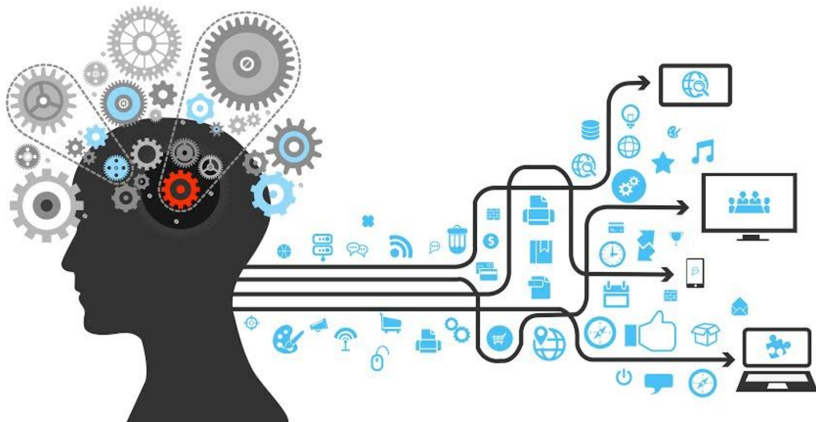
Univerzitet u Beogradu - Elektrotehnički fakultet

Node JS

Predavač: Prof. dr Dražen Drašković

Beograd, maj 2026. godine

Sadržaj



- Šta je NodeJS, npm i instalacija
- Moduli
- Osnovni rad sa konzolom
- Upotreba događaja, osluškivača, tajmera i povratnih poziva (*callback*)
- Upravljanje ulazima i izlazima i rad sa JSON objektima
- Pristup sistemu datoteka
- Implementiranje HTTP servisa, rad sa GET i POST zahtevima

Šta je Node.js?

- Node.js, kao međuplatformsko izvršno JavaScript (JS) okruženje, otovrenog koda, razvijen je sa ciljem da bude skalabilno serversko okruženje koje programerima omogućava da lakše premoste jaz između klijenta i servera.
- Okruženje koje se lako instalira, konfiguriše i primenjuje.
- Izgrađen na *Chrome V8 JS* pokretač (*engine*), istom pokretaču kao i za *Google Chrome.*
- Chrome V8 pokretač je napisan u C++ i odgovoran za kompajliranje JS koda u mašinski kod, omogućavajući brzo izvršavanje.
- Veoma popularan u mnogim kompanijama, zbog skalabilnosti, brzog razvoja i lakog održavanja. Koriste ga LinkedIn, eBay, Netflix, Uber, Groupon, PayPal, Walmart, i drugi.

Kada se koristi Node.js?

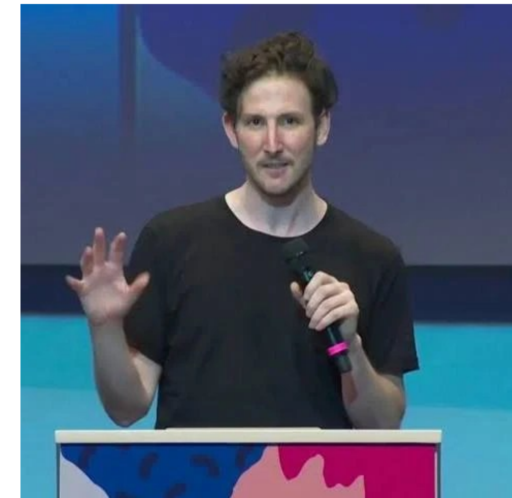
- Node.js omogućava **kreiranje veb servera i mrežnih alata**, koristeći JavaScript kod i **kolekciju modula** koji obrađuju različite osnovne funkcionalnosti.
- **Moduli** obrađuju ulazno/izlazne (I/O) operacije sistema datoteka, umrežavanje, binarne podatke (bafere), kriptografske funkcije, tokove podataka, itd.
- **Radni okviri** (eng. *frameworks*) mogu se koristiti za ubrzavanje razvoja veb aplikacija. Najčešće korišćeni radni okviri su:
  - *Express.js*, *Koa*, *NestJS* (standard za poslovne TS aplikacije) i *Fastify* (alternativa *Express.js*)
- Node.js nije ograničen samo na serversku stranu, već pruža brojne softverske alate za radne procese razvoja klijentskog (*frontend*) dela veb aplikacija.

Čemu služi Node.js?

- Zahvaljujući neblokirajućoj arhitekturi, vođenoj događajima, može da se koristi:
  - kao veb server i razvoj serverskog (*backend*) sloja veb aplikacija,
  - za veb servise (REST, pozadinski veb servisi, kao što su međudomenski i serverski zahtevi),
  - za razvoj aplikacija u realnom vremenu, kao što su chat-ovi, onlajn igre, striming servisi,
  - za izgradnju velikih i visoko skalabilnih sistema.
- Node.js u kombinaciji sa klijentskim MV* okvirom, nerelacionom NoSQL bazom podataka (npr. *MongoDB* ili *Apache CouchDB*) i JSON tehnologijom, nudi objedinjeni JavaScript razvojni stek, koji se može povezati sa nekom tehnologijom na klijentskoj strani (*Angular* ili *React*).
- JavaScript je cilj kompajliranja!
  JavaScript se koristi u raznim NoSQL bazama.
  JSON popularan za razmenu podataka.

Kako je nastao Node.js?

- Ryan Dahl, softverski inženjer, pokrenuo je Node.js projekat 2009. godine, frustriran trenutnim stanjem serverskih tehnologija u industriji.
- Osnovna premisa Node.js tehnologije: najveći broj veb aplikacije je ograničen na ulazno/izlazne (I/O) operacije.
- Aplikacije vezane za I/O operacije su ograničeni pristupom podacima.



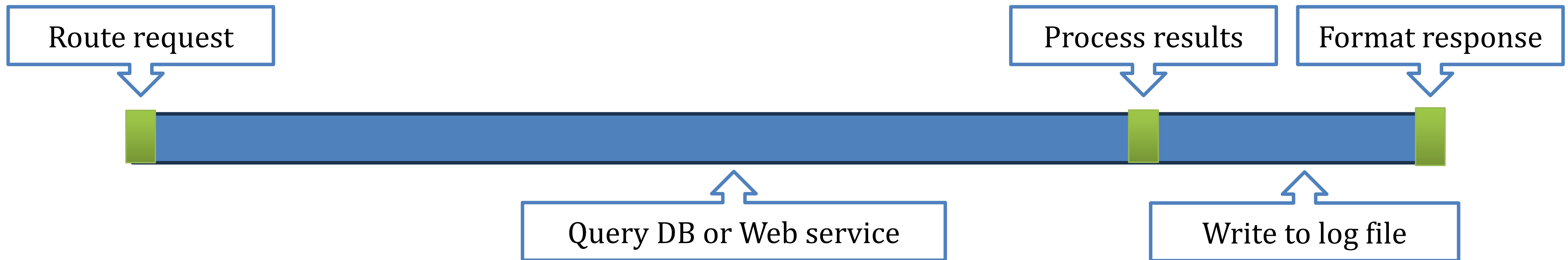
- To su aplikacije gde dodavanje više procesorske (CPU) snage ili RAM memorije često ne pravi veliku razliku – **usko grlo njihovih performansi je latencija I/O operacija**.

Operation	CPU Cycles
L1	3 cycles
L2	14 cycles
RAM	250 cycles
DISK	41 000 000 cycles
NETWORK	240 000 000 cyles

Tabela Latencije I/O operacija (2009).

Blokiranje ulazno/izlazne operacije

- U mnogim programskim jezicima, ulazno/izlazne operacije su blokirajuće – one blokiraju napredak programa dok se čeka na ulazno/izlazni zadatak kao što je čitanje sa čvrstog diska ili slanje mrežnog zahteva.




```
$result = mysql_query ('SELECT * FROM myTable');
print_r ($result);
```

Skaliranje sa nitima

- Ukoliko program blokira ulazno/izlazne operacije, šta server radi kada ima više zahteva za obradu?
- Tipičan pristup je **korišćenje višenitnosti**, tj. korišćenje jedne niti po konekciji, i podešavanja skupa niti (*thread pool*) za te konekcije.

Skaliranje sa nitima (2)

- Iako nam ovaj pristup omogućava **skaliranje dodavanjem više niti**, svaka nit i dalje provodi većinu vremena čekajući na ulazno/izlazne operacije, a ne obrađujući logiku aplikacije.
- Nažalost, kontinuirano dodavanje niti uvodi opterećenje prebacivanja konteksta i koristi značajnu količinu memorije za održavanje stekova izvršavanja.

Skaliranje sa Node.js

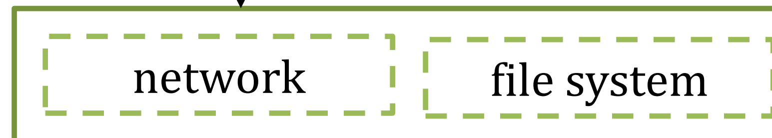
- Node.js koristi **jednu nit**, koristeći **neblokirajući ulaz/izlaz** – svaka funkcija koja vrši ulaz/izlaz se obrađuje asinhrono, a zatim pokreće povratni poziv.

```
db.query("select..", function(result) {
   doSomething(result);
});
nextTask();
```

Obrađuju mnogo istovremenih zahteva u jednom procesu/niti.

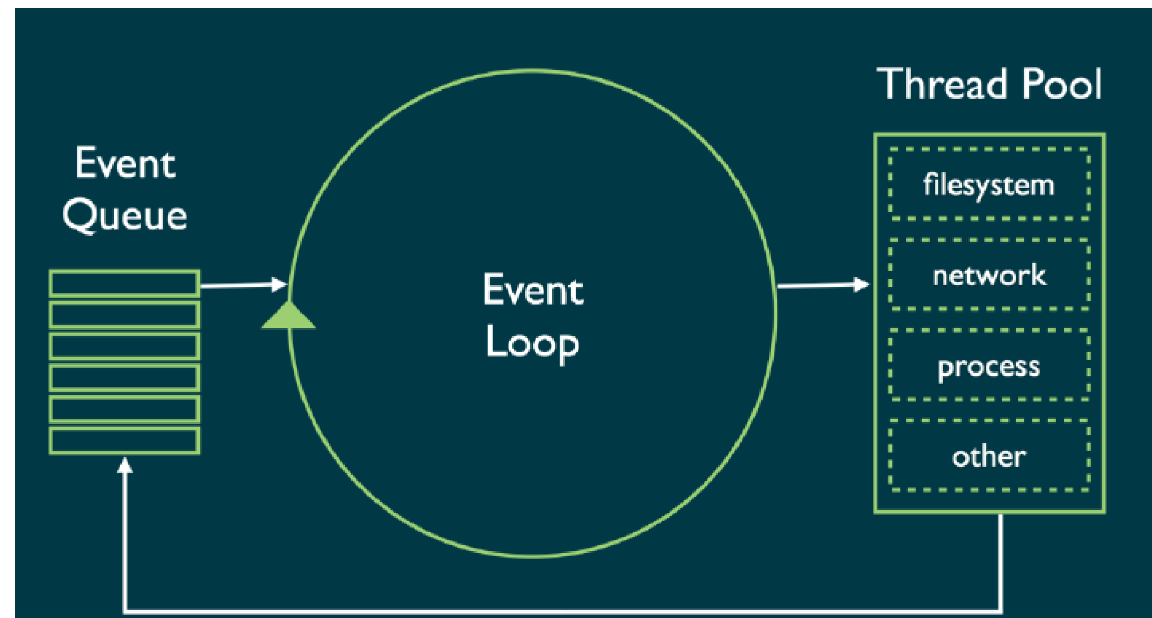
Proces #1

Skup niti i asinhroni I/O APIs



Skaliranje sa Node.js (2)

- Korišćenjem jedne niti sa petljom događaja, umesto više niti, Node.js podržava desetine hiljada istovremenih veza (konekcija) bez troškova promene konteksta niti.
- Zadaci u petlji događaja moraju se brzo izvršavati kako bi se izbeglo blokiranje reda – budite oprezni sa zadacima koji intenzivno koriste procesor (CPU).



Node.js

Moduli

Moduli u Node.js

- Moduli su softverske komponente koje se mogu ponovo koristiti i čine gradivne blokove aplikacije.
- Modularno programiranje je tehnika dizajna softvera koja naglaša razdvajanje funkcionalnosti programa na nezavisne, zamenljive module, tako da svaki pokriva samo jedan aspekt željene funkcionalnosti.
- Moduli treba da budu FIRST:
  - Focused = Fokusirani
  - Independent = Nezavisni
  - Reusable = Reupotrebljivi
  - Small = Mali
  - Testable = Testabilni

Moduli u modernom JavaScript jeziku

- Prednosti modularnog programiranja uključuju:
  - Lakše razumevanje velikog sistema
  - Pojednostavljeno debagovanje
  - Razdvajanje logike
  - Lepše mogućnosti održavanja koda
  - Ponovna upotreba koda
- ECMAScript 5 nije imao ugrađenu podršku za module, dok ES6 i noviji standardi uvode sintaksu modula.
- Prednosti modernih ECMAScript modula (ESM) :
  - Standardizovani, radi u veb pregledaču i *Node.js* (sa type="module").
  - Jasna kontrola vidljivosti (export vs import).
  - Podržava asinhrono učitavanje i tree-shaking (proces eliminisanja mrtvog koda) za optimizaciju veličine paketa.

Node.js nativno teži da premosti jaz između CommonJS i ESM,
kao i da pruža inicijalnu prirodnu/nativnu podršku za TS bez eksternih transpilera (kao što su ts-node ili tsx).

