JavaScript i moderni razvoj aplikacija

Sušanj Samolov, Raul

Sveučilište u Rijeci – Odjel za informatiku

Informacijski i komunikacijski sustavi

Raul Sušanj Samolov

JavaScript i moderni razvoj aplikacija

Diplomski rad

Mentor: izv. prof. dr. sc. Marina Ivašić-Kos

Rijeka, 20.11.2020.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 17. lipnja 2020.

Zadatak za diplomski rad

Pristupnik: **Raul Sušanj Samolov**

Naziv diplomskog rada: **JavaScript i moderni razvoj aplikacija**

Naziv diplomskog rada na engleskom jeziku: **JavaScript and modern application development**

Sadržaj zadatka:

Opisati osnovnu arhitekturu klijent-server sustava te tehnički i ekonomski aspekt razvoj modernih JavaScript aplikacija koje se mogu pokretati na raznovrsnim uređajima i različitim platformama. Proučiti i detljno opisati komponente i načina rada JavaScripta s naglaskom na prednosti i mane vezane za Javascript Engine, upravljanje memorijom te korištenje Web API-je u razvoju web aplikacije.

Opisati i usporediti razvoj poslužiteljskih i razvoj klijentskih aplikacija te postupak testiranja i podizanja sustava. Proučiti i opisati pomoćne alate i infrastrukture poput AWS-a (Amazon Web Services), GCP-a (Google Cloud Platform), Microsoft Azure-a koji su kompatibilni za korištenje u razvoju web aplikacija.

Mentor:

Izv. prof. dr. sc. Marina Ivašić-Kos

Voditeljica za diplomske radove:

Izv. prof. dr. sc. Ana Meštrović

Zadatak preuzet: 17.06.2020.

(potpis pristupnika)

Sadržaj

Sažetak

Zadatak ovog diplomskog rada je detaljan opis razvoja modernih aplikacija koje su napisane u JavaScript programskom jeziku i imaju za cilj da se mogu pokretati na raznovrsnim uređajima i platformama korištenjem programskog jezika JavaScript. Diplomski rad se bavi opisivanjem tehničkog i ekonomskog dijela razvoja aplikacija s detaljnim opisivanjem načina rada JavaScripta s naglaskom na njegove prednosti i mane te pomoćne alate koji su kompatibilni za korištenje.

Ključne riječi

JavaScript, aplikacija, programski jezik, razvoj, programer, inženjer

Uvod

Razvoj modernih aplikacija je danas jedno od najprofitabilnijih poslova na svjetskom tržištu. Gotovo svaka tvrtka koja pruža neke servise ili proizvode posjeduje neku vrstu web aplikacije poput online trgovine ili mobilnu aplikaciju za program lojalnosti ili neku drugu svrhu. Razvoj aplikacija ne zahtijeva nikakvu skupu opremu ili sirovinu, već samo znanje programiranja, neku vrstu računala s operacijskim sustavom i poslužitelja za podizanje sustava. Ipak za razvojne agencije postoji jedan dodatan trošak, ako izuzmemo režije to je plaćanje stručnog kadra za razvijanje takvih aplikacija. Neovisno o troškovima razvoja i s obzirom na obujam posla na tržištu profitabilnost nije upitna. Nabavka opreme, plaćanje poslužitelja i režija nije problem, ali pronalazak stručnog kadra može ponekad biti poprilično velik problem. Razlozi za to su raznovrsni, od manjka kadra na određenim lokacijama, preniske ponude plaća i benefita u usporedbi s konkurentnim agencijama i sl. Svakako se cijeni znanje programiranja i rješavanje kompleksnih logičkih problema, ali ono što se trenutno na tržištu još više cijeni je poznavanje JavaScript programskog jezika. JavaScript je danas jedan od najpopularnijih programskih jezika koji ima veliki broj korisnika, ne samo programera, već i tvrtki koje razvijaju aplikacije. JavaScript nam je omogućio razvoj na širokom spektru uređaja i platforma, a samim time je povećao vrijednost onih ljudi koji ga razumio i znaju koristiti. Tako npr. JavaScript možemo koristiti u mobilnim aplikacijama, što je velika prednost jer danas gotovo sve što želimo možemo odraditi pomoću pametnih telefona i aplikacija koje su instalirane na njima. Do naručivanje hrane, odjeće i obuće, kozmetike, taksi službe, plaćanja računa i razgovora putem video poziva s prijateljima nas danas dijeli svega nekoliko dodira na ekran.

U sklopu ovog rada biti će prikazan proces razvoja moderne aplikacije pisane u JavaScript programskom jeziku od prvog razgovora s klijentom do pisanja završne dokumentacije projekta, a osim samog procesa biti će prikazane funkcionalnosti koje omogućava JavaScript i na koji način programski jezik funkcionira u pozadini, koje su mu prednosti i mane te na koji način se može olakšati i ubrzati razvoj. Osim JavaScripta i procesa razvoja modernih aplikacija, pojasnit ćemo ključne pojmove i razlike između klijetnskih i poslužiteljskih aplikacija. Ukratko klijentske aplikacije laički možemo opisati kao ono što krajnji korisnik vidi na ekranu tj. aplikacija koja se pokreće u web pregledniku ili u obliku aplikacije na nekom uređaju, a poslužiteljske aplikacije su one koje krajnji korisnik ne vidi, već se one pokreću u pozadini na udaljenim poslužiteljima i

brinu o poslovnoj logici i komunikaciji sa bazom podataka. Detaljnije ćemo opisati svaku zasebno u nadolazećim poglavljima.

Povijest JavaScripta "od nastanka do danas"

JavaScript[1] je razvijen 1995. godine od strane programera Brendan Eicha koji je radio za Netscape Communications[2], tvrtka koja je tada imala monopol nad tržištem web preglednika. Eich je kreirao prvu verziju programskog jezika u svega desetak dana, a inspiraciju za strukturu i specifikaciju jezika dobio je od Java programskog jezika. Sredinom 1995. godine točnije u kolovozu, Microsoft započinje preuzimanje tržišta web preglednika sa svojim web preglednikom pod nazivom Internet Explorer. U to vrijeme Netscape Communications i Microsoft postaju glavni suparnici u utrci za najkvalitetnijim i najbržim web preglednikom. Kako je Microsoftov Internet Explorer započeo sa sustizanjem Netscape platforme, Netscape započinje sprječavati to na dva načina. Prvo su krenuli s procesom standardizacije JavaScript kako Microsoft ne bi preuzeo kontrolu nad jezikom. Standardizacija se nazvala ECMAScript te služi i danas u svrhu generalnog programskog standarda za razvoj jezika poput JavaScripta te ga propisuje organizacija Ecma international (European Computer Manufacturers Association)[3]. Microsoftu je JavaScript bio interesantan jer je tada to bilo već gotovo rješenje za web preglednike koje je provjereno radilo. Osim procesa standardizacije Netscape ulazi u partnerstvo s tvrtkom Tvrtka Sun koja je od 1990. godine razvijala i promovirala Java programski jezik kao najbolji alat za softverska rješenja pametnih kućanskih aparata, a kasnije kao najbolje rješenje za web platformu, dijelila je zajednički interes s Netscapeom da spriječi Microsoftov monopol. Sun je pomoću partnerstva s Netscapeom imao benefit promocije svojeg programskog jezika u kombinaciji s još uvijek najpopularnijim web preglednikom, dok je Netscape imao benefit promocije JavaScripta kao službenog kompatibilnog programskog jezika s Java programskim jezikom, drugim riječima JavaScript je dobivao na popularnosti zbog postojećeg i popularnog jezika. Iz istog razloga je JavaScript i preimenovan iz originalnog naziva Mocha[4] u JavaScript [1].

[1] Ben Aston, "A brief history of JavaScript", https://medium.com/@_benaston/lesson-1a-the-history-of-javascript-8c1ce3bffb17 (pristupljeno 08.07.2020.).
[2] https://en.wikipedia.org/wiki/Netscape (pristupljeno 08.07.2020.).
[3] https://www.ecma-international.org/ (pristupljeno 08.07.2020.).
[4] Ben Aston, "A brief history of JavaScript", https://medium.com/@_benaston/lesson-1a-the-history-of-javascript-8c1ce3bffb17 (pristupljeno 08.07.2020.).

Netscape je nudio i svoj proizvod pod nazivom LiveWire[5] koji je omogućavao razvoj na klijentskoj i poslužiteljskoj strani sustava pisan u istom programskom jeziku, takvu funkcionalnost je i Sun razvijao s Javom. S obzirom na limitiranost tadašnjeg weba u usporedbi s kompleksnom Javom, Sun nije gledao na JavaScript kao stvarnu prijetnju te se partnerstvo između te dvije tvrtke i dalje nastavilo. Iako se počelo "čuti" za JavaScript i dalje je bio jedan od nepoželjnih jezika za programere. Kratko vrijeme inicijalnog razvoja jezika, slaba promocija, novi i prvi put viđen dizajn programskog jezika i generalno slab interes za razvoj sustava za web preglednik, rangirali su JavaScript daleko niže od ostalih programskih jezika na ljestvici. Averzija i ismijavanje prema jeziku nije postojala samo od strane programera već i od njegovog začetnika i kreatora Brendan Eicha koji je priznao da mu je žao što se odlučio na neke elemente i dizajn programskog jezika.

Kroz godine JavaScript više nije bio u središtu pozornosti sve dok se nije pojavio sintaktički podskup samog jezika pod nazivom JSON (JavaScript Object Notation[6]) kojeg je razvio Douglas Crockford te je samim time osvježio interes za JavaScriptom. Web je postajao sve veći i interesantniji populaciji, a samim time je bila sve veća potreba za razvojem web platforma tj. web stranica i programera koji će razvijati u JavaScriptu. U 2000-tim godinama popularnost i korištenje JavaScripta postaje sve veća, a jedan od svojih vrhunaca dostiže 2006. godine kada se pojavljuje jedna od najpopularnijih biblioteka za JavaScript, a to je jQuery[7], biblioteka koja je pomogla velikom broju programera da razumije do tada slabo shvaćen i proučen programski jezik.

Potencijal ovog programskog jezika su primijetile i druge vodeće tvrtke te su pokušale razvijati svoje podskupe i supersetove jezika, međutim većina ih nije dospjela do većeg broja programera osim Typescript[8] kojeg je razvio Microsoft te popularizira tako da ga je postavio kao softver otvorenog koda i time pridobio brojne interesente za supersetom JavaScripta koji se danas koristi u velikom broju mobilnih i web aplikacija, ali opet ne koliko i sam JavaScript. Najveći porast korištenja JavaScripta se pojavljuje 2009. godine kada je objavljena peta verzija ECMAScripta, a najveće izmjene i nadopune se događaju 2015. godine kada izlazi šesta verzija ECMAScripta

[5] https://philip.greenspun.com/wtr/livewire.html
[6] https://www.json.org/json-en.html
[7] https://jquery.com/
[8] https://www.typescriptlang.org/

poznatija kao ES6 (te pod novim nazivljem ECMAScript 2015). Kako se standard nije izmjenjivao gotovo šest godina ECMA international odlučuje u buduće objavljivati svake godine novu verziju ECMAScripta kako ne bi opet došlo do velikog broja izmjena i nadopuna koja je otežala učenje postojećim, ali i novim JavaScript programerima.

Danas JavaScript pod standardom ECMAScript 2020 (ES11) stoji kao jedan od najpopularnijih i najkorištenijih programskih jezika u kojem su napisane brojne moderne web i mobilne aplikacije, te se svake godine nadograđuje s novim funkcionalnostima i unaprjeđuje dizajn.

Mogućnosti JavaScripta

JavaScript se godinama tretirao kao skriptni jezik namijenjen za razvoj klijentskih aplikacija za okruženje web preglednika, međutim kroz vrijeme se njegova primjena proširila i na razvoj poslužiteljskih aplikacija te na veći broj okruženja i uređaja.

Istina je da se i danas većinom JavaScript koristi za razvoj klijentskih aplikacija, ali se sve veći broj IT tvrtki i njihovi razvojni inženjeri odlučuju za JavaScript kao jezik za razvoj poslužiteljskih servisa poput API-a (Application Programming Interface), CRON poslova i sl. Nakon Netscapeovog programa Live Wire koji je pružao mogućnost pisanja JavaScripta na poslužiteljskoj strani softvera, 2009. godine se pojavljuje Node.js[9] tzv. JavaScript runtime koji omogućava isto, ali dobiva na sve većoj popularnosti te se danas tretira kao glavno okruženje za pisanje JavaScripta bez potrebe web preglednika. Nakon popularizacije i sve većem broju napisanih servisa u Node.js-u 2010. godine se pojavljuje i upravitelj paketima pod nazivom npm koji je olakšao programerima objavljivanje i preuzimanje dijelova otvorenog koda za Node.js.
Danas npm broji preko 350.000 paketa otvorenog koda te olakšava razvoj klijentskih i poslužiteljskih aplikacija[10].

[9] https://nodejs.org/en/about/
[10] https://www.npmjs.com/

Mogućnosti razvoja za različite platforme

Kada govorimo o JavaScriptu u kontekstu klijentskih aplikacija nailazimo na izazov višeplatformskih rješenja tj. aplikacija koje je moguće pokretati i koristiti na više različitih platformi, većinom misleći na različite operacijske sustave. Moglo bi se reći da je JavaScript preuzeo slogan programskog jezika Java "Write once run everywhere" te je do sada uspješno pokrio gotovo sve popularne operacijske sustave. Tako danas možemo vidjeti aplikacije pisane u JavaScriptu koje se pokreću na Windowsu, Linuxu, Mac OS-u, Androidu, iOS-u, Symphony OS-u (*Slika 1*) i drugim nešto manje popularnim sustavima. Većinom je to postignuto tako da se kreirao tzv. JavaScript most (JavaScript bridge) koji služi za kompajliranje JavaScript koda u izvorni kod platforme. Tako npr. imamo kompajliranje u Java programski jezik za Android sustav i u Objective-C za iOS sustav.

Osim raznolikih operacijskih sustava JavaScript je kompatibilan i sa svim web preglednicima, a to je zahvaljujući ECMAScript standardizaciji. Nakon svake nove funkcionalnosti objavljene od strane ECMA organizacije, vlasnici web preglednika moraju implementirati te iste mogućnosti u svoje preglednike. Neki web preglednici poput Googleovog Chrome-a i Mozillinog Firefoxa podržavaju u samom početku veliki dio mogućnosti dok drugi web preglednici znaju često kasniti s implementacijom te je tada od strane razvojnih inženjera aplikacija za navedene preglednike potrebna dodatna pažnja i implementacija iste funkcionalnosti u aplikaciji na više načina tj. Potrebno je pisanje koda s naznakom za svaki pojedini web preglednik zasebno.

Osim navedenih web preglednika, računalnih i mobilnih operacijskih sustava, JavaScript postaje sve uspješniji i u području operacijskih sustava za nosivu tehnologiju poput pametnih satova i narukvica, operacijskih sustava za televizije, robotiku i sl.

