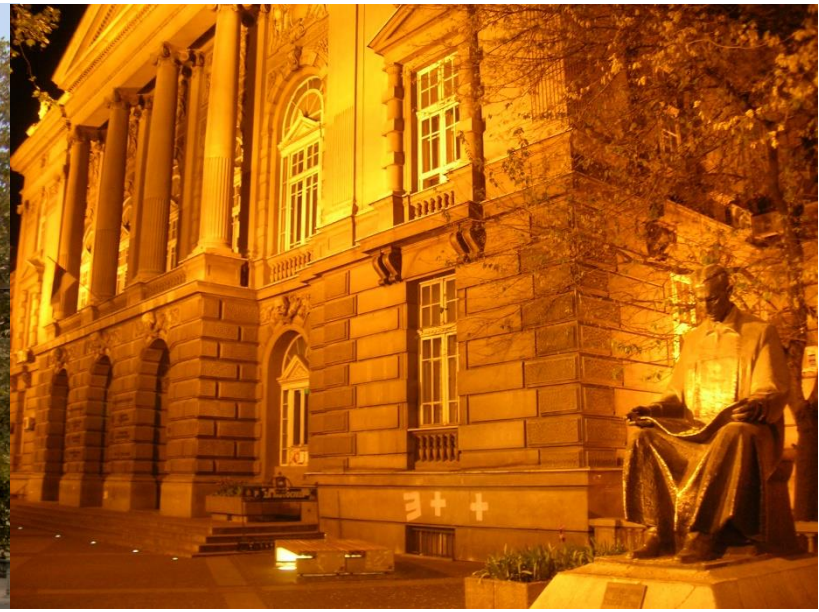
Univerzitet u Beogradu – Elektrotehnički fakultet

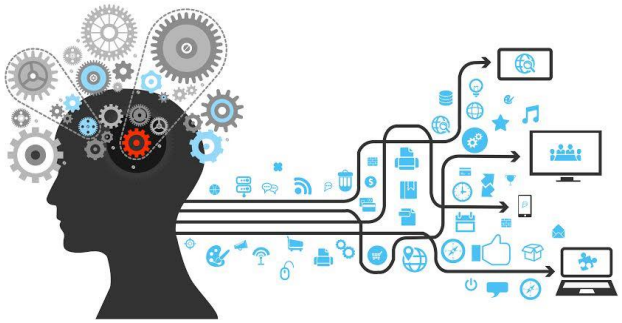
Node JS



Predavač: Dražen Drašković

Beograd, 24. decembar 2018. godine

Sadržaj



- Uvod i instalacija
- Osnovni rad sa konzolom
- Upotreba događaja, osluškivača, tajmera i povratnih poziva
- Upravljanje ulazima i izlazima i rad sa JSON objektima
- Pristup sistemu datoteka
- Implementiranje HTTP servisa, rad sa GET-om i POST-om

Šta je Node.js?

- Node.js, kao modularna platforma, razvijena je u JavaScript-u 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.
- Veoma popularan u mnogim kompanijama, zbog skalabilnosti, brzog razvoja i lakog održavanja.

Čemu služi Node.js?

- Može da se koristi:
  - kao veb server i razvoj backend sloja veb aplikacija,
  - za veb servise (REST, pozadinski veb servisi, kao što su međudomenski i serverski zahtevi),
  - za razvoj igara u realnom vremenu,
  - komunikaciju između više klijenata.

Osnovni moduli kod Node.js

- Console (konzolni pristup)
- Modules (novi moduli)
- Bafer (rad sa TCP)
- HTTP/HTTPS, TLS/SSL
- URL (i analiza putanje)
- Ulaz/izlaz, Path (pristup putanjama datoteke)
- Cluster (upotreba višejezgarnih sistema)
- Net (kreiranje servera i klijenta)
- Crypto (kreiranje prilagođene enkripcije)
- REPL (za komandnu liniju)
- modul tajmera
- dekoder stringa
- modul događaja i toka događaja
- modul znakovnog upita
- modul grešaka
- debager

...

Osnovne komande i folderi

- **node** - komanda za izvršavanje skripta
- **npm** - komanda za upravljanje Node.js paketima
- Svi paketi moraju da budu instalirani u okviru foldera node_modules
- NPM - Node Packaged Module, biblioteka paketa (fajl *package.json* sadrži informativne metapodatke i kontrolne metapodatke)
- Svi registrovani paketi nalaze se u Node Package Registry (NPM registar: https://npmjs.com)
- Node Package manager - program komandne linije preko koga na lak način možemo manipulisati paketima (veza između registra i razvojnog okruženja)

npm komande

Opcija	Opis	Primer
search	Pronalazi pakete modula u repozitorijumu	npm search express
install	Instalira pakete pomoću datoteke package.json iz repozitorijuma ili iz lokala	npm install npm install express
install -g	Instalira paket globalno	npm install express -g
remove	Uklanja paket	npm remove express
pack	Pakuje modul koji je definisan pomoću datoteke package.json u datoteci .tgz	npm pack
view	Prikazuje detalje o modulu	npm view express
publish	Objavljuje modul koji je definisan pomoću datoteke package.json u registru	npm publish
unpublish	Uklanja modul koji je objavljen	npm unpublish mojModul
owner	Omoguaćava da dodate/uklonite/izlistate ime vlasnika paketa u repozitorijumu	npm owner add Drasko mojModul npm owner rm Drasko mojModul npm owner ls mojModul

Direktive u *package.json* (1)

