Javascript na poslužiteljskoj strani

Kolarević, Davor

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku

Davor Kolarević

JavaScript na poslužiteljskoj strani

Diplomski rad

Osijek, 2018.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku

Davor Kolarević

JavaScript na poslužiteljskoj strani

Diplomski rad

Mentor: izv.prof.dr.sc. Domagoj Matijević

Osijek, 2018.

Sadržaj

1 Uvod

Zadatak ovog diplomskog rada sastoji se od dva dijela: praktičnog i teorijskog. Cilj praktičnog dijela je stvaranje poslužitelja Mathos Intranet aplikacije koja je izgrađena na klijent-posluželj modelu. Mathos Intranet je sustav za interno vođenje rada Odjela za matematiku u smislu administriranja programskog i izvedbenog plana nastave, djelatnika i rasporeda sati. Zbog opširnosti i kompleksnosti sustava ovaj rad će pokriti samo izradu modula za vođenje djelatnika, autentifikaciju korisnika unutar aplikacije, programskog plana i vođenje kolegija. Teorijski dio diplomskog rada pokriva tehnologiju korištenu pri izradi Mathos Intranet sustava.

U poglavlju *JavaScript* opisan je istoimeni programski jezik koji je korišten u praktičnom dijelu rada. Na što sažetiji način pokušavaju se objasniti specifičnosti jezika od tipova podataka, pa sve do složenijih dijelova, kao što su funkcijski prostor, proslijeđivanje podataka po vrijednosti, zatvarači i asinkrono izvršavanje zbog kojega je sam jezik specifičan.

Klijent-poslužitelj komunikacija poglavlje opisuje značajke klijent-poslužitelj modela. U kratkom dijelu opisuje se i HTTP protokol, razmjena poruke zahtjeva i odgovora te njihov izgled. Opisan je RESTful dizajn poslužitelja te koje su njegove značajke.

Poglavlje *Node.js* pokriva istoimenu platformu za izvršavanje JavaScript koda izvan web preglednika. Govoriti će se o osnovnim značajkama Node platforme, a poseban naglasak će se staviti na prednosti asinkronog izvršavanja kojeg koristi Node platforma. Node.js omogućuje pokretanje Mathos Intranet poslužitelja napisanog u JavaScript programskog jeziku.

Poglavlje *Express.js* ukratko opisuje JavaScript okvir (engl. *framework*) za izradu HTTP poslužitelja. Pomoću njega na jednostavan i brz način možemo izgraditi HTTP poslužitelj koji je skalabilan i jednostavan za održavanje.

Zadnje poglavlje, *Mathos Intranet*, pokriva izradu praktičnog dijela diplomskog rada, Mathos Intranet sustava.

2 JavaScript

2.1 Povijest

JavaScript[1] je programski jezik nastao 1995. godine u ranim počecima Web-a, a za njegov razvoj zaslužan je Brendan Eich i razvojna softverska tvrtka Netscape. No, jezik nije uvijek nosio ime koje ima danas. Originalno ime mu je bila Mocha, koje mu je dodijelio osnivač Netscapea Marc Andreessen. Iste godine ime mijenja još dva puta, prvo u LiveScript, a ubrzo potom u JavaScript. Njegovi tvorci su ga opisali kao objektno skriptni jezik koji se lako koristi, a čija je svrha stvaranje dinamičkih web aplikacija (prema članku [9]).

Prvotna namjena mu je bila da bude dio upravo Netscape-ovog web preglednika *Netscape Navigator*. Danas ga podržavaju svi moderni web preglednici te uz HTML i CSS čini neizostavni dio web programiranja. JavaScript je omogućio stvaranje bogatih i dinamičkih aplikacija kojima možemo pristupiti direktno iz web preglednika s bilo kojeg računala. Na taj način klasične aplikacije koje moramo instalirati na računalo su gotovo pa i nepotrebne.

Već 1996. JavaScript je predan ECMA organizaciji kako bi dizajnirali standardiziranu specifikaciju jezika na osnovu koje bi drugi dobavljači web preglednika mogli implementirati JavaScript podršku u svoje proizvode. Standardizirana verzija nosi naziv ECMAScript. Prva verzija standarda dovršena je 1997., druga verzija (kratice ES2) 1998., a treća 1999. godine. Dugo godina nakon toga nije bilo novih verzija standarda, sve do 2009. godine kada izlazi ES5 (razvoj 4. verzije je napušten zbog internih neslaganja), a ES6 izlazi 2015. koji je kasnije preimenovan u ECMAScript 2015 (ili ES2015). Već sljedeće godine dolazi ES2016., a zadnja verzija ES2017 je izdana sredinom 2017. godine.

2.2 Kompajler i interpreter

Da bi naučili razliku između kompajlera i interpretera moramo prvo znati kako računalo razumije samo binarni jezik, 0 i 1. Ljudima, tj. programerima je nezgodno, može se reći i gotovo nemoguće pisati programe na takav način. Zbog toga postoje programski jezici koji su skup predefiniranih pravila i sintakse kojima je programerima lakše manipulirati. Programski jezik na taj način predstavlja sloj između jezika kojima komuniciraju ljudi i jezika koji razumiju strojevi. Kako bi kod napisan u programskom jeziku, koji nazivamo izvorni kod (engl. source code), preveli u strojni jezik potreban nam je softver koji može obaviti

[1]Sam programski jezik nema nikakve veze s programski jezikom Javom, iako na prvi pogled djeluje drugačije. Ime JavaScript je dano iz marketinških razloga jer je u to vrijeme Java bila vrlo popularna

takav zadatak. Takav softver naziva se kompajler (engl. compiler). Kompajliranje izvornog koda potrebno je izvršiti samo jednom, a rezultat se pohranjuje na tvrdi disk računala.

Interpeter je također prevoditelj izvornog koda, no umjesto da prevede izvorni kod odjednom, on interpretira i izvršava svaku liniju izvornog koda kao jednu naredbu. Na kraju procesa prevedeni kod se ne sprema i pri svakom novom izvršavanju programa cijeli proces prevođenja izvornog koda se ponavlja. Interpeter je obično brži u obradi koda zbog toga što ima manje faza obrade. Kompajleri su kompliciraniji od interpretera zato što imaju više faza obrade izvornog koda i jer obrađuje cijeli izvorni kod odjednom.

Iako spada pod kategoriju dinamičkih interpeterskih jezika, JavaScript je također i kompajlerski jezik. Razlika je u tome što se u JavaScriptu kompajliranje koda ne događa prije izvršavanja cijelog programa, nego u samom trenutku izvršavanja (točnije nekoliko trenutaka prije). Takav koncept je poznat kao *just-in-time* (JIT) kompajliranje, a osim kod JavaScripta, koristi se i kod kompajliranja Java i .NET programskog koda. Kako je JavaScript pisan i koristi se prvenstveno za Web, svaki web preglednik ima u sebi ugrađen JavaScript interpeter.

Putem svakog web preglednika moguće je pristupiti konzoli u koju se može upisivati JavaScript kod i izvršavati direktno u pregledniku. Inače, kod koji pokreće web aplikacije dio je HTML dokumenata i upisuje se u posebne HTML tagove s nazivom <script>.

2.3 Prototip objekta

Objekti se kreiraju pomoću para otvorenih i zatvorenih vitičastih zagrada u koje upisujemo ključeve i pripadne vrijednosti ili korištenjem operatora *new* i pozivom funkcije *Object()*. Funkcijski poziv se u ovom načinu koristi kao konstruktor za kreiranje novog objekta (vidi [2] str. 117). Objekti kreirani korištenjem operatora *new* i pozivom funkcije konstruktora nasljeđuju ključeve i vrijednosti iz posebnog objekta koji se naziva *prototype*. Konstruktor funkcija se ni po čemu ne razlikuje od drugih funkcija u jeziku. „Konstruktor funkcija“ je zapravo samo naziv za funkciju koja će stvoriti novi objekt s definiranim ključevima. JavaScript u pozadini implicitno kreira novi objekti i proslijeđuje njegovu referencu (pod nazivom *this*) konstruktor funkciji.

```
function Car(doorsNumber, color) {
    this.doorsNumber = doorsNumber;
    this.color = color;
    this.engineStart = function() {
        console.log("Start.");
    }
}

var car = new Car(5, "blue");

car.__proto__ // -> vraca objekt Car
```

Kod 1: Konstruktor

Prototype objekt je moguće definirati na funkcijama i koristi se kao "nacrt" pri kreiranju novih objekata (vidi kod 2). Instance kreirane konstruktor funkcijama nemaju *prototype* ključ, ali imaju svoj prototip: objekt od kojega su naslijedili ključeve. Ukoliko želimo pristupiti prototipu objekta, njegovu referencu možemo dobiti preko ključa __proto__.

```
function Repository() {}; // konstruktor funkcija

Repository.prototype.create = function(obj) {
  console.log(object.name, ' inserted.');
}

var repository = new Repository(); // nova instanca

repository.create({ name: 'my_object'}); // -> 'my_object inserted'
```

Kod 2: Korištenje prototypea

2.4 Tipovi i vrijednosti

Svaki programski jezik temeljen je na osnovnim tipovima (vrijednostima) kojima je potrebno manipulirati. Osnovna karakteristika programskog jezika je skup tipova koje podržava. Tipovi u JavaScriptu se mogu podijeliti u dvije kategorije: *primitivne tipove* i *objekte*. Objekti

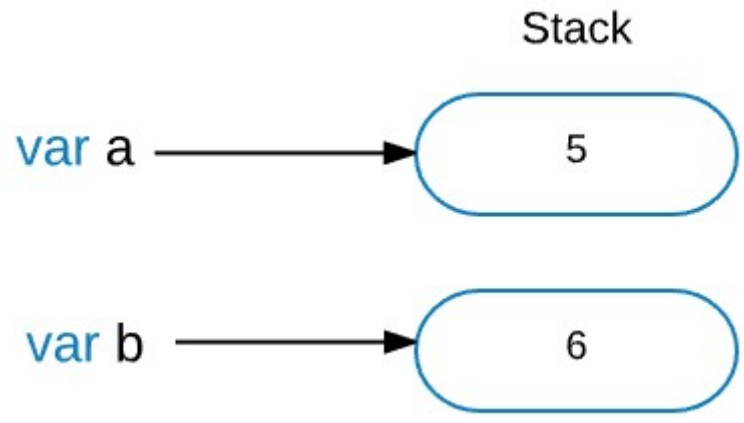
se često nazivaju i *referencnim tipovima* jer im se uvijek pristupa preko reference.

Primitivni tipovi uključuju brojeve (engl. *numbers*), nizove znakova (engl. *strings*) i logičke vrijednosti (engl. *boolean*). Posebni primitivni tipovi su *null* i *undefined*. *Null* označava poseban tip koji se koristi kako bi se označila odsutnost bilo kakve vrijednosti (vidi [2]). Također se može interpretirati kao poseban *objekt* koji govori „objekt ne postoji“.

Drugi tip koji opisuje odsustvo vrijednosti je *undefined*. To su vrijednosti varijabli koje još uvijek nisu inicijalizirane ili vrijednost koju dobijemo ako pokušamo dohvatiti vrijednosti objekta po ključu koji ne postoji. Funkcije koje ne vraćaju nikakvu vrijednost vraćaju *undefined*. Parametri koji nisu proslijeđeni funkcijama također su *undefined* tipa. Operator jednakosti (==) prezentira ih kao jednake, dok ih operator stroge jednakosti razlikuje (===). *Undefined* se treba shvatiti kao odsutnost vrijednosti izazvana greškom ili nekom neočekivanom radnjom, dok se *null* treba shvatiti kako normalna ili očekivana odsutnost vrijednost te je kao takva uvijek bolji izbor za korištenje.

2.4.1 Prosljeđivanje po vrijednosti ili referenci?

Kako je opisano u [7], četvrto poglavlje, svi *primitivni tipovi* spremaju se na posebno mjesto u memoriji Stog[2] (engl. Stack). *Referencni tipovi* (objekti) se spremaju na mjesto u memoriji koje se naziva hrpa (engl. Heap). Svakoj varijabli kojoj pridružimo primitivni tip, pridružujemo joj stvarnu vrijednost podatka (slika 1). U slučaju da primitivnu vrijednost dodijelimo sa jedne varijable na drugu varijablu, vrijednost koja se pohranjuje u varijablu je kopirana vrijednost (vidi programski kod pod 3) pa promjene na jednoj varijabli nisu vidljive na drugoj.