Mogućnosti razvoja za različite uređaje

Iako je JavaScript vezan uz razvoj aplikacija koji se pokreću na nekom web pregledniku ili operacijskom sustavu, veliki napredak je taj programski jezik doživio i na području različitih uređaja. Kako je prije navedeno JavaScript aplikacije danas možemo pronaći i na mobilnim uređajima te nam takve aplikacije ne pružaju samo posebno dizajnirano sučelje za takve uređaje, već nas opskrbljuju s posebnim funkcionalnostima vezane uz sami uređaj.

Tako možemo pronaći mnogo aplikacija koje koriste većinu senzorike pametnih mobitela kao što su akcelerometar, žiroskop, magnetometar, senzor ambijentalne svjetlosti te još neke module poput NFC-a, Bluetooth-a i sl. Komunikaciju JavaScript kodne baze sa senzorikom i modulima pametnog mobitela zahvaljujemo tzv. SDK-ovima (Software Development Kit) koji su razvijeni od strane proizvođača operacijskih sustava i posebnih skupina programera koji zbog svojih potreba razvijaju SDK-ove. Često možemo pronaći gotova rješenja za korištenje takvih modula u samim razvojnim okruženjima (Framework) JavaScripta kao npr. React Native[11] koji služi primarno za razvoj mobilnih aplikacija u JavaScriptu te već posjeduje u sebi izvorne module poput modula za kameru, NFC i sl.

Omogućavanje korištenja punog potencijala operacijskog sustava i hardvera uređaja nam danas omogućava funkcionalnosti poput otključavanja zaslona pomoću snimanja očne mrežnice kamerom pametnog telefona ili pomoću skeniranja otiska prsta, plaćanje računa na blagajni naslanjanjem mobitela na POS uređaje, mjerenje veličine prostorije, pa čak i zabavni sadržaj poput virtualne stvarnosti i igranje mobilnih video igara pomicanjem uređaja u različite smjerove. Sve ove funkcionalnosti se mogu razvijati i u JavaScript programskom jeziku.

Slika 1. Prikaz podržanih platforma i operacijskih sustava

[11] https://reactnative.dev/ (pristupljeno 28.08.2020.)

Planiranje i modeliranje aplikacije

Najvažniji dio bilo kakvog rada je planiranje tog istog rada, temeljito raspisivanje ideja, motivacije, procesa i cilja kako bi se na kraju došlo do željenih rezultata. Isto vrijedi svakako i za razvoj modernih aplikacija bilo koje vrste, a osobito se treba dati naglasak na razvoj poslovnih aplikacija koje znaju ponekad sadržavati visoku razinu kompleksnosti u poslovnoj logici.
U razvoj aplikacija dodajemo još jedan element, a to je modeliranje aplikacija koje proizlazi od samog plana razvoja. U ovom poglavlju bit će detaljno opisan proces planiranja i modeliranja modernih web aplikacija i to iz stajališta industrije i kako se to uistinu odvija. U literaturi se često može pronaći veliki broj koraka za ove procese, međutim u poslovnom svijetu rijetko se koja tvrtka drži svih koraka, a ako to i radi često su ti koraci izmijenjeni i prilagođeni samoj tvrtki. Razlozi izmjena su često manjak ili višak radne snage, vremensko ograničenje ili klijent koji jednostavno zahtjeva da se taj dio razvoja što brže odradi ili da se uopće ne odradi.

Koraci planiranja i modeliranja:

- Razgovor s klijentima
- Razgovor s voditeljom projekta
- Pisanje poslovne specifikacije
- Pisanje tehničke specifikacije
- Procjena projekta
- Modeliranje baze podataka
- Struktura projekta
- Okruženje i infrastruktura projekta

Razgovor s klijentima

Iako je razgovor s klijentima nešto što više obilježava prodajni sektor tvrtke često možemo u industriji vidjeti da sudjeluje i razvojni inženjer koji posjeduje višegodišnje iskustvo. Takve inženjere se najčešće poziva tek na drugi ili treći razgovor s klijentom kako bi se uvrstilo njegovo ili njezino tehničko znanje u sami proces rasprave o željama i zahtjevima klijenta. Uloga

razvojnog inženjera u takvom razgovoru je postavljanje odgovarajućih pitanja i prikupljanje što većeg broja inicijalnih informacija pomoću kojih se kasnije unutar tvrtke i razvojnog tima raspravlja o mogućnostima i opcijama razvoja potencijalne aplikacije. Osim prikupljanja informacija, inženjer stoji klijentu na raspolaganju za odgovaranje na pitanja koja on ili ona može odgovoriti u tom trenutku i koja su vezana uz tehničke detalje projekta. Inženjer ne bi trebao odgovarati na pitanja poput vremenskog perioda izrade ili potencijalnih troškova razvoja i održavanja sustava, taj dio pregovaraju prodajni predstavnici tvrtke. Jako često se događa da klijent ima maksimalno dva pitanja za inženjera o izvedivosti neke specifične funkcionalnosti, dok najčešće klijent ne upućuje nikakva pitanja inženjeru zbog manjka znanja u tom području. Scenariji je znatno drugačiji radi li se o tehnički obrazovanom klijentu ili klijentu iz područja informacijskih tehnologija, tada su često upućena pitanja inženjeru o izboru tehnologija, arhitekturi i sl. segmentima. Ako inženjer u tom trenutku posjeduje odgovor koji može izjaviti sa sigurnošću onda to i čini, međutim ako postoji bilo kakva indicija na nesigurnost u odgovoru tada se predlaže da inženjer naknadno odgovori na to pitanje.
Nakon svakog razgovora tj. sastanka s klijentima potrebno je poslati email koji sadrži sažetak sastanka u pisanom obliku kako ne bi došlo do nesporazuma naknadno. Nužno je dodati u primatelje i razvojnog inženjera koji je sudjelovao u razgovoru, te eventualno druge razvojne inženjere ako za to ima potrebe kako bi se brže dijelile informacije koje su skupljene tijekom razgovora.

Razgovor s voditeljom projekta

U razgovoru s klijentom gotovo uvijek sudjeluje jedan voditelj projekta, najčešće onaj koji će i voditi projekt o kojem se i vodio sastanak. Nakon razgovora s klijentom slijedi razgovor između voditelja projekta, inženjera koji je sudjelovao na prethodnom sastanku i eventualno voditeljem inženjera ako se ne radi o istoj osobi. Na takvom sastanku se komentira i raspravlja o prethodnom sastanku s klijentom, te inženjer koji je zadužen za projekt dijeli s voditeljem projekta sve moguće rizike projekta, raspravlja se o potrebnim izmjenama i dogovara sljedeći sastanak između inženjera i voditelja projekta kako bi se odgovorila još neka pitanja koja zahtijevaju istraživanje i vrijeme za odgovor. Inženjer daje upute voditelju projekta što bi se trebalo komunicirati s klijentom te traži od voditelja projekta neke dodatne informacije i resurse do kojih ima pravo doći samo voditelj projekta. Često su to dodatne informacije o projektu,

postojećoj infrastrukturi, dodatnim potrebama klijenta, prijedlozi klijentu o izmjenama i nadopunama samog projekta zbog poboljšanja efikasnosti ili sprječavanja potencijalnih rizika. Razgovori tj. sastanci između voditelja projekta i razvojnog inženjera su česti i ovise i prirodi projekta i klijenta. Takvi sastanci su glavna indirektna poveznica komunikacije između inženjera i klijenta, a potrebna je kako bi se zaštitilo inženjera od raznovrsnih rasprava s klijentom te da se na takav način inženjer može fokusirati na svoj dio posla, a to je razvoj, dok se voditelj projekta brine o projektu i njegovom vlasniku tj. klijentu.

Razgovori između klijenta i voditelja projekta te razgovori između inženjera i voditelja projekta su dio procesa koji se odvija tijekom cijelog životnog vijeka razvoja aplikacije te ne pripada samo procesu planiranja i modeliranja aplikacije, ali je zato jedan od prvih i najvažnijih segmenata za započeti razvoj aplikacije.

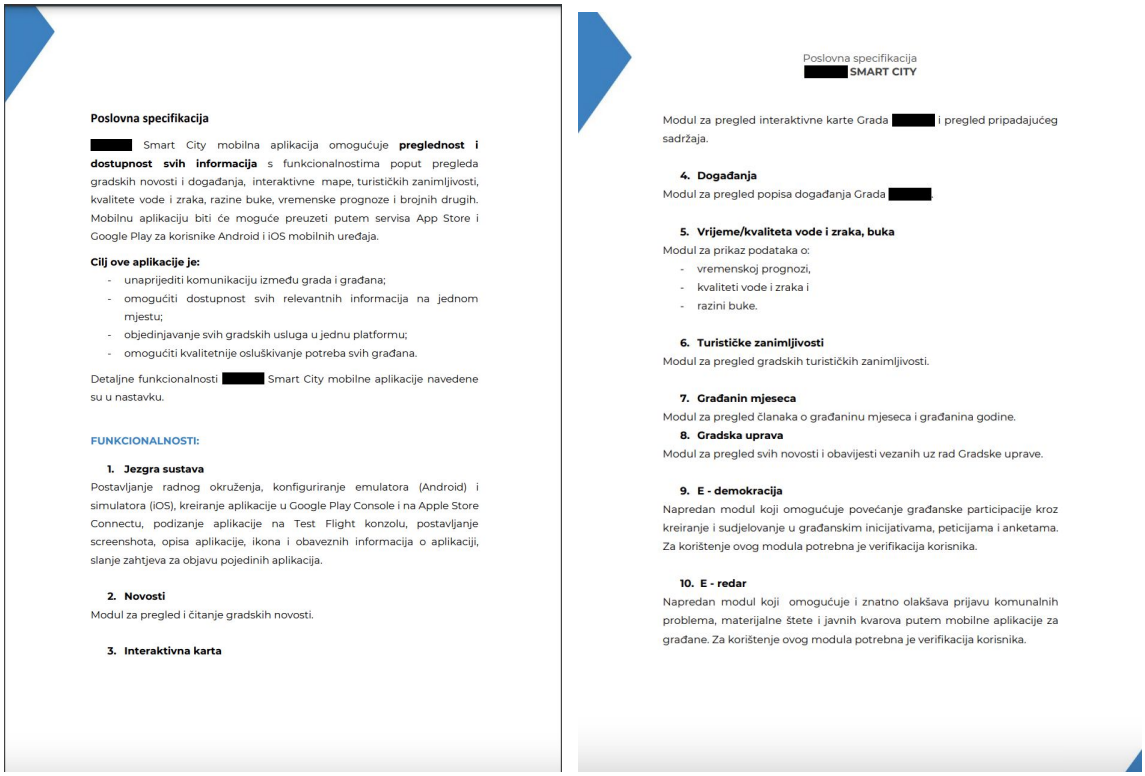
Specifikacije

Prije onih konkretnijih procesa u razvoju aplikacija, potrebno je napraviti specifikaciju aplikacije tj. konkretnije cijeloga projekta. Specifikacija detaljno opisuje kako će se aplikacija razvijati, što će se koristiti za razvoj od resursa, koje su funkcionalnosti aplikacije, kako će se raščlaniti aplikacija u manje module i sl. Svaki od tih dijelova specifikacije se nalazi u zasebnim vrstama specifikacija, a najčešće ih u industriji nazivamo poslovna specifikacija i tehnička specifikacija.

Poslovna specifikacija

Poslovna specifikacija (*Slika 2.*) opisuje poslovnu logiku i funkcionalnosti aplikacije na jedan jednostavan i općeniti način kako bi bio jasan svima koji ga čitaju, a da su upoznati sa strukom poslovne logike ili uključeni na bilo koji način u razvoj aplikacije. U poslovnoj specifikaciji se izbjegavaju stručni termini nepoznati klijentu te se ne spominju detaljni tehnički procesi. Poslovna specifikacija opisuje tok aplikacije s najvažnijim segmentima i poslovne procese klijentove organizacije tako npr. možemo pronaći u poslovnoj specifikaciji strukturu organizacije, poslovne procese svakog odijela organizacije i veze između odijela. Poslovna specifikacije služi više razvojnom timu nego klijentu jer opisuje način na koji organizacija

funkcionira, probleme koje ta organizacija ima i način na koji ih želi riješiti korištenjem aplikacije tj. ugovorenog digitalnog sustava. Isto tako, poslovna specifikacija na neki način služi kao dokument putem kojeg se obje strane, dakle klijent i razvojni tim mogu osigurati da se izvrši samo ono što je dogovoreno. Svakako u samom početku dok još traju i inicijalni sastanci, poslovna specifikacija se može po potrebi izmjenjivati ako je klijent zaboravio napomenuti nešto važno vezano uz organizaciju ili se odlučio za neku izmjenu ili nadopunu u željenom sustavu. Krajnji dokument koji ugovara posao je na kraju ovjereni ugovor između klijenta i razvojne agencije.



Poslovna specifikacija

[redacted] Smart City mobilna aplikacija omogućuje **preglednost i dostupnost svih informacija** s funkcionalnostima poput pregleda gradskih novosti i događanja, interaktivne mape, turističkih zanimljivosti, kvalitete vode i zraka, razine buke, vremenske prognoze i brojnih drugih. Mobilnu aplikaciju biti će moguće preuzeti putem servisa App Store i Google Play za korisnike Android i iOS mobilnih uređaja.

Cilj ove aplikacije je:

- unaprijediti komunikaciju između grada i građana;
- omogućiti dostupnost svih relevantnih informacija na jednom mjestu;
- objedinjavanje svih gradskih usluga u jednu platformu;
- omogućiti kvalitetnije osluškivanje potreba svih građana.

Detaljne funkcionalnosti [redacted] Smart City mobilne aplikacije navedene su u nastavku.

FUNKCIONALNOSTI:

1. **Jezgra sustava**

Postavljanje radnog okruženja, konfiguriranje emulatora (Android) i simulatora (iOS), kreiranje aplikacije u Google Play Console i na Apple Store Connectu, podizanje aplikacije na Test Flight konzolu, postavljanje screenshota, opisa aplikacije, ikona i obaveznih informacija o aplikaciji, slanje zahtjeva za objavu pojedinih aplikacija.

2. **Novosti**

Modul za pregled i čitanje gradskih novosti.

3. **Interaktivna karta**

Poslovna specifikacija
[redacted] SMART CITY

Modul za pregled interaktivne karte Grada [redacted] i pregled pripadajućeg sadržaja.

4. **Događanja**

Modul za pregled popisa događanja Grada [redacted].

5. **Vrijeme/kvaliteta vode i zraka, buka**

Modul za prikaz podataka o:

- vremenskoj prognozi,
- kvaliteti vode i zraka i
- razini buke.

6. **Turističke zanimljivosti**

Modul za pregled gradskih turističkih zanimljivosti.

7. **Građanin mjeseca**

Modul za pregled članaka o građaninu mjeseca i građanina godine.

8. **Gradska uprava**

Modul za pregled svih novosti i obavijesti vezanih uz rad Gradske uprave.

9. **E - demokracija**

Napredan modul koji omogućuje povećanje građanske participacije kroz kreiranje i sudjelovanje u građanskim inicijativama, peticijama i anketama. Za korištenje ovog modula potrebna je verifikacija korisnika.