Direktiva	Opis	Primer
name	označava jedinstveni naziv paketa	"name": "moj_modul"
version	označava verziju modula	"version": 0.0.1
preferGlobal	poželjno instalirati modul globalno	"preferGlobal": true
author	ime autora projekta	"author": "drazen@etf.rs"
description	tekstualni opis modula	"description": "super"
contributors	imena saradnika koji su učestvovali u kreiranju modula	"contributors": [{"name": "nbosko"}]
bin	označava binarnu datoteku koju treba instalirati globalno zajedno sa projekotm	"bin": { "excalibur": "./bin/excalibur"}
scripts	određuje parametre pomoću kojih se izvršavaju aplikacije konzole	"scripts" { "start": "node ./bin/excalibur" }
main	određuje glavnu ulaznu tačku aplikacije	"main": "./bin/nesto"
repository	određuje tip repozitorijuma i lokaciju paketa	"repository": { "type": "git", "location": http://rti.etf.rs/srv.git }

Direktive u *package.json* (2)

Direktiva	Opis	Primer
keywords	određuje rezervisane reči koje se prikazuju u npm pretrazi	"keywords": ["etf", "rti", "si"]
dependencies	označava module i verzije od kojih zavisi modul (džoker znakovi su x)	"dependencies": { "express": "latest", "connect": "2.x.x", "cookies": "*" }
engines	označava verziju modula node pomoću kojeg paket funkcioniše	"engines": { "node": ">=6.5" }

Primer js fajla u Node aplikaciji

```
var censoredWords = ["sad", "bad", "mad"];
var customCensoredWords = [];
function censor(inStr) {
  for (idx in censoredWords) {
    inStr = inStr.replace(censoredWords[idx], "****");
  }
  for (idx in customCensoredWords) {
    inStr = inStr.replace(customCensoredWords[idx], "****");
  }
  return inStr;
}
function addCensoredWord(word){
  customCensoredWords.push(word);
}
function getCensoredWords(){
  return censoredWords.concat(customCensoredWords);
}
exports.censor = censor;
exports.addCensoredWord = addCensoredWord;
exports.getCensoredWords = getCensoredWords;
```

Izvoz 3 funkcije

Primer *package.json* fajla

```
{
  "author": "Brad Dayley",
  "name": "censorify",
  "version": "0.1.1",
  "description": "Censors words out of text",
  "main": "censortext",
  "dependencies": {},
  "engines": {
      "node": "*"
  }
}
```

Obavezni atributi!

Upotreba Node.js modula u Node.js aplikaciji

1. Kreirati novi projekat
2. Uraditi instalaciju postojećeg modula:

   npm install ../censorify/censorify-0.1.1.tgz

   npm install censorify (ako je objavljen)
3. Proveriti da postoji node_modules folder i unutar njega potfolder sa instaliranim modulom
4. Kreirati sledeću .js datoteku i unutar nje učitati instalirani modul:

```
var censor = require("censorify");
console.log(censor.getCensoredWords());
console.log(censor.censor("Some very sad, bad and mad text."));
censor.addCensoredWord("gloomy");
console.log(censor.getCensoredWords());
console.log(censor.censor("A very gloomy day."));
```

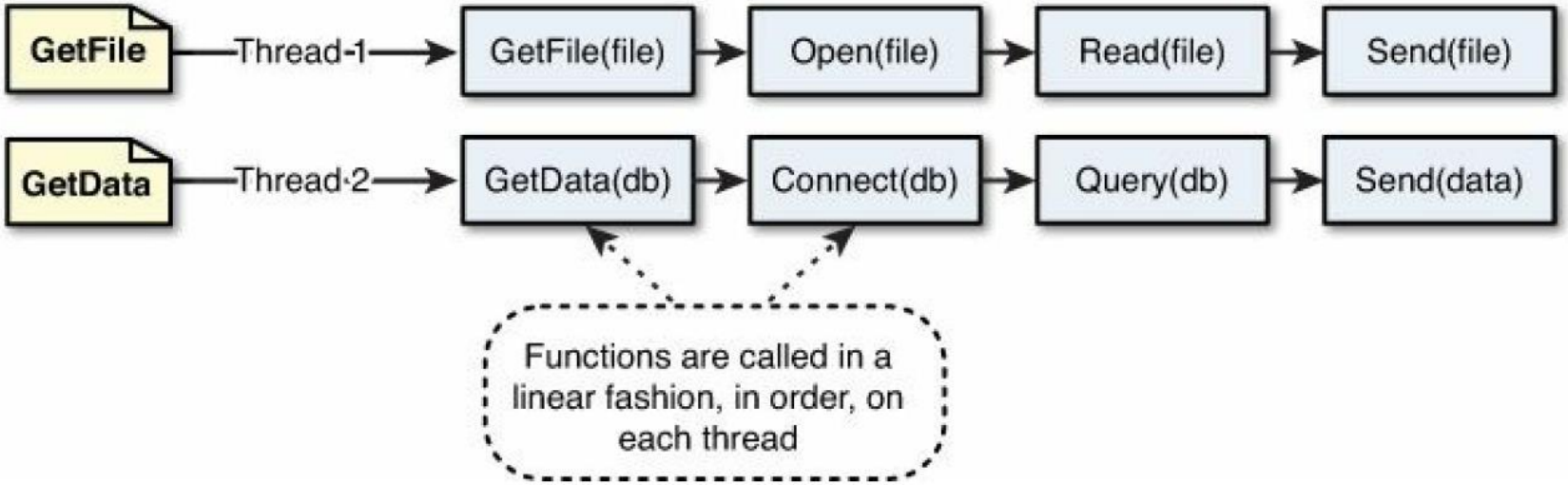
Rad sa konzolom

Funkcija	Opis
log ([data],[....])	Ispisuje podatke u konzolu. Promenljiva podataka može da bude string ili objekat, a mogu biti i dodatni parametri: console.log("Imamo %d studenata", 125);
info ([data], [....])	Isto što i funkcija console.log
error ([data], [....])	Isto što i funkcija console.log uz slanje i procesu stderr
warn ([data], [....])	Isto što i funkcija console.error
dir (obj)	Ispisuje string JavaScript objekta u konzolu, na primer: console.dir({ name: "Drasko", role:"nastavnik" });
time (label)	Određuje trenutnu vremensku oznaku stringu label
timeEnd (label)	Određuje razliku između tekućeg vremena i vremenske oznake dodeljene stringu label
trace (label)	Ispisuje stanje steka trenutne pozicije iz koda
assert (expression, [message])	Ispisuje poruku i stanje steka u konzolu ako izraz daje vrednost false kao rezultat

