UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Miha Podplatnik

PRIMERJAVA OGRODIJ ZA ZALEDNE SISTEME MOBILNIH APLIKACIJ

Zaključno delo

Maribor, avgust 2019

UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Miha Podplatnik

PRIMERJAVA OGRODIJ ZA ZALEDNE SISTEME MOBILNIH APLIKACIJ

Zaključno delo

Maribor, avgust 2019

PRIMERJAVA OGRODIJ ZA ZALEDNE SISTEME MOBILNIH APLIKACIJ

Diplomsko delo

Študent:	Miha Podplatnik
Študijski program:	Informatika in tehnologije komuniciranja UNI
Smer:	Informacijski sistemi
Mentor (ica):	doc. dr. Luka Pavlič, univ. dipl. inž. rač. in inf.
Lektor (ica):	Nina Rajh, mag. prof. slo. jezika in knjiž.

Zahvala

Iskreno se zahvaljujem mentorju, doc. dr. Luki Pavliču, za vse strokovne nasvete, ideje in pomoč pri izdelavi diplomskega dela. Posebno zahvalo namenjam svoji družini in prijateljem, ki so me med študijem podpirali.

Primerjava ogrodij za zaledne sisteme mobilnih aplikacij

Ključne besede: zaledni sistemi, serverless, firebase, express.js, django rest framework.

UDK: 621.397.7-026.26(043.2)

Povzetek

Področje implementacije zalednih sistemov za mobilne rešitve lahko že pred začetkom projekta predstavlja težavo, sploh zaradi tega, ker je na voljo ogromno poti, po katerih se lahko sprehodimo, vse pa nas ne vodijo na cilj, vsaj ne racionalen. V diplomski nalogi so predstavljeni vsi sestavni deli zalednega sistema, izbrane arhitekture in dobre prakse pri zavoju. Osrednji področji diplomske naloge sta izbira in analiza ogrodij iz celotnega spektra, od zelo fleksibilnih do tistih, ki nam vsiljujejo svoj način razvoja. Pozornost smo namenili tako klasičnemu skladu kot tudi modernim platformam, ki ogromno dela opravijo že iz škatle. Za potrebe lažje podaje primerjave smo razvili rešitev, ki rešuje problem iz realnega sveta.

Comparison of frameworks for mobile application backend systems

Keywords: backend systems, serverless, firebase, express.js, django rest framework.

UDK: 621.397.7-026.26(043.2)

Abstract

The implementation of the back-end systems for mobile solutions can be a thorn in the flesh of many developers, even before the project kick-off. Amongst many decisions we make, one of them is also picking the right framework for the job, since not all of them will be the right one for the specification brief, at least not rationally. In this thesis, we present all the moving parts in one's backend system, we discuss various architectures and best practices when in development. The focus of the diploma is the selection and analysis of the frameworks across the spectrum, from very flexible to those strongly opinionated. We also paid attention to both the traditional stack and modern platforms that do a lot of work for us right out of the box. For ease of comparison, we have developed a solution that solves a real-world problem.

KAZALO

KAZALO SLIK

KAZALO TABEL

SEZNAM UPORABLJENIH KRATIC

BAAS	*Backend-as-a-Service*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
SMTP	*Simple Mail Transfer Protocol*
FTP	*File Transfer Protocol*
MEAN	*MongoDB, Express.js, Angular, Node.js*
MVC	*Model–View–Controller*
JSON	*JavaScript Object Notation*
API	*Application programming interface*
SSL	*Secure Sockets Layer*
REST	*Representational state transfer*
JWT	*JSON Web Token*
XML	*eXtensible Markup Language*
DRF	*Django Rest Framework*
CRUD	*Create, read, update and delete*
FCM	*Firebase Cloud Messaging*
HTML	*Hyper Text Markup Language*
CSS	*Cascading Style Sheets*
JS	*JavaScript*
CSRF	*Cross-Site Request Forgery*
DOM	*Document Object Model*
ORM	*Object-relational mapping*
APNS	*Apple Push Notification Service*

1 UVOD

Vedno večji delež mobilnih telefonov na trgu pomeni vedno večje preseljevanje razvoja iz klasičnega fokusa spletnih aplikacij na mobilne platforme. Na dolgi rok imajo uporabniki radi interaktivnost in povezovanje s svojimi prijatelji in tistimi, ki bodo to šele postali. Vse ostale rešitve so samo kratkotrajne. Da bi podpirali take mobilne aplikacije, moramo razviti komplementarni zaledni sistem, ki bo deloval kot poenotena shramba podatkov za vse uporabnike. Ponuditi jim je treba aktualno vsebino in interakcijo z njo. Razvijalcem je na voljo ogromno programskih jezikov in vsaki od teh ima -n ogrodij za razvoj zalednih sistemov. Katerega izbrati za katero rešitev, kjer bomo najboljše uskladili vse vire in dostavili kvaliteten izdelek, pa je lahko večja težava, kot se nam sprva zdi.

1.1 Opredelitev področja in opis problema

V diplomski nalogi bomo raziskovali področje ogrodij za implementacijo zalednih sistemov, ki služijo kot vmesnik poslovne logike med mobilnimi rešitvami in plastjo poslovnih operacij. Vedno bolj se uveljavljajo tudi zaledni sistemi v oblaku, ki temeljijo na »serverless« arhitekturi. Pri izbiri moramo med drugim v obzir vzeti dejavnike, kot so: tip arhitekture, podpora podatkovnim shrambam, podpora arhitekturnim stilom, kot je REST, aktivna skupnost, izobraževalna krivulja, uporabna in pregleda dokumentacija ter ostale lastnosti ogrodij. Izbor tehnologije glede na rešitev pa je samo prvi del. Sledi vzpostavitev razvojnega okolja skozi izbrano zabojno storitev, upravljanje s knjižnicami preko managerja paketov, sama implementacija in testiranje ter končna orkestracija aplikacije na živem strežniku. Vsi ti koraki se lahko kaj hitro prepletejo in otežijo implementacijo še tako trivialne rešitve, zato bomo v nalogi tudi predstavili razvoj realnega projekta z izbranim ogrodjem.

1.2 Cilji diplomskega dela

Cilj diplomske naloge je analizirati predstavnike ogrodij različnih arhitektur za razvoj zalednih sistemov mobilnih aplikacij in analizirati njihove prednosti ter pomanjkljivosti. Povzeti želimo protokole, arhitekturne stile in dokumentacijska orodja, ki so pogosto del razvojnega cikla. V namen primerjave in demonstracije bomo implementirali projekt v

izbranem ogrodju. Predstaviti želimo tudi dobre prakse, ki jim naj sledimo pri razvoju in uporabi ogrodij.

1.3 Predpostavke in omejitve diplomskega dela

V diplomski nalogi bomo predpostavili, da sprednji del celotne rešitve predstavljajo iOS in Android naprave. Ciljamo na mobilne platforme iOS in Android, ki so povezane na zaledni sistem preko vmesnika REST.

Pri izbiri ogrodij se bomo omejili na predstavnika »serverless« arhitekture in tradicionalnega sklada. Primerjali bomo tri izbrana ogrodja in na enem implementirali projekt iz resničnega življenja.

2 ZALEDNI SISTEMI MOBILNIH APLIKACIJ

Pri razvoju mobilnih aplikacij je pomembno, da si zastavimo nekaj ključnih vprašanj. Ali ima naša aplikacija potrebe po shranjevanju uporabniških podatkov? Ali mora naša aplikacija prikazovati dinamične podatke glede na uporabniške akcije? Moramo voditi evidenco uporabnikov preko avtentikacijskega modula? Če je odgovor na katerokoli tako ali podobno zastavljeno vprašanje pritrdilen, potem boste najverjetneje potrebovali zaledni sistem, ki bo izpostavil več različnih končnih točk skozi izbrani vmesnik.

Zaledni sistem, imenovan tudi strežniška stran aplikacije, je tipično sestavljen iz strežnika, ki ob zahtevi servira informacije; aplikacije, ki te informacije obdeluje in posreduje; ter podatkovne shrambe, ki trajno shranjuje in organizira informacije za kasnejši dostop. Če dodamo zraven še možnosti razširljivosti in rasti, pred strežnikom tipično sedi še usmerjevalnik oziroma urejevalnik obremenitve, znotraj aplikacijskega strežnika pa imamo pogosto integrirano vrsto drugih storitev, kot so: sporočanje preko elektronske pošte, potisna sporočila ter čakalne vrste. V laičnem pomenu razvoj zalednih sistemov pomeni pripravo aplikacije sprednjega dela tako, da deluje, tako kot je bilo zamišljeno. Da pa je to mogoče, moramo razumeti nekaj konceptov, ki skupaj tvorijo zaledni sistem oziroma sklad programske opreme. Za vsako aplikacijo bi bilo nespametno implementirati vse možne dele. Bistvo te diferenciacije, da za razvoj zalednih storitev ne obstaja univerzalni recept za izbor sklada, lahko pripišemo uporabi podatkovnih skladišč, oblačnih in tretjih storitev ter vmesnikov, pakiranje aplikacije v zaboje (angl. »Containerization«) ter izboru zalednih sistemov kot storitev (angl. »BaaS – Backend-as-a-Service«), ki imajo že implementirana kompleksna poslovna pravila. Pametni izbor storitev nam pogosto prihrani veliko časa in sredstev za razvoj. [1] [2] [3]

2.1 Struktura zalednih sistemov

Da bi podrobneje razumeli celoten osnovni sklad zalednega sistema, ki je skoraj nujen za obstoj kvalitetne rešitve, si bomo problem razdelili na manjše dele, kot je to praksa tudi pri dejanskem razvoju aplikacije. V grobem, zaledni sistem sestavljajo naslednje štiri večje komponente: [2] [3]

- spletni strežnik – prav vseeno je, ali se nahaja v nam znani zgradbi na fizični lokaciji, ali pa uporabljamo takega v oblaku, strežnik deluje, kot ožilje celotnega sistema in povezuje ostale komponente v omrežju. [2] [3]
- Aplikacija oziroma strežniška aplikacija – angleško jo imenujemo tudi »middleware«, opisuje pa vsakršno programsko opremo na strežniku, ki povezuje sprednji del aplikacije z zalednimi komponentami. Lahko jo organiziramo skozi poljubne sloje, tako poslovne kot tudi predstavitvene. [2] [3]
- Podatkovna baza – je tista stvar, ki naredi aplikacijo dinamično in trajno. Praviloma ob vsaki zahtevi na zaledno aplikacijo, ali iščete prosti Airbnb za naslednja potovanja, ali pa si samo pasete oči po trendovskih oblačilih na priljubljeni spletni trgovini, podatkovna baza prejme niz ukazov, s katerimi izlušči pravilne podatke in jih skozi aplikacijo vrne uporabniku. [2] [3]
- Programski jezik in knjižnice – razvijalci lahko izbirajo med mnogimi jeziki in ogrodji glede na specifikacije, poznavanje življenjskega cikla ter ostalih komponent, ki že živijo na zaledju. [2] [3]

2.1.1 Spletni strežniki

Dandanes se le stežka izognemo tematiki strežnika, ne da bi govorili ali vsaj omenili oblak. Vse bolj dostopnejša so strežniška gostovanja v oblaku, ki izpodrivajo klasična skladišča fizičnih super računalnikov, za katere je vsako podjetje skrbelo v hiši. Strežniki poskrbijo za deljene vire, med drugim tudi za shranjevanje statičnih datotek, šifriranje in rokovanje med strežnikom in klientom, predal elektronske pošte ter izpostavijo končne točke aplikacijskega strežnika. Strežniško arhitekturo drobimo na manjše dele, plasti, ki so lahko sestavljeni iz virtualnih in dodeljenih strežnikov, aplikacijskih strežnikov ter oblačnih storitev. Za opravljanje teh storitev strežniki uporabljajo standardne protokole, kot so: HTTP, HTTPS, SMTP, FTP. Vsak od teh služi svojemu namenu na preddefinirani številki vrat, pred katero stoji IP naslov strežnika – unikaten niz številk, ločen s piko, ki je dodeljen vsaki napravi, ki je povezana v omrežje in je pogosto prevedena v alfa numerični niz preko DNS strežnika. Že prej smo omenili, da te

funkcionalnosti upravljajo super računalniki, ki imajo zaradi svoje obremenjenosti bolj zmogljive procesorje ter večje število pomnilnika. [4] [5]

Ko odjemalec potrebuje določeno datoteko, ki gostuje na strežniku, pošlje zahtevo za datoteko preko protokola HTTP (angl. »Hypertext Transfer Protocol«). Ko zahteva doseže izbrani strežnik, le-ta sprejme zahtevo, poišče ciljni dokument (če ta ne obstaja, vrne status 404, ki je standardiziran odgovor in pomeni, da tega dokumenta ni na strežniku) in ga pošlje nazaj klientu. [6]

Komunikacija običajno poteka preko že prej omenjenega protokola HTTP in HTTPS, ki je varna in šifrirana različica prejšnjega, laično povedano. Protokol je niz pravil za komunikacijo med dvema subjektoma. HTTP protokol opisujeta dve glavni značilnosti: [6]

- tekstovnost – vse zahteve in odgovori so tekstovni in človeku razumljivi. [6]
- Brez stanja – to pomeni, da si noben subjekt v komunikacijskem kanalu ne zapomni prejšnjih komunikacij, zahtev in odgovorov. Da identificiramo uporabnika, s katerim smo si že izmenjali datoteke in informacije, uporabljamo tehnologije na aplikacijskem strežniku. [6]

Med najbolj vidne predstavnike spletni strežnikov uvrščamo dva glavna konkurenta: Nginx in Apache. Slednji je na prelomu prejšnjega desetletja ohranjal 60 % delež na svetovnem spletu. Od takrat podpora Apache strežniku pada in narašča njenemu največjemu tekmecu – Nginx. Popularnost Apache strežnika lahko pripišemo predvsem najbolj razširjenemu skladu za razvoj aplikacij LAMP, ki združuje Linux operacijski sistem, Apache strežnik, MySQL oziroma MSQL podatkovne baze ter php programski jezik. Nginx je nastal na pobudo izziva C10K, ki opredeljuje uspešno serviranje vsaj 1.000 uporabnikom naenkrat. [7] Uporablja asinhrono arhitekturo, ki temelji na dogodkih. Zaradi tega ga povezujejo v sklade s podobno dogodek-orientirano arhitekturo, najpogosteje v tako imenovani MEAN sklad, ki združuje MongoDB podatkovno bazo, Express aplikacijski strežnik, Angular ogrodje in Nginx spletni strežnik. [8]

2.1.2 Middleware

Zaledna infrastruktura je pogosto sestavljena iz ogromno virov in storitev, ki jih je treba varno in zanesljivo povezati v celoto. Za uspešno orkestracijo pogosto uporabljamo strežniško aplikacijo oziroma kakršnokoli programsko opremo (angl. »middleware«). Ta nam omogoča, da definiramo skupek avtomatiziranih pravil za upravljanje, ki sledijo implementirani logiki in odgovorijo na zahtevo. Na primer, »middleware« lahko prepozna, da je v glavah zahteve (angl. »header«) specificiran angleški jezik, zato bo odgovor vseboval samo tekst v angleškem jeziku. Pomaga nam tudi pri razreševanju istočasnih procesov ter razporejanju virov. Aplikacije imajo pogosto možnost horizontalne in vertikalne skalabilnosti, s tem uspešno distribuirajo zahteve več klientov na več serverjev. Kot zadnjo veliko prednost »middleware«-a, lahko izpostavimo varovanje zalednih virov s procesom overjanja in avtorizacije zahtev. Ob posredovanju pravilnih poverilnic se zahteva izvede, v nasprotnem primeru odgovorimo s statusom 401. [9] [10] Z implementacijo dobrega »middleware«-a prihranimo čas pri reševanju trivialnih komunikacijskih težav med storitvami in se lahko bolj posvetimo naši unikati rešitvi. Popularnost tega vmesnega dela je začela naraščati v osemdesetih letih prejšnjega stoletja kot rešitev problema pri povezavi novih, sodobnih aplikacij ter vgrajenih sistemov. [11]

