поставе на чекање док се трансакција не заврши. *NoSQL* базе података немају имплементиран концепт трансакција. То, између осталог, подразумева да не постоје нивои изолације и то је један од компромиса о којима је било већ речи. Уопштено, *NoSQL* системи немају комплексност која је у релационим базама података неопходна и из тог разлога пружају висок степен скалабилности.

Први значајан корак у популаризацији оваквих система донео је *Google* решавајући проблеме са којима су се они сусретали док су развијали своје алгоритме за претрагу. То је изродило читаву палету решења, као што су *GFS*[10], *Chubby*[11], *MapReduce*[12] и *Big Data*[13]. Радови које је *Google* објавио везани за ове системе су, заједно са још неким сродним радовима, довели су до повећаног интересовања за дистрибуиране системе, које је нарочито било изражено у заједници отвореног кода [19]. Један од најзначајнијих резултата које је донело то интересовање је свакако развој *MongoDB* базе података.

6. MongoDB

MongoDB је докуметно-оријентисана база података која има у себи доста корисних уграђених функционалности (као што је подршка за *MapReduce* агрегацију и геоспацијалне индексе). Дизајнирна, првенствено, да буде је моћно, флексибилно и скалабилно складиште података, она комбинује могућност хоризонталног скалирања са многим корисним функционалностима релационих система. Међутим, *MongoDB* не претендује ни на који начин да замени релационе системе. Уместо тога, систем је конципиран да понуди решење за проблеме који су у том тренутку били тешки или немогући за решавање. Он пружа релативно богат скуп функционалности с обзиром на његову једноставност.

MongoDB садржи алате који нису у потпуности подржани ни у једном *NoSQL* систему. Могуће је имплементирати секундарне индексе који могу да буду сложени (да садрже више атрибута), да буду јединствени (*unique*), чак и геоспацијални. *MongoDB* подржава писање ускладиштених *JavaScript* метода који служе као функционални еквивалент ускладиштеним процедурама и функцијама које подржавају релациони системи. Механизми за агрегацију укључују *MapReduce* алгоритам као и друге алате. Поред ових функционалности, *MongoDB* има уграђени протокол за складиштење фајлова и њихових мета података. Наравно, неке функционалности релационих система недостају, попут спајања (*join*) и транскација, које је тешко имплеметирати без жртвовања скалабилности [21].

Сваки аспект *MongoDB*-а је развијен са намером да се одржи константан и висок степен перформанси. Почевши од бинарног протокола који користи за пренос података (уместо *HTTP*-а), документи се допуњавају празним простором, датотеке се алоцирају са додатном меморијом. Подразумевани механизам за складиштење који користи су меморијски мапирани фајлови, што значи да се фајлови чувају у меморији и накнадно синхронизују са копијом смештеном на диску. Тај приступ изискује додатно трошење ресурса, али добитак је значајно побољшање перформанси. Такође, систем користи динамички оптимизатор упита који има могућност памћења најбржег начина за извршење упита. И поред тога, сервер се труди да пребаци процесирање и логику на клијентску страну (било да је у питању обрада у драјверу или у коду апликација) колико је год то могуће.

[10] *Google File System* – дистрибуирани фајл систем
[11] *Chubby* – дистрибуирани систем за координацију
[12] *MapReduce* – систем за паралелно извршавање
[13] *Big Data* – база података оријентисана колонама

6.1 Модел података

Складиште докумената (*document store*) или документно-оријентисане базе података пружају могућност читања, уноса, модификовања и брисања делимично структуираних података (*semi-structured data*). Базе података се ослањају на формате као што су *XML*, *JSON*, *BSON* (бинарни *JSON*) или *YAML* за складиштење података [19]. Основна идеја је да се замени концепт „реда“ из релационог модела флексибилнијим објектом, *документом*.

Документ пружа могућност смештања комплексне хијерархије у један слог. Тај приступ омогућује да подаци буду организовани и смештени у облику који је природан програмерима који користе објектно-оријентисану парадигму.

Документи су организовани у колекције које би могле да се посматрају као аналогон табелама у релационим системима. Постоји једна важна разлика између табела и колекција. Колекције немају схеме, што значи да структура докумената у једној колекцији није фиксирана тако да сваки документ може имати различите атрибуте (колоне). Ово пружа доста флексибилности, првенствено када је питању измена модела података, додавање нових поља своди се на извршење упита који се може извршити без угрожавања рада апликације [21].

Чињеница да се користи формат сличан *JSON*-у за складиштење података чини овај систем изузетно погодним за развој веб апликација. Пошто *JavaScript* имплицитно парсира ове објекте они се користе у тачно оном облику у којем су добијени од сервера што знатно олакшава развој апликација јер нема потребе за додатном обрадом података.

Један аспект у којем се огледа предност *MongoDB* база података када се упореди са другим *NoSQL* базама података јесте једноставност његовог коришћења. Та једноставност је делимично последица тога што је добро документован систем и има велику заједницу, а делимично због тога што је доста лак за учење за некога ко има предзнање *SQL* језика.

Иако *JSON* стандард има доста предности од којих су неке покривене у ранијим поглављима, он је ипак морао бити проширен додатним типовима који су нужни за рад једне базе података. Варијанта *JSON* формата који користи *MongoDB* назива се *BSON.* Основна разлика у односу на *JSON* формат је да се подаци смештају у бинарном облику (уместо у текстуалном) у асоцијативне низове. Осим тога, *BSON* формат уводи нове типове података који нису део *JSON* стандарда [22]. У *JSON* стандарду подржани су бројеви, стрингови, логичка променљиве (*boolean*), низови, објекти и *null.* Поред ових типова *BSON* формат уводи и следеће:

- 32-битне целе бројеве (*integer*)
- 64-битне целе бројеве
- 64-битне бројеве са покретним зарезом (*floating point*)
- *symbol* – један карактер
- *object id* – јединствени 12-бајтни идентификатор објеката. Обично се смешта у облику *ObjectId("...")*

- *date* – датуми се смештају као милисекунде од почетка епохе (временска зона се не смешта)
- регуларни изрази (*regular expression*)
- код (*code*) – у документима се може чувати *JavaScript* код

```
{"x" : function() { /* ... */ }}
```

- бинарни подаци (*binary data*)
- недефинисана вредност (*undefined*) – *JavaScript* прави разлику између *null* и недефинисане вредности
- низови (*array*) – скупови или листе вредности које се представљају у облику низова
- угњеждени документи (*embedded document*) – документи могу имати угњеждене под-документе.

```
{"x" : {"foo" : "bar"}}
```

[21]

6.2 Упити

Упити се у *MongoDB* базама извршавају позивом функција из *JavaScript* конзоле која је саставни део система. Објекат преко кога се приступа изабраној бази података је *db* објекат. Синтакса за извршавање упита је облика:

```
db.[назив колекције].[назив команде]
```

Претраживање се врши коришћењем функције *find* која има неколико опционих аргумената. Може се позивати без аргумената што као резултат враћа све документе из колекције.

```
db.collection.find();
```

Први аргумент који се задаје представља *JSON* објекат који садржи критеријум претраге:

```
db.user.find({ “name”: “Aleksandar” })
```

Постоје одређена ограничења када су параметри претраге у питању. Параметри морају бити константе, тако да се, примера ради, не могу користити као параметри вредности кључева из истог докумената. Ово је могуће решити коришћењем *$match* оператора, али, уопштено гледајући, за већину примена могуће је реструктуирати незнатно документе тако да је довољно извршење једноставних упита. Други аргумент је објекат који означава колоне које треба да се налазе у резултату:

```
db.user.find({}, {“username”: 1, “role_id”: 1})
```

Постоји прегршт уграђених оператора који дефинишу додатне функционалности везане за претрагу. Неки од тих оператора ће бити описани у даљем тексту.

- Оператори поретка, *$lte*, *$gt* и *$gte* одговарају, редом, симболима <, ≤, > и ≥ и *$ne* (*not equal*) еквивалент операцији ≠
- $or оператор проверава услове задате у низу и враћа резултат ако је било који од услова испуњен:

```
db.user.find({ $or: [{“name”: “Aleksandar”}, [{“name”:
                    “Marko”}]})
```

- *$in* и *$nin* оператори проверавају да ли се вредност датог кључа налази у низу или не:

```
db.user.find({“role_id”: {$in: [1, 2, 3]}})
```

- $not је метакондиционал који представља негацију својих под-критеријума:

```
db.user.find({$not: {“role_id”: {$in: [1, 2, 3]}}})
```

Претрага угњеждених документата се може извршити на два начина, претрага целог документа или претрага по паровима (кључ, вредност). Претрага целог документа се врши прослеђивањем документа као параметра претраге. Притом, угњеждени документ мора имати иста поља као и параметар претраге да би се појавио у резултату. Други начин претраге је да се угњеждене вредности претражују на нивоу целог документа користећи синтаксу *“[кључ документа].[кључ под-документа]”.*

```
db.addresses.find({“streets.street_name”: “ Beogradska”})
```

Унос новог документа се врши позивом функције *insert*, која као аргумент прима документ који се уноси:

```
db.user.insert({username”: “Aleksandar”, “role_id”: 1})
```

Ажурирање докумената се извршава позивом функције *update* која као аргументе има критеријум по којем се претражују документи, затим израз којим се врши ажурирање и, на крају, опциони аргумент у којем се могу поставити додатне опције.

```
db.user.update(
      {“name”: “Aleksandar”},
      {$set: {“role_id”: 1}},
      {multi: true});
```

Оператор *$set* служи да се означи да треба ажурирати само поље *role_id*, ако би он изостао то би значило да ће цео објекат бити „преписан“ и уместо њега сачуван објекат је прослеђен. Аргумент *multi* представља индикатор да треба ажурирати више докумената, ако би он изостао, систем би ажурирао само први документ који пронађе.

Ажурирање и упис такође се могу извршити позивом функције *save* која као аргумент узима објекат који треба сачувати, слично као код функције *insert,* с том разликом да се

ажурирање врши ако систем успе да пронађе објекат по идентификатору (*_id*), а унос ако тај објекат не постоји или ако није прослеђен идентификатор.

Брисање се врши позивом функције *delete* која има такође један параметар за претрагу докумената. Опционо, може се додати још један *boolean* параметар који, када има вредност *true*, представља индикатор да треба обрисати само први слог који је пронађен. Подразумевано понашање функције је да се бришу сви објекти који су пронађени.

6.3 Агрегација

У овом поглаву ће укратко бити дат приказ неких од основних функционалности када је агрегација у питању као и приказ агрегационог оквира.

Функција *count* враћа број докумената који испуњавају критеријум претраге који се прослеђује.

```
db.user.count({ “role_id”: 1})
```

Функција *distinct* враћа јединствене вредности наведених атрибута у једном низу.

```
db.user.distinct(“name”)
```

Груписање се врши позивом функције *group* која пружа сличну функционалност као функција *group by* која постоји у *SQL* упитном језику. Параметри функције су *key*, *initial* и *reduce*. Параметар *key* служи да се наведе који резултати се групишу, *initial* параметром се поставља иницијална вредност за сваку групу, док *reduce* параметар означава функцију по којој се извршава груписање. Функција узима два параметра: тренутни документ у итерацији и тренутно стање бројача.

```
db.user.group({
     key: {name: “Aleksandar”},
     initial: {Total: 0}
     reduce: function(item, counter) {
          counter.Total += 1;}});
```

Једна од критика *MongoDB* система била је компликована употреба алата за агрегацију. Уграђена *MapReduce* функционалност је тешка за коришћење и у већини примена, поготово када су пословне апликације у питању, непотребна. Како би се то олакшало имплементиран је агрегациони оквир (*aggregation framework*) који омогућује да се команде уланчавају кроз јединствени канал (*pipeline*) слично као што се команде у *Linux* системима могу уланчавати [23]. Функција *aggregate* служи коришћење функционалности агрегационог оквира и она као аргумент прима низ уланчаних операција:

```
db.user.aggregate([
     {$match: {role_id: 1}},
     {$group: {_id: “$name”, total: {$sum: 1}}}]);
```

Уланчане операције се извршавају секвенцијално, у фазама. Резултат сваке фазе се прослеђује следећој операцији на обраду. У горњем примеру оператор *$match* врши претрагу колекције и прослеђује резултат оператору *$group* који врши груписање и враћа резултат.

