



UNIVERZITET SINGIDUNUM U BEOGRADU
FAKULTET ZA INFORMATIKU I RAČUNARSTVO

Uvod u Node.JS i programiranje vođeno događajima
-Seminarski Rad-
iz predmeta Interakcija čovek računar

Mentor:

Doc. dr Milan Čabarkapa

Student:

Mihailo Joksimović
2015/202251

Beograd, 2017

Predgovor	2
Uvod	3
Node.JS	6
Prvi koraci	6
Arhitektura platforme	10
Dizajn ciljevi	12
Događaji i EventEmitter komponenta	15
Primeri	18
HTTP klijent	18
Rad sa fajlovima	20
Zaključak	23
Reference	23

Predgovor

Cilj ovog rada je da studente upozna i približi im principe rada Node.js-a i asinhronog programiranja.

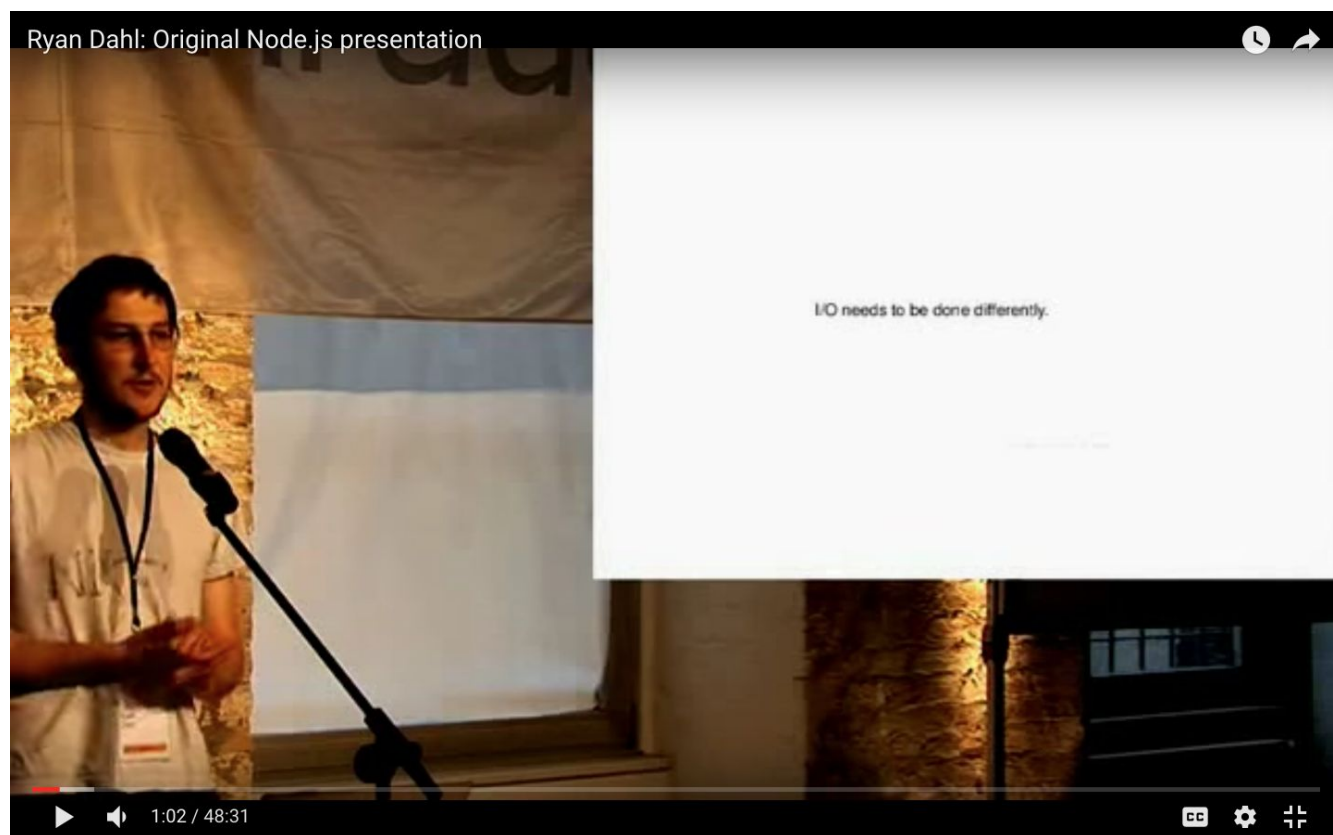
Sa tom idejom, rad počinje i analizira same početke jezika, ciljeve i motive kojima se vodio tvorac ovog jezika - Ryan Dahl. Nakon uvoda, prodiskutovaćemo samu arhitekturu koja pokreće Node uz obavezan osvrt na programiranje vođeno događajima. Rad ćemo završiti prikazom korišćenja nekoliko ugrađenih biblioteka.

Bitno je napomenuti da je rad fokusiran na principe rada Node-a i stoga se velikim delom razmatra filozofija i ciljevi jezika. Smatra se da je čitalac već upoznat sa osnovama programiranja, rada sa ulazom i izlazom, kao i JavaScript-om i njegovom sintaksom.

Nakon iščitavanja rada, čitalac će biti u mogućnosti da samostalno započne izradu svoje prve Node aplikacije.

Uvod

“I/O mora biti urađen drugačije. Način na koji ga koristimo je potpuno pogrešan. Ovo se mora promeniti”. Ovo su reči kojima je Rajan Dal (eng. Ryan Dahl), tvorac Nodejsa započeo prezentaciju na JSConf konferenciji 2009. godine.



Slika 1. Ryan Dahl, JSConf 2009. godine

Node.js je platforma otvorenog koda (eng. open source) na kojoj možemo pisati i pokretati serverske aplikacije korišćenjem dobro poznate JavaScript sintakse. Sama platforma kao i prevodilac (eng. interpreter) koda napisani su u C-u i koriste Googleovu V8 platformu za prevođenje i interpretiranje koda.

Platforma je nastala usled autorove frustracije načinom na koji funkcioniše ulaz/izlaz podataka (u daljem tekstu - I/O) i time što svaka I/O operacija blokira glavnu nit programa. Kako Rajan

navodi, ovo je gubljenje vremena jer umesto čekanja da se završi blokirajuća I/O operacija, aplikacija može obrađivati druge zahteve i iskoristiti procesorsko vreme na mnogo bolje načine.

Kako bi smo bolje dočarali gore pomenuti problem, pogledajmo sledeći kod:

```
1 <?php
2
3 $data = $pdo->query("SELECT COUNT(*) FROM korisnici WHERE aktivan = 1");
4
5 printTable($data);
6
7
```

Slika 2. Sinhroni blokirajući kod

Problem sa ovim kodom je što aplikacija mora da sačeka da se upit završi i za to vreme nije u mogućnosti da uradi bilo šta drugo. Ovo je primer blokade usled čekanja na I/O.