Slika 1: Dodijeljivanje primitivnih tipova varijabli

[2] *last in first out* struktura podatak

S druge strane, JavaScript nam ne dopušta da izravno pristupamo objektima u memoriji. Kada radimo s objektima zapravo radimo s referencama, „pokazivačima“ koji kao vrijednost sadrže adresu (lokaciju) tog objekta u memoriji računala. Varijabla čuva podatak o referenci koja je stvorena na Stogu, a referenca ”pokazuje” na pripadni objekt u memoriji.

```
1 var a = 5;
2
3 var b = a;
4
5 b = 6;
6
7 console.log(a); // -> 5
8
9 console.log(b); // -> 6
```

Kod 3: Pridruživanje primitivnih tipova

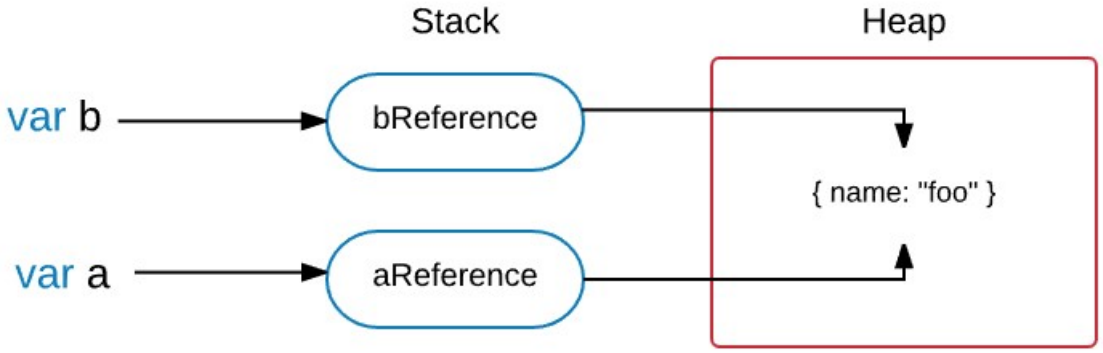
Kada radimo s objektima, referentnim tipovima, moramo biti oprezni jer vrlo lako možemo izazvati neželjene greške u programiranju. Ako novoj varijabli pridružimo već postojeći objekt, neće se stvoriti kopija objekta u memorija. Stvoriti će se pripadna referenca na Stogu koja će pokazivati na već postojeći objekt.

```
1 var a = { name: "foo" };
2
3 var b = a;
4
5 b.name = "bar";
6
7 console.log(a); // -> 'bar'
```

Kod 4: Pridruživanje reference

Na primjer, uzmimo da smo stvorili varijablu a za novostvoreni objekt. Za varijablu a je stvorena njezina referenca na stogu koja pokazuje na taj objekt. Nakon toga smo varijablu a pridružili varijabli b. Tada se za varijablu b stvara njezina referenca na Stogu, ali kako se radi o istom objektu u memoriji ona će pokazivati na isti objekt kao i referenca varijable a. Kada bi promijenili neku vrijednost ključa na objektu koristeći varijablu b, ta promjena

bi bila vidljiva i korištenjem varijable *a*. Ukoliko bi željeli stvoriti kopiju istog objekta morali bi koristi notaciju za kreiranje objekta (parovi vitičastih zagrada) i kopirati pripadne vrijednosti.



Slika 2: Reference u memoriji

Dakle, možemo zaključiti kako se vrijednosti u JavaScriptu prosljeđuju po **vrijednosti** bilo da se radi o primitivnim tipovima ili objektima. Kako se kod objekta stvara kopija reference, a ne kopija objekta, lako nas može navesti na krivi zaključak. Ista stvar vrijedi i za proslijeđivanje vrijednosti kao funkcijskim parametrima.

2.5 Prostor definiranja

U ostalim programskim jezicima svaki blok koda napisan unutar vitičastih zagrada ima svoj prostor (engl. *scope*) i varijable nisu vidljive izvan bloka koda unutar kojeg su definirane. To se naziva blok područje (engl. *block scope*). JavaScript s druge strane koristi funkcijski prostor (engl. *function scope*) koji kaže da su varijable vidljive unutar funkcije koje su definirane i unutar svake funkcije koja je ugniježđena unutar te funkcije. Drugim riječima svaka varijabla koja je definirana unutar funkcije, vidljiva je unutar cijelog blok koda definicije funkcije.

```
1 x = 5;
2 console.log(x); // -> 5
3 var x;
```

Kod 5: Uzdizanje

U JavaScriptu, varijable mogu biti deklarirane nakon što smo im pridružili neku vrijednost (Kod 5). To je zato što JavaScript engine kompajlira cijeli kod prije nego što ga interpetira

(vidi poglavlje 2.2), a kako je pronalazak i deklariranje svih varijabli unutar njihovih područja dio faze kompajliranja, kod iz primjera zapravo izgleda kao u primjeru (Kod 6). Kada u kodu napišemo izraz var x = 5; mi ga vidimo kao jedan izraz, a za JavaScript su to dva odvojena izraza: deklaracija (var x) i pridruživanje vrijednosti (x = 5).

```
1 var x;
2 x = 5;
3 console.log(x);
```

Kod 6: Uzdizanje 2

Ova paradigma u jezika naziva se uzdizanje (engl. *hoisting).* Uzdizanje je prirodno ponašanje JavaScript jezika gdje se sve deklaracije pomiču na vrh područja unutar kojeg su definirane. U drugom primjeru (Slika 7) na prvi pogled se čini da bi u retku 3 rezultat ispisa trebala biti vrijednost *„global“* koja je pridružena varijabli *scope* izvan funkcije. Međutim zbog uzdizanja, deklariranje varijable *scope* unutar područja funkcije *f* pomiče se na vrh bloka funkcije, pa globalna varijabla istog imena postaje nedostupna unutar funkcije.

```
1  var scope = "global";
2  function f() {
3      console.log(scope); // ispisuje "undefined"
4      var scope = "local";
5      console.log(scope); // ispisuje "local"
6  }
7
8  function f() {
9      var scope;
10     console.log(scope);
11     scope = "local";
12     console.log(scope);
13 }
```

Kod 7: Uzdizanje unutar funkcija

Varijable definirane izvan funkcijskog bloka su globalne varijable. Pretpostavimo da koristimo JavaScript modul koji želimo koristiti unutar više različitih programa. Kada koristimo takav modul ne možemo znati hoće li varijable unutar modula doći u konflikt s globalnim

varijablama koje kreiramo unutar programa. Ono što u JavaScriptu možemo napraviti je koristi funkcije za kreiranje posebnog prostora unutar kojeg više nećemo imati globalne varijable, nego lokalne. Pri kreiranje funkcije koristili smo anonimnu samopozivajuću funkciju. Samopozivajuća funkcija je funkcija koja se poziva odmah pri kreiranju. Takva praksa je česta u jeziku.

```
1 (function () {
2     // modul
3     var x = 5; // varijabla x je lokalna unutar modula
4
5     console.log(x); // ispisuje 5
6 })();
7
8 console.log(x); // ReferenceError
```

Kod 8: Funkcijski prostor

2.6 Zatvarači

JavaScript poput ostalih programskih jezika novije genaracije koristi leksički prostor definiranja (engl. *lexical scope*). To znači da pri pozivu, funkcija koristi varijable unutar prostora kojeg je definirana, neovisno o prostoru unutar kojeg je pozvana.

Kako bi to postigli, jezik interno svakom objektu funkcije dodjeljuje referencu na prostorni lanac. Svaki objekt ima pridružen njegov prostorni lanac. Prostorni lanac je niz objekata koji definiraju varijable unutar pripadnog prostora. Prema tome, svaku varijablu koju definiramo možemo shvatiti kao atribut nekog implicitnog objekta, objekta koji je dio prostornog lanca. Kada je pri pokretanju programa potrebno provjeriti vrijednost varijable, provjera počinje s prvim objektom u lancu. U slučaju da nije pronađena vrijednost pripadne varijable, provjerava se sljedeći objekt u lancu. (vidi [2]).

Kombinacija objekta funkcije i prostora unutar kojega se pripadne varijable funkcije rješavaju naziva se **zatvarač** (engl. Closure). Sve funkcije unutar jezika su zatvarači jer i objekti imaju pridružen prostorni lanac.

```
var scope = "global"; // globalna varijabla
function checkscope() {
    var scope = "local"; // lokalna varijabla
    function f() { return scope; }

    return f();
}

checkscope() // -> "local"
```

Kod 9: Zatvarači

Ako promotrimo kod 9 vidimo da funkcija *checkscope* definira lokalnu varijablu *scope* i poziva funkciju *f*.

```
var scope = "global";
function checkscope() {
    var scope = "local";
    function f() { return scope; }

    return f;
}

checkscope()(); // -> "local"
```

Kod 10: Zatvarači 2

Drugi primjer je malo drugačiji. Umjesto da funkcija *checkscope* pozove funkciju *f*, ona vraća objekt funkcije *f*. Pri pozivu funkcije *checkscope* stavljena su dva para zagrada. Jedan zbog poziva funkcije *checkscope*, a drugi zbog poziva funkcije f (koju vraća funkcija *checkscope*)! Možemo primjetiti da je funkcija f pozvana izvan prostora kojeg je i definirana. Dakle, za očekivati bi možda bilo da će u ovom slučaju funkcija f vratiti vrijednost *"global"*, no zbog leksičkog prostora funkcija f koristi varijable iz prostora unutar kojega je i definirana. U ovom slučaju je to prostor unutar funkcije *checkscope* pa će i vrijednost koju vraća biti *"local"*.

2.7 Asinkrono izvršavanje

Kada programiramo u JavaScript programskom jeziku vrlo često smo u situaciji da koristimo *asinkrone* (engl. *asynchronous*) funkcije. Često se pojam asinkronosti krivo poistovjećuje s pojmom *paralelnog izvršavanja*. Pojam asinkronosti veže se uz razliku između izvršavanja koda *sada* i *kasnije* gdje ne možemo točno sa sigurnošću reći kada će se *kasnije* izvršiti. Preciznije rečeno, programski kod koji se ne može izvršiti *sada* izvršiti će se **asinkrono**. S druge strane, *Paralelno izvršavanje* veže se uz izvršavanja više radnji istovremeno što se postiže koristeći *procese* ili *procesne niti* koje izvršavaju programski kod neovisni jedni o drugima.

Jedan od primjera *asinkronog* izvršavanja JavaScript koda su AJAX (*Asynchronous JavaScript And XML*) pozivi pomoću kojih u web pregledniku dohvaćamo podatke s poslužitelja bez da osvježimo cijelu web stranicu. Kada izvršavamo AJAX poziv koristimo *callback* funkciju u kojoj definiramo što treba učiniti kada odgovor od poslužitelja stigne. JavaScript kod će se nastaviti izvršavati, čeka se odgovor servera, a *callback* funkcija će se izvršiti u nekom trenutku *kasnije*. Detaljnije o asikronom izvođenju i što se točno događa u pozadini obrađeno je kasnije u radu u poglavlju 4.3.1.

```
1 // ajax funkcija je neko sucelje za slanje AJAX poziva
2
3 // izvrsava se 'sada'
4 ajax("...", function() {
5     // izvrsava se 'kasnije'
6 });
7 // izvrsava se 'sada'
```

Kod 11: Asikrono izvršavanje koristeći *callback* funkcije

Callback funkcije su osnovni dio *asinkronog* izvršavanja u JavaScript jeziku. Osim što njihovim korištenjem možemo zakomplicirati programski kod (tzv. *callback hell*) koristeći nekoliko ugniježđenih *callback* funkcija, veći je problem u tome što *callback* funkciju proslijeđujemo nekom drugom sučelju kojem dajemo kontrolu nad njezinim pozivanjem[3]. Nerijetko će to sučelje biti nešto što nismo sami napisali, već neki paket funkcionalnosti koji smo pronašli na internetu. Mogu nastati sljedeći problemi:

1. *callback* funkcija se može pozvati prerano jer ne možemo sa sigurnošću znati hoće li se

[3]dizajn princip u kojemu kontrolu nad izvršavanjem funkcija dajemo drugim sučeljima naziva se *Inversion of Control*

pozvati *asikrono* ili ne

2. *callback* funkcija može imati više parametara što povećava mogućnost da se neki od njih ne proslijedi

3. moguća greška koja se dogodi ostane "progutana"

Zbog ovih problema u ES2015 uveden je **Promise** objekt koji je puno bolji izbor pri *asikronim pozivima*.