Неки од оператора груписања су:

- ***$match*** – који врши филтрирање по прослеђеном критеријуму и враћа све документе који га задовоље
- ***$group*** – врши груписање по прослеђеном идентификатору и примењује прослеђене изразе за акумулацију. Враћа један документ за сваку групу
- ***$project*** – врши пројекцију одређених поља и служи за редуковање докумената што скраћује време извршења
- ***$unwind*** – врши деконструкцију по низу који је поље документа тако да враћа један документ за сваки елемент низа. У суштини врши неку врсту денормализације докумената

Постоји још уграђених оператора који врше ограничавање резулата (*$limit* и *$skip*), сортирање (*$sort*) и друге операције, али пошто они неће бити предмет имплементације неће бити разматрани овде. Уланчавање оператора пружа доста флексибилности и контроле приликом вршења агрегације и сасвим довољну функционалност за употребу у пословним апликацијама.

6.4 Индексирање

Индекси у *MongoDB* базама, као и код релационих система, представљају посебну структуру података који помажу уграђеном оптимизатору упита да брже претражује документе. Компромис који се ту прави је веће заузеће простора на диску и смањење перформанси приликом уноса докумената. Подразумевано је да на свакој колекцији постоји индекс по примарном кључу, односно идентификатору. Постоји неколико различитих врста индекса у *MongoDB*-у. Први је индексирање по једном пољу:

```
db.user.ensureIndex({"username" : 1})
```

где вредност 1 означава поредак по којем ће се вршити сортирање објеката у индексу (-1 означава сортирање у обрнутом поретку). Уколико не постоји индекс, претрага се врши класичном *table scan* операцијом која секвенцијално пролази кроз све документе док не пронађе одговарајуће. Такође, постоји могућност креирања сложених (композитних) индекса. Они садрже више колона:

```
db.user.ensureIndex({"username" : 1, "role_id" : 1})
```

Приликом оптимизације упита, треба индексирати сва поља која се користе као критеријум претраге, међутим мора се водити рачуна јер што је сложенији индекс то му више простора треба за складиштење. Такође, могу се индексирати поља угњеждених докумената користећи синтаксу *“[документ].[кључ]”.* Јединствени (*unique*) индекси се могу дефинисати додавањем

{"unique" : true} као другог параметра функције *ensureIndex*. Јединствени индекси су веома корисна функционалност за када постоји потреба за уносом података који морају бити јединствени на нивоу колекције, попут корисничких имена.

6.5 Аутоматска инкрементација

Један од не тако очигледних недостатака *MongoDB* база података је то што се целобројни идентификатори (и идентификатори уопште) не могу аутоматски инкрементирати. Ово јесте мали недостатак јер типично нема много бенефита у коришћењу целих бројева као идентификатора. Нарочито зато што *ObjectId* гарантује да не може доћи до колизија и пружа прилично добре перформансе. Али у неким ситуацијама је згодно имати могућност аутоматске инкрементације. Једна од таквих ситуација је када треба интегрисати *MongoDB* базу у неко окружење које подразумева постојање друге, релационе базе података са којом је потребно делити податке. Но, постоје начини да се имплементира аутоматско инкременрирање коришћењем бројачке колекције о којој ће бити речи у имплементационом делу.

7. Имплементација сервиса

Модел-поглед-контролер (*Model-View-Controller, MVC*) архитектурални образац је постао популаран и широко распрострањен последњих неколико година. Сам образац није нов, описан је још 1988. године од стране Краснера и Поупа[14] као образац за прављење корисничих интерфејса за *SmallTalk-80* [24]. Међутим, популарност је стекао 2007. године са изласком верзије 2.0 *Ruby on Rails* језика који је имао уграђену подршку за *MVC*. Уопштено, *MVC* је постао популаран зато што се уклапа савршено у вишеслојне архитектуре које карактеришу интернет. Два слоја *MVC*-а су већ инхтерентно присутна у клијент-сервер архитектури. Једино што је преостало да се уради јесте да се издвоји у имплементацији пословна логика из оркестрације (односно, контролера).

Овај рад ће покривати имплицитно *MVC* образац, иако ће у њему бити представљени опширније једино модел и делови контролера. Поглед ће моћи да се види једино кроз слике крајњег изгледа клијентске апликације које ће бити укључене у опис ресурса који су дефинисани.

[14] *Glenn Krasner* и *Stephen Pope*, амерички информатичари

7.1 Функционална спецификација

Апликација се састоји из три основне целине: модул за праћење налога за интервенције (односно тикета), модул за извештаје и администрација апликације.

Модул за тикете садржи информације о налозима за интервенције који могу бити везани кориснике, поједине локације или налози опште намене.

Администрација апликације садржи операције везане за одржавање корисничких налога за пријављивање у апликацију.

Аутентификација се врши приликом пријаве у апликацију тако што се успешно верификује лозинка корисника који се пријављује и креира се сесија која се смешта у базу података. Пре извршења било које наредне операције над базом података врши се провера сесије. Ако она не постоји, тражена операција се не извршава.

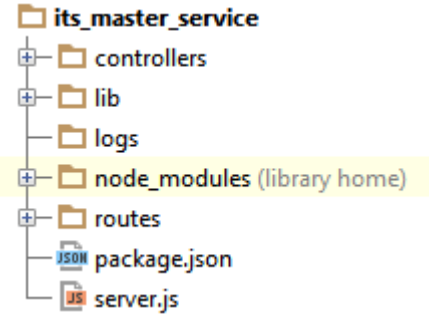
У сервису постоје два нивоа ауторизације:

- **Привилегије корисника** – односе се на привилегије за извршавање различитих операција на сервису (нпр. једино администратор има право уноса и брисања корисника). Овај ниво је реализован коришћењем улога (односно, рола) које се додељују корисницима и у овој имплементацији су препознате и дефинисане четири улоге: *радник*, *супервизор*, *менаџер* и *администратор*
- **Видљивост података** – други ниво ауторизације односи се на видљивост података која је дефинисана за различита одељења и области
    - **Одељења** – сваки корисник може да приступи само информацијама које се односе на одељење којем припада. Ово се првенствено односи на податке о тикетима и њиговим типовима.
    - **Области** – сваки корисник може да приступи само тикетима који се односе на области које је ауторизован да види или тикетима који немају унето то ограничење.

Видљивост података је далеко мањег приоритета од права приступа деловима сервиса. Стога, овај вид ауторизације се не спроводи ригорозно, већ се то постиже имплицитно слањем додатних параметара сервису кроз *URI* или у телу захтева. То значи да под одређеним условима корисник може заобићи ова ограничења и приступити свим подацима. Овај вид ауторизације је првенствено намењен да спречи да корисници буду изложени информацијама које их директно не интересују, а не као вид заштите поверљивих података.

7.2 Организација пројекта

У овом поглављу ће бити пар речи посвећених организацији пројекта сервиса и структури директоријума и фајлова које они садрже.



Слика 2: Структура пројекта

Срж сервиса, као и код већине *Node* апликација, чини фајл који се назива *server.js* који се налази у коренском директоријуму пројекта. У њему је дефинисан костур *Express* апликације као и основне путање ка ресурсима које сервис подржава. Основне путање сервиса су:

- ***auth*** – ресурс који се користи за сврхе аутентификације и ауторизације корисника
- ***user*** – представља апстракцију над колецијом истог назива, у којем су смештене информације о корисницима који приступају сервису, и излаже интерфејс за операције читања, писања, ажурирања и брисања докумената из те колекције
- ***role*** – информације о улогама које имају корисници апликације
- ***department*** – колекција у којој су смештене информације о одељењима којима могу припадати корисници апликације
- ***addresses*** – информације о адресама
- ***ticket*** – основне ПЧМБ операције над колекцијом *ticket*, као и додатне операције које прате пословну логику али их је немогуће дефинисати као чисте операције над ресурсом, као што су додељивање тикета кориснику и слично.
- ***ticket-status*** – информације о статусима тикета
- ***ticket-type*** – информације о типовима тикета
- ***material*** – ПЧМБ операције дефинисане над колекцијом која садржи материјале који могу бити искоришћени приликом решавања тикета
- ***report*** – ресурс који се, попут *auth* ресурса не односи ни на једну колекцију у бази података и служи за дефинисање приступа извештајима опште намене

Ове путање су релативне путање које усмеравају на модуле који садрже дефиниције ресурса. Те дефиниције се налазе у фајловима који су смештени у директоријуму *routes*. Директоријум садржи, у посебним фајловима за сваку основну путању, рутере *Express* оквира који даље дефинишу под-путање којим су дефинисане крајње тачке (*endpoint*) сервиса које заправо представљају ресурсе. У наставку ће бити речи о свакој од њих понаособ. Поред ових фајлова, директоријум се састоји још од *index.js* модула који пописује све појединачне модуле овог директоријума и олакшава учитавање истих у *server.js* апликацији.

```
module.exports = {
    auth: require('./auth'),
    user: require('./user'),
    addresses: require('./addresses'),
    ticket: require('./ticket'),
    ticketStatus: require('./ticket-status'),
    ticketType: require('./ticket-type'),
    material: require('./material'),
    role: require('./role'),
    department: require('./department'),
    report: require('./report')
}
```

Сваки од ових модула има сличну структуру. Дефинисан је рутер у којем се даље дефинишу релативне путање (у односу на основну путању) ка самим ресурсима.

Рутери дефинишу ресурсе и ослушкиваче *HTTP* метода које су дозвољене над њима. Сваки ослушкивач усмерава апликацију ка методу (или групи метода) које су дефинисане у контролерима, а који садрже позиве најнижег нивоа и укључују и директан приступ бази података. Контролери су смештени у директоријуму *controllers* и садрже појединачне фајлове за сваки тип ресурса, као и модуле који садрже помоћне функције најнижег нивоа које се користе на више места. У контролерима су смештене све операције над базом података и свака колекција у бази има свој контролер. О њима ће бити више речи у поглављу 7.5.

Осим рутера и контролера, постоје одређене функције којима није место у поменуте две категорије, то су пре свега неке опште фукције за парсирање података, упис у дневник догађаја, генерички објекти и слично. Овакве методе и објекти смештени су у *lib* директоријум.

У директоријуму *node_modules* смештени су сви независни модули који су преузети помоћу *Node*-овог менаџера пакета и укључени у пројекат. Више о њима и свим осталим компонентама сервиса у поглављима 7.4, 7.5 и у додатку на крају рада.

На крају, упис позива сервиса у дневнике догађаја врши се снимањем текстуалних датотеке које су смештене у *log* директоријуму.

7.3 Модел података

База података се састоји из следећих колекција:

- ***sessions***
- ***roles***
- ***addresses***
- ***departments***
- ***user***
- ***material***
- ***ticket_status***
- ***ticket_type***
- ***ticket***

- ***counters***

Информације о сесијама (*sessions*)

- *_id* : ***string*** – идентификатор сесије; који се, приликом уноса сесије, шаље назад клијентској апликацији
- *user_id* : ***string*** – идентификатор корисника
- *ip_address* : ***string*** – *IP* адреса са које се корисник пријавио
- *creation_time* : ***ISODate*** – датум и време када је сесија направљена
- *expiration_date* : ***ISODate*** – датум и време истека сесије

Улоге корисника (***role***)

- *_id* : ***integer*** – идентификатор роле
- *name* : ***string*** – име роле које је у употреби у клијентској апликацији
- *role* : ***string*** – интерно име роле
- *bit_mask* : ***integer*** – битовска маска која се користи за ауторизацију корисника

Адресе (*addresses*)

- *_id* ***: integer*** – идентификатор области
- *district* : ***string*** – назив области
- *streets* : ***array*** – низ објеката који представљају улице
    - *_id* : ***integer*** – идентификатор улице
    - *street_name* : ***string*** – назив улице
    - *numbers* : ***array*** – низ објеката који означавају бројеве улаза
        - *_id* : ***integer*** – идентификатор улаза
        - *number* : ***string*** – број улаза

Одељења (*departments*)

- *_id* : ***integer*** – идентификатор одељења
- *name* : ***string*** – назив одељења

Корисници (*users*)

- *_id* : ***ObjectId*** – идентификатор корисника
- *name* : ***string*** – име корисника
- *surname* : ***string*** – презиме корисника
- *username* : ***string*** – корисничко име
- *email* : ***string*** – мејл адреса
- *password_hash* : ***string*** – хеш вредност лозинке корисника
- *login_retries* : ***integer*** – број преосталих погрешних покушаја пријаве (подразумевана вредност је 3)
- *districts* : ***array*** – низ идентификатора области које је корисник ауторизован да види
- *role_id :* ***integer*** – идентификатор улоге корисника
- *role_mask* : ***integer*** – битовска маска која се користи за ауторизацију корисника (копија поља *bit_mask* из колеције ***roles***)