Node.js model događaja (1)

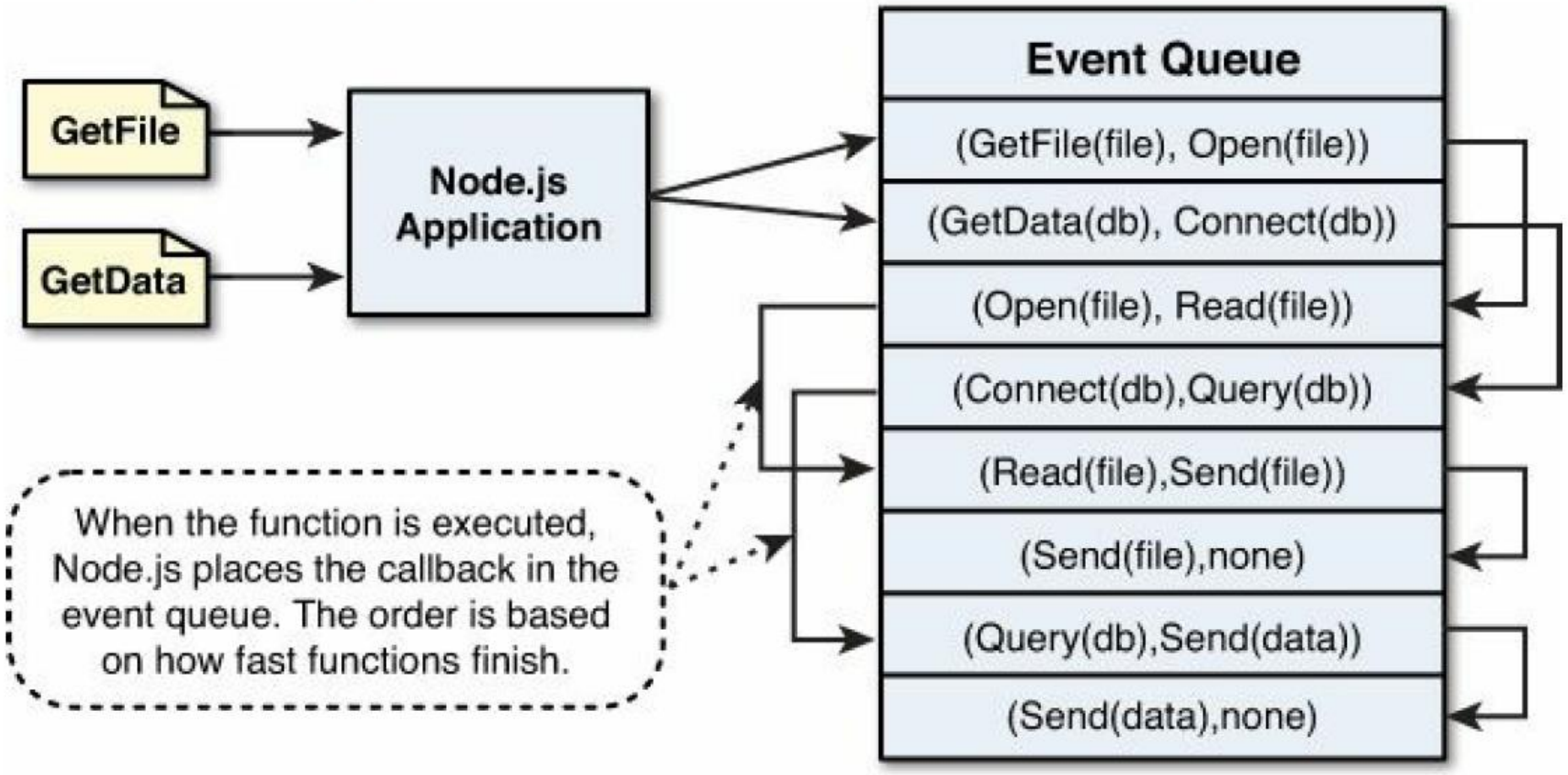
- Node.js aplikacije se pokreću na sinhronom modelu, koji je vođen događajima.
- Node.js implementira skladište niti da izvrši zadatke, ali aplikacija nije zasnovana na konceptu višenitnog programa.
- Standardni pristup: zahtev dolazi do veb servera i dodeljuje se dostupnoj niti, obrađuje se i šalje kao odgovor.

Node.js model događaja (2)



- 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 aplikacija se zatvara.