10. **E - redar**

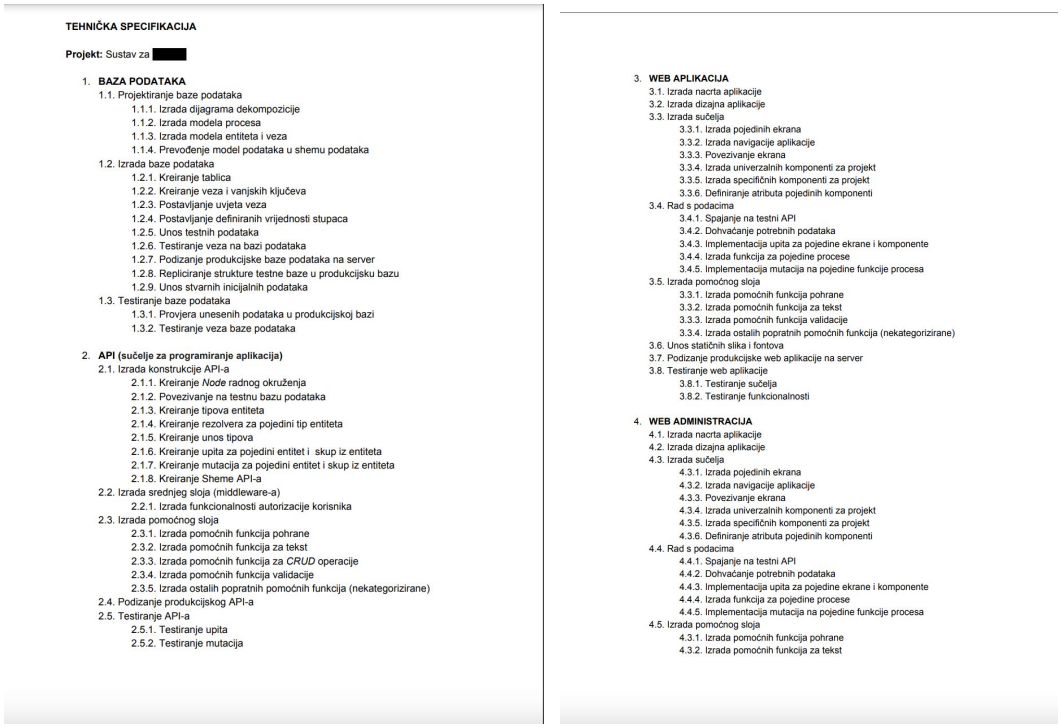
Napredan modul koji omogućuje i znatno olakšava prijavu komunalnih problema, materijalne štete i javnih kvarova putem mobilne aplikacije za građane. Za korištenje ovog modula potrebna je verifikacija korisnika.

Slika 2. Prikaz poslovne dokumentacije

Tehnička specifikacija

Tehnička specifikacija (*Slika 3*) se znatno razlikuje od poslovne i služi klijentu i razvojnom timu kao referenca za detaljan opis funkcionalnosti i tehničkih specifičnosti projekta.

Tehnička specifikacija ima tri verzije, prva ona koju klijent dobiva uz poslovnu specifikaciju te koja sadrži odabir tehnologija u kojima će se razvijati sustav, opis i plan arhitekture sustava, detaljan opis glavnih funkcionalnosti, bazna skica sustava i izgled aplikacije te raščlambu razvoja aplikacije u više manjih modula. Druga verzija tehničke specifikacije je ona koju klijent dobiva pri završetku osnovne izrade sustava, a ona sadrži sve što i prva verzija, ali ju se nadopunjuje sa završnim skicama glavnog dizajna, modelom procesa informacijskog sustava, modelom entiteta i veza sustava i modelom baze podataka. Treća verzija specifikacije je opcionalna i ovisi o daljnjem razvoju sustava, a to je tehnička specifikacija s dopunama i izmjenama te se nju piše ako se sustav nadopunjuje funkcionalnostima ili ako dođe do većih izmjena. Nadopune i izmjene moraju značajno utjecati na strukturu projekta i same aplikacije kako bi se uvrstile u tehničku specifikaciju.



TEHNIČKA SPECIFIKACIJA

Projekt: Sustav za [redacted]

1. **BAZA PODATAKA**
   - 1.1. Projektiranje baze podataka
     - 1.1.1. Izrada dijagrama dekompozicije
     - 1.1.2. Izrada modela procesa
     - 1.1.3. Izrada modela entiteta i veza
     - 1.1.4. Prevođenje model podataka u shemu podataka
   - 1.2. Izrada baze podataka
     - 1.2.1. Kreiranje tablica
     - 1.2.2. Kreiranje veza i vanjskih ključeva
     - 1.2.3. Postavljanje uvjeta veza
     - 1.2.4. Postavljanje definiranih vrijednosti stupaca
     - 1.2.5. Unos testnih podataka
     - 1.2.6. Testiranje veza na bazi podataka
     - 1.2.7. Podizanje produkcijske baze podataka na server
     - 1.2.8. Repliciranje strukture testne baze u produkcijsku bazu
     - 1.2.9. Unos stvarnih inicijalnih podataka
   - 1.3. Testiranje baze podataka
     - 1.3.1. Provjera unesenih podataka u produkcijskoj bazi
     - 1.3.2. Testiranje veza baze podataka
2. **API (sučelje za programiranje aplikacija)**
   - 2.1. Izrada konstrukcije API-a
     - 2.1.1. Kreiranje *Node* radnog okruženja
     - 2.1.2. Povezivanje na testnu bazu podataka
     - 2.1.3. Kreiranje tipova entiteta
     - 2.1.4. Kreiranje rezolvera za pojedini tip entiteta
     - 2.1.5. Kreiranje unos tipova
     - 2.1.6. Kreiranje upita za pojedini entitet i skup iz entiteta
     - 2.1.7. Kreiranje mutacija za pojedini entitet i skup iz entiteta
     - 2.1.8. Kreiranje Sheme API-a
   - 2.2. Izrada srednjeg sloja (middleware-a)
     - 2.2.1. Izrada funkcionalnosti autorizacije korisnika
   - 2.3. Izrada pomoćnog sloja
     - 2.3.1. Izrada pomoćnih funkcija pohrane
     - 2.3.2. Izrada pomoćnih funkcija za tekst
     - 2.3.3. Izrada pomoćnih funkcija za *CRUD* operacije
     - 2.3.4. Izrada pomoćnih funkcija validacije
     - 2.3.5. Izrada ostalih popratnih pomoćnih funkcija (nekategorizirane)
   - 2.4. Podizanje produkcijskog API-a
   - 2.5. Testiranje API-a
     - 2.5.1. Testiranje upita
     - 2.5.2. Testiranje mutacija
3. **WEB APLIKACIJA**
   - 3.1. Izrada nacrta aplikacije
   - 3.2. Izrada dizajna aplikacije
   - 3.3. Izrada sučelja
     - 3.3.1. Izrada pojedinih ekrana
     - 3.3.2. Izrada navigacije aplikacije
     - 3.3.3. Povezivanje ekrana
     - 3.3.4. Izrada univerzalnih komponenti za projekt
     - 3.3.5. Izrada specifičnih komponenti za projekt
     - 3.3.6. Definiranje atributa pojedinih komponenti
   - 3.4. Rad s podacima
     - 3.4.1. Spajanje na testni API
     - 3.4.2. Dohvaćanje potrebnih podataka
     - 3.4.3. Implementacija upita za pojedine ekrane i komponente
     - 3.4.4. Izrada funkcija za pojedine procese
     - 3.4.5. Implementacija mutacija na pojedine funkcije procesa
   - 3.5. Izrada pomoćnog sloja
     - 3.3.1. Izrada pomoćnih funkcija pohrane
     - 3.3.2. Izrada pomoćnih funkcija za tekst
     - 3.3.3. Izrada pomoćnih funkcija validacije
     - 3.3.4. Izrada ostalih popratnih pomoćnih funkcija (nekategorizirane)
   - 3.6. Unos statičnih slika i fontova
   - 3.7. Podizanje produkcijske web aplikacije na server
   - 3.8. Testiranje web aplikacije
     - 3.8.1. Testiranje sučelja
     - 3.8.2. Testiranje funkcionalnosti
4. **WEB ADMINISTRACIJA**
   - 4.1. Izrada nacrta aplikacije
   - 4.2. Izrada dizajna aplikacije
   - 4.3. Izrada sučelja
     - 4.3.1. Izrada pojedinih ekrana
     - 4.3.2. Izrada navigacije aplikacije
     - 4.3.3. Povezivanje ekrana
     - 4.3.4. Izrada univerzalnih komponenti za projekt
     - 4.3.5. Izrada specifičnih komponenti za projekt
     - 4.3.6. Definiranje atributa pojedinih komponenti
   - 4.4. Rad s podacima
     - 4.4.1. Spajanje na testni API
     - 4.4.2. Dohvaćanje potrebnih podataka
     - 4.4.3. Implementacija upita za pojedine ekrane i komponente
     - 4.4.4. Izrada funkcija za pojedine procese
     - 4.4.5. Implementacija mutacija na pojedine funkcije procesa
   - 4.5. Izrada pomoćnog sloja
     - 4.3.1. Izrada pomoćnih funkcija pohrane
     - 4.3.2. Izrada pomoćnih funkcija za tekst

Slika 3. Primjer tehničke specifikacije

Procjena projekta

Procjena projekta je istovremeno proces razvoja aplikacija i dokument koji detaljno opisuje sve prije navedene stavke izrade iz specifikacija u obliku stavke, vremena izrade iskazan u mjeri koju tvrtka koristi i eventualno novčanim iznosima, što ovisi o politici tvrtke i transparentnosti prema klijentu.

Inicijalizacija procjene vrši razvojni inženjer koji radi na datom projektu te čitajući prije napisanu poslovnu i tehničku specifikaciju kreira tablicu koja je podijeljena u razvojne module sa svojim zasebnim stavkama. Tako imamo npr. modul pod nazivom Mobilna aplikacija koja sadrži stavke kao što su Inicijalizacija projekta, Izrada strukture projekta, Podešavanje okruženja i sl. stavke koje su često nazvane i generičkim stavkama, stavkama koje posjeduje većina aplikacija i koje se rijetko izmjenjuju ili uklanjaju. Osim generičkih stavki imamo i specifične stavke kao što bi bila Razvoj nfc modula za kartično plaćanje, a one variraju od aplikacije do aplikacije, čak ista stavka u različitim aplikacijama može zahtijevat više ili manje vremena razvoja sve u ovisnosti s okruženjem sustava. Pored stavki glavnih modula možemo pronaći vrijeme izrade svake stavke. Vrijeme izrade se mjeri u jediničnoj mjeri koju je razvojna agencija definirala, ali je obavezno radi razumijevanja navesti klijentu konverziju jedinične mjere u standardnu kao što su npr. sati. Najčešće će vrijeme razvoja biti iskazano u satima, ali možemo ponekada vidjeti zasebnu mjernu jedinicu kao npr. bodovi. Bodovi su jedna od metoda kojima se služe voditelji projekta i inženjeri, često je jedan bod ekvivalent nekom broju sati, a najčešće ćemo u industriji vidjeti da je jedan bod jednak četiri puna sata.
Rijetko ćemo vidjeti u procijeni da je iskazan i iznos zasebnih stavki ili cjelokupnog rješenja, ali je i to moguće. Prikaz novčanih iznosa ovisi o samoj razvojnoj agenciji, transparentnosti prema klijentu i vlastitim djelatnicima te najvećim djelom modelu poslovanja. Modeli poslovanja su zasebna tema te se neće obraditi u ovom radu, ali ukratko mogu postojati različiti poslovni modeli gdje je jedan od njih naplata po satima razvoja. Naplata po satima razvoja jedina odgovara za prikaz novčanih iznosa pojedinih stavki u procijeni jer se umnoškom broja sati razvoja i fiksne cijene razvoja može doći do određene sume koliko neka stavka iznosi, a na kraju i cjelokupni sustav. Vrijedi i da se npr. stavke vezane uz dizajn naplaćuju po satu u drugačijem iznosu nego npr. stavke za razvoj baze podataka.

Osim modula, stavki, sati razvoja i eventualno iznosa postoje i sigurnosni sati tzv. buffer koji je dodan na ukupan iznos sati projekta. Takvi sati služe ako dođe do neočekivanih problema prilikom razvoja, a to mogu biti tehničke poteškoće, bolovanje djelatnika i sl. da se i dalje ostane u ugovorenom vremenskom prozoru razvoja i završi projekt na vrijeme. Sigurnosni sati ovise o zahtjevnosti projekta i ostalim prije navedenim faktorima, a definira ih inženjer zadužen za projekt. Tvrtke ponekad imaju politiku podjele projekata na manje, srednje i veće te isto tako dodjeljuju i sigurnosne sate u određenom postotku na glavne sate kao npr. mali projekt koji je definiran kao 100 sati razvoja može imati 10% sigurnosnih sati što bi ukupnu sumu razvoja popelo na 110 sati razvoja, isto tako se često ne navode takvi sati klijentu već se raspodijele po stavkama jednako.

Važno je napomenuti da se u stvarnom razvoju gotovo uvijek događa da se sati razvoja neke stavke prebacuju na neku drugu stavku i često se događa da se projekti oduže ili dovrše prijevremeno, tako nešto upućuje na lošu procjenu i zahtjeva detaljnije specifikacije kako bi se u budućnosti izbjegao takav problem.