Moduli u modernom JavaScript jeziku

<table>
<tr><th>Definisanje / izvoz modula</th><th>Uvoz modula</th><th>Podrazumevani (default) izvoz</th><th>Dinamički import (asinhrono učitavanje modula)</th></tr>
<tr>
<td><pre>//math.js

export function add(a, b) {
  return a + b;
}

export const PI = 3.14159;</pre></td>
<td><pre>//drugiskript.js

import { add, PI } from
'./math.js';

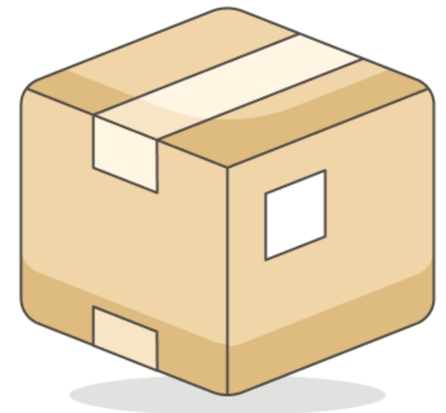
console.log(add(2, 3));
// 5
console.log(PI);
// 3.14159</pre></td>
<td><pre>// logger.js

export default
function log(msg) {
  console.log(msg);
}

// app.js
import log from
'./logger.js';
log('Hello World!');</pre></td>
<td><pre>async function loadModule() {
  const math = await import
                        ('./math.js');
  console.log(math.add(5, 7));
}
loadModule();</pre></td>
</tr>
</table>

NPM - *Node Package Manager*

- **Node.js koristi npm (Node Package Manager), najveću softversku platformu na svetu.**
- **npm** olakšava JavaScript programerima deljenje i ponovnu upotrebu koda u obliku modula.
- **npm** dolazi unapred instaliran sa Node distribucijama.
- **npm** se pokreće kroz komandnu liniju i omogućava preuzimanje modula iz javnog registra paketa koji se održava na veb linku http://npmjs.org



Osnovni ugrađeni moduli kod Node.js

npm (*Node Package Manager*) omogućava slobodno korišćenje milion modula - otvorenih paketa i biblioteka

- Buffer
- C/C++ Addons
- Child Processes
- Cluster
- Console
- Crypto
- Debugger
- DNS
- 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

Osnovne komande i direktorijumi

INSTALACIJA

Download Node.js®

Get Node.js® v26.2.0 Current for Windows using Docker with npm

```
1  # Docker has specific installation instructions for each operating system.
2  # Please refer to the official documentation at https://docker.com/get-started/
3
4  # Pull the Node.js Docker image:
5  docker pull node:26-slim
6
7  # Create a Node.js container and start a Shell session:
8  docker run -it --rm --entrypoint sh node:26-slim
9
10 # Verify the Node.js version:
11 node -v # Should print "v26.2.0".
12
13 # Verify npm version:
14 npm -v # Should print "11.13.0".
```

PowerShell — Copy to clipboard

Docker is a containerization platform. If you encounter any issues please visit Docker's website

Or get a prebuilt Node.js® for Windows running a x64 architecture.

Windows Installer (.msi) | Standalone Binary (.zip)

Osnovne komande i direktorijumi

Provera verzija:
node -v
npm -v

- **node** - komanda za izvršavanje skripta.
- **npm** - komanda za upravljanje Node.js modulima.
- Moduli - delovi reupotrebljivog koda ili biblioteke, koji se mogu ponovo koristiti i obuhvataju određene funkcionalnosti. Moduli mogu biti pojedinačne JS datoteke ili kolekcije datoteka organizovane unutar direktorijuma, obično opisanih datotekom *package.json* (fajl koji sadrži informativne metapodatke i kontrolne metapodatke).
- Svi moduli moraju da budu instalirani u okviru foldera *node_modules*.
- Svi registrovani moduli nalaze se u *Node Package Registry* (NPM registar dostupan na: https://npmjs.com)
- *Node Package Manager* - program komandne linije preko koga na lak način možemo manipulisati modulima (veza između registra i razvojnog okruženja).
- Industrija: koriste se i neki brži i efikasniji menadžeri paketa poput **pnpm** i **Yarn**, kao i alati poput **Bun** koji služi i kao izvršni alat (*runtime*) i kao paket menadžer.

npm najvažnije komande (1)

Komanda	Opis	Primer
npm init	Inicijalizuje novi Node.js projekat, time što kreira datoteku *package.json* za upravljanje metapodacima i zavisnostima.	npm init
npm install	Instalira sve zavisnosti navedene u datoteci *package.json* u okviru lokalnog direktorijuma *node_modules*.	npm install
npm install <package-name>	Instalira specifični paket (i dodaje zavinost u datoteku *package.json*).	npm install express
npm install -g <package-name>	Instalira paket globalno, čineći ga dostupnim iz bilo kog direktorijuma vašeg sistema.	npm install –g express
npm uninstall <package-name>	Uklanja paket iz zavisnosti projekta i briše ga iz direktorijuma *node_modules*.	npm uninstall express
npm update	Ažurira sve pakete navedene u datoteci *package.json* na njihove najnovije verzije unutar navedenih opsega verzija ili poseban paket.	npm update npm update typescript
npm ls	Izlistava sve instalirane pakete u trenutnom projektu, prikazujući njihovo stablo zavisnosti.	npm ls (ili: npm list)
npm outdated	Proverava zastarele pakete u zavisnostima vašeg projekta.	npm outdated
npm run <script-name>	Izvršava prilagođen skript koji je naveden u posebnom odeljku sa skriptama u okviru datoteke *package.json*.	npm run start (start definisano u *package.json*)

npm najvažnije komande (2)

Komanda	Opis	Primer
npm search <keyword>	Pretražuje npm registar za pakete koji odgovaraju traženoj ključnoj reči.	npm search react components
npm audit	Pokreće bezbedonosnu reviziju zavisnosti vašeg projekta kako bi identifikovao potencijalne ranjivosti.	npm audit
npm ci	Vrši čistu instalaciju zavisnosti, osiguravajući da su instalirane tačne verzije navedene u *package-lock.json*, što je posebno korisno u CI/CD okruženjima.	npm ci
npm start	Skraćenica za: npm run start	npm start
npm stop	Pokreće stop skriptu iz *package.json*.	npm stop
npm restart	Pokreće restart skriptu iz *package.json*. Ukoliko nema eksplicitno navedene restart skripte, ali postoje start i stop skripte, onda će se prvi pozvati stopiranje, pa startovanje aplikacije.	npm restart

Primer datoteke *package.json* sa skriptama za start i stop:

```
{
  "scripts": {
    "start": "node app.js",
    "stop": "node cleanup.js"
  }
}
```

Direktive u *package.json* (1)

Direktiva	Opis	Primer
name	Naziv paketa/projekta (mora biti jedinstven ako se objavljuje na npm).	"name": "moja-najbolja-aplikacija"
version	Označava trenutnu verziju paketa (format: major.minor.patch).	"version": "1.0.0"
description	Kratak opis projekta, koristan za dokumentaciju i npm.	"description": "Nova ETF app"
main	Određuje glavnu ulaznu tačku aplikacije ili modula.	"main": "index.js"
type	Definiše da li se koristi **CommonJS** ("commonjs") ili **ES Modules** ("module").	"type": "module"
scripts	Sekcija za definisanje komandi koje se izvršavaju pomoću npm run.	"scripts": { "start": "node index.js", "test": "node moj-test.js" }
keywords	Lista ključnih reči za lakše pretraživanje na npm.	"keywords": ["etf", "rti", "si", "node"]

dependencies	Lista runtime zavisnosti (instaliraju se sa npm install).
	Džoker znakovi su x.
		"dependencies": {
		"express": "latest",
		"connect": "2.x.x",
		"cookies": "*"
		}

author	Ime autora projekta.	"author": "drazen@etf.bg.ac.rs"

Direktive u *package.json* (2)

Direktiva	Opis	Primer
licence	Tip licence (npr. MIT, ISC, GPL).	"license": "MIT"
engines	Ograničenja verzije Node.js ili npm koje paket podržava.	"engines": { "node": ">=20.0.0" }
repository	Info o repozitorijumu projekta (korisno za GitHub projekte).	"repository": { "type": "git", "url": "http://rti.etf.rs/srv.git" }
bugs	Link ka prijavljivanju defekata (bagova).	"bugs": { "url": "https://github.com/user/myapp/issues" }
homepage	Link (URL) ka zvaničnoj stranici ili dokumentaciji.	"homepage": "https://myapp.example.com"
exports	Precizna kontrola koje fajlove/moduli su dostupni korisnicima paketa.	"exports": { ".": "./index.js", "./utils": "./src/utils.js" }
bin	Definiše izvršne fajlove (CLI komande) koje se instaliraju globalno.	"bin": { "mycli": "./bin/cli.js" }
devDependencies	Zavisnosti potrebne samo u razvoju (testiranje, build alati).	"devDependencies": { "jest": "^29.0.0" }
peerDependencies	Očekivane zavisnosti koje korisnik treba da instalira ručno (korisno za biblioteke).	"peerDependencies": { "react": ">=18" }

Primer *package.json* fajla

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "Demo Node.js application",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "test": "node test.js"
  },
  "keywords": ["node", "api", "rti", "si",],
  "author": "Drazen Draskovic <drazen@etf.bg.ac.rs>",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.0.0"
  },
  "engines": {
    "node": ">=20.0.0"
  }
}
```

Primer *JS* fajla u Node.js aplikaciji

```
// utils.js
// Funkcija za računanje zbira elemenata niza
function sumArray(arr) {
  return arr.reduce((acc, val) => acc + val, 0);
}

// Funkcija za pronalaženje maksimuma u nizu
function findMax(arr) {
  return Math.max(...arr);
}

// Funkcija za formatiranje trenutnog datuma
function getFormattedDate() {
  const now = new Date();
  return now.toISOString().split("T")[0]; // npr. "2026-05-22"
}

// Eksportujemo funkcije kako bi bile dostupne iz drugih fajlova
module.exports = {
  sumArray,
  findMax,
  getFormattedDate,
};
```

Korišćenje *JS* fajla u okviru drugog fajla Node.js aplikacije

```
// app.js
const utils = require("./utils");

// Primer niza
const numbers = [5, 12, 8, 130, 44];

console.log("Zbir elemenata:", utils.sumArray(numbers));
console.log("Maksimum:", utils.findMax(numbers));
console.log("Današnji datum:", utils.getFormattedDate());
```

Pokretanje skripte:

node app.js

Rad sa konzolom

- Interfejs komandne linije **(CLI)** - npm pruža CLI alatu da programerima omogući:
  - Instaliranje paketa:
    koristeći komande poput **npm install <package-name>**, npm preuzima navedeni paket iz registra i smešta ga u direktorijum node_modules unutar projekta.
  - Upravljanje zavisnostima:
    npm automatski prati instalirane pakete i njihove verzije u datotekama *package.json* i *package-lock.json*, obezbeđujući konzistentna okruženja u različitim podešavanjima za razvoj aplikacije.
  - Objavljivanje paketa: programeri mogu da kreiraju i objavljuju sopstvene module u npm registru, čineći ih dostupnim drugim programerima/inženjerima za korišćenje.
  - Pokreću skripte: npm može da izvršava prilagođene skripte definisane unutar datoteke *package.json*, u sekciji „scripts“, automatizujući zadatke poput testiranja, izgradnje (*building*) ili objavljivanja (*deploying*) projekata.

Funkcije u radu sa konzolom (1)

Komanda / funkcija	Opis funkcije
console.log(data)	Štampa standardnu poruku na konzolu (najčešće korišćeno za ispis informacija). Promenljiva podataka može da bude string ili objekat, a mogu biti i dodatni parametri: console.log("Imamo %d studenata", 125);
console.info(data)	Kao log(), ali semantički naglašava da je informacija.
console.warn(data)	Ispisuje upozorenje (žutom bojom u većini terminala).
console.error(data)	Ispisuje grešku (crvenom bojom u većini terminala), koristi stderr.
console.debug()	Ispisuje poruku namenjenu za debug, često ekvivalent log(), ali može biti filtriran.
console.trace(label)	Ispisuje stack trace (putanju poziva funkcija do mesta gde je pozvan).
console.table(data)	Formatira niz ili objekat u tabelarni prikaz u konzoli.
console.dir(obj)	Ispisuje string JavaScript objekta u konzolu, na primer: console.dir({ name: "Drasko", role:"nastavnik" });
console.dirxml(obj)	Prikazuje XML/DOM čvorove u čitljivom obliku (više u browseru, u Node-u sličan dir()).
console.assert (expression, [message])	Ispisuje poruku i stanje steka u konzolu ako izraz daje vrednost *false* kao rezultat.

Funkcije u radu sa konzolom (2)

Komanda / funkcija	Opis funkcije
console.count(label)	Broji koliko puta je pozvan sa istim labelom i ispisuje broj.
console.countReset(label)	Resetuje brojač kreiran sa console.count(label).
console.time (label)	Pokreće tajmer sa labelom (za merenje vremena izvršavanja koda).
console.timeLog(label)	Ispisuje proteklo vreme od time() za dati label.
console.timeEnd(label)	Zaustavlja tajmer i ispisuje ukupno vreme izvršavanja koda (odnosno određuje razliku između tekućeg vremena i vremenske oznake dodeljene stringu label)
console.group(label)	Otvara uvučenu grupu logova u konzoli.
console.groupCollapsed (label)	Otvara grupu koja je podrazumevano zatvorena.
console.groupEnd()	Zatvara poslednju otvorenu grupu logova.
console.clear()	Čisti konzolu (zavisi od terminala, uvek ne radi 100%).

Primeri funkcija u radu sa konzolom (1)

Funkcija	Primer
console.log(data)	console.log("Hello, world!");
console.info(data)	console.info("Info: Server started on port 3000");
console.warn(data)	console.warn("Warning: Disk space is low!");
console.error(data)	console.error("Error: Cannot connect to database!");
console.debug()	console.debug("Debug: x =", 42);
console.trace(label)	function test(){ console.trace("Trace point"); } test();
console.table(data)	console.table([{id:1, name:"Ana"},{id:2, name:"Marko"}]);
console.dir(obj)	console.dir({user:{id:1, name:"Ana"}}, {depth: null});
console.dirxml(obj)	console.dirxml({foo: "bar", nested:{a:1}});
console.assert (expression, [message])	console.assert(2+2===5, "Math error!");

Primeri funkcija u radu sa konzolom (2)

Funkcija	Primer
console.count(label)	console.count("click"); console.count("click");
console.countReset(label)	console.count("loop"); console.countReset("loop"); console.count("loop");
console.time (label)	console.time("db"); setTimeout(()=>console.timeEnd("db"),500);
console.timeLog(label)	console.time("process"); setTimeout(()=>console.timeLog("process"),200);
console.timeEnd(label)	console.time("loop"); for(let i=0;i<1e6;i++); console.timeEnd("loop");
console.group(label)	console.group("Auth"); console.log("Step 1"); console.log("Step 2"); console.groupEnd();
console.groupCollapsed (label)	console.groupCollapsed("Details"); console.log("Hidden by default"); console.groupEnd();
console.groupEnd()	console.group("Test"); console.log("Inside group"); console.groupEnd();
console.clear()	console.clear();

Programiranje vođeno događajima (*Event Driven Programming*)

- Programska paradigma u kojoj tok programa određuju događaji, kao što su korisničke akcije, izlazi senzora, ili poruke od drugih programa.
- Dominantna paradigma koja se koristi u grafičkim korisničkim interfejsima (GUI) i JavaScript aplikacijama koje su usmerene na izvršavanje određenih radnji, kao odgovor na korisnički unos.
- Pisanje programa vođenog događajima je lako, ako programski jezik pruža apstrakcije visokog nivoa, kao što su završeci (*closures*).
- *JavaScript* predstavlja jezik vođen događajima – oduvek se bavio interakcijom korisnika, koristio je petlju događaja za osluškivanje događaja, i funkcije povratnog poziva (*callback*) za njihovu obradu.

Niti vs Događaji

```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

Implementacija:

•Promena niti i raspoređivač.

```
startRequest(socket);
listen("requestAvail",processRequest);
listen("processDone",sendReplyToSock);
```

Implementacija:

•Procesiranje događaja u redu za čekanje.