2.7.1 Promise objekt

Promise objekti pružaju zanimljivo rješenje pri rješavanju *asikronih* poziva. *Promise* je JavaScript objekt koji predstavlja rezultat *asinkrone* operacije koja može završiti uspješno ili neuspješno. Drugim riječima, dok je izvršavanje asinkrone operacije u tijeku, njezinu buduću vrijednost predstavlja *Promise* objekt. Promise objekt je uvijek u jednom od tri stanja:

1. u čekanju

2. uspješno izvršen

3. neuspješno izvršen

Kada asinkrona operacija završi, razriješujemo *Promise* objekt, tj. zamijenjujemo ga za rezultat, a nakon toga se poziva jedna od odgovarajućih *callback* funkcija. Ukoliko je uspješno izvršen, pozvati će se prva funkcija, a ukoliko nije, druga funkcija koja predstavlja upravitelj grešaka. Iako se i ovdje koriste *callback* funkcije, *Promise* objekt nam daje dodatnu sigurnost. Ako se i dogodi da nije rezultat *asinkrone* funkcije, *Promise* objekt će uvijek biti razriješen *asinkrono* pa se ne može dogoditi da se razriješi prerano. Nadalje, *Promise* objekt će biti razriješen točno jednom. Ukoliko se dogodi bilo koja greška dok se čeka razrješenje, biti će pozvana *callback* funkcija koja upravlja greškama. *Callback* funkcije koje se koriste primaju točno jedan parametar.

Više o *asinkronom* izvršavanju u JavaScript jeziku i o *Promise* objektima opisano je u [5].

```
1
2  var isSuccess = true;
3
4  var p = new Promise((resolve, reject) => {
5      setTimeout(() => {
6          if (isSuccess) {
7              var data = {
8                  firstName: "John",
9                  lastName: "Doe",
10                 age: 38
11             };
12
13             resolve(data); // uspjeh
14         } else {
15             var error = new Error("Something went wrong.");
16
17             reject(error); // neuspjeh
18         }
19     }, 5000);
20 });
21
22 p.then(function(result) {
23     console.log(result); // uspjeh
24 }, function(result) {
25     console.log(result); // neuspjeh
26 });
```

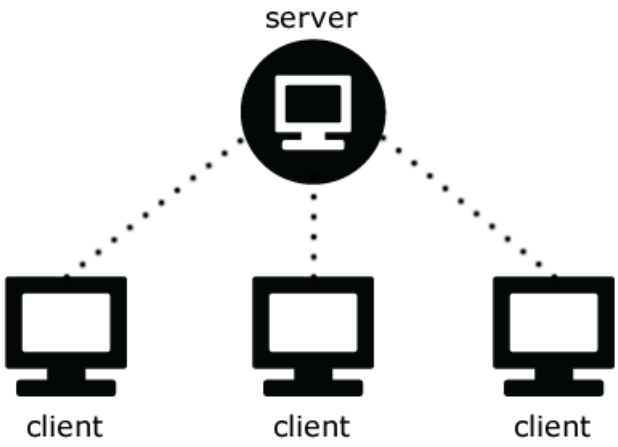
Kod 12: *Promise* objekt će biti razriješen *asinkrono*

3 Klijent-poslužitelj komunikacija

3.1 Uvod

U današnje vrijeme gotovo svi veliki sustavi su distribuirani, što znači da nekoliko udaljenih računala međusobno komunicira preko mreže koristeći neki od protokola za razmjenu podataka. Jedan od distribuiranih sustava je i klijent-poslužitelj model na temelju kojega se stvaraju i web aplikacije (što je bio i cilj ovog rada). Klijent i poslužitelj su dva međusobno odvojena programa koja komuniciraju preko mreže. Klijent je program koji se pokreće kod

krajnjeg korisnika, bilo kao web aplikacija u web pregledniku ili mobilna aplikacija na pametnim telefonima i tabletima. Dakako, klijent može biti i program bez grafičko korisničkog sučelja. S druge strane, poslužitelj je program koji je zadužen za posluživanje podataka klijentima.



Slika 3: Klijent-poslužitelj model

Bitno je napomenuti kako uvijek klijent prvi započinje (inicira) međusobnu komunikaciju. Poslužitelj je tu samo da odgovori na klijentov zahtjev i pošalje mu natrag neku informacija, te stoga uvijek mora biti dostupan i spreman pružiti uslugu klijentu.

Klijenata uvijek ima više. Svaki krajnji korisnik koji koristi web ili mobilnu aplikaciju predstavlja jednog klijenta. S druge strane, poslužitelj je samo jedan program, iako se veliki sustavi zbog potrebe da poslužuju veliki broj klijenata distribuiraju na više računala kako bi se raspodijelio mrežni promet. Također, u klijent-poslužitelj modelu dva klijenta neće nikada međusobno komunicirati direktno, nego samo sa poslužiteljom.

Kako bi klijent i poslužitelj mogli međusobno komunicirati potrebno je da razmjenjuju poruke preko mreže. No prvo nam je potreban protokol koji će definirati kako će biti strukturirane poruke koje se razmjenjuju. Upravo za to su nam potrebni **aplikacijski protokoli**. Aplikacijski protokoli definiraju tip poruke koje se razmjenjuje (npr. zahtjev ili odgovor), sintaksu poruke, tj. ključevi u poruci i njihovo značе, te što predstavljaju njima pridružene vrijednosti.

3.2 HTTP

HTTP (HyperText Transfer Protocol) je aplikacijski protokol za razmjenu podataka na *World Wide Web* mreži. Definira kako klijent zahtjeva podatke od servera, te kako poslužitelj

odgovara na takav zahtjev. HTTP je protokol bez stanja, tj. ne pamti nikakve informacije o klijentu.

RFC specifikacija o HTTP protokolu (RFC 1945, RFC 2616 i RFC 7230) definira i format HTTP poruka. Razlikujemo dva tipa poruka: HTTP zahtjev i HTTP odgovor.

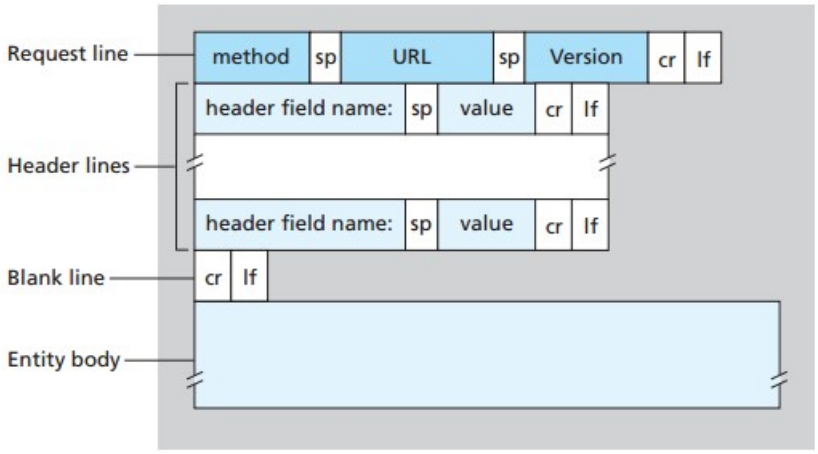
```
1 GET: index.php/nastava/upisi-na-studij HTTP/1.1
2 Host: www.mathos.unios.hr
3 Connection: close
4 User-agent: Mozilla/5.0
5 Accept-language: hr
```

Kod 13: Poruka HTTP zahtjeva

Klijent šalje HTTP zahtjev (primjer 13) poslužitelju u obliku poruke koja je napisana običnim jezikom koji je razumljiv svakom korisniku u ključ-vrijednost odnosu. Iako primjer sadrži samo nekoliko linija, HTTP zahtjev može ih imati više ili manje, ovisno o prirodi zahtjeva. Prva linija HTTP zahtjeva sadrži informaciju o HTTP metodi, URI te verziju protokola. Prva linija se naziva *linija zahtjeva*, a ostale linije čine *zaglavlje* zahtjeva.

Najčešće korištena HTTP metoda je GET. Primarno služi za dohvaćanje sadržaja sa raznih poslužitelja, a pristupamo im preko URI-a (engl. *Uniform Resource Identifier*). URI je također definiran RFC-om (točnije kao RFC 3986), a predstavlja niz znakova koji sadrži informaciju o adresi poslužitelja i putanji resursa koji može biti statički dokument (ali i ne mora). Resurs kojem pristupamo može biti bilo što od dokumenta, slike ili neke smislene informacije. Također je bitno da URI koji dohvaća neki resurs bude jedinstven do na poslužitelj, tj. svaki URI identificira jedinstveni resurs koji želimo dohvatiti sa servera. Poslužitelj je eksplicitno vidljiv kao vrijednost *Host* ključa HTTP zahtjeva. *Host* ključ je obavezan imati pridruženu vrijednost što je i intuitivno jer uvijek moramo znati adresu poslužitelja na koji šaljemo HTTP zahtjev.

Na slici 4 prikazan je generički izgled poruke HTTP zahtjeva. Ako se vratimo na prethodni primjer (primjer 13), vidimo da nedostaje tijelo zahtjeva (engl. *entity body*). U tijelu zahtjeva sadržani su podaci koje klijent šalje na poslužitelj i obično se koristi kod POST i PUT HTTP metoda. U tijelu zahtjeva mogu biti sadržani podaci od podataka prikupljenih kroz forme sve do cijelih datoteka. Ključ u liniji zahtjeva pod nazivom *Content-Type* sadrži informaciju o tipu podatka koji se šalje u tijelu.



Slika 4: generički izgled poruke http zahtjeva (izvor [3])

Ostale HTTP metode su sljedeće:

- **OPTIONS** metoda se koristi za dohvaćanje informacija o komunikaciji na relaciji klijent-poslužitelj. Pomoću ove metode možemo dohvatiti opcije ili zahtjeve poslužitelja s obzirom na neki resurs na serveru bez da iniciramo dohvaćanje resursa.

- **HEAD** metoda je slična GET metodi. Razlika je jedino što HEAD metoda ne smije nikada vratiti sadržaj u tijelu HTTP odgovora, ali informacije sadržane u zaglavlju moraju biti identične kao i kod GET metode. Ova metoda se najčešće koristi za testiranje ruta.

- **POST** metoda na poslužitelj šalje kompleksnije podatke (podatke iz forme ili datoteke) te pretpostavlja da će poslužitelj nad njima izvršiti određenu akciju (obrada i spremanje podataka) i stvoriti novi resurs.

- **PUT** metoda također kao i POST metoda na poslužitelj šalje podatke u tijelu poruke, ali pretpostavlja da se resurs na poslužitelju može identificirati URI-em zahtjeva. U tome slučaju podatak se smatra novom verzijom postojećeg resursa na poslužitelju te se očekuje da se nove promijene obrade i sačuvaju. Ukoliko se niti jedan resurs na poslužitelju ne može identificirati sa URI-em, u tome slučaju poslužitelj može stvoriti novi resurs koristeći poslane podatke.

- **DELETE** metoda zahtjeva od poslužitelja da obriše resurs koji se može identificirati URI-em zahtjeva. Valja napomenuti da klijent ne može garantirati da je resurs uspješno obrisan bez obzira na odgovor poslužitelja.

Nakon obrade HTTP zahtjeva, poslužitelj mora poslati odgovor klijentu.