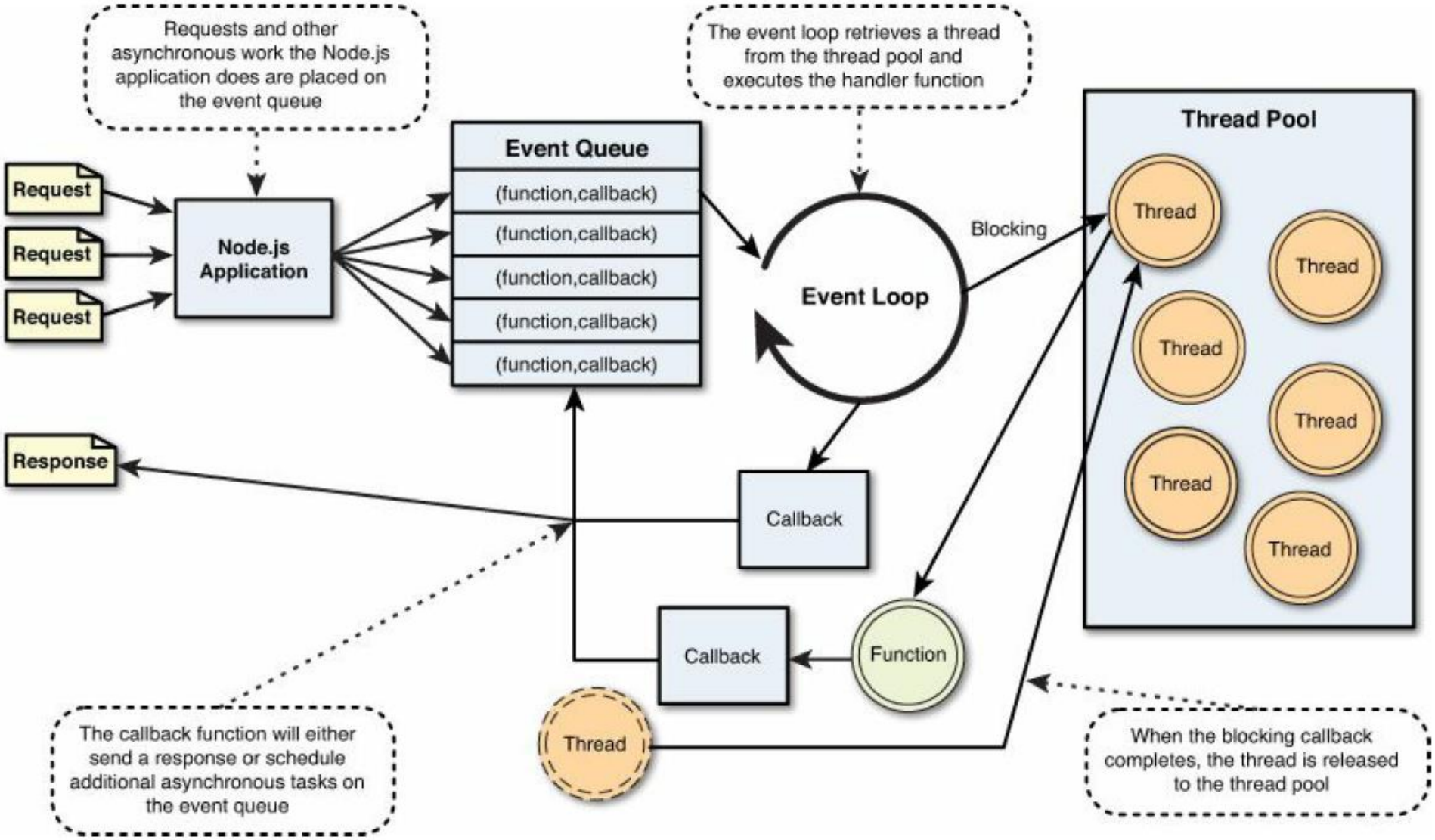
Obrada dva zahteva na sinhronom modelu vođenim događajem - primer



Sinhrone I/O operacije u Node.js

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

Model događaja sa skladištem niti



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 i 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

```
function simpleTimeout(consoleTimer){
  console.timeEnd(consoleTimer);
}
console.time("twoSecond");
setTimeout(simpleTimeout, 2000, "twoSecond");
console.time("oneSecond");
setTimeout(simpleTimeout, 1000, "oneSecond");
console.time("fiveSecond");
setTimeout(simpleTimeout, 5000, "fiveSecond");
console.time("50MilliSecond");
setTimeout(simpleTimeout, 50, "50MilliSecond");
```

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

50MilliSecond: 50.489ms
oneSecond: 1000.688ms
twoSecond: 2000.665ms
fiveSecond: 5000.186ms

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 displayValues(){
  console.log("X=%d; Y=%d; Z=%d", x, y, z);
}
function updateX(){
  x += 1;
}
function updateY(){
  y += 1;
}
function updateZ(){
  z += 1;
  displayValues();
}
setInterval(updateX, 500);
setInterval(updateY, 1000);
setInterval(updateZ, 2000);
```

Izlaz poziva funkcija updateZ(). Funkcija updateX povećava vrednost x 2x brže od y, a funkcija updateY povećava vrednost y 2x brže od z.

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:
  mojInterval = setInterval (mojaFunkcija);
  mojInterval.unref( );
- Pojava velikog broja unref() funkcija može nepovoljno da utiče na performanse Vašeg koda, pa ih koristiti umereno.

Upotreba funkcije nextTick

- 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.
- Za razliku od funkcije setImmediate(), ova funkcija se izvršava pre nego što se aktiviraju događaji ulaza/izlaza.
- Da ne bi došlo do "zamrzavanja" događaja Node.js ograničava broj događaja ove funkcije: process.maxTickDepth = 1000

Upotreba funkcije nextTick - primer

```
var fs = require("fs");
fs.stat("nexttick.js", function(err, stats){
  if(stats) { console.log("nexttick.js Exists"); }
});
setImmediate(function(){
  console.log("Immediate Timer 1 Executed");
});
setImmediate(function(){
  console.log("Immediate Timer 2 Executed");
});
process.nextTick(function(){
  console.log("Next Tick 1 Executed");
});
process.nextTick(function(){
  console.log("Next Tick 2 Executed");
});
```

```
Next Tick 1 Executed
Next Tick 2 Executed
Immediate Timer 1 Executed
nexttick.js Exists
Immediate Timer 2 Executed
```

Implementiranje emitera događaja

- Događaji se emituju pomoću objekta **EventEmitter**.
- Ovaj objekat je smešten u modul događaja.
- Funkcija **emit(eventName, [args])** aktivira događaj eventName i dodaje sve argumente koji su zadati.
- Primer:

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit("nekiDogadjaj");
```

- Događaje možete dodati direktno u JavaScript objekte:

```
function MojObjekat(){
   Events.EventEmmiter.call(this);
}
MojObjekat.prototype.__proto__ = events.EventEmiter.prototype;
```

-
```
var mojObjekat = new MojObjekat();
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)**
    Uklanja funkciju *callback* iz događaja *eventName* objekta *EventEmitter*.

Implementiranje osluškivača događaja i emitera događaja - primer uplata i isplata