Modeliranje baze podataka

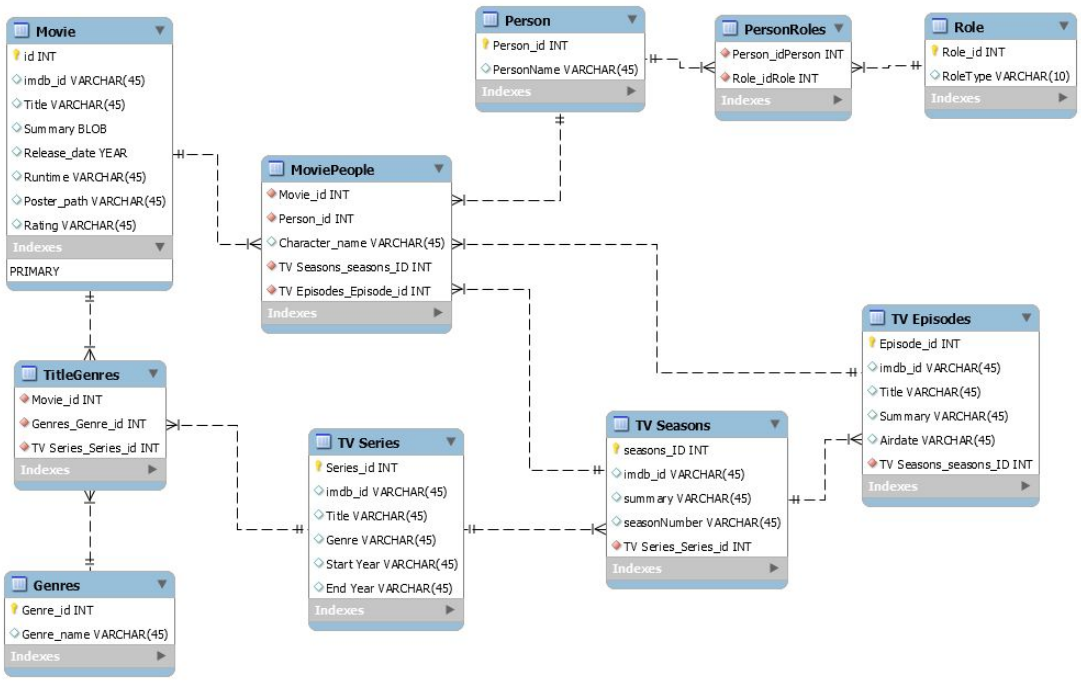
Dosadašnji koraci u procesu razvoja aplikacija su uglavnom sastavni dio planiranja i služe kako bi se postigla maksimalna priprema prije samog početka razvoja. Jedan dodatan i jako važan korak je modeliranje koje je poveznica između plana i razvoja aplikacija. Postoje mnoge metodologije modeliranja informacijskih sustava i na više načina se razlikuje njihova izvedba, međutim cilj im je zajednički, a to je da se na kraju izradi kvalitetna i pouzdana baza podataka. Baza podatak, ako je aplikacija posjeduje, a većinom je to slučaj, je najvažniji segment u aplikaciji ili bilo kakvom drugom informacijskom sustavu. Najčešće postoje nekoliko faza modeliranja informacijskog sustava kao npr. modeliranje procesa, modeliranje entiteta i veza i na kraju modeliranje baze podataka. Neki će tvrditi da su potrebni svi modeli, neki će s druge strane reći da niti jedan model nije potreban. Uglavnom oni koji tvrde da niti jedan model nije potreban su u krivu. Moramo shvatiti sljedeće, a to je da svaki informacijski sustav, svaka aplikacija zahtjeva jako dobru pripremu, jako dobru strukturu i najvažnije jako dobru strukturu baze podataka. Aplikacije možemo usporediti s najjednostavnijim primjerima iz života kao npr.

slaganje ormara koji smo kupili u nekoj trgovini. Kada slažemo kupljeni ormar otprilike znamo kako neki ormar izgleda i otprilike možemo bez uputa složiti taj isti ormar. Ormar ćemo na kraju uz duže potrošeno vrijeme nego što bi trebali i složiti, ali nakon nekog određenog vremena primijetiti ćemo da nam se ormar nagnuo više na jednu stranu ili da nam vrata od ormara zapinju prilikom otvaranja ili da se vrata ne mogu do kraja zatvoriti, možda ćemo primijetiti i da smo krivo montirali kvake za ormar ili naopako postavili poleđinu ormara pa je umjesto bijele boje smeđa. Da smo se odlučili slagati ormar prateći upute i gledajući u sliku ormara kakav bi trebao na kraju izgledati, do ovakvih problema dugoročno ne bi niti došlo, a ovako sada imamo novi, skupo plaćeni ormar, koji je dovoljno funkcionalan da ga ne moramo baciti, ali nam isto tako svakodnevno stvara male probleme koji nas uznemiruju. Poanta ove usporedbe je ta da informacijski sustav možemo složiti bez plana i modela baze i on će na kraju na neki način funkcionirati, međutim dugoročno ćemo morati ispravljati neke funkcionalnosti, primijetiti ćemo da nam je aplikacija sporija nego što smo očekivali, duže vremena ćemo provoditi na razvoju i ispravljanju i na kraju će klijent biti nezadovoljan, a eventualno će i tvrtka trpjeti gubitak zbog nezadovoljnog klijenta.

U nekim tvrtkama se provode sve vrste modeliranja informacijskih sustava, a kod drugih samo neki koji su više prilagođeni prirodi tvrtke, što jako ovisi o stručnosti djelatnika, broju djelatnika, roku izrade i budžetu klijenta. Onaj model koji je najnužniji je Model baze podataka i bez njega jedino što možemo napraviti je klimavi ormar.

Prilikom izrade modela baze podataka možemo koristiti različite metodologije, možemo kreirati model na papiru ili se koristiti nekim prikladnim programom za digitalnu izradu modela. Kako ovaj rad govori o JavaScriptu i razvoju modernih aplikacija nećemo duboko ulaziti u izradu modela baze s obzirom na to da je zasebno područje koje zahtjeva poseban rad. Ono što ukratko i pojednostavljenim vokabularom možemo reći za izradu modela baze podataka je taj da je potrebno skicirati sve moguće entitete tj. tablice koje će naša baza podataka sadržavati. Potrebno je svakoj tablici dodijeliti atribute tj. stupce koje će sadržavati podatke kasnije. Određene atribute treba označiti kao primarne ključeve ili šifrarnike. Šifrarnici nam služe za jedinstveno označavanje jednog retka unesenih podataka kako bi ih razlikovali jer npr. ako imamo tablicu Osoba koja sadrži atribute ime i prezime može doći do zabune ako u tablici imamo dvije osobe s

istim imenom i prezimenom. Osim primarnog ključa moramo u modelu označiti i vanjske ključeve koji su inače primarni ključevi neke druge tablice koja ima vezu s prvom tablicom. S obzirom da je proces kompleksan i zahtjevan za opisati u nastavku *Slika 4.* prikazuje jedan jednostavan model baze podataka. Ovdje spominjemo veze koje je isto važno označiti u modelu i one nam prikazuju relacije između tablica. Sve navedeno je potrebno raditi prema specifikacijama aplikacije te nam je ovdje više od pomoći poslovna specifikacija jer nam ona govori o strukturi klijentove organizacije. Kako cijeli proces modeliranja baze podataka u izvedbi izgleda najviše ovisi o razvojnoj agenciji tj. tvrtki, ali preporučljivi način bi bio da se model baze piše na papir kroz 4 do 9 iteracija kako bi se izbjegao bilo kakav propust, a zatim da se pomoću nekih alata digitalizira kako bi se zbog eventualnih izmjena i dopuna u aplikaciji mogao izmijeniti i on. Nakon što je razvoj završen i kada se postigne konačan model koji bi trebao biti konačan i prije samog početka razvoja, tada se pohranjuje u tehničku specifikaciju i dostavlja se klijentu. Dostavljanje modela baze podataka opet ovisi o politici i transparentnosti razvojne agencije prema svojim klijentima.



Slika 4. Primjer modela baze podataka

Struktura projekta

Nakon što je napravljena finalna verzija modela baze podataka, kreće se s razradom strukture projekta. Struktura projekta se sastoji od poslovne strukture i razvojne strukture. Poslovna struktura je ugl. dio koji odrađuje odjel za vođenje projekta. Oni u procesu poslovne strukture projekta određuju tko će sve sudjelovati od inženjera u projektu, tko će biti glavni inženjer za projekt te raspodjeljuju manje dijelove projekta tj. zadatke pojedinim inženjerima s vremenskim rokom izvedbe. Osim raspodjele zadataka inženjerima, dodjeljuje se zadatak i timu za dizajn koji razvija izgled aplikacija koji je potrebno što prije izraditi kako bi se prezentirao klijentu koji ga mora odobriti i razvojnom timu koji po potrebi dizajna mora eventualno nešto izmijeniti. Kako se većina aplikacija sastoji od klijentskog i poslužiteljskog dijela, najčešće je slučaj da se dizajn dovršava za vrijeme razvoja poslužiteljskog dijela aplikacija kako se ne bi trošilo vrijeme, a opet da se dizajn ne napravi prije svega, jer može doći do toga da se utroši veliki broj sati na izradu dizajna, a klijent u međuvremenu odustane od projekta.

Osim poslovne strukture istovremeno se odvija i razvojna struktura za koju je zadužen glavni inženjer. On definira od kojih dijelova se sastoji cjelokupna aplikacija kao npr. od baze podataka, api-a, mobilnog klijenta i web klijenta te osim toga u dogovoru s voditeljom projekta dodjeljuje zadatke svojim podređenima i dodjeljuje im detaljnije informacije o svakom zadatku zasebno.

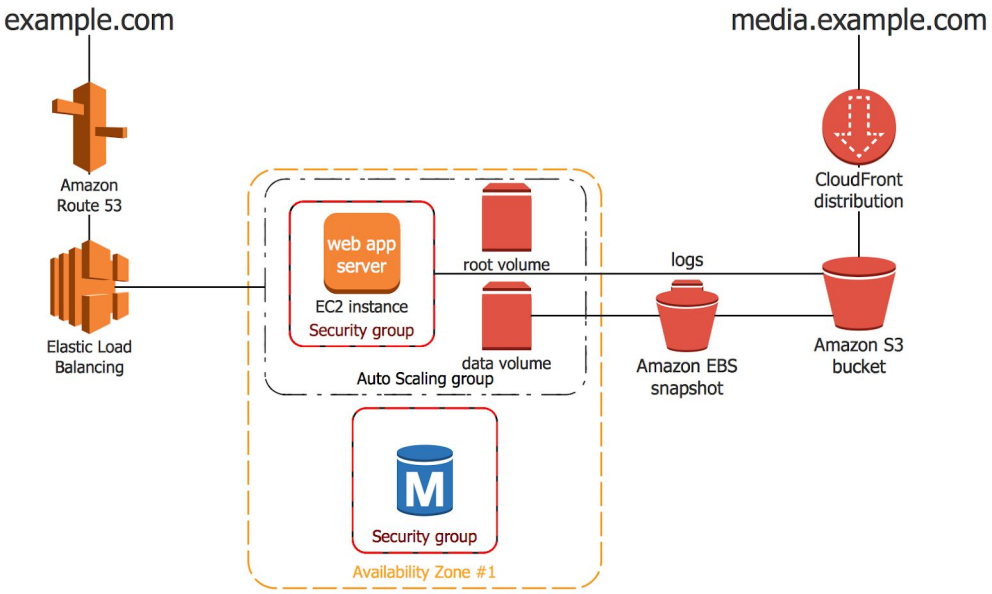
Kako bi dobili konkretniju sliku strukture nekog projekta uzmimo za primjer izradu jednostave web trgovine za automobile. Struktura projekta će izgledai na sljedeći način. Dodjeliti će se jedan dizajner za projekt, zatim će se dodjeliti jedan voditelj projekta, te će se na temelju tehničke specifikacije i procijene dodjeliti određeni broj razvojnih inženjera sa određenom razinom znanja, pa npr. s obzirom da se radi o jednostavnoj web trgovini dodjeliti će se jedan inženjer s visokom razinom senioriteta i jedan inženjer s manje iskustva. Inženjer s više iskustva ili voditelj razvojnog tima će raspodjeliti konkretne tehničke zadatke inženjerima i podjeliti

projekt u logične cijeline kao npr. autentifikacija web trgovine, lista automobila i detalji, naplata i sl.

Okruženje i infrastruktura aplikacija

Prvi put kada inženjeri krenu koristiti računalo i primjenjuju praktične vještine je onaj kada je potrebno podesiti okruženje i infrastrukturu aplikacija koja je definirana prethodno u tehničkom djelu specifikacije. Pod okruženje se smatra da se kreiraju i aktiviraju instance servera za sve zasebne servise koji će se razvijati, to bi podrazumijevalo kreiranje i aktivacija poslužitelja za bazu podataka, API-a, web klijenta te npr. za mobilnog klijenta kreiranje i aktivacija instance u trgovinama aplikacija za podržane platforme (Google Play, App Gallery, App Store). Svaku instancu je potrebno kreirati s potrebnim parametrima od broja procesora do količine memorije i potrebno ih je osigurati na što bolji način kako ne bi bilo tko mogao pristupiti resursima. Od dodatnih stvari je najbolje u samom početku integrirati neku vrstu kvalitetnog alata za praćenje resursa, takvi alati bi bili npr. PM2 ili Supervisor. Za sve navedene instance potrebno je zatim podesiti konkretna okruženja za sustav. Okruženja može biti koliko god razvojni tim smatra da je potrebno, ali najčešće su to testno okruženje, staging okruženje i produkcijsko okruženje. Testno okruženje je dostupno svim razvojnim inženjerima i nije nužno okruženje jer inženjeri imaju mogućnost kreiranja svog lokalnog testnog okruženja, staging okruženje služi za ujedinjenje svih funkcionalnosti razvijene od strane različitih inženjera, testiranje te prezentiranje klijentu, dakle takvo okruženje je dostupno razvojnim inženjerima, testerima i klijentu. Kao zadnje okruženje navodimo produkcijsko i kao što njegovo ime govori, to je glavno okruženje na kojem se pokreće instanca koda koja čini krajnji proizvod dostupan svim korisnicima aplikacija uz moguću restrikciju ovisno o željama klijenta, npr. ako se radi o internoj aplikaciji tvrtke za njihove djelatnike ili o javno dostupnoj aplikaciji dostupnoj samo na određenom geografskom području. Važno je napomenuti da se infrastruktura cjelokupnog informacijskog rješenja mora unaprijed planirati, ali je često šablonski proces u većini razvoja jednostavnih poslovnih aplikacija. Infrastruktur se može kreirati s vlastitim resursima tj. resursima razvojne agencije, što je u današnje jako rijetko i neisplativo dugoročno, mogu se koristiti resursi klijenta što je isto tako vrlo rijetko, a ako i postoje resursi za infrastrukturu često su nekompatibilni i zastarjeli i treća opcija je korištenje već polu gotovih infrastruktura poput AWS-a (Amazon Web

Services)[12], GCP-a[13] (Google Cloud Platform), Microsoft Azure-a[14], Linode-a[15] i sl. kojima je potrebno podesiti malo konfiguraciju i spremni su za korištenje. Jednu jednostavnu infrastrukturu kreiranu pomoću servisa AWS prikazuje u nastavku *Slika 5*.



Slika 5. Primjer modela infrastrukture sustava

Komponente i način rada Javascript-a

JavaScript se kroz godine značajno unaprijedio, unaprijedili su mu se koncepti i dizajni kao programski jezik. Kao i svaki programski jezik, JavaScript se sastoji od nekoliko glavnih komponenata i često ga se u IT zajednici tretira kao poprilično neobičan programski jezik jer ponekad dozvoljava kršenje nekih pravila što zna rezultirati s uspješnim kodom, ali zna i predstavi problem programerima. Kako bi se izbjegle te greške i kako ne bi došlo do nikakvih nelogičnosti potrebno je dobro poznavanje JavaScripta kao programskog jezika i programiranja općenito. Nije dovoljno poznavanje samo if else uvjeta, for i while petlja te ostale koncepata programiranja i sintakse, već je jako zahvalno znanje o funkcioniranju programskog jezika

[12] https://aws.amazon.com/
[13] https://cloud.google.com/
[14] https://azure.microsoft.com/
[15] https://www.linode.com/

“ispod haube” tj. kako neki programski jezik, u ovom slučaju JavaScript procesuira kod koji pišemo u njemu kako bi postigli određenu funkcionalnost.

Javascript Engine (JavaScript pogon)

Prvo što moramo razumjeti podjelu programskog jezika koji može biti programski jezik niske razine i programski jezik visoke razine. Programski jezik visoke razine je onaj koji koriste ljudi tj. programeri. Takvi jezici su čitljivi i razumljivi čovjeku te su u pojednostavljenom obliku. Ako bolje razmislimo gotovo svaki programski jezik je takav, ali ih svejedno dijelimo i na one koji su programski jezici niske razine, to su oni koji su bliži binarnom kodu tj. strojnom jeziku. Dijelimo ih na dvije skupine jer programski jezici neke dijelove odrađuju umjesto programera, tako npr. JavaScript samostalno alocira i dealocira memoriju dok C programski jezik koji je programski jezik niske razine to ne radi, već je to posao programera.