2.1.3 Podatkovna baza

Podatkovna baza, kot srce aplikacije, ima zraven vračanja obstoječih podatkov še možnost sprejemanja novih, spreminjanja starih in brisanje obstoječih podatkov. Te štiri funkcionalnosti pogosto opišemo s kratico CRUD (angl. »create, read, update, delete«). Zraven teh osnovnih operacij lahko podatkovne baze opravljajo praktično katerokoli nalogo, ki jo nastavimo v kontekstu obdelave podatkov – spremljajo, organizirajo in pripravljajo poročila o vsebini. [12]

Podatkovne baze lahko kategoriziramo na mnogo načinov, verjetno najbolj popularna pa je naslednja delitev:

- relacijske podatkovne baze – definirane leta 1970 s strani Edgarja F. Codda, britanskega računalnikarja. Take shrambe shranjujejo svoje podatke v tabele z

vrsticami (hranijo enoten vnos) in stolpci (hranijo specifični podatek), kot lahko vidimo na sliki 2.1. Logika temelji na skupku algebraičnih teorij, poznanih kot relacijska algebra. Prednosti takih tipov baz so v omogočanju kompleksnih poizvedb in atomičnih transakcij (spremembo na več tabelah in vrsticah hkrati, samo ob uspešnosti vseh operacij). Problemi nastanejo pri shranjevanju bitnih zapisov (slike, multimedijske datoteke). Uporabljajo standardizirani poizvedovalni jezik za dostop in manipulacijo z bazo, imenovano SQL (angl. »structured query language«). Najpogostejši predstavniki relacijskih podatkovnih baz so: MySQL, PostgreSQL, SQLite, MSSQL in Oracle. [12] [13]

- Nerelacijske podatkovne baze (NoSQL, angl. »Not only SQL«) – popularizirane v zadnjih dveh desetletjih, predstavljajo rešitev primerov uporabe, ko so podatki nepopolni, nekonsistentni, za povrh pa jih je še ogromno. Takrat potrebujemo bolj fleksibilno rešitev. NoSQL podatkovne baze ne uporabljajo tabel, ampak se uporabljajo za shranjevanje dokumentov, praviloma temeljijo na shemah. Kot take so zelo priljubljene pri »startup«-ih, ki se hitro širijo, saj so nerelacijske podatkovne baze zelo skalabilne in fleksibilne. Delimo jih v naslednje štiri podkategorije [14] [15]:
    - »ključ-vrednostne« podatkovne baze. So najenostavnejše od teh štirih in delujejo točno tako, kot se imenujejo. Vsak podatek je shranjen kot par unikatnega ključa in vrednosti. Uporabljajo se za predpomnjenje (npr. shranjevanje artiklov pri spletnem nakupovanju), ker so zelo hitre. Najbolj pogosta predstavnika sta Redis in Riak. [16]
    - »Grafovne« podatkovne baze. Gre za zaporedje vozlišč in povezav, ki skupaj predstavljajo graf. Primerne so ob velikem številu povezav med objekti, kjer so povezave bolj pomembne od samih podatkov. Poizvedbe, namenjene iskanju odnosov med vozlišči, ki so osnovni gradnik te baze. Najbolj pogosti predstavniki so: Infinitegraph, Neo4j in OrientDB. [15] [16]
    - »Široko stolpčne« podatkovne baze. So razširljive za veliko količino podatkov in namenjene vzporedni obdelavi velike količine le-teh. Pri

relacijskih podatkovnih bazah je velikokrat potrebno prebrati celotno vrstico, vrniti pa samo določen stolpec. Široko stolpčne podatkovne baze rešujejo to zadevo s particioniranjem tabel. V grobem razrežejo tabelo z -n stolpci v -n tabel z enim stolpcem in s tem občutno zmanjšajo čas, potreben za branje podatkovnega tipa. Najbolj pogosti predstavniki so: Cassandra, Apache HBase ter Google BigTable. [16]

- »Dokumentne« (angl. »document«) podatkovne baze. So najbolj pogosti tip pri razvijalcih, in sicer zaradi prilagodljivosti in zmogljivosti. Shranjujejo podatke v dokumentih v obliki »ključ-vrednost«, kot je razvidno na sliki 2.1. Ti dokumenti pa so razvrščeni v kolekcije. Celoten dokument predstavlja en zapis, uporabljajo JSON in XML obliko zapisa. Prednost je ta, da lahko dokumenti vsebujejo ugnezdene druge dokumente. Omogočajo samodejno deljenje in replikacijo. Primerne so za hitro prototipiranje z nestrukturiranimi podatki. Najbolj pogosta predstavnika sta MongoDB in CouchDB. [16]

Slika 2.1: Razlika med relacijsko in ne-relacijsko podatkovno bazo [17]

2.1.4 Programski jezik in ogrodja na zaledju

Glede na specifikacijo sistema, ki ga izdelujemo, lahko izberemo mnogo programskih jezikov in njim komplementarnih ogrodij, ki nam omogočajo lažji in hitrejši razvoj komponent. Programski jeziki se razlikujejo po hitrosti delovanja, sintaksi in semantiki

ter stilu programiranja. Slednje paradigme lahko razdelimo v dve večji podskupini [2] [18]:

- imperativno programiranje – paradigma, kjer preko zaporedja ukazov, algoritma, če želite, opisujemo, kako naj se stanje programa spreminja. Programska oprema, napisana v tem stilu, se pogosto prevede v binarno zaporedje ukazov, ki se preko procesorja hitro izvajajo, saj so, le njegovi, algoritmi prav tako imperativni. [19] Dalje delimo to podskupino na proceduralno programiranje, ki združuje mnogo ukazov v proceduro (popularni v 60ih in 70ih prejšnjega stoletja, med znane predstavnike sodita Fortran in Pascal); ter objektno orientirano programiranje, ki združuje ukaze glede na to, kateri del stanja programa spreminjajo (dandanes zelo razširjen stil, med katere sodita Java in C#, tudi Python in JavaScript do neke mere). [20]
- Deklarativno programiranje – paradigma, kjer preprosto definiramo vrednosti končnih rezultatov, brez da opišemo, kako to želimo izvesti – prepustimo prevajalniku oziroma tolmaču. Podskupino razdelimo še na funkcijsko programiranje, ki je stil grajenja aplikacije skozi strukturo in elemente, kjer definiramo matematične funkcije za izračun in se izogibamo spreminjanju stanja ter podatkov (popularni stil predvsem pri skriptnih jezikih, kot so: Python, Javascript ter njena izpeljanka Typescript); ter manj popularno logično in matematično programiranje. [19] [21]

2.2 Lastnosti pri razvoju za mobilne aplikacije

Večina mobilnih aplikacij in iger potrebuje zaledni sistem za izvedbo stvari, ki jih sama aplikacija ne zmore. Med te spada deljenje in procesiranje podatkov za več uporabnikov, shranjevanje datotek, interakcija ter druge. Pri razvoju zalednih sistemov za mobilne aplikacije moramo biti posebej pazljivi na dodatne zahteve: [22]

- omejimo shrambo na mobilnem telefonu,
- sinhronizacija podatkov istega uporabnika na več napravah – odvisno od tipa povezave,

- primeri uporabe ob izgubi internetne povezave,
- varne in kodirane zahteve,
- pošiljanje potisnih in »in-app« sporočil,
- vračanje samo pomembnih informacij za uporabnika. [22]

Za razliko od klasičnih zalednih sistemov, kjer strežnik v odgovor na zahtevo pogosto servira .html datoteke za prikaz na brskalniku, pri razvoju za mobilne aplikacije uporabljamo drugačno tehniko.

Še vedno ohranjamo vzorec MVC (angl. »Model View Controller«), ampak podatki se več ne servirajo kot dokumenti. Za izmenjavo vsebine uporabljamo podatkovno-izmenjevalne formate, med katerimi je najbolj pogost »Javascript Object Notation« (v nadaljevanju JSON). Format je jezikovno neodvisen. Za skoraj vsak programski jezik obstaja »parser« in »generator« JSON datotek. Zavzame lahko štiri primitivne tipe: [23]

- »Number« (slo. število) – decimalno ali celo število, kot je to definirano v jeziku Javascript,
- »String« (slo. niz) – niz znakov, ki ga iz obeh strani omejujejo dvojni narekovaji,
- »Boolean« – vrednost, ki je lahko samo pravilna (angl. »true«) ali nepravilna (angl. »false«),
- »Null« – ničelna vrednost. [23]

Zraven primitivnih tipov JSON specifikacija narekuje tudi dva strukturna tipa:

- »Array« (slo. polje) – urejeno, z vejico ločeno zaporedje vrednosti, ki so lahko drugačnih tipov, vse skupaj pa omejuje par oglatih oklepajev,
- »Object« (slo. objekt) – neurejeno, z vejico ločeno zaporedje »ključ-vrednost« parov, ki jih omejuje par zavitih oklepajev. Dvopičje (:) uporabljamo za ločevanje med ključem, ki mora biti vedno tipa niz in vrednostjo, ki je lahko poljuben tip. [23]

Strogo sprednji sistemi, kot so mobilne aplikacije, ki vso poslovno logiko prenesejo na strežnik, samo prikazujejo podatke, ki jih strežniki vračajo. S tem ohranjamo aplikacije lahke, hitre in modularne.

Pri izpostavitvi API-jev (angl. »Application programming interface«), je treba poskrbeti za varnost že pri prenosu podatkov. Načeloma imajo vsi programerji namen ustvariti dober zaledni sistem, ampak pogosto se zgodi, da zaradi kratkih rokov uporabijo bližnjice, kar lahko pripomore k varnostni nestabilnosti pri prenosu podatkov. Zagotoviti je potrebno močen SSL certifikat (angl. »Secure sockets layer«), s katerim lahko ves promet preusmerimo na varen HTTPS, kjer je povezava skozi celotno komunikacijo kodirana. Še posebej pomembno je v primerih, kjer preko svetovnega spleta pošiljamo občutljive podatke, kot so: gesla, razni bančni in osebni podatki. [23] [24]

Overjanje pri mobilnih aplikacijah deluje nekoliko drugače kot pri standardnih spletnih ali namiznih aplikacijah. Pogost arhitekturni stil pri izpostavitvi zalednih končnih točk je »Representational State Transfer« (v nadaljevanju REST in Rest), ki je brez stanja, zato moramo ob vsaki zahtevi zraven poslati še izbrane poverilnice. V klasičnih spletnih aplikacijah se pogosto uporabljajo seje in piškotki za overjanje uporabnikov, kar pa na mobilnih napravah odpade. Najbolj preprosti način overjanja v mobilnih aplikacijah je s pomočjo HTTP Basic overjanja, kjer, laično povedano, ob vsaki zahtevi pošljemo »ključ-vrednost« naših poverilnic. To predstavlja veliko varnostno luknjo, saj moramo le-te shranjevati lokalno, do njih pa se lahko napadalec dokoplje, če uporabi tehniko »reverse engineer«, s katero od nazaj naprej pridobi izvorno kodo naše aplikacije. Prav tako konstanto pošiljanje poverilnic preko svetovnega spleta za vsako zahtevo poveča možnost kraje na dolgi rok. [25]

Vse te probleme in še več rešuje protokol OAuth in njegova implementacija OAuth2, s katero zavarujemo naš zaledni sistem pred nezaupnimi napravami. Preko tega lahko uporabimo tako imenovan »Token Authentication« oziroma overjanje z žetoni, deluje pa po naslednjem toku:

- uporabnik preko mobilne aplikacije vpiše uporabniško ime in geslo, ki se preko varne povezave pošlje na zaledni sistem preko POST zahteve.
- Poverilnice se validirajo s podatkovno bazo in ob uspehu se kreira žeton, ki zapade čez določen čas. Žeton in dodatne informacije okrog njega se vrnejo uporabniku.

- Mobilna aplikacija potem uporablja ta žeton ob vsakem klicu na zaledni sistem, ki deluje kot API ključ.
- Ob poteku žetona je uporabnik spet primoran vpisati uporabniško ime in geslo. S tem se cikel spet ponovi.

Ključna prednost OAuth2 protokola je, da aplikaciji ni treba shraniti API ključa, ampak ga dobi iz zaledja kot proces generacije. Za žetone lahko uporabljamo poljuben, naključno generiran niz znakov. Dandanes je najbolj uveljavljen »Json Web Token« (v nadaljevanju JWT), ki je odprtokodni industrijski standard, ki bazira na JSON specifikaciji, za komunikacijo med dvema napravama. Je vedno kriptografsko podpisan in lahko vsebuje zakodirane podatke, s tem pa odpravimo tudi potrebo po shranjevanju žetonov v podatkovne baze. [25] [26]

2.3 Rest

Pogledali smo si samostojne komponente pri razvoju zalednih sistemov, kako delujejo in specifične lastnosti pri razvoju le-teh za mobilne aplikacije. Vse to pa je odveč, če skozi sistem ne steče ustrezna HTTP zahteva, ki potem dobi ustrezen, informacijsko bogat odgovor.

Zahtevo najprej obdela spletni strežnik izbire, ki glede na pot do vira, ali prenese zahtevo na instanco našega aplikacijskega strežnika, ali pa servira dokument kar sam. Po navadi je spletni strežnik zadolžen za vračanje statičnih vsebin, kot so: dokumenti html, javascript skripte in css stili. V našem primeru do pretoka informacij pride izključno preko tekstovnih formatov, kjer serviramo podatke v JSON obliki.

Vsaka HTTP zahteva ima določeno strukturo. Metoda zahteve indicira, katera akcija naj se izvede na končni točki vira. Metode vedno zabeležujemo z veliko začetnico in so preddefinirane. Dodelimo jih lahko v skupino idempotentnih, varnih ali tistih, ki jih lahko predpomnimo (v nadaljevanju »achable«). Izbiramo lahko med: [27]

- GET – idempotentna, varna, možnost predpomnjenja, zahteva predstavitev določenega vira, preko nje samo vračamo podatke;

- POST – uporabimo jo za kreacijo novih entitet na specifični končni točki, kar po navadi rezultira v spremenjenjem stanju na strežniku;
- PUT – idempotentna, zamenja trenutno predstavitev specifičnega podatka s tistimi v telesu zahteve;
- PATCH – uporabimo jo za zamenjavo specifičnega podatka. [27]

Zahteva vsebuje tudi polje zaglavja (angl. »header«), ki poda dodatne informacije o tipu zahteve, o klientu, o strežniku in kako naj slednji obdela zahtevo. So nekakšni »ključ-vrednost« pari, ki jih lahko poljubno definiramo. [28] [27]

Ko zahtevo prejme naš aplikacijski strežnik, najprej poišče ciljno končno točko, če ta obstaja. V nasprotnem primeru vrne status kodo 404, kar je dogovorjena številka, ko strežnik dokumenta ne najde. Če končna točka obstaja, se izvede overjanje uporabnika po izbrani metodi. Končne točke, ki so nezavarovane, ne potrebujejo overjanja in pogosto sprejemajo samo varne metode. Če je overjanje neuspešno, strežnik vrne napako v odgovoru s status kodo 401, kar je še ena dogovorjena številka ob zahtevi z neuspešnimi poverilnicami. Naslednji korak v uspešnem življenjskem ciklu zahteve je izvedba funkcije na končni točki. V našem primeru so koraki odvisni od metode. Na koncu algoritma za procesiranje nam aplikacija pripravi odgovor in ga preko odgovora pošlje v povratni zanki na cilj, od koder je prišla zahteva. Vsak odgovor ima prav tako telo, zaglavje in status kodo, nima pa metode. Kot vidimo v tabeli 1, so status kode vedno tri mestno število od 100 do vključno 511, delimo pa jih glede na prvo števko, gledano od leve proti desni. [28] [29]

Tabela 2.1: HTTP Status kode odgovora

Vzorec status kode	**Opis**
1xx Informacijski odziv	Nakazuje, da je bila zahteva sprejeta in razumeta. Klientu pove, naj počaka na končni odgovor, po dogovoru je sestavljen iz statusne linije in opcijskih »header«-jev.
2xx Uspeh	Nakazuje, da je bila zahteva sprejeta, razumeta in uspešno obdelana.