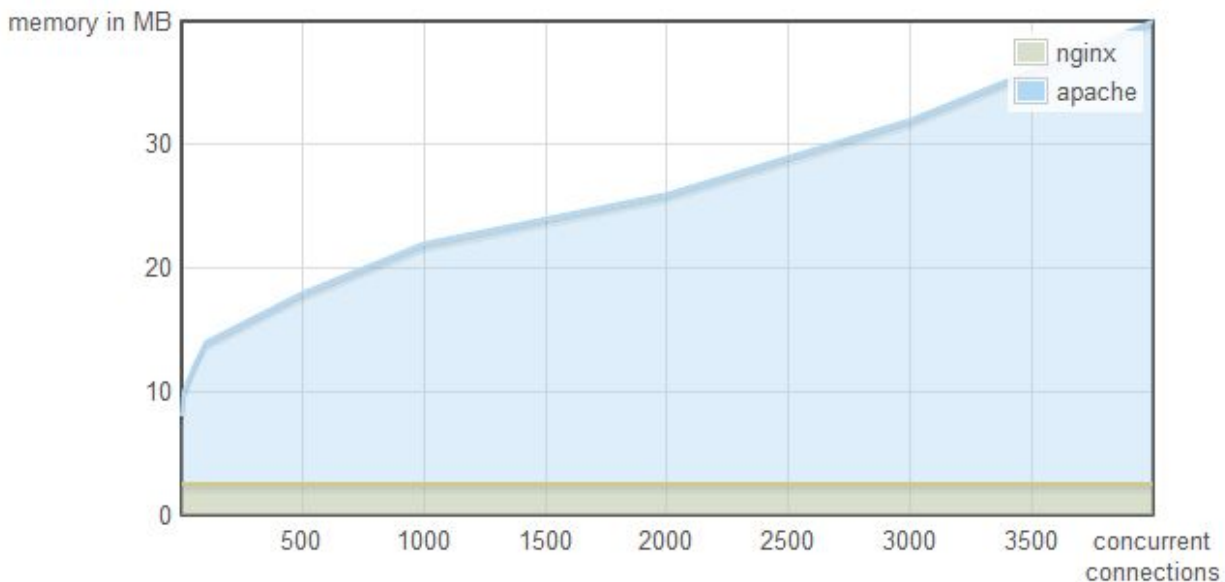
Sa druge strane, pogledajmo sledeći kod:

```
1 ▼ db.korisnici.query(
2     "SELECT COUNT(*) FROM korisnici WHERE aktivan = 1",
3 ▼     function(data) {
4         // Uradi nesto sa podacima ...
5 ▲     }
6 ▲ );
7
8 // Pokreni neki drugi query ...
9
10 // Uradi jos nesto ...
11
12 // I jos nesto ...
```

Slika 3. Asinhroni neblokirajući kod

Šta se promenilo? Na slici 3 možemo videti da će aplikacija pokrenuti upit i umesto da sačeka da se isti završi, ona će odraditi još dve ili tri dodatne operacije. Istovremeno, kada se I/O operacija završi, pokrenuće se povratna funkcija (eng. callback function) i procesiraće se rezultati.

Pogledajmo još jedan primer sa stvarnim rezultatima. U pitanju je poređenje Apache i nginx servera i dijagram opterećenosti u zavisnosti od broja paralelnih konekcija.



Slika 4. Dijagram opterećenosti kod Apache i nginx servera

Na horizontalnoj osi se nalazi broj klijenata povezanih na server, dok se na vertikalnoj nalazi opterećenost serverske memorije. Kao što možemo videti, kod Apachea se zahtevana memorija izuzetno povećava sa povećanjem broja konekcija što direktno utiče na opterećenje servera. Sa druge strane, možemo primetiti da se kod nginx-a opterećenje uopšte ne menja sa povećanjem broja konekcija.

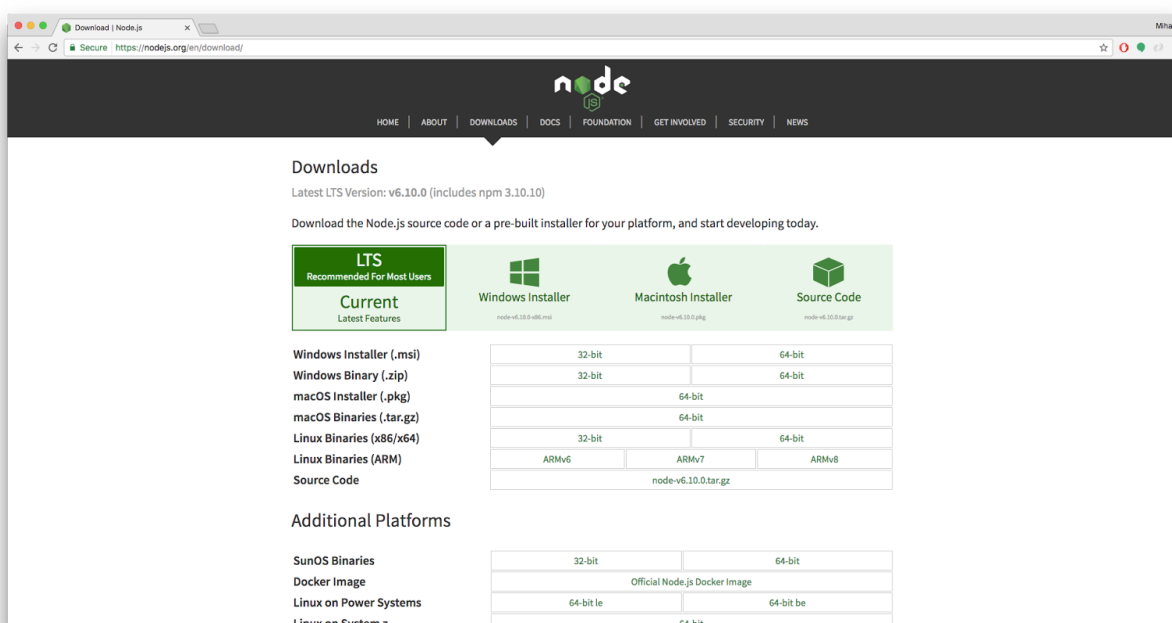
U čemu je razlika? Razlika je u pristupu i opsluživanju klijenata. Naime, Apache koristi po jednu nit (eng. thread) za svakog klijenta, što nam govori da sa povećanjem broja klijenata direktno raste i opterećenje servera. Sa druge strane, nginx koristi arhitekturu vođenu događajima i umesto čekanja na I/O, on opslužuje druge klijente. Ovo je upravo način na koji Nodejs radi i upravo to će biti tema ovog rada.

Na kraju, korisno je pomenuti da je u trenutku pisanja ovog rada, aktuelna i stabilna verzija Nodejsa verzija 6.0. Za ovu verziju se trenutno garantuje dugoročna podrška (eng. LTS - Long Term Support) i ona je skoro u potpunosti kompatibilna sa najnovijom ECMA Script specifikacijom (ES6). Više informacija o samom Nodejsu i kompatibilnosti sa ES6 možete naći na sledećim adresama: <http://nodejs.org> i <http://node.green/>.

Node.JS

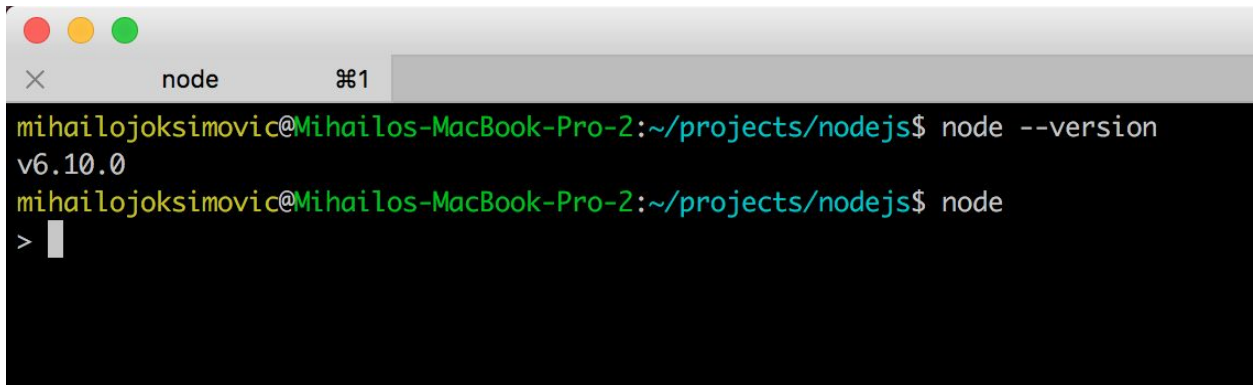
Prvi koraci

Pre nego što počnemo sa korišćenjem Node-a, potrebno je instalirati prevodilac koda, kao i menadžer paketa - *npm* (eng. Node Package Manager). Kao i sve u Node-u, i instalacija je prilično jednostavna. Sve što treba da uradimo je da posetimo sledeću stranicu <https://nodejs.org/en/download/>, izaberemo odgovarajući paket, preuzmemo ga i pokrenemo.