Ove informacije su nam važne jer da bi se neki program uspješno pokrenuo, potrebno je kod koji je pisan u nekom programskom jeziku kompajlirati u najnižu razinu, a to je strojni jezik ili binarni kod koji se sastoji od 0 (nula) i 1 (jedan).

Prilikom kompajliranja JavaScript programskog jezika dolazi u uporabu tzv. JavaScript Engine koji je sastavni dio web preglednika, ali ga možemo pronaći i u prije navedenom runtime okruženju Node.js, međutim trenutno ćemo se fokusirati na web preglednik jer u oba slučaja je proces gotovo pa identičan, a za konkretan primjer ćemo opisati kako funkcionira JavaScript Engine Googleovog Chrome web preglednika pod nazivom V8 Engine jer se on koristi i kao JavaScript Engine (*Slika 7.*) u Node.js-u. Drugi web preglednici imaju svoje JavaScript Engine koji se minimalno razlikuju, neki od njih su npr. Spider Monkey za Firefox Mozillu, JavaScript Core za Safari, Chakra za Microsoft Edge.

JavaScript Engine može čitati i pokretati kod, zato je prvi korak čitanje izvornog koda koji je programer napisao, taj dio procesa još nazivamo parsiranje. Prilikom parsiranja koda, JavaScript engine ga pretvara u Stablo Apstraktne Sintakse (Abstract Syntax Tree, AST) te ga prosljeđuje dalje u alatni lanac kompajlera. Neki od poznatijih kompajlera jesu Acron, Esprima i cheroot. Prvo kroz što se provodi AST, u daljnjem tekstu kod, je interpreter koji pretvara kod na dva sljedeća načina. Prvi način je da uzima jednostavne dijelove koda tj. one dijelove koji su optimizirani i one dijelove koda koji se ne ponavljaju više od jednog puta te ih direktno pretvara

u binarni kod, a drugi način je da uzima složenije dijelove koda i tzv. vruće kodove tj. kodove koji su identični i primaju iste parametre (npr. funkcija koja se poziva na više mjesta), zatim ih optimizira pomoću TurboFan[16] kompajlera za optimizaciju uz pomoć mapiranih podataka prethodnog koda koji je već pretvoren prethodno u binarni kod te na kraju takav optimizirani kod pretvara u binarni kod.

Proces je vrlo jednostavan te ga se skraćeno može opisati bez dodatnih pojašnjenja oko pojedinih pojmova. Pojednostavljeno bi JavaScript Engine opisali kao alat koji prihvaća izvorni kod, zatim ga pretvara u stablo apstraktne sintakse i šalje interpreteru koji jednostavni kod pretvara u binarni kod, a složeniji i ponavljajući kod šalje kompajleru za optimizaciju, koji ga nakon optimizacije pretvori u binarni kod. Cijeli opisani proces se u V8 JavaScript enginu naziva JIT (justi-in-time compilation), a proces optimizacije prikazuje *Slika 6*.

Nažalost se cjelokupan proces rada JavaScripta ne oslanja samo na JavaScript Engine, već su potrebni neki dodatni alati koje pružaju web preglednici, a opisat ćemo ih u sljedećim poglavljima [2],[3].

Memory Heap (Hrpa memorije)

Svaka varijabla, funkcija ili objekt koji definiramo mora se pohraniti negdje u radnu memoriju. Memory Heap je komponenta u web pregledniku koja je zadužena za alociranje memorije u JavaScript kodu te ona za nas odrađuje taj dio posla. Ono na što moramo paziti kada je riječ o memoriji, je to da ju ne preopteretimo jer je memorija uvijek limitirana. Često možemo u IT zajednici čuti frazu memory leak ili na hrvatskom curenje memorije. Do curenja memorije dolazi ako preopteretimo Memory Heap gdje dolazi do prevelikog zauzeća radne memorije na računalu što na kraju može prouzrokovati rušenje aplikacije. Jedna prednost korištenja JavaScript jest ta da nam pruža ugrađeni kolektor smeća (garbage collector). Kolektor smeća je zadužen da oslobodi dijelove radne memorije od onih varijabli, funkcija i objekata čija se referenca više ne koristi u programu. Tako ne moramo brinuti toliko o neiskorištenom kod [2],[3].

[16] https://v8.dev/docs/turbofan

Stog poziva (Call Stack)

Druga vrlo važna komponenta je Stog poziva koji predstavlja frazu da je JavaScript programski jezik s jednom niti tj. poznatiji kao one threaded programming language. To znači da se JavaScript kod izvršava liniju po liniju i sljedeća linija koda se ne može pozvati niti izvršiti dok god prethodna linija nije izvršena. Takve programske jezike i samo programiranje još nazivamo sinkrono. Kako bi se osiguralo da se kod izvršava sinkrono, dolazi u ulogu Stog poziva koji je običan skup pozivanja koda i koristi metodu prvi unutra zadnji van. Stog poziva je sastavni dio runtime memorije te se sastoji od okvira za pozive. Svaki okvir za poziv se sastoji od lokalnih varijabli, argumenata i parametara te povratne adrese rezultata. Kada želimo izvršiti neku funkciju, Stog poziva postavlja okvir na vrh te ga nakon izvršenja izbacuje van.

Ono što ne želimo da se dogodi je tzv. Stack overflow koji je u pojednostavljenim riječima događaj kada se Stog poziva napuni s pozivima preko svog limita. U takvoj situaciji aplikacija će se srušiti i javiti nam da je došlo do Stack overflowa. Stack overflow možemo demonstrirati s neispravno napisanom rekurzivnom funkcijom koja nema uvjet prekida izvršavanja [2],[3] .

Web API

Za JavaScript postoji mnogo fraza koje su diskutabilne tako je i prije navedena fraza, programski jezik s jednom niti diskutabilan, isto tako se kaže da je JavaScript ne blokirajući jezik. Ranije je spomenuto da se funkcije u JavaScriptu izvršavaju jedna za drugom i da se ne mogu izvršavati istovremenu, međutim da je to istina većina aplikacija bi danas bila vrlo spora ili se neke funkcionalnosti ne bi mogle niti implementirati. Tako npr. ne bi mogli poslati korisniku obavijest dok on pretražuje neku listu u aplikaciji koja dohvaća podatke s poslužitelja, već bi morali čekati da korisnik prekine pretraživanje ili bi ga prekinuli u akciji slanjem obavijesti i prikazom iste na sučelju aplikacije.

Da bi se to izbjeglo uvedena je još jedna komponenta koja ne pripada sastavno JavaScript programskom jeziku već web preglednicima, a to je Web API[17]. Web API sadrži većinu funkcionalnosti koje zahtijevaju asinkronu izvedbu te dok se naš program izvršava Web API preuzima asinkroni dio programa. Neke od najkorištenijih funkcija Web API-a jesu fetch,

[17] https://developer.mozilla.org/en-US/docs/Web/API

setTimeout, setInterval i potrebna je implementacija istih od strane programera. Cjelokupni tok neke asinkrone radnje se odvija tako da se definirane asinkrone funkcije pozivaju i obrađuju od strane Web API-a, Web API-i izvršava funkciju, dok se ostatak funkcija koje su sinkrone dalje šalju u Red događanja (Event queue). Kada se funkcija iz Web API-a izvrši, tada šalje funkciju isto u Red događanja. Jedan primjer bi bio s funkcijom setTimeout, koja prima dva parametra, prvi parametar je povratna funkcija, a drugi je vrijeme čekanja za poziv povratne funkcije u milisekundama. Kada engine primijeti da se radi o asinkronoj funkciji, automatski ju šalje Web API-u koji ju izvršava na zasebnoj niti i to u C++ programskom jeziku. Dok se setTimeout ne izvrši u zadanom vremenu, druge funkcije se mogu izvršavati, a kada vrijeme iz setTimeout-a istekne tada se izvršava povratna funkcija iz nje [2],[3].

Ako imamo:

```
console.log("Print 1");
setTimeout(() => {
   console.log("Print 2");
}, 2000);
console.log("Print 3");
```

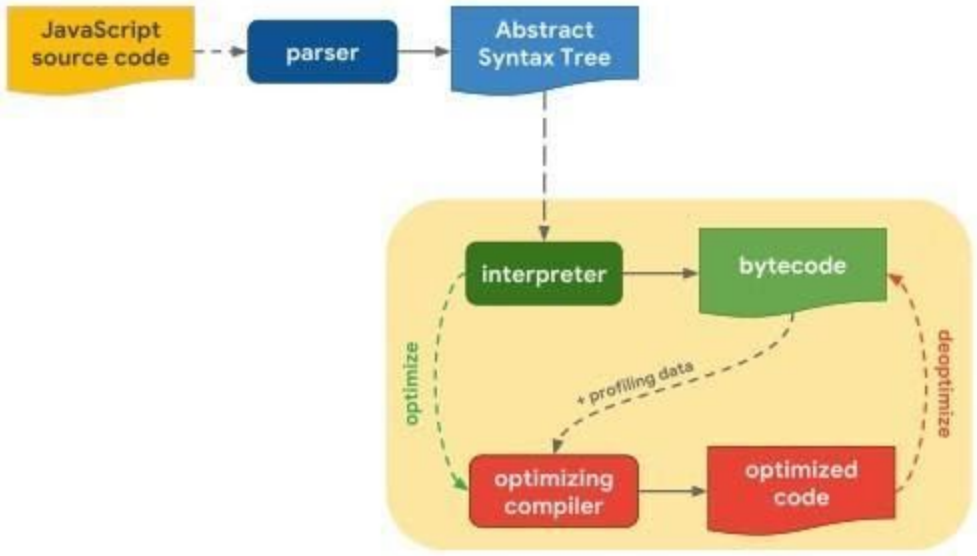
Rezultati:

```
Print 1
Print 3
Print 2 //nakon dvije sekunde
```

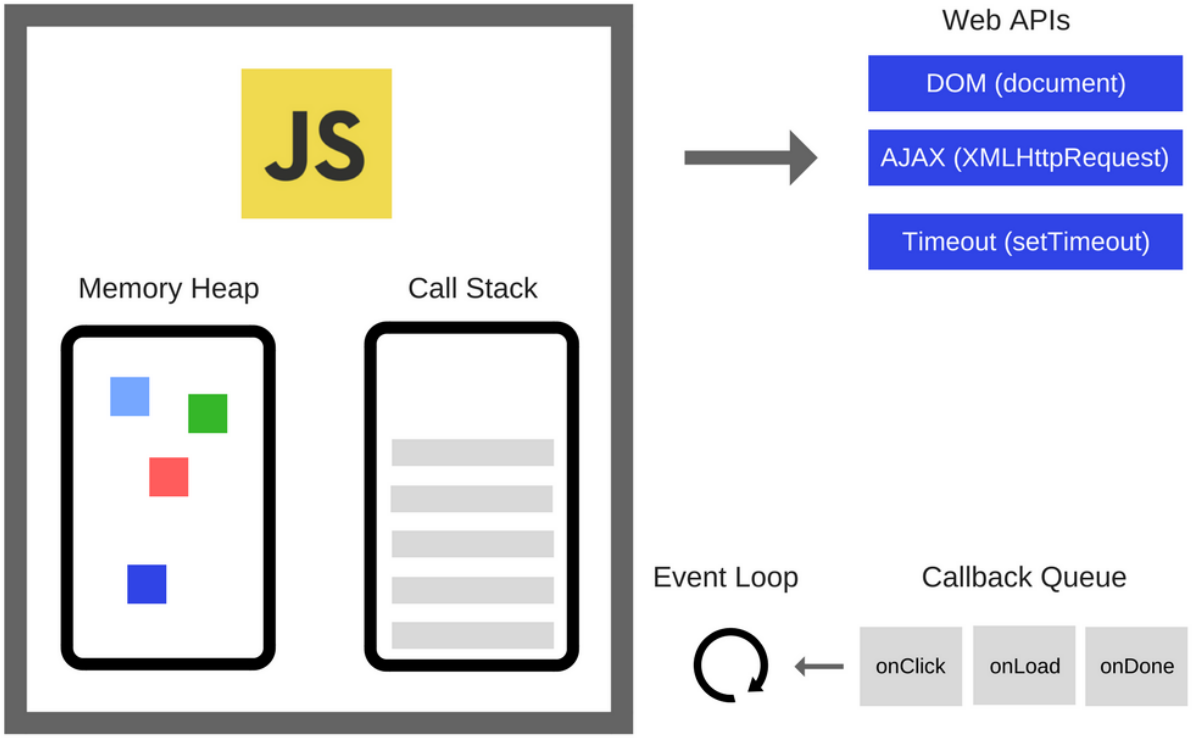
Red događanja, red poslova i petlja događanja

Red događanja (Event queue) koji smo prije spominjali je red u kojem čekaju funkcionalnosti za slanje u Stog poziva na izvršavanje. Od ECMAScript 2015 uvedena je dodatna komponenta tzv. red poslova (Job queue) koji sadrži JavaScript Promise sa resolverima i rejectorima koji imaju prednosti izvršenja nad ostatkom funkcija, što znači da će petlja događanja slati prvo funkcije iz reda poslova u Stog poziva, a onda tek iz reda događanja. Petlja događanja (Event loop) je petlja

koja se beskonačno izvršava i provjerava red poslova i događanja te Stog poziva. Petlju događanja možemo zamisliti kao petlju koja konstantno pita da li je Stog poziva prazan, da li ima kakva funkcija u redu poslova i događanja, te ako je stog prazan, šalje funkciju iz redova u stog. Nakon cijelog procesa se funkcije pozivaju u Stogu poziva na opisan način u prethodnim poglavljima [2],[3].



Slika 6. Prikaz optimizacije izvornog koda



Slika 7. JavaScript engine

Razvoj poslužiteljskih aplikacija

Pojavom Node.js-a i sve češćim korištenjem u razvoju modernih web aplikacija kao alternativa za poslužiteljske servise, povećao se i broj JavaScript programera zbog toga što se pružila mogućnost razvoja vještina programiranja u domeni klijentskog i poslužiteljskog razvoja.

Veliki broj ozbiljnih, produkcijskih poslužiteljskih servisa se danas razvija u Node.js-u iako po performansama nije najbolji izbor. Pravi razlog korištenja JavaScripta na poslužiteljskoj strani je ušteda resursa i radne snage, jer dok je prije bilo obavezno imati jednog programera za klijentsku stranu i jednog programera za poslužiteljsku stranu, danas ta dva posla može odraditi jedan programer u JavaScriptu. Svakako je preporučljivo u industriji da se ipak zapošljava programere za svako područje zasebno, ali tada imamo drugu prednost JavaScripta u razvoju cjelokupnog

sustava, a to je da oni programeri koji rade na klijentskoj strani aplikacije uvijek mogu pomoću programerima na poslužiteljskoj strani i obrnuto. Osim navedenog, postoji visoka razina kompatibilnosti poslužiteljske i klijentske strane te veliki broj dodatnih biblioteka i alata koji jednako dobro funkcioniraju. Postoji nekoliko oblika poslužiteljskih servisa, ali onaj najčešći u razvoju web aplikacija je API (Application Programming Language). API služi za dohvaćanje, slanje i manipulaciju podataka te je glavna poveznica između baze podataka i klijentske strane aplikacije. API odrađuje funkcije kreiranja, čitanja, izmjene i brisanja zapisa iz tj. u bazu podataka. API može sadržavati i dodatne funkcionalnosti ne vezane za bazu podataka.