3xx Preusmeritev	Nakazuje, da zahteva potrebuje dodatne akcije klienta, da zaključi proces. Če je zahteva GET ali HEAD, se preusmeritev pogosto obdela avtomatsko.
4xx Napaka na klientu	Nakazuje, da je zahteva nepopolna oziroma je prišlo do napake zaradi klienta. Združuje status kode za nepopolne zahteve, napačne poverilnice, blokiranje dostopa, preveč zahtev in druge.
5xx Napaka na strežniku	Nakazuje, da strežniku ni uspelo obdelati zahteve zaradi napake pri procesiranju.

3 PRIMERJAVA OGRODIJ

Eden najpomembnejših korakov pri načrtovanju novega spletnega sistema, pa naj bo to na zaledju ali sprednji strani, je izbira ogrodja (angl. »framework«). V industriji ogrodje opišemo kot programsko knjižnico, ki nadomesti konvencionalno kodo z generičnimi funkcijami. Ogrodja so tukaj zato, da nam izboljšajo učinkovitost in hitrost razvoja. S tem porabimo manj časa za reševanje nekih trivialnih problemov in nam ostane več časa za implementacijo rešitve, algoritma, ki rešuje problem resničnega sveta. Prisilijo nas v uporabo konvencij, dobrih praks in s tem zmanjšujejo glavobole in konfuzijo pri vzdrževanju sistema. Ogrodja so načeloma tudi odprtokodna, za njimi pa stoji ekipa izkušenih razvijalcev, ki zadevo redno posodabljajo in vzdržujejo. [30] [31]

A ker so ogrodja tako dobra v sprejemanju odločitev za nas, pogosto postanemo leni, začnemo razmišljati, kako ogrodje hoče, da funkcionalnost implementiramo, ne glede na to, ali je rešitev elegantna in optimalna. Več ogrodij na projektu je med sabo izključljivih, zato enostavno ne moremo stlačiti enega v drugega in uporabiti najboljšega iz obeh svetov. Pogosto nam ogrodja odvzamejo kreativnost. Pri uporabi je treba, tako kot tudi v življenju, izbrati pravo razmerje med številom tujih stvari in naših rešitev. [30] [32]

3.1 Django rest framework

»Django Rest Framework« (v nadaljevanju DRF) je ogrodje, ki v osnovi temelji na komplementarnem ogrodju Django, ki pa sam po sebi ne podpira REST arhitekturnega stila. Slednji je visoko nivojsko ogrodje za gradnjo celovitih spletnih rešitev, napisano v jeziku Python. [33] [34]

Python je interpretiran, objektno orientiran, visoko nivojski programski jezik. Velikokrat na prvo žogo spominja na psevdokodo zaradi svoje, zelo »človeške« sintakse. Prva različica je izšla leta 1991 in se je, v nasprotju s sedanjostjo, bolj nagibala k funkcijskemu programiranju. 9 let kasneje je izšla še druga verzija, tretjo smo dobili 2008. Obe verziji se še danes aktivno razvijata in posodabljata, čeprav se bo uradna podpora druge verzije iztekla s prihajajočim letom. Python uporablja za orkestracijo paketov svoj »manager pip«. [35]

DRF uporablja veliko že implementiranih struktur Django-ta, kot so, na primer: modeli, url usmerjevalniki, nastavitve, k temu pa doda še svoj del v pretvornikih (angl. »serializer«) in pogledih. Pretvorniki omogočajo transformacijo kompleksnih podatkovnih struktur (na primer instance modelov in rezultatov rudarjenja po bazi) v domorodne Python podatkovne tipe, ki jih lahko kasneje, za potrebe REST-a, pretvorimo v JSON, XML ali ostale tipe vsebin. Omogočajo tudi obrnjen postopek, se pravi pretvorbo podatkov v kompleksne tipe. Pogledi (angl. »views«) so pred definirani vmesniki, ki jih uporabimo, da hitro sestavimo končne točke, ki se zelo blizu preslikajo na modele v podatkovni bazi. V ogrodju imamo tudi nize pogledov (angl. »viewsets«), s katerimi združimo logiko več posameznih pogledov v niz, pri čemer lahko hitro zgradimo CRUD (angl. »Create Read Update Delete«) končne točke. [33]

Pogosto se ogrodja označijo z značko mnenjski (angl. »opinionated«) ali fleksibilni (angl. »unopinionated«), kar v kratkem pomeni, da lahko zahteva točno določeno strukturo in način implementacije, ali pa je ta prosta in v rokah razvijalca. Mnenjski načeloma podpirajo hiter razvoj v določeni domeni (rešujejo probleme določenih tipov), čeprav so manj fleksibilni pri reševanju tretjih problemov. DRF-ji so t. i. kiti z mnenjsko značko, saj imajo pripravljene nize komponent, ki podpirajo večino nalog spletnega sistema. Pri uporabi lahko dobimo monolitno izkušnjo, saj je pot implementacije že začrtana, naša naloga je, da ji sledimo. [36]

Django in DRF imata oba komplementarno dokumentacijo, ki je zelo obširna ter porazdeljena na stopnje. Ker sta oba objektno orientirana, je dokumentacija spisana za vsak model posebej, znotraj le-te pa so opisana vsa polja in funkcije s primeri. Moje osebno mnenje je, da sta ti dokumentaciji eni najboljših, če ne najboljši, s katerimi sem delal. [33] [37] [38]

DRF za izvajanje uporablja več niti (angl. »multi-threading«) in s tem omogoča serviranje več zahtev naenkrat, vertikalna skalabilnost postane bolj efektivna. Ker so niti predhodno izpraznjene, nam ni treba skrbeti, da bi katerikoli proces sebično izkoriščal procesor. Tak model delovanja sicer zahteva precej virov. [37] [39]

Čeprav ima DRF (in Django) obširno in dobro dokumentacijo, je učna krivulja precej strma, veliko bolj kot pri, na primer, Express.js. Čeprav je Python tretji najbolj popularen programski jezik na GitHubu [40], na spletu manjka začetnih vodičev po ogrodju. To je posledica zelo dobre dokumentacije, ki je v tem primeru lahko dvorezni meč. Na strmo učno krivuljo vpliva tudi relativno kompleksna »Hello world!« aplikacija in cel kup inštrukcij za vzpostavitev okolje, ki zajemajo vsaj namestitev python-a, virtualnih okolij, raznih paktov in ogrodij ter ročno namestitev podatkovne baze. Veliko truda za malo rezultata. Po uspešni implementaciji je potrebno zadevo še namestiti na proxy strežnik in konfigurirati aplikacijski strežnik, za kar pa so, tako dokumentacija kot vodiči, precej šibki. [38]

Kot veliko prednost DRF se navaja njihov sistem brskanja po API-ju (angl. »Browsable API«) in že implementiranje metode overjanja. Prejšnji nam avtomatsko generira človeku prijazne HTML strani za vsako končno točko. Tem stranem je lahko slediti in uporabljati, hkrati pa nam ponujajo hiter vpogled v strukturo. DRF podpira OAuth1 in OAuth2 metode overjanja z mnogo ostalimi tehnikami, ki pa so po potrebi razširljive (na primer overjanje preko »Json Web Token«). [41] [42]

3.2 Express.js

Express.js (v nadaljevanju Express) je hitro, fleksibilno in minimalistično spletno ogrodje, ki ga poganja Node.js, sicer odprto-kodno okolje, ki uporablja JavaScript kot programski jezik. Express, laično povedano, je tako zelo minimalističen, da v bistvu vsebuje skoraj samo spletni strežnik in usmerjevalnik prometa, vendar ga lahko preko lastnih skript in knjižnic preuredimo do potankosti. [43] [44]

JavaScript (v nadaljevanju JS in .js) je visokonivojski, interpretiran skripti jezik, ki sledi smernicam ECMAScript, slednja pa je specifikacija za skriptne jezike, standardizirana iz strani Ecma International. Ustvarjena je bila kot osrednje telo za standardizacijo JS, ni pa to edini jezik, za katerega so prav slednji določili specifikacijo (na primer JScript in ActionScript). Zraven HTML in CSS je ena izmed glavnih tehnologij svetovnega spleta. JS prinese dinamiko v dinamične spletne strani, saj lahko v izvajanju manipulira z HTML, CSS in DOM (angl. »Document Object Model«). Za razliko od python-a uporablja zavite

oklepaje, vključno z vdolbinami (angl. »Indentation«); je prav tako dinamično pisan, to pomeni, da spremenljivke niso določene s tipom; uporablja prototipni, objektno orientiran model in implementira prvo razredne funkcije. [45] [46]

Express se ponaša s štirimi velikimi funkcionalnostmi, ki pokrijejo skoraj celoten sklad izdelave spletnega sistema. Sam po sebi ponuja niz robustnih komponent za izgradnjo aplikacij za splet ali mobilni svet. Najlažje si to zamislimo kot neki skupek Django-ta (ki ponuja izdelavo spletnih aplikacij) in DRF-ja (izdelavo API-jev). Če Django v svojih predlogah za dinamično prikazovanje strani uporablja jezik Smarty oziroma Jinja2, Express uporablja Jade, Handlebars, možnost pa ima kreirati svoj jezik za manipulacijo s predlogami. S pomočjo »middleware« funkcij in ostalih HTTP komponent lahko kreiramo API-je po mnogih arhitekturnih stilih, najbolje pa je podprt REST. Že v prvem stavku smo omenili, da je zelo hiter, saj je express samo tanka abstraktna plast nad Node.js HTTP plastjo. Tu pridobimo na hitrosti, saj nobene dodatne stvari ne ovirajo izvajanja – to pa pomeni tudi, da sam ne ponuja preddefiniranih visokonivojskih komponent, kot so, na primer, Djang-ovi Viewset-i. To so opazili mnogi kreatorji ogrodij in na express-u je zrasel še en velik ekosistem ogrodij, ki bazirajo in dopolnjujejo express z dodatnimi funkcionalnostmi in komponentami. Najbolj znan je Nest.js, ki je namenjen izdelavi tako imenovanih »enterprise« spletnih aplikacij. [43] [47]

Kar se tiče začrtanih vzorcev izdelave z express-om, je slednji zelo fleksibilen. Ne sili nas v uporabo nobenih oblikovnih vzorcev, niti nam ne ponuja začrtane strukture datotek, le-ta je prepuščena razvijalcu. Ni pa omejen navzgor, še zmeraj lahko implementiramo MVC (v dokumentaciji se celo na rahlo priporoča), MVP in druge. To je sicer velika prednost, ko gradimo svoje osebne projekte, a ko se začne ekipa širiti, lahko to pomanjkanje standardizacije postane veliko breme, tudi do te mere, da je projekt neobvladljiv. Takrat vstopi »middleware«, ki rešuje vsaj del tega problema, in sicer skozi dodatna vodila na življenjskem ciklu in skozi zahteve ter prinaša mero poslovne logike. Pogosto preko »middleware« funkcij preverjamo poverilnice, validiramo uporabniško stopnjo in dostop ter zavračamo CSRF napade. [43] [48] [49]

Node.js je nitno okolje, ki uporablja model dogodkovne zanke za izvajanje (angl. »Event loop«). Na prvo žogo se to sliši zelo slabo, kar se tiče hitrosti izvajanja. Ampak uporablja dogodkovno zanko z več delavci (angl. »workers«) v ozadju. Glavna zanka je eno nitna, ampak večina operacij se izvaja na ostalih nitih zato, ker so API-ji in Node napisani kot asinhrone funkcije. Med sabo se ne blokirajo pri izvajanju in ne čakajo ena na drugo. Zaradi tega je nastala paradigma zaobljube (angl. »promises«), ki je objekt predstave v pričakovanju dokončanja (oziroma padca) asinhrone operacije. V primeru, da ne želimo uporabljati zaobljub, lahko pričakamo, da se asinhrona funkcija izvede z uporabo zavarovane ključne besede »await« pred klicem funkcije. [50] [51]

Express, tako kot tudi JavaScript, ki je najbolj popularen jezik na GitHub-u, uživa precejšnjo podporo skupnosti. Na spletu lahko najdemo veliko število vodičev po raznoraznih tematikah in paradigmah express-a, čeprav same dokumentacije nima tako dobre kot, na primer, Django ali DRF. A je ne potrebuje, saj, razen osnovnih komponent, ne ponuja zapletenih funkcij in stilov. Zato pa je manager paketov za JS imenovan npm (njegova alternativa je yarn), dobro obložen z vrsto pomožnih knjižnic, ki vsebujejo dobro dokumentacijo. Ker je skupnost ogromna in se na časovnem intervalu kreira veliko vsebin in vodičev razvijalcem za JS in Express, je učna krivulja zelo položna. [40]

3.3 Firebase

Firebase je mobilna in spletna platforma za razvoj aplikacij, ustvarjena od istoimenskega podjetja, leta 2011, tri leta kasneje pa jih je svojim oblačnim storitvam dodal tudi Google. Firebase lahko opišemo kot storitev zalednega sistema (angl. »BaaS – Backend-as-a-Service«), ki nam, med drugim, ponuja shranjevanje dokumentov, uporabo ne relacijske podatkovne baze, storitev gostovanja in implementacijo zalednega sistema po meri, da naštejemo najvidnejše predstavnike. Prvinska naloga storitev je poenostaviti razvoj aplikacij na vseh plasteh: overjanje, zaledni sistemi, trajno shranjevanje podatkov in lansiranje ter skalabilnost sistema. [52] [53] [54]

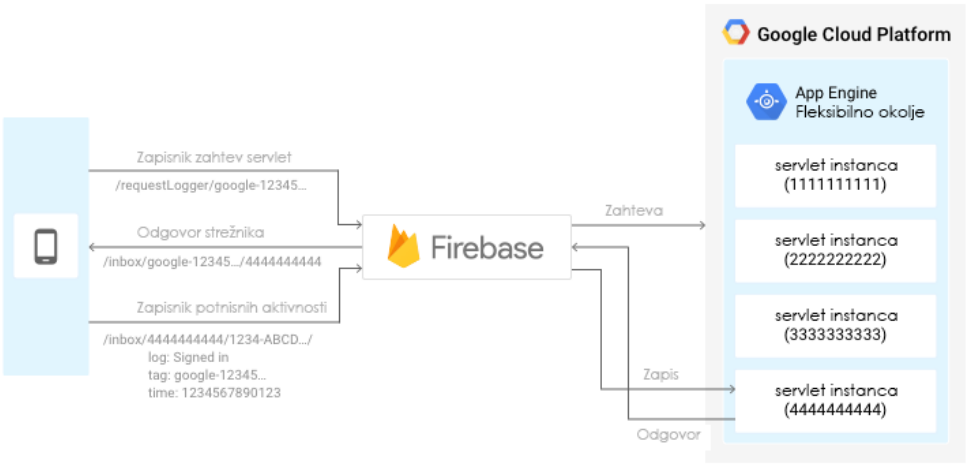
Da bi lahko razumeli prihodnje primerjave, je potrebno razumeti razlike med ogrodji, kot sta DRF in Express, ter BaaS in Firebase. Ogrodja ponudijo poti do izdelave sistema, s

tem, da med sabo povežejo vse potrebne komponente za razvoj. Na ogrodje lahko, na primer, povežemo katerokoli podatkovno bazo, ključnega pomena je tudi to, da moramo za strežnike skrbeti sami. Na drugi strani nam Firebase ustvari plast abstrakcije nad celotnim procesom izdelave sistema. Zaradi tega smo lahko omejeni na nekatere funkcionalnosti, čeprav naj bi s Firebase-om lahko ustvaril vse, kar lahko z express-om (predvsem zaradi »Firebase Cloud Functions« storitve). Največja prednost Firebase-a pa je sposobnost spremljanja posodobitev na podatkovni bazi v realnem času preko dogodkov in »serverless« pristopa. [54] [53] [55]