Niti vs Događaji sa povratnim pozivom

```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

Implementacija:

•Promena niti i raspoređivač.

```
readRequest(socket, function(request)
{
   processRequest(request,
      function (reply) {
         sendReply(socket, reply);
      });
});
```

Implementacija:

•Procesiranje događaja u redu za čekanje.

Red događaja

Unutrašnja petlja:

```
while (true) {
    if (!eventQueue.notEmpty()) {
        eventQueue.pop().call();
    }
}
```

- Pri hvatanju događaja, nikada se ne dešava wait/block.
- Primer *readRequest(socket)*;

1. `launchReadRequest(socket);`
   // odmah vraća rezultat
2. Kada se čitanje završi:
   `eventQueue.push(readDoneEventHandler);`

Programiranje sa događajima i povratnim pozivima

- Ključne razlike:
- *Niti – blokiranje i čekanje je transparentno*
- *Događaji – blokiranje i čekanje zahteva povratne pozive (callback)*

- Treba da razmišljamo ovako:
- Ako kod ne blokira – isto kao programiranje niti
- Ako kod blokira (ili treba da blokira) – potrebno je podesiti povratni poziv
- Često ono što je bila RETURN naredba, postaje poziv funkcije.

Primer

- Niti

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

- Povratni pozivi

```
step1(function (r1) {
    console.log('step1 done', r1);
    step2(r1, function (r2) {
        console.log('step2 done', r2);
        step3(r2, function (r3) {
            console.log('step3 done',r3);
        });
    });
});
console.log('All Done!'); // Wrong!
```

Radi za neblokirajuće pozive
u oba stila,
i niti i povratni pozivi.
A za blokirajuće pozive?

Primer

- Niti

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

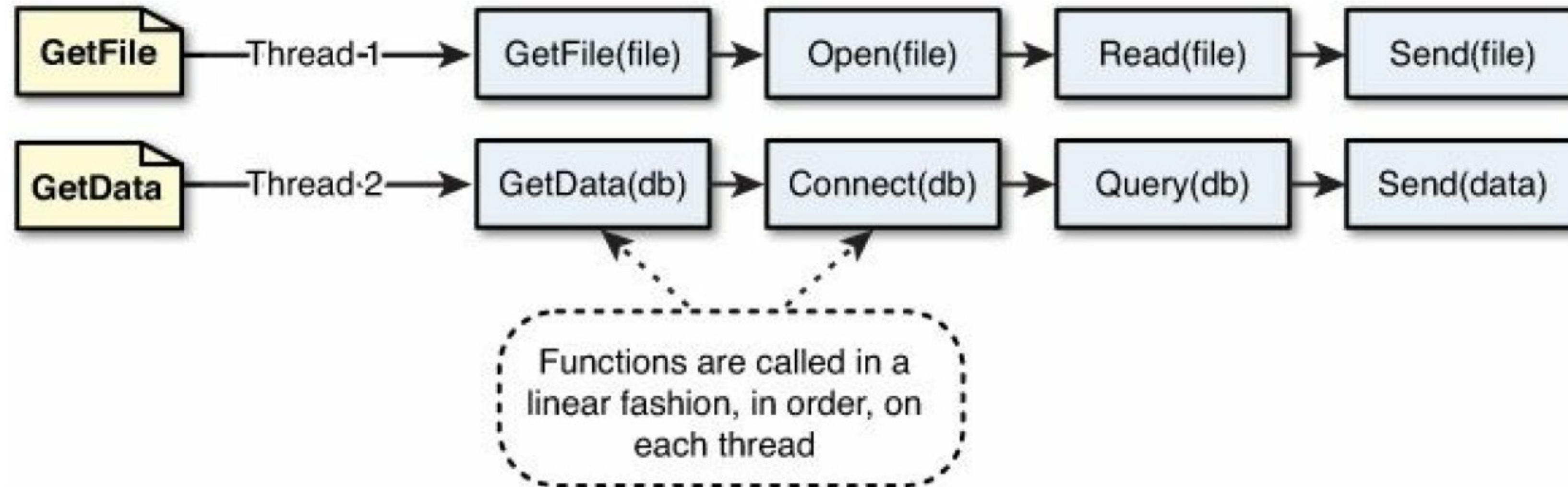
- Povratni pozivi

```
step1(function (r1) {
    console.log('step1 done', r1);
    step2(r1, function (r2) {
        console.log('step2 done', r2);
        step3(r2, function (r3) {
            console.log('step3 done',r3);
            console.log('All Done!');
        });
    });
});
```

Node.js vođen događajima

- *Node.js* donosi programiranje vođeno događajima na serversku stranu.
- Osnovni dizajn iza *Node.js* je da se aplikacija izvršava u jednoj niti,
  a svi događaji se obrađuju asinhrono.
- *Node.js* koristi arhitekturu petlje događaja,
  kako bi programiranje visoko skalabilnih servera bilo jednostavno i bezbedno.
- Konkurentnost programiranja je teška – *Node.js* zaobilazi ovaj izazov, ali još uvek nudi impresivne performanse.
- Da bi podržao pristup petlje događaja, *Node.js* isporučuje skup neblokirajućih I/O modula – to su interfejsi ka stvarima kao što su fajl sistem ili baze podataka, koji funkcionišu na način vođen događajima.

Node.js neblokirajući I/O model događaja



- Zahtevi se dodaju u red događaja, a zatim ih preuzima jedna nit koja pokreće petlju događaja. Petlja redom preuzima najvišu stavku u redu događaja, pa sledeću, itd.
- Kada se izvršava kod koji više nije aktivan ili koji sadrži sinhrone I/O operacije, funkcija se ne poziva direktno,
  već se stavlja u red događaja zajedno sa povratnim pozivom.
- Kada se izvrše svi događaji u redu, *Node.js* aplikacija se zatvara.

Obrada dva zahteva na modelu vođenim događajem - primer

GetFile
GetData
Node.js Application
Event Queue
(GetFile(file), Open(file))
(GetData(db), Connect(db))
(Open(file), Read(file))
(Connect(db),Query(db))
(Read(file),Send(file))
(Send(file),none)
(Query(db),Send(data))
(Send(data),none)
When the function is executed, Node.js places the callback in the event queue. The order is based on how fast functions finish.

- Primeri I/O operacija:
  - Čitanje datoteke
  - Upiti nad bazom podataka
  - Socket zahtev
  - Pristup udaljenom servisu

Model događaja sa skladištem niti

Requests and other asynchronous work the Node.js application does are placed on the event queue

The event loop retrieves a thread from the thread pool and executes the handler function

Thread Pool

Request

Request

Request

Node.js Application

Event Queue

(function,callback)

(function,callback)

(function,callback)

(function,callback)

(function,callback)

Event Loop

Blocking

Thread

Thread

Thread

Thread

Thread

Thread

Response

Callback

Callback

Function

Thread

The callback function will either send a response or schedule additional asynchronous tasks on the event queue

When the blocking callback completes, the thread is released to the thread pool

Načini prosleđivanja funkcije povratnog poziva

- Pozvati jednu od sinhronih I/O operacija iz biblioteke poziva (npr. pisanje datoteke ili povezivanje sa bazom podataka)
- Dodati ugrađeni osluškivač događaja u ugrađeni događaj (npr. *http.request* ili *server.connection*)
- Kreirati sopstvene emitere događaja u koje ćete dodati prilagođene "osluškivače"
- Upotrebiti opciju *process.nextTick* da biste zakazali izvršenje poslova koji će biti preuzeti u sledećoj iteraciji petlje događaja.
- Upotrebiti tajmere da biste zakazali izvršenje poslova nakon određenog vremena ili u intervalima (funkcije *setTimeout*/*setInterval*)

Implementiranje tajmera kod Node.js

- Tajmer isteka vremena
  - koriste se za odlaganje poslova na određeno vreme
  - kada istekne vreme, funkcija povratnog poziva se izvršava i tajmer nestaje
  - koriste se samo za poslove koji treba da se izvrše jednom
- Intervalni (periodični) tajmer
  - koriste se za izvršavanje poslova u određenim vremenskim intervalima
  - kada zadato vreme kašnjenja istekne, funkcija povratnog poziva se izvršava i zatim se ponavlja nakon određenog vremenskog intervala
  - koriste se za poslove koji se redovno izvršavaju
- Neposredni tajmer
  - koriste se za izvršavanje funkcije čim se izvrše povratni pozivi događaja ulaza/izlaza, ali pre nego što se izvrše događaji timeout i interval
  - važni su da bi se prepustili segmenti koji se dugo izvršavaju drugim povratnim pozivima, da ne bi došlo do "zamrzavanja" događaja I/O

Tajmer isteka vremena

- Kreiraju se pomoću:
  `setTimeout (callback, delayMilliseconds, [args]);`
- Primer:
  `setTimeout (mojaFunkcija, 1000);`
- Ova funkcija se jednom pozove, nakon 1 sekunde.
- Ova funkcija vraća ID objekta tajmera, a da bi se otkazala funkcija, ovaj ID može da se prosledi f-ji clearTimeout(id) u bilo kom trenutku pre nego što istekne taj broj milisekundi:

```
id = setTimeout (mojaFunkcija, 5000);
...
clearTimeout (id);
```

Tajmer isteka vremena – primer (1)

Osnovni primer:

```
setTimeout(() => {
  console.log("Pozdrav nakon 2 sekunde!");
}, 2000);
```

Primer sa prosleđenim argumentom:

```
function pozdrav(ime) {
  console.log(`Zdravo, ${ime}!`);
}

setTimeout(pozdrav, 1500, "Nina"); //ispis teksta nakon 1.5s
```

Tajmer isteka vremena – primer (2)

```
function mojafunkcija(consoleTimer){
   console.timeEnd(consoleTimer);
}

console.time("DveSekunde");
setTimeout(mojafunkcija, 2000, "DveSekunde");
console.time("JednaSekunda");
setTimeout(mojafunkcija, 1000, "JednaSekunda");
console.time("PetSekundi");
setTimeout(mojafunkcija, 5000, "PetSekundi");
console.time("50mili");
setTimeout(mojafunkcija, 50, "50mili");
```

Izlazi će biti prikazivani nakon ~50 milisekundi, 1000, 2000 i 5000 milisekundi:

50mili: 51.123ms
JednaSekunda: 1001.456ms
DveSekunde: 2002.789ms
PetSekundi: 5003.210ms

Intervalni (periodični) tajmer

- Kreiraju se pomoću:
  `setInterval (callback, delayMilliseconds, [args]);`
- Primer:
  `setInterval (mojaFunkcija, 1000);`
- Ova funkcija se periodično poziva nakon svake sekunde.
- Ova funkcija vraća ID objekta tajmera, a da bi se otkazala funkcija, ovaj ID može da se prosledi f-ji clearInterval(id) u bilo kom trenutku pre nego što istekne taj broj milisekundi:

```
id = setInterval (mojaFunkcija, 5000);
...
clearInterval (id);
```

Intervalni (periodični) tajmer - primer

```
var x = 0, y = 0, z = 0;
function prikaziVrednosti() {
   console.log("X=%d; Y=%d; Z=%d", x, y, z);
}
function azurirajX() {
   x += 1;
}
function azurirajY() {
   y += 1;
}
function azurirajZ() {
   z += 1;
   prikaziVrednosti();
}

setInterval(azurirajX, 500);   // na svakih 0.5 sekundi povećava X
setInterval(azurirajY, 1000);  // na svakih 1 sekundu povećava Y
setInterval(azurirajZ, 2000);  // na svakih 2 sekunde povećava Z i ispisuje sve
```

Izlaz poziva funkcija azurirajZ(). Funkcija azurirajX() povećava vrednost x tačno 2x brže od y, a funkcija azurirajY() povećava vrednost y tačno 2x brže od z.

Izlaz:
X=4; Y=2; Z=1
X=8; Y=4; Z=2
X=12; Y=6; Z=3
X=16; Y=8; Z=4
...

Neposredni tajmer

- Kreiraju se pomoću:
  `setImmediate (callback, [args]);`
- Kada pozove funkcije setImmediate(), funkcija povratnog poziva se smešta u red događaja i pojavljuje se jednom u svakoj iteraciji kroz petlju reda događaja, nakon što su događaji I/O pozvani.
- Ova funkcija vraća ID objekta tajmera.
  Pre nego što se preuzme iz reda događaja, ovaj ID može da se prosledi f-ji clearImmediate(id):

```
id = setImmediate (mojaFunkcija);
...
clearInterval (id);
```

Dereferenciranje tajmera iz petlje događaja

- Kada su funkcije događaja jedini događaji koji su ostali u redu događaja, Node.js ima mehanizam kojim se ukazuje događaju da ne treba da nastavi (ako je jedini u redu).
- Primer derefenciranja tajmera:

```
function mojaFunkcija() {
   console.log("Pozvana funkcija!");
}

let mojInterval = setInterval(mojaFunkcija, 1000);
mojInterval.unref();
```

- Pojava velikog broja unref() funkcija može nepovoljno da utiče na performanse Vašeg koda, pa ih koristiti umereno.

Upotreba funkcije nextTick

- Specijalna funkcija za zakazivanje izvršenja poslova u redu događaja: **`process.nextTick(callback)`**
- Pomoću ove funkcije se zakazuje izvršavanje poslova koji mogu da se pokrenu u sledećoj iteraciji kroz petlju događaja (pre nego što događaj petlje pređe na sledeću iteraciju).
- Za razliku od funkcije *setImmediate()*, ova funkcija se izvršava pre nego što se aktiviraju događaji ulaza/izlaza.
- Dakle, ako imaš asinkroni kod (timere, IO, promise-e), process.nextTick **uzima prioritet** i izvršava se pre njih.
- Da ne bi došlo do "zamrzavanja" događaja *Node.js* ograničava broj događaja ove funkcije: process.maxTickDepth = 1000
- Ako zoveš rekurzivno ovu f-ju, zaglavljuje se petlja događaja (ostali callback nikada ne dođu na red).

Upotreba funkcije nextTick - primer