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 31 Aug 2017 17:43:27 GMT
Server: Apache/2.4.10 (Debian)
Content-Length: 6821
Content-Type: text/html; charset=utf-8
```

Kod 14: Poruka HTTP odgovora

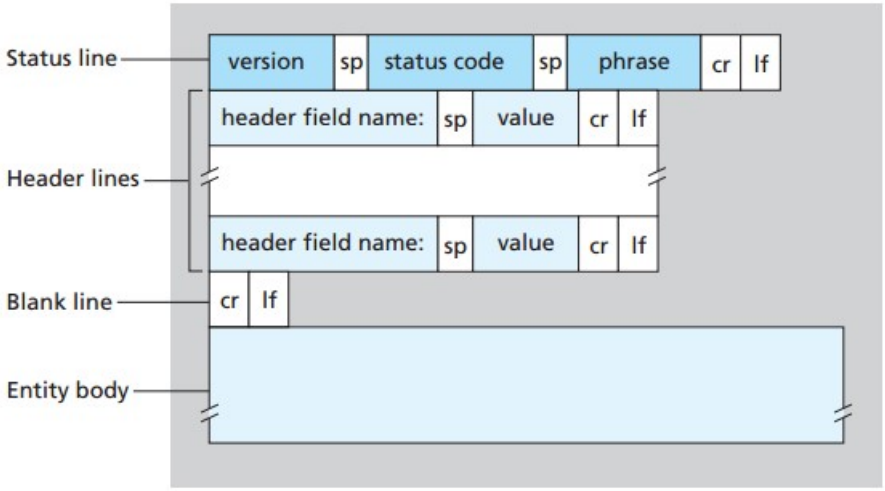
HTTP odgovor vrlo je sličan zahtjevu (vidi 14). Na isti način je uređen po ključ-vrijednost principu. Prva linija koja se naziva *linija statusa* sadrži informaciju o verziji protokola (u primjeru je to HTTP/1.1), broju statusa i odgovarajuće poruke statusa. Kod 200 znači da je poslužitelj uspješno odgovorio na HTTP zahtjev. HTTP statusom možemo na sistematičan način opisati svaku situaciju koja se dogodi u komunikaciji. Tako na primjer, kod 200 znači da je sve prošlo u redu, dok kod 500 znači da se dogodila neka iznenadna pogreška na poslužitelju i daljnje izvršavanje HTTP zahtjeva nije moguće. Kod 404 označava da poslužitelj ne može pronaći resurs koji je klijent zatražio, itd. U RFC 2616 dokumentu definirano je nekoliko HTTP statusa koji su sistematizirani u 5 kategorija:

1. 1xx **Informacije**: zahtjev je zaprimljen i obrada zahtjeva je u postupku
2. 2xx **Uspjeh**: zahtjev je uspješno zaprimljen, razumljiv poslužitelju i prihvaćen
3. 3xx **Preusmjeravanje**: Treba izvršiti dodatne akcije kako bi se HTTP zahtjev uspješno obradio
4. 4xx **Pogreška klijenta**: HTTP zahtjev ne sadrži valjane informacije za njegovu obradu
5. 5xx **Pogreška poslužitelja**: Poslužitelj nije u mogućnosti obraditi HTTP zahtjev

HTTP statuse moguće je proširit i nije potrebno da web aplikacije razumiju sve statuse. Bitno je samo da aplikacija razumije kategoriju statusa i nepoznati status može tretirati kao bilo koji drugi primljeni status iz iste kategorije. Na primjer, ako aplikacija zaprimi status 521, treba samo razumijeti da je došlo do pogreške na poslužitelju.

Zaglavlje HTTP odgovora čine ključ-vrijednost linije. Vrijednost *Connection* ključa govori hoće li poslužitelj zatvoriti vezu sa klijentom. Vrijednost *Date* ključa sadrži informaciju o datumu i vremenu kada je poslužitelj poslao HTTP odgovor. Vrijednost ključa *Server* sadrži

informaciju o poslužitelju koji je poslao HTTP odgovor. *Content-Lenght* i *Content-Type* sadrži informaciju veličini sadržaja odgovora u bajtovima, te tip podatka sadržanog u **tijelu** poruke. Detaljnije informacije o ključevima zaglavlja mogu se pronaći u RFC dokumentu 2616 u kojemu se definirane HTTP poruke. Generički izgled odgovora prikazan je na slici 5 i osim prve linije ne razlikuje se previše od poruke HTTP zahtjeva.



Slika 5: generički izgled poruke http odgovora (izvor [3])

3.3 RESTful API

Kada smo govorili o klijent-poslužitelj arhitekturi rekli smo da je riječ o distribuiranom sustavu i da obično jedan poslužitelj pruža usluge klijentima. Klijent može biti bilo koja mobilna, web ili desktop aplikacija koja koristi usluge poslužitelja. Dapače, ne mora uopće biti riječ o istim aplikacijama, ali sve one komuniciraju s poslužiteljem na isti način i po istim pravilima preko odgovarajućeg sučelja. To sučelje se naziva **Aplikacijsko programsko sučelje** (engl. *Application Programming Interface*) ili skraćeno **API**. API je jednostavno i jasno programsko sučelje pomoću kojega različiti programi ili programske komponente komuniciraju. API ne pruža korisničko sučelje, tj. sva komunikaciju se odvija ispod površine, nevidljivo oku korisnika aplikacije.

Reprezentacijsko stanje prijenosa (engl. Representational state transfer) ili skraćeno **REST**[4] vrsta je arhitekture distribuiranih sustava, točnije Web servisa. Ideja REST-a je pristup i manipulacija samim resursima na poslužitelju koristeći HTTP protokol (vidi 3.2). Zbog toga se kaže da je REST orijentiran prema resursima. REST arhitektura je bazirana na sljedećim pravilima:

[4]REST nije službeni standard, već skup principa koje je 2000. godine iznio Roy Fielding u svojem doktorskom radu [1]

1. **klijent-poslužitelj**: odvajanjem korisničkog sučelja od logike za rad i spremanje podataka omogućujemo implementiranje korisničkog sučelja na različite platforme i jednostavnije skaliranje poslužitelja. Nadalje, ovakva arhitektura omogućuje da se i klijent i poslužitelj razviju nezavisno jedan o drugom.

2. **uniformno sučelje**: sva sučelja trebaju biti standardizirana što pojednostavljuje komunikaciju između klijenta i poslužitelja, a servis koji svako sučelje pruža je nevidljiv klijentu. Standardizacija svih sučelja kao nedostatak donosi neučinkovitost sučelja zbog toga što se svaki resurs šalje na standardizirani način, a ne na način koji je specifičan za potrebe klijenta. Ovaj zahtjev uključuje sljedeće podzahtjeve:

   - Svaki resurs na poslužitelju mora biti moguće jedinstveno identificati (pomoću URI-a)
   - Klijentu je dozvoljeno da zahtjeva vrstu reprezentacije resursa koji je zatražio. Npr. isti resurs različitim klijentima može biti poslan u JSON ili XML formatu
   - U zahtjevu klijenta može biti definirano željeno stanje pojedinog resursa, a odgovor poslužitelja može sadržavati trenutno stanje resursa
   - Svaki resurs može sadržavati vezu prema drugim resursima

3. **bez stanja**: poslužitelj nije zadužen za čuvanje stanja o klijentima. Zbog toga svaki zahtjev koji klijent šalje poslužitelju mora sadržavati sve potrebne informacije da bi se on uspješno izvršio. Klijent je zadužen za vođenje komunikacije, a poslužitelj je tu samo da pruža informacije.

4. **predmemorija**: poslužitelj u odgovoru eksplicitno definira može li se resurs koji se šalje klijentu spremiti u predmemoriju klijenta. Korištenjem predmemorije smanjuje se konstantna (ponekad i nepotrebna) komunikacija između klijenta i poslužitelja. Korištenje predmemorije kao posljedicu može imati da klijent koristi resurse koji su možda tijekom vremena drastično izmjenjeni.

5. **slojevitost sustava**: arhitektura poslužitelja može biti posložena u hijerarhijske slojeve, tj. dodatan softver koji stoji iznad samog poslužitelja (npr. proxy server) ili podjela servisa na različite procese ili poslužitelje. Klijent ni u kojem trenutku ne treba znati komunicira li direktno sa poslužiteljem ili ne. Nedostatak je što se na taj način produžuje vrijeme čekanja odgovara kod klijenta.

6. **kod na zahtjev**: moguće je slanje klijentu programskog koda u obliku skripti koje proširuju funkcionalnost klijenta. Na taj način poslužitelj klijentu može poslati određenu programsku logiku samo onda kada mu je stvarno potrebna.

Narušavanjem nekog od navedenih uvjeta (1)-(5) smatra se da sučelje koje pruža poslužitelj nije **RESTful**. Zadnji uvjet (6), *kod na zahtjev*, smatra se opcionalnim te je dozvoljeno njegovo narušavanje. Razlog je što on pruža samo dodatnu korist klijentu, te se može dogoditi da neki od klijenata ne podržava ovaj uvjet (npr. u web pregledniku moguće je zabraniti izvršavanje JavaScript koda).

4 Node.js

4.1 Uvod

Dugo godina JavaScript se koristio samo u web preglednicima koji imaju ugrađen engine za izvršavanje JavaScript koda. No, sam jezik je kompletan i sposoban za puno ozbiljnije zadaće nego što ga se do tada koristilo (manipulacija DOM strukturom i animacije). Node.js (u nastavku Node) je JavaScript platforma koja omogućuje izvršavanje JavaScript koda izvan web preglednika. Kako bi mogli pokrenuti program izvan web preglednika potrebno je JavaScript kod komplajlirati i interpretirati. Node u tu svrhu koristi Googleov V8 engine za izvršavanje JavaScript koda. V8 je dio Googleovog web preglednika Chrome gdje se koristi upravo u tu svrhu.

Node kao i svaka druga platforma razvijen je po osnovnim načelima koja utječu na razvoj aplikacija. Jezgra platforme, tj. njezin skup funkcionalnosti je sveden na minimum. Sve ostalo je dostupno preko vanjskih paketa. Takav način razvoja je izazvao pozitivnu reakciju među programerima jer im omogućuje da sami eksperimentiraju s raznim modulima koji rješavaju određeni problem, umjesto da im je nametnuto jedno rješenje koje bi bilo dio jezgre.

4.2 Node moduli

JavaScript kod koji se izvršava u web preglednicima piše se u HTML dokumentu unutar <script> HTML oznaka. Jedna od najvećih mana jezika je dijeljenje jednog globalnog prostora definiranja između svih skripti. Drugim riječima, svaka varijabla stvorena unutar jednog bloka <script> tagova je vidljiva i unutar svakog drugog <script> bloka. Zbog toga se stvara potencijalni problem pisanja preko varijabli (tzv. *overwrite*) kada to ne želimo i uzrok može biti vrlo teško za otkriti pri pronalaženju grešaka.

Jedno od rješenja je korištenje funkcija za stvaranje zatvorenih prostora definiranja (poglavlje 2.5), no razvojem kompleksnih web aplikacija dolazi do pojave sustava koji je zadužen upravo za ubacivanje odvojenih JavaScript skripti[5] (modula). Takvi sustavi za upravljanje

[5] AMD i CommonJS

modulima omogućuju pisanje koda koji je lakši za održavanje, čitljivi i spreman za ponovnu upotrebu u drugim aplikacijama.

Moduli su blokovi programskog koda koji na Node platformi služe za strukturiranje aplikacija, tj. raspodjelu koda u odvojene nezavisne datoteke. Koristeći module možemo samo jedan dio funkcionalnosti otvoriti prema van, a ostatak funkcionalnosti može ostati sakriven.

Node za upravljanje modulim koristi *CommonJS* standard razvijen 2009. godine (iste godine kao Node) za standardiziranje upravljanja JavaScript modulima izvan web preglednika. *CommonJS* standard se temeljni na sljedećim točkama.

- Svaka datoteka (JavaScript skripta) je zaseban modul
- Svaka datoteka ima pristup vlastitoj definiciji modula preko *module* varijable
- Objekti koje želimo izložiti kada modul bude uključen u druge module definiran je u *module.exports* varijabli
- Za uključivanje drugih modula koristi se *require* funkcija

Node je izgrađen na temelju *CommonJS* standarda, ali također uvodi vlastita proširenja funkcionalnosti. Moduli se referenciraju ili po putanji do datoteke modula ili po imenu koji će se implicitno mapirati u putanju do datoteke ukoliko nije riječ o modulu iz standardne Node jezgre.

Više odvojenih modula koji spojeni u cjelinu obavljaju određenu zadaću čine paket. Osnovne meta informacije o paketu navedenu su u *package.json* datoteci. Node dolazi uz servis za distribuciju vanjskih paketa unutar programa. Taj servis se naziva NPM[6]. On nam olakšava dijeljenje već gotovih paketa koji rješavaju određeni problem. NPM dolazi uz alat za komandnu liniju pomoću kojega možemo dohvaćati pakete s njegovog servisa u oblaku. Sve informacije o vanjskim paketima koje koristimo također su navedene u *package.json* datoteci. Zbog toga se *package.json* datoteka može smatrati centralnim mjesto gdje su sadržane sve ključne informacije o našem Node programu, te ga lako možemo distribuirati kao paket na NPM servisu.

[6]skraćenica od *Node package manager*

```
// module_1.js

function private Function() {
    console.log("I'm private.");
}

function printA() {
    console.log("A");
}

function printB() {
    console.log("B");
}

module.exports.printA = printA;
module.exports.printB = printB;
```

Kod 15: Modul export