Firebase kot tak ne uporablja enega programskega jezika. Njihove uradne knjižnice so sicer primarno namenjene JavaScript razvijalcem sprednjega dela, vendar obstajajo knjižnice za vse bolj popularne jezike, narejene iz strani skupnosti, in uradno podprte od ustvarjalcev Firebase-a. V primeru, da uporabljamo nepodprt jezik, nam je kot zadnja, vendar še vedno močna, alternativa na voljo celotna REST API zbirka končnih točk, s katerimi lahko ustvarimo vse, kar je možno z uradnimi knjižnicami in tistimi, ustvarjenimi s strani skupnosti. [56]

Firebase kot vodilno funkcionalnost, zaradi nje je podjetje sploh nastalo, omenja podatkovno bazo v realnem času. V klasičnih shrambah podatkov, ki jih lahko priključimo na Express, DRF in ostale, nam baza servira podatke po tem, ko mi za njih zaprosimo preko HTTP ali katerega koli drugega klica. Firebase pa zadevo obravnava nekoliko drugače. Ko se povežemo na Firebase bazo (imamo dva tipa – Firestone in Realtime, obe ne relacijski), se povežemo preko WebSocket protokola. Slednji implementira duplex komunikacijski protokol, pri katerem se povezava med ponudnikom in odjemalcem ohrani. Tako nam lahko Firebase baza servira nove podatke takoj, ob posodobitvi kolekcij. WebSocket-i so tudi bistveno hitrejši od HTTP povezave, saj za vsako sporočilo ni potrebno izpeljati postopka overjanja. Da implementiramo tak algoritem na standardne bazne spremembe, bi v DRF uporabili signale na modelih in jih preko knjižnice Django Channels poslali odjemalcu, v Express-u pa bi uporabili »post save« funkcije, katere bi preko Socket.io knjižnice servirali na ustrezne končne točke. [53] [57] [58]

Funkcionalnost, zaradi katere lahko primerjamo ogrodja S Firebase-om, pa je imenovana »Cloud Functions«. Slednje omogočajo izvajanje zaledne kode kot odzive na dogodke katerekoli od Firebase lastnosti ter HTTP zahtev. Namestimo lahko katerekoli knjižnice ter jih povežemo s poljubnim tretjim ogrodjem. Lahko bi rekli, da z njimi zmoremo vse, kar zmoremo lokalno preko Express-a in DRF-ja. Prednost je ta, da se kode izvajajo v oblačnih storitvah, do poklica je v nekakšnem stanju »spanja«. Z uporabo »Cloud Functions« se izognemo urejanju strežnika in problemov s skalabilnostjo, saj nam to zadevo uredi Firebase. Tak pristop imenujemo »serverless« oziroma brezstrežniški pristop. Na voljo imamo razne sprožilce, ki določijo, kdaj naj se te funkcije zaženejo. Najbolj pogosti in sovpadljivi z ogrodji so sprožilci na postopku overjanja (angl. »Firebase Authentication Triggers«), sprožilci na podatkovni bazi v realnem času (angl. »Cloud Firestore Triggers in Realtime Database Triggers«) in HTTP sprožilci (angl. »HTTP Triggers«). V primeru, da je instanca še zaposlena z neko nalogo, nam Google hitro kreira več delavcev (angl. »workers«), da se delo opravi hitreje, kot to vidimo na sliki 3.1. Na platformi imamo več instanc aplikacije v eni oblačni storitvi. Če pa funkcionalnost miruje, se instance pobrišejo. Zaradi tega dinamičnega dodajanja in odvzemanja instance lahko čas za izvedbo ene naloge zelo variira. Pri razvoju sistema z uporabo »Firebase Functions« se mora vsaka koda, za pravilno delovanje in odzivanje, naložiti na »Google Cloud«, pri čemer sta lahko razvoj in testiranje precej zamudna, če imamo šibko internetno povezavo. [59] [60]



Slika 3.1: Pregled serverless gostovanja - več instanc aplikacije na strežniku [61]

Firebase izven svojih zmožnosti in API-jev ne začrta oblikovnih vzorcev in strukture aplikacije, dokler le-ta vsebuje vse potrebne datoteke. Po fleksibilnosti leži nekje med express-om, ki je čisto svoboden in Django-om, ki nam vsili način implementacije, vendar je še vedno bližje fleksibilnim kot strukturiranim ogrodjem. [53]

Firebase sicer nima dogovorjenega jezika uporabe, vendar kljub temu v vseh svojih izrezkih primerov primarno uporablja Javascript. Google ima, kot je znano, vrhunsko dokumentacijo za vse svoje storitve in prav nič ni drugače pri Firebase-u. Dokumentacija je razčlenjena na poglavja, glede na funkcijo, tej pa sledijo še podpoglavja, glede na primere uporabe. Opisi postopka so na veliki stopnji abstrakcije, tako da jo lahko implementiramo s poljubnim programskim jezikom. K dobrim vodičem uradne dokumentacije lahko dodamo še ogromno člankov s strani skupnosti, kot je to značilno za vsak Googlov produkt. [62]

V nasprotju z Express-om in DRF-om, ki sta »samo« ogrodji, je Firebase celotna zbirka funkcionalnosti, ki je z obsegom precej večji od obeh. Praksa pri učenju takih zadev je, da se nikoli ne učimo zadeve kot celote, ampak uporabljamo principe, kot je, na primer: »deli in vladaj«. Pogosto tudi ne rabimo vseh funkcionalnosti Firebase-a, zato poberemo samo tiste, ki nam prinašajo dodano vrednost. S tem Firebase, kljub solidni

dokumentaciji, predstavlja veliko novih konceptov gradnje in uporabe, s čimer je učna krivulja precej strma in predvsem dolga. Ker je zadeva v lasti multinacionalke, se zadeve hitro spreminjajo, zato nam prejšnje znanje ne prinese dolgoročnega uspeha, ampak ga je treba neprestano nadgrajevati. [63]

3.4 Dobre prakse

Ko govorimo o dobrih praksah, ni točne definicije, kaj od razvojnega ciklusa spada v zaledne sisteme. Če gradimo zaledni sistem za spletno aplikacijo, imamo drugačne specifikacije in probleme kot, če implementiramo sisteme za IoT (angl. »Internet of Things«), ali za mobilno platformo. Za naslednje dobre prakse predpostavimo, da gradimo RESTFul API zaledni sistem, ki kot vsebino servira samo JSON datoteke, kot odjemalca pa definiramo mobilno aplikacijo na obeh večjih operacijskih sistemih, to sta Android in iOS.

3.4.1 Predpomnjenje

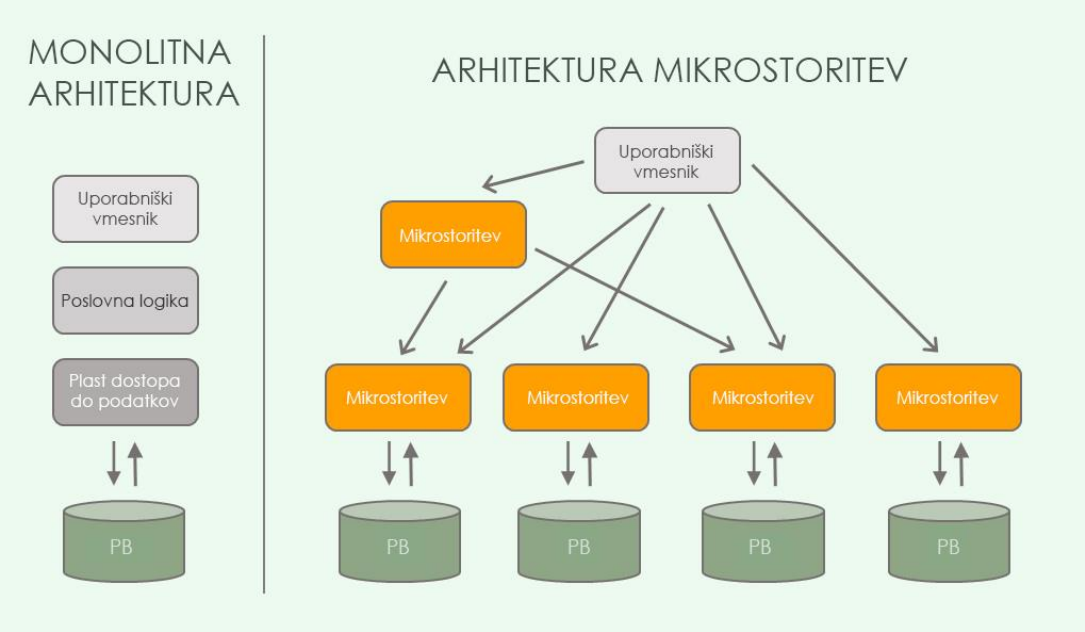
V računalništvu, predpomnjenje (angl. »cache«, v nadaljevanju cache), je plast hitrega shranjevanja začasnih delov podatkov, za potrebe hitrega odziva strežnika na že znano zahtevo, namesto da bi ob vsaki novi zahtevi znova in znova dostopali do trajnih podatkov. Caching omogoča, da efektivno uporabimo že prej vrnjene ali preoblikovane podatke. V distribuiranih programskih okoljih namenska predpomnilniška plast omogoča instancam aplikacije, da izvajajo svoj življenjski cikel brez interference ene z drugo. Cache deluje kot mediator in je lahko dostopen do mnogih aplikacij ter njihovih topologij. To je še posebej pomembno pri okoljih, kjer se instance dinamično dodajajo in odvzemajo glede na promet. Za validnost podatkov v cache poskrbimo s tako imenovanim časom življenja podatka (angl. TTL – »Time to live«). Cache-amo torej podatke, ki se periodično ponavljajo glede na zahtevo (na primer lista prijateljev uporabnika) in podatke, ki imajo kratek čas življenja (na primer rezervacija artikla v košarico). [64]

3.4.2 Varnost

Osnovni komunikacijski protokol na svetovnem spletu je sicer HTTP, vendar mobilne platforme uveljavljajo HTTPS z modernimi metodami šifriranja, uporabo certifikatov in algoritma javni-privatni ključ. Zaledni sistem za mobilne aplikacije mora biti dostopen samo preko HTTPS, saj smo brez njega ranljivi za zlonamerne napade tipa »Man-in-the-middle«, kjer napadalec prestreza zahteve, ki lahko vsebujejo uporabniške informacije, kot so: gesla in številke bančnih kartic. Priporočljiva je uporaba SSL certifikata, tako na produkcijskem kot na testnem okolju. S tem odpravimo težave protokola že na testnem okolju, tako da nimamo težav pri migraciji. V produkciji ključne spremenljivke vedno definiramo v okolju (angl. »environment variables«) in jih ne potisnemo v odlagališče na sistemu za upravljanje z izvorno kodo. Sem sodijo razni ključi, gesla, certifikati in uporabniška imena za tretje storitve, podatkovne shrambe in dostop do drugih okolij. [65] [66]

3.4.3 Mikrostoritve

Moderne zaledne rešitve, še posebej tiste, ki jih označimo z »serverless« opcijo, se pogosto držijo arhitekturnega stila mikrostoritev (angl. »microservices«). Skozi implementacijo sistema se srečamo z ogromno moduli, preferenčno šibko sklopljenimi, ki opravljajo svojo funkcionalnost, na primer: modul overjanja, modul plačevanja, modul prijatelji, modul urejanja osebnih podatkov in tako dalje. Na sliki 3.2 lahko vidimo primerjavo med klasično, monolitno arhitekturo, kjer imamo poslovno logiko močno sklopljeno v en modul, ter arhitekturo manjših mikrostoritev. Šibko sklopljene storitve omogočajo ponovno uporabnost na drugih aplikacijah, s tem prihranimo čas, denar in vire pri implementaciji. Taka implementacija temelji na mikrostoritvah, ki omogočajo rapiden, pogost in zanesljiv razvoj kompleksnih, za podjetja pripravljenih (angl. »enterprise«) aplikacij. Pri implementaciji REST API-ja lahko tako arhitekturo uporabimo pri ločevanju končnih točk. Medtem ko sta strani za potrditev emaila in ponastavljanje gesla dostopni na naslovu »/app«, lahko REST API prikažemo na naslovu »/api/v<ŠTEVILKA_VERZIJE>«, plačilni sistem pa na »/payment« končni točki. Tako lahko razvijemo in uvajamo vsak modul posebej, ko je ta pripravljen. [67] [68] [69]



Slika 3.2: Arhitektura mikrostoritev [70]

3.4.4 Strežnik vedno na voljo

Da prihranimo stroške pri prenosu podatkov in baterijo mobilne naprave, nam je v interesu, da slednja opravlja čim manj dela oziroma podatke samo prikazuje. Čeprav so mobiteli po zmogljivosti primerljivi z nizkocenovnimi računalniki, redko srečamo mobilno napravo, ki ne bi imela koristi od tega, da se podatki obdelujejo na tretjem, bolj zmogljivem sistemu. Če podatke pridobivamo iz API-ja, so le-ti bolj varno shranjeni, sinhronizirani čez vse naprave, mobilne naprave pa se lahko fokusirajo na domorodne funkcionalnosti. Pomembno je, da preselimo čim več poslovne logike na zaledne sisteme in s tem ohranimo mobilno aplikacijo lahkotno. Z vsemi temi tehnikami prenesemo veliko breme na strežnik, ki mora biti dovolj močan, da obravnava vse prihajajoče zahteve. V primeru preobremenitve in padca strežnika lahko to hitro pomeni tudi konec uporabljanja mobilne aplikacije. Uporabniki le-teh imajo le malo potrpljenja, če zadeva ne deluje, kot je obljubljeno oziroma kot so navajeni. Treba si je zagotoviti dovolj dober sklad, ki bo znal odreagirati ob padcu sistema in bo, ob večjih obremenitvah, znal dodati sveže instance za hitrejše delovanje. [65]

4 ZALEDNI SISTEM NA PRIMERU

4.1 Priprava okolja

Za pravilen razvoj zalednega sistema oziroma kakega koli sistema, si moramo pripraviti ustrezno razvojno okolje, ki vsebuje ustrezna orodja in programe, ki nam omogočajo implementacijo in pomagajo pri uravnavanju stroškov, časa ali produktivnosti. Ker je razvoj programske opreme nenehen proces, je praksa taka, da si pripravimo tri ločena razvojna okolja, ki se razlikujejo predvsem po specifični uporabi virov: [71]