```
console.log("Start");

setTimeout(() => {
  console.log("setTimeout");
}, 0);

setImmediate(() => {
  console.log("setImmediate");
});

process.nextTick(() => {
  console.log("nextTick");
});
console.log("End");
```

Mogući izlaz:

Start
End
nextTick
setTimeout
setImmediate

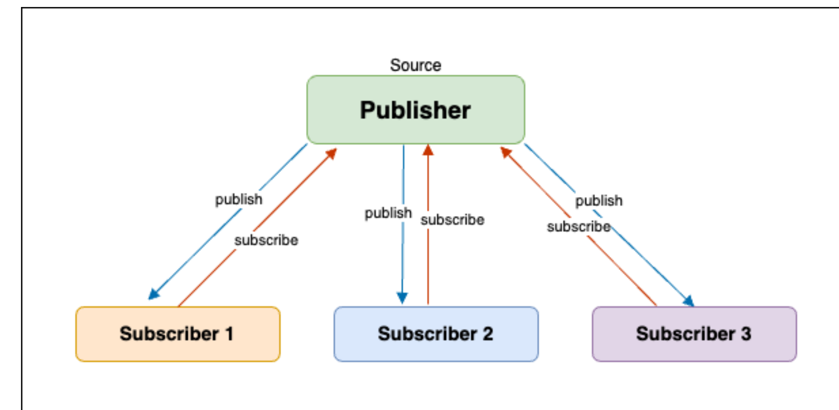
Zaključak:
process.nextTick uvek ide pre ostalih *callback*-ova, čak i pre setTimeout(..., 0) i setImmediate.

Kada se koristi nextTick?

1. Kada treba da završiš inicijalizaciju pre nego što se pozovu event handleri.
2. Kada želiš da izbegneš stack overflow kod rekurzivnih asinhronih poziva.

Obrazac Slušalac/Emiter

- Pri programiranju sa događajima (a ne sa nitima), često se koristi obrazac slušalac/emiter (*Listener*/*Emitter*).
- Slušalac (*Listener*) je funkcija koja se poziva kada se događaj signalizira.
  - Treba da bude upoznat sa DOM programiranjem (addEventListener)
- Emiter (*Emitter*) treba da signalizira da se događaj dogodio.
  - Emituje događaj koji uzrokuje poziv svih funkcija slušaoca.
- *Event Emitter* radi po obrascu publisher -> subscriber



Implementiranje emitera događaja

- Događaji se emituju pomoću objekta **EventEmitter** (iz modula events).
- Funkcija **emit(eventName, [args])** aktivira događaj eventName i dodaje sve argumente koji su zadati.

Objašnjenje:

- emit("nekiDogadjaj") : aktivira sve callbackove registrovane za taj događaj.
- on("nekiDogadjaj", callback) : registruje callback koji se poziva kada se emit-uje događaj.
- Nasleđivanje EventEmitter omogućava objektu da ima sopstvene događaje, baš kao što smo pokazali sa MojObjekat.

```
const EventEmitter = require('events');

class MojObjekat extends EventEmitter {
  constructor() {
    super(); // poziva konstruktor EventEmitter
  }
}

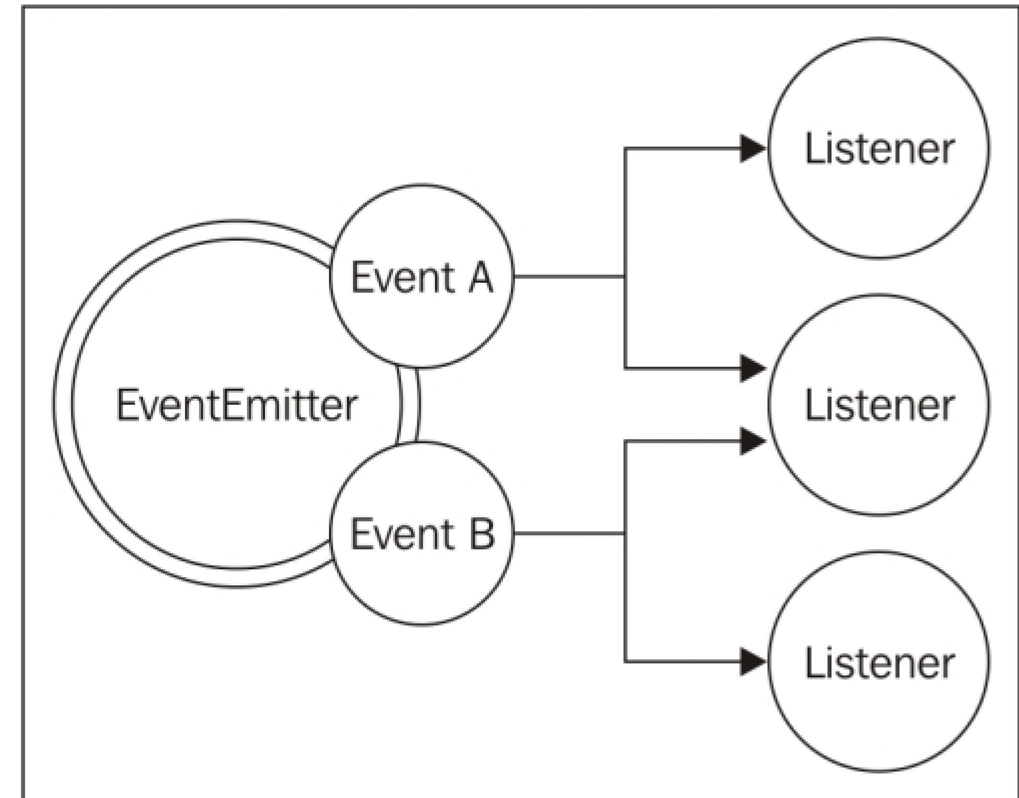
// Kreiramo objekat
const mojObjekat = new MojObjekat();

// Pretplatimo se na događaj
mojObjekat.on("nekiDogadjaj", () => {
  console.log("Događaj 'nekiDogadjaj' je emitovan!");
});

// Emitujemo događaj
mojObjekat.emit("nekiDogadjaj");
```

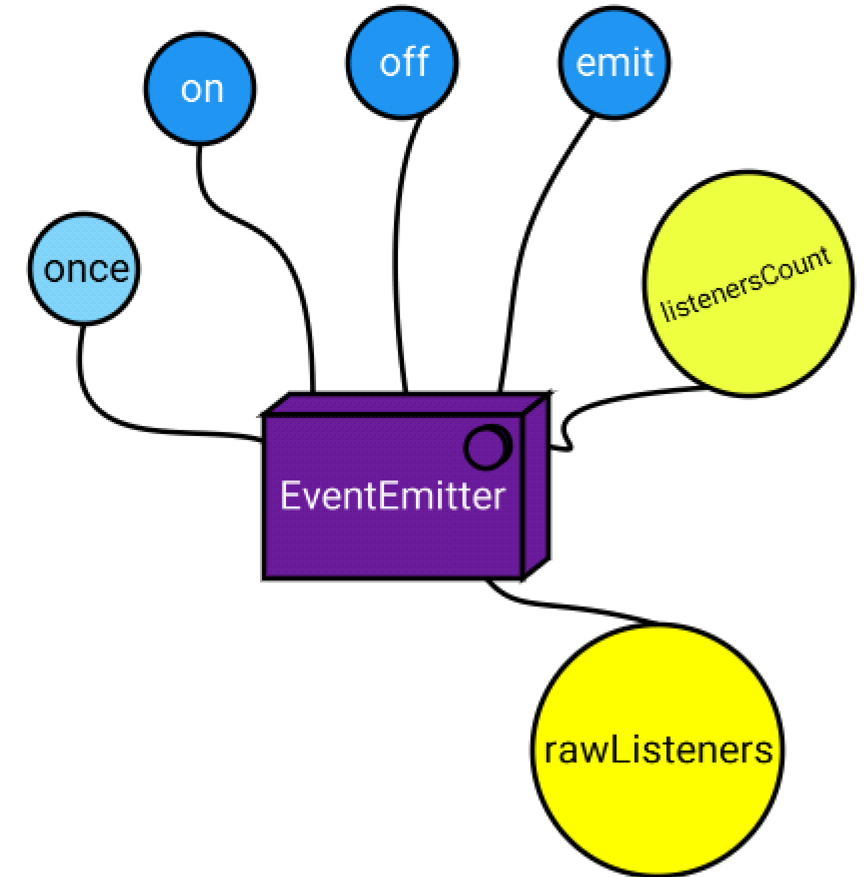
Dodavanje osluškivača događaja u objekte

- Nakon instanciranja objekta koji može da emituje događaje, može mu se dodati osluškivač.
- Osluškivači se dodaju u objekat *EventEmitter* pomoću jedne od funkcija:
  - **.addListener(eventName, callback)**
    Dodaje funkciju *callback* u osluškivače objekta;
    kada se aktivira događaj eventName,
    funkcija *callback* se smešta u red događaja
    da bi bila izvršena.
  - **.on(eventName, callback)**
    Funkcija koja je ista kao i .addListener( ).
  - **.once(eventName, callback)**
    Samo kada se funkcija *eventName* pozove prvi put,
    tada se *callback* funkcija smešta u red događaja,
    da bi bila izvršena.



Uklanjanje osluškivača iz objekata

- Pomoćne funkcije na objektu *EventEmitter* pomoću kojih možemo upravljati osluškivačima koje smo već dodali:
  - **.listeners(eventName)**
    Vraća niz funkcija osluškivača koje su povezane sa događajem *eventName*.
  - **.setMaxListeners(n)**
    Aktivira upozorenje ako je više od n osluškivača dodato u objekat *EventEmitter*. Podrazumevana vrednost je 10.
  - **.removeListener(eventName, callback)**
    ili
    .off(eventName, callback)
    Uklanja funkciju *callback* iz događaja *eventName* objekta *EventEmitter*.



Implementiranje osluškivača događaja i emitera događaja - primer uplata i isplate

```
const EventEmitter = require('events');
class Racun extends EventEmitter {
    constructor() {
        super(); // Poziva EventEmitter konstruktor
        this.balans = 0;
    }
    depozit(iznos) {
        this.balans += iznos;
        this.emit('balansPromenjen');
    }
    podigni(iznos) {
        this.balans -= iznos;
        this.emit('balansPromenjen');
    }
}
function prikaziStanje() {
    console.log("Racun balans: €%d", this.balans);
}
function proveriMinus() {
    if (this.balans < 0) {
        console.log("Racun u minusu!!!");
    }
}
```

```
function proveriCilj(rac, cilj) {
    if (rac.balans > cilj) {
        console.log("Dostignut cilj!!!");
    }
}

const bankracun = new Racun();
// Registrujemo event listenere
bankracun.on("balansPromenjen", prikaziStanje);
bankracun.on("balansPromenjen", proveriMinus);
bankracun.on("balansPromenjen", function() {
    proveriCilj(this, 1000);
});

// Testiramo funkcionalnost
bankracun.depozit(220);   // +220
bankracun.depozit(320);
// +320 → balans = 540
bankracun.depozit(600);
// +600 → balans = 1140 → Dostignut cilj!!!
bankracun.podigni(1200);
// -1200 → balans = -60 → Racun u minusu!!!
```

Implementiranje osluškivača događaja - isti primer sa arrow funkcijama

```
const EventEmitter = require('events');

class Racun extends EventEmitter {
    balans = 0; // Možemo koristiti public field

    depozit = (iznos) => {
        this.balans += iznos;
        this.emit('balansPromenjen');
    }

    podigni = (iznos) => {
        this.balans -= iznos;
        this.emit('balansPromenjen');
    }
}
// Arrow funkcije za event listenere
const prikaziStanje = () => {
    console.log(`Racun balans: €
                ${bankracun.balans}`);
};
const proveriMinus = () => {
    if (bankracun.balans < 0) {
        console.log("Racun u minusu!!!");
    }
};
```

```
const proveriCilj = (cilj) => {
    if (bankracun.balans > cilj) {
        console.log("Dostignut cilj!!!");
    }
};

const bankracun = new Racun();
// Registrujemo event listenere
bankracun.on("balansPromenjen", prikaziStanje);
bankracun.on("balansPromenjen", proveriMinus);
bankracun.on("balansPromenjen", () =>
proveriCilj(1000));

// Testiramo funkcionalnost
bankracun.depozit(220);   // +220
bankracun.depozit(320);
// +320 → balans = 540
bankracun.depozit(600);
// +600 → balans = 1140 → Dostignut cilj!!!
bankracun.podigni(1200);
// -1200 → balans = -60 → Racun u minusu!!!
```

Implementiranje osluškivača događaja - isti primer sa inline event listenerima

```
const EventEmitter = require('events');
class Racun extends EventEmitter {
    #balans = 0;
    #cilj;

    constructor(cilj = 1000) {
        super();
        this.#cilj = cilj;

        // Inline event listenere koristimo direktno sa arrow funkcijama
        this.on('balansPromenjen', () => console.log(`Racun balans: € ${this.#balans}`));
        this.on('balansPromenjen', () => this.#balans < 0 && console.log("Racun u minusu!!!"));
        this.on('balansPromenjen', () => this.#balans > this.#cilj && console.log("Dostignut cilj!!!"));
    }

    depozit = (iznos) => {
        this.#balans += iznos;
        this.emit('balansPromenjen');
    }
    podigni = (iznos) => {
        this.#balans -= iznos;
        this.emit('balansPromenjen');
    }
}
```

Privatna polja #balans i #cilj
nisu dostupna spolja, čuva se enkapsulacija.

Inline event listenere – sve tri funkcionalnosti
(prikaz, minus, cilj) direktno u konstruktoru.