```
var module_1 = require('./module_1');

module_1.printA(); // -> A
module_1.printB(); // -> B

module_1.privateFunction(); // error
```

Kod 16: Modul import

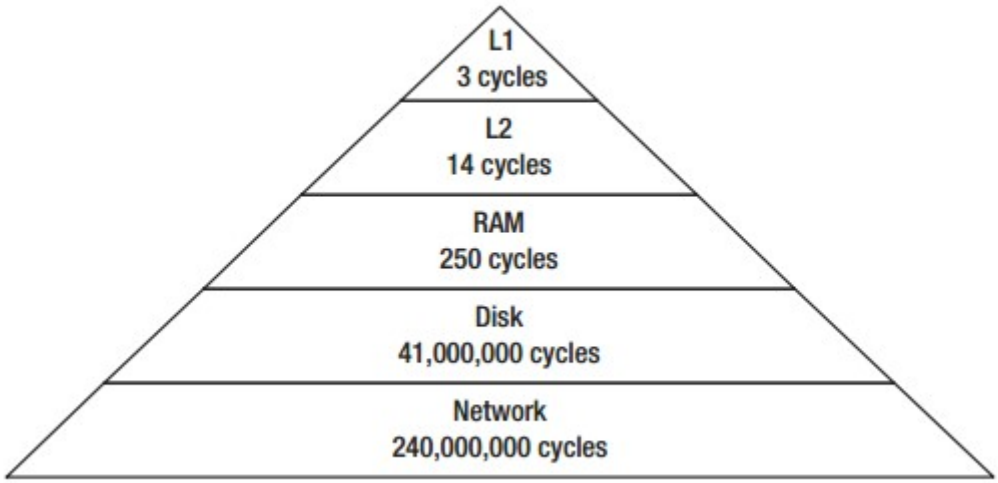
```
{
  "name": "mathosnet",
  "version": "0.8.3",
  "description": "Intranet application for department of mathematics",
  "main": "server.js",
  "dependencies": {
    "body-parser": "1.17.1",
    "express": "4.15.2",
    "inversify": "3.3.0",
    "jsonwebtoken": "7.3.0",
    "mysql": "2.11.1",
    "path": "0.12.7",
    "q": "1.5.0",
    "reflect-metadata": "0.1.10",
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start:dev": "set NODE_ENV=development&& node ./server.js",
    "start:prod": "set NODE_ENV=production&& node ./server.js",
  },
  "author": "mathos",
  "license": "ISC"
}
```

Kod 17: *package.json* datoteka korištena u Mathos Intranet projektu

4.3 Ulazno/izlazni problem

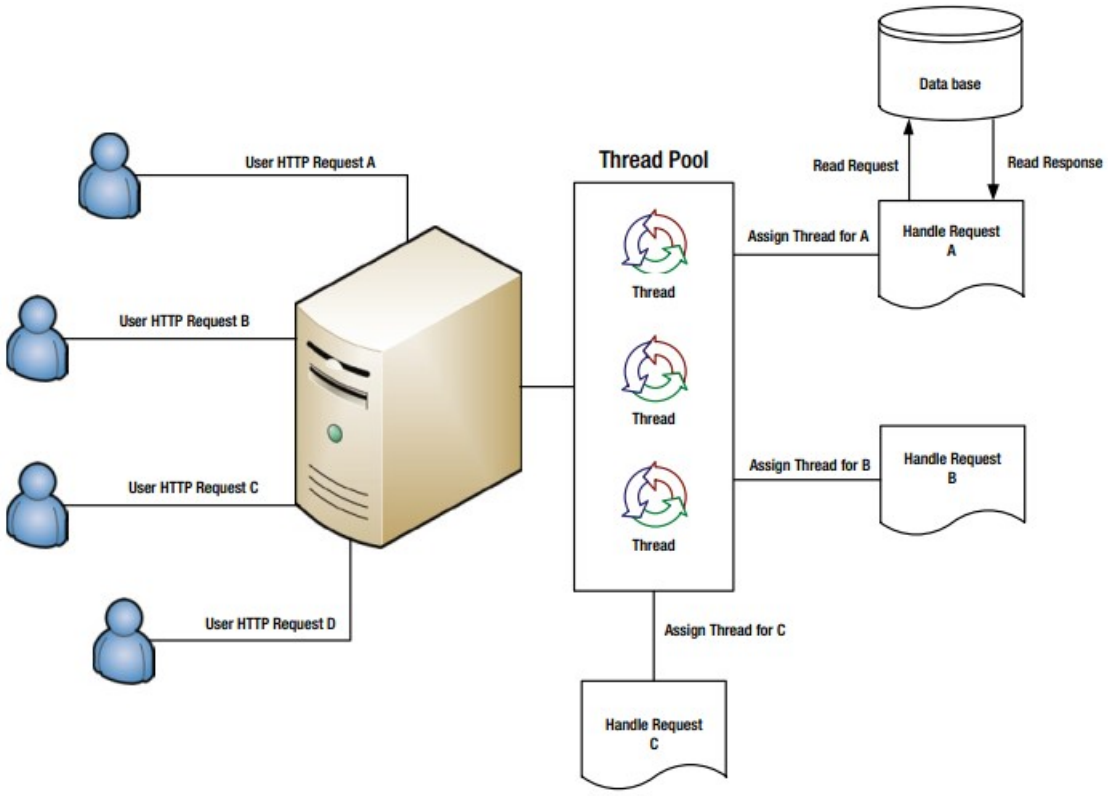
Korištenje ulazno/izlaznih operacija[7] može lako blokirati izvršavanje programa. Drugim riječima, sve dok npr. dohvaćanje nekog podatka iz baze podataka ne završi, program ne može nastaviti s izvršavanjem. Iz slike 6 vidljivo je kako je potrebno znatno više otkucaja procesora da se pristupi podatku pohranjenom na tvrdom disku ili mreži nego što je potrebno da se ista radnja obavi za podatke spremljene u cache memoriji ili radnom memoriji računala (RAM). Više otkucaja procesora znači i duže vrijeme čekanja da se akcija izvrši. Većina web aplikacija oslanja se upravo na rad s podacima pohranjenim na tvrdom disku ili preko mreže (npr. pristup bazi podataka na udaljenom računalu).

[7] čitanje i pisanje podataka na tvrdom disku, pristup bazi podataka, mreži



Slika 6: Brzine pristupa različitim memorijama u otkucajima procesora (izvor [6])

Da poslužitelj obradi pristigli HTTP zahtjev, koji većinom zahtjeva i pristup bazi podataka, potrebno je vremena i zahtjev će ostati otvoren sve dok pristup bazi podataka i njihova obrada ne završi. Takav postupak traži veliku potrošnju resursa računala (radne memorije i procesora). Tradicionalni poslužitelji problem obrade velikog broja ovakvih HTTP zahtjeva rješavaju na način da za svaki pristigli zahtjev otvaraju novi proces ili procesnu nit (engl. *thread*) zadužen za njegovu obradu. Pokretanje novog procesa je spor i skup postupak u okvirima radne memorije i korištenja procesora. Da bi doskočili problemu, moderni poslužitelji su počeli koristiti procesne niti iz nitnih bazena (engl. *thread pool*) koji održava već stvorene procesne niti. Kada HTTP zahtjev stigne na obradu, dodijelimo mu procesnu nit zaduženu za obradu (Slika 7). Procesna nit je rezervirana sve dok zahtjev nije obrađen i pri završetku vraćamo ju nazad u bazen. Kako ne moramo svaki put stvarati novi proces, ovakav način obrade HTTP zahtjeva je učinkovitiji od procesa i lakši u smislu potrošnje resursa radne memorije i procesora. Međutim, svaka procesna nit i dalje zahtjeva dio zauzeća memorije i procesora, pa korištenje više niti može s vremenom izazvati isti problem te ne predstavlja dugotrajno rješenje.



Slika 7: Obrada HTTP zahtjeva pomoću procesnih niti (izvor [6])

4.3.1 Bolji način?

U Nodu ovaj je problem riješen na sasvim drugačiji način. JavaScript po svojoj arhitekturi koristi samo jednu procesnu nit u kojoj se izvršava kod napisan od strane razvojnog programera. Međutim, okolina u kojoj se izvršava JavaScript kod pruža API za izvršavanje operacija koje su ulazno/izlazne u pozadinskoj programskoj niti. Bitno je napomenuti da taj API nije dio ECMAScript specifikacije. Npr. XmlHttpRequest za slanje AJAX poziva i setTimeout() funkcija dio su API-a koji pruža web preglednik, a u Nodu slične operacije omogućava C++ API. Kada operacija završi poziva se *callback* funkcija koju je prethodno potrebno definirati. Ovakav način programiranja naziva se **programiranje navođenje događajem** (engl. *event-driven programming*) gdje na neki događaj (završetak upita prema bazi podataka) reagiramo aktiviranjem funkcije koja se izvršava kao reakcija na taj događaj (callback funkcije).

U primjeru na slici 18 prvo kreiramo callback funkciju u kojoj definiramo što ćemo napraviti s podacima kada se upit prema bazi izvrši i prosljeđujemo ga *query* funkciji kao argument. Primjetimo da *query* funkcija neće vratiti rezultat, nego će ga proslijediti *callback* funkciji kada bude spreman. Sam programski jezik JavaScript zbog *zatvarača* (vidi poglavlje 2.6)

omogućuje da *callback* funkcija koja se prosljeđuje drugoj funkciji koristi varijable iz prostoru u kojem je definirana, a ne iz prostora u kojem je pozvana. Na taj način *callback* funkcije možemo prosljeđivati drugim funkcijama kao argument bez da brinemo o okruženju u kojem će se pozvati.