Slika 5. NodeJS download strana

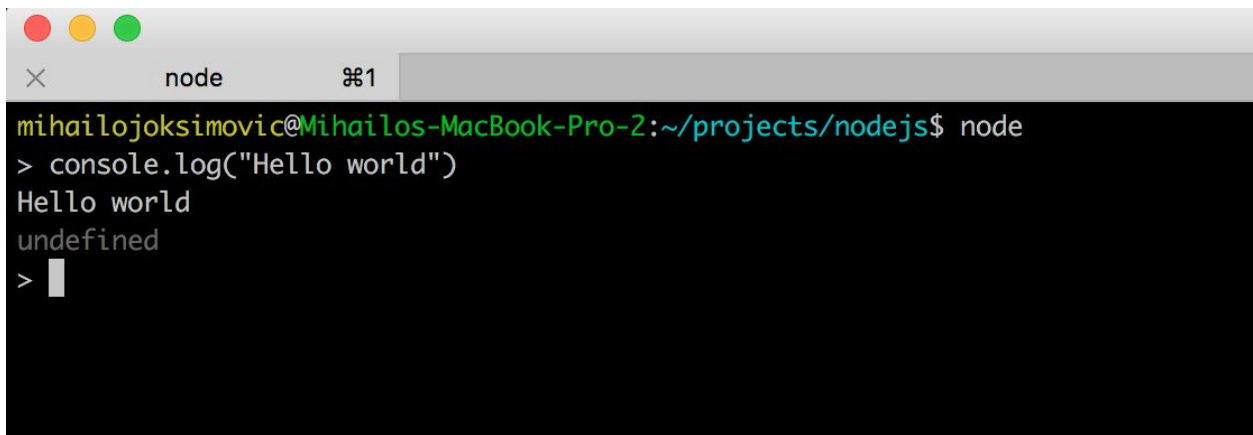
Nakon preuzimanja i uspešne instalacije, pokretanje naredbe `node --version` treba da nam prikaže verziju Node-a koju smo upravo instalirali. Takođe, pokretanjem naredbe `node` pojaviće nam se interaktivni ekran u kome možemo izvršavati naredbe.

A terminal window titled 'node' with a tab labeled '⌘1'. The prompt is 'mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs\$'. The user enters 'node --version' and the output is 'v6.10.0'. Then the user enters 'node' and the prompt returns to '>'.

```
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node --version
v6.10.0
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node
>
```

Slika 6. Node u konzoli

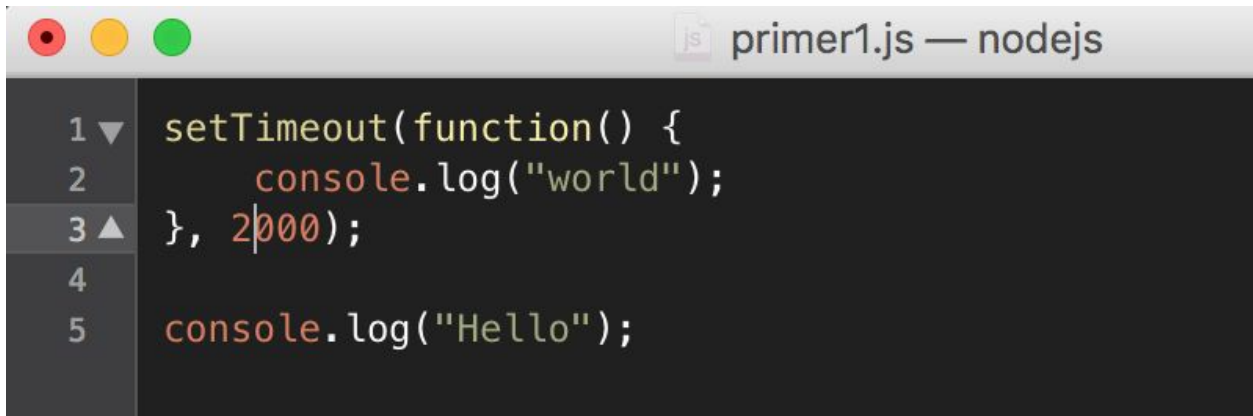
Kako tradicija nalaže, prvi korak pri učenju novog programskog jezika jeste ispisivanje teksta “Hello world” na ekranu. U Node-u to izgleda ovako:

A terminal window titled 'node' with a tab labeled '⌘1'. The prompt is 'mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs\$'. The user enters 'node' and then '> console.log("Hello world")'. The output is 'Hello world' followed by 'undefined' on the next line. The prompt returns to '>'.

```
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node
> console.log("Hello world")
Hello world
undefined
>
```

Prilično jednostavno, zar ne? Ono što možemo primetiti je da smo upravo iskoristili klasičnu JavaScript sintaksu, kao i dobro poznatu `console` komponentu kako bi smo na ekranu ispisali “Hello world” tekst. To je upravo ono što nam Node nudi - dobro poznata JavaScript sintaksa, kombinovana sa mnoštvom korisnih paketa i asinhronom arhitekturom vođenom događajima.

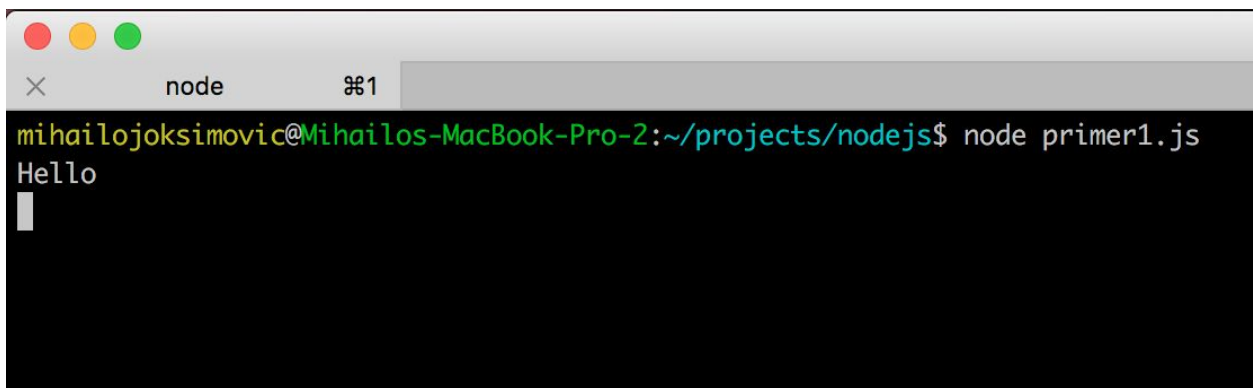
Sada ćemo prikazati jedan bolji primer, koji koristi gore pomenutu arhitekturu vođenu događajima (eng. Event-driven architecture). Prvo ćemo otkucati kôd koristeći dobro poznati editor teksta - TextMate:



```
1 ▼ setTimeout(function() {  
2     console.log("world");  
3 ▲ }, 2000);  
4  
5 console.log("Hello");
```