```
// Testiranje
const bankracun = new Racun();

bankracun.depozit(220);
bankracun.depozit(320);
bankracun.depozit(600);
bankracun.podigni(1200);
```

Implementiranje povratnih poziva - stariji pristupi

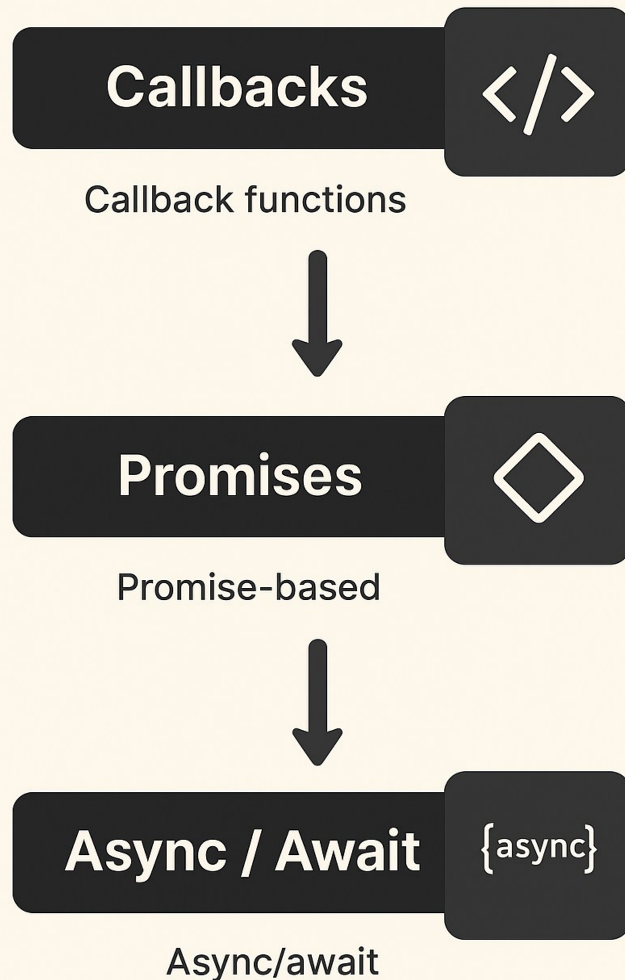
- Tri tradicionalne implementacije povratnih poziva (*callback*)

Varijanta #1: **Prosleđivanje parametara funkciji povratnog poziva**	**Varijanta #2:** **Obrada parametara funkcije povratnog poziva unutar petlje**	**Varijanta #3:** **Ugnežđavanje povratnih poziva**

`fs.readFile('file.txt', (err, data) => {`
`  if (err) throw err;`
`  console.log(data.toString());`
`});`
	`const files = ['a.txt', 'b.txt'];`
	`files.forEach(file => {`
	`  fs.readFile(file, (err, data) => {`
	`    if (err) throw err;`
	`    console.log(data.toString());`
	`  });`
	`});`
		`fs.readFile('a.txt', (err, data) => {`
		`  fs.readFile('b.txt', (err, data2) => {`
		`    fs.readFile('c.txt', (err, data3) => {`
		`      console.log(data + data2 + data3);`
		`    });`
		`  });`
		`});`

Funkcioniše, ali nije najmodernije.	Još uvek validno, ali opet može postati teško za praćenje kod kompleksnih asinhronih tokova.	Zastareo pristup i ne preporučuje se više njegovo korišćenje.
Problem: Lako dolazi do „callback hell-a“ (piramida propasti) kada se lanci callback-a ugnezde ili treba više asinhronih operacija u nizu.	Problem: Ako treba da sinhronizuješ rezultate više fajlova, dolazi do velikih komplikacija.	Problem: Teško za čitanje, održavanje i debagovanje.

Implementiranje povratnih poziva - noviji pristupi



EVENT-DRIVEN	PROMISES	ASYNC/AWAIT
EventEmitter.on()	new Promise()	async function
EventEmitter.emit	resolve?	await
	Yes	
	then()	try/catch

Pristup vođen događajima

NodeJS je prirodno vođen događajima. Ovo znači da objekti (obično *EventEmitter*) emituju događaje (*emit*) i drugi kod može da se "pretplati" (*on*) da reaguje na te događaje.

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

// Pretplata na događaj
myEmitter.on('greeting', (name) => {
    console.log(`Hello, ${name}!`);
});

// Emitovanje događaja
myEmitter.emit('greeting', 'Dražen');
```

Prednosti:

- Efikasan za višestruke, nezavisne događaje.
- Dobro za aplikacije sa mnogo I/O operacija.
- Nativno podržan dobro u NodeJS-u, jer je dobar za asinhrone tokove gde se događaji mogu emitovati i slušati, npr. WebSocket, UI događaji, ili bankovni računi.

Nedostaci:

- Može dovesti do „callback hell“ (piramide propasti) kada se događaji ugnežđuju.
- Teže za debagovanje i praćenje toka programa.
- Nema ugrađene mehanizme za redosled izvršavanja ili hvatanje grešaka.

Pristup sa objektima *Promises*

Promises su objekti koji predstavljaju rezultat asinhrone operacije koja može biti uspešna (*resolve*) ili neuspešna (*reject*). One pomažu da se izbegne „callback hell" i olakšava upravljanje asinhronim kodom.

```
function asyncOperation(success) {
return new Promise((resolve, reject) => {
      setTimeout(() => {
         if (success) resolve('Uspelo!');
         else reject('Neuspeh!');
      }, 1000);
 });
}
asyncOperation(true)
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Prednosti:

- Bolja čitljivost u poređenju sa ugnježdenim *callback*-ovima.
- Jasna sintaksa za uspeh ili hvatanje greške (then/catch).
- Lako se kombinuju sa Promise.all, Promise.race.

Nedostaci:

- Sintaksa sa .then().catch() može i dalje postati dugačka za kompleksne tokove.

Async / await pristup

Async/await je najmoderniji pristup za callback, koji omogućava da asinhroni kod izgleda kao sinhroni, što znatno poboljšava čitljivost.

```
function asyncOperation(success) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (success) resolve('Uspelo!');
            else reject('Neuspeh!');
        }, 1000);
    });
}
async function run() {
    try {
        const result = await asyncOperation(true);
        console.log(result);
    } catch (error) {
        console.error(error);
    }
}
run();
```

Prednosti:

- Kod je linearan, izgleda kao sinhroni, i lakši za čitanje, ali je i dalje asinhroni.
- Jednostavno upravljanje greškama (try/catch).
- Lako kombinovanje više asinhronih operacija u sekvenci.

Nedostaci:

- Morate biti unutar async funkcije.
- Ako se ne koristi pažljivo sa paralelnim operacijama, može smanjiti performanse (sekvencijalno izvršavanje).

Node.js

Upravljanje ulazima i izlazima podataka kod Node.js

Upotreba JSON

- JSON = JavaScript Object Notation
- JSON predstavlja tekstualni format za razmenu podataka koji je lako čitljiv ljudima, i u JavaScript-u se koriste **JSON.stringify() za konverziju objekta u string** i **JSON.parse() za obrnuti proces, konverzije stringa u objekat**.
- JSON se često koristi za:
  - slanje podataka sa klijenta na server i obrnuto (npr. u AJAX pozivima),
  - komunikaciju između različitih procesa ili servisa,
  - skladištenje podataka u bazama (posebno NoSQL baze kao što je MongoDB).
- Razlike u odnosu na XML standard:
  - JSON je efikasniji (lakši za parsiranje) i kraći (sadrži manje znakova).
  - Serijalizacija/deserijalizacija pomoću JSON-a je mnogo brža nego pomoću XML-a, zbog jednostavnije strukture.
  - Sintaksa je preglednija i slična JavaScript objektima, pa ga programeri lakše razumeju.
- JSON **ne podržava funkcije, datume ili undefined vrednosti** u objektima, pa se one moraju drugačije tretirati prilikom serijalizacije.

Pretvaranje JSON u JS objekte

- JSON je tekstualni format za razmenu podataka, zasnovan na JavaScript objektnoj sintaksi. Da bi se korektan JSON pretvorio u JS objekat koristi se metoda **JSON.parse(string)**.
- Primer:

```
var companyString = ' {"name": "Samsung",
               "models": ["S9", "A7", "C6"],
               "location": "South Korea"}';
var companyObj = JSON.parse(companyString);
console.log(companyObj.name);    // Samsung
console.log(companyObj.models);  // ["S9", "A7", "C6"]
```

Pretvaranje JS objekata u JSON string

- Koristi se sledeća metoda: **JSON.stringify(objekat)**,
  gde objekat može biti običan JS objekat, niz, broj, boolean itd.
- JSON string se nakon toga može uskladištiti u datotetku, ili u bazu podataka, poslati pomoću HTTP veze, ili upisati u tok/bafer,...
- Primer:

```
var facultyObj = {
    name: "ETF",
    labs: ["P25", "P26", "SI60", "SI70"],
    location: "Bulevar kralja Aleksandra 73"
};
var facultyString = JSON.stringify(facultyObj);
console.log(facultyString);
// Output: '{"name":"ETF","labs":["P25","P26","SI60","SI70"],
//          "location":"Bulevar kralja Aleksandra 73"}'
```

Upotreba modula Buffer za baferisanje podataka

- *Buffer* predstavlja niz okteta za rad sa binarnim podacima u Node.js.
- Omogućava kreiranje, čitanje, pisanje i manipulaciju binarnim podacima.
- Buffer je globalan, pa se ne mora učitavati sa require().
- Podaci se skladište u memoriji izvan V8 heap-a („sirova“ memorija).
- Metodi za kodiranje:

Metod	Opis
utf8	Kodirani Unicode znakovi sa više bajtova
utf16le	Kodirani znakovi malog endiana, od 2 do 4 bajta
ucs2	Isto kao utf16le
base64	Kodira string u formatu base64
Hex	Kodira svaki bajt kao 2 heksadecimalna znaka

Metode za kreiranje objekata *Buffer*

- Postoje 3 metode za kreiranje objekata Buffer:

- **Buffer.alloc(sizeInBytes)** – kreira buffer zadate veličine, inicijalizovan nulama.
  var buf256 = Buffer.alloc(256);
- **Buffer.from(arrayOfOctets)** – kreira buffer od niza okteta.
  var bufOkteti = Buffer.from([0x6f, 0x63, 0x74, 0x65, 0x74, 0x73]);
- **Buffer.from(string, [encoding])** – kreira buffer od stringa sa opcionim kodiranjem (default 'utf8’).
  var bufUtf = Buffer.from("Neki utf8 tekst", 'utf8');

Metode za upisivanje podataka u objekte Buffer

Metod	Opis	Primer

buffer.write(string, [offset], [length], [encoding])	Upisuje broj bajtova length, iz stringa datog kao prvi argument, počev od indeksa offset (Ako se offset ili length ne navedu, podrazumevaju se 0 i ostatak stringa; encoding je opcioni, default 'utf8').	var buf = Buffer.alloc(10);
		buf.write("Hello", 0, 5, "utf8");
		// Upisuje "Hello" u buffer

buffer[offset] = value	Zamenjuje podatke na indeksu offset navedenom vrednošću (value mora biti broj između 0 i 255 (jedan bajt)).	var buf = Buffer.alloc(4); buf[0] = 0x41; // ASCII 'A'
buffer.fill(value, [offset], [end])	Upisuje vrednost svakogbajta u bafer, od indeksa offset, do indeksa end (value može biti broj, string ili drugi buffer; Ako se offset ili end ne navedu, podrazumeva se cela dužina buffera).	var buf = Buffer.alloc(5); buf.fill(0x20); // Popunjava ceo buffer sa space (0x20)

writeInt8 (value, offset, [noAssert])
writeInt16LE (value, offset, [noAssert])
writeInt16BE (value, offset, [noAssert])
	Različite metode za upisivanje 8-bitnih celih brojeva, ili neoznačenih celih brojeva, upisivanje, 16-bitni označenih brojeva u little/big endian formatu, itd.	var buf = Buffer.alloc(4);
		buf.writeInt16LE(512, 0);
		// Upisuje 512 kao 16-bit LE broj

Metode za čitanje iz objekta Buffer

Metod	Opis
buffer.toString([encoding], [start], [end])	Vraća string sa dekodiranim znakovima koji su određeni pomoću kodnog rasporeda, od indeksa start, do indeksa end bafera. Ako start i end nisu definisani, koristi se od početka do kraja bafera.
stringDecoder.write(buffer)	Vraća dekodiranu verziju stringa bafera.
buffer[offset]	Vraća vrednost okteta u baferu na kraju određenog indeksa offset.
readInt8(offset, [noAssert]) readInt16LE(offset, [noAssert]) readInt16BE(offset, [noAssert])	Veći broj metoda objekta Buffer za pisanje celih brojeva.

Druge važne metode i svojstva Buffer-a

- Dužina bafera:
  Buffer.byteLength(string, [encoding]);
  Buffer("tekst").length;
- Kopiranje:
  copy(targetBuffer, [targetStart], [sourceStart], [sourceIndex])
- Deljenje bafera:
  Buffer("123456789").slice([start], [end])
- Spajanje bafera:
  Buffer.concat(list, [totalLength])

Tokovi (*streams*)

- **Tokovi** (*streams*) predstavljaju osnovu za rad sa podacima koji dolaze postepeno (chunk po chunk), umesto da se sve učitava u memoriju odjednom.
- Popularni način strukturiranja servera:
  - Mrežni interfejs <-> TCP/IP protocol procesiranje <-> HTTP protkol procesiranje <-> programski kod
  - Dodavanje modula u tok: Mrežni interfejs <-> TCP/IP protocol procesiranje <-> *Enkripcija* <-> HTTP protkol procesiranje
- U praksi su ključni za:
  - performanse (obrada velikih fajlova ili mrežnih odgovora bez potpunog učitavanja u RAM),
  - obradu u realnom vremenu (streaming podaci, audio/video, API response),
  - protočne obrade / *pipelines* (povezivanje više tokova radi transformacije podataka).
- Node API-jevi intenzivno koriste tokove:
  - Tokovi za čitanje (primer: fs.createReadStream)
  - Tokovi za upisivanje (primer: fs.createWriteStream)
  - Duplex tok (primer: net.createConnection)
  - Transform tok (primeri: zlib, crypto)

Primeri tokova *Readable*

- Uobičajeni primeri tokova:
  - **HTTP odgovori na klijentu**, npr. npr. http.get(...) vraća *Readable stream* (*response*).
  - **HTTP zahtevi na server**, odnosno objekat *req* u povratnom pozivu servera (*http.createServer*) je *Readable*.
  - **Tokovi za čitanje fs**: *fs* → *fs.createReadStream*(*path*) je tipičan primer.
  - **Tokovi zlib**: npr. *zlib.createGzip()* ili *zlib.createGunzip()* vraćaju transform tokove (*Duplex*).
  - **Tokovi crypto**: *crypto.createCipheriv*() ili *crypto.createDecipheriv*() vraćaju *Duplex* tokove.
  - **TCP socketi**: *net.Socket* objekti su *Duplex* (mogu i čitati i pisati).
  - Podređeni procesi **stdout** i **stderr**: *child.stdout* i *child.stderr* su *Readable* tokovi.
  - **process.stdin**: globalni *Readable* tok za unos sa tastature/terminala.

Tokovi *Readable* za čitanje podataka

- Tokovi *Readable* obezbeđuju metod *read*(*[size]*) za čitanje podataka, gde *size* određuje broj bajtova za čitanje iz toka.
- Ovaj metod vraća *Buffer*, *string* (ako je postavljen encoding preko *setEncoding()*) ili *null* (ako trenutno nema podataka).
- Tokovi *Readable* izlažu sledeće događaje:
  - *readable* - emituje se kada se delovi podataka mogu čitati iz toka metodom *read()*;
  - *data* - kada se registruje listener za *data*, tok prelazi u *flowing mode* i emituje *data* događaje sve dok ne iscrpi sve podatke ili se pozove *pause()*;
  - *end* - emituje ga tok kada više nema podataka;
  - *close* - emituje se kada se zatvori osnovni izvor, kao što je datoteka;
  - *error* - emituje se kada dođe do greške u toku čitanja.

Funkcije kod objekta *Readable*

Metod	Opis
read([size])	Čita podatke iz toka. Može da čita podatke String, Buffer i null.
setEncoding(encoding)	Postavlja kodiranje koje će biti upotrebljeno kada se koristi read( ) i vrati String.
pause( )	Zaustavlja emitovanje događaja podataka pomoću objekta.
resume( )	Ponovo pokreće emitovanje događaja pomoću objekta.
pipe(destination, [options])	Usmerava izlaz ovog toka u tok objekta Writable, koji je određen pomoću iskaza destination.option u JS objektu.
unpipe([destination])	Isključuje objekat iz odredišta Writable.

Primeri tokova *Writable*

- Uobičajeni primeri tokova:
  - **HTTP zahtevi na klijentu**, npr. objekat req dobijen iz *http.request()* ili *http.get()*.
  - **HTTP odgovori na serveru**, objekat res iz povratnog poziva *http.createServer*(...).
  - **Tokovi za upisivanje podataka fs**: *fs.createWriteStream*(*path*).
  - **Tokovi zlib**: *zlib.createGzip()* i *zlib.createGunzip()* su *Duplex* (mogu biti i *Readable* i *Writable*).
  - **Tokovi crypto**: recimo *crypto.createCipheriv()* i *crypto.createDecipheriv()*, isto su *Duplex*.
  - **TCP socketi**: *net.Socket* objekti, koji su takođe Duplex (Writable + Readable).
  - podređeni procesi **stdin**: *child.stdin* je *Writable*.
  - **process.stdout**, **process.stderr**: globalni *Writable* tokovi.
- Kao i kod *Readable* tokova, većina ovih primera su zapravo *Duplex* (npr. socketi, zlib, crypto), ali se potpuno legitimno koriste i kao *Writable*.
- API za rad je isti: metode *write*(*chunk*), *end()*, i događaji '*drain*', '*finish*', '*error*'.
- Moderno je koristiti i *pipeline* (npr. *await pipeline*(readable, zlib.createGzip(), writable)) kako bi se pojednostavilo rukovanje greškama i završavanjem toka.

Tokovi *Writable* za upisivanje podataka

- Tokovi *Writable* obezbeđuju metod write(chunk, [encoding], [callback]) za upisivanje podataka, u kome chunk sadrži podatke koji će se upisivati, encoding obezbeđuje kodiranje stringa, a callback funkciju povratnog poziva, koja se izvršava kada se podaci "potroše". Funkcija vraća true kada su podaci uspešno upisani.
- Tokovi *Writable* izlažu i sledeće događaje:
  - *drain* - nakon što poziv wirte( ) vrati vrednost false, emituje se događaj drain da bi osluškivači bili obavešteni kada mogu da upišu još podataka;
  - *finish* - emituje se kada je metod pozvan na objekat Writable. Tokom emitovanja ovog događaja svi baferi su ispražnjeni i podaci više neće biti prihvaćeni.
  - *pipe* - emituje se kada je pozvan metod pipe( ) na tokove Readable, radi dodavanja toka Writable kao odredišta.
  - *unpipe* - emituje se kada je metod unpipe( ) pozvan na tok Readable, radi uklanjanja toka Writable kao odredišta.

Funkcije kod objekta *Writable*

Metod	**Opis**
write (chuck, [encoding], [callback])	Upisuje delove podataka na lokaciji podataka objekta toka. Podaci mogu biti *String* ili *Buffer*. Ako je određen *callback*, on se poziva nakon što su svi podaci iskorišćeni.
end([chunk], [encoding], [callback])	Isti kao metod *write*( ), osim što postavlja *Writable* u stanje u kome više ne prihvata podatke i šalje događaj *event*.

Tok *Duplex*

- **Duplex tok** kombinuje *Readable* i *Writable* tokove u jednom objektu.
- Primer Duplex toka je TCP socket (net.Socket), ali i neki tokovi iz zlib i crypto modula.
- **Implementacija metoda**
  - Ako kreirate sopstveni Duplex tok koristeći stream.Duplex, treba da implementirate:
    - _read(size) => kako tok Readable dobija podatke
    - _write(chunk, encoding, callback) => kako tok Writable upisuje podatke
  - Takođe, često se implementira i _final(callback) za završetak Writable dela, i _destroy(err, callback) za čišćenje resursa.
- Napomena o redosledu
  - Nije obavezno implementirati _read i _write ako koristite već gotove Duplex tokove (npr. socket, zlib), oni već imaju implementaciju. Obavezno je samo kada pravite sopstveni prilagođeni Duplex tok.
- Duplex tok emituje događaje i za Readable ('data', 'end', 'readable') i za Writable ('drain', 'finish', 'error').

Tokovi *Transform*

- Tok *Transform* proširuje tok *Duplex*, i modifikuje podatke između tokova *Writable* i *Readable*, što može biti korisno kada se modifikuju podaci iz jednog sistema u drugi (dok podaci prolaze kroz tok možete ih procesirati/enkodirati/kompresovati/šifrovati).
- Primeri tokova *Transform* su:
  - tokovi *zlib*: *zlib.createGzip(), zlib.createGunzip()*
  - tokovi *crypto*: *crypto.createCipheriv(), crypto.createDecipheriv()*
- Implementacija metoda
  - Za sopstveni Transform tok, obično se implementira metoda _transform(chunk, encoding, callback).
    - _transform kombinuje u sebi funkcionalnost _write i _read i automatski se brine o protoku podataka.
  - Ne implementirati direktno _read() i _write() (za razliku od običnog Duplex toka), ali _transform() je obavezan.
  - Opciono se može implementirati _flush(callback) ako želite da obradite preostale podatke kada tok završi.
- Transform tok i dalje nasleđuje Duplex, pa _read() i _write() postoje, ali ih <u>ne morate redefinisati</u>.

Spajanje tokova *Readable* sa tokovima *Writable*

- Objekte toka možemo iskoristiti spajanjem tokova Readable sa Writable, pomoću funkcije `pipe (writableStream, [options])`
- Izlaz iz toka *Readable* je direktan ulaz u tok *Writable*
- Opcioni parametar *options* prihvata objekat sa svojstvom *end*, koje je postavljeno na *true*/*false*. Ako *end* ima vrednost *true*, tok *Writable* se završava kada i tok *Readable* (ukoliko je false, Writable tok ostaje otvoren nakon što Readable završi).
- Primer: `readStream.pipe(writeStream, { end: true });`
- Pomoću `unpipe(destinationStream)` tokovi mogu da se odspoje i tako prestaje automatsko slanje podataka (za tok koji smo prethodno povezali).

Node.js

Pristup sistemu datoteka iz okruženja Node.js

Sinhroni i asinhroni pozivi

- Preduslov rada sa datotekama je učitan modul fs:
  `const fs = require('fs');` ili `const fs = require('fs').promises;`
- Modul fs omogućava da skoro sve funkcije budu dostupne u asinhronom i u sinhronom obliku.
- Na primer: *writeFile( )* kao funkcija u asinhronom obliku,
  ima svoj sinhroni oblik *writeFileSync( ).*
- Sinhroni pozivi: blokiraju petlju događaja. Celo Node.js okruženje čeka, dok se poziv ne završi (zatim se kontrola vraća na nit).
- Asinhroni pozivi: stavljaju operaciju u red događaja i odmah vraćaju kontrolu nazad,
  a rezultat se obrađuje kasnije, kroz povratni poziv ili Promise.
- **Asinhroni pozivi su standard i preporučeni**.
  Oni omogućavaju da Node.js ostane neblokirajući i da može da obrađuje više zahteva istovremeno.
- Sinhroni pozivi se retko koriste u produkciji, uglavnom i skriptama i prilikom pokretanja servera,
  ili u manjim alatima, gde blokiranje nije problem.

Error-first callback uzorak

- Tradicionalni Node.js **asinhroni pozivi** očekuju funkciju povratnog poziva (*callback*) kao poslednji parametar:

```
fs.readFile('file.txt', 'utf8', function(err, data) {
    if (err) {
        console.error('Došlo je do greške:', err);
    } else {
        console.log('Sadržaj fajla:', data);
    }
});
```

- Prvi argument u povratnom pozivu je greška (err), a drugi argument daje rezultat (npr. pročitan sadržaj fajla).

Primer sa asinhronim fs pozivom i obradom greške

```
const fs = require('fs');

fs.readFile('nonexistent.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Došlo je do greške pri čitanju fajla:', err.message);
    return;
  }
  console.log('Sadržaj fajla:', data);
});
```

- Ako fajl ne postoji => err je objekat greške (Error) i sadržaj fajla (data) je undefined.
- Ako fajl postoji => err je null i data sadrži sadržaj fajla.

Otvaranje i zatvaranje datoteka

- Otvaranje datoteke:
  Asinhrono: `fs.open (path, flags, [mode], callback)`
  Sinhrono: `fs.openSync (path, flags, [mode])`
- Parametri:
  `path` - određuje string putanje do datoteke (apsolutne ili relativne).
  `flags` - određuje režim otvaranja datoteke (r – čitanje, w – pisanje, a – dodavanje/*append*)
  `mode` – opciono, postavlja permisije datoteke, relevantno pri kreiranju datoteke.
  `callback` - funkcija koja prima parametre err (objekat greške) i fd (*file descriptor*)
- Zatvaranje datoteke:
  Asinhrono: `fs.close (fd, callback)`
  Sinhrono: `fs.closeSync (fd)`

fd - deskriptor datoteke (integer), koji predstavlja otvorenu datoteku i koji može da se koristi za čitanje datoteke ili upisivanje u datoteku.

Primer asinhronog otvaranja, pisanja i zatvaranja

```
const fs = require('fs');

fs.open('example.txt', 'w', (err, fd) => {
  if (err) throw err;

  fs.write(fd, 'Hello Node.js!', (err) => {
    if (err) throw err;

    fs.close(fd, (err) => {
      if (err) throw err;
      console.log('Fajl je zatvoren.');
    });
  });
});
```

Primer asinhronog otvaranja, pisanja i zatvaranja

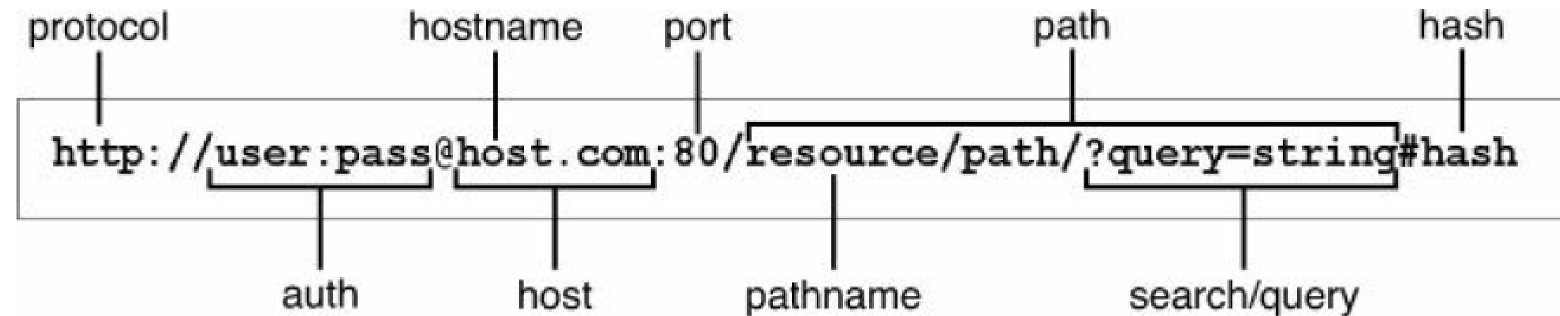
```
const fs = require('fs').promises;
async function writeFileExample() {
  let fd; //deskriptor fajla
  try {
    // Otvaranje fajla za pisanje ('w' - kreira ili prepisuje fajl)
    fd = await fs.open('example.txt', 'w');
    // Pisanje sadržaja u fajl
    await fd.write('Hello Node.js!');
    console.log('Podaci su uspešno upisani u fajl.');
  } catch (err) {
    console.error('Došlo je do greške:', err.message);
  } finally {
    // Zatvaranje fajla (ako je otvoren)
    if (fd) {
      await fd.close();
      console.log('Fajl je zatvoren.');
    }
  }
}
```

U modernom kodu preporučuje se fs.promises modul i async/await.

Node.js

Implementiranje HTTP servisa

Obrada URL adresa



- **Express.js**
  - Radni okvir koji olakšava pravljenje punog veb servisa (rute, *middleware*, obrada zahteva i odgovora).
  - Koristi http ili https module „ispod haube“ za pravljenje servera.
  - Robusno rutiranje, fokus na visoke performanse, izvršni fajl za brzo generisanje aplikacija.
- **http / https**
  - Node.js moduli niskog nivoa koji omogućavaju pravljenje servera i klijenata.
  - Ne pružaju rute, middleware ili lako rukovanje zahtevima i odgovorima.
- Primer:

```
const myUrl = new URL('https://rti.etf.bg.ac.rs/path?name=test');
console.log(myUrl.hostname); // rti.etf.bg.ac.rs
console.log(myUrl.pathname); // /path
console.log(myUrl.searchParams.get('name')); // test
```

Svojstva objekta URL

Svojstvo	Opis
href	Cela URL adresa kao string, uključujući *protokol*, *host*, *path* i *query*.
origin	Skraćeni deo URL-a koji uključuje protokol i host (bez *path* i *query*).
protocol	Protokol URL-a (npr. http: ili https:), uključuje dvotačku.
username/password	Informacije o autentifikaciji (username, password)
host	Deo *host* pune URL adrese, uključujući informacije o delu *port*.
hostname	Samo domen ili IP (bez porta) (npr. example.com).
port	Broj porta u URL-u (npr. 8080). Ako nije definisan, prazan string.
pathname	Deo URL-a koji označava putanju na serveru (npr. /path/to/file).
search	*Query* string uključujući ? (npr. ?name=test&age=20).
searchParams	URLSearchParams objekat za lako čitanje i menjanje query parametara.
hash	Fragment identifikator posle # (npr. #section1).

URL objekat u Node.js - primer

```
// Kreiranje URL objekta
const myUrl = new URL('https://username:password@example.com:8080/path/to/page?name=Nikola&age=30#bio');
// Pristup svojstvima
console.log('href:', myUrl.href);                       // Cela URL adresa
console.log('origin:', myUrl.origin);                   // https://example.com:8080
console.log('protocol:', myUrl.protocol);               // https:
console.log('username:', myUrl.username);               // username
console.log('password:', myUrl.password);               // password
console.log('host:', myUrl.host);                       // example.com:8080
console.log('hostname:', myUrl.hostname);               // example.com
console.log('port:', myUrl.port);                       // 8080
console.log('pathname:', myUrl.pathname);               // /path/to/page
console.log('search:', myUrl.search);                   // ?name=Nikola&age=30
console.log('searchParams:', myUrl.searchParams); // URLSearchParams { 'name' => 'Nikola', 'age' => '30' }
console.log('hash:', myUrl.hash);                       // #bio