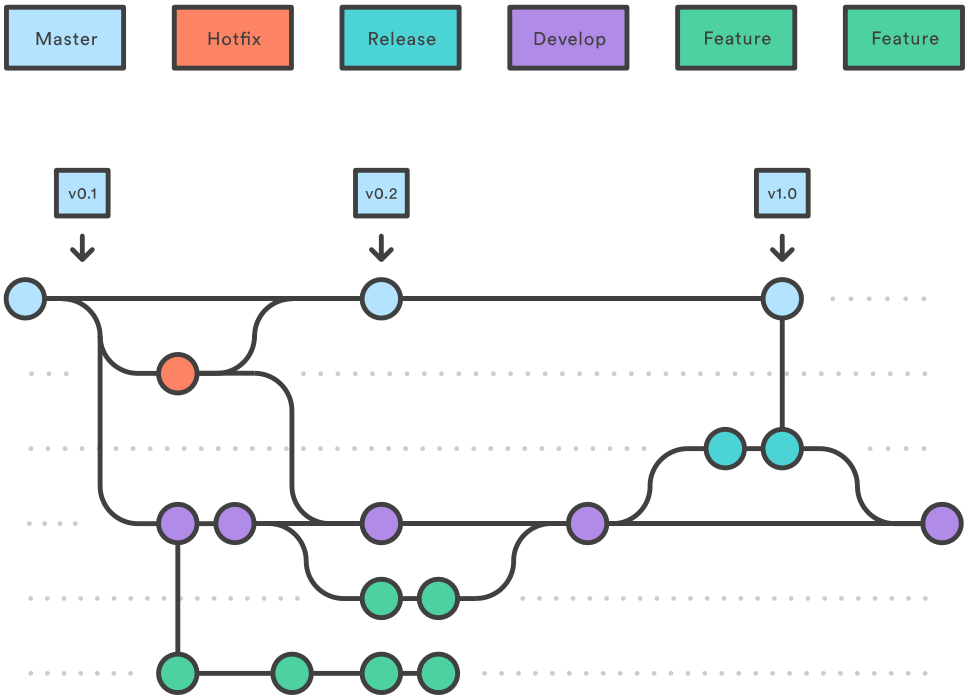
- razvojno okolje – okolje, v katerem ekipa razvijalcev piše kodo, eksperimentira, implementira nove funkcionalnosti, poganja teste in odpravlja hrošče. [71]
- Testno okolje (angl. »Staging«) – okolje, ki naj bi bilo identično kot produkcijsko, vendar še vseeno prilagojeno za nenehno testiranje in poganjanje integracijskih testov (na primer plačevanje s testnimi karticami). Tako lahko zagotovimo kvaliteto pod produkcijskimi pogoji, še preden sistem dejansko uvedemo v javno rabo. [71]
- Produkcijsko okolje – okolje, ki uporablja namensko strojno opremo in zadnjo stabilno verzijo sistema, je prosto dostopno uporabnikom, nanj so vezani produkcijski odjemalci (angl. »frontend«). Je zelo občutljivo in predstavlja blagovno znamko na spletu. [71]

V našem primeru si bomo pogledali, kako pripraviti razvojno okolje.

Razvijalci kodirajo na različnih operacijskih sistemih, pa naj bo to Windows, iOS ali Linux. Končni sistemi pa se skoraj vedno izvajajo na neki verziji Linuxa. Da bi bilo razvojno okolje neodvisno od računalnika razvijalca, uporabimo orodje Vagrant. Le-ta omogoča gradnjo in upravljanje okolij v virtualnih strojih, v enem samem delovnem ciklu. Omogoča nam hitro namestitev in izbiro izmed mnogo že prej konfiguriranih okolij in operacijskih sistemov, reproduciranje in samo eno datoteko, v kateri definiramo ciljno mapo, določimo, katere t. i. porte želimo odpreti, kako dostopati do virtualnega stroja in tako dalje. Preko namenske spletne strani prenesemo Vagrant in VirtualBox ter poženemo

»vagrant init«. S tem si inicializiramo verzijo vagranta, ki jo zaženemo z »vagrant up«, do nje pa se preko ključev SSH povežemo s komando »vagrant ssh«. [72]

Za sistem za upravljanje z verzijami bomo uporabili t. i. git. Kot nadgradnjo le-tega in za bolj strukturirano delovanje v ekipi, bomo uporabljali Gitflow vzorec, ki definira strikten model vejitve in poimenovanja okrog projekta. Vzorec ne doda nobenih novih konceptov ali komand, le poskrbi za specificirano vejitev in njihovo medsebojno interakcijo. Kot je razvidno iz slike 4.1, definira dve glavni vejitvi: »develop« in »master«. Prejšnja služi kot veja za integracijo novih funkcionalnosti, master pa kot uradna stabilna veja z vso zgodovino. Zraven tega definira 3 začasne tipe vejitev. Prva taka je veja »feature« oziroma funkcionalnost. Veji se izven develop-a ter se vanj tudi združuje, ko je le-ta funkcionalnost implementirana in testirana. Nikoli ni v interakciji z vejo master. Druga je veja »release« oziroma veja za izdajanje neke verzije produkta. Tudi ta se veji izven develop-a, v njej pa se generira dokumentacija, popravki ter ostala priprava za končno verzijo produkta. Ko je vse urejeno, se združi v vejo master. Zadnja začasna vejitev je »hotfix«, ki je namenjena nujnim hitrim popravkom in je edina, ki se veji izven master-ja. Ko je popravek narejen, se hotfix vejitev združi v obe glavni veji – master in develop, kot je razvidno na sliki 4.1. Tako imamo pripravljen neodvisni operacijski sistem in git model, ki podpira večje skupine. [73]



Slika 4.1: Gitflow diagram z vsemi vejitvami [73]

4.2 Arhitektura

V sklopu zalednega sistema je, za produkcijsko uporabo, premalo, da imamo samo aplikacijski strežnik. Le -ta bo uporabljen samo za uveljavljanje poslovnih pravil. Zahtevo klienta najprej sprejme proxy strežnik, v našem primeru smo uporabili Nginx server, ki je poskrbel za 4 stvari:

- uravnavanje obremenitev (angl. »load balancing«) – zahteve bo preusmerjal na katerokoli od naših -n instanc. S tem se obremenitev porazdeli in zahteva je hitreje obdelana.
- Predpomnjenje (angl. »caching«) – skrbi za predpomnjenje pogostih vsebin našega aplikacijskega strežnika in s tem dodatno razbremeni sistem.
- Varovanje – nginx ima implementirane varnostne protokole proti napadom DDoS (angl. »Distributed Denial of Service«), kjer napadalci poskušajo sesuti aplikacijo s preobremenitvijo. [74]
- SSL (angl. »Secure Sockets Layer«) – na strežnik namestimo SSL certifikat in uredimo povezavo samo preko varne HTTPS povezave.

Za njim smo postavili implementacijo rešitve na aplikacijski strežnik z več instancami, ki bodo imele izpostavljene končne točke za uporabo na sprednji strani. Za aplikacijsko predpomnjenje, posredovanje sporočil in javno oddajanje preko WebSocketov smo pripravili nerelacijsko »ključ-vrednost« podatkovno bazo Redis. Za trajno shranjevanje podatkov bomo uporabili podatkovno bazo, katere tip je odvisen predvsem od funkcionalnosti sistema. V našem primeru bomo uporabili relacijsko bazo PostgreSQL, primerna bi bila tudi MySQL, Oracle, MariaDB ter katerakoli druga relacijska baza. Specifična izbira je izven domene tega zaključnega dela. Odjemalca celotne rešitve bosta mobilna operacijska sistema iOS in Android.

4.3 Funkcionalnosti

V nadaljevanju bomo implementirali zaledni sistem za mobilne rešitve, ki služi kot družabnik (angl. »companion«) obstoječemu dnevniku oziroma plannerju. Kot fizični produkt imamo pametni dnevnik, ki ima že v oblikovanju podporo za razpoznavanje celic skozi kamero na aplikaciji. Slednja omogoča lahko organizacijo in sortiranje dnevnih nalog, skozi poljubno uporabniško hitrost. Eno deluje z roko v roki z drugo. Naloga backenda je podprtje sistema skozi implementacijo naslednjih funkcionalnosti:

- overjanje uporabnikov – ključna in osnovna funkcionalnost pri zagotavljanju varnega sistema. Specifikacija od nas zahteva registracijo preko forme na appu (lokalna) in vpis v sistem preko uporabniškega imena in gesla, vpisanega pri registraciji. Omogočiti moramo tudi overjanje in vpis preko socialnih ponudnikov: Google, Facebook in Twitter. Za vsako od teh načinov overjanja moramo implementirati svoj vpisni pretok.
- Podpora nakupu licence – ko uporabnik kupi fizični dnevnik, dobi na zadnji strani QR kodo, ki vsebuje preddefinirani niz znakov. Slednji deluje kot licenčna koda, ki odobri uporabo aplikacije uporabniku do konca leta, v katerem je kupljen. Ker je dnevnik enoletni, bi bilo neumno, da slednji omogoči uporabo aplikacije še za drugo leto. Če pa uporabnik ne kupi fizičnega dnevnika, lahko še zmeraj uporablja vse funkcionalnosti aplikacije, vendar mora kupiti licenčno kodo preko nakupov znotraj aplikacije (angl. »in-app purchase«).

- Uporabniške funkcionalnosti – uporabnik ima možnost urejanja profila, iskanje, dodajanje in odstranjevanje prijateljev ter mesečnega pregleda aktivnosti.
- Podpora različnim elementom dnevnika – definira tri tipe elementov: »dailyplan« (razpored dneva), »checklist« (seznam) in »note« (opombe). Vsaka od teh ima različne atribute (na primer: datum, kategorija, značke in podobno) ter vsebuje -n celic, ki so lahko ali tekstovne ali slikovne. Celice se lahko zapolnijo ročno ali pa preko skeniranja celic v dnevniku preko aplikacije. Uporabnik lahko preišče elemente po vseh dodatnih atributih.
- Obveščanje preko potisnih sporočil – uporabnik si lahko v aplikaciji določi ob kateri uri in koliko prej ga naj sistem obvesti o obveznosti. Tako funkcionalnost bomo dosegli z uporabo FCM (angl. »Firebase Cloud Messaging«), ki omogoča pošiljanje potisnih sporočil na obe platformi.

4.4 Primer rešitve

Za potrebe demonstracije rešitve bomo naštete funkcionalnosti implementirali v ogrodju Django Rest Framework, uporabili pa bomo tudi Firebase kot ponudnika za pošiljanje potisnih sporočil. Uporabili bomo verzijo Python 3.6, Django 2.1.2 ter Django Rest Framework 3.9.0. V nadaljevanju si bomo pogledali najpomembnejše odseke kode za naštete funkcionalnosti.

Da bi razumeli ločevanje samih modulov med sabo, si bomo najprej ogledali, kako smo si oblikovali skupine končnih točk. Na sliki 4.1 imamo, od vrstice 45 do vključno 50, prikazano ločevanje modulov na sklope. Naš sistem bomo mobilni platformi predstavili na delu »api/v1/«, kar označuje, da so naslednje končne točke vmesnik za klienta. Admin in Docs končni točki sta namenjeni razvijalcem v testnem okolju ter naročniku v produkcijskem. Imamo torej štiri glavne poddomene, in sicer se vrstica 46 nanaša na overjanje, vrstica 47 na uporabniške možnosti, 48 na manipulacijo s komponentami in elementi dnevnika, 50 na registracijo naprav za pošiljanje potisnih sporočil.

```
44  urlpatterns = [
45      path('admin/', admin.site.urls),
46      path('api/v1/', include('social_login.urls')),
47      path('api/v1/users/', include('userprofile.urls')),
48      path('api/v1/components/', include('elements.urls')),
49      path('docs/', include_docs_urls(title='OrgaNicer API docs')),
50      path('api/v1/devices/', FCMDeviceAuthorizedViewSet.as_view({'post': 'create'})),
51      #url(r'^swagger(?P<format>\.json|\.yaml)$', schema_view.without_ui(cache_timeout=0), name='schema-json'),
52      #url(r'^swagger/$', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
53      #url(r'^redoc/$', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
54  ]
55  # urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
56  # urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

Slika 4.2: Izhodiščno usmerjevanje končnih točk

Lokalna registracija je precej preprosta, saj razen validacije polj in zapisovanja v bazo ne kreiramo nič posebnega. Drugače je pri toku overjanja preko Google-a, Twitter-ja ali Facebook-a. Pri slednji dobimo iz klienta dostopni žeton (angl. »access token«) ter ciljnega ponudnika. S slednjima dvema ter parom ključev za korespondenčnega ponudnika naredimo zahtevo na njihov zaledni sistem, kjer opravimo validacijo žetona. V primeru, da tok steče brez težav, dobimo v odgovoru tista polja, ki smo jih zahtevali (angl. »scope«), po navadi je to vsaj ime, email in datum registracije. Zakaj po navadi? Ker se ta polja razlikujejo od ponudnika do ponudnika. Na sliki 4.2 lahko vidimo celoten tok socialnega overjanja, najbolj pa nas zanimata vrstici 266 in 268. Pri prejšnji naznanimo, da želimo uporabnike z istim email naslovom sklopiti v en račun na našem sistemu. Na primer, uporabnik se je nekoč registriral z Google ponudnikom, čez nekaj časa pa še s Facebook-om. V primeru, da je email, ki ga dobimo ob validaciji žetona isti, sklopimo obstoječ in novi račun.

```
258 SOCIAL_AUTH_PIPELINE = (
259     'social_core.pipeline.social_auth.social_details',
260     'social_core.pipeline.social_auth.social_uid',
261     'social_core.pipeline.social_auth.auth_allowed',
262     'social_core.pipeline.social_auth.social_user',
263     'social_core.pipeline.user.get_username',
264     'social_core.pipeline.mail.mail_validation',
265 
266     'social_core.pipeline.social_auth.associate_by_email',
267     'social_core.pipeline.user.create_user',
268     'userprofile.pipeline.save_profile',
269     'social_core.pipeline.social_auth.associate_user',
270     'social_core.pipeline.social_auth.load_extra_data',
271 )
```

Slika 4.3: Tok overjanja preko tretjega ponudnika

V vrstici 268 na sliki 4.2 definiramo tok po meri, v katerem uveljavljamo posebna poslovna pravila za sistem. Ker nam različni ponudniki vračajo različno členjena polja, si jih moramo preko adapterja prilagoditi za sistem. Na sliki 4.3 lahko vidimo taki adapter za overjanje preko Facebook-a. V profilu, ki je povezan z uporabnikom, moramo zajeti ključne podatke. S tem je overjanje uspešno zaključeno, uporabniku pa v naslednjem koraku generiramo žeton za uporabo našega vmesnika.

```
7  def save_profile(backend, user, response, *args, **kwargs):
8      if backend.name == "facebook":
9          #print(response)
10         if not user.first_name:
11             User.objects.filter(id=user.id).update(first_name=response['first_name'],
12                                                    last_name=response['last_name'],)
13         try:
14             UserProfile.objects.create(
15                 id=user.id,
16                 user=user,
17                 photo_social_url=response['picture']['data']['url'],
18                 name=response['name'], )
19         except IntegrityError:
20             UserProfile.objects.filter(user=user).update(photo_social_url=response['picture']['data']['url'],
21                                                          name=response['name'], )
```

Slika 4.4: Poslovna pravila po meri - Facebook

Uporabnik si mora po registraciji računa pridobiti naročnino, s katero mu je omogočeno uporabljanje aplikacije, ki so za plačilnim zidom. Ima možnost prebrati kodo na zadnji strani dnevnika ali opraviti plačilo znotraj aplikacije. V obeh primerih ustvarimo naročniški model, ki ga povežemo z uporabnikom. Na sliki 4.4 je prikaz Django modela,

ki se potem, preko orodja za modeliranje objektov v tabele (angl. ORM – »Object-relational mapping«), preslika v bazo. Vsaka naročnina je ali mesečna ali pa letna in izhaja iz treh prej omenjenih virov. Če v zahtevo dobimo polje »ios_receipt« oziroma »transaction_id« moramo oba verificirati pri ponudniku plačil znotraj aplikacije. V primeru, da dobimo samo polje »qrcode«, moramo samo razčleniti kodo iz formata, ji preveriti validnost in jo povezati z uporabnikom.