```
var events = require('events');
function Account() {
  this.balance = 0;
  events.EventEmitter.call(this);
  this.deposit = function(amount){
    this.balance += amount;
    this.emit('balanceChanged');
  };
  this.withdraw = function(amount){
    this.balance -= amount;
    this.emit('balanceChanged');
  };
}
Account.prototype.__proto__ = events.EventEmitter.prototype;
function displayBalance(){
  console.log("Account balance: $%d", this.balance);
}
function checkOverdraw(){
  if (this.balance < 0){
    console.log("Account overdrawn!!!");
  }
}
function checkGoal(acc, goal){
  if (acc.balance > goal){
    console.log("Goal Achieved!!!");
  }
}
```

```
var account = new Account();
account.on("balanceChanged", displayBalance);
account.on("balanceChanged", checkOverdraw);
account.on("balanceChanged", function(){
  checkGoal(this, 1000);
});
account.deposit(220);
account.deposit(320);
account.deposit(600);
account.withdraw(1200);
```

```
Account balance: $220
Account balance: $540
Account balance: $1140
Goal Achieved!!!
Account balance: $-60
Account overdrawn!!!
```

Implementiranje povratnih poziva

- Tri implementacije povratnih poziva:
  - Prosleđivanje parametara funkciji povratnog poziva;
  - Obrada parametara funkcije povratnog poziva unutar petlje;
  - Ugnežđavanje povratnih poziva.

Prosleđivanje dodatnih parametara povratnim pozivima

- Većini povratnih poziva se automatski prosleđuju parametri, kao što su greška ili bafer rezultata.
- Prosleđivanje dodatnih parametara u povratnom pozivu se radi tako što se implementira parametar u anonimnu funkciju i poziva stvarni povratni poziv sa parametrima iz anonimne funkcije.

Primer prosleđivanja dodatnih parametara

```
var events = require('events');
function CarShow() {
  events.EventEmitter.call(this);
  this.seeCar = function(make){
    this.emit('sawCar', make);
  };
}
CarShow.prototype.__proto__ = events.EventEmitter.prototype;
var show = new CarShow();
function logCar(make){
  console.log("Saw a " + make);
}
function logColorCar(make, color){
  console.log("Saw a %s %s", color, make);
}
show.on("sawCar", logCar);
show.on("sawCar", function(make){
  var colors = ['red', 'blue', 'black'];
  var color = colors[Math.floor(Math.random()*3)];
  logColorCar(make, color);
});
show.seeCar("Ferrari");
show.seeCar("Porsche");
show.seeCar("Bugatti");
show.seeCar("Lamborghini");
show.seeCar("Aston Martin");
```

Dodat parametar, koji događaj sawCar emituje!

Implementacija funkcije *closure* u povratnim pozivima

- Closure - JS termin koji označava da su promenljive sa oblašću važenja funkcije, a ne sa nadređenom oblašću važenja funkcije.
- Kada se izvršavaju asinhroni povratni pozivi, oni se ponavljaju kroz listu i menjaju vrednosti u svakoj iteraciji, pa oblast važenja funkcije može da se promeni.
- Ako povratni poziv treba da pristupi promenljivim u nadređenoj oblasti važenja funkcije, morate da dodate funkciju *closure* da bi vrednosti promenjivih bile dostupne kada se povratni poziv preuzme iz reda događaja. Funkciju closure se jednostavno dodaje, tako što se enkapsulira asinhroni poziv unutar bloka funkcije i prosledi mu promenljive koje su potrebne.

Kreiranje omotačke funkcije, radi dodavanja funkcije closure

```
function logCar(logMsg, callback){
  process.nextTick(function() {
    callback(logMsg);
  });
}
var cars = ["Ferrari", "Porsche", "Bugatti"];
for (var idx in cars){
  var message = "Saw a " + cars[idx];
  logCar(message, function(){
    console.log("Normal Callback: " + message);
  });
}
for (var idx in cars){
  var message = "Saw a " + cars[idx];
  (function(msg){
    logCar(msg, function(){
      console.log("Closure Callback: " + msg);
    });
  })(message);
}
```

asinhroni povratni poziv

osnovni povratni poziv

omotačka funkcija koja se prosleđuje promenljivoj message kao msg parametar, čija vrednost ostaje u povratnom pozivu

Ulančavanje povratnih poziva

- Kada su 2 asinhrone funkcije smeštene u red događaja, ne možemo znati kojim redosledom će se one izvršavati.
- Ako želimo da znamo redosled, možemo implementirati ulančavanje povratnih poziva, tako što će povratni poziv iz asinhrone funkcije pozvati funkciju, sve dok više ne bude poslova u redu događaja. Na taj način asinhrona f-ja će biti samo jednom u redu događaja.

Ulančavanje povratnih poziva - primer

```
function logCar(car, callback){
  console.log("Saw a %s", car);
  if(cars.length){
    process.nextTick(function(){
      callback();
    });
  }
}
function logCars(cars){
  var car = cars.pop();
  logCar(car, function(){
    logCars(cars);
  });
}
var cars = ["Ferrari", "Porsche", "Bugatti",
            "Lamborghini", "Aston Martin"];
logCars(cars);
```

```
Saw a Aston Martin
Saw a Lamborghini
Saw a Bugatti
Saw a Porsche
Saw a Ferrari
```

Node.js

Upravljanje ulazima i izlazima podataka kod Node.js

Upotreba JSON

- JSON = Java Script Object Notation
- Jednostavan metod za pretvaranje JavaScript objekata u string (i obrnuto).
- JSON omogućava: jednostavnu serijalizaciju objekata podataka prilikom njihovog prosleđivanja iz klijenta na server, iz procesa u proces, ili iz toka u tok, ili prilikom skladištenja u bazu podataka.
- Razlike u odnosu na XML standard:
  - JSON je efikasniji i sadrži manje znakova.
  - Serijalizacija/deserijalizacija pomoću JSON-a je mnogo brža nego pomoću XML-a.
  - Programeri lakše čitaju JSON kod, jer je sličan JS sintaksi.

Pretvaranje JSON u JS objekte

- JSON prikazuje JavaScript u formi stringa. Da bi se korektan JSON pretvorio u JS objekat, koristi se sledeća 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);
console.log(companyObj.models);
```