// Rad sa searchParams
console.log('name param:', myUrl.searchParams.get('name')); // Nikola
myUrl.searchParams.set('name', 'Andjela');
console.log('Updated href:', myUrl.href);               // href sa izmenjenim parametrom
```

Objekat http.ClientRequest

- Objekat *ClientRequest* se kreira interno kada se pozove metod `http.request()` prilikom izrade HTTP klijenta. To je zahtev koji se izvršava na serveru.
- *ClientRequest* implementira tok Writable da bi bile omogućene sve funkcije objekta toka Writable (poziv write(chunk) za slanje tela zahteva, ili end() kada završimo sa slanjem).

Objekat http.ClientRequest - primer

```
const http = require('http');
const options = {
  hostname: 'www.myserver.com',
  port: 8080,
  path: '/',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': Buffer.byteLength
     (JSON.stringify({ key: 'value' }))
  }
};

const req = http.request(options, (res) => {
  let data = '';
  res.on('data', (chunk) => {
    data += chunk;
  });
  res.on('end', () => {
    console.log('Response:', data);
  });
});

req.on('error', (e) => {
  console.error('Problem with request:', e.message);
});

// Write data to request body
const postData = JSON.stringify({ key: 'value' });
req.write(postData);
req.end();
```

Objekat http.ClientRequest - alternativa sa *fetch API*

```
import fetch from 'node-fetch';

const url = 'https://www.myserver.com/';
const postData = { key: 'value' };

fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(postData)
})
  .then((res) => res.text())
  .then((data) => console.log('Response:', data))
  .catch((error) => console.error('Error:', error));
```

Svojstva koja se zadaju objektu ClientRequest

Svojstvo	Opcije
host	Naziv domena kao string (podrazumevana vrednost: *localhost*).
hostname	Isto kao host, bez porta.
port	Broj porta udaljenog servera (podrazumevano: 80).
localAddress	Opcionalna lokalna adresa sa koje će zahtev biti poslat (npr. IP).
socketPath	Putanja do Unix soketa ako se koristi umesto host/port.
method	String koji određuje metod HTTP zahteva (GET/POST/PUT)
path	String koji određuje potrebnu izvornu pitanju, podrazumevano / Primer: /knjige.html?id=396
headers	Objekat sa HTTP zaglavljima. Primer: { 'content-length':'750', 'content-type': 'application/json' }
auth	Autentifikacija u formatu username:password (koristi se za Basic auth).
agent	http.Agent objekat koji kontroliše pooling konekcija; ako je false, koristi se novi socket za svaki zahtev.

Metodi koji su dostupni na objektima ClientRequest

Metod	Opis
write (chunk, [encoding])	Upisuje objekat tela podataka chunk, Buffer ili String u zahtev. Ovo omogućava da učitate podatke u tok Writable objekta ClientRequest.
end ([data], [encoding])	Upisuje opcione podatke u telo zahteva, a zatim će isprazniti tok Writable i završiti zahtev.
abort ( )	Prekida trenutni zahtev.
setTimeout(timeout, [callback])	Podešava vreme prekida zahteva.
setNoDelay ([noDelay])	Onemogućava Nagleov algoritam, koji baferiše podatke pre slanja.
setSocketKeepAlive ([enable], [initialDelay])	Omogućava i onemogućava funkciju keep-alive na zahtev klijenta. Parametar enable je podrazumevano postavljen na vrednost false, pa se onemogućava. Parametar initialDelay određuje kašnjenje između poslednjeg paketa podataka i prvog zahteva keep-alive.

Objekat http.ServerResponse

- Objekat *ServerResponse* se kreira pomoću HTTP servera interno kada je primljen događaj *response*. On se prosleđuje hendleru događaja *request* kao drugi argument, a koristi se za formulisanje odgovora i njegovo slanje klijentu.
- Objekat *ServerResponse* implementira tok *Writable* da bi bile omogućene sve funkcije objekta toka *Writable*.

Svojstvo	Opis
close	Emituje se kada je klijentska veza zatvorena, pre slanja metoda response.end(), radi završetka i iskorišćavanja odgovora.
headersSent	Vrednost true ako je poslato zaglavlje, i false ako nije.
sendDate	Vrednost true kada se zaglavlje Date automatski generiše i šalje.
statusCode	Omogućava statusni kod odgovora bez eksplicitnog pisanja zaglavlja (npr. response.statusCode = 500)

Metodi dostupni na objektima *ServerResponse*

Metod	Opis
writeContinue( )	Šalje poruku HTTP/1.1 100 Continue klijentu sa zahtevom da treba poslati telo poruke.
writeHead (statusCode, [reasonPhrase], [headers])	Upisuje zaglavlje odgovora u zahtev. Parametar statusCode može imati vrednosti 200, 401 ili 500 (trocifreni kod HTTP statusa). Parametar reasonPhrase je razlog pojave statusa, a headers objekat zaglavlja.
setTimeout (msecs, callback)	Postavlja vreme prekida socketa za klijentsku vezu u milisekundama, zajedno sa funkcijom callback koja se izvršava kada se desi prekid.
setHeader (name, value)	Postavlja vrednost određenog zaglavlja u kome je name naziv HTTP zaglavlja, a value je vrednost zaglavlja.
getHeader (name)	Dohvata vrednost HTTP zaglavlja, koje je postavljeno u odgovoru.
removeHeader (name)	Uklanja HTTP zaglavlje koje je postavljeno u odgovoru.
write (chunk, [encoding])	Upisuje objekat podataka chunk, Buffer ili Stirng u odgovor toka Writable. Podaci se upisuju samo u delu tela odgovora.
addTrailers(headers)	Dodaje prateća HTTP zaglavlja na kraj odgovora.
end([data], [encoding])	Upisuje opcione podatke u telo odgovora, a zatim prazni tok.

Objekat *http.IncomingMessage*

- Objekat *IncomingMessage* se kreira pomoću HTTP servera ili HTTP klijenta.
  Na strani servera zahtev klijenta je predstavljen pomoću *IncomingMessage*,
  a na strani klijenta odgovor servera je predstavljen pomoću objekta *IncomingMessage*.
- Objekat *IncomingMessage* implementira tok *Readable*, čime se omogućava čitanje zahteva klijenta ili odgovora servera kao izvor toka.

http.IncomingMessage objekat - svojstva

Svojstvo	Opis
httpVersion	Verzija HTTP protokola (npr. '1.1').
complete	Boolean, true ako je telo odgovora potpuno primljeno.
headers	Objekt sa header-ima odgovora (npr. { 'content-type': 'application/json' }).
rawHeaders	Niz zaglavlja u originalnom redosledu ([key1, value1, key2, value2, ...]).
trailers	Zaglavlja koja dolaze nakon tela odgovora (chunked encoding).
rawTrailers	Niz trailer zaglavlja u originalnom obliku.
statusCode	HTTP status kod odgovora (npr. 200, 404).
statusMessage	Poruka statusa odgovora (npr. 'OK').
method	HTTP metoda zahteva (GET, POST, itd.), važi za IncomingMessage kada je server prima zahtev.
url	Putanja URL-a, važi za IncomingMessage na serveru (npr. /path?x=1).
socket	TCP socket (net.Socket) preko kojeg dolazi zahtev/odgovor.
connection	Alias za socket.

http.IncomingMessage objekat - metode i događaji

Metod	Opis
setEncoding(encoding)	Postavlja enkoding za čitanje podataka (utf8, ascii, itd.).
pause()	Pauzira čitanje podataka sa toka.
resume()	Nastavlja čitanje podataka sa toka.
destroy([error])	Uništava tok i zatvara konekciju.
on(event, listener)	Dodavanje event handlera; ključni događaji: 'data', 'end', 'error', 'close', 'aborted', 'readable'.
read([size])	Čita podatke iz toka (koristi se u režimu čitanja po delovima).

Događaj	Opis
'data'	Emituje se kada stigne deo podataka; koristi se u toku čitanja.
'end'	Emituje se kada su svi podaci primljeni.
'error'	Emituje se u slučaju greške u toku odgovora/zahteva.
'close'	Emituje se kada se tok zatvori.
'aborted'	Emituje se kada klijent ili server prekine konekciju pre kraja prenosa.
'readable'	Signalizira da podaci mogu biti čitani pomoću read() metode.

Objekat *http.Server*

- http.Server objekat nasleđuje *EventEmitter*, što znači da mogu da se koriste metode on(), once(), emit() i slično.
- Server emituje različite događaje koji omogućavaju rukovanje zahtevima, greškama i konekcijama.
- Nije striktno “obavezno” implementirati sve događaje, ali u praksi morate obraditi bar događaj 'request' (ili koristiti funkciju povratnog poziva prilikom kreiranja servera), jer je to osnovni način da HTTP server odgovara na zahteve klijenata.
  - 'request' – emituje se svaki put kada server primi HTTP zahtev.
  - Handler za ovaj događaj prima dva argumenta: request (*IncomingMessage*) i response (*ServerResponse*).

Događaji koje mogu da aktiviraju objekti Server

Događaj	Kratak opis
'request'	Aktivira se uvek kada server primi HTTP zahtev klijenta.
'connection'	Aktivira se kada je uspostavljena nova TCP konekcija.
'close'	Aktivira se kada je server zatvoren i više ne prihvata konekcije.
'checkContinue'	Aktivira se kada je primljen zahtev koji sadrži zaglavlje Expect:100-continue
'upgrade'	Emituje se kada klijent zahteva nadogradnju protokola (101 Switching Protocols, npr. WebSocket).
'clientError'	Emituje se kada socket klijentske veze prikazuje grešku.

Node.js

Express.js radni okvir

Express.js - veb radni okvir za Node.JS

- Brz, neupadljiv i minimalistički veb radni okvir (eng. *framework*)
- Relativno tanak sloj preko osnovnih funkcionalnosti Node.JS
- Šta je potrebno implementatoru veb servera?
  - Da tumači HTTP: prihvatanje TCP konekcija, obrada HTTP zahteva, slanje HTTP odgovora
  - Rutiranje: mapiranje URL adresa na funkciju veb servera za taj URL
    - Potrebna je podrška za tabelu rutiranja (kao što je React Router)
  - Podrška za međusloj (*middleware*): omogućavanje dodavanja slojeva za obradu zahteva
    - Olakšava dodavanje prilagođene podrške za sesije, kolačiće, bezbednost, kompresiju, itd.

Uključivanje Express.js

- `let expressApp = express();` //modul koristi projektni uzorak Fabrika

  //expressApp objekat ima metode za:
  - Rutiranje HTTP zahteva
  - Renderovanje HTML (pokreće neki pretprocesor, kao što je npr. Jade engine)
  - Konfiguriše posrednički sloj (*middleware*) i pretprocesore
- ```
  expressApp.get('/', function(httpRequest, httpResponse) {
      httpResponse.send('Zdravo svima’);
  });
  ```
- `expressApp.listen(3000); //podrazumevana adresa za localhost koristi port 3000`
  ```

Express.js rutiranje

- Korišćenjem HTTP metoda:
  - `expressApp.get(urlPath, requestProcessFunction)`
  - `expressApp.post(urlPath, requestProcessFunction)`
  - `expressApp.put(urlPath, requestProcessFunction)`
  - `expressApp.delete(urlPath, requestProcessFunction)`
  - `expressApp.all(urlPath, requestProcessFunction)`

- Postoje i mnoge druge ređe korišćene metode
- urlPath može da sadrži i parametre, npr. `'/user/:user_id'`

Objekat httpRequest

- `expressApp.get('/user/:user_id', function(httpRequest, httpResponse)`
- Objekat sa velikim brojem svojstava
- Posrednički sloj (kao što su JSON parseri ili upravljači sesijom) mogu da dodaju svojstva kao što su:
  - `request.params` – objekat sadrži URL parametre rute (npr. user_id)
  - `request.query` – objekat sadrži paramtere upita (npr. &id=9 => {id: '9' })
  - `request.body` - objekat sadrži parsirano telo
  - `request.get(field)` – vraća vrednost specifičnog polja HTTP zaglavlja

Objekat httpResponse

- `expressApp.get('/user/:user_id', function(httpRequest, httpResponse)`
- Objekat sa velikim brojem metoda za postavljanje polja HTTP odgovora
  - `response.write(content)` – gradi telo odgovora (*response*) sa sadržajem
  - `response.status(code)` – postavlja HTTP statusni kod za odgovor
  - `response.set(prop, vrednost)` – postavlja svojstvo u zaglavlju odgovora na vrednost
  - `response.end()` – završava zahtev odgovarajući na njega
  - `response.end(poruka)` – završava zahtev odgovarajući na njega sa porukom
  - `response.send(content)` – izvršava *write()* i *end()*
- Metode vraćaju objekat odgovora tako da se nadovezuju (tj. return this;)
- Primer:

```
response.status(code).write(content1).write(content2).end();
```

Posredični sloj (*middleware*)

- Omogućiti drugom softverskom sistemu da se umeša u naše zahteve:

```
expressApp.all(urlPath, function (request, response, next) {
    // Uraditi bilo koju obradu na zahtev (ili postavite odgovor)
    next(); // predati kontrolu sledecem rukovaocu
});
```

- Možemo dodati Middleware u globalni niz middleware aplikacije:

```
expressApp.use(function (request, response, next) {
    // telo funkcije
});
```

- Funkcija dobija tri parametra:
  - `request` – objekat zahteva (http.IncomingMessage) proširen Express-om.
  - `response` – objekat odgovora (http.ServerResponse) proširen Express-om.
  - `next` – povratni poziv koji treba pozvati kada middleware završi svoj podao, kako bi Express nastavio dalje kroz sledeći posredični sloj ili rutu.