Slika 7. Primer jednostavnog, asinhronog koda vođenog događajima

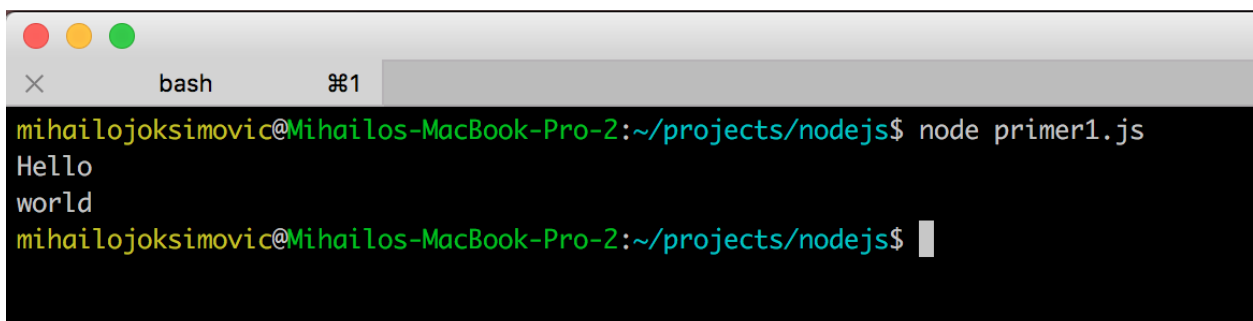
Nakon izvršavanja skripta sa gore pomenutim kôdom, na ekranu će se pojaviti sledeći sadržaj:



```
node  ⌘1  
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node primer1.js  
Hello  
█
```

Slika 8. Izvršavanje primer1.js skripta

Dve sekunde kasnije, na ekranu će se prikazati sledeći sadržaj:



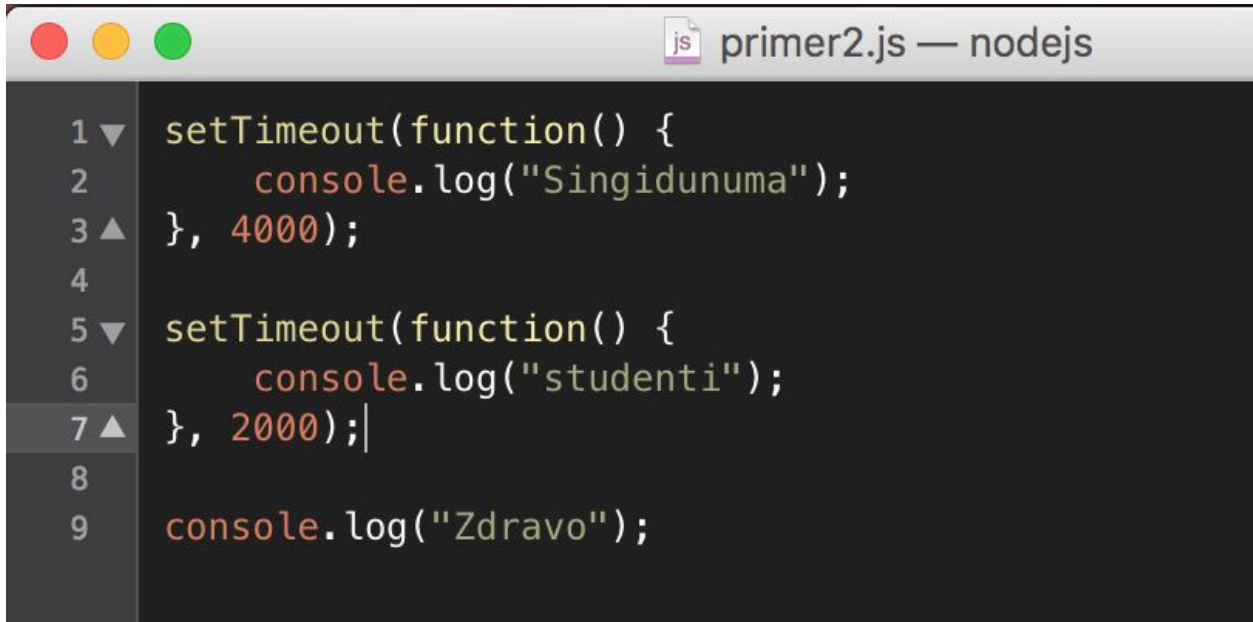
```
bash  ⌘1  
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node primer1.js  
Hello  
world  
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ █
```

Slika 9. Kraj izvršavanja skripta

Iz ovog primera možemo primetiti nekoliko stvari:

- Node koristi staru, poznatu JavaScript sintaksu
- Linija 5 se izvršila iako se kôd sa linija 1-3 još uvek nije izvršio
- Dve sekunde kasnije, kada je “okinut” događaj, tekst sa linije 2 je ispisan na ekranu
- Nakon što su sve povratne (eng. callback) funkcije pozvane, izvršavanje programa je završeno

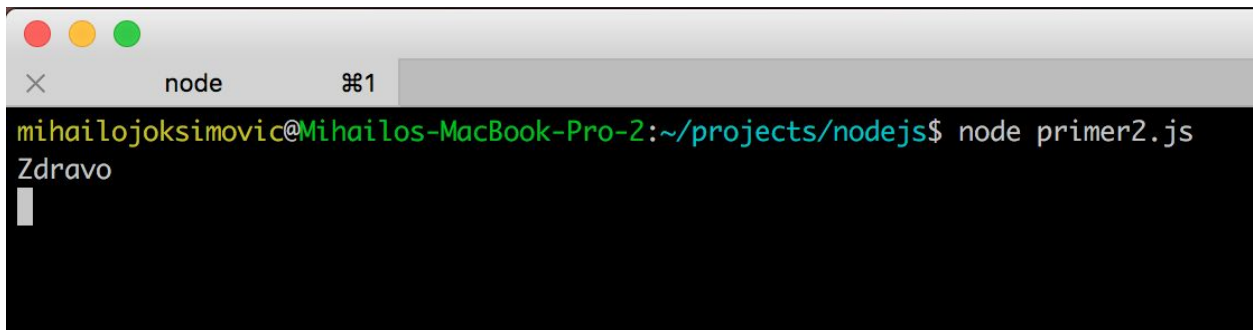
Pogledajmo još jedan primer:



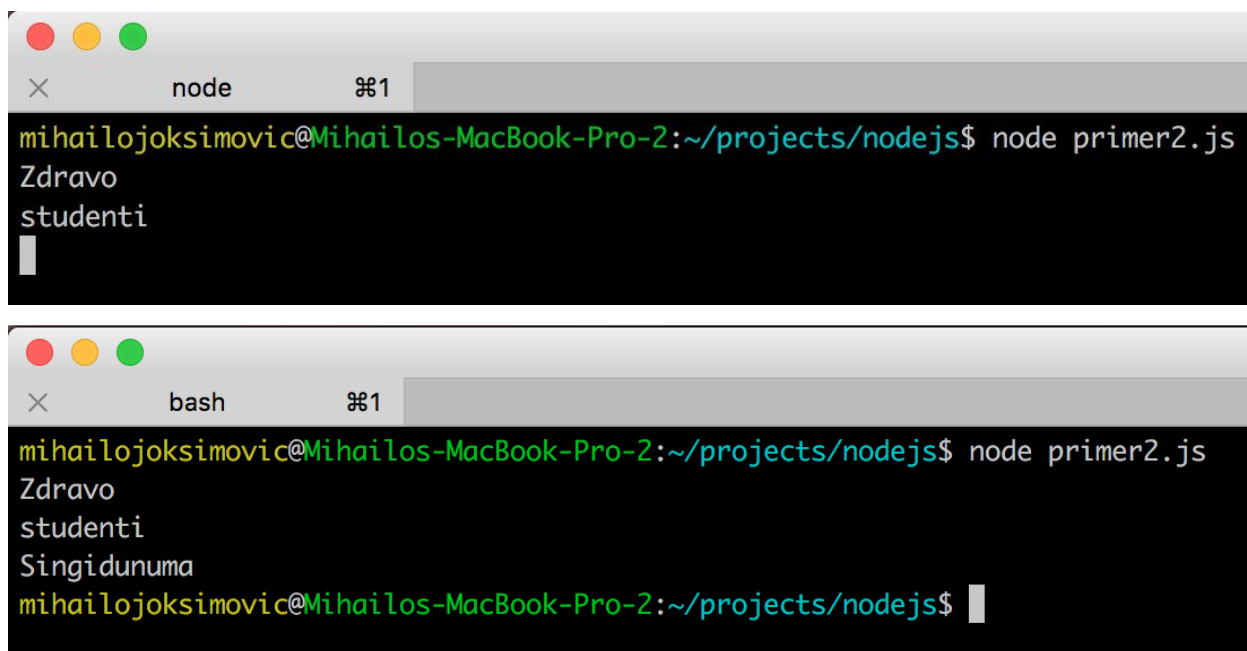
```
1 ▼ setTimeout(function() {  
2     console.log("Singidunuma");  
3 ▲ }, 4000);  
4  
5 ▼ setTimeout(function() {  
6     console.log("studenti");  
7 ▲ }, 2000);  
8  
9 console.log("Zdravo");
```