Pretvaranje JS objekata u JSON string

- Koristi se sledeća metoda:
  JSON.stringify(tekst)
- 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);
```

Upotreba modula Buffer za baferisanje podataka

- Baferisani podaci se sastoje od niza okteta u formatu velikog ili malog endiana. Modul *Buffer* omogućava kreiranje, čitanje i pisanje binarnih podataka i manipulisanje njima.
- Modul *Buffer* je globalan, pa se ne mora učitavati sa require().
- Baferisani podaci se skladište u alokacijama "sirove" memorije.
- 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

Kreiranje bafera

- Postoje 3 metode za kreiranje objekata Buffer:
  - new Buffer (sizeInBytes)
  - new Buffer (octetArray)
  - new Buffer (string, [encoding])
- Primeri:
  - var buf256 = new Buffer(256);
  - var bufOkteti = new Buffer([0x6f, 0x63, 0x74, 0x65, 0x74, 0x73]);
  - var bufUtf = new Buffer("Neki utf8 tekst", 'utf8');

Metode za upisivanje podataka u objekte Buffer

Metod	Opis
buffer.write(string, [offset], [length], [encoding])	Upisuje broj bajtova length, iz stringa datog kao prvi argument, počev od indeksa offset.
buffer[offset] = value	Zamenjuje podatke na indeksu offset navedenom vrednošću.
buffer.fill(value, [offset], [end])	Upisuje vrednost svakogbajta u bafer, od indeksa offset, do indeksa end.
writeInt8 (value, offset, [noAssert]) writeInt16LE (value, offset, [noAssert]) writeInt16BE (value, offset, [noAssert])	Različite metode za upisivanje celih brojeva, neoznačenih celih brojeva, itd.

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 *Readable* za čitanje podataka (1)

- Uobičajeni primeri tokova:
  - HTTP odgovori na klijentu
  - HTTP zahtevi na serveru
  - tokovi za čitanje fs
  - tokovi zlib
  - tokovi crypto
  - TCP socketi
  - podređeni procesi stdout i stderr
  - process.stdin

Tokovi *Readable* za čitanje podataka (2)

- Tokovi Readable obezbeđuju metod read([size]) za čitanje podataka, gde size određuje broj bajtova za čitanje iz toka.
- Ovaj metod vraća string, Buffer ili null.
- Tokovi Readable izlažu sledeće događaje:
  - readable - emituje se kada se delovi podataka mogu čitati iz toka;
  - Data - kada su dodati hendleri događaja data, tok se pretvara u režim protoka, a hendler data se poziva neprekidno, sve dok se ne iskoriste svi podaci
  - 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 prilikom prijema podataka

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.

Tokovi *Writable* za upisivanje podataka(1)

- Uobičajeni primeri tokova:
  - HTTP zahtevi na klijentu
  - HTTP odgovori na serveru
  - tokovi za upisivanje podataka fs
  - tokovi zlib
  - tokovi crypto
  - TCP socketi
  - podređeni procesi stdin
  - process.stdout, process.stderr

Tokovi *Writable* za upisivanje podataka(2)

- 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*

- Predstavlja kombinaciju funkcija tokova *Readable* i *Writable*.
- Primer Duplex toka je TCP socket.
- Kada se implementira tok *Duplex*, moraju da se implementiraju metode _read(size) i _write(data, encoding, callback)

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.
- Primeri tokova *Transform* su:
  - tokovi zlib
  - tokovi crypto
- Za razliku od *Duplex*, kod *Transform* ne treba da se implementiraju prototipske metode _read( ) i _write( )

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*
- 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*.
- Primer: readStream.pipe(writeStream, { end: true });
- Pomoću *unpipe(destinationStream)* tokovi mogu da se podele.

Node.js

Pristup sistemu datoteka iz okruženja Node.js

Sinhroni i asinhroni pozivi (1)

- Preduslov rada sa datotekama je učitan modul:
  var fs = require('fs');
- Modul fs omogućava da skoro sve funkcije budu dostupne u asinhronom i u sinhronom obliku.
- Na primer: *write( )* kao funkcija u asinhronom obliku, ima svoj sinhroni oblik *writeSync( )*
- Sinhroni pozivi sistema datoteka će biti blokirani, sve dok se poziv ne završi, a zatim se kontrola vraća na nit.
- Asinhroni pozivi se stavljaju u red događaja da bi bili pokrenuti kasnije. Ovo omogućava pozivima da se uklope u Node.js model događaja, ali to može stvoriti probleme kada izvršavate kod, jer nit koja se poziva nastavlja da se izvršava pre nego što petlja događaja prihvati asinhroni poziv.

Sinhroni i asinhroni pozivi (2)

- Za asinhrone pozive potrebna je funkcija povratnog poziva kao dodatni parametar.
- Izuzeci se automatski obrađuju pomoću asinhronih poziva, a objekat greške se prosleđuje kao prvi parametar ako se pojavi izuzetak.
- Sinhroni pozivi se odmah izvršavaju, a izvršenje se ne vraća na aktivnu nit dok se ne završe pozivi. Asinhroni pozivi se stavljaju u red događaja, a izvršenje se vraća u kod niti koji je pokrenut, ali stvarni poziv neće biti izvršen dok ga ne prihvati petlja događaja.

Otvaranje i zatvaranje datoteka (1)

- Otvaranje datoteke:
  fs.*open (path, flags, [mode], callback)*
  fs.*openSync (path, flags, [mode])*

  callback - funkcija koja prima parametre err i fd
  path - određuje string standardne putanje sistema datoteka
  flag - određuje režim otvaranja datoteke

- Zatvaranje datoteke:
  fs.*close (fd, callback)*
  fs.*closeSync (fd)*

  fd - deskriptor datoteke koji može da se koristi za čitanje datoteke ili upisivanje u datoteku

Otvaranje i zatvaranje datoteka (2)

- Asinhroni režim:

```
fs.open("mojFajl", 'w', function(err, fd){
    if(!err){
        fs.close(fd);
    }
}
```

- Sinhroni režim:

```
var fd = fs.openSync("mojFajl", 'w');
fs.closeSync(fd);
```

Jednostavni upis u datoteku

- Primer upisivanja JSON stringa u datoteku pomoću *writeFile*:

```
var fs = require('fs');
var config = {
  maxFiles: 20,
  maxConnections: 15,
  rootPath: "/webroot"
};
var configTxt = JSON.stringify(config);
var options = {encoding:'utf8', flag:'w'};
fs.writeFile('config.txt', configTxt, options, function(err){
  if (err){
    console.log("Config Write Failed.");
  } else {
    console.log("Config Saved.");
  }
});
```

Sinhrono upisivanje u datoteku

```
var fs = require('fs');
var veggieTray = ['carrots', 'celery', 'olives'];
fd = fs.openSync('veggie.txt', 'w');
while (veggieTray.length){
  veggie = veggieTray.pop() + " ";
  var bytes = fs.writeSync(fd, veggie, null, null);
  console.log("Wrote %s %dbytes", veggie, bytes);
}
fs.closeSync(fd);
```

Asinhrono upisivanje u datoteku

```
var fs = require('fs');
var fruitBowl = ['apple', 'orange', 'banana', 'grapes'];
function writeFruit(fd){
  if (fruitBowl.length){
    var fruit = fruitBowl.pop() + " ";
    fs.write(fd, fruit, null, null, function(err, bytes){
      if (err){
        console.log("File Write Failed.");
      } else {
        console.log("Wrote: %s %dbytes", fruit, bytes);
        writeFruit(fd);
      }
    });
  } else {
    fs.close(fd);
  }
}
fs.open('fruit.txt', 'w', function(err, fd){
  writeFruit(fd);
});
```

Upisivanje tokova u datoteku

```
var fs = require('fs');
var grains = ['wheat', 'rice', 'oats'];
var options = { encoding: 'utf8', flag: 'w' };
var fileWriteStream = fs.createWriteStream("grains.txt",  options);
fileWriteStream.on("close", function(){
  console.log("File Closed.");
});
while (grains.length){
  var data = grains.pop() + " ";
  fileWriteStream.write(data);
  console.log("Wrote: %s", data);
}
fileWriteStream.end();
```

Jednostavno čitanje datoteke

- Čitanje datoteke JSON stringa na objektu:

```
var fs = require('fs');
var options = {encoding:'utf8', flag:'r'};
fs.readFile('config.txt', options, function(err, data){
  if (err){
    console.log("Failed to open Config File.");
  } else {
    console.log("Config Loaded.");
    var config = JSON.parse(data);
    console.log("Max Files: " + config.maxFiles);
    console.log("Max Connections: " + config.maxConnections);
    console.log("Root Path: " + config.rootPath);
  }
});
```

Sinhrono čitanje datoteke

```
var fs = require('fs');
fd = fs.openSync('veggie.txt', 'r');
var veggies = "";
do {
  var buf = new Buffer(5);
  buf.fill();
  var bytes = fs.readSync(fd, buf, null, 5);
  console.log("read %dbytes", bytes);
  veggies += buf.toString();
} while (bytes > 0);
fs.closeSync(fd);
console.log("Veggies: " + veggies);
```

Asinhrono čitanje datoteka

```
var fs = require('fs');
function readFruit(fd, fruits){
  var buf = new Buffer(5);
  buf.fill();
  fs.read(fd, buf, 0, 5, null, function(err, bytes, data){
      if ( bytes > 0) {
        console.log("read %dbytes", bytes);
        fruits += data;
        readFruit(fd, fruits);
      } else {
        fs.close(fd);
        console.log ("Fruits: %s", fruits);
      }
  });
}
fs.open('fruit.txt', 'r', function(err, fd){
  readFruit(fd, "");
});
```

Čitanje tokova podataka iz datoteke

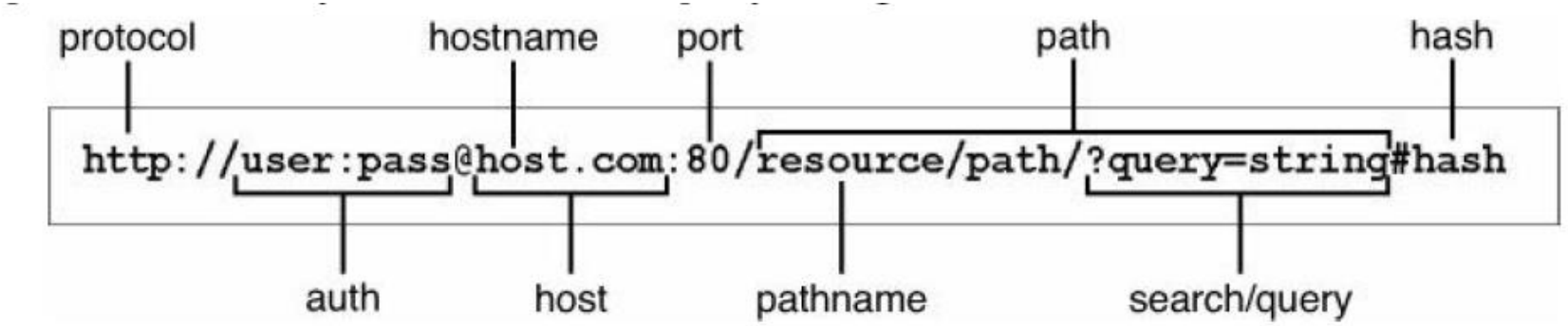
```
var fs = require('fs');
var options = { encoding: 'utf8', flag: 'r' };
var fileReadStream = fs.createReadStream("grains.txt",  options);
fileReadStream.on('data', function(chunk) {
  console.log('Grains: %s', chunk);
  console.log('Read %d bytes of data.', chunk.length);
});
fileReadStream.on("close", function(){
  console.log("File Closed.");
});
```

Node.js

Implementiranje HTTP servisa

Obrada URL adresa

- Modul *express* implementira pun veb servis,
  moduli *http* i *https* obezbeđuju pozadinske veb servise
- Kreiranje objekta URL iz stringa URL-a:
  url.parse(urlString, [parseQueryString], [slashesDenoteHost])
  fleg *parseQueryString* = false
  fleg *slashesDenoteHost* = false



Svojstva objekta URL

Svojstvo	Opis
href	String punog URL-a koji je prethodno raščlanjen
protocol	Protokol *request* koji je ispisan malim slovima
host	Deo *host* puno URL-a, uključujući informacije o delu *port*
auth	Informacije o delu *authentication* URL-a
hostname	Deo *hostname* hosta koji je ispisan malim slovima
port	Deo *port* broja hosta
pathname	Deo *path* URL-a, uključujući kosu crtu na početku
search	Deo *query* string URL-a, uključujući vodeći znak pitanja
path	Puna putanja (*path*) uključujući pathname i search
query	Deo parameter stringa upita ili raščlanjeni objekat, koji sadrži parametar stringa query i vrednosti ako je parametar parseQueryString = true
hash	Deo *hash* URL-a uključujući i znak #

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

Objekat http.ClientRequest - primer

- Osnovna implementacija objekta ClientRequest:

```
var http = require('http');
var options = {
  hostname: 'www.myserver.com',
  path: '/',
  port: '8080',
  method: 'POST'
};
var req = http.request(options, function(response){
  var str = ''
  response.on('data', function (chunk) {
    str += chunk;
  });
  response.on('end', function () {
    console.log(str);
  });
});
req.end();
```

Svojstva koja se zadaju objektu ClientRequest

Svojstvo	Opcije
host	Naziv domena ili IP adresa servera na koji se šalje zahtev (podrazumevana vrednost: *localhost*)
hostname	Isto kao host, uz to podržava i metod: url.parse( )
port	Port udaljenog servera (podrazumevano: 80)
localAddress	Lokalni interfejs za povezivanje na mrežu
socketPath	Socket Unix domena (koristi se host:port ili socketPath)
method	String koji određuje metod HTTP zahteva (GET/POST/CONNECT)
path	String koji određuje potrebnu izvornu pitanju, podrazumevano / Primer: /knjige.html?id=396
headers	Objekat koji sadrži zaglavlje zahteva Primer: { 'content-length':'750', 'content-type': 'text/plain' }
auth	Osnovna autentifikacija u formi user:password
agent	Definiše ponašanje Agent (podraz. Connection:keep-alive)

ClientResponse i njegovi događaji

Svojstvo	Opis
response	Emituje se kada je odgovor na zahtev primljen od servera. Hendler povratnog poziva prima objekat *IncomingMessage* kao jedini parametar.
socket	Emituje se nakon što je socket dodeljen zahtevu.
connect	Emituje se uvek kada server odgovori na zahtev koji je pokrenut pomoću metoda *CONNECT*. Ako ovaj događaj nije obradio klijent, veza će biti zatvorena.
upgrade	Emituje se kada server odgovori na zahtev koji sadrži zahtev Update u zaglavljima.
continue	Emituje se kada server pošalje odgovor *100 Continue HTTP*, tako što klijentu ukazuje da treba da pošalje telo zahteva.

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.

Događaji, svojstva i metodi na objektima IncomingMessage

Metod/događaj /svojstvo	Opis
close	Emituje se kada je osnovni socket zatvoren.
httpVersion	Određuje verziju HTTP-a koji se koristi za izradu zahteva/odg.
headers	Objekat koji sadrži zaglavlja koja su poslata sa zahtevom/odg.
trailers	Objekat koji sadrži prateća zaglavlja koja su poslata u zaht./odg.
method	Određuje metod za zahtev/odg. - GET, POST, CONNECT
url	Ovo je string URL-a koji je poslat na server i može da se prosledi metodu url.parse ( ). Ovaj atribut se koristi na HTTP serveru!
statusCode	Određuje statusni kod sa servera. Korsiti se samo na klijentu!
socket	Ovo je upravljač objekta net.Socket koji se koristi za komunikaciju sa klijentom/serverom
setTimeout (msecs,callback)	Postavlja vreme prekida socketa za vezu u milisekundama, zajedno sa funkcijom callback.

Objekat http.Server

- Objekat Server implementira EventEmitter i emituje događaje koji su navedeni u sledećoj tabeli.
- Prilikom implementiranja HTTP servera morate da upravljate makar jednim događajem ili svim ovim događajima. Na primer, potreban nam je bar hendler događaja koji će obraditi događaj event, koji se aktivira kada je primljen zahtev od klijenta.

Događaji koje mogu da aktiviraju objekti Server

Događaj	Kratak opis
request	Aktivira se uvek kada server primi zahtev klijenta.
connection	Aktivira se kada je uspostavljen novi TCP tok.
close	Aktivira se kada je server zatvoren.
checkContinue	Aktivira se kada je primljen zahtev koji sadrži zaglavlje Expect:100-continue
connect	Emituje se kada je primljen zahtev HTTP CONNECT.
upgrade	Emituje se kada klijent zahteva HTTP nadgradnju.
clientError	Emituje se kada socket klijentske veze prikazuje grešku.