REST-full i GraphQL API arhitektura

Postoje nekoliko arhitektura API-a, ali oni najkorišteniji jesu REST-full i GraphQL API. Često možemo pročitati članke o diskusijama koja arhitektura je bolja, ali u stvarnosti prednost arhitektura ovisi o projektu na kojem se radi. Ipak pokazalo se da je GraphQL API riješio neke probleme s kojima se REST API susreće, pa se možda može zaključka koji koncept ima prednost u kojoj vrsti projekta. Svaki od navedenih API-a ima svoje prednosti i mane, ali ono što im je zajedničko, jest to da za konačan cilj imaju ulogu slanja podataka putem HTTP protokola za čitanje i manipulaciju nad bazom podataka.

REST-full API

REST-full ili kraće nazvan REST API, je najstarija arhitektura i koristi se u jako velikom broju poslužiteljskih servisa za aplikacije. Način na koji funkcionira je taj da se kreira poslužitelj na kojem će se pokretati API, najčešće korištene biblioteke za kreiranje poslužitelja jesu Express.js, Koa i Hapi. Postoje neke razlike u navedenim serverima, ali se s bilo kojim od ta tri može postići isto. Nakon uspješno kreiranog, podešenog i pokrenutog poslužitelja, potrebno je kreirati krajnje točke. Krajnje točke jesu rute API-a na kojima se izvršava neka operacija. Kako bi ih se definiralo potrebno im je proslijediti željenu HTTP metodu, putanju i operaciju u obliku povratne funkcije. HTTP metode jesu GET za čitanje podataka, POST za kreiranje podataka, PUT za izmjenu i zamjenu cjelokupnog skupa podataka, PATCH za izmjenu određenog podatka, DELETE za brisanje podataka. Nakon definirane HTTP metode na krajnjoj točki, definiramo

putanju koja odgovara url-u naše aplikacija, ali ne moramo definirati http/s protokol i domenu, već samo one rute koje želimo da sadrže neku od operacija npr. krajnja točka /korisnici će u web pregledniku izgledati s domenom https://rest-full-api.com/korisnici. Nakon definirane metode i putanje možemo započeti s raspisivanjem logike operacije u povratnoj funkciji. Unutar povratne funkcije definiramo pomoću nekih dodatnih biblioteka operacije nad bazom podataka i manipulaciju podataka prije upisa u bazu ili slanja na klijentsku aplikaciju. Za primjer tako jedne funkcije možemo uzeti operaciju čitanja svih aktivnih korisnika iz baze podataka. Tada ćemo pomoću biblioteke za slanje SQL upita nad entitetom baze podataka korisnik dohvatiti sve retke kojima nije istekao rok, a dobiveni odgovor tj. rezultate ćemo uz pomoć http funkcije send(), poslati klijentskoj aplikaciji koja će se pobrinuti za prikaz podataka.

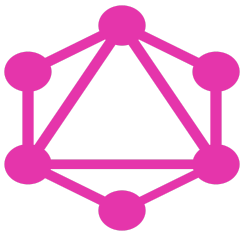
Princip rada i implementacija REST API-a je vrlo jednostavna i pruža mogućnost brzog razvoja bez previše razmišljanja, ali nažalost ima i nekoliko mana. Jedna od mana je ta što je za svaki entitet baze podataka potrebno pisati odvojene krajnje točke pa ćemo tako npr. za dohvat svih blog objava nekog korisnika imati minimalno dvije krajnje točke, točku za dohvat korisnika i točku za dohvat blog objava. Druga veća mana REST API arhitekture je suvišan dohvat podataka, to znači da ako jedan entitet npr. korisnik sadrži stupce podataka kao što su id, ime i prezime i mi želimo dohvatiti ime korisnika, morat ćemo dohvatiti id i prezime tj. sve podatke koje taj entitet sadrži, a način za spriječiti suvišan dohvat podataka nažalost ne postoji [4].

GraphQL API

GraphQL (*Slika 8.)* API ima drugačiji pristup podacima tako što ne tretira podatke različitih entiteta kao zasebne krajnje točke već se sastoji od jedne krajnje točke koja se dijeli na dva čvora, a ta dva čvora se dalje dijele na manje čvorove. Takvim načinom rada rada GraphQL API se definira kao API sa stablastom strukturom u kojem su svi podaci entiteta jednako povezani tj. povezani su u onoj mjeri u kojoj ih programer poveže prilikom razvoja. GraphQL API se sastoji od sheme koja sadrži tipove podataka i resolvera koji vrše operaciju nad bazom podataka. U shemi definiramo sve tipove podataka koje ćemo koristiti u aplikaciji, često na njih gledamo kao reprezentacije entiteta iz baze i možemo tipove podataka ugnijezditi u druge tipove podataka za koje često povlačimo referencu na vanjske ključeve. Ovisno koji GraphQL server koristimo, koristit ćemo i njegove primitivne tipove podataka za definiranje naših tipova podataka u shemi.

Neke od najkorištenijih biblioteka za kreiranje graphql servera jesu express-graphql, Apollo server i GraphQL Yoga. Isto kao i kod REST servera, postoje razlike u bibliotekama ali se s bilo kojom od njih dobiva željena funkcionalnost te sve biblioteke sadrže ekvivalente tipova podataka poput string, number, boolean i sl. Za svaki osnovni tip podataka u shemi, moramo kreirati po potrebi ulazni tip podataka i tip podataka za filtriranje, ako želimo kasnije unositi nove zapise u bazu podataka ili dohvaćati podatke po filterima. Kada smo definirali shemu moramo kreirati i željene resolvere. Resolveri nisu ništa drugo nego najobičnije funkcije koje odrađuju neku operaciju nad bazom podataka. Prije navedena dva čvora od kojih se sastoji krajnja točka, definiramo kao točku za upite (Query) i točku za mutacije (Mutation). Točka za upite nam služi za čitanje podataka iz baze dok nam je točka za mutacije zaslužna za sve vrste manipulacija nad bazom. Prilikom definiranja resolvera moramo obratiti pažnju da kreiramo samo one resolvere koji su nam potrebni kasnije u aplikaciji. Moramo definirati resolver za upit pojedinog zapisa entiteta i skupa zapisa entiteta, tako npr. moramo imati resolver koji dohvaća pojedinog korisnika po jedinstvenom identifikatoru i resolver koji dohvaća skup korisnika s filterom ili bez. Svakom resolveru se dodjeljuje tip iz sheme kako bi se znao u kakvom formatu mora vratiti željene rezultate, argumente koje može upit zaprimiti i povratna funkcija koja čita podatke iz baze. Kada definiramo resolver za neku mutaciju moramo isto tako dodijeliti tip podataka koji vraćamo, zatim argumente koje može mutacija primiti i povratnu funkciju koja izvršava operaciju nad bazom podataka. Konkretan dohvat i manipulacija podataka se izvršava kao i kod REST API arhitekture, pomoću dodatnih biblioteka. Postoje dodatne mogućnosti definiranja univerzalnih tipova podataka koji se kasnije primjenjuju na ostale tipove podataka, a to su u praksi najčešće paginatori i brojači.

Možemo zaključiti da GraphQL API rješava gotovo sve probleme s kojima se možemo susresti u REST-full API arhitekturi i da je zbog svoje kompleksnosti najpovoljniji za korištenje u velikim aplikacijama. Svakako postoje i nedostaci koje GraphQL nosi sa sobom, a to je otežano rukovanje s greškama, slaba implementacija predmemoriranja te sama kompleksnost arhitekture. Ono što rješava ipak takve probleme jesu dodatne biblioteke za rukovanje s greškama i za predmemoriranje podataka, a samu kompleksnost arhitekture ne možemo riješiti, ali zato možemo ubrzati razvoj korištenjem biblioteka i skripti za autogeneriranje sheme prema uvidu na bazu podataka [4].



Slika 8. GraphQL logo

Razvoj klijentskih aplikacija

Istina je da razvoj poslužiteljskih servisa zahtjeva veliko znanje poznavanja principa rada baze podataka, mrežnog sustava i protokola, kibernetičke sigurnosti i drugih složenijih koncepata i dugi niz godina se smatralo navedene vještine i razvoj poslužiteljskih servisa zahtjevnijim i ozbiljnijim dijelom razvoja aplikacija. Danas je situacija nešto drugačija i kompleksnost razvoja klijentskih aplikacija se gotovo pa izjednačio s razvojem poslužiteljskih servisa, moglo bi se ponekad zaključiti da je čak krivulja učenja teža u razvoju klijentskih aplikacija nego poslužiteljskih. Kompleksnost razvoja na klijentskim aplikacijama možemo zahvaliti novim i modernim bibliotekama i razvojnim okruženjima čija je svrha ubrzati i pojednostaviti razvoj. Iako je danas kompleksnost veća zbog uvođenja novih biblioteka i alata, ta kompleksnost rješava mnoge probleme s kojima su se programeri prije susretali.

JavaScript u ovom području dominira u usporedbi s drugim programskim jezicima i pruža najveću paletu alata za razvoj klijentskih aplikacija. Gotovo je moguće svaku aplikaciju danas napisati u JavaScriptu, radilo se o web, mobilnoj ili nekoj drugoj platformi. JavaScript se u razvoju klijentskih aplikacija toliko razvio da je u nekim oblicima čak zamijenio HTML i CSS te je moguće cijelu aplikaciju napisati samo u JavaScriptu. Danas pomoću JavaScripta možemo kreirati HTML elemente i manipulirati s njima, a isto tako možemo dodavati stilove i stilske klase koje definiramo kao JavaScript objekte. Iako nam je potreban samo JavaScript prilikom razvoja, na kraju se ipak kod napisan u JavaScript translatira u dobri stari HTML i CSS, tako da

je ključno i jako dobro poznavanje tih dviju tehnologija kako bi se mogle razvijati moderne i kvalitetne klijentske aplikacije.

Slika 9. Logo HTML5, CSS3 i JavaScript

HTML

HTML (HyperText Markup Language) je najjednostavniji gradivni blok web aplikacija čija je zadaća definiranje strukture aplikacije i povezivanja ekrana međusobno. HTML koristi tzv. markup anotaciju a definirana je određenim brojem imenovanih elemenata. Elemente je moguće prema zadanim pravilima ugnježđivat te se tako stvara struktura web aplikacije. Neka zadana pravila jesu da se svi elementi moraju pisati posebnom sintaksom koja je oblika *<Oznaka></Oznaka>,* obavezni su znakovi veće i manje koji obgrljuju naziv elementa napisan malim slovima. Takva oznaka se piše prilikom otvaranja elementa, a zatim i prilikom zatvaranja elementa s dodatkom znaka kose crte prije znaka veće. Između dvije navedene oznake se mogu ugnježđivati dodatni HTML elementi ili se može dodati sadržaj ako to određeni element dozvoljava, također svaki HTML dokument mora biti pohranjen u datoteci s ekstenzijom .html. Svi definirani HTML elementi sadrže posebna pravila i hijerarhiju. Neke elemente ne možemo ugnijezditi u druge elemente zbog pravila hijerarhije, a neki elementi ne mogu poprimiti atribute od drugih kao što je slučaj s elementom p koji je element za ispis teksta i ne može poprimiti atribut href za dodjelu poveznice elementu a koji je element za strukturu poveznice. U nastavku se nalazi primjer HTML koda (*Primjer sintakse HTML-a*).

Primjer sintakse HTML-a:

```
<!DOCTYPE html>
<html>
<head>
<title>Naslov stranice</title>
</head>
<body>

<h1>Primjer naslova</h1>
<p>Primjer paragrafa.</p>

</body>
</html>
```

Najkorišteniji HTML elementi:

- **head** - zaglavlje html dokumenta za meta podatke
- **body** - tijelo html dokumenta
- **section** - sekcija html dokumenta
- **footer** - podnožje html dokumenta
- **div** - blok element
- **h1/h2/h3/h4/h5/h6** - naslovi i podnaslovi
- **p** - tekst
- **span** - umetak
- **a** - link
- **img** - slika
- **ul/li** - lista i elementi liste

Postoje i univerzalni atributi koji se mogu primijeniti na gotovo svim HTML elementima, najpoznatiji su style i class i id. Style atribut služi za dodjeljivanje stila elementa unutar same oznake elementa dok class atribut dodjeljuje naziv klase elementa koji se kasnije može koristiti za referencu više elemenata i tako se jednim zapisom stila dodjeljuje isti stil svim referenciranim elementima. Zadnji navedeni atribut id je atribut za jedinstvenu identifikaciju elementa putem

kojeg se određeni element može referencirati. Trenutna verzija HTML-a je verzija 5 u kojoj su dodani novi elementi, atributi i pravila [5].

CSS

CSS (Cascading Style Sheets) je standardni, deskriptivni jezik tablice stilova za definiranje stila tj. izgleda HTML i XML dokumenata. Pomoću CSS-a određujemo različite atribute HTML elemenata poput veličine, boje, margina i sl. Podržan je veliki broj digitalnih mjernih jedinica uključujući i neke posebne mjerne jedinice koje je moguće koristiti samo u web okruženju tj. okruženju koje dozvoljava neki oblik CSS-a.

Kako bi mogli dodijeliti pojedini stil zasebnom elementu ili skupu elemenata koristimo se prije navedenim jedinstvenim identifikatorima (id), klasama (class) i nazivima elemenata (npr. div), a moguće je i preciznije definiranje stilova uz pomoć CSS funkcionalnosti nasljeđivanja elemenata, pa tako možemo npr. odrediti da nam je svaki treći element oznake p crvene boje ili da je svaki element a koji se nalazi u drugom elementu div podebljan.

CSS se može i pomoći HTML atributa priložiti direktno u element unutar vitičastih zagrada, a preporučeni način je zasebno odvojiti stilove u datoteku s ekstenzijom .css koji se priloži u HTML dokument uz pomoć link elementa unutar kojeg se navodi da se radi o CSS datoteci i putanji na kojoj se taj dokument nalazi. Unutar odvojenog CSS dokumenta nižu CSS objekti koji imaju oblik identifikator { atribut: vrijednost } (*Primjer sintakse CSS-a*) [5].

Primjer sintakse CSS-a:

```
h1 {
 color: "pink"
}

#someId {
 background-color: "grey"
 padding: 10px
}

.someClass {
```

```
  margin: 0 10px 12px 3px
  border: "1px solid black"
}
```

Razvojna okruženja i biblioteke

Razvoj modernih aplikacija u JavaScriptu kakve danas znamo ne bi bio moguć bez nekih dodatnih alata koji su ubrzali razvoj i pridonijeli visokoj kvaliteti. Takve alate nazivamo razvojnim okruženjima (Farmworks) i bibliotekama (libraries), a neke od njih smo već spominjali u prethodnim poglavljima.