- *role* : ***string*** – интерно име роле (копија поља *role* из колеције ***roles***)
- *department_id* : ***integer*** – идентификатор одељења којем припада корисник
- *department* : ***string*** – назив одељења којем припада корисник

Материјали (*materials*)

- *_id* : ***ObjectId*** – идентификатор материла
- *code* : ***string*** – шифра материјала
- *name* : ***string*** – назив материјала
- *unit* : ***string*** – јединица мере
- *department_id* : ***integer*** – идентификатор одељења којем је на располагању материјал

Статуси тикета (***ticket_status*)**

- *_id* : ***ObjectId*** – идентификатор статуса
- *status* : ***string*** – назив статуса

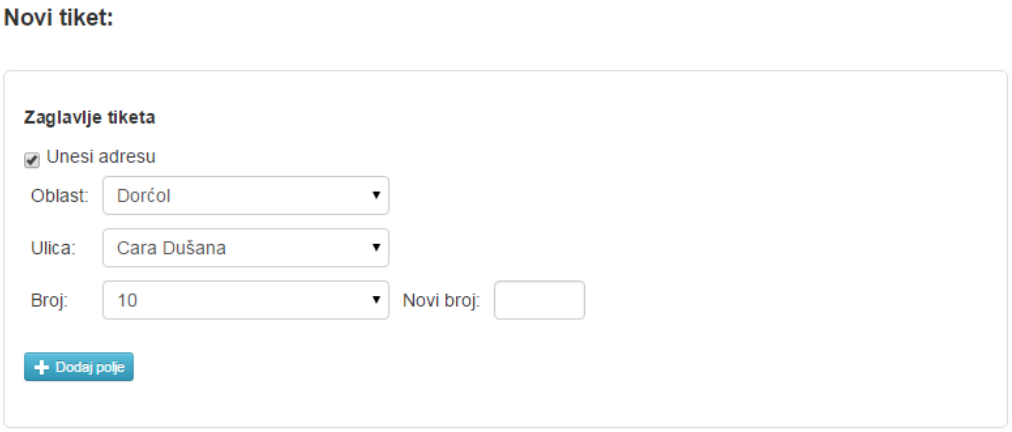
Типови тикета (***ticket_types***)

- *_id* : ***integer*** – идентификатор типа тикета
- *group* : ***string*** – типови се групишу по категоријама проблема
- *type* : ***string*** – назив типа тикета
- *department_id* : ***integer*** – идентификатор одељења
- *department* : ***string*** – назив одељења

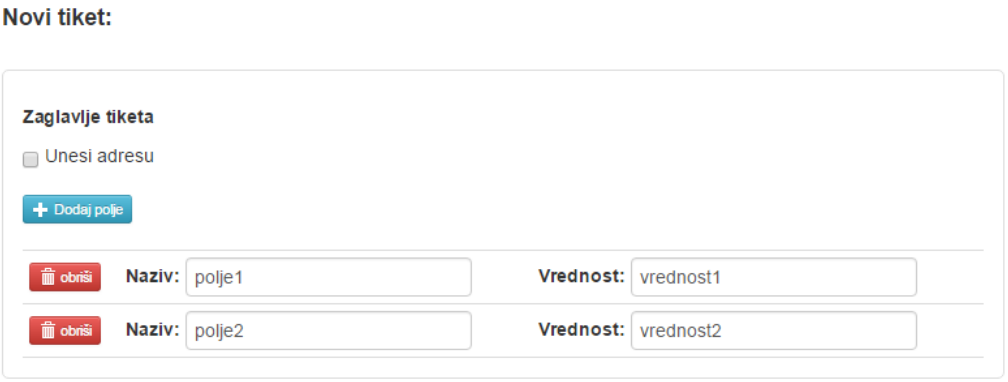
Тикети (***ticket*)**

- *_id* : ***integer*** – идентификатор тикета
- *number* : ***string*** – број тикета заправо представља идентификатор допуњен водећим нулама тако да има 6 цифара; користи се у извештајима
- *ref_number* : ***NumberLong*** – број који представља страни кључ налога у спољној бази података која се користи за унос тикета са техничким проблемима
- *type_id* : ***integer*** – идентификатор типа тикета
- *type* : ***string*** – назив типа тикета
- *status_id* : ***integer*** – идентификатор статуса тикета
- *status* : ***string*** – назив статуса тикета
- *parent_id* : ***integer*** – идентификатор тикета који је родитељ датог тикета; поставља се приликом спајања тикета. Ако је тикет независан ово поље има вредност *null*
- *header* : ***Object*** – представља генеричко заглавље тикета
    - *district_id [опционо]*: ***integer*** – идентификатор области (из колекције *addresses*)
    - *street_id [опционо]* : ***integer*** – идентификатор улице (из колекције *addresses*)
    - *entrance_id [опционо]* : ***integer*** – идентификатор улаза (из колекције *addresses*)
    - *address [опционо]* : ***string*** – пун назив адресе (улица, улаз, број и област)

    - *customer [опционо]* : ***Object*** – објекат који садржи основе информације о кориснику код кога је потребно извршити интервенцију (поставља се из спољне апликације)
        - *customer_id [опционо]* : ***integer*** – идентификатор корисника
        - *name [опционо]* : ***string*** – име корисника
        - *phone [опционо]* : ***string*** – телефон корисника
        - *mobile [опционо]* : ***string*** – мобилни телефон корисника
    - *custom_fields: [опционо]* : ***array*** – низ произвољних објеката облика *{ кључ : vrednost }* који се могу задати у клијентској апликацији
- *statuses* : ***array*** – низ објеката који представљају статусе тикета
    - *status* : ***string*** – назив статуса
    - *comment* : ***string*** *[опционо]* – коментар везан за статус. Не уноси се приликом промене статуса на „додељен“
    - *assigned_to :* ***Object*** *[опционо]* – објекат који представља информације о кориснику коме је додељен тикет. Уноси се једино ако је статус који се уноси „додељен“.
        - *_id* : ***ObjectId*** – идентификатор корисника
        - *username* : ***string*** – корисничко име корисника коме је додељен тикет
    - *user* : ***string*** – име корисника који је унео статус
    - *date* : ***ISODate*** – датум уноса
- *materials* : ***array*** – низ објеката који представљају утрошене материјале који су искоришћени приликом решавања тикета
    - *_id* : ***ObjectId*** – идентификатор материјала (из колекције *material*)
    - *code* : ***string*** – шифра материјала (из колекције *material*)
    - *name* : ***string*** – назив материјала (из колекције *material*)
    - *unit* : ***string*** – јединица мере (из колекције *material*)
    - *quantity* : ***integer*** – количина изражена у задатој јединици мере
- *signatures* : ***Object*** – објекат који се уноси приликом издавања радног налога и садржи имена потписника документа
    - *worker* : ***string*** – радник који потписује радни налог
    - *customer* : ***string*** – корисник који потписује радни налог
- *assigned_to* : ***Object*** – објекат који чува информације о кориснику којем је додељен тикет; може имати вредност *null* ако тикет није додељен
    - *_id* : ***ObjectId*** – идентификатор корисника
    - *name* : ***string*** – име корисника
- *level* : ***integer*** – број који одређује ниво интервенције према интерним шифарницима
- *closed* : ***boolean*** – индикатор да ли је тикет затворен или не
- *department_id* : ***integer*** – идентификатор одељења којем је додељен тикет
- *change_date* : ***ISODate*** – датум последње измене тикета



Слика 3: Унос адресе у заглавље тикета



Слика 4: Унос нових поља у заглавље тикета

Бројачи (*counters*) – ова колекција представља начин да се имплементирају целобројни идентификатори који се аутоматски инкрементирају

- *_id*: ***string*** – идентификатор бројача представљен именом поља
- *seq* : ***integer*** – тренутни највећи идентификатор за дато поље у бази података

7.4 Помоћне функције и објекти

Директоријум *lib* садржи помоћне фукције и објекте који ће бити овде објашњени. Најважније од њих су *JSONResponse*, *access-control* и *logger* модули.

Одговор сервера (*JSONResponse*) - Добра пракса код писања *REST* сервиса јесте да се одговори који се добијају од стране сервера буду форматирани на јединствен начин. Дефинисање структуираног објекта који ће се враћати код сваког одговора јесте један начин да се то уради.

```
module.exports = function(success, message, code, data) {
    this.Success = success;
    this.Message = message;
```

```
        this.code = (typeof code === 'undefined') ? (success ? 200 : 500) : code;
        this.data = (typeof data === 'undefined') ? {} : data;
    };
```

Објекат има четири атрибута који чине сваки одговор који сервис шаље. То су логичка променљива *success* која се има тачну вредност ако је захтев извршен без грешака, а нетачно ако је дошло до грешке. Атрибут *message* узима вредност текстуалне поруке, која може бити или потврда да је операција успешно извршена или порука о грешци. Атрибут *data*, уколико је постављен, садржи податке које сервер враћа назад клијентској апликацији обично када је операција читања у питању. У њему може бити смештен *JSON* објекат произвољног садржаја. И коначно, атрибут *code* представља статусни код *HTTP* одговора који сервер шаље. Он је додат, првенствено због комплетности поруке (јер се статусни код орговора већ налази у заглављу одговора), али може имати корисне намене на клијентској страни (као што је редирекција на странице грешке и слично). Атрибути *data* и *code* су опциони атрибути.

Овако дефинисан објекат се може иницијализовати позивом:

```
new JSONResponse(true, 'Uspešno izvršeno!', 200, { value: "" }),
```

што креира *JSON* објекат који са следећом структуром:

```
{ Success: true,
  Message: 'Uspešno izvršeno!',
  code: 200,
  data: { value: "" }}
```

Овај објекат се прослеђује у сваком телу одговора.

Контрола приступа (*access-control*) – представља конфигурациони модул који дефинише нивое приступа за улоге које постоје. Објекат *userRoles* садржи називе улога које постоје у бази података и њигове битовске маске.

```
var userRoles = {
    user: 2,
    supervisor : 4,
    manager: 8,
    admin: 16
}
```

На основу овога може се конструисати следећи објекат који дефинише нивое приступа у апликацији:

```
var accessLevels = {
    user:  { bitMask:
                        userRoles.user |
                        userRoles.supervisor |
                        userRoles.admin },

    supervisor: { bitMask:
                        userRoles.supervisor |
                        userRoles.manager |
                        userRoles.admin },

    manager: { bitMask:
                        userRoles.manager |
                        userRoles.admin },
```

```
    admin: { bitMask:
                            userRoles.admin }
}
```

Овај објекат дефинише ниво коме се приступа и битску дисјункцију улога које могу да му приступе. На овај начин, провера да ли корисник има привилегије одређеног нивоа своди се на проверу битске конјункције маске нивоа и маске корисника (која се налази у документу сваког корисника).

На крају, модул враћа објекат нивоа приступа на уобичајан начин:

```
exports.accessLevels = accessLevels;
```

Овај модул се учитава у глобалну променљиву *global.accessLevels*, којој се може приступити из целог сервиса.

Упис у дневник догађаја (*logger*) – У сервису је имплементирана функционалност евидентирања свих одговора које сервер шаље клијентима. Упис у дневник је организовано тако што се снимају дневне датотеке евиденције (чији су називи облика *its_service.log.dd.MM.yyyy*) у *logs* директоријум који се налази у корену апликације. Евидентирање је имплементирано коришћењем независног модула који се зове *winston.*

```
var logger = require('winston');

logger.add(logger.transports.DailyRotateFile, {
    filename: '../logs/its_service.log',
    datePattern: '.dd.MM.yyyy'
});
```

Овај модул пружа могућност креирања датотека у облику *JSON* објеката који имају следећу структуру:

```
{[тип уноса]: [порука], [опциони објекат]}
```

Winston модул има дефинисана три подразумевана типа уноса: информација (*info*), упозорење (*warn*) и грешка (*error*). По потреби, могуће је дефинисати и друге типове, али за потребе овог рада довољна је подразумевана функционалност.