Slika 10. Kod skripta primer2.js

Da li možete da pretpostavite šta će biti izlaz ovog programa?



```
node  ⌘1  
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node primer2.js  
Zdravo  
█
```



```
node
miailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node primer2.js
Zdravo
studenti

bash
miailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$ node primer2.js
Zdravo
studenti
Singidunuma
miailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs$
```

Slika 11. Izlaz programa nakon pokretanja, zatim 2s kasnije i 4s kasnije

Ako ste pretpostavili da će na ekranu biti ispisan tekst “Zdravo”, zatim dve sekunde kasnije “studenti” i na kraju četiri sekunde kasnije “Singidunuma” - čestitamo, bili ste u pravu!

Možemo zaključiti da je za razliku od dobro poznatih jezika - C++, C#, Java, Python i drugih - Node.js potpuno asinhron i vođen događajima (naravno, ovo moguće i u drugim jezicima, ali je malo komplikovanije). U daljem tekstu fokusiraćemo se na osnovne dizajn principe jezika, ukratko ćemo objasniti pojam arhitekture vođene događajima i daćemo primere nekoliko korisnih komponenti.

Sve primere koji su navedeni ovde, možete preuzeti na sledećoj strani:

<https://github.com/MihailoJksimovic/introduction-to-nodejs-code-camples>

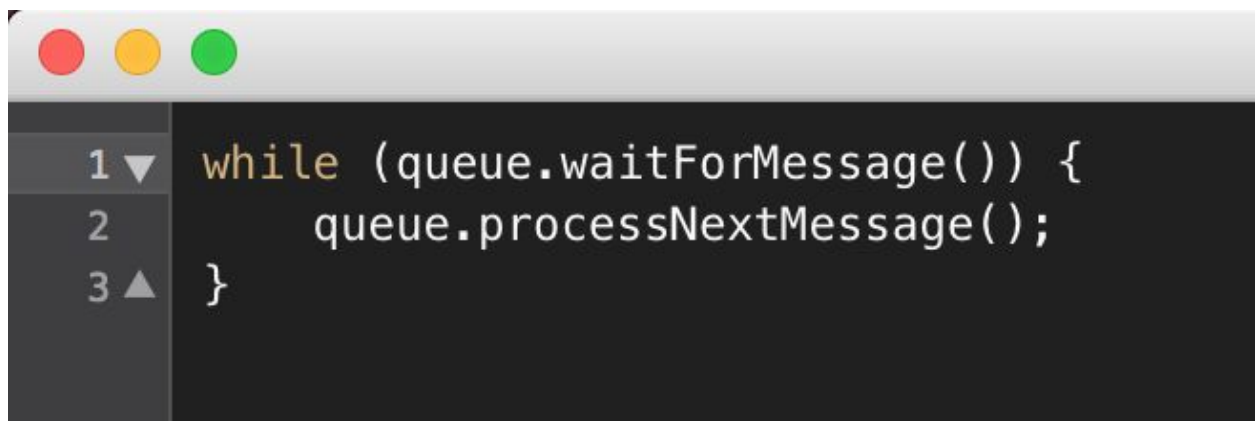
Arhitektura platforme

Kako bi smo bolje razumeli principe funkcionisanja Node-a, nakratko ćemo se pozabaviti samom arhitekturom platforme.

Prisetimo se citata sa početka rada - *“I/O mora biti urađen drugačije. Način na koji ga koristimo je potpuno pogrešan. Ovo se mora promeniti”*. Autor Nodea je primarno bio nezadovoljan time što većina programa većinu vremena provede čekajući na neki resurs (podatke sa diska, podatke sa mreže, itd).

Kako bi se prevazišao problem blokirajućih operacija, iskorišćen je princip petlje događaja (eng. event loop) koji se od inače koristi i u JavaScriptu. Naime, sam program se, grubo gledano, sastoji od jedne glavne niti i jedne sporedne niti koja obrađuje spore, blokirajuće operacije. Glavna komponenta koja omogućava “asinhrono” izvršavanje je libuv biblioteka. Više informacija možete naći na <http://libuv.org/>

Dakle, najjednostavnije gledano, srž programa se može predstaviti sledećim kodom:



```
1 while (queue.waitForMessage()) {  
2   queue.processNextMessage();  
3 }
```

Slika 14. Petlja koja čeka i obrađuje nove događaje

Sledeći dijagram će nam dati detaljniji i bolji uvid:



Slika 15. Node-ova petlja događaja

Kako ovo funkcionise:

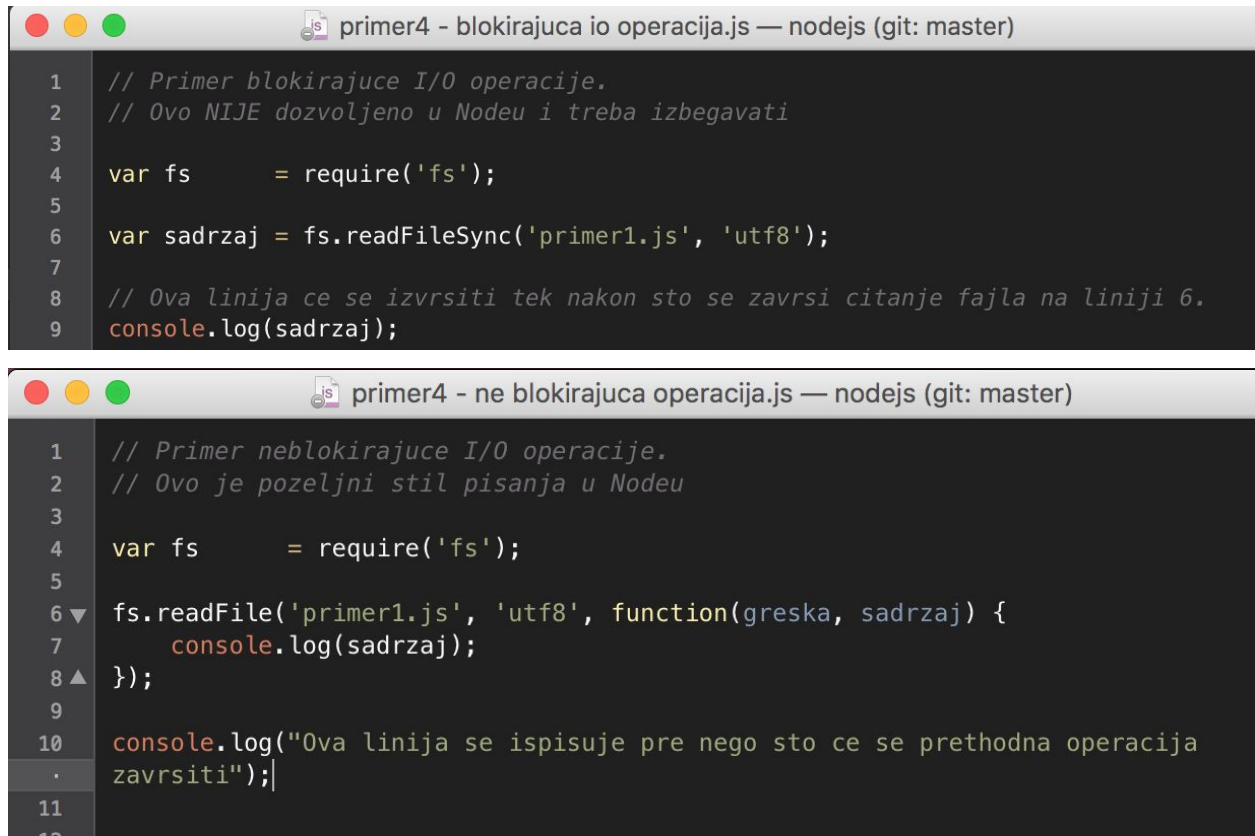
1. U početnom trenutku, program ne radi ništa i čeka na novi zahtev
2. Sledećeg trenutka, stiže novi zahtev od korisnika (npr. pročitaj sadržaj fajla “fajl.txt”)
3. Program ovaj zahtev prosleđuje File System komponenti koja ovaj zahtev smešta na stek (eng. stack) intenzivnih operacija. Nakon toga, glavnoj petlji se vraća objekat koji predstavlja obećanje (eng. promise) da će se javiti nazad kada operacija bude izvršena
4. Pošto je operacija uspešno izvršena, program nastavlja da se vrti u petlji i može da **nastavi da opslužuje druge zahteve**
5. Nekoliko trenutaka kasnije, kada je operacija iz tačke 3 izvršena, prethodno dato obećanje se ispunjava, poziva se povratna funkcija i program može da obrati rezultate