Potrebno je znati razliku između razvojnog okruženja i biblioteke. Ključna razlika ova dva pojma jest ta da razvojno okruženje ima kontrolu nad tijekom razvoja programa te dozvoljava dodavanje izvornog koda od strane programera, dok biblioteke prepuštaju razvoj toka aplikacije programeru te se mogu pozivati po potrebi.

Razvojna okruženja se sastoje od pojedinih modula koje možemo, ali i ne moramo koristiti, tako npr. imamo modul za predmemoriranje koji će postojati u toku aplikacije iako ga ne planiramo koristiti. Kod biblioteka je situacija drugačija, modul se sastoji od skupa metoda i objekata koje možemo pozivati unutar izvornog koda kada želimo i gdje želimo. Za one metode i objekte koje ne pozovemo neće se kreirati referenca u radnoj memoriji.

Prilikom razvoja modernih aplikacija u industriji, gotovo je nemoguće ne koristiti biblioteke zato što bi razvoj takvih aplikacija trajao predugo i ne bi se isplatio krajnjim klijentima. Tako koristimo razne biblioteke za manipulaciju vremena, biblioteke za predmemoriranje, biblioteke za pojednostavljeno stiliziranje i mnoge druge. Postoji ponekad i javna rasprava da li je neki alat biblioteka ili razvojno okruženje, ali presudan faktor je prije navedena kontrola toka razvoja aplikacije. Najveća rasprava se vrti oko alata React.js koji prema svim standardima spada u skupinu biblioteka, ali u kombinaciji s nekoliko drugih biblioteka koje su napravljene samo za React.js[18] ukupno čini razvojno okruženje po definiciji. U nastavku ćemo opisati React.js kao najkorišteniju JavaScript biblioteku današnjice pomoću koje su napravljene mnoge velike aplikacije poput Facebooka, Instagrama, Netflixa, Dropboxa i sl.

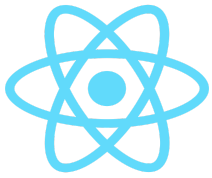
[18] https://reactjs.org/

React.js

U prošlosti je najpopularnija JavaScript biblioteka bila jQuery koja je omogućavala olakšanu manipulaciju nad DOM-om. DOM je stablasta struktura koja opisuje izgled i cjelokupnu strukturu aplikacija, a sastoji se od HTML elemenata. React.js (*Slika 10.*) biblioteka u daljnjem tekstu samo React je preuzeo ulogu jQueryja nakon 2013. godine kada je objavljen kao otvoreni kod, a do tada je bio korišten samo interno od strane Facebooka gdje je i razvijen. Razvio ga je programer Jordan Walke, koji je dobio inspiraciju za takvom bibliotekom od XHP biblioteke za komponente koja se koristila u PHP programskom jeziku. Ciljevi Reacta su olakšati manipulaciju nad DOM-om i ubrzani razvoj korisničkog sučelja. Osnovni princip rada Reacta je tzv. Lego princip razvijanje korisničkog sučelja od komponenata (Components). Moguće je ugnježđivati već zadane komponente međusobno, te se tako dobivaju nove kompleksnije komponente. Zadane komponente primaju vrijednosti za zadana svojstva (props) te se putem njih proširuje funkcionalnost same komponente, a komponente koje smo sami razvili primaju samo svojstva koja definiramo. Princip prosljeđivanja podataka iz komponente u komponentu se odvija kroz svojstva, a privremeno memoriranje neke vrijednosti možemo ostvariti drugim konceptom pod nazivom stanje (state). Stanje je poseban JavaScript objekt, specifičan za React u kojem se mogu pohranjivati vrijednosti u obliku ključ-vrijednost. Izmjene nad stanjem su moguće samo uz pomoć jedne specifične funkcije, a to je funkcija setState(), koja za argument prima objekt sa sadržajem željenog ključa koji želimo izmijeniti i pripadajuću novu vrijednost. Moguće je razvijati komponente sa stanjem i bez stanja, još ih nazivamo komponenta razreda (Class component) i funkcionalne komponente. Komponente razreda su do 2018. godine bile jedina vrsta komponente koja je mogla imati svoje stanje dok funkcionalne komponente to nisu mogle.

Sredinom 2018. godine uvedene su neke dodatne metode u React koje su omogućile postavljanje stanja u funkcionalne komponente, takve metode se nazivaju kuke (hooks). Jedan od dodatnih koncepata Reacta jesu metode životnog ciklusa (lifecycle methods) koje isto tako nisu bile dostupne u funkcionalnim komponentama do pojave kuka. Metode životnog ciklusa jesu metode koje se pokreću u određenom trenutku životnog ciklusa aplikacije te unutar njih možemo izvršiti neku željenu funkciju. Neke od njih se pokreću prije prikaza komponente, neke prilikom izmjene na komponenti, a neke kada se komponente više ne prikazuje. Kuka useEffect je uvela

jedinstvenu metodu koja objedinjava sva tri scenarija, samo je potrebno definirati da li ta metoda ovisi o nekom parametru komponente u kojoj je definirana [6].



Slika 10. React.js logo

React.js sintaksa

Primjer unosa biblioteke:

```
import React from 'react';
```

Primjer komponente razreda (Class component):

```
class Hello extends React.Component {
  render () {
    return <div className='message-box'>
      Hello {this.props.name}
    </div>
  }
}
```

Primjer funkcionalne komponente (Class component):

```
function MyComponent ({ name }) {
  return <div className='message-box'>
    Hello {name}
  </div>
}
```

Primjer korištenja stanja:

```
constructor(props) {
  super(props)
  this.state = { username: undefined }
}

this.setState({ username: 'rstacruz' })
```

Primjer korištenja svojstva komponente:

```
<Video fullscreen={true} autoplay={false} />

render () {
  this.props.fullscreen
  const { fullscreen, autoplay } = this.props}
```

Primjer korištenja kuke stanja:

```
import React, { useState } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
```

```
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Primjer korištenja kuke za efekte:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

    useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
)}
```

React Native

Ideja hibridnih aplikacija postoji već dugi niz godina i postojale su neke biblioteke i neka razvojna okruženja koja su omogućavala razvoj mobilnih aplikacija pisanjem izvornog koda u web tehnologijama, konkretno u JavaScriptu. Nažalost rijetko su se programeri usudili eksperimentirati s takvim alatima sve dok se 2015. godine nije pojavio React Native. Dodatna biblioteka koja radi u kombinaciji sa Reactom je zainteresirala širu JavaScript zajednicu i s obzirom na to da je React do tada poprimio poprilično dobru reputaciju nije postojalo toliko straha da se isproba nešto novo. Prednost koju je većina web programera vidjelo u React Nativu

jest mogućnost razvoja mobilnih aplikacija u programskom jeziku kojeg već dobro poznaju. Osim navedene prednosti dodatna prednost je bila što se pisanjem jednog izvornog koda pokrivalo razvoj aplikacije za čak tri platforme Android, iOS, Windows, a danas broji i četvrtu Symphony OS. Implementacija izvornog koda u React Nativu je identična onoj u React.js-u s nekoliko razlika. Ključne razlike su da React Native ne koristi iste elemente kao React, npr. dok u Reactu koristimo div u React Nativu se koristi View, a za elemente poput p,h1,h2 i sl. za prikaz teksta se koristi komponenta Text. React Native ima prednost pred drugim sličnim bibliotekama, a to je da može pristupiti matičnim modulima (native modules) poput kamere, bluetootha, nfc-a i cijelom skupu senzorike pametnih mobitela.

Osim razvoja mobilnih aplikacija, React Native je moguće koristiti za razvoj aplikacija za stolna i prijenosna računala, nosive uređaje poput pametnih satova i narukvica, pa čak i pametne televizije [7].

Pomoćne biblioteke

Vrijedi spomenuti neke od najpopularnijih pomoćnih biblioteka koje se mogu koristiti u bilo kojem JavaScript okruženju, ali su primarno dizajnirane za React.js. Neke od tih biblioteka služe za dohvaćanje podataka iz API-a, neke za jednostavnije kontroliranje stanja u aplikaciji, a druge služe za ubrzanu implementaciju i rješavanje složenih problema.

Apollo Client

Dohvat podataka u React.js aplikacijama se može izvršiti pomoću zadane funkcije fetch iz Web API-a čiji su obavezni argumenti krajnja točka na API i povratna funkcija koja vraća odgovor s poslužitelja. Cijela funkcija se mora odvijati asinkrono jer u suprotnom poziv na poslužitelja može potrajati duže nego što smo očekivali i cijela aplikacija se može zaustaviti. Iako je fetch funkcija efikasna i dovoljna za pozivanje operacija iz API-a, ako želimo implementirati u sustav GraphQL API-a možemo nepotrebno zakomplicirati cijeli izvorni kod, što bi moglo degradirati performanse aplikacija. Ako na poslužiteljskoj strani imamo GraphQL API najbolji način za pozivanje operacija je korištenje alata Apollo Client. Da bi se koristio Apollo Client u aplikaciji potrebno je instalirati paket @apollo-client, a zatim zamotati cijelu React aplikaciju konkretno početnu komponentu App s Apollo Client komponentom ApolloProvider koja prima svojstvo

client čija vrijednost mora biti krajnja točka na GraphQL API. Apollo Client se kroz kratko vrijeme puno puta izmijenio po pitanju arhitekture i sintakse i trenutno zahtjeva korištenje kuka. Tako imamo kuku za pozivanje upita useQuery i kuku za pozivanje mutacija useMutation. Navedene kuke primaju kao argumente upit napisan u graphql upitnom jeziku i povratnu funkciju koja vraća tri parametra. Prvi parametar je loading i jednak je istini dok se upit izvršava, drugi je error koji vraća graphql definiranu grešku s API-a ili mrežnu grešku ako dođe do takvih i treći parametar je data čija je vrijednost objekt koji sadrži zatražene podatke. Ako ne dođe do greške možemo pohraniti rezultate u stanje neke komponente, a zatim ih proslijediti u druge komponente i prikazati na korisničkom sučelju ili iskoristiti za neke druge operacije.

Redux

Veliki nedostatak u Reactu je upravljanje stanjem i najjednostavniji, ali i ne najbolji način prosljeđivanja nekog stanja iz komponente u komponentu je putem svojstva. Za hijerarhiju od dvije do tri komponente to nije problem, ali zamislimo kada bi trebali neko stanje provesti kroz pet ili deset komponenata. Kako bi se riješio takav problem dobro je koristiti neku biblioteku za upravljanje stanje, jedna od najpopularnijih je Redux biblioteka. Redux funkcionira tako da definira jedno globalno stanje kojem se može pristupiti iz bilo koje komponente neovisno o hijerarhiji [6].

React Router

Jedna od neophodnih biblioteka je React Router koja omogućava navigiranje između ekrana. Princip je sličan kao i kod drugih biblioteka. Prvo se mora zamotati cijela aplikacija tj. glavna komponenta aplikacije App sa zadanom komponentom iz biblioteke koja se zove Router. Nakon što smo zamotali cijelu aplikaciju s komponentom Router možemo definirati poveznice na željene komponente tj. ekrane pomoću komponente Link iz React Routera. Komponenta Link prima svojstvo to koje mora biti jednako putanji komponente kojoj želimo pristupiti. Unutar otvorene i zatvorene oznake komponente upisujemo naziv poveznice koji želimo prikazati korisniku. Kako nam putanje bez pripadajuće komponente neće previše pomoći moramo dodatno unutar Switch komponente definirati za svaku definiranu poveznicu jednu Route komponentu unutar koje postavljamo željenu komponentu kojoj želimo pristupiti. Kada ne bi koristili React

Router za navigaciju, mogli bi koristiti zadani HTML element a, ali bi se onda funkcionalno teže kretali između komponenata. Kada kažemo funkcionalno mislimo na način kretanja bez korištenja komponenata. Pomoću React Router kuka možemo u funkcionalnim komponentama razvijati akcije koje će se izvršiti prilikom nekog događaja u aplikaciji, npr. klikanjem na dugme ili prelaskom kursora preko nekog određenog elementa na sučelju.

Dodatne korisne biblioteke

Postoji veliki broj biblioteka za JavaScript i danas uz pomoć takvih biblioteka koje nisu uvijek nužno napisani u JavaScript programskom jeziku, već mogu biti napisane u Pythonu, C-u, Javi i drugim jezicima niže razine, ipak možemo koristiti JavaScript za raznovrsna područja u informacijskim tehnologijama.

- **Brain.js** - biblioteka za razvoj GPU ubrzane neuronske mreže
- **TensorFlow.js** - biblioteka za strojno učenje
- **Johnny-Five** - biblioteka za razvoj programa na području robotike i IoT-a
- **Babylon.js** - razvojno okruženje za razvoj kompjuterskih igara
- **D3.js** - biblioteka za vizualizaciju podataka
- **PixiJS** - biblioteka za 3d modeliranje

Povezivanje klijentskih i poslužiteljskih aplikacija

Sada kada smo upoznati s načinom rada klijentskih i poslužiteljskih aplikacija koje su napisane u JavaScriptu, možemo opisati način komunikacije istih i arhitekturu kojom se koriste. Generalno u razvoju modernih aplikacija najčešće se koristi MVP (Model View Controller[19]) arhitektura jer je njezin fokus na bazi podataka, a većina današnjih aplikacija koristi u svom toku podatke vezane uz poslovanje, korisnike, proizvode i sl. te je iz tog razloga MVP idealna arhitektura. Baza podataka, poslužiteljski servisi i klijentske aplikacije povezuju se tako da je poslužiteljski servis, najčešće API glavna poveznica. Poslužiteljski servis moramo povezati s bazom podataka.

[19] Erin Doherty, MVC Architecture in 5 minutes: a tutorial for beginners, https://www.educative.io/blog/mvc-tutorial, pristupljeno 29.11.2020

Poslužiteljske servise napisane u JavaScriptu najčešće povezujemo pomoću nekih gotovih biblioteka koje pružaju metode za kreiranje konekcije na bazu podataka, neki od popularnijih alata jesu Prism.io[20], Knex.js[21] (*Primjer koda za stvaranje konekcije na bazu podatka*) i Sequelize[22]. Kada smo uspostavili uspješnu vezu između baze podataka i poslužiteljskog servisa moramo uspostaviti i vezu između poslužiteljskog servisa i klijentskih aplikacija. Takvu komunikaciju možemo realizirati pomoću Ajax (Asynchronous JavaScript And XML[23]) http upita što je starija varijanta ili na moderniji način pomoću Web API metode fetch[24] (*Primjer korištenja metode fetch*). Osim navedenih načina isto možemo postići i pomoću nekih biblioteka za dohvaćanje podataka, a neke od najpopularnijih i najpouzdanijih biblioteka jesu SWR[25] i Apollo Client[26]. Nakon što smo uspješno uspostavili veze između baze podataka i poslužitelja te poslužitelja i klijentskih aplikacija (*Slika 11.*), tok komunikacije cijelog sustava je realiziran.