```
50  class Subscription(models.Model):
51      ONE_YEAR = '1Y'
52      ONE_MONTH = '1M'
53      TYPES_OF_SUBSCRIPTIONS = (
54          (ONE_YEAR, 'ONE_YEAR'),
55          (ONE_MONTH, 'ONE_MONTH'),
56      )
57      QRCODE = 'QRcode'
58      ANDROID = 'Android'
59      IOS = 'iOS'
60      TYPES_OF_SOURCES = (
61          (QRCODE, 'QRCODE'),
62          (ANDROID, 'ANDROID'),
63          (IOS, 'IOS')
64      )
65
66      user = models.ForeignKey('auth.User', on_delete=models.CASCADE, related_name='subscriptions')
67      qr_code = models.ForeignKey(QRCode, on_delete=models.CASCADE, related_name='qrcode', null=True)
68      valid_from = models.DateField(null=True, blank=True)
69      valid_to = models.DateField(null=True, blank=True)
70      ios_receipt = models.CharField(null=True, max_length=7000, blank=True)
71      transaction_id = models.CharField(null=True, max_length=191, blank=True)
72      type = models.CharField(max_length=20, choices=TYPES_OF_SUBSCRIPTIONS, default=ONE_YEAR, )
73      source = models.CharField(max_length=191, choices=TYPES_OF_SOURCES, null=False)
74
75      class Meta:
76          verbose_name = 'Subscription'
77          verbose_name_plural = 'Subscriptions'
78
79      def __str__(self):
80          return "{} (source: {}, duration: {})".format(self.user.username, self.source, self.type)
81
```

Slika 4.5: Naročniški model

Pri implementaciji uporabniških razmerij oziroma prijateljstev se srečamo s povezavo mnogo-mnogo na isto tabelo User. To je unikatni vzorec, ki ga začnemo reševati z dodatno vmesno tabelo, ki bo služila za razrešitev mnogo-mnogo povezave v dve ena-mnogo povezavi. Dalje smo prijateljstva implementirali na način, da v vmesni tabeli hranimo oba uporabnika v razmerju v svojih poljih, stanje tega razmerja, ki je lahko od «neobstoječega«, »na čakanju«, »prijateljstvo sprejeto« do drugih. Najbolj pa nas

zanima, kdo je sprožil zadnjo zahtevo oziroma kdo je zadnji posodobil status. Zato tudi hranimo ID uporabnika zadnje akcije. Na podlagi teh štirih polj lahko potem implementiramo pravila razmerja v smislu, kdo lahko spremeni status in na kaj. Preprečiti moramo, na primer, da uporabnik sproži prošnjo za razmerja in jo tudi potrdi. V tem primeru je ključno polje ID uporabnika zadnje akcije, ki preverja, na kateri strani je žogica, kdo lahko servira katero akcijo. Na sliki 4.5 lahko vidimo tipično komponento DRF-ja, to je viewset. Naš razred podeduje splošni viewset »GenericViewSet«, ki omogoča nadaljnje dedovanje iz paketa »mixins«, ki vsebuje dodatne funkcije in polja za preprosto kreacijo CRUD končnih točk. V tem primeru uporabljamo listanje razmerij »ListModelMixin« (pregled vseh razmerij v vseh stanjih), detajlni pregled enega »RetrieveModelMixin«, ustvarjanje novih razmerij »CreateModelMixin« (pošiljanje nove prošnje za prijateljstvo) ter posodabljanje stanja razmerja »UpdateModelMixin«. V nadaljevanju definiramo, katere dostopne pravice mora uporabnik imeti za dostop do te končne točke. Ker je vse vezano na uporabnika, mora seveda biti overjen in za detajlne poglede imeti svoje mesto v razmerju. V vrstici 293 definiramo, kateri model naj pogledi uporabljajo, v naslednji pa, kako naj se instanca preslika v berljiv JSON format preko serializatorja (angl. »serializer«). Metodi življenjskega cikla pogleda »get_throttles« in »get_queryset« nam definirata število klicev na določen časovni interval ter podrobno filtriranje instanc za vsak pogled, v podanem vrstnem redu.

```
class UserRelationshipViewSet(viewsets.GenericViewSet,
                              mixins.ListModelMixin,
                              mixins.RetrieveModelMixin,
                              mixins.CreateModelMixin,
                              mixins.UpdateModelMixin):
    """

    pending:
    Returns the list of pending friend invites with user and relationship_id

    """
    permission_classes = [permissions.IsAuthenticated, TokenHasReadWriteScope, HasPartInRelationship]
    queryset = UserRelationship.objects.all()
    serializer_class = UserRelationshipSerializer

    def get_throttles(self):
        if self.action in ['invite']:
            self.throttle_scope = 'friends.invite'
        return super().get_throttles()

    def get_queryset(self):
        if self.action == 'list':
            queryset = User.objects.all().filter(Q(user_first_relationship__user_second_id=self.request.user.id,
                                                   user_first_relationship__type=UserRelationship.FRIENDS) |
                                                 Q(user_second_relationship__user_first_id=self.request.user.id,
                                                   user_second_relationship__type=UserRelationship.FRIENDS)).distinct()
            return queryset
        if self.action == 'retrieve':
            return self.queryset.filter(
                Q(user_first=self.request.user) |
                Q(user_second=self.request.user),
            ).distinct()
        if self.action == 'update':
            return self.queryset.filter(
                Q(user_first=self.request.user) |
                Q(user_second=self.request.user),
            ).distinct()
        return self.queryset
```

Slika 4.6: Nabor pogledov uporabniških razmerij

Seveda pa je glavna funkcionalnost aplikacij, ki delujejo kot družabniki, podpora glavnemu produktu – dnevniku. Ker komponente slednjega zahtevajo deljenje celic in zapisov z drugimi uporabniki, smo podobni sistem, kot so razmerja, uporabili pri deljenju komponent. Pri skupnem delu več uporabnikov na eni instanci je pomembno, da se spremembe prikazujejo v realnem času. Da smo dosegli željen cilj, smo uporabili protokol vtičnikov (angl. »WebSocket«), ki ga za Django implementira knjižnica Channels. V aplikaciji je smiselno deliti opombe in sezname (»Note« in »Checklist«), zato smo, kot vidimo na sliki 4.6, definirali funkcijo, ki jo prožijo dekoraterji prej omenjenih komponent na življenjskih ciklih shranjevanja (to zajema kreacije novih instanc in posodobitev obstoječih) ter brisanja. Ob proženju funkcije najprej pridobimo ustrezno plast v knjižnici za vtičnike, nato pripravimo podatke in jih pošljemo na ustrezno končno točko strežnika za vtičnike. Ker se te spremembe dogajajo pogosto, v telesu pošljemo

samo identifikator starša celice, ki se je spremenila. Mobilne aplikacije pa potem same razpoznavajo, ali je treba polja posodobiti takoj, ali pa v naslednjih akcijah uporabnikov.

```
@receiver(post_save, sender=NoteElement)
@receiver(post_save, sender=ChecklistElement)
@receiver(pre_delete, sender=NoteElement)
@receiver(pre_delete, sender=ChecklistElement)
def notify_on_element_update(sender, instance, **kwargs):
    # print(instance.parent._meta.verbose_name)
    parent_name = instance.parent._meta.verbose_name
    layer = get_channel_layer()

    # SEND TO OWNER
    build_message(parent_name, instance.parent.user.id, instance.parent.id, layer)

    if parent_name == "Checklist":
        # SEND TO SHARED USERS - CHECKLIST
        for shared_checklist in instance.parent.shared_checklist.filter(type=SharedChecklist.ACCEPTED):
            build_message(parent_name, shared_checklist.shared_user.id, shared_checklist.shared_checklist.id, layer)

    elif parent_name == "Note":
        for shared_note in instance.parent.shared_note.filter(type=SharedNote.ACCEPTED):
            build_message(parent_name, shared_note.shared_user.id, shared_note.shared_note.id, layer)


def build_message(parent_name, shared_user_id, shared_element_id, layer):
    async_to_sync(layer.group_send)('{}_{}'.format(parent_name, shared_user_id),
                                    {
                                        "message": {
                                            "parent_id": shared_element_id,
                                        },
                                        "type": "element_update"
                                    })
```

Slika 4.7: Pošiljanje podatkov v realnem času preko vtičnikov

Komponenta »dailyplan« se kreira za vsak dan nova ter ima, kot privzeto, 24 celic, vsaka celica predstavlja eno uro v dnevu. Uporabnik lahko za vsako celico določi svoje obvestilo, ki ga bo prejel kot potisno na telefon. Nastavi lahko poljubno uro in datum za prejem. Da zagotovimo kvalitetno pokrivanje tega cilja, smo definirali »cronjob«. Cron je časovni urnik za izvedbo nalog v operacijskih sistemih, ki temeljijo na Unixu. Uporabljamo jih za vzdrževanje in časovno urejanje aplikacij. Cronu določimo, na kakšen časovni interval naj se izvede (na primer vsako minuto, vsako sredo, ob treh zjutraj) in katera datoteka oziroma funkcija naj se izvede. [75] Na sliki 4.7 je prikazan cron za pošiljanje obvestil preko FCM, kjer naprej preverjamo, ali obstajajo katera še ne poslana obvestila na današnji datum in točno ta čas. V pozivnem toku najdemo FCM žeton tega uporabnika in pripravimo ustrezne podatke za pošiljanje sporočila. Ker celica lahko vsebuje ali niz znakov ali sliko, moramo pripraviti ustrezno sporočilo (slika se mora prikazovati v potisnem sporočilu). Nato pripravimo svoj proces za asinhrono pošiljanje

sporočil in izvedemo klic na FCM, ki posreduje sporočilo dalje do mobilne naprave. Tak sistem pošiljanja uporabljamo še pri pošiljanju prošnje uporabnika in pri ostalih delih, kjer je uporabniška izkušnja prejemanja potisnega sporočila dobra.

```
class CronNotifications(CronJobBase):
    RUN_EVERY_MINS = 0
    schedule = Schedule(run_every_mins=RUN_EVERY_MINS)
    code = "userprofile.crons.cron_notifications"

    def do(self):
        time = str(datetime.datetime.now().time())[0:5]
        notifications = Notification.objects.filter(sent=False, date=datetime.date.today(), time=time,
                                                    user__user_profile__can_receive_notifications=True)
        users = notifications.values_list('user', flat=True).distinct()
        devices = FCMDevice.objects.filter(user__in=users, active=True)
        for notification in notifications:
            try:
                device = devices.get(user=notification.user)
                if notification.dailyplan_cell.text is None or notification.dailyplan_cell.text is "":
                    picture_link = notification.dailyplan_cell.image.url
                    is_image = 1
                else:
                    picture_link = ""
                    is_image = 0
                p = multiprocessing.Process(target=self.notify, args=(
                    device, picture_link, notification.dailyplan_cell.text, is_image,
                    notification.dailyplan_cell.parent.date, notification.dailyplan_cell.time))
                p.start()
            except FCMDevice.DoesNotExist:
                pass