Koraci 3 i 4 su suština Nodea i predstavljaju ono u čemu je on i najbolji - mogućnost paralelnog procesiranja velikog broja zahteva, bez potrebe za kreiranjem dodatnih niti. Naravno, upravo u ovome leži i mana Nodea - nije pogodan za obradu intenzivnih operacija koje zauzimaju procesorsko vreme. Više reči o ovome možete naći u poslednjim poglavljima.

Dizajn ciljevi

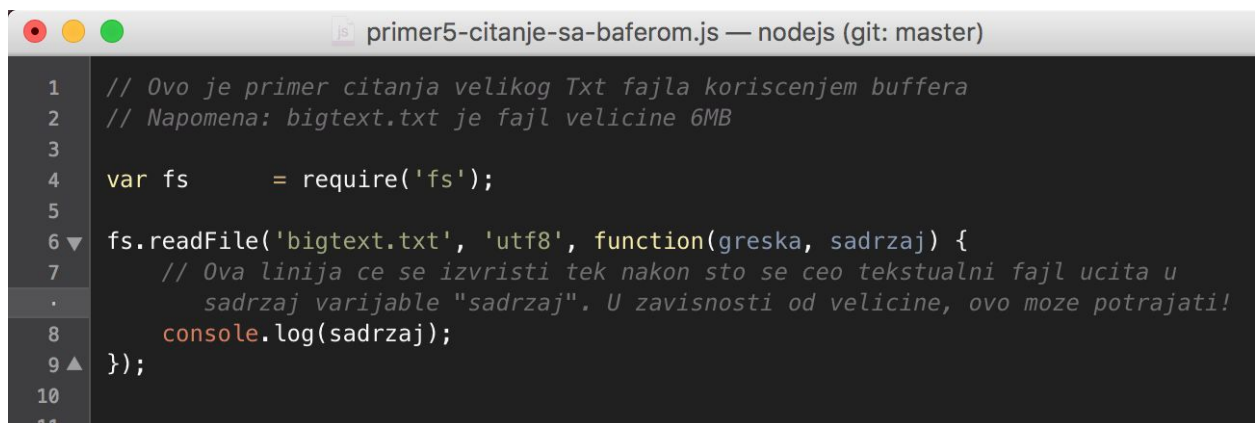
Kako bi smo bolje razumeli principe funkcionisanja, bitno je napomenuti glavne dizajn ciljeve kojima se Rajan Dal vodio kada je kreirao prvu verziju Node-a:

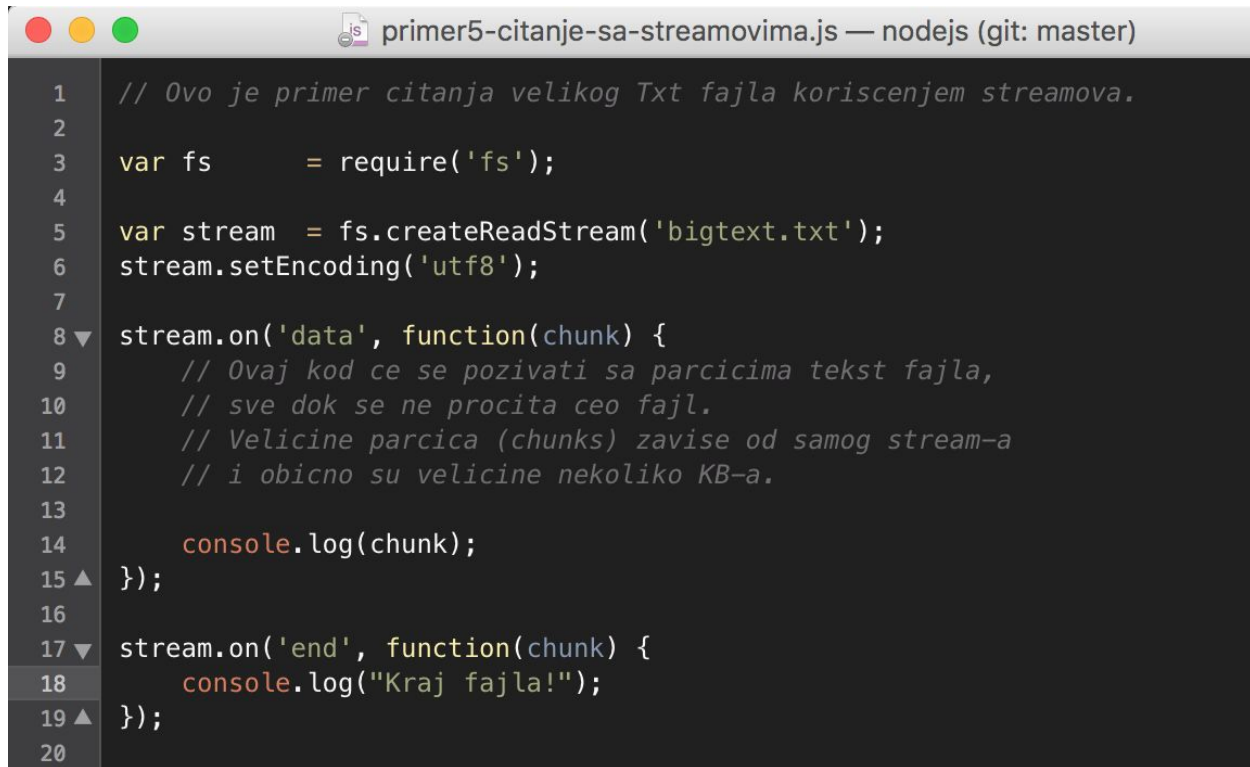
1. Blokirajuće I/O operacije se **moraju izbegavati** - umesto toga, **moramo koristiti povratne funkcije** kako bi smo pročitali podatke sa diska, mreže ili nekog drugog procesa.