У опционом објекту који се прослеђује у имплментационом делу биће уобичајно да се шаљу информације у ком модулу се позив десио и који су били параметри, док ће у тексту поруке бити наглашено која је функција покренута.

```
logger.info('changePassword: Password changed successfully!', {
        module: 'user.js',
        params: {userId: id }});
```

У наставку ће бити подразумевано да се приликом сваког слања одговора од стране сервера позива експлицитно одговарајућа функција *logger* објекта која врши упис операције која је извршена. Такође, сваки позив функције за евидентирање се исписује у конзоли сервиса, тако да је могуће и пратити извршавање сервиса у реалном времену.

7.4.1 Интерне функције у контролерима

Поред функција које су изложене кроз позиве сервиса, контролери могу садржати и интерне функције које су у употреби у осталим деловима сервиса, те је важно да овде буду описане.

Провера привилегија корисника (*hasUserAccess*) – интерна функција која чита податке о кориснику из колекције *user* и врши проверу, битском дисјункцијом, да ли корисник има задату привилегију или не.

Улазни параметри:

- *id* – идентификатор корисника
- *access_level* – ниво приступа за који проверавамо
- *callback* – повратна функција која враћа логичку вредност да ли корисник има задате привилегије или не

Провера:

```
db.collection('user', function(err, collection) {
    collection.findOne({_id: id}, function(err, item) {
        if(item) {
            var permission = item.role_mask & access_level.bitMask;
            callback(permission);
        }
        else
            callback(false);
    });
});
```

Провера супервизорских привилегија (*isUserSupervisor*) – је функција која врши проверу да ли дати корисник има супервизорска права. Ова функција користи интерну функцију *hasUserAccess*, а разликује се од ње по томе што је она функција је изложена клијенским позивима сервиса. Другим, речима, она се користи као посредничка функција у рутерима.

Улазни параметри:

- *req.query.UserId*

Функција:

```
var id = new ObjectId(req.query.UserId),
    accessLevels = global.accessLevels;

hasUserAccess(id, accessLevels.manager, function(permission) {
    if(permission)
        next();
    else
        res.status(401).json(/*...*/);
});
```

Поред ове дефинишу се функције за нивое менаџер и администратор, које су аналогне овој функцији.

7.5 Ресурси

У овом одељку ће бити речи о ресурсима који су дефинисани у сервису. Али, пре тога, потребно је неколико напомена о начину на који ће код бити представљен у наставку овог одељка.

У наставку ће бити подразумено два сваки захтев у крајњим гранама извршења враћа клијенту одговор о успешно извршеној операцији или о насталој грешци форматиран у облику *JSONResponse* објекта.

Приликом интеракције са базом података користећи *mongodb* модул, стандардна обрада грешака подразумева враћање *error* објекта. Надаље ће се подразумевати да је начин на који се обрађују овакве грешке слање поруке о грешци и прекидање даљег тога извршења функције (или везаних функција).

У упитном низу сваког захтева који клијент шаље серверу налази се идентификатор сесије који је клијент добио након успешне пријаве. Свака провера сесије на страни сервера, врши се читањем овог параметра и претрагом актуелних сесија смештених у бази података.

Ови случајеви неће бити експлицитно наведени приликом описа функционалности у циљу постизања краћег и концизног записа. Такође, функције углавном неће бити представљене у целости из истог разлога, већ ће бити дискутовано о њима парцијално кроз најзначајније целине.

7.5.1 Аутентификација корисника

Аутентификација корисника је дефинисана у модулима ***routes/auth.js*** и његовом контролеру, модулу ***controllers/auth.js***.

auth – ресурс који служи за аутентификацију корисника, креирање и брисање сесије. Ово заправо представља одступање од *REST* приступа, али је ово одступање широко распрострањено и у том погледу се често праве изузеци.

```
router.route('/')
    .get(authController.checkSession)
    .put(authController.validateParameters,
            authController.authenticateUser,
            authController.createSession)
    .delete(authController.logoffUser);
```

Овај ресурс има дефинисане следеће методе:

GET – метод се користи за проверу да ли постоји сесија. У упитном низу позива уносе се подаци о идентификатору сесије и *IP* адреси одакле је захтев послат.

Улазни параметри:

- *req.query.SessionId* – идентификатор сесије
- *req.query.IPAddress* – *IP* адреса одакле је послат захтев

Провера сесије (*checkSession*) – функција проверава да ли у колекцији *sessions* постоји унос који одговара задатом идентификатору сесије и *IP* адреси а да датум истека сесије није застарео (односно да је вредност строго већа од тренутног момента).

```
db.collection('sessions', function(err, collection) {
    collection.findOne({
            _id: session_id,
            ip_address: ip_address,
            expiration_date: {$gt: new Date()}},

            function(err, item) { /*...*/ });
    });
}
```

PUT – се користи за аутентификацију корисника и унос сесије и састоји се из три функције, *validateParameters*, *authenticateUser* и *createSession*, које се секвенцијално позивају и које, као што њихова имена говоре, служе, редом, за валидацију улазних параметара, аутентификацију корисника и унос сесије у базу података и враћање *user* објекта који је у употреби у клијенској апликацији, као и идентификатора креиране сесије.

Улазни параметри:

- *req.body.Username* – корисничко име
- *req.body.Password* – лозинка корисника
- *req.body.IPAddress* – *IP* адреса одакле је стигао захтев

Валидација улазних параметара (*validateParameters*) – функција проверава да ли су у телу захтева дефинисани корисничко име и лозинка. Ако јесу прелази се на следећу функцију у ланцу, а ако нису враћа се порука о грешци.

Аутентификација корисника (*authenticateUser*) – врши претрагу колекције *user* по корисничком имену и враћа објекат који садржи документ тог корисника из базе података. Ако не постоји корисник, клијенту се шаље порука о грешци. Након успешног учитавања корисника, врши се провера да ли је налог блокиран (провером вредности *login_retries*), ако није врши се провера хеш вредности лозинке са унетом лозинком коришћењем функције *compareSync* модула *bcrypt-nodejs* [15].

```
bcrypt.compareSync(password, item.password_hash)
```

Ако је ова провера успешна прелази се на фукнцију која уноси сесију у базу података и враћа у телу одгвоора информације о кориснику које су потребне клијенту уз идентификатор сесије. Да

[15] Више информација у додатку на крају рада

би се избегло поновно читање информација о кориснику из базе података, с обзиром да је већ учитан документ са овим подацима да би се проверила лозинка, може се формирати објекат који ће бити додат у *req* објекат и прећи на извршење последње функције. Пошто овај објекат деле све уланчане функције, ово представља згодан начин да се интерно прослеђују објекти између функција у истом извршном стеку. Овом објекту се приступа на уобичајан начин у наредној функцији у ланцу.

```
req.user_data = {
    _id: userId,
    name: username,
    role_mask: role_mask,
    email: email,
    districts: districts,
    department_id: department_id };
```

Међутим, уколико није прошла провера лозинке потребно је ажурирати документ корисника тако што ће бити смањен број његових преосталих покушаја пријаве и биће враћена порука о грешци која има *HTTP* статусни код 401 (неауторизован клијент).

```
login_retries = login_retries - 1;

db.collection('user').update({

    {_id : userId},
    {$set: { login_retries: login_retries }

}, function(err, object) {
       if(err) {/*...*/}
       else
           res.status(401).json(/*...*/);
});
```

Унос сесије (*createSession*) – у функцији се врши унос документа у колекцију *sessions* и поставља се број поновних покушаја пријаве корисника на подразумевану вредност. Идентификатор сесије се формира коришћењем модула *crypto.* Конкретно, идентификатор сесије је хеш вредност добијена применом *SHA1* алгоритма на насумични низ од 20 бајтова. Обе функције су уграђене унутар *crypto* библиотеке.

```
var seed = crypto.randomBytes(20);
var id = crypto.createHash('sha1').update(seed).digest('hex');
```

Након формирања идентификатора, постављају се датуми почетка и истека сесије (сесија је валидна до 24 часа након креирања) и у базу се уноси објекат облика:

```
{ _id : id,
  user_id: userId,
  ip_address: ipAddress,
  creation_time: created,
  expiration_date: expires }
```

После успешног уноса сесије, поставља се подразумевана вредност поновних погрешних покушаја пријаве на 3, формира се одговор који у свом телу садржи објетак *User*, добијен из објекта *req.user_data* прослеђеног из претходне функције, као и *SessionId* вредности која је формирана и унета у базу.

7.5.2 Корисници

Постоје два ресурса која су дефинисана као апстракција колекције *user.* Први је ресурс који је дефинисан основном путањом сервиса (*/user*) која дефинише операције читања и уноса у корисника у базу. Други ресурс се односи на појединачне кориснике и има дефинисане операције за читање, ажурирање података и хеш вредности лозинке, и брисање корисника. Он је везан за путању */user/:id*, где *:id* представља идентификатор корисника. За измену података над овим ресурсом потребно је да корисник има администраторске привилегије.

user – за приступ овом ресурсу је обавезно је да корисник буде пријављен. То је реализовано кроз позив функције *all* која се извршава за све позиве над овим ресурсом и која позива метод из контролора аутентификације који проверава да ли постоји сесија.

```
router.route('/')
    .all(authController.isValidSession)
    .get(userController.getUsers)
    .post(authController.isUserAdministrator, userController.addUser);
```

GET – метод врши претрагу корисника.

Улазни параметри:

- *req.query.RoleId* – опциони параметар који, ако је постављен враћа кориснике који имају задату ролу
- *req.query.DepartmentId* – опциони параметар који, ако је постављен враћа кориснике који припадају задатом одељењу

	Korisničko ime	Ime i prezime	Odeljenje	Uloga
izmeni obriši	admin	Administrator	Tehnika	Administrator
izmeni obriši	milan	Milan Đorđević	Tehnika	Administrator
izmeni obriši	milan_r	Milan Đorđević	IT	Radnik
izmeni obriši	milan_s	Milan Đorđević	Tehnika	Supervizor
izmeni obriši	radnik	Radnik	Tehnika	Radnik

Слика 5: Преглед корисника

POST – метод има улогу уноса новог корисника.

Улазни параметри:

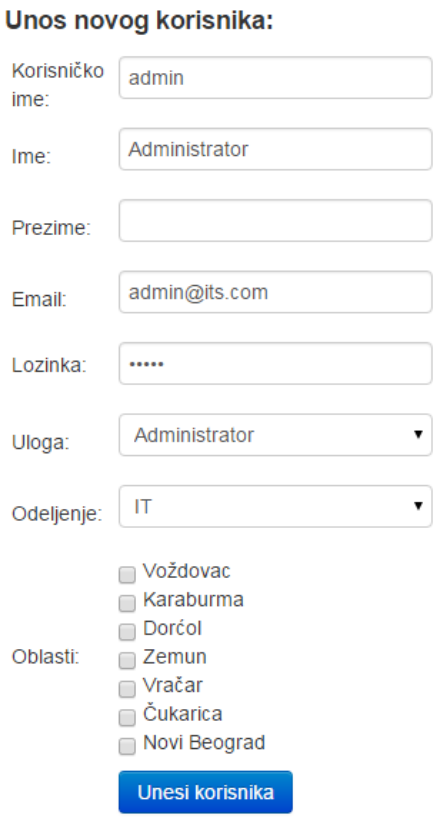
- *req.body.Name* – име корисника
- *req.body.Surname* – презиме корисника
- *req.body.Username* – корисничко име
- *req.body.Password* – лозинка
- *req.body.Email* – мејл адреса
- *req.body.Districts* – низ области којима припада корисник

- *req.body.RoleId* – идентификатор улоге корисника
- *req.body.RoleMask* – битска маска улоге
- *req.body.Role* – назив улоге
- *req.body.DepartmentId* – идентификатор одељења
- *req.body.Department* – назив одељења

Сам унос корисника, подразумева читање улазних параметара и формирање хеша за задату лозинку у текстуалном објекту. За хеширање је одгвооран модул *bcrypt*.

```
var salt = bcrypt.genSaltSync(10);
var password_hash = bcrypt.hashSync(password, salt);
```

Функција *genSaltSync* генерише такозвано „зрно соли“ (*salt*), односно насумични низ катактера који се додаје лозинци пре хеширања. Функција *genSaltSync* узима као параметар број пролаза кроз алгоритам. Већи број пролаза гарантује већи степен насумичности овог низа. Коначно, функција *hashSync* генерише хеш вредност унете лозинке и зрна соли које је генерисано за њу.