Tipične upotrebe i tok izvršavanja

```
expressApp.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

- Tipične upotrebe
  - Prijava u sistem – beleženje svakog zahteva
  - Autentifikacija/autorizacija– proveriti da li je korisnik prijavljen, u suprotnom poslati odgovor o grešci i ne pozvati metodu next(): `res.status(403).send('Forbidden')`.
  - Parsirati telo zahteva kao JSON i priložiti objekat na request.body i pozvati metodu next(). Npr. `express.json()` ili `express.urlencoded()`
  - Upravljanje sesijama i kolačićima, kompresija, enkripcija, i slično.
  - Globalno rukovanje greškama: ako middleware ima 4 argumenta (err, req, res, next), Express ga tretira kao error-handling middleware.
- Tok izvršavanja
  1. Express dobija HTTP zahtev.
  2. Prolazi kroz niz middleware funkcija u redosledu kojim su registrovane sa app.use() ili app.METHOD().
  3. Svaki middleware može:
     1. obraditi zahtev i završiti odgovor (*res.send(...)*, *res.end()*), ili
     2. pozvati *next()*, čime prepušta kontrolu sledećem middleware-u.

Osnovni primer Express.js servera

```
//webServer.js
let express = require('express');
let app = express(); // Kreiranje express objekta koji koristimo za registraciju middleware i ruta

app.use(express.static(__dirname)); // Dodavanje static middleware, tj. Sve fajlove iz direktorijuma
                                    // u kojem se nalazi skripta __dirname, server ce izlagati kao
                                    // staticke resurse (HTML, CSS, JS, slike…)

app.get('/', function (request, response) { // Jednostavan rukovalac zahtevima. Registrujemo rutu GET /
   response.send('Simple web server of files from ' + __dirname); });


app.listen(3000, function () { // Pokretanje Express servera na portu 3000 za zahteve
   console.log('Listening at http://localhost:3000 exporting the directory ' + __dirname);
});
```

- Ovim kodom pokrećemo server, serviramo fajlove iz trenutnog direktorijuma i vraćamo poruku na ruti /
- Ako u direktorijumu postoji index.html, Express će ga servirati kada odeš na putanju http://localhost:3000/, ali ruta app.get('/') će imati prioritet, jer se rute obrađuju redosledom registracije.

Složeniji primer Express.js servera

```
app.get('/user/list', (req, res) => {
  res.json(models.userListModel());
});

app.get('/user/:id', (req, res) => {
  const id = req.params.id;
  const user = models.userModel(id);

  if (!user) {
    console.log(`User with _id: ${id} not found.`);
    return res.status(404).send('Not found');
  }

  res.json(user);
});
```

Ruta /user/:id :

- čita parametar id iz URL-a (req.params.id),
- poziva models.userModel(id) da dobije korisnika,
- ako korisnik ne postoji (null), vraća status 404 i poruku,
- inače vraća korisnika (npr. možemo poslati i status 200, ovako: res.status(200).send(user);).

Jednostavan kod za preuzimanje modela iz JSON datoteke

```
const fs = require('fs').promises;

expressApp.get("/object/:objid", async (req, res) => {
  const dbFile = "DB" + req.params.objid;

  try {
    const contents = await fs.readFile(dbFile, 'utf8');
    const obj = JSON.parse(contents);
    obj.date = new Date();

    res.json(obj); //vracamo JSON objekat klijentu*
  } catch (error) {
    if (error.code === 'ENOENT') {
      res.status(404).send('Object not found');
    } else {
      res.status(500).send(error.message);
    }
  }
});
```

1. Uvozimo fs modul
2. Primamo zahtev tipa GET /object/:objid (npr. /object/7).
3. Na osnovu objid formiramo ime fajla (npr. DB7).
4. Pokušavamo da pročitamo fajl asinhrono, metodom *readFile* i parsiramo ga iz JSON u JS objekat.
5. Ako je uspešno:
   1. dodamo polje date sa trenutnim datumom, i
   2. pošaljemo JSON odgovor klijentu.
6. Ako fajl ne postoji (kod ENOENT) => vraća se 404 Not Found.
7. Ako dođe do druge greške => vraća se 500 Internal Server Error.

* - Express automatski postavlja *Content-Type* na *application/json*

Korišćenje send() / end()

- Kada je potrebno pozvati send() / end()
  - Kada želite da završite odgovor na klijentu.
  - res.send(), res.json(), res.end(), ili slični modeli (res.redirect(), res.sendFile()) završavaju HTTP odgovor.
  - Ako ih ne pozovete, klijent će čekati odgovor, dok ne istekne timeotu (što obično vodi do greške).
- Kada se koristi koji metod:
  - res.send() kada želite kraće odgovore
  - res.json() kada vraćate JSON
  - res.end() kada radite streamovanje fajla, npr. sa res.write()
- Ako middleware prosleđuje kontrolu dalje pozivom next() i ne šalje odgovor sam, tada send/end neće biti pozvani odmah:

```
app.use((req, res, next) => {
    console.log(req.method, req.url);
    next(); // šalje dalje – drugi middleware ili ruta će pozvati send/end
});
```

Rukovanje nizom asinhronih operacija – primer preuzimanja više modela

```
const fs = require('fs').promises; // fs.promises za async/await umesto callback

expressApp.get("/commentsOf/:objid", async (req, res) => {
  const dbFile = "DB" + req.params.objid; //uzimamo objekat iz URL
  const comments = [];

  try {
    // Čitanje glavnog fajla objekta, parsiramo JSON koji sadrži listu imena fajlova komentara
    const contents = await fs.readFile(dbFile, 'utf8');
    const obj = JSON.parse(contents);

    // Čitanje svih komentara paralelno i asinhrono
    const commentPromises = obj.comments.map(async (commentFile) => {
      const commentContents = await fs.readFile("DB" + commentFile, 'utf8');
      return JSON.parse(commentContents);
    });
```

Primer preuzimanja više modela (nastavak)

```
    // Čekamo da se svi komentari učitaju pomoću Promise.all (može i preko async.each)
    const allComments = await Promise.all(commentPromises);
    res.json(allComments); //vraća JSON niz svih komentara sa Content-Type application/json

  } catch (error) { //bilo koja greška (nepostojanje fajla, JSON parsiranje greška) se hvata ovde
    res.status(500).send(error.message);
  }
});
```

Kakav smo stek napravili i koje alate još uključiti

- *Frontend*: Angular (SPA radni okvir za korisnički interfejs).
- *Backend*: Node.js + Express.js (server za API krajnje tačke i statičke fajlove).
- *Paket menadžer*: *npm* (ili moderniji *pnpm*) za upravljanje zavisnostima i skriptama.
- *Build / Bundler*: Angular CLI sa ugrađenim Webpack (kompajlira *TypeScript,* pakuje JavaScript, CSS, slike i druge resurse).
- *Linter / formater*: ESLint + Prettier (održavanje konzistentnog koda i standarda).
- *Nodemon* - automatsko resetovanje Node.js servera prilikom promena koda.

- Za generisanje projekata: Vite, Next.js, Nest CLI, create-react-app
- Zastareli alati: Grunt, Bower, Yeoman – ne preporučuju se kod novih projekata

Šta su Next.js i NestJS?

- **Next.js** - radni okvir punog steka iznad React biblioteke
  - Koristi se za pravljenje brzih UI (*frontend*) sa naprednim opcijama kao što su **serversko renderovanje veb strana** (engl. *Server-Side Rendering*, skr. SSR) i **statičko generisanje sajtova** (engl. *Static Site Generation*, skr. SSG).
  - *Next.js* koristi *Node.js* u pozadini tokom razvoja (za kompajliranje koda) i u produkciji da bi se izvršio serverski JS kod koji generiše HTML, pre nego što se pošalje pretraživaču.
  - Omogućava pisanje API ruta (backend funkcija) unutar istog projekta.
- **NestJS** - progresivni radni okvir za pravljenje isključivo *backend* aplikacija (API-ja, mikroservisa, velikih poslovanih - *enterprise* sistema)
  - Donosi strogu arhitekturu koda na serverskoj strani (*backend*).
  - Duboko inspirisan *Angular* tehnologijom – koristi koncepte moduli, kontroleri, servisi i injektirana zavisnost, i primarno je oslonjen na *TypeScript* programskom jeziku.
  - *NextJS* je apstrakcija iznad *Node.js*; koristi *Node.js* runtime, a ispod haube se oslanja na robusne *HTTP* servere, kao što su *Express.js* (podrazumevano) i *Fastify* (za bolje performanse).

Node.js ekosistem u industriji

- Revolucija punog steka: *Next.js* je redefinisao kako se pravi moderan veb sistem. Njegova ideja je da briše granicu između klijentskog i serverskog dela (*frontend* i *backend* dela).
- Na serverskom delu *Express.js* je bio standard, međutim on je minimalistički i ne nameće nikakvu strukturu, što dovodi do neurednog koda (kod velikih projekata).
- *NestJS* je industrijski standard za velike poslovne *Node.js* aplikacije, jer rešava problem dobre organizacije koda i čiste arhitekture.

Next.js - Rešavanje problema praznog HTML

- Tradicionalne jednostranične aplikacije (SPA) serveru šalju skoro prazan HTML fajl i ogroman JS paket. Klijentov pretraživač onda to preuzme, parsira i izvrši JS da bi korisnik nešto video. To usporava prvo učitavanje i loša je za SEO (veb pretraživači vide praznu veb stranu, pre izvršavanja JS koda).
- *Next.js* uvodi renderovanje na serverskoj strani (SSR).
- Kada korisnik zatraži veb stranu,*Node.js* na serveru unapred izvrši React kod, renderuje kompletan HTML sa svim podacima iz baze i pošalje gotov, bogat HTML kod pretraživaču.
- Korisnik vidi odmah sadržaj, a pretraživači (*Google Search Engine*) ga indeksiraju bez problema.

Next.js - hibridni model

- Next.js ne insistira da inženjer/programmer odabere jedan način renderovanja.
- Na nivou svake pojedinačne veb starane može se birati strategija:
  - SSR: veb strana se generiše na serverskoj strani, pri svakom zahtevu
    npr. Profil korisnika, Glavna tabla/*dashboard* sa živim podacima, itd.
  - CSR: klasičan React model gde se dinamički delovi (poput čet prozora) renderuju direktno kod klijenta.
  - SSG: veb strana se generiše samo jednom, u fazi izgradnje (build),
    npr. blog post, stranica „O nama“, stranica „Kontakt podaci“, itd.
    Server je servira trenutno, kao statički HTML kod.

Full-stack pristup u jednom direktorijumu

- Potreba za dva odvojena projekta (*frontend* i *backend*) se gubi.
  - Unutar *Next.js* proejkta postoji poseban direktorijum (obično *app/api/*) gde svaki JS/TS fajl automatski postaje *Node.JS* API krajnja tačka (engl. *endpoint*).
  - Napravite fajl *route.js*, napišete standardnu asinhronu funkciju koja dovlači podatke iz baze podataka i dobijate funkcionalan backend API unutar iste aplikacije.
- *React Server Components* (RSC)
  - Najaktuelnija paradigma, uvedena u novijim verzijama kroz *App Router*.
  - Komponente su podrazumevano serverske. One se izvršavaju isključivo na *Node.js* serveru.
  - Prednost: Unutar same komponente možemo direktno pisati upite ka bazi podataka (bez *Axios*); klijentu se šalje samo čist HTML, a JS potreban za tu komponentu nikada ne napušta server, što drastično smanjuje veličinu koda koji korisnik mora da preuzme.

NestJS - rešavanje problema „špageti“ koda

- *Express.js* je minimalistički radni okvir:
  ne nameće nikakvu strukturu,
  programer sam bira gde stavlja rute,
  gde poslovnu logiku ili upite ka bazi.
- *Express.js* je dobar za manje projeke,
  ali kod velikih projekata vodi ka „špageti“ kodu.

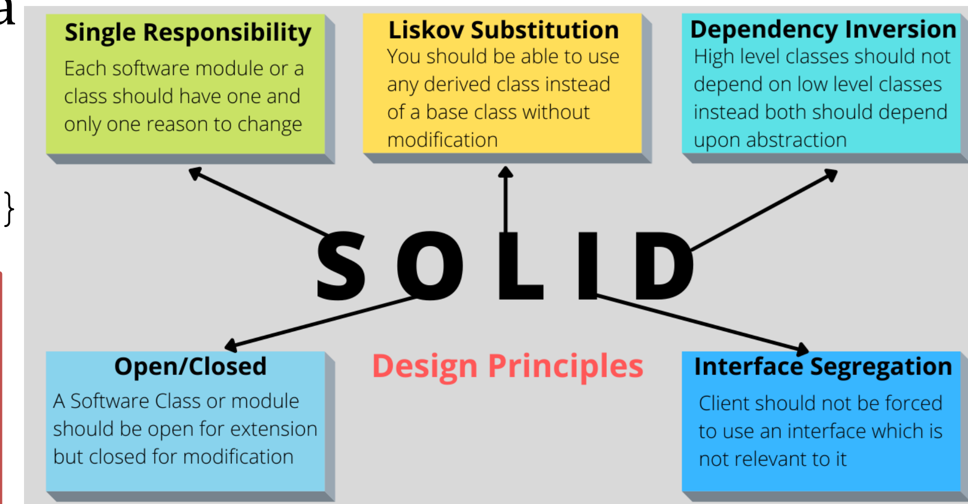
- *NestJS* uvodi strogu i jasnu arhitekturu.
  Arhitektura po uzoru na Angular.
- Primorava nas da organizuje kod u jasne celine:
  - Module, za grupisanje srosnih funkcionalnosti.
  - Kontrolere, samo za upravljanje HTTP zahtevima i rutama.
  - Servise, gde živi poslovna logika i komunikacija sa bazom.

NestJS - prvoklasna podrška za TS i injektirana zavisnost kao standard

- *NestJS* je napravljen pomoću i za *TypeScript* programski jezik.
- *NestJS* koristi napredne koncepte programskog jezika *TS* kao što su dekoratori (npr. @Controller(), @Get(), @Post()), što kod čini izuzetno čitljivim, deklarativnim i elegantnim.
- *NestJS* u svojoj srži ima moćan *IoC* (Inversion of Control) kontejner, koji upravlja zavisnostima među klasama.
- Umesto da ručno instanciraju klase unutar kontrolera npr. `const userService = new UserService( )` *NestJS* to radi automatski kroz konstruktor:

```
constructor(private userService:userService) {}
```

Ovo je savršen praktičan primer kako se u realnoj industriji primenjuju ključni softverski šabloni (*Design Patterns*) i SOLID principi, što olakšava pisanje jediničnih (*Unit*) testova, jer se servisi mogu lako „mokovati“ (*mock*).



NestJS - modularnost i ekosistem

- *NestJS* dolazi sa fenomenalnim CLI alatom (*Command Line Interface*) i ugrađenom podrškom za sve što modernoj serverskoj strani veb aplikacije treba.
- Jednom komandom u terminalu kreiraju se celi moduli.
- *NestJS* ima zvanične, sjajno integrisane module za:
  - Validaciju podataka (pomoću dekoratora kao što su @IsEmail( ), @IsInt( )
  - Autentifikaciju i autorizaciju (Passport, JWT, Guards)
  - Rad sa azama podataka (Mongoose, Prisma, TypeORM)
  - Mikroservise, WebSockets, GraphQL