```
1  // callback funkcija
2  queryFinished = function (result) {
3      console.log(result);
4  }
5
6  db.query("SELECT * FROM posts WHERE id = 1", queryFinished);
7
8  console.log("Hello");
9
10 // redoslijed izvrsavanja:
11 // -> "Hello"
12 // ...
13 // => query result
```

Kod 18: Primjer upita prema bazi s *callback* funkcijom

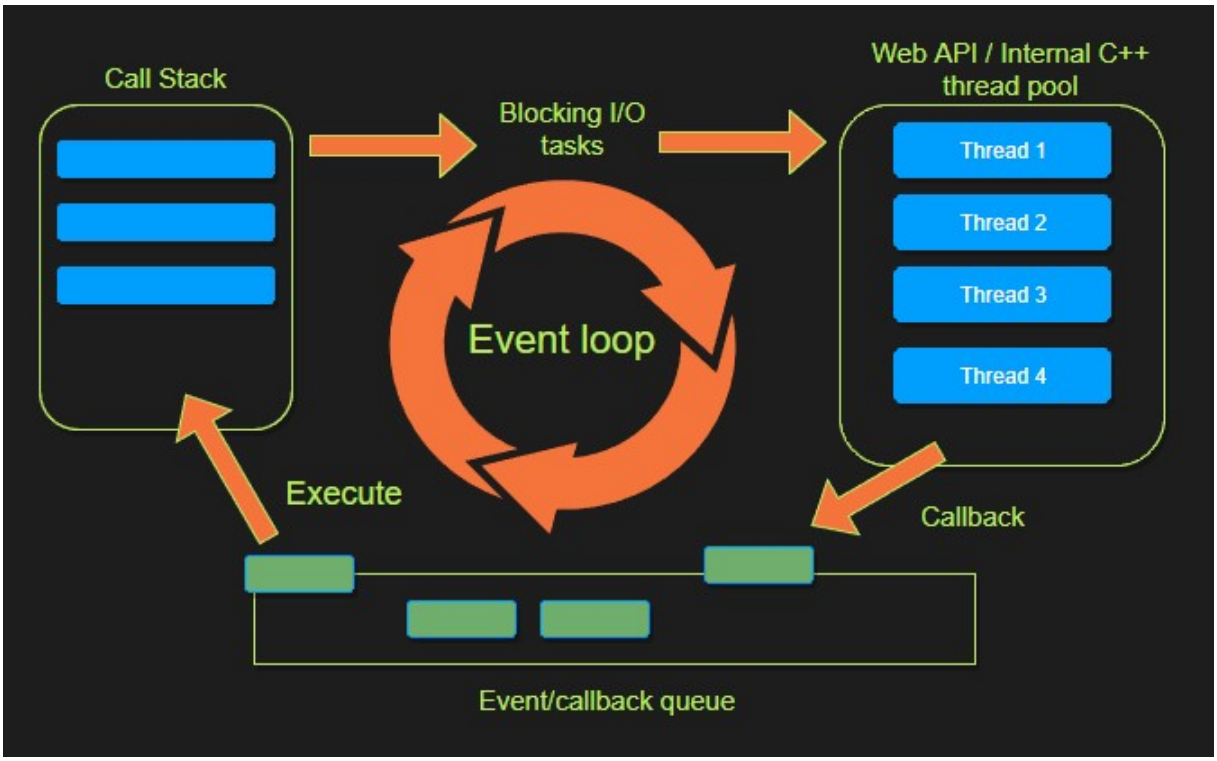
Kada pozovemo JavaScript funkciju ona se postavlja na takozvani *pozivni stog* (engl. *call stack*) gdje čeka da se izvrši. Kako se JavaScript kod izvršava u jednoj programskoj niti, funkcije će se izvršavati jedna po jedna. Svaka funkcija na pozivajućem stogu izvršava se dok ne završi radnja i ni jedna druga radnja ju ne može prekinuti ili biti pozvana između. Blokirajuće operacije, koje se izvršavaju putem API-a JavaScriptovog okruženja u pozadinskoj niti, nakon završetka *callback* funkciju postavljaju na *callback red* (engl. *callback queue*). Međutim, da bi se *callback* funkcija izvršila ona mora biti na *pozivnom stogu*. Kako onda proslijediti *callback* funkciju *pozivnom stogu* iz *callback reda*? Za to je zadužena *petlja događaja* (engl. *event loop*) koja ukoliko je *pozivni stog* prazan i nema niti jedne funkcije koja se mora izvršiti, provjerava i dohvaća prvu *callback* funkciju s *callback reda* i postavljaju za izvršavanje na *pozivni stog*. Korisnici i razvojni programeri ni u kojem trenutku ne mogu znati kada će se izvršiti *callback* funkcija. Ovakav način programiranja je poznat i pod nazivom **Asinkrono programiranje** (engl. *Asynchronous programming*).

Koraci izvršavanja:

1. klijent šalje HTTP zahtjev poslužitelju
2. iako petlja izvršavanja koristi samo jednu procesnu nit, Node interno održava bazen

procesnih niti

3. poslužitelj HTTP zahtjeve prosljeđuje u *callback red*
4. petlja izvršavanja u svakoj iteraciji provjerava ima li operacija u redu na čekanju za izvršavanje
5. petlja dohvaća HTTP zahtjev na obradu
   (a) započinje obrada zahtjeva
   (b) ako zahtjev ne sadrži blokirajuću ulazno/izlaznu operaciju (kao što je dohvaćanje podataka iz baze), zahtjev se obradi u glavnoj procesnoj niti, a u suprotnom radi sljedeće:
      i. pogledaj ima li dostupnih procesnih niti u bazenu
      ii. dohvati jednu nit i dodijeli joj obradu blokirajuće operacije
      iii. kada operacija završi, *callback* funkcija se proslijeđuje u *callback red* kojeg opet kupi petlja izvršavanja



Slika 8: Petlja izvršavanja

4.4 Performanse

Nakon što smo iznijeli cijelu teoriju o ulazno/izlaznom problemu i vidjeli kako je on riješen u Node-u, napraviti ćemo usporedbu performansi s poslužiteljem koji ne koristi asinkrone pozive izrađenim koristeći programski jezik Javom[8]. U primjeru, Java i Node poslužitelji obradi HTTP zahtjeva pristupaju na dva različita načina kako bi postigli konkurentnost (engl. *concurrency*). Korišteni web okvir za Java programskim jezik svakom pristiglom HTTP zahtjevu dodijeli njegovu procesnu nit. S druge strane, Node koristi asinkrone pozive (o čemu smo govorili u prošlom poglavlju). Svi rezultati izraženi su kao broj obrađenih HTTP zahtjeva u sekundi (r/s).

4.4.1 Brzina izračuna

U prvom testu ćemo usporediti brzinu izračuna koji nemaju nikakve blokirajuće ulazno/izlazne operacije. Zadatak je bio pronaći 500 prvih prostih brojeva. Prvi primjer je koristio samo 1 procesnu nit i 1 proces, a u drugom je korišteno 70 procesnih niti za izračune u Javi, i 8 procesa za izračune u Node-u[9].

jezik	1 procesna nit/proces	70 procesnih niti/8 procesa
Java	181 r/s	698 r/s
Node	86 r/s	379 r/s

4.4.2 Učitavanje datoteka

U drugom primjeru bilo je potrebno pri HTTP zahtjevu učitati datoteku s tvrdog diska veličine 5 MB. Ranije smo već spomenuli kako je ovakva operacija opterećenje za procesor (slika 6).

jezik	1 procesna nit/proces	70 procesnih niti/8 procesa
Java	37 r/s	41 r/s
Node	127 r/s	350 r/s

4.4.3 Upit prema bazi

U posljednjem je primjeru bilo još potrebno napraviti duži upit prema bazi podataka.

jezik	70 procesnih niti/8 procesa
Java	6 r/s
Node	64 r/s

[8]Izvor testa je dan na [11]

[9]Node konkurentnost može postići klasteriranjem poslužitelja na više procesa vidi [6] str. 257

4.5 Zaključak

Iz rezultata vidimo da je Java poslužitelj puno brži u operacijama koje ne zahtjevaju ulazno/izlazne operacije. To je ipak zbog činjenice što je Java kompajlerski jezik i sam kod se izvršava puno brže. Kako web aplikacije većinom zahtjevaju ulazno/izlazne operacije pri gotovo svakom HTTP zahtjevu, puno zanimljivi je drugi slučaj kada je bilo potrebno učitati datoteku. Tada se Node poslužitelj iskazao kao puno brži (čak 8 puta pri uvođenju konkurentnog izvršavanja).

U još jednog test primjeru u kojem je bilo potrebno izvršiti duži upit prema bazi podataka (što je također ulazno/izlazna operacija), Node poslužitelj je bio brži čak **10 puta**.

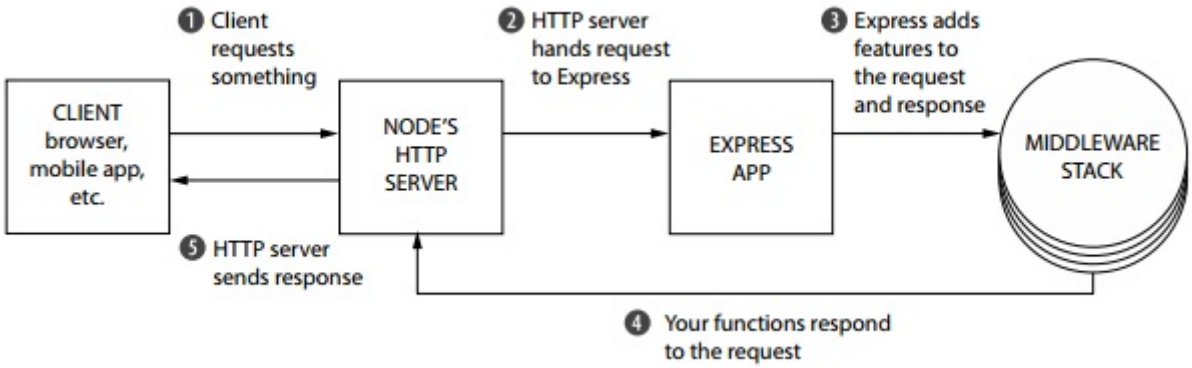
Razlog je što web okvir za Java programski jezik svakom novom HTTP zahtjevu dodijeljuju procesnu nit. Pri porastu velikog broja HTTP zahtjeva broj zauzetih procesnih niti raste, a procesor je više zauzet promjenom procesnih niti nego onim što treba napraviti. Node poslužitelj je ovdje u prednosti jer koristi jednu procesnu nit. Dok se blokirajuće operacije rješavaju u pozadinskim procesnim nitima, glavna procesna niti može nastaviti raditi što već treba, te je na taj način smanjena opterećenost procesora.

5 Express.js

Express.js (dalje samo Express) je web okvir (engl. *web framework*) za izradu poslužiteljskog dijela web aplikacija. Okvir čini pojednostavljeni sloj koji stoji iznad *http* modula iz Node jezgre. Express je dostupan kao NPM paket i moguće ga je uključiti u vlastiti projekt kao ovisnost.

Stvaranje poslužitelja koristeći samo *http* modul iz Node jezgre može se zakomplicirati i rezultirati aplikacijom koju je dalje teško održavati. Koristeći samo *http* modul, koristimo jednu funkciju za procesiranje svakog HTTP zahtjeva. Ta funkcija naziva se *upravljač zahtjevima* (engl. *request handler*) i kao parametre prima dva JavaScript objekta koji predstavljaju HTTP zahtjev i HTTP odgovor. Express nam donosi sljedeća poboljšanja:

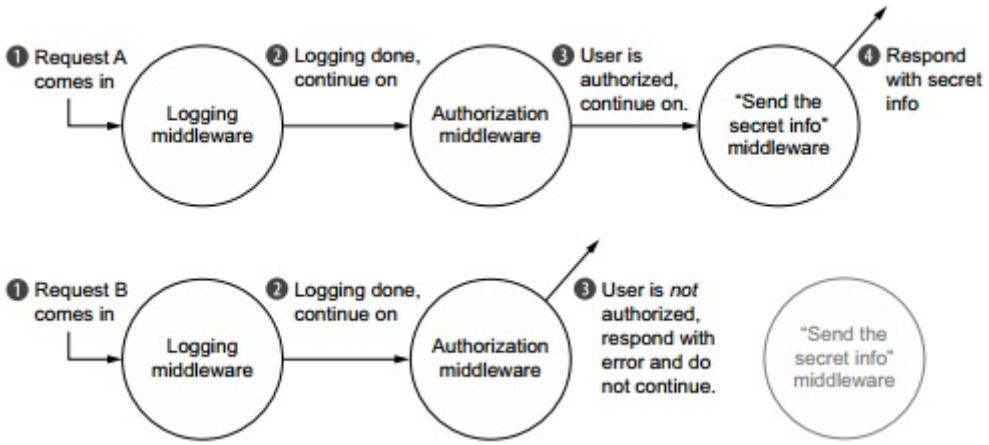
- donosi proširenje funkcionalnosti i stvara sloj iznad postojećeg Node *http* modula te na taj način sakriva kompleksne dijelove implementacije
- umjesto jedne funkcije za procesiranje HTTP zahtjeva, za svaki HTTP zahtjev možemo koristiti posebnu funkciju koja se u Express okviru naziva *middleware* funkcija



Slika 9: Tok HTTP zahtjeva i odgovora koristeći Express (izvor [4])

5.1 Middleware

Middleware[10] je ništa drugo nego funkcija koja za parametre prima JavaScript objekte koji predstavljaju HTTP zahtjev i odgovor koji se šalje natrag klijentu, te *callback* funkciju koja predstavlja sljedeću funkciju u lancu. Svaki HTTP zahtjev koji stigne na poslužitelj prolazi *middleware* lancem od vrha prema dolje. Ako želimo pristigli HTTP zahtjev proslijediti sljedećoj funkciji u lancu pozvati ćemo funkciju *next()*, u suprotnom zatvaramo zahtjev (šaljemo odgovarajući odgovor klijentu).



Slika 10: Tok HTTP zahtjeva kroz *middleware* lanac (izvor [4])

Na primjer, uobičajena je praksa korištenja *middleware* funkcije koja će dokumentirati vrijeme i url svakog HTTP zahtjeva koji stigne na poslužitelj (kod 19). Takva funkcija stoji na

[10]Riječ *middleware* nije nova. U softverskom svijetu predstavlja sloj apstrakcije između aplikacijskog sloja i operacijskog sustava pružajući odgovarajuće servise

vrhu lanca jer treba procesirati svaki pristigli HTTP zahtjev. Sljedeća *middleware* funkcija u lancu je obično *middleware* za provjeru korisnikovih privilegija što ima smisla jer nema potrebe da uopće obrađujemo HTTP zahtjev koji dohvaća povjerljive podatke korisniku koji uopće nema prava da im pristupi. Na *middleware* funkcije možemo dodati i prefiks putanje ili samo određene HTTP metode pa takva *middleware* funkcija može obraditi samo HTTP zahtjev koji odgovara dodatnim ograničenjima.