    def notify(self, device, image, text, is_image, dailyplan_date, dailyplan_cell_time):
        kwargs = {"extra_kwargs": {"mutable_content": True}}
        device.send_message(
            title="Dailyplan",
            body="Your dailyplan is coming up.",
            data={"d": dailyplan_date.isoformat(), "i": is_image, "u": image},
            **kwargs)
```

Slika 4.8: Cronjob za pošiljanje obvestil

5 IZBIRA OGRODJA IN UGOTOVITVE

Še preden začnemo izbirati ustrezna orodja za implementacijo, se je najprej potrebno do popolnosti pozanimati o funkcionalnostih. Različni problemi potrebujejo različne rešitve, sploh ko se pogovarjamo o zalednih sistemih, ki podpirajo mobilne aplikacije. Slednje imajo dostop do velikega nabora strojnih komponent, ki jih je treba vključiti v podporo. Čeprav bi si vedno želeli izbrati najboljše za nas, pa nam pogosto razpoložljivi viri tega ne omogočajo. Sijajno je, če imamo na razpolago močne strežnike in velik nabor tretjih storitev, ki prevzemajo naloge namesto nas (overjanje, skalabilnost strežnika, skalabilnost podatkovne shrambe, storitve za pošiljanje emailov in smsov, plačilni sistemi in druge), vendar pogosto temu ni tako, zato je potrebo izbrati pravo mero.

5.1 Primerjalna analiza ogrodij za zaledne sisteme

V tem poglavju bomo analizirali opisane zaledne sisteme po funkcionalnostih, ki nam jih nudijo, in po specifikaciji dejanskega projekta, ki smo ga implementirali. Vse verzije ogrodij so bile v času pisanja diplomske naloge posodobljene na najnovejšo verzijo, uporabljali smo zadnjo javno dokumentacijo.

Za analizo smo izbrali naslednje sklade ogrodij in njihove verzije:

- **Django & Django Rest Framework** (verzija 3.10.2), tabela 5.1.
- **Node.js** (verzija 10.16.2) in **Express.js** (verzija 4.17.1), tabela 5.2.
- **Firebase Cloud Firestone, Firebase Cloud Functions, Firebase Authentication, Firebase Hosting, Firebase Cloud Messaging** (uporabljena aktualna verzija v mesecu avgustu 2019, dostopna na https://console.firebase.google.com), tabela 5.3.

Pri analizi smo se osredotočili na naslednje funkcionalnosti, ki so po našem mnenju najbolj pomembne pri izbiri ogrodja:

1. **programski jezik**: kateri programski jezik je uporabljen za interakcijo z ogrodjem.
2. **Zrelost ogrodja**: čez čas ogrodja pridobijo na zrelosti in postajajo vedno bolj neprebojna.

3. **Strukturirana implementacija**: kje na spektru med fleksibilnostjo in strukturiranimi vzorci ležijo ogrodja.
4. **Podpora REST**: ključna podporna storitev pri izdelavi zalednih sistemov za mobilne aplikacije.
5. **Domorodna podpora podatkovni bazi**: ali imajo ogrodja podporo za različne tipe podatkovnih baz.
6. **Cena**: v mislih je treba vedno imeti balansiranje stroškov, pogledali si bomo cenovne pakete ogrodij, če jih le-ti imajo.
7. **Podpora skupnosti**: skupnost ohranja ogrodje pri živem in razvija knjižnice.
8. **Orodja za razvijalce**: kaj vse ogrodje ponuja razvijalcem, da jim prihrani vire.
9. **Podpora za vtičnike**: aplikacije v realnem času so dandanes standard, pogledali si bomo, kako ogrodja implementirajo WebSocket protokol.
10. **Varnost**: katere preventive nam ogrodje ponuja izven škatle.
11. **Primernost za različne projekte**: vsako ogrodje ni primerno za vse velikosti projektov. Pogledali si bomo, ali so ogrodja bolj primerna za manjše ali večje projekte.
12. **Podpora za potisna sporočila**: pri podpori za mobilne naprave moramo poskrbeti, da bomo lahko pošiljali potisna poročila. Pogledali si bomo, katera ogrodja imajo to možnost.

Tabela 5.1: Primerjalna tabela za Django Rest Framework

	Django Rest Framework
Programski jezik	Python >=3.5
Zrelost ogrodja	Prva verzija, verzija 0.1, je objavljena februarja 2011, prva verzija trenutne serije, verzija 3.0, pa decembra 2014. [76]
Strukturirana implementacija	DRF ima točno določeno strukturo in ponuja mnogo gradnikov, ki vzpodbujajo in implementirajo pripravljen MVC model. Delno k temu pripomore tudi ogrodje Django, na katerem DRF temelji.
Podpora REST	Podpira domorodno.

Domorodna podpora podatkovni bazi	Django Rest Framework vsebuje že integriran ORM, ki je neodvisen od podatkovne baze, pripete v sistem. Pripnemo lahko katerokoli bazo, za katero obstaja motor (angl. »engine«).
Cena	Brezplačno.
Podpora skupnosti	Django Rest Framework ima na githubu 15.200+ zvezdic v trenutku pisanja diplomske naloge, kar ga uvršča med najboljše tri knjižnice za python (pred njim sta le Django in Sentry, ogrodje za spremljanje padcev python aplikacij). [77]
Orodja za razvijalce	Django Rest Framework ima sam po sebi modul spletnega pregledovanja API-ja, ki avtomatsko generira interaktivne spletne strani za interakcijo z našimi končnimi točkami. Podeduje tudi admin panel za spremljanje objektov iz Djangota.
Podpora za vtičnike	Domorodno ne podpira vtičnikov, vendar lahko uporabimo komplementarno knjižnico Django Channels.
Varnost	Iz škatle dobimo protekcijo proti CSRF, injiciranju SQLa, Clickjacking, validacija zaglavja in druge. [78]
Primernost za različne projekte	Primeren je za srednje do velike projekte. Pri malih projektih, sploh zaradi zapletene konfiguracije in uvajanja, je lahko ogrodje pretirano (angl. »overkill«).
Podpora za potisna sporočila	Ne podpira domorodno.
Primer projekta	Potovalni blog, športni novinarski portal.

Tabela 5.2: Primerjalna tabela za Express.js

	Express.js
Programski jezik	Node.js (JavaScript)
Zrelost ogrodja	Trenutna verzija Express.js je bila objavljena 9. aprila 2014, trenutno je v pripravi nova, 5. verzija, ki je v stanju alpha. [79]
Strukturirana implementacija	Express se ponaša s tem, da nima mnenjske implementacije, je zelo fleksibilen in razvijalcem ne vsiljuje strukture.
Podpora REST	Podpira domorodno.

Domorodna podpora podatkovni bazi	Ne podpira domorodno. Za interakcijo z bazami se uporabljajo knjižnice.
Cena	Brezplačno.
Podpora skupnosti	Express ima na githubu slabih 3.000 zvezdic v trenutku pisanja diplomske naloge. Precej manj od pričakovanj, saj je javascript najbolj popularen programski jezik na istem portalu. [80] [40]
Orodja za razvijalce	Ne vsebuje nobenih orodij za razvijalce.
Podpora za vtičnike	Ne podpira domorodno, vendar lahko uporabljamo različne knjižnice, najpogosteje Socket.io.
Varnost	Domorodno lahko prilagajamo sejo in piškotke, vendar nam to ne pomaga, če izdelujemo REST API.
Primernost za različne projekte	Zelo primeren za majhne projekte, ker je namestitev hitra, dokumentacija jasna in ker samo orodje ne vsebuje veliko funkcionalnosti.
Podpora za potisna sporočila	Ne podpira domorodno.
Primer projekta	Najem pametnih omaric na plaži, bencinska črpalka brez osebja.

Tabela 5.3: Primerjalna tabela za Firebase

	Firebase
Programski jezik	Uradni SDK: Node.js (Javascript), Java, Python, Go, C# [81]
Zrelost ogrodja	Različna ogrodja imajo različno starost. Začelo se je s podatkovno bazo v realnem času 2011. Od pridružitve Googlu 2014, je nastalo še 17 dodatnih platform. [53]
Strukturirana implementacija	Firebase-ove platforme imajo določeno strukturno implementacijo z vsemi komponentami. Firebase Functions lahko razširimo s katerokoli javascript knjižnico.
Podpora REST	Podpira domorodno.
Domorodna podpora podatkovni bazi	Podpira interakcijo z bazo in vsebuje dve bazi v realnem času, obe ne relacijski. [52]
Cena	Freemium model. Brezplačen začetni program, ki je plačljiv v nadaljevanju. [82]

Podpora skupnosti	Firebase uporablja 1.5 milijona strani, ima podporo različnih jezikov in Googla. Vse to prispeva k precej dobri podpori skupnosti. [83]
Orodja za razvijalce	Vsebuje testna okolja, interaktivno konzolo, preko katere je možno konfigurirati sistem.
Podpora za vtičnike	Podpira domorodno.
Varnost	Ker je zadeva »serverless«, ima večino vgrajene varnosti zagotovljene od Googla.
Primernost za različne projekte	Primerno za vse vrste projektov. Namestitev oziroma konfiguracija je preprosta in hitra. K temu pripomorejo SDK-ji.
Podpora za potisna sporočila	Podpira domorodno (»Firebase Cloud Messaging«).
Primer projekta	Družbeno omrežje, sistemi za deljenje vsebin.

5.2 Rezultati

Rezultati analize prikazujejo vsako od ogrodij, po funkcionalnosti, na nekem spektru uporabnosti. Prikazujejo tudi poslovna pravila kreatorjev ogrodij. Ogrodja, kot so: Django Rest Framework in Firebaseov-e platforme so zelo strukturirane, pripravljene že imajo komponente za hiter razvoj in implementacijo v sistem, medtem ko se pri express-u veliko modulov more razviti iz nič. Čeprav se pri slednjem kompenzira veliko število knjižnic in modulov razvitih za node in express specifično. Ogromno stvari, ki jih Django Rest Framework ponuja iz škatle, lahko najdemo pri express-u v obliki knjižnic. Pod to spadajo omejevalniki zahtev, motorji za podatkovne shrambe, postopki overjanja, razčlenjevanje telesa zahteve (angl. »body parser«), razni zapisovalniki aktivnosti in tako dalje. Res pa je, da so slednji vsi razviti od skupnosti in po navadi niso uradno podprti od kreatorjev, kot je to pri DRF in Firebase-u. Pri slednjem tudi hitro naletimo na plačilno blokado, kar pa je smiselno, saj nam ponuja tudi gostovanje same aplikacije in podatkovne baze v Google okolju. Če bi našo aplikacijo spisali v DRF ali Express-u in bi jo hoteli namestiti na strežnik, kar je samoumevno, saj delamo REST API za mobilne rešitve, bi hitro prišli, seveda za isto zmogljive strežnike, čez ceno, katero nam ponuja Firebase v srednjem paketu. Prav tako bi mogli poskrbeti za konfiguracijo, za spremljanje aktivnosti in druge zadeve, ki poberejo ogromno virov tudi izkušenim »dev ops«

razvijalcem. Pri Firebase-u se zanašamo na Googlove strokovnjake, ki so preddefinirali okolja. Bilo bi neumno dvomiti o suverenosti podjetja, katerega matična družba zaseda četrto mesto na lestvici največjih tehnoloških podjetij, je samo za Microsoftom, Samsungom in Applom. [84] Negativna vrednota »serverless« pristopa je, da imamo omejeno možnost konfiguracije po želji oziroma ta sploh ne obstaja.

Pri razvoju je treba tudi razmišljati o časovnih virih; kaj vse je potrebno razviti in koliko časa bo trajalo testiranje. Pri DRF-ju se srečamo z admin panelom, ki dovoli hitre spremembe objektov v podatkovni bazi, s čimer bistveno pospeši razvoj. Sam pogrešam nekaj takega pri express--u, saj moram pri razvoju pogosto ali ustvarjati končne točke za razvijalce, ali pa posredovati direktno v bazi, nobena od teh rešitev pa ni trivialna. Prav tako pri DRF-ju uporabljamo znani ORM, razlika je samo od motorjev, ki jih priklopimo v sistem. Zelo malo razlike bi bilo, če v DRF aplikaciji uporabljamo MongoDB ali pa MySQL, PostgreSQL. V Express-u moramo zato uporabljati različne knjižnice in se priučiti specifičnega sistema rudarjenja. V Firebas-u je to že nekoliko bolj oteženo, saj nam podatkovne baze v realnem času omogočajo le preprosta iskanja po bazi. Za kaj bolj kompleksnega moramo shranjevati redundantne vrednosti, ki združujejo več polj. Prav tako ni iskanja po podobnosti (v MySQL imamo na primer operator LIKE).

Mobilne aplikacije niso samo omejene na klasični REST API za pridobivanje podatkov, vedno več specifikacij zahteva posodabljanje v realnem času in potisna sporočila. Vsa omenjena ogrodja sicer na neki način podpirajo vtičnike za duplex povezavo, ampak najpopularnejša Express knjižnica Socket.io uporablja transportni protokol po meri, ki je zgrajen na WebSocket-u. Zato je potrebno imeti tudi odjemalca s strani Socket.io, kar pa lahko povzroča težavo, če le-ta ni dosegljiv za mobilno platformo. Pogosto se to zgodi, če mobilne aplikacije gradimo prek ogrodij za sinhrono razvijanje iOS in Android aplikacij. Podobna situacija je pri Firebas-u, vendar ima slednji zelo dobre SDK-je za mobilne platforme in so taki problemi praktično neobstoječi. DRF ima rešitev, imenovano Django Channels, ki temelji na specifikaciji WebSocket-ov in se je nanj praktično mogoče povezati preko kateregakoli jezika in ogrodja. Za potisna sporočila je zadeva nekoliko drugačna. DRF in Express nimata domorodne podpore zanje. Uporabljamo lahko »Firebase Cloud Messaging« za obe platformi, ali pa omenjeno samo za Android ter

APNS (angl. »Apple Push Notification Service«) za Apple. Za karkoli se odločimo, pri obeh je situacija podobna. Poudariti moramo končno točko, preko katere se mobilna naprava lahko registrira in poveže z uporabnikom ter nam posreduje žeton. Ta žeton potem uporabljamo na omenjenih storitvah za pošiljanje potisnih sporočil. Za obe ogrodji obstajajo dobre knjižnice, imata pa tudi FCM in APNS dobre dokumentacije. Čeprav pri izbiri ogrodja velja, da lahko z vsemi zrelimi dosežemo skoraj vse, je pri izbiri pomembno, da izberemo tisto, pri katerem bomo zagotovili kvaliteten produkt pri najmanjši porabi virov. Pogosto izbira ni črno-bela situacija, ampak bomo uporabili več ogrodij in storitev, da dosežemo končni cilj. V našem projektu smo izbrali kombinacijo DRF in Firebase-a, saj nam funkcionalnosti ne narekujejo kaj več od CRUD aplikacije z overjanjem in pošiljanjem potisnih sporočil.

6 SKLEP

Skozi celotno nalogo smo si pogledali potek izdelave zalednega sistema za mobilne aplikacije. Veliko teh korakov lahko uporabimo tudi pri izdelavi sistema za druge odjemalce. Pogledali smo si, kateri sestavni deli tvorijo celotno zaledno rešitev, celotno pod zahteve, ki gre skozi različne vrste strežnikov in podatkovnih shramb, da lahko mobilnim napravam zagotovimo najhitrejšo odzivnost, ki poveča uporabniško izkušnjo. Sprehodili smo se skozi značilnosti pri razvoju za mobilne odjemalce, na kaj moramo paziti pri takem razvoju v nasprotju z razvojem za, na primer, spletne odjemalce ali celo namizne, vdelane sisteme. Izbrali smo si tri zelo različna orodja v različnih jezikih, implementacije in funkcionalnosti ter si pogledali, kaj nam ti ponujajo, saj so vsi trenutno aktualni in med najbolj pogosto uporabljenimi za razvoj REST storitev. Vključili smo tudi en zaledni sistem kot storitev, »Firebase«, ki ponuja celoten komplet pri razvoju in je, za moje izkušnje, najbolj primeren, če zadevo gradimo sami, smo edini razvijalec na projektu, tako imenovani »full-stack« razvijalec. V primeru, da sisteme gradi celotna ekipa, je praksa, da se zadeve razdelijo na zaledne sisteme in odjemalce oziroma prednje sisteme. Tako vzpodbujamo tudi pristop specializacije razvijalcev za eno stvar, pri čemer lahko prihranimo vire in ponudimo boljši izdelek. Za primer delovanja smo si izbrali projekt iz resničnega življenja in ga pripeljali v življenje skozi DRF, ki je nekje na sredini spektra med fleksibilnim in strukturiranim ogrodjem, napram Express-u in Firebase-u, respektivno. S funkcionalnostmi smo zajeli vse vidike razvoja, od preprostega CRUD sistema, do interakcije v realnem času, preko vtičnikov in potisnih sporočil.

Cilj diplomskega dela je bil, po predstavitvi razvojnega cikla, primerjati kandidate za razvoj celotne zaledne informacijske rešitve. Pogledali smo si skupne točke in drastične razlike med izbranimi ogrodji, jih primerjali ter predstavili, kateri je najbolj primeren za kateri navidezni projekt. Zastavljen cilj smo uspešno dosegli in podkrepili s preučeno literaturo in razvito rešitvijo.

6.1 Možne izboljšave in pot nadaljnje raziskave

Seveda bi bilo dobro vključiti še kakšno ogrodje, ki si spet ne deli podobnosti z ostalimi. V mislih imam predvsem ogrodja, ki se ponašajo z »enterprise ready« vmesnikom. Med

te spada paket Spring, napisan v Javi, ter .NET ekosistem, napisan v C++/C#. Oba sta tudi statično pisana programska jezika, kar pomeni, da naj bi, vsaj v teoriji, bila nekoliko hitrejša, čeprav ne moremo posplošiti tega, vsaka prevladuje v svojih kategorijah. [85] Popularen je tudi phpjev Laravel. Pri primerjavi bi lahko implementirali različne tipe sistemov z zelo specifičnimi funkcionalnostmi, v vseh ogrodjih. To bi povečalo podatkovni vzorec. Vse sisteme bi lahko uvedli na iste, strojno gledano, zmogljive naprave in tako pridobili enaka okolja za testiranje, s čimer bi izločili še eno spremenljivko, ki vpliva na analizo.

7 VIRI IN LITERATURA