Слика 6: Унос корисника

Након тога, дефинише се објекат *user* и врши се унос корисника у базу података.

```
var user = {
    name: name,
    surname: surname,
    username: username,
    email: email,
    password_hash: password_hash,
    login_retries: 3,
    districts: districts,
```

```
    role_id: roleId,
    role_mask: roleMask,
    role: role,
    department_id: departmentId,
    department: department
};

db.collection('user').insert(user, function (err, item) {/*...*/})
```

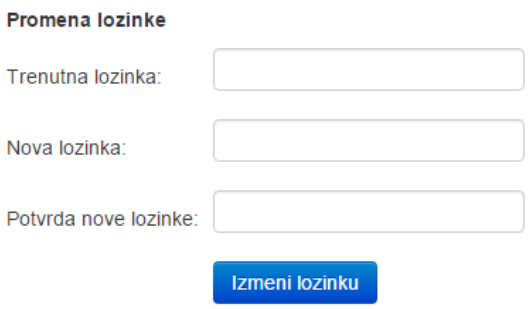
user/:id – ресурс излаже операције које су омогућене над појединачним корисницима. То су опрације промене лозинке, промене података и брисање корисника. Рутер овог ресурса садржи следеће методе:

```
router.route('/:id')
    .all(authController.isValidSession)
    .post(userController.validatePassword,
            userController.changePassword)
    .all(authController.isUserAdministrator)
    .put(userController.changeUserData)
    .delete(userController.deleteUser);
```

POST – метод дефинише измену лозинке корисника[16]. Измена лозинке се врши из два корака. Први је провера тренутне лозинке и валидација нове лозинке, а други је ажурирање базе података са новом вредношћу.

Улазни параметри:

- *req.params.id* – идентификатор корисника
- *req.body.Password* – тренутна лозинка
- *req.body.NewPassword* – нова лозинка
- *req.body.NewPasswordConfirm* – потврда нове лозинке



Слика 7: Промена лозинке

Провера тренутне лозинке (*validatePassword*) – ова функција врши позив интерне функције контролера аутентификације која враћа резултат претраге корисника по идентификатору који је прослеђен у телу захтева. Затим се врши провера хеш вредности на начин који је већ описан у функцијама за пријаву корисника. Ако хеш вредност одговара послатој лозинци, прелази се на валидацију нове лозинке.

[16] *REST* архитектура предвиђа могућност коришћења *POST* метода за специфичне операције које нису ПЧАБ операције, о чему је већ било речи.

Валидација лозинке састоји се из провере дужине лозинке (подразумевано је да лозинка има између 5 и 12 карактера) и поређење лозинке са потврдом. Валидација дужине врши се коришћењем модула *validator*, позивом:

```
validator.isLength(new_password, 5, 12)
```

Уколико је валидација успешно извршена, прелази се на следећу функцију.

Измена лозинке (*changePassword*) – Генерише се хеш вредност лозинке, на начин који је описан у методи за унос новог корисника и затим се ажурира база података позивањем функције која ажурира документ у бази података и клијенту се шаље одговарајући одговор.

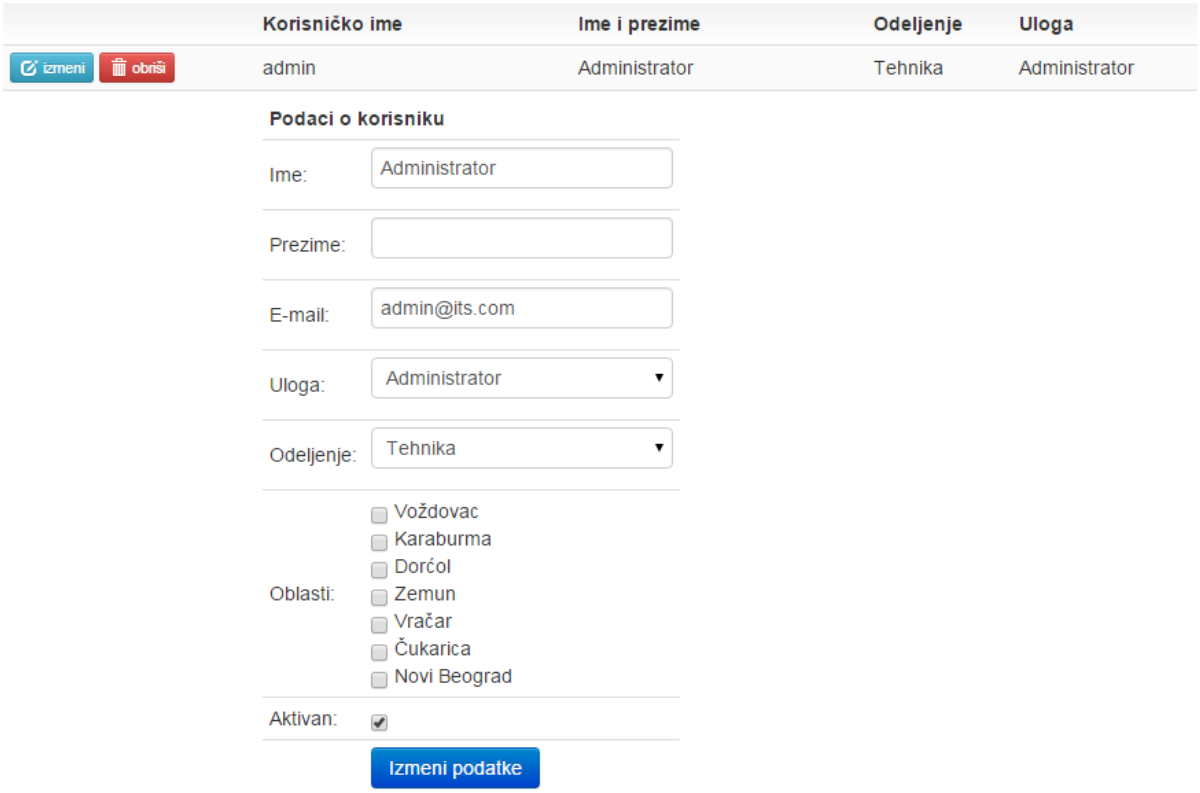
```
db.collection('user')
        .update( {_id : id},
                 {$set: { password_hash : password_hash }}
    , function(err, object) { /*...*/ });
```

Наредна два метода која су дефинисана (*PUT* и *DELETE*) захтевају да корисник има најмање супервизорске привилегије, што се осигурава позивом функције *all* која се извршава у случају оба позива.

PUT – методом се дефинише ажурирање података корисника које се реализује позивом *changeUserData* функције. Пре ажурирања података ради се валидација улазних параметара на серверској страни позивом функције *validateParameters.*

Улазни параметри:

- *req.params.id* – идентификатор корисника
- *req.body.Name* – име корисника
- *req.body.Surname* – презиме корисника
- *req.body.Email* – мејл адреса корисника
- *req.body.Districts* – области
- *req.body.RoleMask* – битска маска којом се одређује ниво привилегија корисника
- *req.body.Role* – назив улоге коју корисник има
- *req.body.DepartmentId* – идентификатор одељења којем корисник припада
- *req.body.Department* – назив одељења којем корисник припада
- *req.body.Active* – индикатор да ли је корисник активан или не



Слика 8: Измена корисника

Валидација параметара (*validateParameters*) – функција врши валидацију података користећи уграђене методе модула *validator*. Параметри, име и презиме корисника треба да се састоје само из великих и малих слова, битовска маска и идентификатор одељења морају бити цели бројеви, индикатор да ли је корисник активан мора бити логичка константа.

```
if((validator.isEmail(email) || !(data.param = 'email'))
    && (validator.isAlpha(name) || !(data.param = 'name'))
    && (validator.isAlpha(surname) || !(data.param = 'surname'))
    && (validator.isInt(roleMask) || !(data.param = 'roleMask'))
    && (validator.isInt(departmentId) || !(data.param = 'departmentId'))
    && (validator.isBoolean(active) || !(data.param = 'active'))) {
    next();
}
```

Може се приметити да се не врши валидација свих параметара. Разлог за то је што су неки од параметара представљају дупликате података из других табела (називи одељења и улога) који се аутоматски постављају. Ови параметри не учествују у упитима и њихова сврха је, првенствено да служе презентационом слоју клијента и, стога, нема потребе вршити валидацију над њима.

Измена података (*changeUserData*) – ова функција формира објекат *user* који садржи колоне које се ажурирају. Такође, поставља се вредност параметра *login_retries* на подразумевану вредност уколико је тачан индикатор *active.* Врши се ажурирање колекције *user* и клијенту се враћа одговор о успешно урађеној операцији.

```
var user = {
    name: name,
    surname: surname,
    email: email,
    districts: districts,
    role_mask: roleMask,
    role: role,
    department_id: departmentId,
    department: department };

if(active) user.login_retries = 3;

db.collection('user').update(
    { _id: user_id },
    { $set: user }, function(err, item){ /*...*/ })
```

DELETE – метод се користи за брисање документа корисника. Позива се функција *remove* која претражује колекцију и брише документ.

Улазни параметри:

- *req.params.id* – идентификатор корисника прослеђен кроз *URI* захтева.

```
db.collection('user', function(err, collection) {
    collection.remove({_id: user_id}, function(err, item) { /*..*/ });
});
```

7.5.3 Улоге корисника и одељења

Ресурс који описује улоге корисника дефинисан је као апстракција колекције *role* и пружа једино могућност читања улога које корисници могу имати, са ограничењем да корисници морају бити пријављени. Рутер је дефинисан следећим функцијама:

```
router.route('/')
    .all(authController.isValidSession)
    .get(roleController.getRoles);
```

Слично као улоге и одељења су дефинисана само за читање. Ресурс је апстракција колекције *departments* следећим рутером:

```
router.route('/')
    .all(authController.isValidSession)
    .get(departmentController.getDepartments);
```

Како ове функције описују операције читања без параметра, нема потребе давати њихов ближи опис.

7.5.4 Адресе

Адресе се дефинишу ресурсом *addresses* који представља апстракцију истоимене табеле. Над овом табелом, слично као код одељења и улога, дефинисана је само операција читања, али због специфичности структуре колекције и операција које су потребне, заслужује да буде детаљније описана овде.

```
router.route('/')
    .all(authController.isValidSession)
    .get(addressController.getAddresses, addressController.getDistricts);
```

GET – врши учитавање адреса на основу низа области. Као што се може приметити у дефиницији рутера, операција читања има две повезане методе. Једна је намењена за читање адреса, док друга служи за читање области, које представљају подскуп адреса. Ово су две операције које имају различите намене, али како су дефинисане као операције над истим ресурсом, има смисла да буду смештене на истом месту. Начин на који су ови повезани позиви реализовани биће јаснији ако се погледају улазни параметри.

Улазни параметри:

- *req.query.UserId* – идентификатор корисника који врши читање
- *req.query.RestrictDdistricts* – опциони параметар који представља означава да претрагу треба ограничити на области

Читање адреса (*getAddresses*) - врши проверу да ли је постављен параметар *RestrictDistricts* и, ако јесте, прелази се у извршавање функције *getDistricts*. Ако није постављен тај параметар, области се читају из колекције *user* за корисника који је пријављен.

```
db.collection('addresses', function(err, collection) {
    collection.find({_id: {$in: item.districts}})
        .sort({
               district: 1,
               "district.streets.street_name" : 1,
                "district.streets.street_name.numbers" : 1 },
    function(err, item){ /*...*/ })});
```

Резултат упита се сортира по именима области, као и по угњежденим вредностима имена улица и по бројевима. Дакле, овде се може видети на који начин се објекти могу сортирати по под-документима.

Читање области (*getDistricts*) – функција врши читање свих области из базе. Ова функција је у употреби када је потребно попунити листу области приликом уноса и измене податке о корисницима.

```
db.collection('addresses', function(err, collection) {
    collection.find({}, {"_id" : 1, "district" : 1},
  function(err, item) { /*...*/});
});
```

У другом аргументу функције *find* поставља се објекат који означава који атрибути ће се налазити у објекту који ће бити враћен из базе података. У овом случају потребни су само идентификатор и назив области.