```
1 var app = express();
2
3 app.use(function (req, res, next) {
4     console.log("%s %s - %s", (new Date()).toString(), req.method, req.url);
5
6     return next(); // skoci u sljedecu middleware funkciju u lancu
7 });
```

Kod 19: *middleware* funkcija za dokumentiranje HTTP zahtjeva

Naposlijetku, jedna od *middleware* funkcija će i odgovoriti klijentu pozivom odgovarajuće metode (*send()*, *sendFile()* ili samo *end()*) na *res* objektu. Pri tome treba biti oprezan jer slanje odgovora više od jednom za svaki pojedini HTTP zahtjev rezultirati će terminalnom greškom u kodu.

Još jedna stvar koju je bitno za napomenuti je da je redoslijed *middleware* funkcija primjenjenih u *app.use()* bitan! Drugim riječima, redoslijed kojim će se odgovarajuće *middleware* funkcije pozivati ovisi o njihovoj poziciji u kodu. Tada ako želimo da se *middleware* funkcija za dokumentiranje svakog HTTP zahtjeva izvrši prije *middleware* funkcije za provjeru privilegija, funkcija za dokumentiranje mora biti iznad nje i u kodu!

5.2 Usporedba s ostalim web okvirima i performanse

Express nije jedini web okvir za Node ali je svakako najpopularniji[11]. Od ostalih web okvira za pisanje poslužitelja za Node platformu među poznatijima su Hapi.js, Restify, Total i Koa (razvijen od strane tima koji je bio zadužen i za Express). Među njima svoje mjesto traže i dva relativno nova web okvira, Sails i Adonis. Pogledati ćemo usporedbu performansi Express web okvira i ostalih sličnih okvira za Node[12]. Također ćemo u usporedbu uključiti i *http* iz Node jezgre. Test je proveden tako da su na poslužitelj slani HTTP zahtjevi, a mjerilo

[11]Prema broju preuzimanja na NPM servisu za kolovoz 2017.

[12]Izvor testa je dan na [10]

se koliko zahtjeva poslužitelj obradi u sekundi. Istovremeno je slano 100 HTTP zahtjeva u minuti dok nije dosegnut broj od 50 000 zahtjeva ili je prošlo 20 sekundi od početka testa. Poslužitelj je slao jednostavni odgovor, i za ozbiljnije testove trebalo bi napraviti poslužitelj koji se koristi za stvarne probleme. Test je proveden na dva okruženja: Linux sustavu pokrenutom na Windows operacijskom sustavu i virtualnom stroju iz donjeg cjenovnog razreda slabijih performansi.

Web okvir	Lokalno računalo	Udaljeno računalo
Expres.js	1745 r/s	2875 r/s
Hapi.js	1094 r/s	688 r/s
Node	2291 r/s	5092 r/s
Restify	1759 r/s	2380 r/s
Koa2	1887 r/s	3317 r/s
Total	2144 r/s	3924 r/s
Sails	1554 r/s	772 r/s
Adonis	2177 r/s	962 r/s

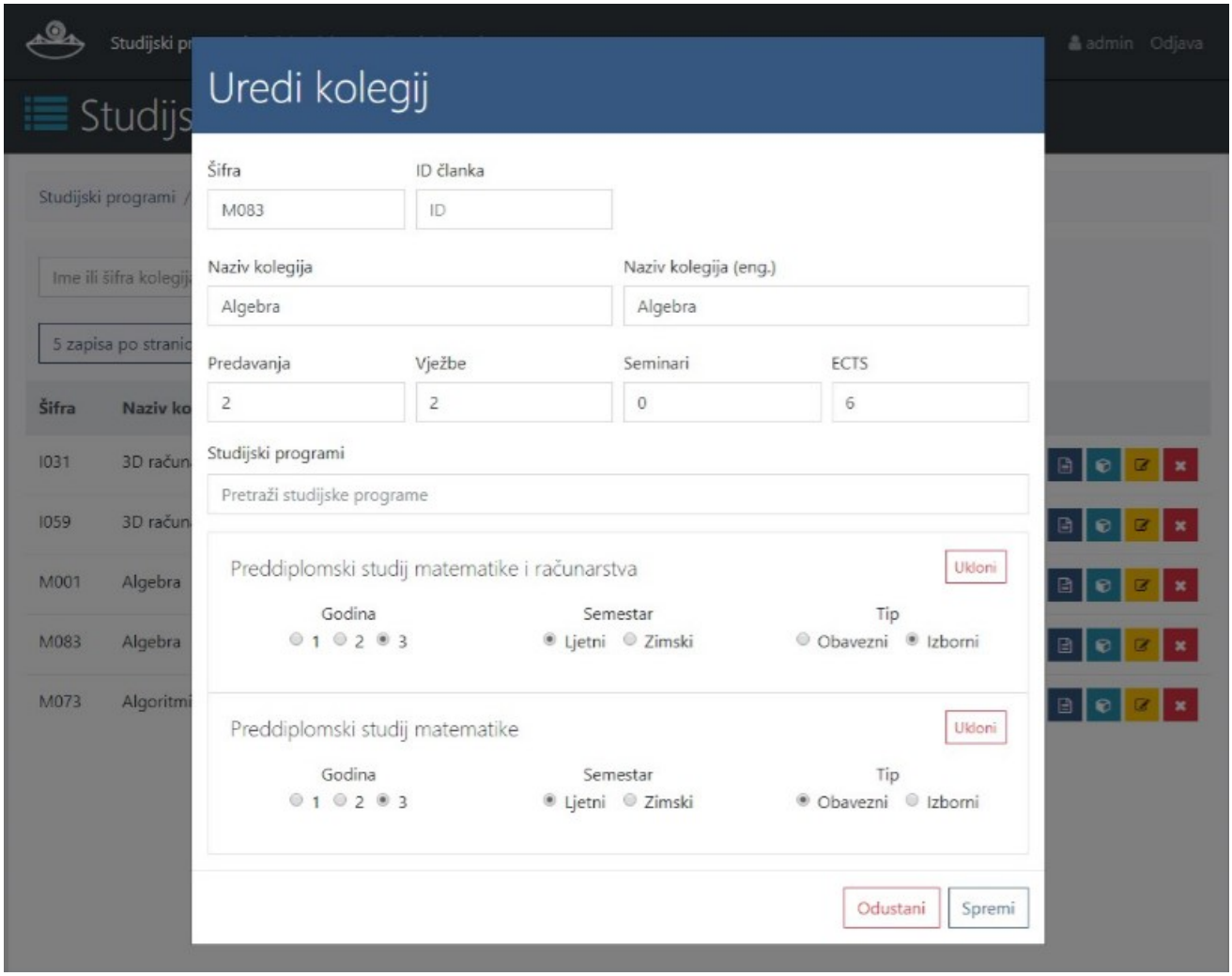
Ono što možemo primjetiti iz provedenog testa je da jezgreni *http* modul postiže najbolje rezultate od svih web okvira što nije iznenađujuće jer se koristi čisti modul iz jezgre, a web okviri su inače nadogranja. Koristeći samo *http* modul iz Node jezgre povlači sa sobom i kompleksnosti koda koje on donosi. Od web okvira kao najbolji se pokazao Total, a Express.js koji smo pokrili u ovom radu nalazi se na diobi 4 i 5 mjesta. Dakako treba napomenuti kako ovo nije stvarno okruženje te za stvarne testove bi bilo potrebno napisati kompleksnije poslužitelje kakvi se inače koriste za posluživanje podataka. Razlika između rezultata na različitim okruženjima je rezultat konfiguracija različitih performansi.

6 Mathos Intranet

6.1 Uvod

Mathos Intranet je web aplikacija čija je svrha vođenje svakodnevnog rada Odjela za matematiku. Sastoji se tri modula: studijski programi, djelatnici i izvedbeni plan. Modul studijskih programa ima svrhu administriranja studijskih programa koji se izvode ili će se možda izvoditi u budućnosti na Odjelu za matematiku. Izvedbeni plan generira plan koji se izvodi tijekom akademske godine. Djelatnici, koji čine i najkompleksniji modul unutar aplikacije, služi za administriranje nastavnog i nenastavnog osoblja odjela. Aplikacija nema sistem vanjske registracije korisnika, nego se oni dodaju interno unutar aplikacije od strane postojećih korisnika (administratora).

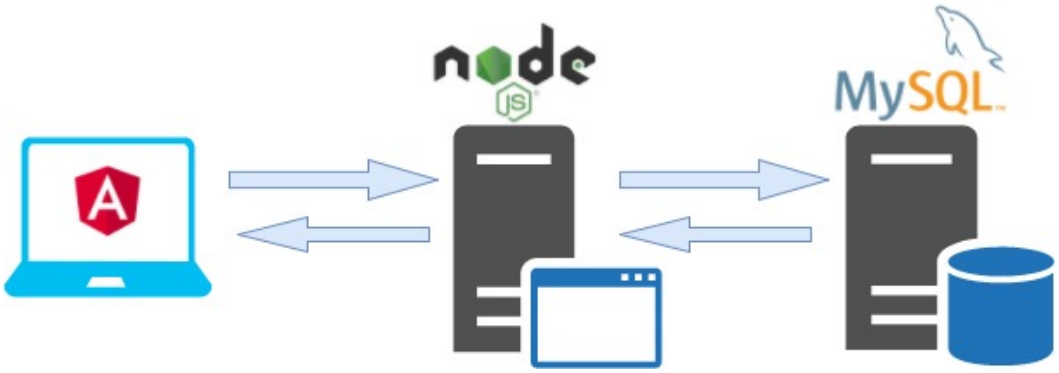
Klijentski dio aplikacije izgrađen je koristeći Angular platformu. Angular nam omogućava brzo i jednostavno programiranje kompleksnih klijentskih aplikacija koristeći JavaScript programski jezik.



Slika 11: grafičko sučelje klijentskog dijela Mathos Intranet aplikacije

Poslužiteljski dio Mathos Intranet aplikacije, što je i tema ovog rada, izgrađen je koristeći JavaScript programski jezik, Node.js platformu za pokretanje JavaScript koda, te Express.js web okvira za HTTP poslužitelj. Podaci se spremaju u MySQL bazu podataka, koja za potrebe ovog projekta sadrži 58 tablica (entiteti unutar baze u koje se spremaju potrebni podaci). MySQL je relacijska baza podataka unutar koje se podaci spremaju u tablice, gdje pojedina tablica može imati vezu na neku drugu tablicu. Te veze zovu se relacije.

Cijela aplikacija uređena je u tri reda, tzv. *3-tier* arhitektura. Definira ju odvajanje različitih funkcionalnosti aplikacije na tri fizički odvojena računala koji obavljaju ulogu: *prezentacije*, *logike* i *upravljanje podacima*.



Slika 12: *3-tier* arhitektura

Prezentacijsku logiku obavlja klijentski dio aplikacije (Angular aplikacija). Iako je tijekom dugo godina bilo uobičajeno da prezentacijski dio obavlja ulogu samo prezentiranja podataka korisniku, korištenjem Angular platforme na efikasan način dio logike aplikacije se može prebaciti iz drugog reda u prezentacijski red.

Za *logiku* aplikacije zadužen je poslužitelj. Njegova je uloga dohvaćanje odgovarajućih podataka i spremanje novih. Kao srednji red služi kao veza između klijenta i baze podataka.

Zadnji je red zadužen za *upravljanje podacima*. Odvajanje baze podataka na odvojeno računalo kao benefit nam donosi:

1. poslužitelj je potpuno nezavisan o bazi podataka
2. zbog sigurnosnih razloga klijent nikada neće imati informaciji o računalu na kojem je smještena baza podataka, niti pristup njemu
3. odvajanje baze podataka na drugo računalo poboljšava i performanse aplikacije jer svako računalo obavlja samo jednu zadaću

6.2 Arhitektura poslužitelja

Pri dizajniranju poslužitelja korištena je slojevita arhitektura: dijelovi aplikacije koji imaju istu funkcionalnost grupiraju se u odvojene slojeve koji se slažu jedan iznad drugog (vidi [8]). Svaki sloj čini apstrakciju oko jednog dijela logike koju poslužitelj treba obaviti. Svaka komponenta unutar pojedinog horizontalnog sloja može komunicirati ili sa bilo kojom komponentom iz istog sloja ili s bilo kojom komponentom iz sloja direktno ispod. Komunikacija se odvija preko dobro definiranih sučelja. Osnovni principi koji definiraju slojevitu arhitekturu su:

- **Apstrakcija**: slojevita arhitektura apstraktira logiku aplikacije u nekoliko slojeva gdje svaki sloj pruža dovoljno informacija da se razumije njegova uloga

- **Enkapsulacija**: tijekom dizajna ne trebaju se uzeti u obzir tipovi podataka, funkcije ili implementacija jer oni ne predstavljaju ograničenje na arhitekturu
- **Kohezija**: dobro definirati odgovornosti svakoj sloja i osigurati da svaka komponenta unutar sloja obavlja funkcionalnost koja je usko vezana za logiku sloju
- **Ponovno korištenje**: niži slojevi nemaju nikakve zavisnosti na višlje slojeve čime se omogućava njihova ponovna upotreba u drugim dijelovima aplikacije
- **Slobodna povezanost**: komunikacija između slojeva odvija se preko dobro apstraktiranih sučelja čime se omogućuje slobodna povezanost (engl. *loose coupling*) između slojeva

Poslužiteljski dio Mathos Intranet aplikacije organiziran je u tri sloja:

1. *kontroler*: dio poslužitelja koji je zadužen za komunikaciju s klijentom, tj. obradu HTTP zahtjeva
2. *servis*: sloj zadužen za poslovnu logiku aplikacije, tj. upisivanje svih podataka u bazu podataka, provjeru autentifikacije korisnika i preslagivanje podataka u potrebne hijerarhijske strukture
3. *repozitorij*: zadužen za komunikaciju s bazom podataka. Repozitorij ne komunicira direktno s bazom podataka, nego s *db* apstrakcijom

Svaki pristigli HTTP zahtjev od strane klijenta prolazi od gornjeg sloja (kontroler) sve do baze podataka i nazad. Kontroler poziva odgovarajuću metodu servisa koji poziva odgovarajuću metodu repozitorija zaduženog za dohvaćanje podataka iz baze. To je ujedno i nedostatak ovakve arhitekture jer da bi se obradio HTTP zahtjev, on mora proći kroz nekoliko slojeva i nazad što može utjecati na performanse poslužitelja.

6.3 Ubacivanje ovisnosti

Kako aplikacija s vremenom raste, tako se generira sve više komponenti, reprezentirane kao JavaScript objekti, koje međusobno stvaraju ovisnost o drugim komponentama. Tada se prirodno nameće pitanje na koji način stvoriti ovisnosti između komponenti tako da što manje ovise jedne o drugima. Ovaj problem rješava se primjenom principa *Ubacivanja ovisnosti* (engl. *Dependency injection*). Osnovna ideja principa je sve komponente u kodu držati odvojeno, a potrebnu komponentu na odgovarajući način "ubaciti" tamo gdje je potrebna. Za kreiranje komponenti zadužen je *spremnik ovisnosti* (engl. *dependency injection container*) koji nakon što ih stvori, ubacuje i po potrebi na pravo mjesto. U Mathos Intranet projektu korišten je InversifyJS, *spremnik ovisnosti* za JavaScript.

Komponente su unutar aplikacije reprezentirane kao JavaScript klase koristeći konstruktor funkcije (vidi 2.3). Sve komponente koje su joj potrebne, potrebno je navesti u konstruktor funkciji (primjer 21).