Slika 12. Primer blokirajuće i neblokirajuće I/O operacije

2. **Strimovi (eng. streams) umesto bafera (eng. buffer)** - uvek koristite strimove umesto bafera. Na ovaj način štedimo resurse i dobijamo najbolje performanse iz Nodea.





```
1 // Ovo je primer citanja velikog Txt fajla koriscenjem streamova.
2
3 var fs      = require('fs');
4
5 var stream = fs.createReadStream('bigtext.txt');
6 stream.setEncoding('utf8');
7
8 stream.on('data', function(chunk) {
9     // Ovaj kod ce se pozivati sa parvicima tekst fajla,
10    // sve dok se ne procita ceo fajl.
11    // Velicine parcica (chunks) zavise od samog stream-a
12    // i obicno su velicine nekoliko KB-a.
13
14    console.log(chunk);
15 });
16
17 stream.on('end', function(chunk) {
18     console.log("Kraj fajla!");
19 });
20
```

Slika 13. Citanje koriscenjem bafera i strima

3. Ugrađena **podrška za najkorišćenije servise** - jedna od osnovnih filozofija Nodejsa jeste da podrška za najkorišćenije servise (TCP, HTTP, DNS) mora biti ugrađena u jezgro jezika.

- | | | | |
|---------------------|-------------------------------|----------------|-------------------|
| • Assertion Testing | • Buffer | • C/C++ Addons | • Child Processes |
| • Cluster | • Command Line Options | • Console | • Crypto |
| • Debugger | • DNS | • Domain | • Errors |
| • Events | • File System | • Globals | • HTTP |
| • HTTPS | • Modules | • Net | • OS |
| • Path | • Process | • Punycode | • Query Strings |
| • Readline | • REPL | • Stream | • String Decoder |
| • Timers | • TLS/SSL | • TTY | • UDP/Datagram |
| • URL | • Utilities | • V8 | • VM |
| • ZLIB | • GitHub Repo & Issue Tracker | • Mailing List | |

Slika 14. Lista svih ugrađenih modula

4. Jezički interfejs (eng. API - Application Programming Interface) **mora biti dobro poznat** - sintaksu moraju razumeti i frontend developeri i hakeri!

5. Jezik **mora biti nezavistan od platforme** (eng. platform independant) - Node se može pokretati na svim popularnim platformama (Windows, Linux, FreeBSD, ...)

Događaji i EventEmitter komponenta

Kao što smo prethodno napomenuli, ideja Nodea je da bude asinhron i da se sve dugačke operacije procesiraju van glavne niti. Kako bi se olakšao rad sa događajima, uvedena je *EventEmitter* biblioteka koje predstavlja jednu od glavnih komponenti.

Naime, skoro sve klase i komponente u Node paketu emituju događaje (eng. event) svaki put kada se nešto bitno dogodi. Ovu mogućnost im pruža gore pomenuta biblioteka koja uvodi `emit()` i `on()` metode za slanje odnosno osluškivanje događaja.

Pogledajmo jednostavan primer:

```
1  const EventEmitter = require('events');
2
3  class MojPosiljalacDogadjaja extends EventEmitter {}
4
5  const mojPosiljalac = new MojPosiljalacDogadjaja();
6
7  ▼ mojPosiljalac.on('dogadjaj', () => {
8    console.log('Uhvacen dogadjaj!');
9  ▲ });
10
11  console.log("Pre slanja dogadjaja");
12
13  mojPosiljalac.emit('dogadjaj');
14
15  console.log("Nakon slanja dogadjaja");
16
17
```

```
mihaiojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$ node primer6-event-emitter.js
Pre slanja dogadjaja
Uhvacen dogadjaj!
Nakon slanja dogadjaja
mihaiojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$
```

Slika 16. EventEmitter i izlaz programa

Iako je primer vrlo jednostavan, možemo uočiti kako Node funkcioniše. Pogledajmo sada sledeći primer:

```
1 console.log("Ovo je tekst na liniji 1");
2
3 ▼ setTimeout(function() {
4     console.log("Ovo je tekst na liniji 4");
5 ▲ }, 0);
6
7 console.log("Ovo je tekst na liniji 7");
```

Slika 17. Primer asinhronog koda

Da li možete da pretpostavite kojim redosledom će biti ispisan tekst na ekranu?

Ako ste rekli - linija 1, 7, 4 - bili ste u pravu:

```
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$ node primer7.js
Ovo je tekst na liniji 1
Ovo je tekst na liniji 7
Ovo je tekst na liniji 4
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$
```

Slika 18. Izlaz programa sa slike 17

Sada uzmimo sledeći primer:

```
1  const EventEmitter = require('events');
2
3  class MojPosiljalacDogadjaja extends EventEmitter {}
4
5  const mojPosiljalac = new MojPosiljalacDogadjaja();
6
7  console.log("Pocetak programa");
8
9  ▼ setTimeout(function() {
10     console.log("Unutar setTimeout() funkcije");
11  ▲ }, 0);
12
13  ▼ mojPosiljalac.on('dogadjaj', () => {
14     console.log('Unutar osluskivaca dogadjaja');
15  ▲ });
16
17  mojPosiljalac.emit("dogadjaj");
18
19  console.log("Kraj programa");
20
21
```

```
mihailojoksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$ node primer7-1.js
Pocetak programa
Unutar osluskivaca dogadjaja
Kraj programa
Unutar setTimeout() funkcije
mihailojoksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$
```

Slika 19. Primer programa sa dogadjajima i timeout-om

Šta se upravo desilo? Tekst unutar `setTimeout()` funkcije koji je trebao da se izvrši odmah nakon linije 7, izvršen je tek na kraju.

Radi se o tome da svaki program ima dve liste čekanja sa zadacima koji treba da se izvrše (eng. task queue). Na jednu listu odlaze jednostavni zadaci (tzv. mikro zadaci, eng. microtasks) i u tu grupu između ostalog spada i `.on()` odnosno osluškivači događaja, dok na drugu listu odlaze komplikovaniji zadaci (tzv. makro zadaci, eng. macrotasks) i tu između ostalog spada i `setTimeout()` funkcija.

Pri pokretanju programa, Node obrađuje jednu po jednu liniju programa i raspoređuje ih na mikro i makro liste čekanja. Pravilo je da se pri svakoj iteraciji Node petlje (prisetimo se teksta sa [početka ovog poglavlja](#)) prvo procesiraju svi jednostavni (mikro) zadaci koji stoje na listi čekanja, a zatim se procesira samo jedan zadatak sa liste zahtevnijih (makro) zadataka. Dakle, konkretno u našem primeru, prvo su obrađeni jednostavni zadaci (linije 7, 14, 17 i 19), a zatim je obrađen makro zadatak sa linije 9 (`setTimeout()`).

Ukoliko ste počeli da razmišljate u pravcu toga da ovo zapravo nije potpuno asinhrono, jer se ipak sve izvršava po nekom redosledu - bili ste u pravu. Princip funkcionisanja jeste asinhron, ali sama srž programa je u potpunosti sinhrona i ne bi bilo dobro da je obrnuto (zamislite da se u toku izvršavanja linije 19 istovremeno izvrši i linija 10 - dobili bi smo izmešan tekst na ekranu).

Naravno, dalja diskusija ove teme prevazilaze okvire ovog rada. Ukoliko vas zanimaju detalji, pogledajte sledeći članak koji odlično opisuje principe rada:

<https://blog.risingstack.com/node-js-at-scale-understanding-node-js-event-loop/>.

Iz perspektive programera, odnosno naše perspektive - Node je asinhron i programe treba da pišemo imajući to u vidu.

Primeri

Do sada smo se upoznali i bavili internim stvarima. Sada ćemo prikazati nekoliko primera koji demonstriraju princip rada nekih od ugrađenih komponenti.