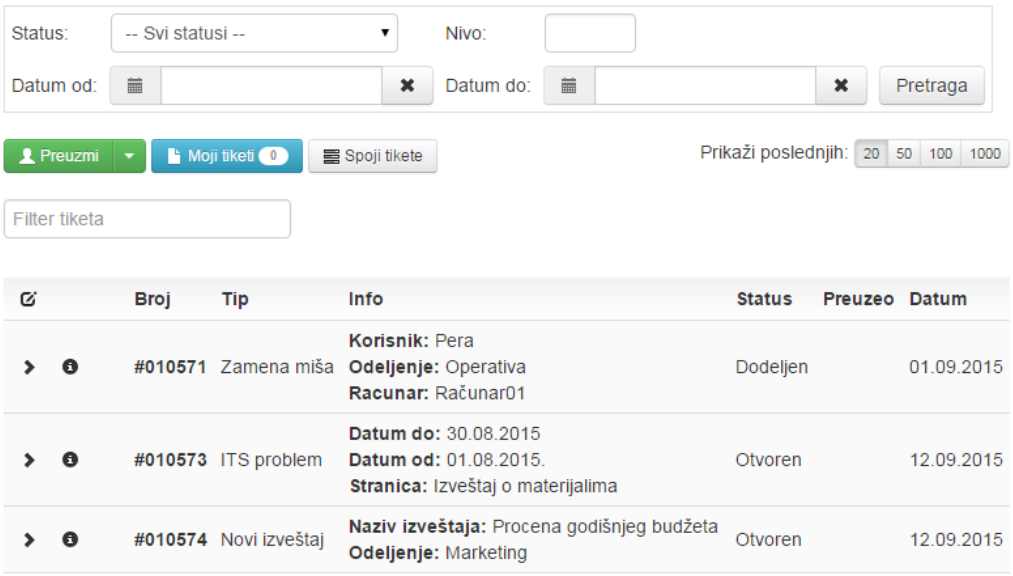
7.5.5 Тикети

Дефинисана су четири ресуса везана за тикете. Они дефинишу операције над појединачним тикетима, групом тикета или неким под-документима који припадају једном тикету.

ticket – дефинише операције претраге и уноса тикета.

```
router.route('/')
    .all(authController.isValidSession)
    .get(ticketController.buildQueryObject, ticketController.getTickets)
    .post(ticketController.addTicket);
```

GET – метод служи за претраживање тикета и састоји се из две целине: изградње упитног објекта и саме претраге.



Слика 9: Претрага тикета

Улазни параметри:

- *req.query.StatusId* – идентификатор статуса тикета
- *req.query.MyTickets* – индикатор да ли резултат треба да садржи само тикете додељене кориснику који врши претрагу
- *req.query.UserId* – идентификатор корисника
- *req.query.Level* – ниво тикета
- *req.query.DateFrom* – доње ограничење датума измене тикета
- *req.query.DateTo* – горње ограничење датума измене тикета

- *req.query.Districts* – низ идентификатора области
- *req.query.DepartmentId* – идентификатор одељења
- *req.query.TicketIds* – низ идентификатора тикета (ако је потребно вратити само одређене тикете)
- *req.query.Limit* – ограничење броја тикета који се враћају у резултату

Сви параметри који се шаљу су опциони и могу се изоставити по потреби.

Изградња упитног објекта (*buildQueryObject*) – врши валидацију улазних параметара и на основу њих формира упитни објекат који прослеђује функцији за претрагу путем *req.queryObject.*

Претрага тикета (*getTickets*) – Преузима упитни објекат из захтева, врши проверу да ли ограничења резултата (ако није задат број ограничава се резултат на 100 докумената) и врши претрагу колекције *ticket.*

```
var query = req.queryObject;
var limit = parseInt(req.query.Limit);

if(!(limit > 0)) limit = 100;


db.collection('ticket', function(err, collection) {
    collection.find(query, { /* пројекција */ })
        .sort({ change_date: 1 }).limit(limit)
        .toArray(function(err, item) {
        /*...*/
    });
});
```

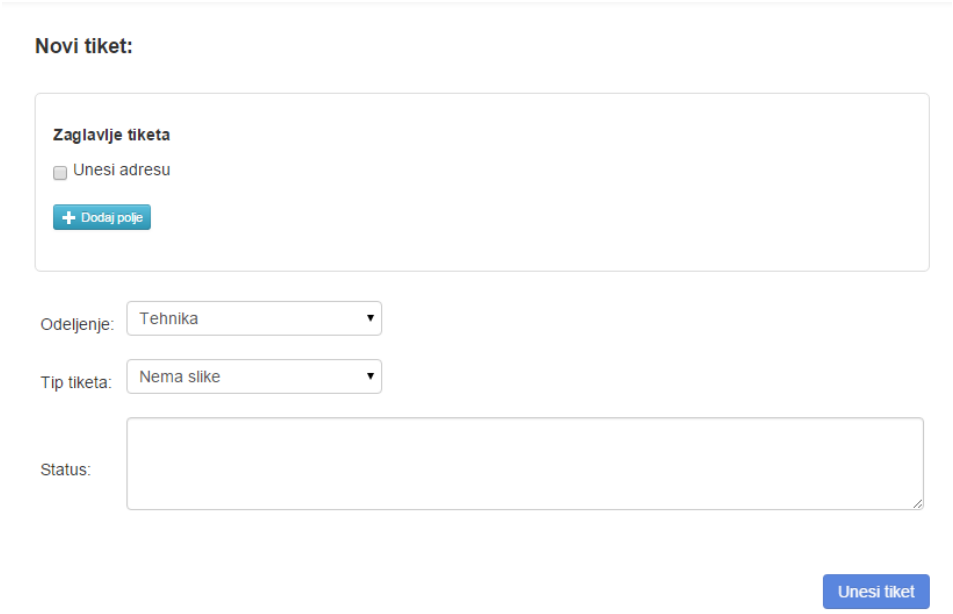
Резултат је ограничен на следеће атрибуте (задате као други параметра функције *find*):

```
{ status_id: true,
  status: true,
  statuses: true,
  number: true,
  type: true,
  header: true,
  assigned_to: true,
  has_children: true,
  change_date: true }
```

POST– метод је одговоран за унос новог тикета.

Улазни параметри:

- *req.body.Ticket* – објекат који садржи репрезентацију тикета.
- *req.body.ChildIds* – низ идентификатора тикета којима унети тикет треба да постане родитељ



Слика 10: Станица за унос новог тикета

Унос тикета (*addTicket*) – функција се састоји из три дела: постављања новог идентификатора тикета, уноса тикета и ажурирања докумената за децу тикете.

Стање целобројних идентификатора који се користе налази се у колекцији *counters.* Постављање новог идентификатора врши се претрагом и инкрементацијом вредности из ове колекције. Ово је могуће урадити једним упитом коришћењем *findAndModify* функције *MongoDB*-а. У колекцији се тражи документ са вредношћу идентификатора „*ticketId*“ и инкрементира се поље секвенца (*seq*) за 1 коришћењем оператора *$inc.* На крају, поставља се опција *new* која омогућава да се врати вредност управо ажурираног документа назад у објекту *object.*

```
db.collection('counters')
    .findAndModify(
            { _id: "ticketId" }, {},
            { $inc: { seq: 1 }}, { new: true },

            function(err, object) { /*...*/ });
```

Након тога се формира објекат *ticket* тако што преузима поља прослеђеног објекта *req.body.Ticket,* негов идентификатор се поставља на управо креирану вредност (*object.seq*) и постављају се подразумеване вредности поља тикета:

```
ticket._id = object.seq;

var number = "000000"+ticket._id;

ticket.number = number.substr(number.length-6);
ticket.ref_number = null;
ticket.parent_id = null;
ticket.materials = [];
ticket.signatures = null;
ticket.assigned_to = null;
ticket.level = null;
ticket.closed = false;
```

```
if(childIds.length > 0) ticket.has_children = true;
```

Овако формирани тикет се уноси у базу података позивом функције *save.*

```
db.collection('ticket').save(ticket, /*...*/);
```

На крају је потребно извршити ажурирање тикета који су придружени овом тикету следећим позивом:

```
db.collection('ticket')
    .update(
        { _id: { $in: childIds }, parent_id: null },
        { $set: {parent_id: ticket._id }},
        { multi: true},

       function(err, object) { /*...*/ });
```

Pridruženi tiketi:

	Broj	Tip	Info	Status	Preuzeo	Datum
obriši	#010571	Zamena miša	**Korisnik:** Pera **Odeljenje:** Operativa **Racunar:** Računar01	Dodeljen		01.09.2015
obriši	#010575	Zamena monitora	**Korisnik:** Pera **Odeljenje:** Operativa **Racunar:** Računar01	Otvoren		12.09.2015

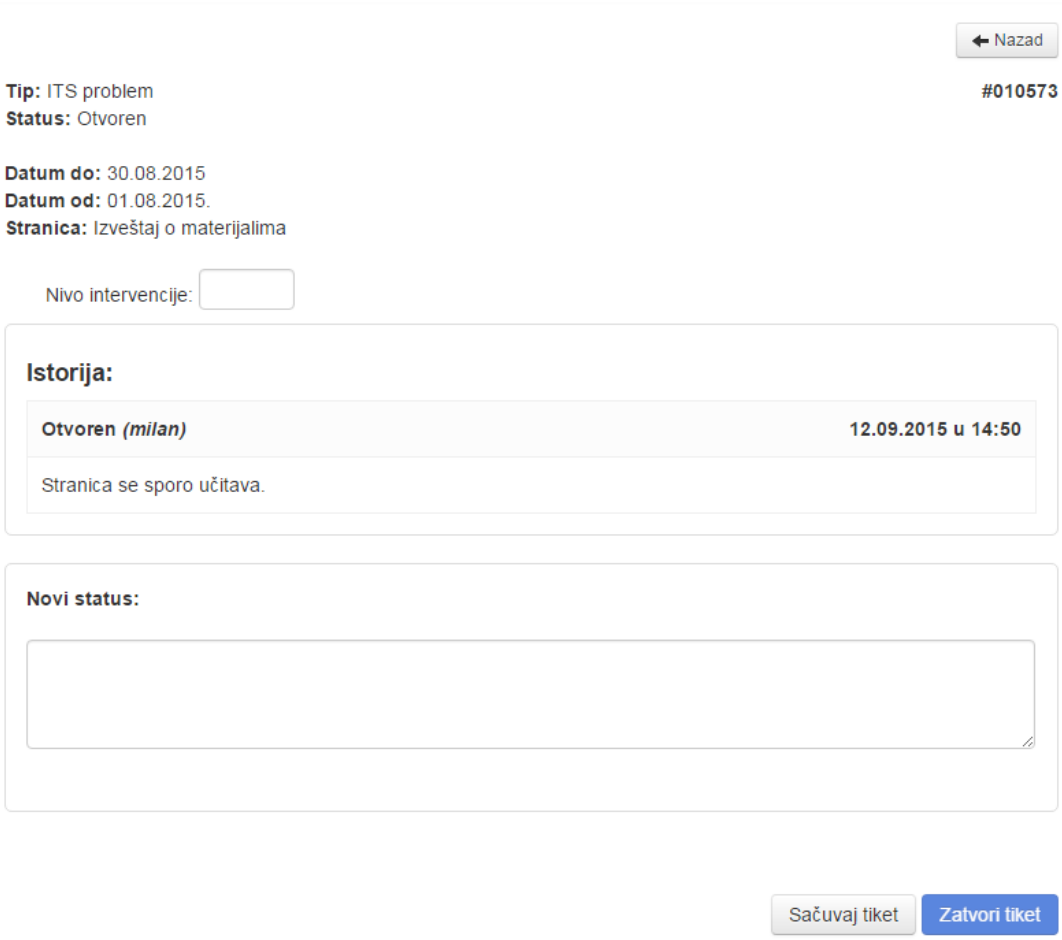
Слика 11: Унос придружених тикета

ticket/:id – дефинише операције читања и ажурирања појединачних тикета

```
router.route('/:id')
    .all(authController.isValidSession)
    .get(ticketController.getTicketById)
    .put(ticketController.saveTicket);
```

GET – метода дефинише операцију претраге колекције *ticket* по идентификатору тикета који је прослеђен кроз параметар *URI*-ја (*req.params.id*).

PUT – метода дефинише опрерацију ажурирања тикета тако што чита тикет из тела захтева и уноси га у базу коришћењем *save* функције, на исти начин на који се то ради приликом уноса тикета. Објекат је у потпуности дефинисан на клијентској страни.



Слика 12: Страница за ажурирање тикета

ticket/:id/child – дефинише операције читања и ажурирања статуса које се врше над децом задатог тикета.

```
router.route('/:id/child')
    .all(authController.isValidSession)
    .get(ticketController.getChildrenById)
    .put(ticketController.updateChildrenStatus);
```

GET – метода врши претрагу тикета по идентификатору тикета родитеља који је задат кроз параметар *URI*-ја

PUT – метода дефинише опрерацију ажурирања статуса деце тикета.