```
var CourseController = (function () {

    let _courseService;
    let _logger;

    function CourseController(courseService, logger) {
        _courseService = courseService;
        _logger = logger;
    }

    inversify.decorate(inversify.injectable(), CourseController);
    inversify.decorate(
        inversify.inject(serviceTypes.CourseService),
        CourseController, 0);
    inversify.decorate(
        inversify.inject(utilityTypes.Logger),
        CourseController, 1
    );

    return CourseController;
}());
```

Kod 20: Zavisne komponente unutar konstruktor funkcije

Da bi *spremnik ovisnosti* mogao stvarati potrebne komponente i ubaciti ih na pravo mjesto, potrebno je instancirati spremnik unutar kojeg registriramo sve komponente. Komponente se identificiraju (vežu) za identifikator. Osnovna ideja principa *ubacivanja ovisnosti* je identificirati svaku klasu sa sučeljem (engl. *interface*), no kako ih JavaScript ne podržava, InversifyJS koristi *string* identifikatore. Moguće je kontrolirati stvaranje komponenti unutar spremnika, tj. kontrolirati njihov djelokrug (engl. *scope*). InversifyJS podržava dva stupnja:

- *transient*: nova komponenta se stvara svaki put kad je zatražena
- *singleton*: komponenta se stvara samo jednom, kešira se i svaki put kad je zatražena koristi se ista istanca

U primjeru vidimo da je samo *db* komponenta navedena u *singleton* djelokrugu. Razlog je što ona zadužena za stvaranje i održavanje konekcija prema bazi podataka i ne želimo

da se nove veze stvaraju svaki put kada je *db* komponenta zatražena, nego samo jednom. Generalno treba biti oprezan pri korištenju *singleton* djelokruga. Ukoliko imamo na nekom mjestu definirano stanje komponente, korištenje *singleton* djelokruga treba izbjegavati jer kao posljedica mogu se pojaviti greške koje je jako teško za pronaći.

```
let containerModule = new inversify.ContainerModule((bind, unbind) => {
    bind(databaseTypes.options).toConstantValue(config.db);
    bind(databaseTypes.Db).to(Db).inSingletonScope();
    bind(databaseTypes.EmployeeRepository).to(EmployeeRepository);
    bind(databaseTypes.UserRepository).to(UserRepository);
});

}());
```

Kod 21: Zavisne komponente unutar konstruktor funkcije

Poslužiteljski dio je podijeljen u nekoliko modula za *ubacivanje ovisnosti* koji se ubacuju u glavni korijenski spremnik (primjer 22).

```
// Declare bindings
let container = new inversify.Container();

container.load(
    databaseModule,
    utilityModule,
    serviceModule,
    controllerModule
);
```

Kod 22: Glavni spremnik ovisnosti

6.4 Baza podataka

Svrha Mathos Intranet aplikacije je vođenje administrativnih zadaća fakulteta, od uređivanja kolegija, studijskih programa sve do vođenja svih zaposlenika. Time se stvara potreba za uređenom kolekcijom podataka u koju na efikasan način možemo unositi i čitati podatke. Mathos Intranet aplikacija koristi relacijski model baze podataka.

Relacijska baza podatke sprema u relacije koje se mogu interpretira kao tablice. Svaki zapis unutar relacije sadrži jedinstveni ključ koji ga identificira unutar njegove relacije, te proizvoljan broj atributa koji ga opisuju. Veze između relacija (različitih tablica unutar baze podataka) uspostavljaju se koristeći referencirajući odgovarajuće vrijednosti atributa druge relacije. Npr. ako zapis prve tablice želi referencirati zapis druge zablice, tada u jednom od svojih atributa mora sadržavati identifikator zapisa iz druge tablice. Tu nam dodatno pomaže takozvani princip **referencijalnog integriteta** koji nam garantira da podatak koji se refencira u drugoj tablici stvarno tamo i postoji. Referencijalni integritet se uspostavlja označavanjem atributa kao *strani ključ* unutar relacije.

Kako bi na efikasan način upravljali relacijskom bazom podataka, potreban nam je sustav za upravljanje bazom podataka. Mathos Intranet aplikacija koristi besplatni sustav za upravljanje otvorenog koda MySQL.

Aplikacija sve podatke sprema unutar 58 takvih relacija uređenih odgovarajućim međusobnim vezama.

```
1  Db.prototype.execute = async function (queryString, queryParams) {
2      let deferred = q.defer();
3      try {
4          let connection = await getConnection();
5          connection.query(queryString, queryParams, function (err, rows, fields
               ) {
6              if (err) {
7                  deferred.reject(err);
8              } else {
9                  deferred.resolve(rows);
10             }
11             connection.release();
12         });
13         return deferred.promise;
14     } catch (err) {
15         throw err;
16     }
17 }
```

Kod 23: metoda koja izvršava SQL upite prema MySQL bazi podataka

U kodu poslužitelja, baza podataka reprezentirana je kao *db* objekt. *Db* objekt kreira se samo jednom i po potrebi ubacuje u druge JavaScript klase. Zadaća mu je kreiranje nekoliko

konekcija prema bazi podataka i izvršavanje SQL upita. Sve SQL naredbe izvršavaju se preko funkcije *execute()* koja kao parametre prima SQL upit i parametre.

```
1  CourseRepository.prototype.add = async function (item) {
2      return await _db.execute(courseProcedures.add, [
3          item.sifra,
4          item.ime,
5          item.P,
6          item.V,
7          item.S,
8          item.ECTS,
9          item.clanak_id
10     ]);
11 }
```

Kod 24: primjena *db* apstrakcije unutar repozitorija

Bitno je napomenuti da je *db* objekt direktno vezan za MySQL bazu podataka zbog *mysql* NPM modula koji se koristi. Repozitoriji koji su zaduženi za dohvaćanje podataka iz baze ni u kojem trenutku ne znaju s kojom bazom podataka komuniciraju upravo zbog *db* objekta koji stvara apstrakciju oko baze podataka. Ukoliko bi se odlučili koristi neki drugi sustav za upravljanje bazama podataka, npr. PostgreSQL, morali bi napraviti novu instancu *db* objekta koji bi bio zadužen za komunikaciju s PostreSQL bazom podataka i samo ga ubaciti u repozitorije umjesto postojeć *db* objekta.

Literatura

[1] R. FIELDING, *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000.

[2] D. FLANAGAN, *Javascript: The Definitive Guide*, O'Reilly Media, Sebastopol, CA, 2011.

[3] J. KUROSE, K. ROSE, *Computer Networking: A Top-Down Approach*, Pearson, New Jersey, 2013.

[4] A. MARDAN, *Pro Express.js*, Apress, New York, 2014.

[5] K. SIMPSON, *You don't know JS: Async and Performance*, O'Reilly Media, Sebastopol, CA, 2015.

[6] B. SYED, *Beginnig Node.js*, Apress, New York, 2014.

[7] N. C. ZAKAS, *Professional Javascript for Web Developers*, Third Edition, John Wiley & Sons, Inc., Indianapolis, 2012.

[8] *Architectural Patterns And Styles*, MSDN Library, https://msdn.microsoft.com/en-us/library/ee658117.aspx#LayeredStyle

[9] *Javascript announcement press release*, Netscape Communications Corporation and Sun Microsystems, Inc, 4.12.1995. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html

[10] *Node.js performance 2017: v7.9.0 vs. Hapi, Express.js, Restify and Koa and more*, https://raygun.com/blog/node-js-performance-2017/

[11] *Performance Comparison: Java vs Node*, https://www.tandemseven.com/blog/performance-java-vs-node/

Zaključak

Iako je dugo vremena bio smatran kao programski jezik weba, pojavom Node.js platforme koja koristi Google-ov V8 engine za pokretanje koda, JavaScript se nameće kao ozbiljan jezik i za poslužiteljsku stranu. Tada možemo koristi JSON notaciju i na klijentu i poslužitelju bez pretvorbe podataka. Node zbog asinkronog izvršavanja operacija i jedne procesne niti koja izvršava kod, daje puno bolje performanse pod velikim opterečenjem (što smo i zaključili u poglavlju 4.4) od standardnih poslužitelja koji koriste procesne niti za svaki pristigli HTTP zahtjev. Uz kombinaciju sa svojom minimalnom jezgrom koja se može dodatno proširit funkcionalnostima preko NPM servisa, Node se promiče kao ozbiljna platforma za stvaranje poslužiteljskih programa. Jedan od takvih web okvira je i *Express.js* koji nam olakšava izradu HTTP poslužitelja. Iako njegove performanse nisu bolje u odnosu na standardni *http* modul koji dolazi sa Node.js jezgrom, on nam donosi neke druge prednosti kao što su jednostavnost implementacije velikih poslužitelja i jednostavnost održavanja što je ključno u kasnijim fazama projekta.

Sažetak

Ovaj diplomski rad opisuje korištenje JavaScript programskog jezika na poslužiteljskog strani. JavaScript je od programskog jezika koji se u svojim počecima koristio za jednostavne animacije napredovao do ozbiljnog programskog jezika pomoću kojega je moguće stvoriti *full stack* web aplikacije. Objasniti će se klijent-poslužitelj model koji se koristi za izradu modernih web aplikacija i REST princip koji je u današnje vrijeme gotovo neizostavni način izrade poslužitelja. Platforma koja nam omogućuje izvršavanja JavaScript programskog jezika izvan web preglednika je Node.js. Logika Node.js platforme je mala jezgra s osnovnim funkcionalnostima, a ostale funkcionalnosti se proširuju takozvanim modulima koji se na jednostavan način distribuiraju preko NPM-a (*Node package manager*). U kombinaciji s Express.js okvirom moguće je na brz i jednostavan način stvoriti REST model poslužitelja koji nudi odlične performanse.

Ključne riječi: JavaScript, asinkrono, klijent-poslužitelj, HTTP, REST, Node.js, moduli, NPM, Express.js, middleware, slojevita arhitektura, ubacivanje ovisnosti, MySQL

Summary

In this master thesis we will describe use of JavaScript programming language for building web servers. From language that was used for animating web pages, JavaScript grew to mature language for building full stack web applications. We will describe client-server model that is used for modern web applications and REST principle that is almost indispensable for today's web servers. Thanks to Node.js platform we can use JavaScript outside web browser. Node.js philosophy is small kernel that can be expanded with outside modules provided from NPM (*Node package manager*) service. Combined with Express.js package for simple and fast development of HTTP servers, we can build REST web servers that will have excellent performances.

Keywords: JavaScript, asynchronous, client-server, HTTP, REST, Node.js, modules, NPM, Express.js, middleware, layered architecture, dependency injection, MySQL

Životopis

Davor Kolarević rođen je 4. lipnja 1993. u Našicama. Školovanje započinje 2000. godine u Osnovnoj školi Josipa Jurja Strossmayera u Đurđenovcu, a nakon završetka srednjoškolsko obrazovanje nastavlja u općoj gimnaziji Srednje škole Isidora Kršnjavoga u Našicama. 2012. godine upisuje preddiplomski studij Matematika na Odjelu za matematiku Sveučilišta Josipa Jurja Strossmayera u Osijeku. Nakon uspješnog završetka preddiplomskog studija s završnim radom na temu *Sustav za mrežnu pohranu*, 2015. godine upisuje i diplomski studij Matematike i računarstvo na istom odjelu. Iste godine sudjeluje i na natjecanju iz programiranju IEEEXtreme. Tijekom diplomskog studija stekao je praktična znanja na praksama u tvrtkama *Farmeron* u Osijeku i *Adacta* u Zagrebu, a po završetku studija radno iskustvo stječe u softverskoj tvrtci *Mono Software* u Osijeku kao razvojni programer. U 2017. godini sudjeluje i kao predavač na ljetnoj školi programiranja *CodeClub*.