HTTP klijent

Prva komponenta koju ćemo demonstrirati jeste HTTP komponenta. Naime, za potrebe ovog primera povezaćemo se na <http://nodejs.org/dist/index.json> (adresa daje listu svih distribucija Nodea-a u JSON formatu), parsiraćemo rezultate i ispisati ih na izlazu. Pogledajmo kod:

```

1  const http = require('http');
2
3  http.get('http://nodejs.org/dist/index.json', function(response) {
4      let rawData = '';
5
6      response.on('data', function(chunk) {
7          rawData += chunk;
8      });
9
10     response.on('end', function() {
11         try {
12             let jsonData = JSON.parse(rawData);
13             console.log(jsonData);
14         } catch (e) {
15             console.log(e);
16         }
17     });
18 });

```

```

mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$ node primer8-1--http-klijent.js
[ { version: 'v7.7.4',
  date: '2017-03-21',
  files:
    [ 'aix-ppc64',
      'headers',
      'linux-arm64',
      'linux-armv6l',
      'linux-armv7l',
      'linux-ppc64le',
      'linux-x64',
      'linux-x86',
      'osx-x64-pkg',
      'osx-x64-tar',
      'src',
      'sunos-x64',
      'sunos-x86',
      'win-x64-exe',
      'win-x64-msi',
      'win-x86-exe',
      'win-x86-msi' ],
  npm: '4.1.2',
  v8: '5.5.372.42',
  uv: '1.11.0',
  zlib: '1.2.11',
  openssl: '1.0.2k',
  modules: '51',
  lts: false },
  { version: 'v7.7.3',

```

Slika 20. Povezivanje na nodejs.org i ispis procesiranih JSON rezultata

Primećujete standardnu strukturu - u pitanju je I/O operacija koja može potrajati, što znači da će ista biti procesirana asinhrono. Stoga, nakon prosleđivanja URL-a na liniji 3, prosleđuje se i povratna funkcija koja će se pozvati u trenutku kada se operacija izvrši.

Dalje, na liniji 6 počinjemo da osluškujemo na događaj `data`. Naime, Node preferira da koristi tokove podataka (eng. streams) i stoga će odgovor od servera stizati u komadima (eng. chunks). Da bi smo konstruisali celokupan odgovor, podatke ćemo čuvati i nadovezivati u promenljivu `rawData`.

Na kraju, kada je poslat kompletan odgovor od servera, emituje se “end” događaj. U ovom trenutku znamo da je stigao celokupan odgovor i možemo ga prikazati na ekranu.

Više informacija o HTTP komponenti možete pronaći na sledejeć adresi:

<https://nodejs.org/api/http.html>

Rad sa fajlovima

U sledećem primeru prikazaćemo program koji putem standardnog ulaza dobija ime direktorijuma koji želimo da isčitamo, zatim iterira kroz sve fajlove u tom direktorijumu i prikazuje njihov sadržaj na standardnom izlazu. Pogledajmo kôd:

```
1  const fs    = require('fs');
2  const path  = require('path');
3
4  // Parametri 0 i 1 sadrže putanju do programa koji se izvršava (nodejs)
5  // i ime skripte koja je pokrenuta (ova skripta). Stoga, treba nam treći
6  // parametar koji sadrži direktorijum koji želimo da isčitamo
7  const dir   = process.argv[2];
8
9  fs.readdir(dir, 'utf8', function(error, files) {
10     if (error) {
11         console.error(error);
12         return;
13     }
14
15     // Iteriramo kroz sve fajlove koji se nalaze u direktorijumu
16     files.forEach(function(file) {
17         // path.format() će nam vratiti punu putanju do fajla
18         let punaPutanja = path.format({
19             dir:    dir,
20             base:   file
21         });
22
23         fs.readFile(punaPutanja, 'utf8', function(error, data) {
24             if (error) {
25                 console.error(error);
26                 return;
27             }
28             console.log("-----\n-----\nSadržaj fajla %s je: \n\n%s\n\n", file, data);
29         });
30     });
31 });
32
```

```
mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$ node primer9-sinhrono-iscitavanje-fajlova.js ./txt-fajlovi/
Sadržaj fajla A.txt je:
Ovo je fajl A i ovo je njegov sadržaj ...

Sadržaj fajla B.txt je:
Ovo je fajl B. I ovo je njegov sadržaj ....

mihailojksimovic@Mihailos-MacBook-Pro-2:~/projects/nodejs (master)$
```

Slika 21. Primer isčitavanja fajlova iz direktorijuma

Nakon uključivanja potrebnih biblioteka, na liniji 7 uzimamo direktorijum koji nam je prosleđen putem komandne linije. Zatim koristimo `readdir()` metodu koja se koristi za iščitavanje fajlova u direktorijumu. Primetićete da se metoda kao argument prosleđuje povratna funkcija sa dva parametra, gde je prvi parametar greška, a drugi parametar lista fajlova. Ovo je standardni format koji koriste sve Node povratne funkcije - prvi parametar je uvek greška (ukoliko je ima), dok je drugi parametar podatak koji smo tražili.

Nakon toga gradimo punu putanju do fajla i na sličan način i na kraju isčitavamo i prikazujemo sadržaj fajla.

Bitno je napomenuti da se u ovom primeru za prikaz fajlova koristi baferovanje (eng. buffering), odnosno sadržaj celog fajla se prvo smesti u promenljivu `data`, a zatim se ispisuje na ekranu. Kao što smo već napomenuli, ovo nije poželjan način i treba ga izbegavati u korist tokova podataka. Pogledajmo izmenjeni primer:

```
1  const fs    = require('fs');
2  const path  = require('path');
3
4  // Parametri 0 i 1 sadrže putanju do programa koji se izvršava (nodejs)
5  // i ime skripte koja je pokrenuta (ova skripta). Stoga, treba nam treci
6  // parametar koji sadrži direktorijum koji želimo da isčitamo
7  const dir   = process.argv[2];
8
9  fs.readdir(dir, 'utf8', function(error, files) {
10     if (error) {
11         console.error(error);
12         return;
13     }
14
15     // Iteriramo kroz sve fajlove koji se nalaze u direktorijumu
16     files.forEach(function(file) {
17         // path.format() ce nam vratiti punu putanju do fajla
18         let punaPutanja = path.format({
19             dir:    dir,
20             base:   file
21         });
22
23         var tokPodataka = fs.createReadStream(punaPutanja, 'utf8');
24
25         tokPodataka.on('data', function(chunk) {
26             console.log("Sadržaj fajla %s je: \n", path.basename(tokPodataka.path));
27
28             console.log(chunk);
29         });
30     });
31 });
```

Slika 22. Primer iščitavanja fajlova korišćenjem tokova podataka

Kao što možemo videti, umesto `readFile()` iskorišćena je `createReadStream()` metoda. Ova metoda otvara tok podataka ka fajlu i emituje `data` događaj svaki put kada je deo podataka dostupan. Ostatak koda je ostao nepromenjen.

Zaključak

Iz prethodnih izlaganja možemo zaključiti da je Node.js odličan jezik za obradu jednostavnijih zahteva koji većinu vremena provedu čekajući na podatke, kao što su na primer web aplikacije.

Sa druge strane, operacije koje zahtevaju procesorsku snagu treba izbegavati i koristiti jezike predviđene za njih (npr. C++, Javu, itd.).

Treba imati u vidu i činjenicu da je iako ima izuzetno veliku zajednicu, u pitanju relativno mlad jezik koji je još uvek u razvoju.

Ukoliko želite da nastavite sa izučavanjem, možete početi od glavne stranice <http://nodejs.org>. Za pravljenje web aplikacija obavezno pogledajte Express framework - <https://expressjs.com/>, dok za poveću listu biblioteka možete posetiti <https://www.codementor.io/ashish1dev/list-of-useful-nodejs-modules-du107mev3>.

Odličan resurs za učenje kroz male zadatke možete pronaći na <https://github.com/workshopper/learnyounode>.

Na kraju, svi zainteresovani se mogu uključiti u razvoj i održavanje jezika putem GitHub-a na adresi <https://github.com/nodejs/node>.

Srećno!

Reference

<https://nodejs.org/en/>

<https://www.youtube.com/watch?v=ztspvPYybiY&t=2107s>

<https://blog.risingstack.com/node-js-at-scale-understanding-node-js-event-loop/>

<https://github.com/workshopper/learnyounode>

<https://github.com/MihailoJoksimovic/introduction-to-nodejs-code-camples>

<http://codewinds.com/blog/2013-08-04-nodejs-readable-streams.html>