Улазни параметри:

- *req.params.id* – идентификатор тикета родитеља
- *req.body.Status* – објекат који представља статус који треба да буде додат статусима деце тикета

Ажурирање статуса (*updateChildrenStatus*) – Функција врши ажурирање параметара *status_id*, *status* и *change_date* и додаје објекат *status* на крај низа *statuses* у колекцији *ticket.* Ово се у *MongoDB*-у може урадити у једном упиту помоћу оператора *$addToSet*.

```
db.collection('ticket')
    .update(
        { parent_id: ticketId },
        { $set: {
            status_id: status.status_id,
            status: status.status,
            change_date: status.date },
          $addToSet: { statuses : status }},
        { multi: true } , function(err, item){ /* ... */});
```

Оператор *$addToSet* је веома користан, јер би у противном морала бити написана два упита за ажурирање ових параметара. Штавише, слични упити у релационим системима се, по дефиницији, извршавају над више табела што подразумева коришћење трансакција. *MongoDB* ово решава на крајње елегантан начин не ризикујући притом брзину уноса.

ticket/:id/child/:childId – дефинише операцију уклањања родитеља са наведеног тикета.

```
router.route('/:id/child/:childId')
    .all(authController.isValidSession)
    .delete(authController.isUserSupervisor,
            ticketController.removeTicketParent);
```

DELETE – стриктно говорећи ова метода не дефинише операцију брисања већ ажурирања, али оваква дефиниција има смисла када се узме у обзир контекст детета тикета који више не постоји након извршења ове операције. Ову операцију могу вршити само супервизори.

Улазни параметри:

- *req.params.id* – идентификатор тикета родитеља
- *req.params.childId* – идентификатор тикета детета
- *req.query.UserId* – идентификатор корисника који врши операцију

Провера привилегија (*isUserSupervisor*) – врши се позивом функције функције контролера аутентификације која је описана у претходним одељцима.

Уклањање родитеља тикета (*removeTicketParent*) – врши се претрага тикета по идентификатору који има задатог родитеља и за нађене документе врши се поставља се вредност поља *parent_id* на *null.*

```
db.collection('ticket')
      .update({_id: ticketId, parent_id: parentId},
            {$set: { parent_id: null } },
      function(err, object) { /* ... */ });
```

ticket/user – дефинише операцију групног додељивања тикета кориснику.

```
router.route('/user')
    .all(authController.isValidSession)
    .post(ticketController.assignUser);
```

POST – метод такође, попут *DELETE* метода код операције уклањања родитеља тикета, не репрезентује оно што се дешава у бази података[17], али, поново, има смисла ако се посматра као ресурс „корисник тикета“.

Улазни параметри:

- *req.query.UserId* – идентификатор корисника
- *req.body.User* – објекат који садржи корисника коме се додељују тикети
- *req.body.TicketIds* – низ идентификатора тикета који се додељују кориснику

Додела кориснику (*assignUser*) – Супервизорима је омогућено додељивање корисника тикетима који су већ додељени неком кориснику, те је, пре ажурирања, потребно извршити проверу да ли корисник има довољно привилегија и на основу тога дефинисати упитни објекат.

Провера привилегија се врши позивом интернет функције *userHasAccess* контролера аутентификације која је већ описана. Упитни објекат се дефинише на следећи начин.

```
var query =  {_id : {$in: ticketIds}};

if(!hasPermission)
    query.assigned_to = null;
```

Дакле, уколико корисник нема довољно привилегија, поставља се у упитном објекту ограничење да тикет мора бити недодељен. Пошто додела корисника, подразумева и промену статуса тикета у статус „додељен“, дефинише се објекат *status* који ће бити додат низу *statuses* датог тикета. Овај објекат има другачију структуру од осталих статуса који се додају (нема поља *comment,* а има поље *assigned_to* које се уноси само код овог статуса).

```
var status = {
    status_id : 4,
    status : "Dodeljen",
    assigned_to: assigned_to,
    user : user.name,
    date : new Date()
}
```

Клијентској апликацији се оставља да обрађује и правилно приказује овакве случајеве. Након овога, могуће је извршити ажурирање колекције наредним упитом:

```
db.collection('ticket')
    .update(query,
        {$set: {
            status_id: status.status_id,
            status: status.status,
```

[17] *POST* операције дефинишу унос ресурса, о чему је већ било речи.

```
            assigned_to: assigned_to,
            change_date: new Date()},
        $addToSet: {statuses: status}}, {multi: true},
    function(err, object) {/*...*/});
```

Може се приметити да је овај упит сличан упиту за промену статуса тикета (са изузетком додељивања корисника) па нема потребе за детаљнијим објашњењима.

ticket/user/:id – ресурс служи да преброји тикете додељене задатом кориснику и врати информацију клијенту.

```
router.route('/user/:id')
    .all(authController.isValidSession)
    .post(ticketController.getMyTicketCount);
```

POST – метода дефинише неспецифичну функционалност бројања тикета корисника.

Улазни параметри:

- *req.params.id* – идентификатор корисника прослеђен као параметар *URI*-ја

Читање броја додељених тикета (*getMyTicketCount*) – Функција врши претрагу тикета који су додељени кориснику, имају статус који је различит од „затворен“ (од значаја овде су само актуелни тикети) и који није дете ниједног тикета. И врши сабирање броја докумената који задовољавају ове критеријуме користећи *MongoDB* функцију *count*.

```
db.collection('ticket', function(err, collection) {
    collection.count({
        status_id: { $ne: 3 },
        "assigned_to._id": id,
        parent_id: null }, function(err, item) {/*...*/});
```

7.5.6 Статуси и типови тикета

Ресурси који се односе на статусе и типове тикета имају дефинисане само операције читања. Штавише, ове операције не прихватају улазне параметре стога им неће бити посвећена велика пажња. Статуси тикета представљени су ресурсом *ticket-status* за који је дефинисан следећи рутер:

```
router.route('/')
    .all(authController.isValidSession)
    .get(ticketStatusController.getTicketStatuses);
```

GET – метод служи за читање статуса из колекције *ticket_status*.

Типови тикета су представљени ресурсом *ticket-type* који врши читање колекције *ticket_type* на потпуно аналоган начин као и за статусе.

```
router.route('/')
    .all(authController.isValidSession)
    .get(ticketTypeController.getTicketTypes);
```

GET – метод служи за читање типова тикета из колекције *ticket_type.*

7.5.7 Материјали

Материјали се дефинишу помоћу два ресурса. Један који се кориснити за читање свих материјала и који је мапиран за путању */material* и други који дефинише операције над појединачним документима у колекцији *materials,* који се референцира путањом */material/:id.*

material – овај ресурс има дефинисану само операцију читања ресрса. Рутер је дефинисан следећим функцијама:

```
router.route('/')
    .all(authController.isValidSession)
    .get(materialController.getMaterial)
    .post(materialController.saveMaterial);
```

GET – метода дефинише операцију читања колекције *material.* Она нема дефинисане параметре претраге.

Улазни параметри:

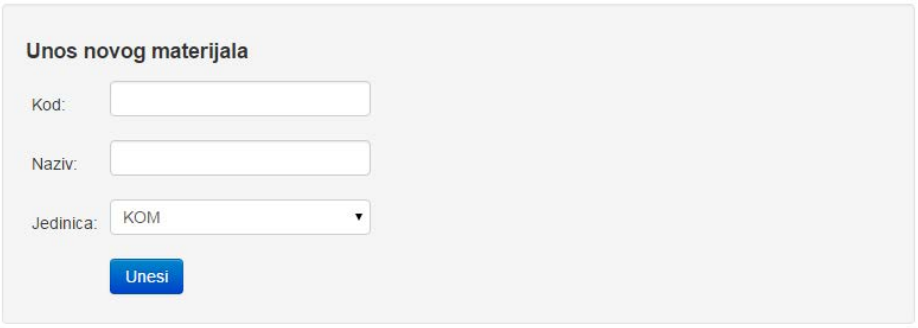
- *req.body.DepartmentId* – идентификатор одељења које располаже овим материјалом

Слика 13: Преглед материјала

POST – метод дефинише унос новог материјала у колекцију

Улазни параметри:

- *req.body.Code* – шифра материјала
- *req.body.Name* – назив материјала
- *req.body.Unit* – јединица мере
- *req.body.DepartmentId* – идентификатор одељења



Слика 14: Унос материјала

Унос материјала (*saveMaterial*) – функција формира објекат који садржи потребна поља и уноси га користећи функцију *save.* Ова функција се такође користи за ажурирање материјала[18]. Тако да се врши провера да ли је прослеђен параметар *id* у *URI*-ју и, ако јесте, ова функција ће извршити ажурирање објекта. Но, овде то не може да се деси због дефиниције руте.

```
var material_id = req.params.id,
    code = req.body.Code,
    name = req.body.Name,
    unit = req.body.Unit;

var material = { code: code, name: name, unit: unit };

if(typeof material_id !== 'undefined')
    material._id = new ObjectId(material_id);
```

Након дефиниције објекта врши се чување објекта следећим позивом:

```
db.collection('material', function(err, collection) {
    collection.save(material, function(err, item) { /*...*/ });
});
```

material/:id – ресурс дефинише операцију брисања ресурса.

```
router.route('/:id')
    .all(authController.isValidSession)
    .put(materialController.saveMaterial),
    .delete(materialController.deleteMaterial);
```

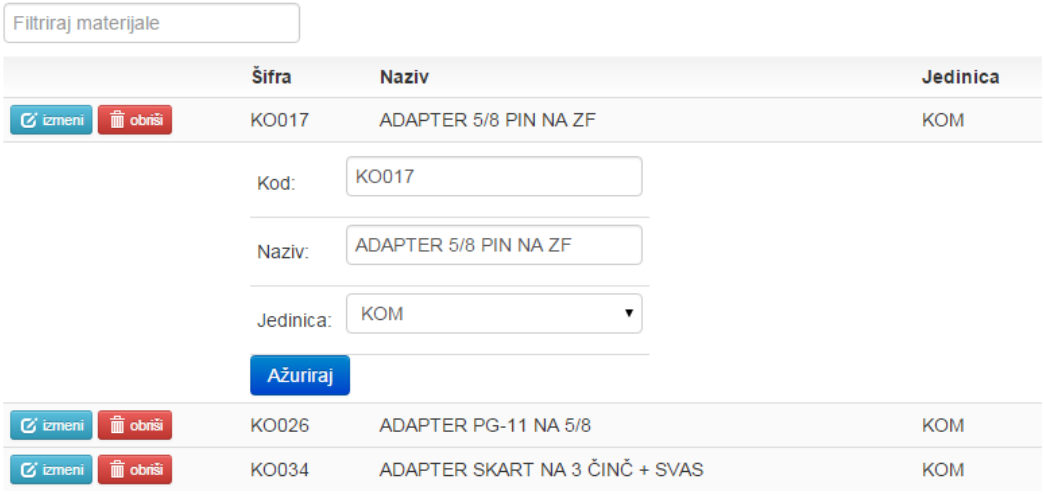
PUT – метод дефинише операцију ажурирања материјала.

Улазни параметри:

- *req.params.id* – идентификатор материјала
- *req.body.Code* – шифра материјала
- *req.body.Name* – назив материјала
- *req.body.Unit* – јединица мере

[18] Особина *save* функције је да, уколико је постављен идентификатор објекта који се снима, врши ажурирање тог објекта, у супротном се врши унос.

- *req.body.DepartmentId* – идентификатор одељења



Слика 15: Измена материјала

Унос материјала (*saveMaterial*) – функција која се користи и за унос материјала и већ је описана. Једина разлика је што је у овом ресурсу дефинише идентификатор материјала који се ажурира кроз параметар *URI*-ја.

DELETE – брише документ из колекције на основу прослеђеног идентификатора материала, *req.params.id.*

7.5.8 Извештаји

Извештај представља неку врсту ресурса омотача који пружа могућност дефиниције различитих извештаја који нису стриктно везани за једну колекцију. Ово је у складу са дефиницијом ресурса која каже да он може бити све што може бити концептуализовано. У овом раду биће представљена два извештаја који презентују две различите агрегационе функционалности *MongoDB* базе података. Над извештајима је дефинисана једино операција читања.

report/ticket/material – описује извештај утрошених материјала на завореним тикетима задатог одељења.