[1] R. Sanghvi, „A Guide to the Back End of a Mobile App: Business,“ 29 avgust 2018. [Elektronski]. Available: https://www.business.com/articles/mobile-app-back-end/. [Poskus dostopa junij 2019].

[2] C. Wodehouse, „A Beginner’s Guide to Back-End Development: Upwork,“ [Elektronski]. Available: https://www.upwork.com/hiring/development/a-beginners-guide-to-back-end-development/. [Poskus dostopa junij 2019].

[3] S. Srivastava, „A Quick guide on Mobile App Backend Development for Busy People: Appinventiv,“ 23 januar 2018. [Elektronski]. Available: https://appinventiv.com/blog/quick-guide-mobile-app-backend-development/. [Poskus dostopa junij 2018].

[4] M. Arlitt in C. Williamson, „Internet Web Servers: Workload Characterization and Performance Implications,“ *IEEE/Acm Transactions On Networking,* Izv. 5, 1997.

[5] C. Wodehouse, „A Guide to Server Technology: Upwork,“ [Elektronski]. Available: https://www.upwork.com/hiring/development/a-guide-to-server-technology/. [Poskus dostopa junij 2019].

[6] „What is a web server?: Mozilla,“ 23 marec 2019. [Elektronski]. Available: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server. [Poskus dostopa junij 2019].

[7] D. Kegel, „The C10K problem,“ 5 februar 2014. [Elektronski]. Available: http://www.kegel.com/c10k.html. [Poskus dostopa junij 2018].

[8] A. Leslie, „NGINX vs. Apache (Pro/Con Review, Uses, & Hosting for Each): Hostingadvice,“ 11 januar 2018. [Elektronski]. Available: https://www.hostingadvice.com/how-to/nginx-vs-apache/. [Poskus dostopa junij 2019].

[9] M. Rouse, „Making the switch to Oracle Fusion Apps: Searchmicroservices,“ december 2017. [Elektronski]. Available: https://searchmicroservices.techtarget.com/definition/middleware. [Poskus dostopa junij 2019].

[10] I. Abbadi, „Middleware Services at Cloud Application Layer,“ *Advances in Computing and Communications,* 2011.

[11] N. Gall, „Origin of the term middleware,“ 2005.

[12] C. Wodehouse, „A Guide to Database Technology: Upwork,“ [Elektronski]. Available: https://www.upwork.com/hiring/data/a-guide-to-database-technology/. [Poskus dostopa junij 2018].

[13] J. Homan, „Relational vs. non-relational databases: Which one is right for you?: Pluralsight,“ 5 april 2014. [Elektronski]. Available: https://www.pluralsight.com/blog/software-development/relational-non-relational-databases. [Poskus dostopa junij 2019].

[14] Z. W. Wu, „Relational VS Non-Relational Databases: Medium,“ 3 oktober 2018. [Elektronski]. Available: https://medium.com/@zhenwu93/relational-vs-non-relational-databases-8336870da8bc. [Poskus dostopa 2019 junij].

[15] „Types of NoSQL Databases: MongoDB,“ [Elektronski]. Available: https://www.mongodb.com/scale/types-of-nosql-databases. [Poskus dostopa junij 2019].

[16] M. Turkanović, „NoSQL Podatkovne baze,“ Maribor, 2018.

[17] C. Wodehouse, „SQL vs. NoSQL Databases: What’s the Difference?,“ Upwork, [Elektronski]. Available: https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/. [Poskus dostopa avgust 2019].

[18] C. Wodehouse, „Web Development Languages 101,“ Upwork, [Elektronski]. Available: https://www.upwork.com/hiring/development/web-development-languages-101/. [Poskus dostopa junij 2019].

[19] I. Mundy, „Declarative vs Imperative Programming,“ Codeburst, 20 Februar 2017. [Elektronski]. Available: https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2. [Poskus dostopa junij 2019].

[20] K. Eliason, „Difference between object-oriented programming and procedural programming languages,“ 1 Avgust Neonbrand. [Elektronski]. Available: https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/. [Poskus dostopa junij 2019].

[21] R. Harper, „What, If Anything, Is A Declarative Language?,“ Existentialtype, 18 Julij 2013. [Elektronski]. Available: https://existentialtype.wordpress.com/2013/07/18/what-if-anything-is-a-declarative-language/. [Poskus dostopa junij 2019].

[22] „Mobile app backend services,“ Google Cloud, 13 maj 2019. [Elektronski]. Available: https://cloud.google.com/solutions/mobile/mobile-app-backend-services. [Poskus dostopa junij 2019].

[23] J. Laanstra, „Offline Data and Synchronization for a Mobile Backend as a Service system.,“ 2013.

[24] „Best Practices for Building Secure APIs,“ Medium, 30 Marec 2018. [Elektronski]. Available: https://medium.com/apis-and-digital-transformation/best-practices-for-building-secure-apis-2b4eb8071d41. [Poskus dostopa junij 2019].

[25] R. Degges, „The Ultimate Guide to Mobile API Security,“ Stormpath, 23 Marec 2015. [Elektronski]. Available: https://stormpath.com/blog/the-ultimate-guide-to-mobile-api-security. [Poskus dostopa junij 2019].

[26] „JSON Web Token,“ JWT.io, [Elektronski]. Available: https://jwt.io/. [Poskus dostopa julij 2019].

[27] „Anatomy of an HTTP Transaction,“ Node docs, [Elektronski]. Available: https://nodejs.org/es/docs/guides/anatomy-of-an-http-transaction/. [Poskus dostopa junij 2019].

[28] „HTTP request methods,“ MDN web docs, 23 Marec 2019. [Elektronski]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. [Poskus dostopa julij 2019].

[29] „HTTP response status codes,“ MDN web docs, 30 Maj 2019. [Elektronski]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. [Poskus dostopa junij 2019].

[30] M. Myles, „Why frameworks ?,“ 8thlight, 12 september 2012. [Elektronski]. Available: https://8thlight.com/blog/myles-megyesi/2012/09/12/why-frameworks.html. [Poskus dostopa julij 2019].

[31] „Why the frameworks are important in web development?,“ xcelance, 15 september 2017. [Elektronski]. Available: https://www.xcelance.com/importance-frameworks-web-development/. [Poskus dostopa julij 2019].

[32] T. Petricek, „Why frameworks are evil,“ tomasp, 3 marec 2015. [Elektronski]. Available: http://tomasp.net/blog/2015/library-frameworks/. [Poskus dostopa julij 2019].

[33] „Docs,“ Django Rest Frameworks, [Elektronski]. Available: https://www.django-rest-framework.org/. [Poskus dostopa julij 2019].

[34] „Meet Django,“ Django, [Elektronski]. Available: https://www.djangoproject.com/. [Poskus dostopa julij 2019].

[35] „What is Python? Executive Summary,“ Python, [Elektronski]. Available: https://www.python.org/doc/essays/blurb/. [Poskus dostopa julij 2019].

[36] C. Mills, „Django introduction,“ Mozilla Developers, 8 april 2019. [Elektronski]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction. [Poskus dostopa julij 2019].

[37] „Django documentation,“ Django, [Elektronski]. Available: https://docs.djangoproject.com/en/2.2/. [Poskus dostopa julij 2019].

[38] W. Vincent, „Learning Django: What Makes it Hard?,“ wsvincent, 18 oktober 2018. [Elektronski]. Available: https://wsvincent.com/learning-django/. [Poskus dostopa julij 2019].

[39] J. Robin, „Django Multiprocessing,“ Talentpair Medium, 13 september 2017. [Elektronski]. Available: https://engineering.talentpair.com/django-multiprocessing-153dbcf51dab. [Poskus dostopa julij 2019].

[40] „Info,“ GitHut, [Elektronski]. Available: https://githut.info/. [Poskus dostopa julij 2019].

[41] „Authentication,“ Django REST Framework, [Elektronski]. Available: https://www.django-rest-framework.org/api-guide/authentication/#authentication. [Poskus dostopa julij 2019].

[42] „The Browsable API,“ Django REST Framework, [Elektronski]. Available: https://www.django-rest-framework.org/topics/browsable-api/. [Poskus dostopa julij 2019].

[43] „About,“ Express, [Elektronski]. Available: https://expressjs.com/. [Poskus dostopa avgust 2019].

[44] „About,“ Node.js, [Elektronski]. Available: https://nodejs.org/en/about/. [Poskus dostopa avgust 2019].

[45] B. Terlson, B. Farias in J. Harband , „ECMAScript® 2019 Language Specification,“ Junij 2019. [Elektronski]. Available: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf. [Poskus dostopa avgust 2019].

[46] „Javascript,“ Wikipedia, 7 avgust 2019. [Elektronski]. Available: https://en.wikipedia.org/wiki/JavaScript. [Poskus dostopa avgust 2019].

[47] K. Mysliwiec, „Introduction,“ Nest.js, 2019. [Elektronski]. Available: https://docs.nestjs.com/. [Poskus dostopa avgust 2019].

[48] „Middleware Docs,“ Express, [Elektronski]. Available: https://expressjs.com/en/resources/middleware.html. [Poskus dostopa avgust 2019].

[49] C. Yang, „Express, Koa, Meteor, Sails.js: Four Frameworks Of The Apocalypse,“ toptal, 2017. [Elektronski]. Available: https://www.toptal.com/nodejs/nodejs-frameworks-comparison. [Poskus dostopa avgust 2019].

[50] P. Chandrayan, „How Node.Js Single Thread mechanism Work ? Understanding Event Loop in NodeJs,“ Codeburst - Medium, 25 november 2017. [Elektronski]. Available: https://codeburst.io/how-node-js-single-thread-mechanism-work-understanding-event-loop-in-nodejs-230f7440b0ea. [Poskus dostopa avgust 2019].

[51] „Promise,“ Mozilla developer, 10 avgust 2019. [Elektronski]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Poskus dostopa avgust 2019].

[52] „Firebase,“ Firebase Google, [Elektronski]. Available: https://firebase.google.com/. [Poskus dostopa avgust 2019].

[53] C. Esplin, „What is Firebase ?,“ Medium, 24 oktober 2016. [Elektronski]. Available: https://howtofirebase.com/what-is-firebase-fcb8614ba442. [Poskus dostopa avgust 2019].

[54] „Products,“ Firebase Google, [Elektronski]. Available: https://firebase.google.com/products/. [Poskus dostopa avgust 2019].

[55] S. Plangi, „Overview of Backend as a Service platforms,“ 12 maj 2016. [Elektronski]. Available: http://ds.cs.ut.ee/courses/course-files/Siim%20Plangi%20final%20report%20May2016-2.pdf. [Poskus dostopa avgust 2019].

[56] „Installation & Setup for REST API,“ Firebase Google, [Elektronski]. Available: https://firebase.google.com/docs/database/rest/start. [Poskus dostopa avgust 2019].

[57] „Consumers,“ Django Channels, [Elektronski]. Available: https://channels.readthedocs.io/en/latest/topics/consumers.html. [Poskus dostopa avgust 2019].

[58] „Server API,“ Socket.io Docs, [Elektronski]. Available: https://socket.io/docs/server-api/. [Poskus dostopa avgust 2019].

[59] P. Bover, „Firebase Cloud Functions: the great, the meh, and the ugly,“ FreeCodeCamp, 28 may 2018. [Elektronski]. Available: https://www.freecodecamp.org/news/firebase-cloud-functions-the-great-the-meh-and-the-ugly-c4562c6dc65d/. [Poskus dostopa avgust 2019].

[60] „Cloud Functions for Firebase,“ Firebase Google, [Elektronski]. Available: https://firebase.google.com/docs/functions. [Poskus dostopa avgust 2019].

[61] T. Yagihashi, „Learn to build a mobile backend service with Firebase and App Engine,“ Google Cloud, 30 junij 2016. [Elektronski]. Available: https://cloud.google.com/blog/products/gcp/learn-to-build-a-mobile-backend-service-with-firebase-and-app-engine. [Poskus dostopa avgust 2019].

[62] „Docs,“ Firebase Google, [Elektronski]. Available: https://firebase.google.com/docs/. [Poskus dostopa avgust 2019].

[63] J. Beswick, „Lessons from a small Firebase project.,“ Medium, 14 marec 2018. [Elektronski]. Available: https://itnext.io/lessons-from-a-long-week-with-firebase-b433ce8ee49e. [Poskus dostopa avgust 2019].

[64] „Caching Overview,“ Amazon Web Services, 2019. [Elektronski]. Available: https://aws.amazon.com/caching/. [Poskus dostopa avgust 2019].

[65] M. Tea, „A Massive Guide to Building a RESTful API for Your Mobile App,“ Savvyapps, 19 julij 2017. [Elektronski]. Available: https://savvyapps.com/blog/how-to-build-restful-api-mobile-app. [Poskus dostopa avgust 2019].

[66] C. Labs, „12 and 1 ideas how to enhance backend data security,“ Medium, 16 februar 2017. [Elektronski]. Available: https://medium.com/@cossacklabs/12-and-1-ideas-how-to-enhance-backend-data-security-4b8ceb5ccb88. [Poskus dostopa avgust 2019].

[67] „Serve dynamic content and host microservices using Firebase Hosting,“ FIrebase Google, [Elektronski]. Available: https://firebase.google.com/docs/hosting/serverless-overview. [Poskus dostopa avgust 2019].

[68] C. Richardson, „What are microservices,“ Microservices.io, [Elektronski]. Available: https://microservices.io/. [Poskus dostopa avgust 2019].

[69] M. Branko Jurič, „Zakaj je prehod na arhitekturo mikrostoritev, API-je in DevOps tako pomemben in kaj prinaša,“ src.si, 2019. [Elektronski]. Available: https://www.src.si/revija/zakaj-je-prehod-na-arhitekturo-mikrostoritev-api-je-in-devops-tako-pomemben-in-kaj-prinasa/. [Poskus dostopa avgust 2019].

[70] J. Gage, „Introduction to Microservices,“ algorithmia, 23 april 2018. [Elektronski]. Available: https://blog.algorithmia.com/introduction-to-microservices/. [Poskus dostopa avgust 2019].

[71] V. Munjal, „Why should we have separate development, testing, and production environments?,“ Linux Together, 8 julij 2018. [Elektronski]. Available: https://linuxtogether.org/why-should-we-have-separate-development-testing-and-production-environments/. [Poskus dostopa avgust 2019].

[72] „Introduction to Vagrant,“ Vagrant, [Elektronski]. Available: https://www.vagrantup.com/intro/index.html. [Poskus dostopa avgust 2019].

[73] „Gitflow Workflow,“ Atlassian, [Elektronski]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Poskus dostopa avgust 2019].

[74] „What is Distributed Denial of Service (DDoS)?,“ Nginx, [Elektronski]. Available: https://www.nginx.com/resources/glossary/distributed-denial-of-service/. [Poskus dostopa avgust 2019].

[75] „Newbie Introduction to cron,“ Unixgeeks, 30 december 1999. [Elektronski]. Available: http://www.unixgeeks.org/security/newbie/unix/cron-1.html. [Poskus dostopa avgust 2019].

[76] „django rest framework histroy,“ pypi, [Elektronski]. Available: https://pypi.org/project/djangorestframework/#history. [Poskus dostopa avgust 2019].

[77] „Python repository search,“ github, [Elektronski]. Available: https://github.com/search?l=Python&o=desc&q=django&s=stars&type=Repositories. [Poskus dostopa avgust 2019].

[78] „Security in Django,“ Django Docs, [Elektronski]. Available: https://docs.djangoproject.com/en/2.2/topics/security/. [Poskus dostopa avgust 2019].

[79] „Express.js History,“ Github, [Elektronski]. Available: https://github.com/expressjs/express/blob/master/History.md#400--2014-04-09. [Poskus dostopa avgust 2019].

[80] „Express.js,“ Github, [Elektronski]. Available: https://github.com/expressjs/expressjs.com. [Poskus dostopa avgust 2019].

[81] „Firebase Admin SDK,“ Firebase Docs, [Elektronski]. Available: https://firebase.google.com/docs/admin/setup. [Poskus dostopa avgust 2019].

[82] „Firebase Pricing,“ Firebase, [Elektronski]. Available: https://firebase.google.com/pricing. [Poskus dostopa avgust 2019].

[83] „Firebase,“ Wikipedia.org, 9 avgust 2019. [Elektronski]. Available: https://en.wikipedia.org/wiki/Firebase. [Poskus dostopa avgust 2019].

[84] J. Ponciano, „The Largest Technology Companies In 2019: Apple Reigns As Smartphones Slip And Cloud Services Thrive,“ Forbes, 15 maj 2019. [Elektronski]. Available: https://www.forbes.com/sites/jonathanponciano/2019/05/15/worlds-largest-tech-companies-2019/#3a8f8ae0734f. [Poskus dostopa avgust 2019].

[85] J. Jenson, „Performance Comparison: Java vs Node,“ tandemseven, 31 avgust 2018. [Elektronski]. Available: https://www.tandemseven.com/blog/performance-java-vs-node/. [Poskus dostopa avgust 2019].

[86] „Designing scalable backend infrastructures from scratch - Codeacademy,“ [Elektronski]. Available: https://www.codecademy.com/articles/back-end-architecture. [Poskus dostopa junij 2019].

[87] A. Chauhan, „Designing scalable backend infrastructures from scratch: Medium,“ 30 april 2017. [Elektronski]. Available: https://medium.com/@helloansh/designing-scalable-backend-infrastructures-from-scratch-af80f5767ccc. [Poskus dostopa junij 2019].

[88] S. Brooke, „Choosing the Right Backend Technology for Your Application: Medium,“ 30 april 2018. [Elektronski]. Available: https://techburst.io/choosing-the-right-backend-technology-for-your-application-137979173c37. [Poskus dostopa junij 2019].

[89] V. Srivastava, „A Quick Introduction to Back-End Mobile App Development,“ Appypie, 18 Oktober 2018. [Elektronski]. Available: https://www.appypie.com/blog/a-comprehensive-guide-to-back-end-mobile-app-development. [Poskus dostopa junij 2019].