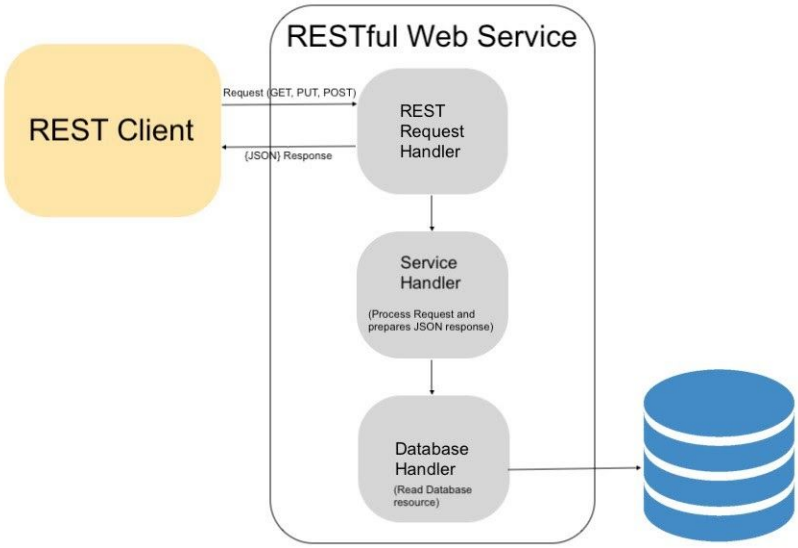
Primjer koda za stvaranje konekcije na bazu podatka:

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  }
});
```

Primjer korištenja metode fetch

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

[20] https://www.prisma.io/
[21] http://knexjs.org/
[22] https://sequelize.org/
[23] https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX
[24] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[25] https://swr.vercel.app/
[26] https://www.apollographql.com/



Slika 11. Prikaz komunikacije baze podataka, poslužitelja i klijentske aplikacije

Testiranje

Jedan od najvrjednijih procesa prilikom razvoja modernih aplikacija je testiranje. Segment testiranja je često zanemaren tako da se samo površno testiraju aplikacija od strane programera, a ako neka tvrtka ima djelatnike samo za testiranje, QA inženjere, često se upotrebljava samo koncept E2E (end to end) testiranja. E2E testiranje je kada netko iz razvojnog tima testira aplikaciju tako da ju koristi kako bi je i krajnji korisnik koristio. Osim takve verzije E2E testiranja, moguće je napisati automatske E2E testove. Postoji nekoliko vrsta testiranja kao što su E2E unit testovi, integracijski testovi i produkcijski testovi (*Primjer testa za provjeru vraćanja greške napisan u Jasmine okruženju*).

Najključniji testovi su E2E i unit testovi, dok E2E simulira ponašanje krajnjeg korisnika, unit testovi testiraju tj. provjeravaju da li neka funkcija vraća ispravan rezultat u nekom određenom scenariju korištenja aplikacije. Preporuka je da se uz E2E testiranje s ljudskim faktorom implementiraju i automatizirani E2E i unit testovi. Osim što je preporuka pisanje testova, postoji i princip razvoja aplikacija vođen testovima, što znači da se prvo pišu testovi za potencijalnu funkciju, a zatim funkcija koja mora za svaki dati test vratiti željeni rezultat. Zbog kompleksnosti

nekih testova i velikog broja linija koda koje je potrebno napisati, koriste se neka razvojna okruženja za testiranje poput MochaJS-a, Jesta, Jasmine, Karme i drugih [8].

MochaJS

MochaJS[27] je jedno od najstarijih i vrlo pouzdanih razvojnih okruženja za testiranje. Svoje operacije provodi putem Node.js-a, a moguće ga je koristiti na klijentskim i poslužiteljskim aplikacijama. Kako provodi testove u Node.js-u, odlično surađuje s programom za ispravljanje pogrešaka (*debugger*) i generira precizne izvještaje.

Jest

Najpopularnije razvojno okruženje za testiranje je Jest[28] koji je razvijen od tvrtke Facebook. Popularnost je stekao jer je primarno bio namijenjen za React.js biblioteku, ali je primjenjiv i na druge JavaScript biblioteke i razvojna okruženja. Jest je vrlo brzo razvojno okruženje s izuzetno dobrim performansama, dobro je dokumentiran i pruža mogućnost korištenja u svim JavaScript razvojnim okruženjima koja koriste Babel konfiguraciju poput Reacta, Angulara, Vue.js-a, Node.js-a i sl .

Primjer testa za provjeru vraćanja greške napisan u Jasmine okruženju:

```
it('should throw an expection for invalid selector', () => {
  const selector = 1;
  expect(() => domSelector(selector)).toThrowError(Error);
});
```

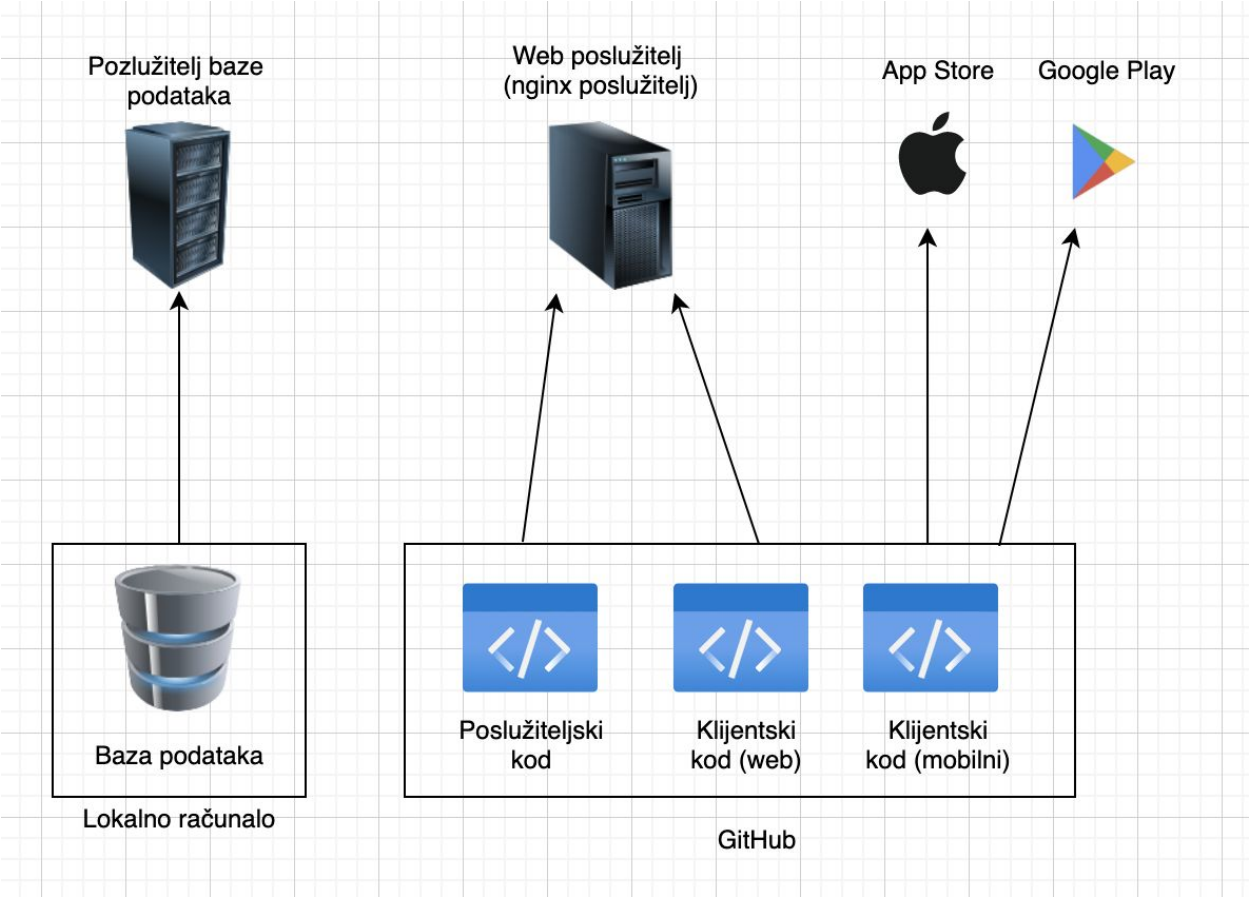
[27] https://mochajs.org/
[28] https://jestjs.io/

Podizanje sustava

Pod sustav se smatra cjelokupna aplikacija koja se može sastojati od baze podataka, poslužiteljskih servisa i klijentskih aplikacija. Sustav možemo podignuti na više različitih okruženja, a podignuti sustav se ovisno o okruženju može razlikovati u verziji. Najčešće razvojne agencije podižu sustav na tri različita okruženja. Postoji testno okruženje koje se nalazi na udaljenom poslužitelju i dostupno je samo razvojnim programerima, zatim se postavlja staging okruženje koje stoji na raspolaganju voditeljima projekta i klijentima i zadnje okruženje koje se podiže je produkcijsko. Prije podizanja sustava potrebno je podesiti okruženja prema kriterijima koje zahtjeva JavaScript aplikacija. Potrebno je prvo podignuti bazu podataka za koju je preporučljivo da se podiže na zaseban poslužitelj zbog zauzeća memorije i dodatne sigurnosti. Nakon uspješno podignute baze podataka, podižu se poslužiteljski servisi i klijentske aplikacije. Oboje se može nalaziti na istome poslužitelju, ali i ne mora. Kada je riječ o JavaScriptu najbolji izbor kreiranja softverskog poslužitelja je nginx poslužitelj unutar čije se konfiguracije mora podesiti na kojim ulazima će se pokretati poslužiteljski servis, a na kojem klijentska aplikacija. Unutar nginx konfiguracije se definira i domena putem koje će se moći pristupiti aplikaciji i servisima. Izvorni kod sustava se može ručno prenijeti na poslužitelje ili putem git alata dohvatiti iz repozitorija, što je i preporučeni način (*Slika 11.*). Nakon cijelog postupka potrebno je pokrenuti nginx poslužitelj, zatim poslužiteljske servise, a klijentsku aplikaciju se treba prvo "izgraditi" (build) i zatim pokrenuti uz pomoć pripadajućih skripti. Kako se gotovo uvijek mora s vremena na vrijeme ponovo pokrenuti poslužitelj zbog izmjena u izvornom kodu ili ako dođe do neke greške u sustavu ili samom poslužitelju, dobra ideja je postaviti alate za automatsko ponovno pokretanje sustava u slučaju greške i praćenje sustava. Jedan od najjednostavnijih u osnovnoj verziji besplatnih alata je PM2.

Kada se radi o mobilnim klijentskim aplikacijama proces podizanja je drugačiji. Mobilne aplikacije je potrebno podignuti na trgovine za aplikacije za svaku platformu zasebno, tako imamo Google Play za Android platformu, App Store za iOS platformu i App Gallery za Symphony OS. Pomoću zadanih naredbi koje pokrećemo unutar projekta kreiramo uvez aplikacije (app bundle) koji je potrebno podignuti na navedene trgovine putem koje krajnji

korisnici mogu preuzeti aplikaciju. U trgovinama aplikacija nije potrebno nikakvo dodatno konfiguriranje osim unosa osnovnih podataka o aplikaciji [9].



Slika 12. Shema procesa podizanja sustava

Dokumentacija razvoja i projekta

Ključan dio nakon svake završene faze razvoja, da li se radilo o krajnjoj fazi ili nekoj ranijoj je dokumentacija razvoja i projekta. Dokumentacija razvoja dokumentira tehničke aspekte sustava kao što su model baze podataka, model entiteta i veza, strukturu projekta, biblioteke koje se koriste, nejasne dijelove izvornog koda i sl., dok se dokumentacija projekta bavi dokumentiranjem sustava, najčešće klijentske aplikacije. Dokumentacija projekta služi kao uputstvo za krajnjeg korisnika i sadrži tko je sudjelovao na projektu, koliko je sati na kojem djelu sustava je utrošeno, kako se koristi sustav i druge korisne informacije.

Zaključak

S JavaScript programskim jezikom danas možemo razvijati moderne aplikacije za gotovo sve uređaje koje pokreću razni sustavi. Prednosti JavaScripta jesu jednostavna implementacija, krivulja učenja samog jezika je jednostavna, pokreće se na velikom broju uređaja i platforma i posjeduje jednu od najvećih zajednica u svijetu informacijskih tehnologija. Povijest JavaScripta vrlo interesantna gdje se od jednog primitivnog skriptnog jezika, napisanog u svega 10 dana došlo do najopširnijeg programskog jezika današnjice za kojeg postoji preko 3 tisuće raznovrsnih biblioteka i razvojnih okruženja.

Danas kad spomenemo modernu web ili mobilnu aplikaciju, možemo s velikom sigurnošću reći da je barem klijentska aplikacija razvijena u JavaScript programskom jeziku. Najčešći oblik implementacije JavaScripta na klijentskim aplikacijama je razvojno okruženje React.js i React Native, dok poslužiteljske servise implementiramo pomoću Node.js-a.

Projekt započinje od prvog razgovora s klijentom, a završava gotovo pa nikada osim ako se sustav ugasi. Konstanta unaprjeđenja, izmjene i ispravci su normalna pojava u razvoju mobilnih aplikacija, a tako je potrebna i konstanta edukacija tehničkog i ne tehničkog osoblja kako bi se postigli na postojećim i budućim projektima što bolji rezultati.

Iako će svaki entuzijast za nekim programskim jezikom tvrditi da je onaj jezik u kojem razvija jedan od najboljih, možemo se složiti da JavaScript zbog navedenih argumenata ima prednost nad većinom drugih programskih jezika. Svaki programski jezik ima svoje tzv. uspone i padove u popularnosti, pa tako i JavaScript kojem popularnost trenutno još uvijek raste, ali se usprkos tome ne smijemo zadržavati samo na njemu, već moramo proučavati svaki dan nove tehnologije koje će možda riješiti poteškoće s kojima se nose drugi programski jezici kao što je JavaScript.

Literatura

1. Ben Aston, “A brief history of JavaScript”, https://medium.com/@_benaston/lesson-1a-the-history-of-javascript-8c1ce3bffb17 (pristupljeno 08.07.2020.).

2. Marijn Haverbeke, *Eloquent JavaScript, 3rd edition*, No Starch Press, San Francisco 2018.

3. Alexander Zlatkov , “How JavaScript works: an overview of the engine, the runtime, and the call stack”, https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf (pristupljeno 11.08.2020).

4. Adrian Senecki, “GraphQL vs REST – the battle of API designs”, https://tsh.io/blog/graphql-vs-rest/ (pristupljeno 21.08.2020).

5. Jon Ducket, *HTML & CSS, Design and Build Websites*, John Wiley & Sons, Inc. Indianapolis 2011.

6. Alex Banks, Eve Porcello, *Learning React, Functional web development with React and Redux*, O’Reilly Media, Inc., Sebastopol 2017.

7. Bonnie Eisenman, *Learning React Native, Building native mobile apps with JavaScript*, O’Reilly Media, Inc., Sebastopol 2016.

8. Jash Unadkat, “Top 5 JavaScript Testing Frameworsk”, https://www.browserstack.com/guide/top-javascript-testing-frameworks (pristupljeno 12.11.2020).

9. Brad Traversy, “Node.js Deployment”, https://gist.github.com/bradtraversy/cd90d1ed3c462fe3bddd11bf8953a896 (pristupljeno 21.11.2020).

Popis slika

Popis priloga