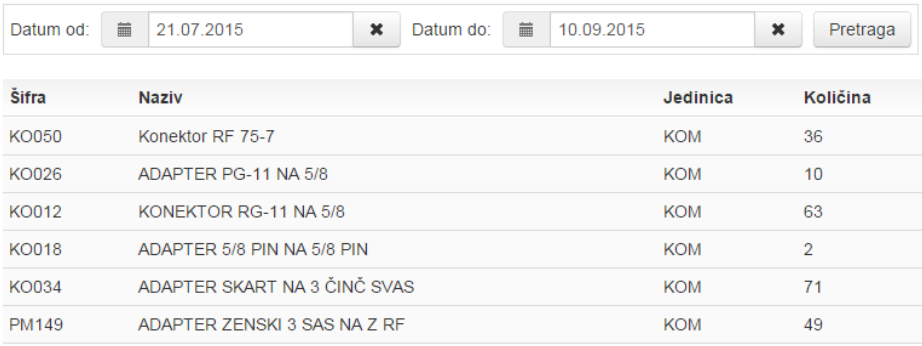
```
router.route('/ticket/material')
    .all(authController.isValidSession, authController.isUserSupervisor)
    .get(reportController.getUsedMaterialReport);
```

GET – метода врши претрагу колекције *ticket* и агрегацију по материјалима коришћењем функционалности уланчавања агрегационог оквира *MongoDB* базе података.

Улазни параметри:

- *req.query.dateFrom* – доње ограничење датума последње измене тикета
- *req.query.dateTo* – горње ограничење датума последње измене тикета

- *req.query.DepartmentId* – идентификатор одељења којем је додељен тикет. Овај параметар је опцион



Šifra	Naziv	Jedinica	Količina
KO050	Konektor RF 75-7	KOM	36
KO026	ADAPTER PG-11 NA 5/8	KOM	10
KO012	KONEKTOR RG-11 NA 5/8	KOM	63
KO018	ADAPTER 5/8 PIN NA 5/8 PIN	KOM	2
KO034	ADAPTER SKART NA 3 ČINČ SVAS	KOM	71
PM149	ADAPTER ZENSKI 3 SAS NA Z RF	KOM	49

Слика 16: Извештај о утрошеном материјалу код затворених тикета

Учитавање извештаја (*getUsedMaterialReport*) – извештај се реализује позивом функције *aggregate* са четири уланчана оператора, редом, *$match, $unwind, $project* и *$group***.**

- *$match* – оператор дефинише услове претраге тикета која је дата објектом:

```
{"status_id" : 3,
 "change_date" : {$gte: date_from, $lt: date_to },
 "department_id" : department_id }
```

Који враћа низ тикета са статусом 3 (затворени тикети) измењених у задатом временском интервалу и који припадају (уколико је параметар постављен) задатом одељењу. Овај резултат се прослеђује наредном оператору.

- *$unwind* – врши „размотавање“ по атрибуту *materials*.

```
{$unwind: "$materials" }
```

То значи да су документи разбијени на тај начин да су сада материјали на истом нивоу. Прецизније, редови који су садржали низове материјала су сада мултипликовани у резултату и имају само по један документ са материјалом.

- *$project* – Након разбијања под-докумената тако да чине јединствене записе у редовима резултата, потребно је извршити пројекцију како би се издвојили само они атрибути који су потребни у овом извештају, а то су назив, шифра и јединица мере материјала.

```
{ _id:
        { name: "$materials.name",
          code: "$materials.code",
          unit: "$materials.unit" },
"quantity": "$materials.quantity"}
```

Груписање се врши по идентификатору, који је обавезна колона, тако да се у овом извештају могу навести све три потребне колоне као идентификатор по којем ће се вршити груписање.

Поред идентификатора, потребно је дефинисати још и колону квантитет по којој ће сумирати свака група

- *$group* – на крају, груписање се врши тако што се дефинише идентификатор по којем се врши груписање (односно идентификатор дефинисан у претходном оператору) као и функција која сумира вредности квантитета материјала

```
{ _id: "$_id", sum: { $sum: "$quantity" }}
```

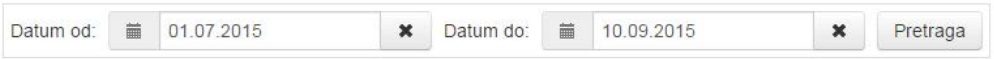
report/ticket/levels – представља извештај који сумира број тикета по нивоу интервенције за изабрани период.

```
router.route('/ticket/levels')
    .all(authController.isValidSession, authController.isUserSupervisor)
    .get(reportController.getTicketLevelsReport);
```

GET - метода врши претрагу колекције *ticket* и груписање по нивоима интервенције.

Улазни параметри:

- *req.query.dateFrom* – доње ограничење датума последње измене тикета
- *req.query.dateTo* – горње ограничење датума последње измене тикета
- *department_id* – идентификатор одељења којем је додељен тикет. Овај параметар је опцион



Nivo	Broj
1	2
2	1
4	1
5	3
Bez nivoa	25
Ukupno:	32

Слика 17: Извештај о нивоима интервенција затворених тикета

Учитавање извештаја (*getTicketLevelsReport*) – врши се претрага тикета са истим параметрима као и код извештаја о утрошеним материјалима. Груписање је дефинисано на следећи начин:

```
{ _id: { $ifNull: ["$level", "Bez nivoa" ]}, count: { $sum: 1 }}
```

Оператор *$ifNull* врши проверу да ли атрибут *level* има вредност *null* и, ако има, враћа вредност *„Bez nivoa“*. На крају се врши сортирање по идентификатору резултата (односно по нивоима) позивом:

```
{$sort: {_id: 1}}
```

8. Закључак

У раду су представљени кључни аспекти *REST* архитектуре, као и прикази две најпопуларније серверске технологије данашњице које се користе за развој ове класе сервиса. Ови прикази укључују и историјат настанка како *JavaScript* серверских технологија, тако и целог *NoSQL* покрета. У општем приказу *Node.js* и *MongoDB* технологија дискутовано је о мотивима везаним за њихов настанак, начин на који функционишу и дат је преглед њихових предности и мана.

Пре свега, треба напоменути да, и поред свих предности ових двеју технологија оне неће, барем не у потпуности, заменити нити најпопуларније веб сервере, ни релационе базе података. Оне нису ни дизајниране да то учине нити претендују на тако нешто. Уместо тога, оне су замишљене да реше проблеме за које нису предвиђени њихови претходници. Оне пружају алтернативу постојећим архитектурама у смислу лакшег развоја и постизања бољих перформанси и флексибилности, подносећи притом одређене жртве о којима је дискутовано у овом раду.

У том погледу потпуно је јасно зашто су ове две технологије постале најпопуларније када је у питању развој флексибилних и брзих клијент-сервер апликација. *Node* није дизајниран да представља конкуренцију *Apache* веб серверу, већ да пружи начин да се пишу брзе апликације са јако малим одзивом. Свакако је ограничење *Node*-а то што му недостају многе функционалности које су уграђене у *Apache,* пре свега сигурносне природе. Оне морају бити посебно имплементиране у самим апликацијама које се развијају, али то је свесно донета одлука у циљу очувања једноставности и флексибилности платформе. Са друге стране, *MongoDB* никада не треба бити коришћен за имплементацију система са критичим операцијама за које треба гарантовати атомичност трансакција и нивое изолације. Примера ради, ниједна банка неће никада базирати своје системе на *NoSQL* базама података. У том погледу ни *MongoDB,* а ни остале *NoSQL* базе података, не представљају конкуренцију релационим системима.

Дакле, ове технологије имају своју сферу примене и већина њихових ограничења потиче из свесно донетих компромиса направљених са специфичним циљем у виду. Са друге стране, њихова популарност је велика јер су међу првима, вероватно и на бољи начин од конкуренције, одговорили на изазове перформанси и скалирања, жртвујући релативно мали део особина које имају њихови претходници. Оне нису инстант решења свих могућих проблема и свакако да нису погодне за коришћење у свим могућим сценаријима, али у свом пољу могу одговорити сваком изазову који се пред њих наметне.

На крају, оно што је евидентно је да ни *Node,* ни *MongoDB,* иако већ сада представљају зреле технологије, нису достигле свој врхунац. То се може видети по променама које се уводе приликом изласка нових верзија и, поготово, по начину на који заједница отвореног кода прати и узима активно учешће у њиховом развоју. Субјективно мишљење аутора овог рада је да је будућност светла за ове технологије и од њих се може очекивати много.

Литература

[1] H. Haas и A. Brown, „Web Services Glossary“, *World Wide Web Consortium*, 2004. [На Интернету]. Available at: http://www.w3.org/TR/ws-gloss/.

[2] F. Doglio, *Pro REST API Development with Node.js*. Apress, 2015.

[3] D. Box, „A brief history of SOAP“, 2001. [На Интернету]. Available at: http://www.xml.com/pub/a/ws/2001/04/04/soap.html.

[4] D. Benslimane, S. Dustdar, и A. Sheth, „Services Mashups: The New Generation of Web Applications“, *IEEE Internet Comput.*, том 12, изд. 5, 2008.

[5] K. Mockford, „Web Services architecture“, *BT Technology Journal*, 2004. [На Интернету]. Available at: http://www.w3.org/TR/ws-arch/.

[6] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures“, University of California, Irvine, 2000.

[7] Wikipedia, „Web service.“ [На Интернету]. Available at: https://en.wikipedia.org/wiki/Web_service.

[8] L. Richardson и S. Ruby, *RESTful Web Services*. O'Reilly, 2007.

[9] B. Lavoie и H. F. Nielsen, „Web Characterization Terminology & Definitions Sheet“, 1999. [На Интернету]. Available at: http://www.w3.org/1999/05/WCA-terms/.

[10] „Hypertext Transfer Protocol -- HTTP/1.1“, 1999. [На Интернету]. Available at: https://tools.ietf.org/html/rfc2616.

[11] S. Allamaraju, *RESTful Web Services Cookbook*. O'Reilly, 2010.

[12] C. Ihrig, *Pro Node.js for Developers*. Apress, 2013.

[13] Ecma Int., „Standard ECMA-262“, 2011. [На Интернету]. Available at: http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[14] „Chrome V8 | Google Developers.“ [На Интернету]. Available at: https://developers.google.com/v8/.

[15] M. Cantelon, M. Harter, N. Rajlich, F. O. By, и I. Z. Schlueter, *Node.js in Action*. Manning Publications Co., 2014.

[16] A. Q. Haviv, *MEAN Web Development*. Packt Publishing, 2014.

[17] A. Vlăduțu, *Mastering Web Application Development with Express*. Packt Publishing, 2014.

[18] T. Haerder и A. Reuter, „Principles of transaction-oriented database recovery“, 1983.

[19] G. Vaish, *Getting Started with NoSQL*. Packt Publishing, 2013.

[20] E. Brewer, „CAP twelve years later: How the ‚rules‘ have changed“, *Computer (Long. Beach. Calif).*, том 45, изд. 2, стр. 23–29, 2012.

[21] K. Chodorow и M. Dirolf, *MongoDB: The Definitive Guide*. O’Reilly, 2010.

[22] MongoDB Inc., „BSON - Binary JSON“, 2013. [На Интернету]. Available at: http://bsonspec.org/.

[23] David, P. Membrey, и E. P. Howes, *MongoDB Basics*. Apress, 2014.

[24] G. E. Krasner и S. T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*, том 1, изд. 3. 1988.

Додатак: Независни модули коришћени у раду

Назив модула	Веб страна	Опис модула
bcrypt-nodejs	https://www.npmjs.com/package/bcrypt-nodejs	Модул *bcrypt-nodejs* представља имплементацију *bcrypt* алгоритма за хеширање. У раду се користи за хеширање и проверу лозинки
body-parser	https://github.com/expressjs/body-parser	Модул *body-parser* је посреднички програм који се служу за парсирање тела *HTTP* захтева. Био је интегрисан у *Express* програмски оквир до изласка верзије 4.
crypto	https://github.com/Gozala/crypto	Садржи имплементацију стандардних алгоритама за хеширање *MD5* и *SHA1.* MD5 се у раду користи за креирање кључева за сесије
express	http://expressjs.com	Модул програмског оквира *Express.*
mongodb	http://docs.mongodb.org/ecosystem/drivers/node-js	Драјвер за приступ *MongoDB* бази података писан за *Node.*
validator	https://github.com/chriso/validator.js	Модул за валидацију и санирање података на серверској страни
winston	https://github.com/winstonjs/winston	Флексибилни модул за креирање дневника